UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

FERRAMENTA DE AUXÍLIO AO PROCESSO DE DESENVOLVIMENTO DE SOFTWARE INTEGRANDO TECNOLOGIAS OTIMIZADORAS

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

ROGER ANDERSON SCHMIDT

BLUMENAU, JUNHO/2001

2001/1-63

FERRAMENTA DE AUXÍLIO AO PROCESSO DE DESENVOLVIMENTO DE SOFTWARE INTEGRANDO TECNOLOGIAS OTIMIZADORAS

ROGER ANDERSON SCHMIDT

ESTE TRABALHO DE ESTÁGIO SUPERVISIONADO FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE ESTÁGIO SUPERVISIONADO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

Prof. Marcel Hugo — Supervisor na FURB

Ricardo de Freitas Becker — Orientador na Empresa

Prof. José Roque Voltolini da Silva — Coordenador na FURB do Estágio Supervisionado

BANCA EXAMINADORA

Prof. Marcel Hugo

Prof. Everaldo Artur Grahl

Prof. José Roque Voltolini da Silva

DEDICATÓRIA

A Anne e ao pequeno Alan, por me presentear com uma nova forma de enxergar a vida.

AGRADECIMENTOS

A Deus por ter me dado saúde e condições de cursar esta faculdade.

Aos meus pais Hercílio Schmidt e Marília Francisca Schmidt, pela formação que se esforçaram para me conceder, cujos frutos continuarei colhendo durante toda a minha vida.

Ao amigo e orientador, Ricardo de Freitas Becker, por ter fornecido condições para a construção deste trabalho e por ter, novamente, depositado confiança em minha pessoa.

Aos professores Marcel Hugo e José Roque Voltolini da Silva, pela atenção e auxílio dispensados na elaboração deste trabalho.

À minha “cúmplice” Anne Jacqueline Pauli, pela compreensão, incentivo e apoio, os quais foram essenciais para atingir este objetivo.

Aos amigos da Mult Sistemas pelo incentivo, em especial ao colega Cleison Vander Ambrosi, pela assistência, prestados na elaboração deste trabalho.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

RESUMO

Este trabalho descreve o desenvolvimento de uma ferramenta CASE (*Computer Aided Software Engineering*), contextualizando-a na área de tecnologias otimizadoras. A ferramenta administra uma coleção de objetos que representam componentes da modelagem de dados, cujas classes foram especificadas a partir da complementação de definições ao repositório de dados da ferramenta CA ERwin. O trabalho permite a geração de código-fonte através da implementação de um interpretador direto para uma linguagem de comandos, utilizando técnicas da área de compiladores. Os *scripts* codificados nesta linguagem geram construções de código-fonte baseadas em recursos de uma biblioteca de classes implementada na linguagem *Object Pascal* pela empresa alvo do estágio, a Mult Sistemas Ltda. O trabalho apresenta ainda um estudo de caso que ilustra o emprego atual da ferramenta na referida empresa, resultando na geração de classes especializadas para a camada de persistência de um sistema de informação.

ABSTRACT

This work describes the development of a CASE tool (Computer Aided Software Engineering), contexting it in the optimizator technologies area. The tool manages an object collection that represents the data modeling components, whose classes were specified using the complementation to definitions of CA Erwin's data repository. The work allows the source-code generation through the implementation of a direct interpreter for a commands language, using techniques of the compiler's area. The scripts codified in this language generate source-code constructions based in class library resources implemented in Object Pascal language by the company where the apprenticeship has done, Mult Sistemas Ltd. The work also presents a study case that illustrates the current use of the tool in the referred company, resulting in the specialized classes generation for the information system persistence layer.

1 INTRODUÇÃO

O ser humano possui uma forma natural de organizar o conhecimento sobre o mundo. Desde a infância, ele aprende a categorizar e a utilizar sensatamente uma enorme massa de conhecimento. Através do estudo das técnicas naturais do funcionamento humano originou-se um conjunto de técnicas aplicadas a construção de sistemas computacionais, denominadas técnicas orientadas a objetos (Martin, 1994, p. 4).

O estabelecimento de padrões industriais para aspectos relacionados às técnicas orientadas a objetos é responsabilidade de uma associação comercial internacional, a OMG (*Object Management Group*). Segundo a OMG, a questão mais importante da revolução tecnológica baseada em objetos é a de reduzir a distância semântica que existe atualmente entre os modelos utilizados nas linguagens de programação e nos sistemas gerenciadores de bancos de dados e os modelos conceituais utilizados pelo homem quando pensa sobre o mundo real (Martin, 1994, p. 391).

O termo revolução industrial do software tem sido utilizado para descrever o movimento em direção a uma era em que os softwares serão compilados de componentes reutilizáveis, os quais deverão tornar-se progressivamente mais complexos em seu interior, porém sua utilização cada vez mais simplificada.

As técnicas orientadas a objetos fornecem benefícios substanciais ao desenvolvimento de software, contudo, quando utilizadas isoladamente não podem prover a magnitude da mudança necessária a chamada revolução industrial do software. Para tanto, estas técnicas tem que ser combinadas de forma sinérgica a outras tecnologias de automação de desenvolvimento de software, denominadas “tecnologias otimizadoras” (Martin, 1994, p. 6).

O presente trabalho contempla um estudo sobre as tecnologias otimizadoras, visando identificar o contexto onde o produto de software originado pelo mesmo encontra-se inserido.

A engenharia de software possibilita o controle do processo de desenvolvimento de software e oferece ao profissional uma base para a construção de software de alta qualidade, produtivamente. A Engenharia de Software Auxiliada por Computador, ou CASE (*Computer Aided Software Engineering*) consiste em uma denominação para as ferramentas de suporte ao desenvolvimento de software. Esta classe de ferramentas combina software, hardware e um

banco de dados para criar um ambiente de engenharia de software, análogo ao projeto auxiliado por computador (CAD) para o hardware (Pressman, 1995, p. 31).

Gane (1990) expõe que não é necessário ao software apresentar capacidade gráfica para ser considerado um produto CASE, pois a característica distintiva nesta classe de software reside no fato de construir um banco de dados do projeto, em um nível mais alto do que comandos de linguagem de programação, ou definição de elementos de dados.

O banco de dados utilizado pelas ferramentas CASE é denominado repositório de dados e armazena todas as informações sobre os sistemas, os projetos e sobre o código, de forma não-redundante. A partir das informações contidas no repositório podem ser criadas novas informações que serão também colocadas neste repositório (Martin, 1994, p. 339).

O repositório de dados CASE é de vital importância na integração das várias ferramentas. A partir de suas definições, são construídos objetos cada vez mais complexos, uma vez que objetos vão sendo construídos a partir de outros e assim por diante (Martin, 1994, p. 14).

O presente trabalho constrói um repositório de dados complementar integrando-se com a ferramenta CASE de modelagem de dados CA Erwin. Segundo Pompilho (1994, p. 183), a modelagem de dados consiste em um método de análise de sistemas que busca especificar a perspectiva dos dados, permitindo organizá-los em estruturas bem definidas, estabelecer regras de dependência entre eles, produzindo um modelo expresso por uma representação, ao mesmo tempo, descritiva e diagramática.

As definições obtidas a partir dos referidos repositórios de dados servem como base para a especificação de uma biblioteca de classes, as quais correspondem aos componentes primitivos da modelagem de dados, ou seja, entidades, atributos, relacionamentos e domínios. A persistência das classes é realizada através da utilização de um banco de dados relacional.

A ferramenta CASE proposta pelo trabalho objetiva ainda a geração de código-fonte potencializado pelo acesso às instâncias das classes do repositório de dados. Para tanto, implementou-se um interpretador de uma linguagem de comandos, através da qual são codificados *scripts* para aplicações diversas.

Aho (1995, p. 2) conceitua o interpretador como uma ferramenta de software que, em lugar de produzir um programa alvo como resultado da tradução, realiza as operações especificadas pelo programa fonte.

Os *scripts* agregam flexibilidade ao processo, pois viabilizam a geração de código para diferentes linguagens. Na prática, esta geração resulta da utilização de comandos disponíveis na linguagem especificada que alimentam um arquivo do tipo texto. Dessa forma, através de *scripts* específicos podem ser obtidas outras aplicações, como a geração de relatórios de documentação, ou arquivos na linguagem HTML (*Hyper Text Markup Language*). O acesso aos objetos instanciados a partir do repositório da ferramenta proposta, é realizado através de outro conjunto de comandos disponível na linguagem.

No intuito de fundamentar o desenvolvimento do interpretador e a concepção da linguagem, realizou-se um estudo de técnicas da área de compiladores. Para especificação da sintaxe da linguagem, utilizou-se a notação BNF (*Backus-Naur Form*) e para a semântica foi empregada a notação da gramática de atributos, ambas descritas no cap. 2.3.

O presente trabalho caracteriza-se por um estágio supervisionado, cuja construção desenvolveu-se no ambiente interno de uma empresa produtora de software, a Mult Sistemas Ltda.

1.1 A EMPRESA MULT SISTEMAS

A Mult Sistemas é uma empresa do ramo de desenvolvimento de sistemas de informação, voltada principalmente à área de gestão empresarial.

Sediada em Santa Catarina, na cidade de Blumenau, a empresa conta atualmente com 42 colaboradores que atendem a mais de 50 revendedores distribuídos em 20 estados e uma carteira de aproximadamente 2000 empresas clientes em todo o Brasil. Fundada em 1990, a empresa encontra-se consolidada no mercado nacional e é uma das 5 maiores produtoras de sistemas de gestão empresarial de Blumenau.

A metodologia de desenvolvimento de software adotada pela empresa é a análise e projeto orientados a objeto. Atualmente, a linguagem de programação utilizada para

codificação de seus produtos é o *Object Pascal*, empregada na implementação da biblioteca de classes que amparou a construção do presente trabalho.

1.2 OBJETIVOS DO TRABALHO

O presente trabalho tem como objetivo principal desenvolver uma ferramenta CASE que gere código-fonte a partir das definições de um repositório de dados, cuja estrutura é resultante da complementação de definições ao repositório da ferramenta de modelagem de dados CA ERwin.

Os objetivos específicos do trabalho são:

a) ilustrar a combinação de tecnologias otimizadoras, visando avaliar os benefícios apresentados por Martin (1994);
b) acessar em modo completo as definições contidas no repositório de dados da ferramenta CASE CA Erwin;
c) processar a engenharia reversa de determinados fragmentos de código-fonte escrito na linguagem Pascal, referentes a uma geração de sistemas legados;
d) complementar o repositório de dados com definições que enriqueçam os arquivos de código-fonte gerados pela ferramenta;
e) implementar um interpretador de *scripts* de uma linguagem de comandos, que disponibilize funções para manipulação dos objetos instanciados a partir das definições do repositório de dados, tais como módulos, entidades, atributos, índices, entre outros;
f) codificar *scripts* para geração de construções de código-fonte baseadas em recursos de uma biblioteca de classes implementada na linguagem *Object Pascal*.

1.3 ESTRUTURA DO TRABALHO

O primeiro capítulo apresenta uma introdução ao trabalho, iniciando alguns conceitos empregados em sua elaboração. São apresentados ainda, os objetivos e a organização do texto.

O segundo capítulo compila a fundamentação teórica aplicada no desenvolvimento do trabalho. Apresenta informações sobre tecnologias otimizadoras no desenvolvimento de software, além de um breve descritivo da ferramenta CASE CA ERwin. O capítulo reúne

ainda os conceitos e técnicas relacionadas à área de compiladores empregadas na especificação de uma gramática para uma linguagem de comandos, bem como na construção de seu interpretador.

O terceiro capítulo apresenta considerações sobre o desenvolvimento da ferramenta, destacando as atividades realizadas durante as fases do ciclo de vida adotado. Ao longo do capítulo são descritos os pontos de aplicação e a forma de utilização da fundamentação teórica descrita no capítulo anterior. É apresentado ainda um estudo de caso ilustrando uma aplicação prática da ferramenta de software em questão.

O quarto capítulo apresenta a conclusão do trabalho, bem como sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão apresentadas considerações sobre tecnologias otimizadoras no desenvolvimento de software, bem como uma breve descrição sobre a ferramenta CASE CA ERwin. Serão abordados ainda conceitos e técnicas relacionadas à área de compiladores, direcionados especialmente à fundamentação necessária para construção de interpretadores diretos.

2.1 TECNOLOGIAS OTIMIZADORAS

O termo “revolução industrial do software” tem sido utilizado para descrever o movimento em direção a uma era em que os softwares serão compilados de componentes reutilizáveis, os quais serão construídos a partir de outros componentes e publicados em grandes bibliotecas. Existe a necessidade de uma migração da era de pacotes monolíticos de software, onde um único fornecedor desenvolve o pacote inteiro, para uma era onde o software é montado de componentes e pacotes de vários fornecedores. Os componentes deverão tornar-se progressivamente mais complexos em seu interior, contudo a interação com eles será cada vez mais simples. Este prognóstico foi extraído de Martin (1994, p. 6).

Martin (1994, p. 6) expõe ainda que a aplicação do conceito de “componentização”, torna-se viável quando amparada pelas técnicas baseadas em objeto. Estas técnicas permitem que os softwares sejam construídos de objetos que tenham seu comportamento especificado. Os próprios objetos podem ser construídos de outros objetos, e assim sucessivamente. Os sistemas são construídos de componentes testados, com solicitações padronizadas invocando as operações do componente, o qual também consiste em um objeto.

As técnicas baseadas em objetos sozinhas não podem prover a magnitude da mudança necessária para a revolução industrial do software. Contudo, quando combinadas de forma sinérgica com outras tecnologias, tornam-se extremamente poderosas. Ao conjunto destas tecnologias convenciona-se chamar de “tecnologias otimizadoras” de desenvolvimento de software (Martin, 1994, p. 7).

O Quadro 2-1 relaciona um conjunto de tecnologias otimizadoras de software, eleitas por Martin (1994), as quais serão descritas no decorrer deste capítulo.

Quadro 2-1 – EXEMPLOS DE TECNOLOGIAS OTIMIZADORAS

- Ferramentas CASE
- Programação Visual
- Geradores de Código
- Repositório e Coordenador de Repositório
- Metodologias Baseadas em Repositório
- Engenharia da Informação
- Bancos de Dados Orientados a Objetos
- Linguagens Não-Procedimentais
- Métodos Formais Baseados na Matemática
- Motores de Inferência
- Tecnologia Cliente-Servidor
- Bibliotecas de Classes
- Análise e Projeto Orientados a Objetos

Fonte: (Martin, 1994, p. 7)

2.1.2 INTEGRANDO TECNOLOGIAS OTIMIZADORAS

Martin (1994, p. 7) afirma que uma revolução industrial em software não resulta de nenhuma das tecnologias otimizadoras isoladamente, mas de todas integradas em uma estrutura baseada em objetos. O software, como as máquinas, será construído de componentes reutilizáveis. Os projetistas não precisarão conhecer as engrenagens internas dos componentes.

Martin (1994, p. 7) ilustra sua afirmação anterior através do exemplo de que um projetista de um gravador de videocassete não o projetaria transistor a transistor, mas sim a partir de componentes de alto nível. Um *chip* pode conter cem mil transistores, e, analogamente, devem existir componentes de software de alto nível, contendo milhares de linhas de código, projetadas a zero erro.

2.1.3 FERRAMENTAS CASE

Particularmente importante para o futuro do desenvolvimento de software é o crescimento da indústria de CASE (*Computer Aided Software Engineering*, ou Engenharia de Software Auxiliada por Computador). As ferramentas CASE ocupam, na indústria de software, o lugar equivalente ao CAD (*Computer Aided Design*) nas atividades de *layout* de circuitos e projetos de *chips*.

À proporção que as ferramentas CASE se tornaram mais potentes, grande parte do projeto passou a ser sintetizada, em vez de construída artesanalmente. As ferramentas de projeto foram associadas a um repositório contendo informações utilizadas na construção do projeto. Este repositório, em seguida, passou a armazenar as informações envolvidas no planejamento, na análise, no projeto e na geração de código. Com a evolução das ferramentas, o repositório passou a conter também, gabaritos e máscaras, e elementos de construção reutilizáveis para serem customizados para o sistema em questão, além de um modelo de aplicativo que pudesse ser alterado. Particularmente importante, na atualidade, é que o repositório contenha classes baseadas em objetos, reutilizáveis, projetadas para serem incorporadas aos aplicativos. As ferramentas começaram realmente a tornar-se poderosas quando esses recursos foram integrados. As ferramentas CASE para planejamento, para modelagem de dados e de processos, e para criar projetos foram integradas com geradores de códigos. As linguagens não-procedimentais, inclusive SQL, e geradores de relatórios foram também integrados ao ambiente CASE (Martin, 1994, p. 329).

Martin (1994, p. 8) conceitua ferramentas CASE como softwares que empregam as representações gráficas na tela para ajudar a planejar, analisar, projetar e gerar software.

Contrariando o conceito anterior, Gane (1990) expõe que não é necessário ter capacidade gráfica para ser um produto CASE, pois a característica distintiva nesta classe de software é o fato de construir um banco de dados do projeto, a um nível mais alto do que comandos de linguagem de programação, ou definição de elementos de dados.

Gane (1990) afirma ainda que os produtos CASE são um subgrupo especial de auxiliares de desenvolvimento e manutenção. As ferramentas CASE extraem a lógica em nível de especificação de código, dessa forma, são definidas por construir um banco de dados do projeto.

Segundo Fisher (1990, p. 5), o objetivo principal da tecnologia CASE é separar o projeto do programa aplicativo, da implementação do código. O desenvolvimento tradicional de software dá ênfase à implementação, codificação e depuração. A engenharia de software computadorizada concentra-se na análise dos requisitos e na especificação do projeto.

Fisher (1990, p. 20) afirma que para muitas organizações de desenvolvimento de software as vantagens qualitativas das ferramentas CASE tem um peso maior que as quantitativas. O tempo gasto no desenvolvimento está quase sempre menor com o auxílio das ferramentas CASE, mas talvez seu maior benefício represente a forma de garantia de que a tarefa está sendo executada devidamente, de acordo com as especificações. A tecnologia CASE tem como principais vantagens:

a) especificações completas dos requisitos;
b) especificações minuciosas do projeto;
c) especificações atuais do projeto;
d) redução do tempo de desenvolvimento;
e) código altamente flexível e de fácil manutenção.

2.1.3.1 CATEGORIAS DE FERRAMENTAS CASE

Segundo Martin (1994, p. 330), as ferramentas CASE podem ser categorizadas como CASE Integrado (ou I-CASE), CASE Fragmentado e CASE-IE (*Information Engineering*, ou Engenharia da Informação).

O CASE Integrado refere-se a um pacote de ferramentas CASE que suporta todos os estágios do ciclo de vida de desenvolvimento, inclusive o da geração de código, com um único repositório, logicamente consistente. No CASE Integrado, as ferramentas para todos os estágios do desenvolvimento se unem e alimentam um gerador de código.

Ao contrário do CASE Integrado, o CASE Fragmentado suporta apenas uma porção do ciclo de vida do software. Esta categoria de CASE inclui as ferramentas de *front-end*, que agrupam a parte de análise, e as ferramentas de *back-end*, que referem-se à geração de código. Algumas suportam engenharia da informação, apresentada no cap. 2.1.8, e auxiliam na construção do modelo da empresa e na análise das áreas de negócio.

Existem ainda os CASE-IE (*Information Engineering*) que suportam totalmente a engenharia da informação.

Tal como as linguagens de programação, as ferramentas CASE podem ser classificadas em não-baseadas em objetos, baseadas em objetos puras e baseadas em objetos híbridas.

A indústria de ferramentas CASE se desenvolveu no final dos anos 80, construindo somente ferramentas não-baseadas em objetos.

A medida em que as técnicas baseadas em objetos tornaram-se melhor compreendidas, as ferramentas CASE acrescentaram conceitos baseados em objetos aos seus conjuntos, embora os diagramas CASE básicos permaneceram os mesmos, os quais apoiavam técnicas estruturadas tradicionais. As ferramentas possuíam uma aparência baseada em objetos, mas sua essência básica não refletia a transferência do paradigma das técnicas estruturadas para as técnicas baseadas em objetos.

As ferramentas CASE-OO puras suportam as técnicas baseadas em objetos, além de gerar código. Algumas foram desenvolvidas para se trabalhar em uma estação de trabalho isolada, para apoiar um profissional de desenvolvimento. Porém, existe uma carência de CASE OO integradas, as quais permitem que as equipes compartilhem um repositório, de maneira que o desenvolvimento possa progredir de modelos inteligentes da empresa, para geração eficiente de sistemas.

Qualquer uma das categorias descritas pode ser não-baseada em objetos, tradicional com aspectos de baseada em objetos, puramente baseada em objeto, ou pode, ainda, suportar integralmente tanto as técnicas tradicionais quanto às técnicas baseadas em objetos (Martin, 1994, p. 328).

2.1.4 PROGRAMAÇÃO VISUAL

Segundo Martin (1994, p. 7), a programação visual expressa o projeto de programas em gráficos, cor, e possivelmente, som. Os objetos são representados mais concretamente, para que possam ser visualizados como máquinas físicas que transitam de estado a estado.

A programação visual permite que o técnico de desenvolvimento entre, compreenda, imagine, rode, depure e manipule os programas, usando, basicamente, notações visuais.

2.1.5 GERADORES DE CÓDIGO

Fisher (1990, p. 27) afirma que a geração automática de código define a capacidade de gerar automaticamente um software funcional ou compilável diretamente de uma especificação de projeto.

A geração automática do código, integral ou parcial, fornece as seguintes vantagens principais: redução do tempo de desenvolvimento, pois minimiza a necessidade de codificação manual, e aumento da confiabilidade do código gerado, o qual foi produzido por uma ferramenta depurada e testada (Fisher, 1990, p. 10).

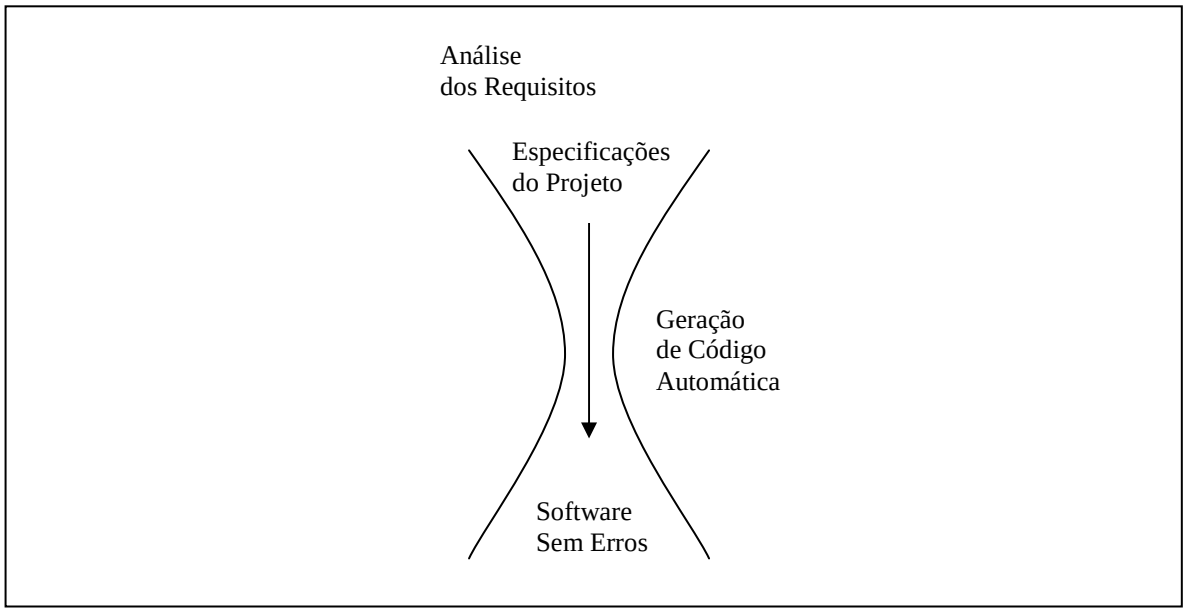
Martin (1994, p. 327) expõe o seguinte prognóstico: “Os geradores de código de qualidade deverão tornar-se o principal meio de se criar aplicativos baseados em objetos, com um repositório CASE e classes reaproveitáveis. Com as ferramentas CASE-OO, a ênfase maior da construção de sistemas será sobre a modelagem e sobre o projeto, e não sobre a programação. Dessa forma, as metodologias de análise baseadas em objetos não deveriam estar atreladas a linguagens baseadas em objetos e seus recursos. Uma vez que a tecnologia de desenvolvimento tem mudado tão rapidamente, deve-se analisar os sistemas, independentemente dos recursos de programação utilizados“.

Compilando-se as considerações relacionadas a geração automática de código de Martin (1994) e Fisher (1990), obtém-se a afirmação de que esta é a meta suprema da maioria dos produtores de ferramentas CASE. Infelizmente, a geração de código integral de uma aplicação ainda não é fato comum. Portanto, existe um engarrafamento fundamental da tecnologia CASE, ilustrado na Figura 2-1, sendo que a programação automática é um problema difícil e ainda considerado tópico de pesquisa.

As ferramentas I-CASE mais potentes permitem que o projeto seja sintetizado a partir de construções de alto nível contidas no repositório. O projetista monta o projeto e cria sua lógica detalhada. Parte do projeto pode ser gerada a partir de declarações comportamentais e estruturais de alto nível.

O projeto alimenta um gerador de código que cria integralmente o código a zero erro de programação. O processo de testes é mais dirigido à detecção de erros de projeto do que de erros de detalhes de codificação (Martin, 1994, p. 333).

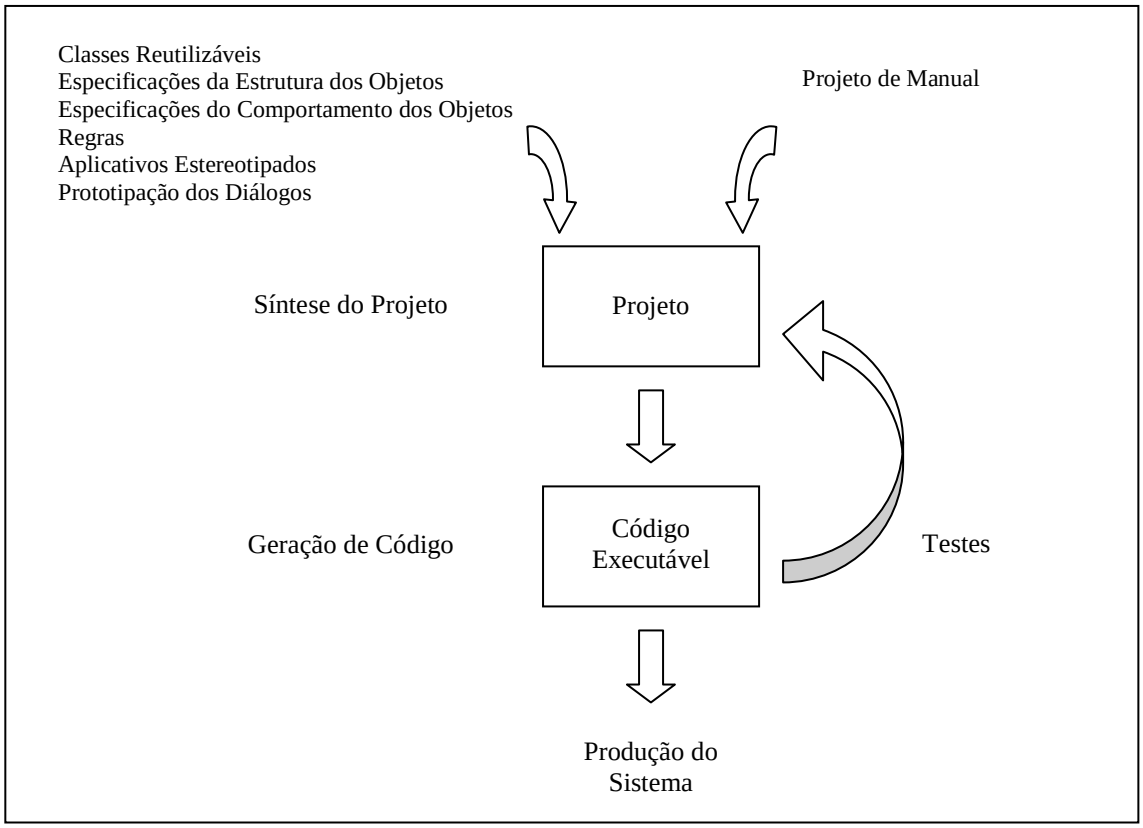
Figura 2-1 – O ENGARRAFAMENTO CASE



Fonte: (Fisher, 1990, p. 27)

A combinação de síntese do projeto e de geração de código, ilustrada na Figura 2-2, possibilita a construção rápida de aplicativos de alta qualidade.

Figura 2-2 – SÍNTESE DO PROJETO E GERAÇÃO DE CÓDIGO



Fonte: (Martin, 1994, p. 334)

2.1.6 REPOSITÓRIO

O repositório consiste em um banco de dados que armazena todas as informações sobre os sistemas, os projetos e sobre o código, de forma não-redundante. A partir das informações contidas no repositório podem ser criadas novas informações que serão também colocadas neste repositório (Martin, 1994, p. 339).

Um repositório é mais amplo do que um dicionário de dados. Um dicionário contém nomes e descrições de itens de dados, processos, variáveis e assim por diante. Um repositório contém uma representação completa codificada utilizada no planejamento, na análise e na geração de código. O repositório armazena o significado representado pelos diagramas CASE e obriga que futuras alimentações sejam consistentes com essa representação. O repositório, portanto, “entende” o projeto, enquanto que um simples dicionário não é capaz disto (Martin, 1994, p. 340).

O software “coordenador do repositório” aplica métodos aos dados do repositório para assegurar que os dados e as representações CASE dos dados sejam consistentes e íntegras.

O repositório proporciona uma interface sem costuras entre as ferramentas do conjunto I-CASE, pois estas empregam um repositório único, eliminando redundâncias (Martin, 1994, p. 10).

2.1.7 METODOLOGIAS BASEADAS EM REPOSITÓRIO

Um repositório expansível armazena modelos, especificações, projetos e componentes reutilizáveis de que se compõe o software. As metodologias de desenvolvimento de sistemas dizem respeito à construção de uma coleção extensiva de conhecimento em um repositório, onde a geração de código é provocada e dirigida. Uma coleção crescente de componentes reutilizáveis está armazenada no repositório.

Uma metodologia baseada em repositório é desenvolvida visando obter o máximo de vantagens do conjunto I-CASE e maximizar o reaproveitamento.

Particularmente importantes são as metodologias baseadas em repositório para desenvolvimento em alta velocidade (Martin, 1994, p.10).

2.1.8 ENGENHARIA DA INFORMAÇÃO

Martin (1994, p. 280) conceitua que a engenharia da informação (IE) consiste na aplicação de um conjunto de técnicas interligadas para o planejamento, a análise, o projeto, a construção e manutenção dos sistemas de informação de toda uma empresa, ou de uma área de porte na empresa. A engenharia de software emprega técnicas estruturadas a um projeto, enquanto que a engenharia da informação aplica técnicas estruturadas ou baseadas em objetos à empresa como um todo ou a um grande segmento da empresa.

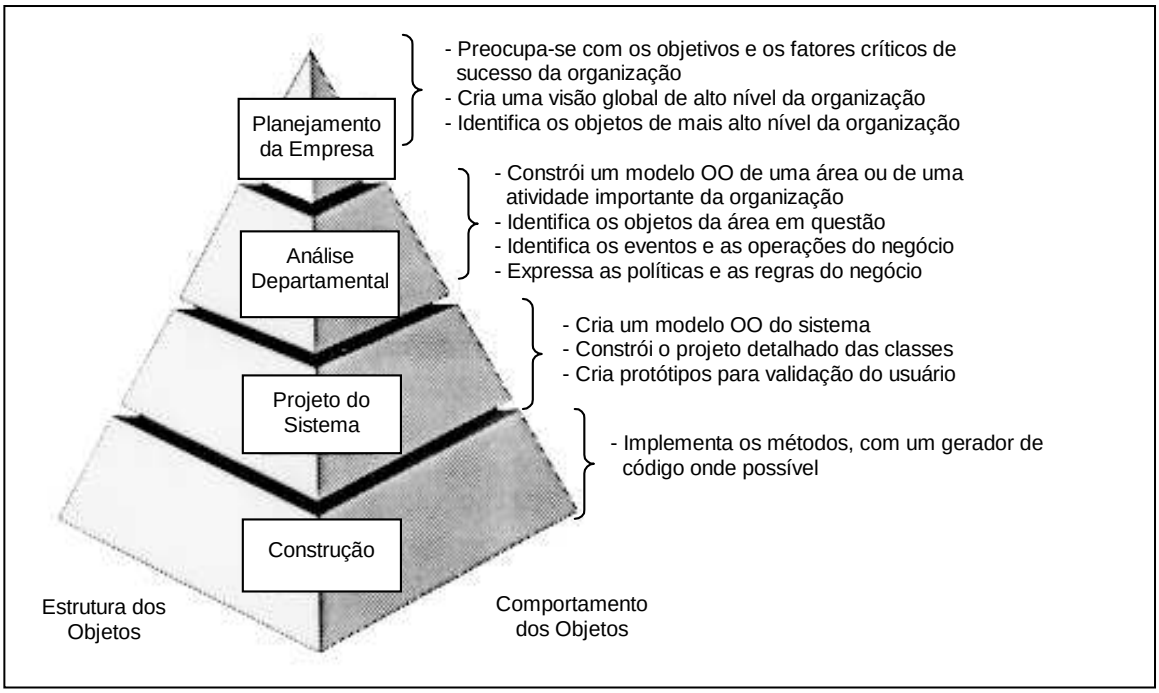
Assim como diferentes organizações variam em suas práticas de engenharia de software, também existem variantes sobre o tema engenharia da informação, que não deve ser encarada como uma metodologia, mas sim como uma classe genérica de metodologias. A abordagem baseada em objetos é a variante mais eficaz da engenharia da informação, pois cada sistema é construído considerando a integração com os demais sistemas, o que elimina problemas comuns apresentados pelo processamento de dados tradicional, como incompatibilidades e redundâncias (Martin, 1994, p. 280).

Com a engenharia da informação são criados planos e modelos de alto nível e os sistemas desenvolvidos separadamente se integram através desses planos e modelos. O planejamento estratégico é aplicado à empresa inteira. Uma análise mais detalhada é feita em áreas separadas da empresa, ou em correntes de atividades relacionadas que permeiam a empresa. As técnicas de projeto e construção são aplicadas a cada sistema.

A pirâmide usada para representar a engenharia da informação indica que ela avança com consistência, através de seus estágios progressivamente mais detalhados: do planejamento da empresa toda, para a análise das áreas funcionais, para o projeto do sistema, e depois para a construção. A Figura 2-3 fornece a versão dessa pirâmide para a engenharia da informação baseada em objetos.

Segundo Martin (1994, p. 280), um dos objetivos da engenharia da informação é o de identificar os tipos de objetos comuns a toda a empresa e, conseqüentemente, minimizar o esforço redundante no desenvolvimento e na manutenção dos sistemas. Esta tarefa torna-se mais facilitada, quando adotadas as estruturas formais de classes da análise e do projeto baseados em objetos.

Figura 2-3 – PIRÂMIDE DA ENGENHARIA DA INFORMAÇÃO BASEADA EM OBJETOS



Fonte: (Martin, 1994, p. 283)

A engenharia da informação aplica o desenvolvimento baseado em repositório a toda uma empresa para integrar o planejamento, o projeto e a construção de sistemas que precisam operar interativamente através da organização. Ela cria um modelo da empresa e tenta reprojetar os sistemas de informação para serem tão eficientes quanto possível (Martin, 1994, p. 10).

2.1.9 BANCOS DE DADOS ORIENTADOS A OBJETOS

Um banco de dados orientado a objetos consiste em um banco de dados que suporta o paradigma da orientação a objetos. Dessa forma, armazena dados e operações, ao invés de apenas dados. É projetado para ser fisicamente eficiente no armazenamento de objetos complexos. Pode, dentre outras tarefas, impedir o acesso aos dados (ou atributos), salvo se for feito através de operações (ou métodos) armazenadas. Empregando o vocabulário orientado a objetos, os atributos permanecem encapsulados (Martin, 1994, p. 350).

Martin (1994, p. 357) classifica os bancos de dados em passivos e ativos. O banco de dados clássico, onde enquadra-se o relacional, é passivo, o qual meramente armazena os

dados de uma forma independente dos processos. Por outro lado, um banco de dados ativo toma certas ações, automaticamente, quando uma tentativa é feita para leitura ou atualização dos dados. Esta característica é referenciada como “inteligência” do banco de dados.

Martin (1994, p. 370) afirma que os bancos de dados relacionais têm como objetivo a independência de dados. Os dados são normalizados, de forma que possam ser utilizados por muitos processos ainda não previstos. Os bancos de dados baseados em objetos têm por objetivo suportar classes com encapsulamento, onde os dados são utilizados com métodos específicos. Eles têm independência de classes, ao invés de independência de dados.

2.1.10 LINGUAGENS NÃO-PROCEDIMENTAIS

As linguagens tradicionais de programação, tais como COBOL, PL/I, FORTRAN, PASCAL e C são procedimentais, ou seja, os programadores fornecem instruções precisas, detalhadas, de “como” cada ação deve ser executada. Com linguagens não-procedimentais o usuário determina “o que” deve ser feito, abstraindo o procedimento detalhado da execução.

A maioria das linguagens de manipulação de dados (*query languages*), dos geradores de relatório, dos pacotes gráficos e dos geradores de aplicativo consistem em linguagens não-procedimentais. As linguagens de quarta geração (4GLs) combinam características procedimentais e não-procedimentais (Pressman, 1995, p. 703) (Martin, 1994, p. 259).

Segundo Pressman (1995, p. 703), uma linguagem 4GL possibilita que o usuário especifique condições e as correspondentes ações (o componente procedimental), encorajando, ao mesmo tempo, o usuário a indicar o resultado desejado (o componente não-procedimental) e então aplicar seu conhecimento específico do domínio para preencher os detalhes procedimentais.

Martin (1994, p. 259) expõe que a possibilidade de uma linguagem combinar os tipos de declarações procedimentais e não-procedimentais é geralmente desejável, porque as operações não-procedimentais aceleram e simplificam o uso da linguagem, enquanto que o código procedimental amplia o alcance dos aplicativos, dando-lhes mais flexibilidade de manipulação lógica.

Muitas ferramentas não-procedimentais evoluíram de “linguagens” para formas de interação do tipo GUI (*Graphic User Interface*) com a qual o usuário realiza ações através de componentes visuais, como apontar para áreas de planilhas ou texto com o *mouse* e clicar sobre ícones e menus. O usuário preenche determinados campos da tela, ou aponta para eles e, assim, manipula os dados. Dessa manipulação resultam declarações de especificações que o software pode traduzir em código executável (Martin, 1994, p. 259).

2.1.11 MÉTODOS MATEMÁTICOS FORMAIS

Martin (1994, p. 377) apresenta as seguintes considerações expostas por Tony Hoare, um dos pioneiros na definição dos métodos formais, que enfatiza em toda sua obra que programas são expressões matemáticas: “Os métodos formais descrevem com precisão sem precedentes e em cada mínimo detalhe o comportamento, intencional ou não intencional, do computador no qual eles são executados. Os programas têm se tornado tão complicados que não tem sido de maneira alguma prático raciocinar matematicamente sobre seu comportamento, pois o único meio de descobrir o que fazem é a experimentação. Infelizmente, sem fundamentação matemática, estes experimentos não correspondem a uma ciência, porque é impossível fazer generalizações a partir de seus resultados ou publicá-los para proveito de outros cientistas”.

Martin (1994, p. 379) afirma que as técnicas matemáticas propõem-se a comprovar a precisão de programas e criar técnicas de especificação para programas verificáveis.

Para tornar possível a aplicação da matemática à programação, deve-se dividir programas em componentes que sejam suficientemente pequenos e disciplinados. Dessa forma, os métodos baseados em objetos são candidatos à aplicação de técnicas matemáticas, pois são pequenos e executam uma operação específica.

É inconcebível que todos os projetistas de sistemas de informação adquiram habilidades matemáticas para verificar a correção de seus projetos. Por isso, as ferramentas CASE devem capacitar seus usuários a montar projetos a partir de elementos comprovadamente corretos. A matemática é vital, mas tem que ser abstraída dos profissionais de desenvolvimento de sistemas (Martin, 1994, p. 379).

2.1.12 MOTOR DE INFERÊNCIA

Segundo Martin (1994, p. 157), o desenvolvimento baseado em objetos deve ter por objetivo evitar a atividade de programação sempre e onde é possível. Dessa forma, o código dos sistemas deve ser gerado a partir de modelos que sejam facilmente compreensíveis aos usuários finais e com os quais eles possam fazer experimentações.

O comportamento desejado dos sistemas pode ser descrito com o auxílio de regras. Estas regras têm que ser rígidas e precisas, de forma que possam constituir a base para a geração de código. Contudo, as regras devem ser entendidas pelo usuário final, que deve ser capaz de verificar se as regras realmente expressam as políticas do negócio e o comportamento que deseja do sistema.

Um motor de inferência utiliza uma coleção de fatos e regras sobre uma área específica do conhecimento, e faz deduções usando técnicas de inferência lógica. Ele pode responder a uma solicitação, selecionando as regras e disparando-as, encadeando efetivamente as regras para executar um raciocínio diferencial. Através do encadeamento das regras, o motor de inferência permite realizar deduções complexas sem necessidade de codificação de um aplicativo.

O conceito de motor de inferência é empregado na construção de sistemas especialistas. Os sistemas especialistas armazenam conhecimento depurado em forma de fatos e regras e fazem deduções com a utilização de um motor de inferência. O objetivo é fazer com que a máquina dê pareceres em um domínio específico e restrito de conhecimento, como se fosse um humano especialista no assunto.

Da mesma forma que os programas tradicionais, que sofrem repetidos acréscimos, os grandes sistemas especialistas, que acumulam vastas coleções de regras, tornam-se difíceis de serem compreendidos. A solução consiste em projetar utilizando as técnicas de orientação a objetos, onde as classes têm métodos que são relativamente simples e o modelo global torna-se de fácil compreensão (Martin, 1994, p. 171).

2.1.13 TECNOLOGIA CLIENTE-SERVIDOR

A teoria cliente-servidor consiste em um conceito lógico. O cliente e o servidor podem ou não existir em máquinas físicas distintas. A tecnologia cliente-servidor é um dos paradigmas, ou modelos, para interação entre processos de software em execução concorrente.

Os processos cliente enviam pedidos a um processo servidor, que responde com o resultado destes pedidos. Como o nome indica, os processos servidores oferecem serviços aos seus clientes, normalmente por meio de processamento específico que só eles podem fazer. O processo cliente, livre da complexidade e esforço adicional do processamento da transação pode realizar outro trabalho útil. A interação entre os processos cliente e servidor é uma troca cooperativa, transacional, em que o cliente é ativo e o servidor é reativo, o que consiste na principal característica de uma tecnologia cliente-servidor.

Em um verdadeiro ambiente cliente-servidor, os processos cliente e servidor são distintos e podem executar na mesma máquina ou em máquinas diferentes. Logicamente, sistemas cliente-servidor são mais interessantes quando os processos cliente e servidor são executados em máquinas diferentes conectadas por meio de uma rede, cujas implementações mais comuns utilizam redes locais (LANs).

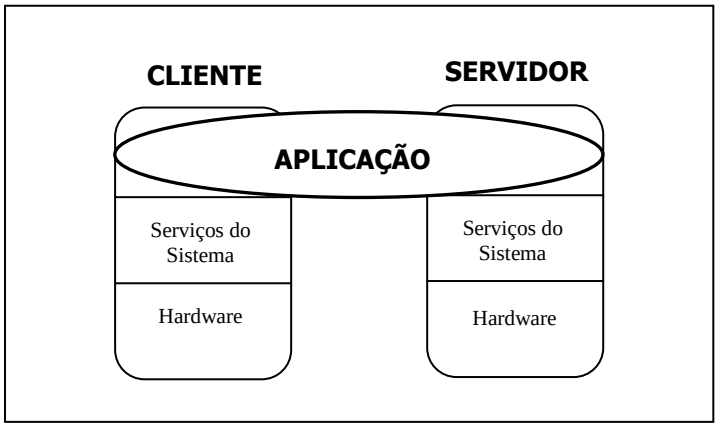
Existe uma crença de que a tecnologia cliente-servidor é sinônimo de bancos de dados que suportam a linguagem SQL (*Structured Query Language*). É verdade que o uso mais freqüente da tecnologia cliente-servidor encontra-se nas aplicações de bancos de dados em rede. Contudo, este representa apenas um tipo comum de aplicação da tecnologia, pois é possível construir uma série de sistemas cliente-servidor que não use qualquer comando SQL (Renaud, 1994, p. 3).

A afirmação anterior pode ser atestada por Martin (1994, p. 398) onde é descrita a arquitetura CORBA (*Common Object Request Broker Architecture*), do OMG (*Object Management Group*), a qual emprega conceitos de objetos-cliente e servidor, onde o objeto-cliente solicita serviços a um objeto-servidor e o objeto-servidor aceita a solicitação e executa o serviço. Martin (1994, p. 398) afirma ainda que o cliente e o servidor poderiam estar na mesma máquina, ou em máquinas diferentes.

Há três camadas básicas dentro de sistemas de cliente e servidor, ilustradas na Figura 2-4. Essas camadas são um tanto arbitrárias, contudo são úteis para entender os componentes de sistemas cliente-servidor. A primeira camada é camada de serviços do sistema, compreendendo o sistema operacional, componentes de rede e janelas, quando existirem. Esta camada inclui todo o software usado pela aplicação para controlar o hardware. A camada superior é a camada de aplicação, que consiste nos processos cliente-servidor e quaisquer outros processos da aplicação.

A principal diferença entre implementações de camada de aplicação e serviços do sistema é o local relativo dos componentes na hierarquia. Em implementações cliente-servidor na camada de aplicação, o “usuário” é a pessoa e os processos “cliente” e “servidor” compõem a aplicação. Em implementações de camada de serviços do sistema, o “usuário” normalmente é um programa de aplicação, o “cliente” é um processo redirecionador e o “servidor” é um processo de resposta.

Figura 2-4 – SISTEMAS CLIENTE-SERVIDOR



Fonte: (Renaud, 1994, p. 7)

2.1.14 BIBLIOTECAS DE CLASSES

Uma maneira particularmente importante de evitar programação é a de empregar classes reaproveitáveis. Algumas classes são independentes do aplicativo e provêem funções primitivas. O conjunto destas classes é chamado de biblioteca (Martin, 1994, p. 277).

Uma biblioteca de classes contém tipos de objetos implementados de forma reutilizável. A intenção é a de se alcançar o grau máximo de reaproveitamento em desenvolvimento de software. O software biblioteca-de-classe auxilia os técnicos de

desenvolvimento a encontrar, adaptar e utilizar as classes de que necessitam (Martin, 1994, p.12).

À medida que o reaproveitamento se torna uma cultura da empresa, mais aplicativos podem ser construídos a partir de componentes reaproveitáveis. Um projetista de sistemas, por exemplo, ao precisar de uma classe, deve primeiro verificar se existe alguma classe com funcionalidade similar na biblioteca para utilização direta, ou adaptação. Caso não exista, o técnico cria uma classe de maneira que esta possa ser reaproveitada futuramente e publica na biblioteca.

Quando uma empresa estabelece sua biblioteca de classes, com algumas classes de aplicativos desenvolvidas na própria empresa e outras provenientes de um fornecedor de software, existe um considerável ganho de produtividade no desenvolvimento (Martin, 1994, p. 297).

2.1.15 ANÁLISE E PROJETO ORIENTADOS A OBJETOS

Nas metodologias tradicionais de desenvolvimento, os modelos conceituais usados para análise diferem dos usados para o projeto. A programação tem ainda uma terceira visão do mundo. Os analistas usam modelos de entidade-relacionamento, decomposição funcional e matrizes; os projetistas usam diagramas de fluxo de dados, diagramas de estrutura e diagramas de ação; os programadores usam a estrutura da linguagem de programação utilizada.

Nas técnicas baseadas em objetos, analistas, projetistas, programadores e, particularmente importante, usuários finais, todos usam o mesmo modelo conceitual. A transição de análise para o projeto é tão natural que torna-se difícil especificar seus limites, especialmente quando uma ferramenta CASE utiliza o mesmo paradigma para análise e projeto, e gera código-fonte (Martin, 1994, p. 76). A Figura 2-5 apresenta um gráfico comparativo entre as metodologias tradicionais e a tecnologia baseada em objetos.

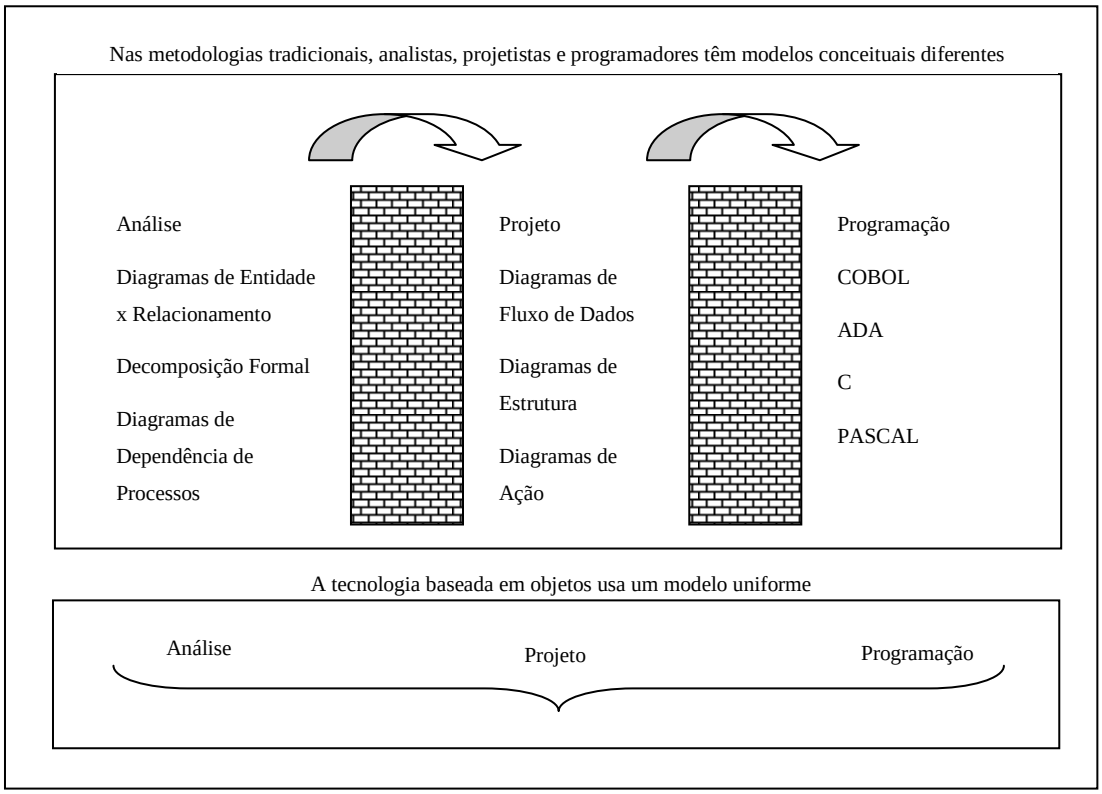
O emprego de um único modelo conceitual, suportado por uma ferramenta CASE integrada para o referido modelo, resulta nas seguintes vantagens:

a) maior produtividade;
b) menos erros;

c) melhor comunicação entre usuários, analistas, projetistas e desenvolvedores;
d) resultados de melhor qualidade;
e) mais flexibilidade;
f) maior criatividade.

O analista orientado a objetos enxerga o mundo em forma de objetos com estruturas de dados e métodos, bem como de eventos que disparam operações que alteram o estado dos objetos. As operações ocorrem quando os objetos solicitam outros objetos. O analista cria diagramas das estruturas dos objetos e dos eventos que alteram esses objetos. O modelo do projetista é similar ao modelo do analista, mas detalhado o suficiente para gerar código (Martin, 1994, p. 12).

Figura 2-5 – GRÁFICO COMPARATIVO ENTRE METODOLOGIAS TRADICIONAIS E A TECNOLOGIA BASEADA EM OBJETOS



Fonte: Baseado em (Martin, 1994, p.77)

Pressman (1995, p. 349) afirma que a modelagem de dados pode ser vista como um subconjunto da análise orientada a objeto, pois usando o diagrama entidade-relacionamento

como notação fundamental, a modelagem de dados concentra-se na definição de objetos de dados (objetos que não encapsulam processamento) e na maneira pela qual eles se relacionam.

A UML (*Unified Modeling Language*) é uma linguagem de modelagem que procura através de diagramas, definir as principais fases dentro das diversas metodologias baseadas em objetos, tais como as fases de análise, projeto e implementação. A linguagem UML pretende unificar as formas de modelagem entre os diversos métodos disponíveis (Rezende, 1999, p. 207).

Rezende (1999, p. 208) define que uma linguagem de modelagem consiste das notações (símbolos usados em um modelo), e do conjunto de regras que determinam como devem ser usadas estas notações.

2.2 A FERRAMENTA CASE ERWIN

O ERwin é uma ferramenta CASE de modelagem de dados, produzida pela empresa *Computer Associates* (CA), que suporta a manutenção de bancos de dados em vários ambientes.

Pompilho (1994, p. 183) conceitua modelagem de dados como um método de análise de sistemas que busca especificar, a partir dos fatos relevantes que estejam associados ao domínio do conhecimento analisado, a perspectiva dos dados, permitindo organizá-los em estruturas bem definidas, estabelecer regras de dependência entre eles, produzindo um modelo expresso por uma representação, ao mesmo tempo, descritiva e diagramática.

Os componentes primitivos do modelo de dados, que representam o mundo real, são:

a) entidade: algo sobre o qual deseja-se manter informações e que desempenha papel específico no sistema que está sendo modelado, cuja identificação depende inteiramente do contexto em que estiver inserida. Um grupo de entidades possuindo os mesmos atributos forma um “conjunto” ou uma “classe de entidades”;
b) atributo: característica ou propriedade da entidade;
c) domínio: conjunto de valores válidos para um determinado atributo;
d) relacionamento: mapeamento, ou regra de associação entre dois conjuntos, estabelecido entre duas classes de entidades.

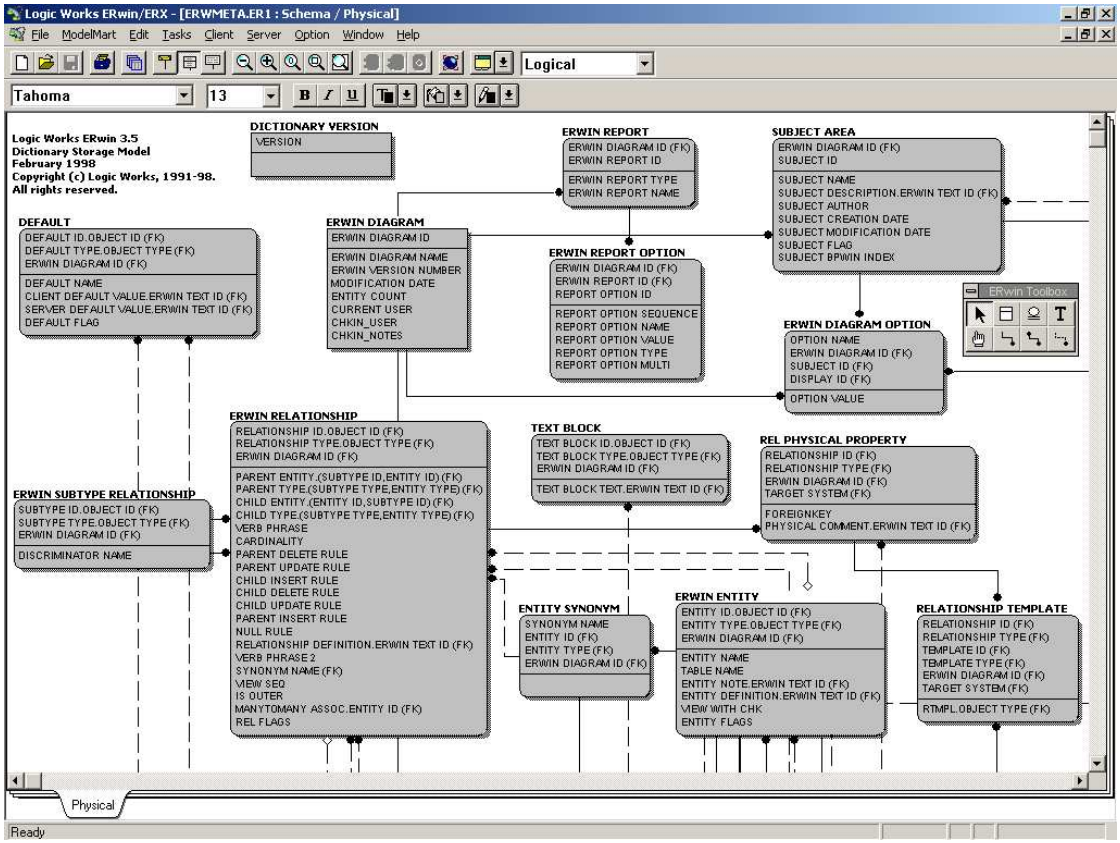
Estes componentes são representados através do Diagrama Entidade x Relacionamento (DER), que consiste em uma técnica diagramática fundamental para a modelagem de dados. Existem várias notações que podem ser usadas para expressar o DER (Pressman, 1995, p. 346) (Pompilho, 1994, p. 183).

O ERwin permite a confecção do DERs lógico e físico, suportando as notações IDEF1X (*Integration DEFinition for Information Modeling*), IE (*Information Engineering*) e DM (*Dimensional Modeling*), sendo a última aplicada somente à visão lógica do diagrama.

O ERwin oferece um mecanismo de acesso direto ao repositório de dados. Para tanto, permite a exportação das especificações dos diagramas para um banco de dados relacional, gerado a partir de um modelo de dados especial, denominado *ERwin Dictionary Metamodel*, ou Meta-modelo do Dicionário do ERwin, fornecido pelo fabricante. Este mecanismo será ilustrado em detalhes na seção 3.2.1.

A Figura 2-6 apresenta a janela principal da ferramenta ERwin, contendo algumas tabelas do meta-modelo.

Figura 2-6 - JANELA PRINCIPAL DO ERWIN



A ferramenta administra um dicionário de domínios (*Domain Dictionary*) que organiza uma estrutura hierárquica, a qual permite aos domínios herdar características de outros domínios e realizar especializações.

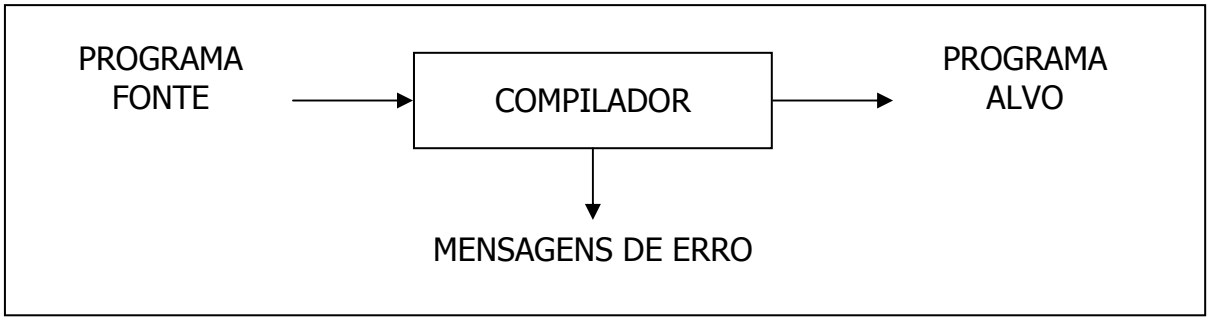
O ERwin permite a organização dos componentes do modelo de dados em áreas de assunto (*Subject Areas*) visando facilitar a visualização das entidades relevantes em determinado enfoque lógico. Ainda relacionada à apresentação do diagrama, a ferramenta permite a seleção individual das características que se deseja visualizar no diagrama, recurso denominado *Stored Display*.

O recurso de geração de esquemas (*Schema Generation*) permite a criação dos componentes do diagrama no banco de dados utilizado. O ERwin possui uma metalinguagem interpretada para geração dos scripts SQL correspondentes ao banco de dados selecionado. As *triggers* também podem ser escritas através da metalinguagem, e geradas opcionalmente pela ferramenta.

2.3 COMPILADORES

Um compilador consiste em um software que lê um programa escrito numa linguagem – a linguagem fonte – e o traduz em programa equivalente numa outra linguagem – a linguagem alvo, conforme esquematizado na Figura 2-7 (Aho, 1995, p. 1).

Figura 2-7 – UM COMPILADOR



Fonte: (Aho, 1995, p. 1)

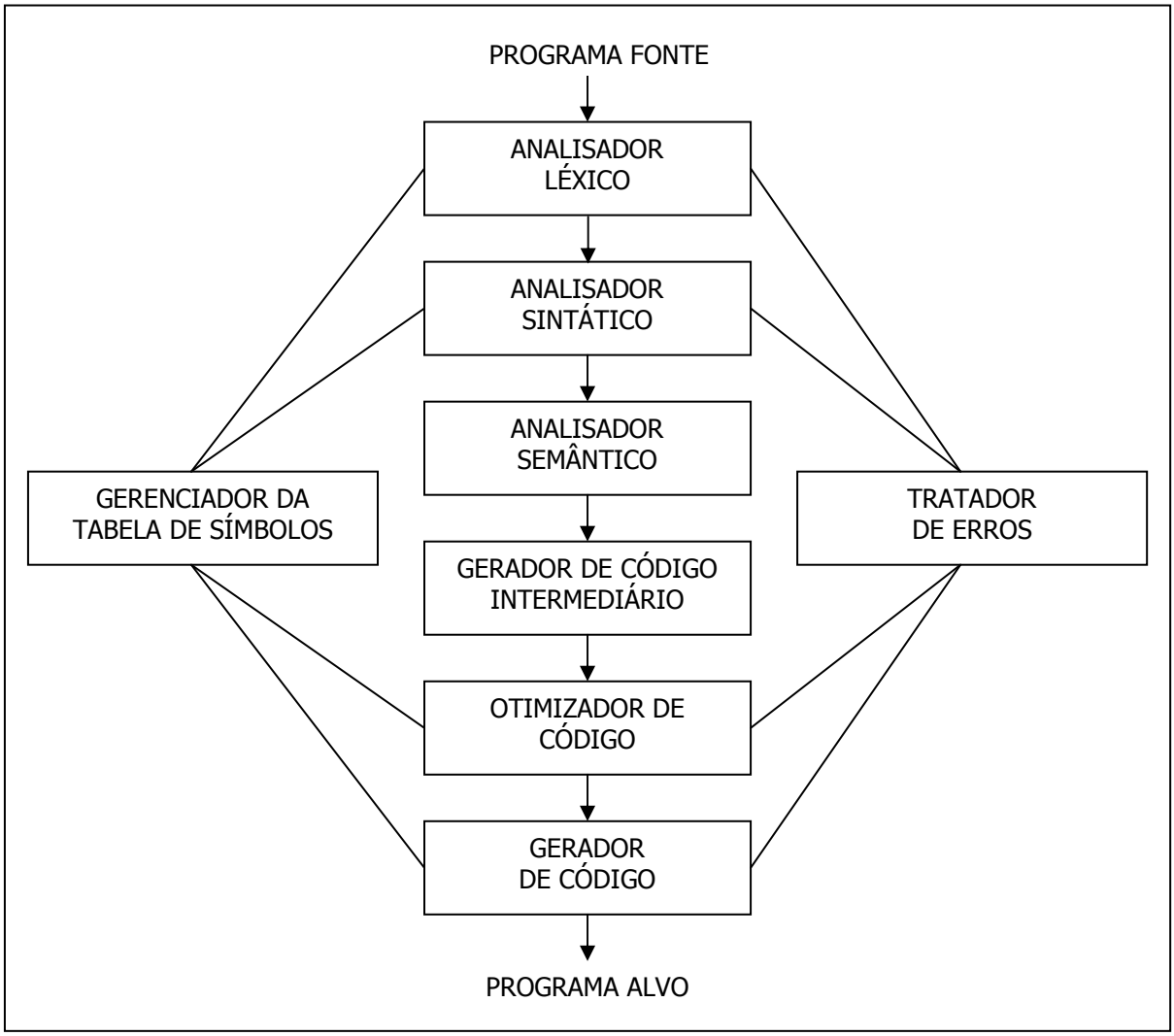
Adicionalmente à tradução propriamente dita, cabe aos compiladores, em geral, executar inúmeras funções auxiliares, sendo que a mais importante é a detecção de erros. Esta função permite que o compilador identifique ao usuário os pontos do programa fonte onde existem erros. Um erro corresponde a uma não conformidade com a gramática especificada

para a linguagem, cujos conceitos serão apresentados no decorrer deste capítulo (Aho, 1995, p. 1,5), (Neto, 1987, p. 4).

Segundo Aho (1995, p. 1), o processo de compilação é dividido em duas partes: a análise e a síntese. A parte de análise divide o programa fonte nas partes constituintes e cria uma representação intermediária do mesmo. A parte de síntese constrói o programa alvo desejado, a partir da representação intermediária resultante da fase de análise.

Conceitualmente, um compilador opera em fases, agrupadas nas partes de análise ou síntese, cada uma das quais transforma o programa fonte de uma representação para outra. A Figura 2-8 apresenta uma composição típica de um compilador.

Figura 2-8 – FASES DE UM COMPILADOR



Fonte: (Aho, 1995, p. 5)

As fases de análise correspondem aos analisadores léxico, sintático e semântico, sendo que as fases de síntese englobam o gerador de código intermediário, o otimizador de código e o gerador de código. Como ilustrado na Figura 2-8, todas as fases interagem com o gerenciador da tabela de símbolos e com o tratador de erros.

2.3.1 INTERPRETADORES

Convém ressaltar que o presente trabalho emprega técnicas da área de compiladores objetivando implementar um interpretador para uma pseudolinguagem. Aho (1995, p. 2) conceitua o interpretador como uma ferramenta de software que, em lugar de produzir um programa alvo como resultado da tradução, realiza as operações especificadas pelo programa fonte.

Aho (1995, p. 2) apresenta ainda que os interpretadores são freqüentemente usados para executar linguagens de comandos, dado que cada operador numa determinada linguagem é usualmente uma invocação de uma rotina complexa, como por exemplo, os interpretadores de *query* de bancos de dados.

Segundo Neto (1987, p. 15) existem duas classes principais de interpretadores: os diretos e os processadores de linguagens. Os interpretadores diretos apresentam os resultados da execução durante o processo de análise do programa fonte. Os processadores de linguagem traduzem o programa-fonte para um código intermediário, o qual, ao invés de ser convertido para linguagem de máquina, é utilizado diretamente para ser executado.

O presente trabalho fundamentará, portanto, as fases do processo de compilação que atendem à implementação de um interpretador direto, iniciando pelas fases de análise descritas a seguir.

2.3.2 ANÁLISE LÉXICA

A tarefa principal da análise léxica é a de ler um fluxo de caracteres de entrada que constituem o programa-fonte e produzir uma seqüência de *token*s que deve ser utilizada pelo analisador sintático, ou *parser*.

Como o analisador léxico é a parte do compilador que lê o texto-fonte, também pode realizar algumas tarefas secundárias, como remover os comentários e espaços em branco e correlacionar as mensagens de erro com o programa-fonte, identificando o número da linha onde o erro ocorreu (Aho, 1995, p. 38).

Antes de prosseguir, convém esclarecer o significado dos termos *token*, padrão e lexema, comumente utilizados na análise léxica.

Um *token* representa uma seqüência de caracteres com um significado coletivo, sendo a menor unidade de informação de uma linguagem.

Um padrão é uma regra que descreve o conjunto de lexemas que podem representar um *token* particular nos programas-fonte.

Um lexema corresponde a um conjunto de caracteres no programa-fonte que é reconhecido pelo padrão de algum *token*.

Para ilustrar os conceitos, o Quadro 2-2 apresenta exemplos de *token*s.

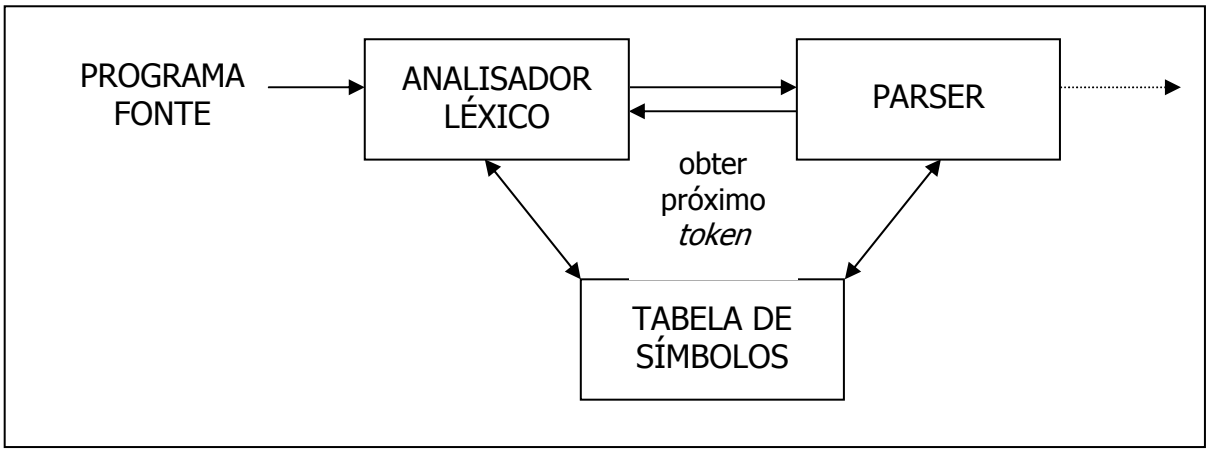
Quadro 2-2 – EXEMPLOS DE *TOKENS*

TOKEN	**LEXEMAS EXEMPLO**	**DESCRIÇÃO INFORMAL DO PADRÃO**
const	`const`	constante
if	`if`	comando condicional
relação	`<, <=, =, <>, >, >=`	operadores relacionais
id	`pi, contador, D2`	letra seguida por letras e/ou dígitos
num	`3.1416, 0, 6.02E23`	qualquer constante numérica
literal	“conteúdo da memória”	quaisquer caracteres entre aspas, exceto aspas

Fonte: (Aho, 1995, p. 39)

A interação entre o analisador léxico e o *parser* é ilustrada pela Figura 2-9. Pode-se observar que o *parser* solicita um próximo *token* ao analisador léxico, formando uma espécie de par produtor x consumidor.

Figura 2-9 – INTERAÇÃO ENTRE ANALISADOR LÉXICO E *PARSER*



Fonte: (Aho, 1995, p. 39)

2.3.2.1 EXPRESSÕES REGULARES

Antes de iniciar a abordagem das expressões regulares, convém introduzir os conceitos relacionados de alfabeto, cadeia e linguagem, como segue:

a) alfabeto: qualquer conjunto finito de símbolos. EBCDIC e ASCII são dois exemplos de alfabetos de computadores;
b) cadeia: seqüência de símbolos retirados de um alfabeto. Tem como sinônimo o termo “palavra”. O comprimento da cadeia *s* corresponde ao número de ocorrências de símbolos em *s*. A cadeia vazia, denotada por $\in$, é uma cadeia especial de comprimento zero;
c) linguagem: qualquer conjunto de cadeias sobre algum alfabeto fixo, como por exemplo, o conjunto de todos os programas Pascal sintaticamente bem-formados.

Existem várias operações importantes que podem ser aplicadas às linguagens, porém, para a análise léxica possuem maior relevância a união, concatenação e fechamento, apresentadas no Quadro 2-3.

As expressões regulares são uma notação importante para especificar padrões. Cada padrão corresponde a um conjunto de cadeias e, dessa forma, as expressões regulares servirão como nome para conjuntos de cadeias. A linguagem denotada por uma expressão regular é dita conjunto regular (Aho, 1995, p. 43).

Quadro 2-3 – DEFINIÇÕES DE OPERAÇÕES EM LINGUAGENS

OPERAÇÃO	DEFINIÇÃO
união de L e M, escrita $L \cup M$	$L \cup M = \{ s \mid s$ está em L ou s está em $M \}$
concatenação de L e M, escrita LM	$LM = \{ st \mid s$ está em L e t está em $M \}$
fechamento Kleene de L, escrito L^*	$L^* = \cup_{i=0}^{\infty} L^i$; L^* denota "zero ou mais concatenações de" L
fechamento positivo de L, escrito L^+	$L^+ = \cup_{i=1}^{\infty} L^i$; L^+ denota "uma ou mais concatenações de" L

Fonte: (Aho, 1995, p. 43)

2.3.2.2 DEFINIÇÕES REGULARES

Se $\sum$ for um alfabeto de símbolos básicos, então uma definição regular é uma seqüência de definições da forma expressa no Quadro 2-4, onde cada d_i é um nome distinto e cada r_i, uma expressão sobre os símbolos em $\sum \cup \{d_1, d_2, ..., d_{i-1}\}$, isto é, os símbolos básicos e os nomes previamente definidos. Restringindo-se cada r_i a símbolos de $\sum$ e aos nomes previamente definidos, pode-se construir uma expressão regular sobre $\sum$ para qualquer r_i substituindo-se repetidamente os nomes de expressões regulares pelas expressões que os mesmos denotam. Se r_i usasse d_j para algum $j \geq i$, então r_i poderia ser recursivamente definida e o processo de substituição não terminaria (Aho, 1995, p. 44).

Quadro 2-4 – DEFINIÇÕES REGULARES

$$d_1 \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

$$\ldots$$

$$d_n \rightarrow r_n$$

Fonte: (Aho, 1995, p. 44)

Para ilustrar o conceito, o Quadro 2-5 apresenta um exemplo da definição regular para o conjunto de identificadores Pascal, o qual consiste em um conjunto de cadeias de letras e dígitos, iniciando por uma letra.

Quadro 2-5 – EXEMPLO DE DEFINIÇÃO REGULAR

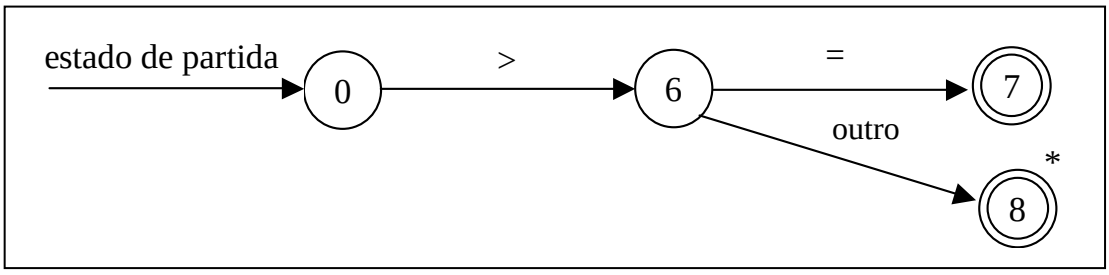
```
letra  → A | B ... | Z | a | b | ... | z
digito → 0 | 1 | ... | 9
id     → letra | (letra | digito)*
```

Fonte: (Aho, 1995, p. 44)

2.3.2.3 DIAGRAMAS DE TRANSIÇÕES

O diagrama de transições consiste em um fluxograma estilizado que delineia as ações que tomam lugar quando um analisador léxico é chamado pelo *parser* a fim de obter o próximo *token*. Conforme o exemplo da Figura 2-10, as posições num diagrama de transições são desenhadas como círculos e são chamadas de estados. Os estados são conectados por setas, chamadas de lados. Os lados que deixam o estado *s* possuem rótulos indicando os caracteres de entrada que podem aparecer após o diagrama de transições ter atingido o estado *s*. O rótulo outro se refere a qualquer caractere que não seja indicado por qualquer um dos lados que deixam *s*.

Figura 2-10 – EXEMPLO DE UM DIAGRAMA DE TRANSIÇÕES



Fonte: (Aho, 1995, p. 46)

Um estado é rotulado como estado de partida, o qual consiste no estado inicial do diagrama de transições, onde reside o controle quando tem início o reconhecimento de um *token*. Certos estados podem ter ações que são executadas quando atingidos pelo fluxo de controle. Ao entrar em um estado lê-se o próximo caractere da entrada. Se existir um lado a partir do estado corrente, cujo rótulo se iguale a esse caractere de entrada, o fluxo é desviado então para o estado apontado por aquele lado. De outra forma, indica-se uma falha. O círculo duplo nos estados 7 e 8 da Figura 2-10 informa que os mesmos são estados de aceitação, ou

estados finais, nos quais os *tokens* correspondentes foram encontrados. O símbolo * é utilizado para indicar estados onde é necessária uma retração da entrada (Aho, 1995, p. 46).

2.3.3 ANÁLISE SINTÁTICA

A análise sintática tem como tarefa principal obter uma cadeia de *token*s proveniente do analisador léxico e verificar se a mesma pode ser gerada pela gramática da linguagem-fonte. Durante este processo deve relatar quaisquer erros de sintaxe de uma forma inteligível (Aho, 1995, p. 72). Um analisador sintático também é conhecido como *parser*.

Os métodos de análise sintática mais comumente usados nos compiladores são classificados como *top-down* ou como *bottom-up*. Como o próprio nome sugere, os analisadores *top-down* constroem árvores do topo (raiz) para o fundo (folhas), enquanto que os analisadores *bottom-up* realizam o inverso. Em ambos os casos, a entrada é varrida da esquerda para a direita, um símbolo de cada vez (Aho, 1995, p. 72).

Segundo Neto (1987, p. 126), a análise sintática engloba diversas funções adicionais de grande importância, relacionadas a seguir:

a) recuperação de erros: possibilidade do analisador recuperar-se dos erros que ocorram mais comumente, a fim de poder continuar processando o resto de sua entrada;
b) correção de erros: possibilidade do próprio analisador introduzir alterações no código-fonte incorreto, tornando-o, pelo menos, sintaticamente correto;
c) montagem da árvore abstrata da sentença: levantar para a cadeia de entrada a seqüência de derivação da mesma;
d) comando de ativação do analisador léxico: ativar o analisador léxico, em função da sintaxe, quando detectar a necessidade de novos *token*s;
e) ativação das rotinas de análise semântica: rotinas responsáveis pelo gerenciamento dos objetos da linguagem;
f) ativação das rotinas de síntese do código objeto: estas rotinas encarregam-se de produzir o código-objeto correspondente ao programa-fonte, sendo o núcleo das atividades semânticas do compilador.

Nos capítulos posteriores serão fundamentadas as técnicas de análise sintática utilizadas na especificação da linguagem implementada no trabalho.

2.3.3.1 GRAMÁTICAS LIVRES DE CONTEXTO

Uma gramática corresponde ao conjunto de leis de formação que definem de maneira rigorosa o modo de formar textos corretos em uma linguagem. A linguagem, por sua vez, é definida como sendo o conjunto de todos os textos que podem ser gerados a partir da gramática que a define (Neto, 1987, p. 5).

Aho (1995, p. 74) afirma que muitas construções de linguagens de programação possuem uma estrutura inerentemente recursiva que pode ser identificada por gramáticas livres de contexto, como exemplificado no Quadro 2-6.

Quadro 2-6 - REGRA PARA ENUNCIADO CONDICIONAL

Se S_1 e S_2 são enunciados e E é uma expressão, então "**if** E **then** S_1 **else** S_2" é um enunciado.

Fonte: (Aho, 1995, p. 74)

Utilizando-se a variável sintática *cmd* para denotar a classe de comandos e *expr* para a classe de expressões, pode-se expressar o exemplo do enunciado condicional usando a produção do Quadro 2-7. A seta deve ser lida como "pode ter a forma", ou "define-se como" (Aho, 1995, p. 74), (Neto, 1987, p. 48).

Quadro 2-7 - PRODUÇÃO GRAMATICAL PARA ENUNCIADO CONDICIONAL

cmd → **if** *expr* **then** *cmd* **else** *cmd*

Fonte: (Aho, 1995, 74)

Segundo Aho (1995, p. 75), uma gramática livre de contexto é formada pelos seguintes componentes:

a) terminais: são símbolos básicos a partir dos quais as cadeias são formadas. Também denominados de "*tokens*". Na produção do Quadro 2-7 cada uma das palavras-chave *if*, *then* e *else* é um terminal;

b) não-terminais: são variáveis sintáticas que denotam novas produções. Na produção do Quadro 2-7 *cmd* e *expr* são não-terminais. Os não-terminais definem conjuntos de cadeias que auxiliam a definição da linguagem gerada pela gramática e impõem uma estrutura hierárquica à linguagem;
c) símbolo de partida: corresponde a um não terminal que denota um conjunto de cadeias que forma a linguagem definida pela gramática;
d) conjunto de produções: especificam a forma pela qual os terminais e não-terminais podem ser combinados a fim de formar cadeias. Cada produção consiste em um não-terminal (lado esquerdo da produção), seguido por uma seta ($\rightarrow$) e por uma cadeia de terminais e não-terminais (lado direito da produção).

2.3.3.2 BNF (BACKUS NAUR FORM)

Neto (1987) expõe que uma metalinguagem consiste em uma linguagem utilizada como forma de representação ou de definição de outras linguagens, exemplificando, através das linguagens naturais, que uma gramática da língua inglesa pode ser redigida em francês, para uso de pessoas da língua francesa. Neste caso, a linguagem definida é o inglês, e a metalinguagem é o francês.

A BNF (*Backus Naur Form*) é um dos exemplos mais conhecidos de metalinguagem, e tem sido utilizada com sucesso para especificação da sintaxe de linguagens de programação, desde que foi publicada pela primeira vez no relatório de especificação da linguagem Algol 60 por John Backus e Peter Naur.

Trata-se de uma notação recursiva de formalização da sintaxe de linguagens através de produções gramaticais, permitindo assim a criação de dispositivos de geração de sentenças. Para tanto, cada produção corresponde a uma regra de substituição, em que para cada símbolo da metalinguagem são associadas uma ou mais cadeias de símbolos, indicando as diversas possibilidades de substituição. Os símbolos em questão correspondem a não-terminais da gramática que está sendo especificada. As cadeias podem ser formadas de terminais e ou não-terminais e do símbolo $\in$, que representa a cadeia vazia. A simbologia adotada pela BNF é apresentada no Quadro 2-8 (Neto, 1987, p. 48).

Quadro 2-8 – SIMBOLOGIA ADOTADA PELA BNF

$\rightarrow$	é o símbolo da metalinguagem que associa a um não-terminal um conjunto de cadeias de terminais e/ou não-terminais, incluindo o símbolo da cadeia vazia. O não-terminal em questão é escrito à esquerda deste símbolo, e as diversas cadeias, à sua direita. Lê-se como “pode ser”, ou “define-se como”.
|	é o símbolo da metalinguagem que separa as diversas cadeias que constam à direita do símbolo $\rightarrow$. Lê-se como “ou”.
$\in$	representa a cadeia vazia na notação BNF.
X	representa um terminal da linguagem que está sendo definida, e pertence ao conjunto de todos os *tokens* que compõem as sentenças da linguagem.
E	representa um não terminal, cujo nome é dado por uma cadeia de caracteres quaisquer.

Fonte: Baseado em (Neto, 1987, p. 48)

2.3.3.3 ANÁLISE GRAMATICAL *TOP-DOWN*

A análise gramatical *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, segundo Aho (apud Schmitz, 1999).

A análise gramatical descendente recursiva é um método *top-down* de análise sintática, no qual executa-se um conjunto de procedimentos recursivos para processar a entrada. Um procedimento é associado a cada não-terminal de uma gramática, iniciando com uma chamada para o procedimento correspondente ao não-terminal de partida.

2.3.3.4 ELIMINAÇÃO DA AMBIGUIDADE

Segundo Aho (1995, p. 77) uma gramática é dita ambígua quando produz mais de uma árvore gramatical para alguma sentença, ou, de outra forma, que produz mais de uma derivação à esquerda, ou à direita para a mesma sentença.

Algumas vezes uma gramática ambígua pode ser reescrita de forma a eliminar a ambigüidade. O Quadro 2-9, o Quadro 2-10, o Quadro 2-11 e a Figura 2-11 apresentam um exemplo de gramática ambígua e de como reescrevê-la para eliminar a ambigüidade. No Quadro 2-9 o termo "*outro*" significa qualquer outra sentença.

Quadro 2-9 - GRAMÁTICA AMBÍGUA DO "*ELSE-VAZIO*"

```
cmd → if expr then cmd
    | if expr then cmd else cmd
    | outro
```

Fonte: (Aho, 1995, p. 78)

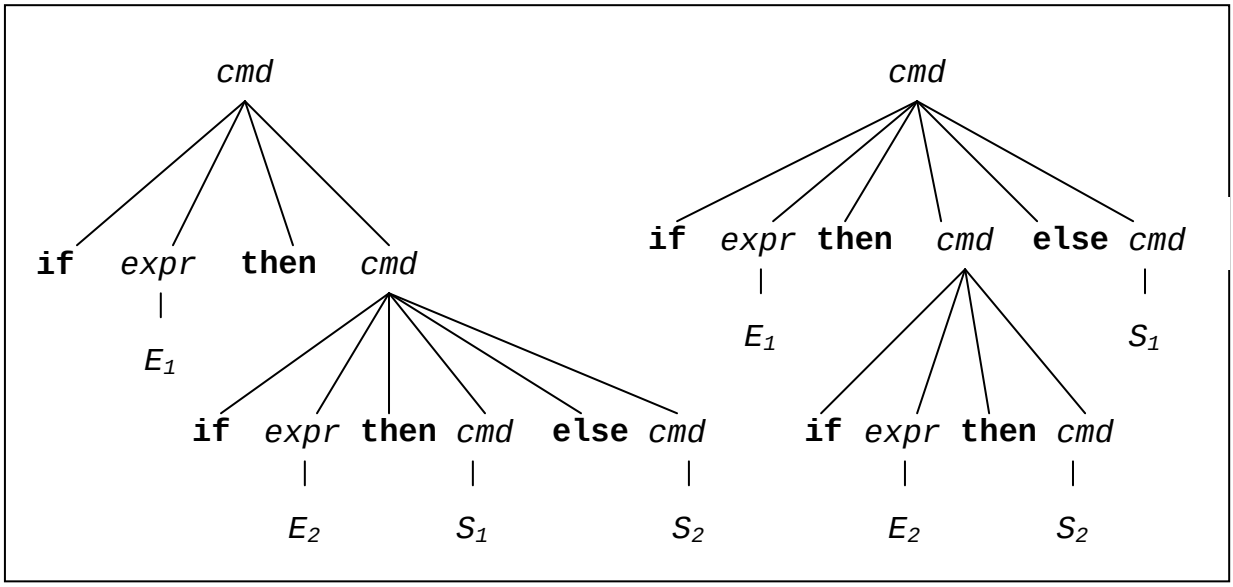
Quadro 2-10 – SENTENÇA CONDICIONAL COMPOSTA

```
if E1 then if E2 then S1 else S2
```

Fonte: (Aho, 1995, p. 78)

A gramática descrita no Quadro 2-9 é ambígua, uma vez que a cadeia exemplo apresentada no Quadro 2-10 possui as duas árvores (Figura 2-11).

Figura 2-11 – DUAS ÁRVORES GRAMATICAIS PARA UMA SENTENÇA AMBÍGUA



Fonte: (Aho, 1995, p. 78)

Em todas as linguagens de programação com enunciados condicionais desta forma, a primeira árvore gramatical é preferida. A regra geral é "associar cada *else* ao *then* anterior mais próximo ainda não associado". Esta regra de inambigüidade pode ser incorporada diretamente à gramática. Por exemplo, pode-se reescrever a gramática do Quadro 2-9 sob a forma inambígüa, apresentada no Quadro 2-11.

Quadro 2-11 – GRAMÁTICA DO "*ELSE-VAZIO*" APÓS A ELIMINAÇÃO DA AMBIGUIDADE

```
                cmd → cmd_associado
                    | cmd_não_associado
    cmd_associado → if expr then cmd_associado
                      else cmd_associado
                    | outro
cmd_não associado → if expr then cmd
                    | if expr then cmd_associado
                      else cmd_não associado
```

Fonte: (Aho, 1995, p. 79)

2.3.3.5 ELIMINAÇÃO DA RECURSÃO À ESQUERDA

Uma gramática é recursiva à esquerda quando possui um não-terminal A tal que exista uma derivação $A \rightarrow A\alpha \mid \beta$ para alguma cadeia α (Aho, 1995, p. 79).

Os métodos de análise sintática *top-down* não podem processar gramáticas recursivas à esquerda e, conseqüentemente, uma transformação que elimina a recursão à esquerda é necessária.

Exemplificando o método de eliminação, a produção recursiva $A \rightarrow A\alpha \mid \beta$ poderia ser substituída pelas produções não recursivas apresentadas no Quadro 2-12.

Quadro 2-12 – PRODUÇÕES NÃO RECURSIVAS

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \in$$

Fonte: (Aho, 1995, p. 79)

Não importa quantas produções –A existam, pode-se eliminar a recursão imediata das mesmas pela seguinte técnica. Primeiro, agrupa-se as produções –A como no Quadro 2-13, onde nenhum β começa por uma A. Em seguida, substitui-se as produções –A conforme o Quadro 2-14.

Quadro 2-13 – PRODUÇÕES-A AGRUPADAS

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

Fonte: (Aho, 1995, p. 79)

Quadro 2-14 – PRODUÇÕES-A COM A ELIMINAÇÃO DA RECURSIVIDADE À ESQUERDA

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$
$A' \rightarrow \alpha 1 A' \mid \alpha 2 A' \mid \ldots \mid \alpha m A'$

Fonte: (Aho, 1995, p. 79)

O não-terminal A gera as mesmas cadeias que antes, mas já não é recursiva à esquerda. Este procedimento elimina todas as recursões imediatas à esquerda das produções A e A’ (desde que nenhum α1 seja ∈), mas não elimina a recursão à esquerda envolvendo derivações em um ou mais passos (Aho, 1995, p. 79). O Quadro 2-15 apresenta um exemplo de gramática que apresenta recursão à esquerda indireta.

Quadro 2-15 –GRAMÁTICA COM RECURSÃO INDIRETA

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid \in$

Fonte: (Aho, 1995, p. 79)

O não-terminal S é recursivo à esquerda porque S → Aa → Sda, mas não é imediatamente recursivo. Aho (1995, p.79) apresenta um método para solução deste problema que consiste na substituição das produções da forma inversa àquela em que foram criadas, portanto substitui S na produção A resultando na produção A → Ac | Aad | bd | ∈, que passa a apresentar recursividade direta, que após eliminada gera as produções dispostas no Quadro 2-16.

Quadro 2-16 – GRAMÁTICA COM A RECURSÃO INDIRETA ELIMINADA

$$S \rightarrow Aa \mid b$$
$$A \rightarrow bdA' \mid A'$$
$$A' \rightarrow cA' \mid adA' \mid \in$$

Fonte: (Aho, 1995, p. 79)

2.3.3.6 FATORAÇÃO À ESQUERDA

A fatoração à esquerda é uma transformação gramatical útil para a criação de uma gramática adequada à análise sintática preditiva. A idéia básica reside em, quando não estiver claro qual das duas produções alternativas usar para expandir um não-terminal A, deve-se estar capacitado a reescrever as produções-A e postergar a decisão até que tenha sido visto o suficiente da entrada para realizar-se a escolha certa. Uma construção onde a fatoração à esquerda torna-se necessária é apresentada no Quadro 2-17.

Quadro 2-17 – EXEMPLO DE PRODUÇÕES QUE REQUEREM FATORAÇÃO À ESQUERDA

cmd → **if** *expr* **then** *cmd* **else** *cmd*
| **if** *expr* **then** *cmd*

Fonte: (Aho, 1995, p. 79)

Considerando as produções do Quadro 2-17, ao identificar-se o *token* de entrada *if*, não se pode imediatamente dizer qual produção escolher a fim de expandir *cmd*. Em geral, 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, não sabe-se se a expansão é de A em $\alpha\beta_1$, ou A em $\alpha\beta_2$. Entretanto, pode-se adiar a decisão expandido-se A para αA'. Então após identificar-se a entrada derivada a partir de α, expande-se A' em β_1 ou em β_2. Logo, as produções originais, fatoradas à esquerda correspondem às apresentadas no Quadro 2-18.

Quadro 2-18 – PRODUÇÃO EXEMPLO FATORADA À ESQUERDA

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

Fonte: (Aho, 1995, p. 80)

2.3.4 ANÁLISE SEMÂNTICA

A terceira grande tarefa do compilador refere-se à tradução propriamente dita do programa-fonte para a forma do código-objeto. Em geral, a geração de código vem acompanhada das atividades de análise semântica, responsáveis pela captação do sentido do texto-fonte, operação essencial à realização da tradução do mesmo, por parte das rotinas de geração de código. Muitos autores denominam essa classe de tarefas do compilador de ações semânticas (Neto, 1987, p. 130).

A semântica de uma sentença consiste no exato significado por ela assumido dentro do texto em que tal sentença se encontra, no programa-fonte. A semântica de uma linguagem é a interpretação que se pode atribuir ao conjunto de todas as suas sentenças (Neto, 1987, p. 130).

A seguir estão relacionadas algumas das principais funções das ações semânticas do compilador, apresentadas por Neto (1987, p. 130):

a) criar e manter as 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 de identificadores;
f) verificar o escopo dos identificadores;
g) verificar a compatibilidade de tipos;
h) representar o ambiente de execução dos procedimentos;
i) efetuar a tradução do programa.

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, sendo que escondem muitos detalhes de implementação e liberam o usuário de especificar exatamente a ordem na qual as traduções têm lugar. Os esquemas de tradução indicam a ordem na qual as regras semânticas são avaliadas e assim permitem que alguns detalhes de implementação sejam evidenciados (Aho, 1995, p. 120).

2.3.4.1 GRAMÁTICA DE ATRIBUTOS

Uma definição dirigida pela sintaxe é uma generalização de uma gramática livre de contexto na qual cada símbolo gramatical possui um conjunto associado de atributos, particionados em dois subconjuntos, chamados de atributos sintetizados e atributos herdados daquele símbolo gramatical. Um atributo pode representar qualquer tipo de informação como uma cadeia, um número, um tipo, uma localização de memória, etc. O valor de um atributo em um nó da árvore gramatical é definido por uma regra semântica associada à produção daquele nó (Aho, 1995, p. 120).

Numa definição dirigida pela sintaxe, cada produção gramatical $A \rightarrow \alpha$ tem associada a si um conjunto de regras semânticas da forma $b := f(c_1, c_2, ..., c_k)$, onde f é uma função e vigora uma das duas situações seguintes, mas não ambas:

a) b é um atributo sintetizado de A e $c_1, c_2, ..., c_k$ são atributos pertencentes aos símbolos gramaticais da produção, ou
b) b é um atributo herdado, pertencente a um dos símbolos gramaticais do lado direito da produção, e $c_1, c_2, ..., ck$ são atributos pertencentes aos símbolos gramaticais da produção.

Tanto para atributos herdados quanto para atributos sintetizados, o atributo b depende dos atributos $c_1 , c_2 , \ldots, c_k$. As funções nas regras semânticas serão freqüentemente escritas como expressões e chamadas de procedimentos ou fragmentos de programas.

Uma gramática de atributos consiste em uma definição dirigida pela sintaxe, na qual as funções das regras semânticas não têm efeitos colaterais, ou seja, não alteram seus parâmetros ou variável não local (Aho, 1995, p. 120).

2.3.4.1.1 ATRIBUTOS SINTETIZADOS

Um atributo é dito sintetizado se seu valor num nó da árvore gramatical é determinado a partir dos valores dos atributos dos filhos daquele nó. Os atributos sintetizados possuem a propriedade de que podem ser avaliados durante um único caminhamento *bottom-up* da árvore gramatical. Uma definição dirigida pela sintaxe que use exclusivamente atributos sintetizados é dita uma definição S-atribuída (Aho, 1995, p. 15).

Considerando como exemplo a produção *T ::= T + F,* o valor do atributo *T.val* a esse nó é definido no Quadro 2-19.

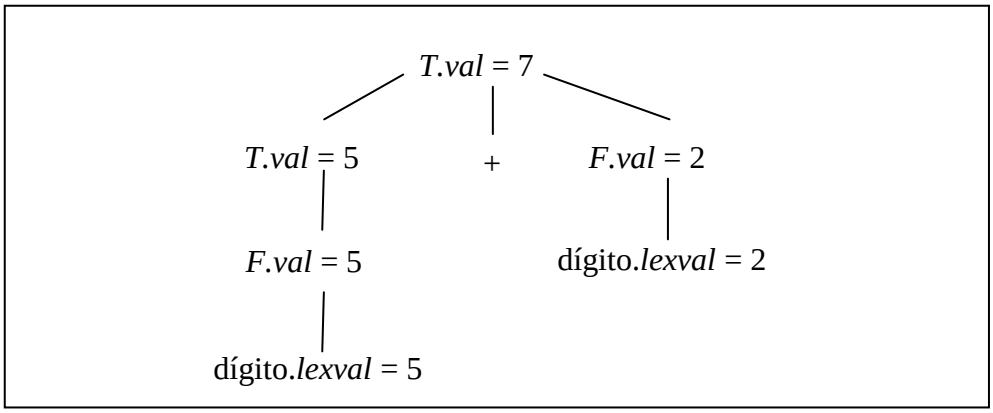
Quadro 2-19 - DEFINIÇÃO DIRIGIDA PELA SINTAXE DO NÃO TERMINAL *T*

PRODUÇÃO	REGRAS SEMÂNTICAS
T ::= T_1 + F	*T.val := T_1.val + F.val*
T ::= F	*T.val := F.val*
F ::= **dígito**	*F.val :=* **dígito***.lexval*

Fonte: Baseado em (Aho, 1995, p. 130)

A Figura 2-12 apresenta uma árvore gramatical anotada para a entrada 5 + 2. Ao aplicar a regra semântica a esse nó, *T_1 .val* possui o valor 5, proveniente do filho à esquerda, e *F.val* o valor 2, do filho à direita. Por conseguinte, *T.val* adquire o valor 7 a esse nó.

Figura 2-12 - ÁRVORE GRAMATICAL ANOTADA PARA 5+2



Fonte: Baseado em (Aho, 1995, p. 130)

2.3.4.1.2 ATRIBUTOS HERDADOS

Um atributo herdado é aquele cujo valor a um nó de uma árvore gramatical é definido em termos do 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.

O Quadro 2-20 demonstra o uso de um atributo herdado para distribuir as informações de tipo para os vários identificadores numa declaração. A regra semântica *L.in := T.tipo*, associada à produção *D ::= T L*, faz o atributo herdado *L.in* igual ao tipo na declaração. As

regras então propagam esse tipo pela árvore gramatical abaixo, usando o atributo herdado *L.in*. As regras associadas às produções para *L* chamam o procedimento *incluir_tipo* para incluir o tipo de cada identificador na sua entrada respectiva na tabela de símbolos (apontado pelo atributo *entrada*) (Aho, 1995, p. 122).

Apesar de ser sempre possível se reescrever uma definição dirigida pela sintaxe de forma a se usar somente atributos sintetizados, estas definições dirigidas pela sintaxe com atributos herdados são freqüentemente mais naturais (Aho, 1995, p. 121).

Quadro 2-20 - DEFINIÇÃO DIRIGIDA PELA SINTAXE TENDO *L.in* COMO ATRIBUTO HERDADO

PRODUÇÃO	REGRAS SEMÂNTICAS
D ::= T L	*L.in := T.tipo*
T ::= **int**	*L.tipo := inteiro*
T ::= **real**	*L.tipo := real*
L ::= L_1 ,**id**	*L_1.in := L.in*
	incluir_tipo (**id**, *entrada*, *L.in*)
L ::= **id**	*incluir_tipo* (**id**, *entrada*, *L.in*)

Fonte: (Aho, 1995, p. 122)

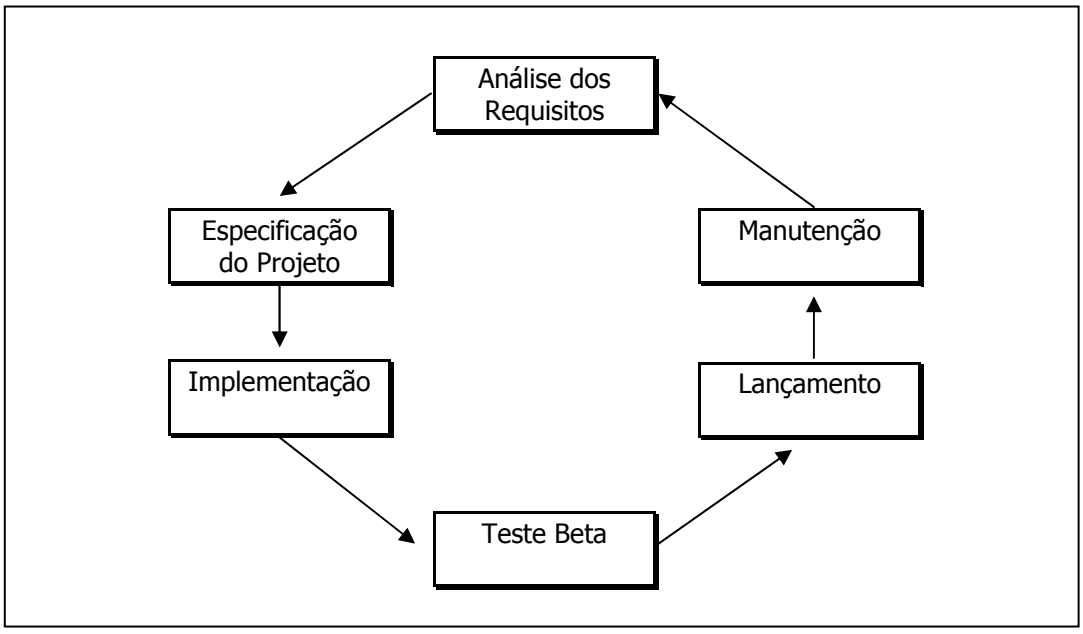
3 DESENVOLVIMENTO DO TRABALHO

Neste capítulo serão descritas as atividades desempenhadas em cada fase do modelo de desenvolvimento de software adotado para construção do trabalho. Também serão relacionados e discutidos os resultados obtidos a partir do mesmo.

O modelo de desenvolvimento de software adotado pela empresa Mult Sistemas Ltda, e por conseguinte utilizado no presente trabalho, denomina-se modelo cíclico. Segundo Fisher (1990, p. 9), o modelo cíclico consiste em uma forma mais instrutiva de imaginar o desenvolvimento de software do que o processo linear ou tradicional, pois assume o software como um processo contínuo, que geralmente necessita de uma atividade de manutenção constante.

O modelo cíclico é composto pelas fases ilustradas na Figura 3-1. O detalhamento das atividades em cada fase será apresentado a seguir.

Figura 3-1 –MODELO CÍCLICO DE DESENVOLVIMENTO DE SOFTWARE



Fonte: (Fisher, 1990, p. 9)

3.1 ANÁLISE DOS REQUISITOS

Nesta seção serão apresentados os principais requisitos do trabalho, através de uma contextualização do cenário que originou seu desenvolvimento, bem como as funcionalidades propostas e características gerais.

O presente trabalho caracteriza-se em um estágio supervisionado desenvolvido na empresa Mult Sistemas Ltda, onde foi identificada a necessidade de sua concepção. Portanto, desde o início de sua construção o protótipo objetivou tornar-se efetivamente um produto final de auxílio ao processo de desenvolvimento, aplicado ao ambiente de produção da referida empresa. O nome comercial atribuído a esta ferramenta é *Struct*, pelo fato de manipular uma diversidade de estruturas de dados armazenadas em seu repositório.

A fase de análise de requisitos foi realizada através de reuniões e entrevistas com os usuários finais, representados pelos técnicos do departamento de desenvolvimento da empresa. Dentre estes usuários, o orientador do trabalho, Sr. Ricardo de Freitas Becker, exerceu significativa participação neste processo.

A identificação da necessidade de desenvolvimento do software surgiu a partir de um planejamento realizado pela empresa alvo do estágio que visava a evolução tecnológica de uma linha de produtos legados. Esta linha encontra-se codificada na linguagem de programação Borland Pascal 6.0, com a aplicação híbrida das técnicas de orientação a objetos e programação estruturada, utilizando o gerenciador de arquivos Turbo Power BTree-Filer para suportar a persistência dos dados.

A estratégia de evolução tecnológica definida pela empresa previa a criação de um novo ambiente de desenvolvimento que incorporasse as seguintes características:

a) emprego de diagramas da linguagem UML (*Unified Modeling Language*) para especificação dos sistemas de informação;
b) construção de uma biblioteca de classes em conformidade com a arquitetura de três camadas, apresentada por Ambler (1998, p. 99);
c) generalização da camada de persistência no sentido de suportar aplicações de bancos de dados e gerenciadores de arquivos, através de uma estrutura de independência da origem de dados;

d) utilização de ferramentas CASE que, além de atender aos propósitos de análise, permitam a geração de código-fonte com especializações de classes da biblioteca a partir das definições de seus repositórios;
e) desenvolvimento de um software utilitário que permitisse a engenharia reversa de determinados fragmentos de código-fonte dos sistemas legados, visando aproveitar definições conceituais.

A referida empresa adotava a ferramenta CASE CA ERwin para modelagem de dados destinada a alguns produtos especiais, ou seja, independentes da linha de produtos legados em questão. Os diagramas entidade x relacionamento confeccionados através do ERwin originavam *scripts* SQL para manutenção dos bancos de dados. Tinha-se conhecimento de que o referido CASE fornecia uma alternativa de acesso completo ao seu repositório de dados.

Durante o planejamento do novo ambiente, questionou-se a possibilidade de implementar uma ferramenta que extraísse definições comuns a partir dos arquivos de código-fonte dos sistemas legados. Esta engenharia reversa poderia, por exemplo, obter a estrutura (*layout*) dos arquivos e alimentar o repositório de uma ferramenta de modelagem de dados com estas definições, o que tornaria a evolução dos sistemas para o ambiente de banco de dados mais natural. Definiu-se então a primeira aplicabilidade da ferramenta objeto deste trabalho, que consiste na alimentação do repositório de dados do CA ERwin, a partir de definições extraídas de código-fonte.

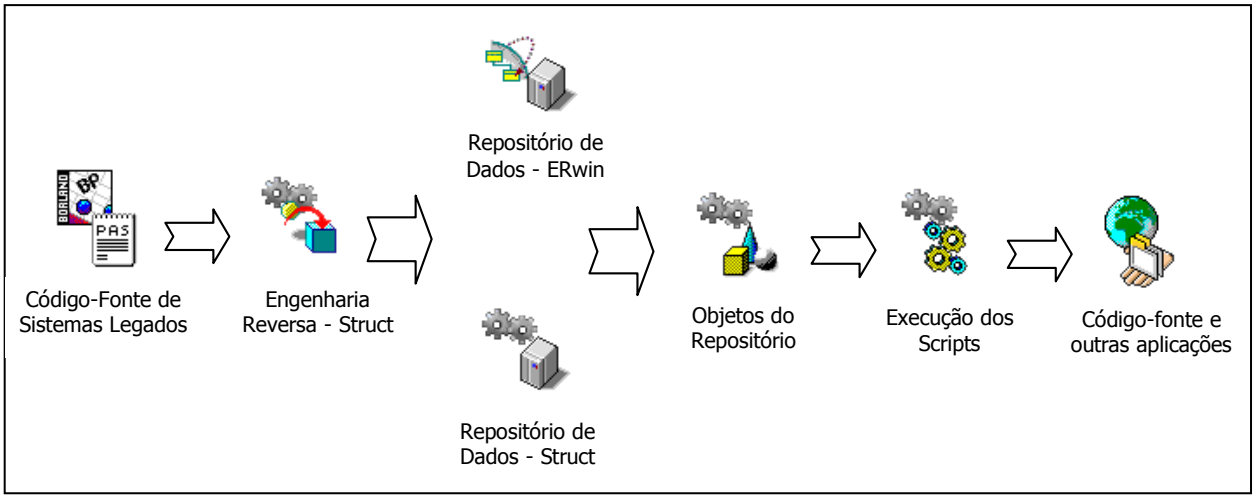
Em seguida, definiu-se a criação de um repositório de dados complementar ao repositório da ferramenta CA ERwin, que conteria características adicionais, relacionadas aos objetos pré-existentes no modelo de dados, bem como, novos objetos que possam incorporar-se aos recursos genéricos publicados na biblioteca de classes. Um coordenador de repositório, conceituado por Martin (1994, p. 9), manipularia os objetos de forma a obedecer às regras estruturais assumidas no novo ambiente de desenvolvimento.

O repositório original da ferramenta CA ERwin, aliado ao repositório complementar, armazenaria um conjunto de especificações de objetos disponibilizados para utilização por um gerador de código-fonte. Para flexibilizar o potencial e facilitar a manutenção, este gerador de código interpretaria arquivos de *script*, escritos em uma linguagem de alto-nível, portanto com alto grau de legibilidade.

Como fruto das discussões com os usuários finais, surgiu ainda a requisição de manter um único modelo de dados para a geração de sistemas legados e sistemas novos, visando eliminar prováveis conflitos e incompatibilidades geradas pelos processos de desenvolvimento paralelos. Atendendo a esta definição a ferramenta especificou mecanismos de independência entre tipos de dados, pois o mesmo repositório serviria como base para geração de código-fonte, tanto para sistemas de bancos de dados, como para gerenciadores de arquivos. A seqüência de processos é ilustrada através do esquema funcional da ferramenta apresentado na Figura 3-2.

Portanto, a aplicação inicial da ferramenta consistiria na geração automática de classes de persistência correspondentes às entidades contidas no repositório. Vislumbrou-se a possibilidade de geração de fontes estruturais da camada de negócio e fontes básicos da camada de apresentação. A partir da definição de características adicionais às entidades, atributos, índices e relacionamentos, e a criação de *scripts* específicos, as possibilidades de geração de fontes seriam inúmeras.

Figura 3-2 – ESQUEMA FUNCIONAL DA FERRAMENTA *STRUCT*



3.2 ESPECIFICAÇÃO DO PROJETO

Esta seção descreve as atividades desempenhadas na fase de especificação do projeto, apresentando os modelos, diagramas e estruturas de dados que constituem na representação lógica do trabalho.

Para especificação das classes que compõem a ferramenta *Struct* foi empregada a linguagem de modelagem UML, cujos diagramas foram confeccionados através da ferramenta CASE Rational Rose 2000.

O ambiente de desenvolvimento construído pela Mult Sistemas Ltda determina convenções para nomenclatura de recursos, sobre as quais tornam-se necessários alguns esclarecimentos. O nome das classes é formado pela seguinte concatenação:

a) o primeiro caractere indica o tipo de recurso, onde as letras "*u*" e "*T*" correspondem a *unit* e *type*, respectivamente;
b) o próximo caractere identifica a camada onde o recurso está embutido, utilizando as letras "*p*", "*l*", "*a*", representando respectivamente as camadas de persistência, lógica e armazenamento;
c) as próximas três letras representam uma sigla atribuída à classe ancestral, sendo que a sigla "*Cls*" é utilizada para denotar qualquer classe externa ao ambiente; e
d) o nome literal da classe altamente significativo.

A notação prevê ainda que os nomes de métodos iniciem com um verbo no infinitivo e os nomes de atributos sejam prefixados por uma letra que identifique o tipo de dado correspondente.

Como pôde ser observado no detalhe da nomenclatura, o ambiente de desenvolvimento utilizado emprega a estruturação das classes em três camadas conforme a arquitetura tipo-classe exposta por Ambler (1998, p. 88).

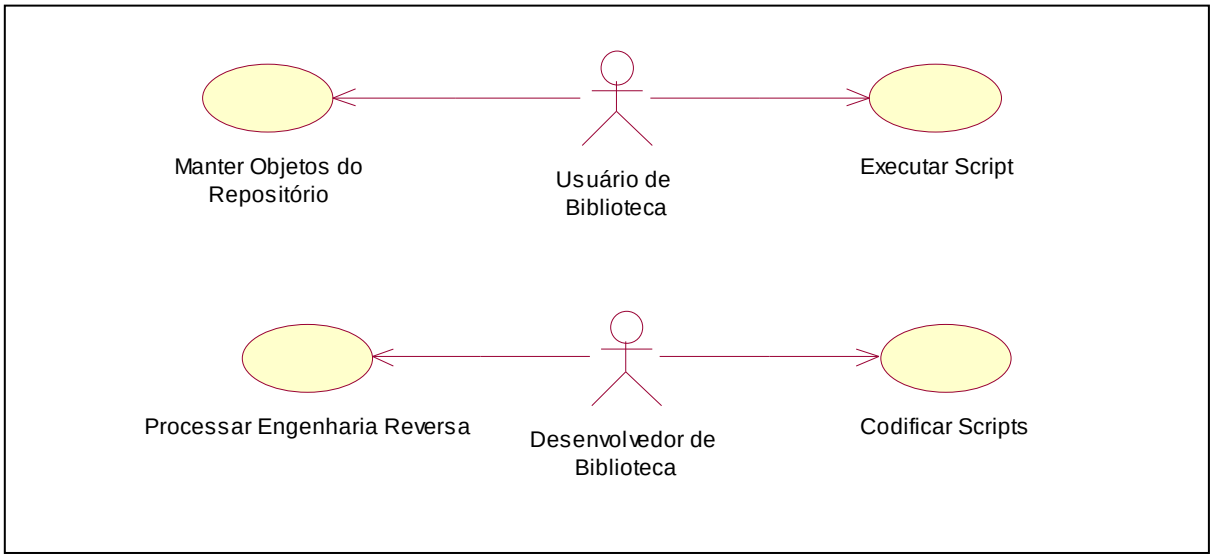
A Figura 3-3 apresenta os casos de uso identificados a partir da funcionalidade da ferramenta *Struct*. Interagem com o software dois atores: o usuário de biblioteca, que mantém os objetos do repositório e executa *scripts*; e o desenvolvedor de biblioteca, que processa a engenharia reversa e codifica os *scripts*.

O desenvolvedor de biblioteca interage com o sistema para execução da rotina de engenharia reversa a partir do código-fonte de sistemas legados. Esta rotina é executada uma única vez para cada sistema visando alimentar as definições do repositório de dados da ferramenta. Para tanto, o referido ator deve conhecer a estrutura de organização dos arquivos fonte dos sistemas legados e informar corretamente os parâmetros solicitados. Ao final do

processo, o referido ator deve analisar o arquivo de *log* resultante, visando identificar os objetos gerados, bem como detectar possíveis problemas ocorridos durante o processo. O desenvolvedor de biblioteca deve ainda escrever ou codificar os arquivos de *script* para propósitos gerais, como geração de código, documentação, entre outros. Para tanto, este ator deve apresentar amplo domínio da linguagem especificada pelo trabalho.

Cabe ao usuário da biblioteca a inclusão, alteração e exclusão dos objetos disponíveis na ferramenta, tais como entidades, atributos, índices e relacionamentos. Os atributos destes objetos são acessados através de comandos da linguagem codificados nos arquivos de *scripts*. Portanto, devem ser preenchidos corretamente para que o arquivo de saída gerado pela execução do *script* seja válido. Esta execução é disparada pelo próprio ator usuário de biblioteca, que deve atentar para possíveis críticas ocorridas durante o processo, como campos obrigatórios não informados, conteúdos duplicados, entre outros.

Figura 3-3 – CASOS DE USO DA FERRAMENTA *STRUCT*

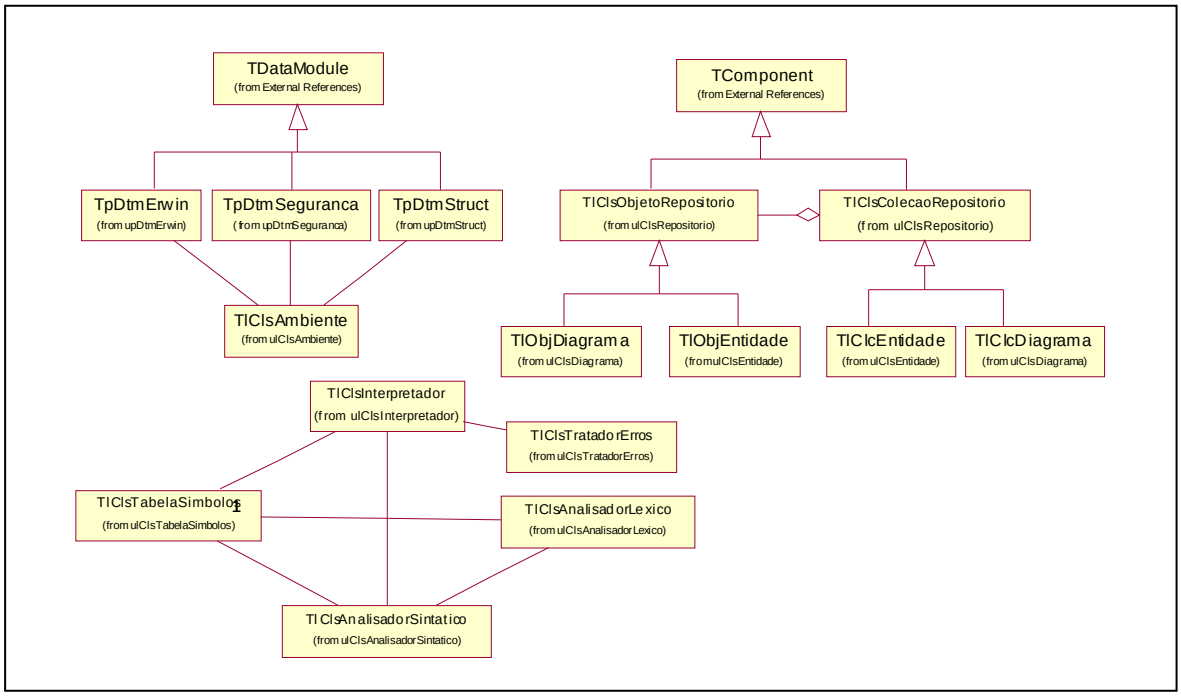


A Figura 3-4 apresenta um diagrama com as principais classes especificadas na ferramenta. A classe *TlClsAmbiente* é responsável pelas características e funcionalidades do ambiente da aplicação, como por exemplo, instanciação das classes de persistência, leitura e gravação das preferências globais e controle dos projetos abertos.

O ambiente de desenvolvimento Borland Delphi, utilizado na implementação do trabalho, suporta aplicações baseadas em uma biblioteca de classes de componentes visuais, denominada VCL (*Visual Component Library*). Em algumas situações, portanto, faz-se

necessária uma breve descrição das classes da VCL utilizadas, visando facilitar o entendimento do modelo.

Figura 3-4 - DIAGRAMA DE CLASSES PRINCIPAIS (VISÃO GLOBAL)



A especificação do protótipo será abordada em cinco etapas: repositório de dados do CA ERwin, repositório de dados complementar, coordenador de repositório, engenharia reversa e interpretador, que serão apresentadas nas seções seguintes.

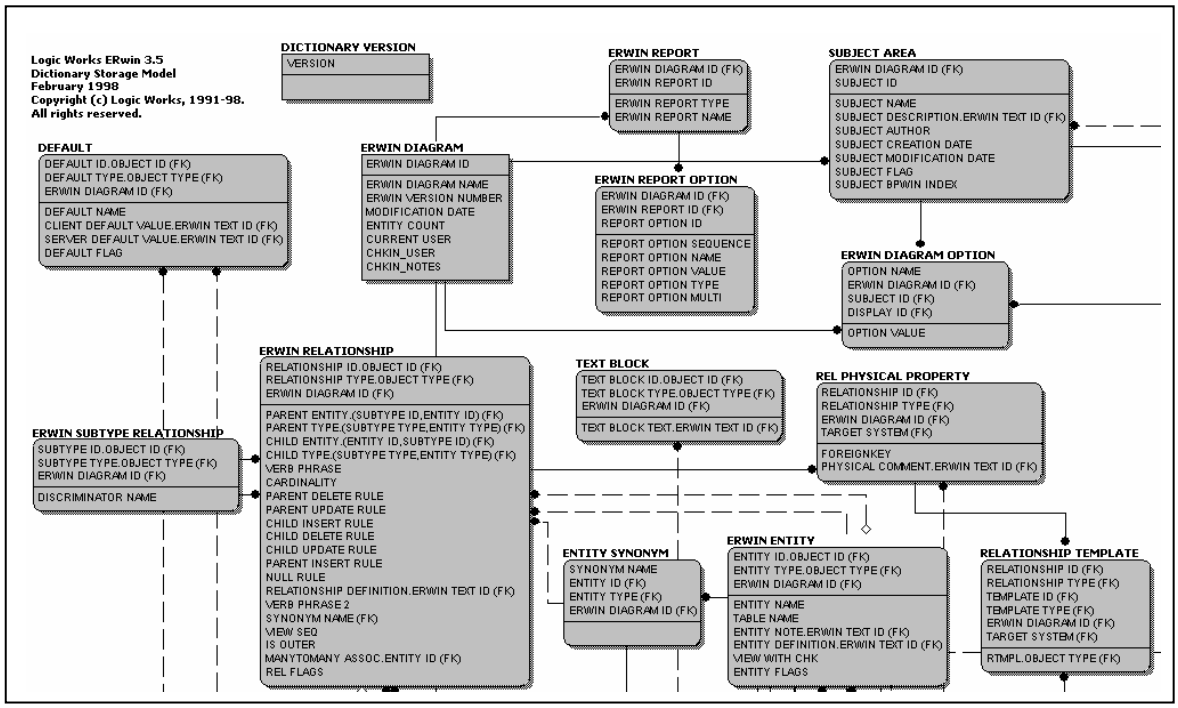
3.2.1 REPOSITÓRIO DE DADOS DO CA ERWIN

A ferramenta CASE CA ERwin armazena as definições de seus diagramas entidade x relacionamento de duas formas: em arquivos binários com formato não divulgado pelo fabricante, ou através de um mecanismo que transfere as definições para um banco de dados, cuja forma explora-se no trabalho.

O termo repositório de dados é utilizado no trabalho para referir-se tanto ao dicionário de dados gerado pela ferramenta CA ERwin, como ao conjunto de tabelas que armazenam as definições complementares criadas pela ferramenta *Struct*, pois ambos enquadram-se no conceito de repositório de dados apresentado por Gane (1990).

Um modelo de dados especial, denominado *ERwin Dictionary Metamodel*, ou Meta-modelo do Dicionário do ERwin é fornecido pelo fabricante para viabilizar a geração física do repositório em um banco de dados. Este modelo de dados, cuja visão parcial é apresentada na Figura 3-5 contém as entidades correspondentes aos diversos tipos de objetos manipulados pela ferramenta, os atributos destes objetos, bem como os relacionamentos entre eles. Por questões de otimização de performance de acesso físico ao repositório, os relacionamentos não são gerados fisicamente.

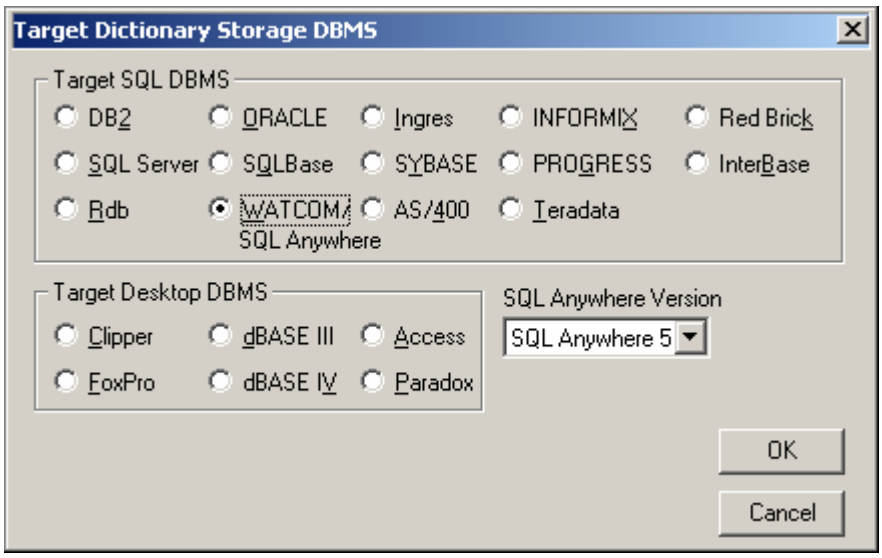
Figura 3-5 –METAMODELO DO REPOSITÓRIO DO ERWIN (VISÃO PARCIAL)



Com base neste meta-modelo um banco de dados é gerado através do *script* DDL originado pela ferramenta. A partir deste processo, o repositório de dados encontra-se apto a armazenar as definições de quaisquer diagramas ER, através das opções de *check-in* e *check-out* (conforme diálogo da Figura 3-7), as quais correspondem à atualização do repositório em função do diagrama e geração do diagrama em função do repositório, respectivamente.

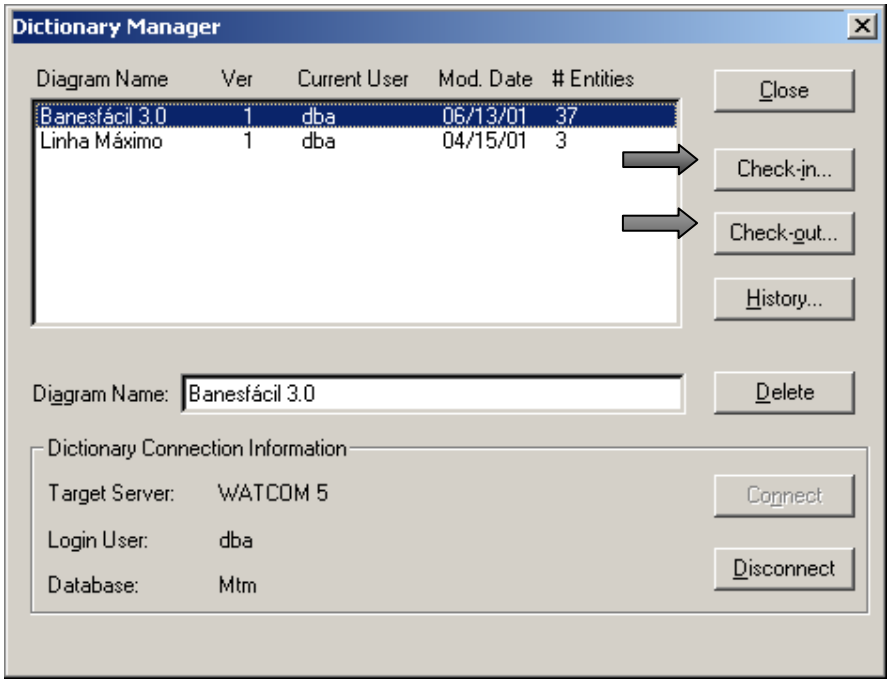
O repositório pode ser gerado para diversos bancos de dados, bem como para alguns gerenciadores de arquivos, conforme o diálogo da Figura 3-6. O destino do repositório adotado pela ferramenta *Struct* é o SGBD (Sistema Gerenciador de Banco de Dados) Sybase SQL Anywhere versão 5.0.

Figura 3-6 – DIÁLOGO DE SELEÇÃO DO DESTINO PARA O REPOSITÓRIO DE DADOS



Após a seleção do banco de dados destino, a partir do diálogo da Figura 3-6, o ERwin requer a conexão com o banco de dados, estabelecida através da seleção de uma origem de dados de um *driver* ODBC (*Open Database Conectivity*). Em seguida, a janela de diálogo da Figura 3-7 é apresentada.

Figura 3-7 – JANELA DE DIÁLOGO DO GERENCIADOR DE DICIONÁRIO (*DICTIONARY MANAGER*)



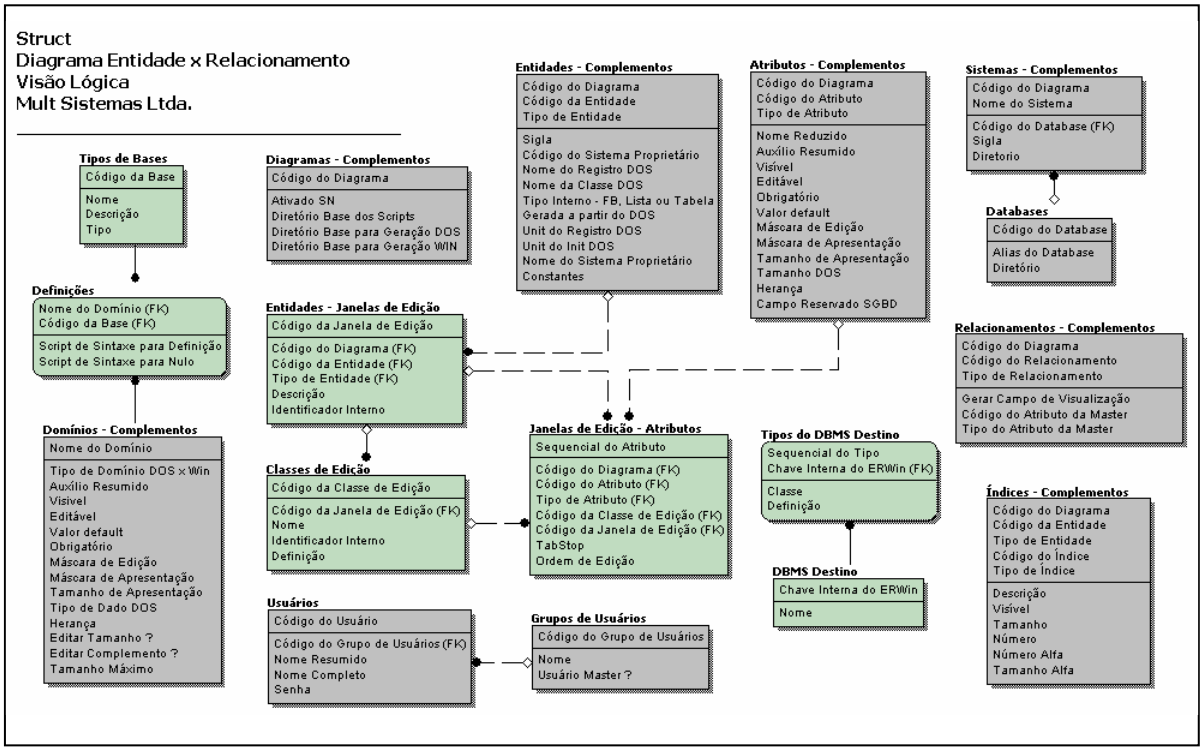
A partir da seleção da opção "*check-in*", o banco de dados selecionado é alimentado com as definições do diagrama corrente.

3.2.2 REPOSITÓRIO DE DADOS COMPLEMENTAR

Como fruto da especificação de requisitos, a ferramenta *Struct* mantém um repositório de dados complementar ao repositório do CA ERwin. Este repositório incorpora características adicionais aos objetos originais, além de atender a requisitos independentes do repositório original. Portanto, de um modo geral, a funcionalidade do código-fonte a ser gerado pela ferramenta aumenta em função da quantidade de definições agregadas ao repositório de dados complementar.

A Figura 3-8 apresenta o modelo de dados do repositório complementar. Não são estabelecidos relacionamentos explícitos com as entidades do repositório de dados do ERwin, pois este não permite que os relacionamentos sejam incluídos no banco de dados do repositório.

Figura 3-8 – DER DO REPOSITÓRIO COMPLEMENTAR DO *STRUCT* (VISÃO LÓGICA)



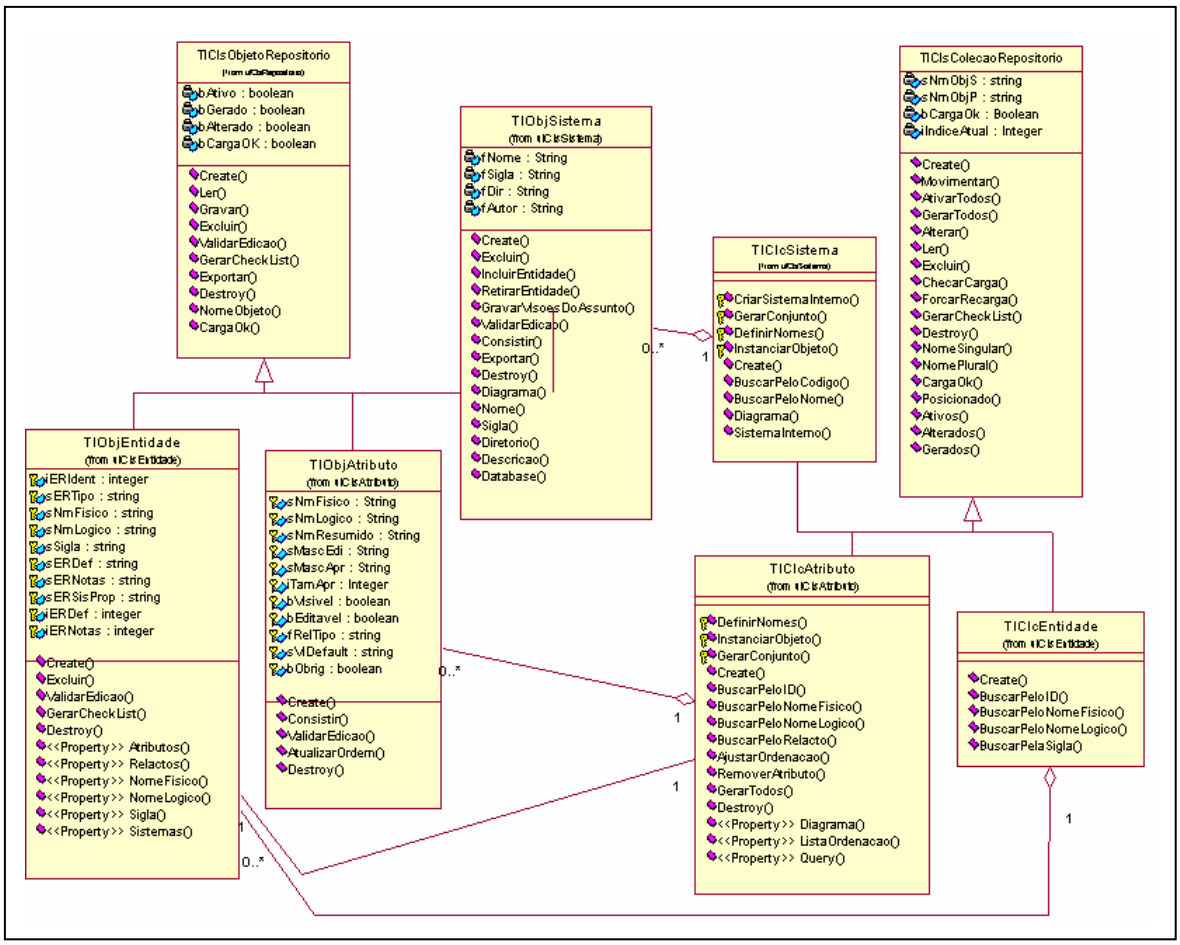
Contudo, os relacionamentos são mantidos através da inclusão dos campos da chave primária das tabelas do ERwin nas tabelas complementares correspondentes. Como por exemplo, a tabela de nome lógico "*Entidades – Complementos*", identificada na Figura 3-8, copia os campos da chave primária da tabela "*ERwin Entity*" visualizada na Figura 3-5.

3.2.3 COORDENADOR DE REPOSITÓRIO

Martin (1995, p. 9) conceitua que o software coordenador do repositório aplica métodos aos dados do repositório para assegurar que os dados e as representações CASE dos dados sejam consistentes e íntegras.

O presente trabalho implementa um mecanismo coordenador de repositório através da estrutura de classes dispostas no diagrama da Figura 3-9.

Figura 3-9 - CLASSES DO COORDENADOR DE REPOSITÓRIO DO *STRUCT*



Cada componente manipulado pelo repositório do *Struct* é construído a partir do acesso de leitura a uma ou mais tabelas, sendo modelado como uma classe lógica, que acessa várias tabelas para obter e persistir suas definições.

O componente pode ser tratado como uma instância, quando manipulado individualmente, ou como uma coleção, quando manipulado em conjunto. Para distinguir os dois tratamentos lógicos, foram modeladas classes de objeto e de coleções de objetos, descendentes das classes *TlObjRepositorio* e *TlClcRepositorio*, respectivamente.

A camada de persistência para os componentes do repositório do ERwin é assistida pela classe *TpDtmErwin*. Esta classe é uma especialização de uma classe da VCL, denominada *TDataModule*, a qual possui a propriedade de agregar objetos de acesso a origem de dados, servindo como uma espécie de depósito de objetos de persistência.

3.2.4 ENGENHARIA REVERSA

A engenharia reversa aplicada pela ferramenta *Struct* tem como propósito alimentar o repositório de dados a partir de construções contidas em arquivos código-fonte escritos na linguagem Pascal. Como identificado na fase de análise de requisitos, a aplicação prática da engenharia reversa consiste em extrair informações relacionadas aos arquivos de dados manipulados por uma geração de produtos legados.

Para controle da persistência dos dados, os produtos legados utilizam um gerenciador de arquivos produzido pela empresa Turbo Power, denominado BTree-Filer. Este gerenciador implementa uma classe abstrata denominada *FileBlock*, que provê métodos para acesso ao arquivo de dados.

O processo de engenharia identifica no código-fonte as estruturas relevantes, dispostas a seguir por ordem de avaliação:

a) procedimento de instanciação da classe *fileblock* (Quadro 3-1);
b) estrutura de dados lógica que corresponde ao formato do registro do arquivo (Quadro 3-2);
c) declaração da classe descendente de *fileblock* (Quadro 3-3);
d) implementação do método de composição das chaves dos arquivos de índice (Quadro 3-4).

Quadro 3-1 – PROCEDIMENTO DE INSTANCIAÇÃO DA CLASSE *FILEBLOCK*

```
procedure IniciaClientes;
begin
  New (CLI, Init (ARQCLI, 'Clientes', 'CLIENTES.DTA', PathRaiz,
      'CLI', @RegCli^, SizeOf (CliReg)));
  if (CLI <> nil) then
    with CLI^ do
    begin
      FillChar (RegCli^,SizeOf(CliReg),0);
      NumKeys  := 2;
      IID[1].KeyL      := 4;     {código do cliente}
      IID[1].AllowDupK := False;
      IID[2].KeyL      := 50;    {nome do cliente}
      IID[2].AllowDupK := False;
    end
end;
```

Quadro 3-2 – ESTRUTURA DE DADOS LÓGICA DE UM *FILEBLOCK*

```
CliReg = record
  Status                : longint;     {Campo Interno}
  CdCli                 : longint;     {Código do Cliente}
  Nome                  : string[50];
  Endereco              : string[50];
  ...
  Tipo                  : byte;        {0=Física, 1=Jurídica}
end;
```

Quadro 3-3 – DECLARAÇÃO DA CLASSE DESCENDENTE DE *FILEBLOCK*

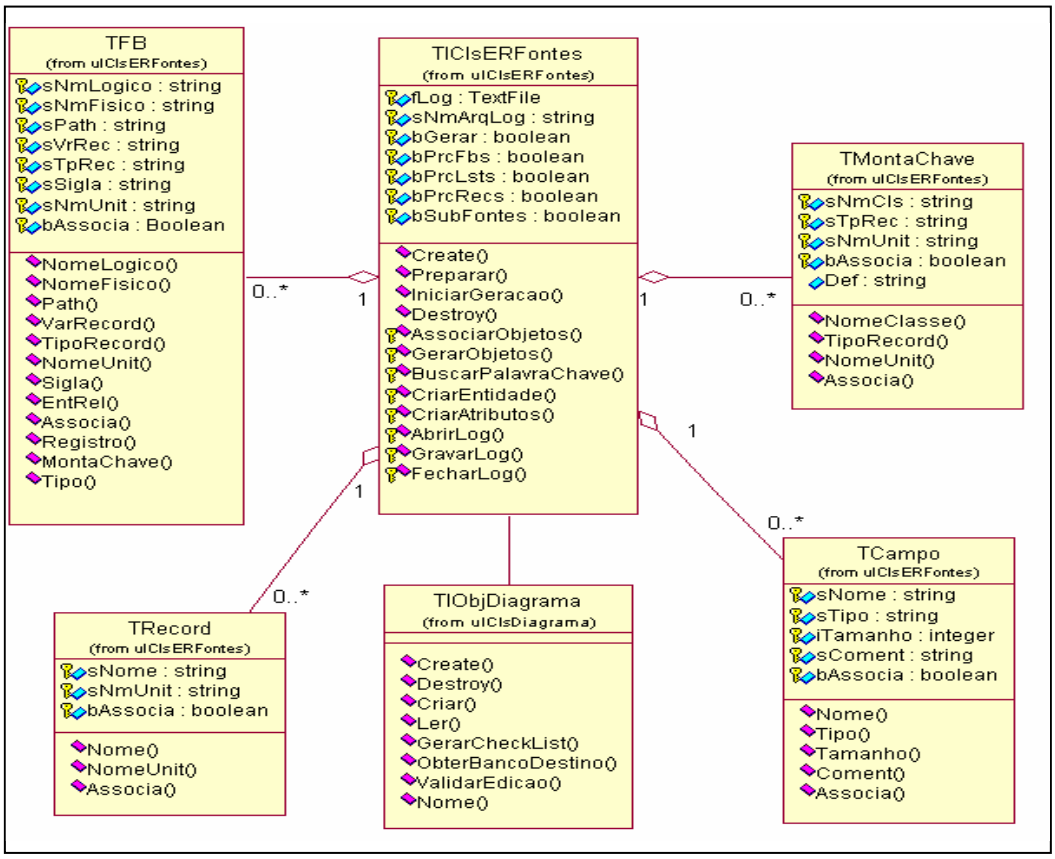
```
CliFileBlockPtr = ^CliFileBlock;
CliFileBlock = object (FileBlock)
  function  MontaChave (var Rec; KeyNr, PassNr : Word) :
                    IsamKeyStr; virtual oiFileBlock_MontaChave;
  ...
end;
```

Quadro 3-4 –MÉTODO DE COMPOSIÇÃO DE ÍNDICES

```
function CliFileBlock.MontaChave (var Rec; KeyNr, PassNr :
                                  Word) : IsamKeyStr;
begin
  MontaChave := '';
  with CliReg (Rec) do
    case KeyNr of
      1 : if PassNr = 1 then
            MontaChave := LongToSortString (CodCli);
      2 : if PassNr = 1 then
            MontaChave := StUpCase (CodCliAlt);
    end;
end;
```

O mecanismo de engenharia reversa é composto pelas classes dispostas no diagrama da Figura 3-10. As classes *TFB*, *TRecord*, *TCampo* e *TMontaChave* correspondem aos componentes *fileblock*, registro lógico, campo e índice, respectivamente. A associação destes elementos permite a geração das estruturas de um modelo de dados, onde *fileblock* corresponde a uma entidade com o conjunto de campos representados por uma lista de instâncias de *TCampo* e um conjunto de chaves alternativas originadas a partir de uma lista de instâncias da classe *TMontaChave*. Os tipos de dados associados aos campos da estrutura *TRecord* implicitamente originam os domínios do banco de dados.

Figura 3-10 – DIAGRAMA DE CLASSES DA ENGENHARIA REVERSA



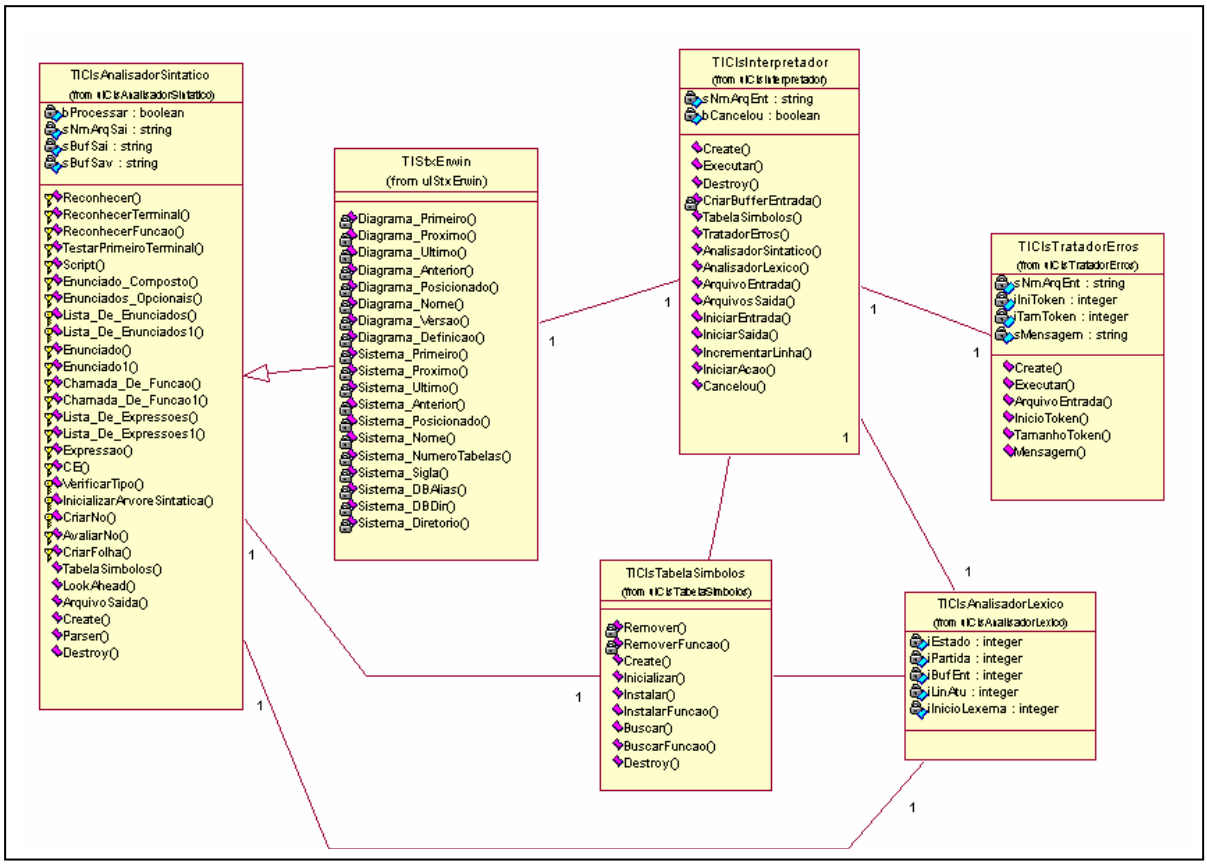
Dessa forma, é realizada a tradução da estrutura de elementos de um gerenciador de arquivos em um modelo de dados. A partir deste modelo a ferramenta CASE ERwin permite a geração *scripts* SQL para diversos sistemas gerenciadores de bancos de dados.

Os componentes são identificados no código-fonte de forma independente num primeiro momento, sendo associados em seguida. Para verificar possíveis pendências, entre outros controles, um arquivo de *log* é alimentado durante o processo.

3.2.5 INTERPRETADOR

A geração de código-fonte proposta pelo trabalho tornou-se possível através da concepção de um interpretador para uma linguagem de comandos. Inicialmente será apresentada uma visão global sobre as classes que compõem o interpretador, ilustradas no diagrama da Figura 3-11.

Figura 3-11 – DIAGRAMA DE CLASSES DO INTERPRETADOR



Basicamente, a classe do interpretador prepara os componentes necessários ao processo de interpretação. Através de uma chamada ao analisador sintático, esta classe desencadeia efetivamente o processo de interpretação. Cabe a classe do interpretador inclusive, a tarefa de comunicar-se com a camada de apresentação, notificando ao usuário os eventos ocorridos durante o processo. O interpretador também redireciona qualquer ocorrência de exceção ao tratador de erros.

As fases do processo de compilação empregadas na construção do interpretador em questão correspondem à análise léxica, análise sintática e análise semântica, as quais interagem com a tabela de símbolos e com o tratador de erros.

A classe do analisador sintático, ou *parser*, interage diretamente com o analisador léxico, compartilhando algumas estruturas de dados, descritas no decorrer do trabalho. O *parser* interage ainda com a classe da tabela de símbolos e possui ações semânticas implementadas sob a forma de métodos. Estudou-se a possibilidade de implementar uma classe distinta também para o analisador semântico, porém esta decisão tornaria o processo excessivamente burocrático, visto que, a gramática de atributos cria uma relação próxima entre a produção sintática e a regra semântica.

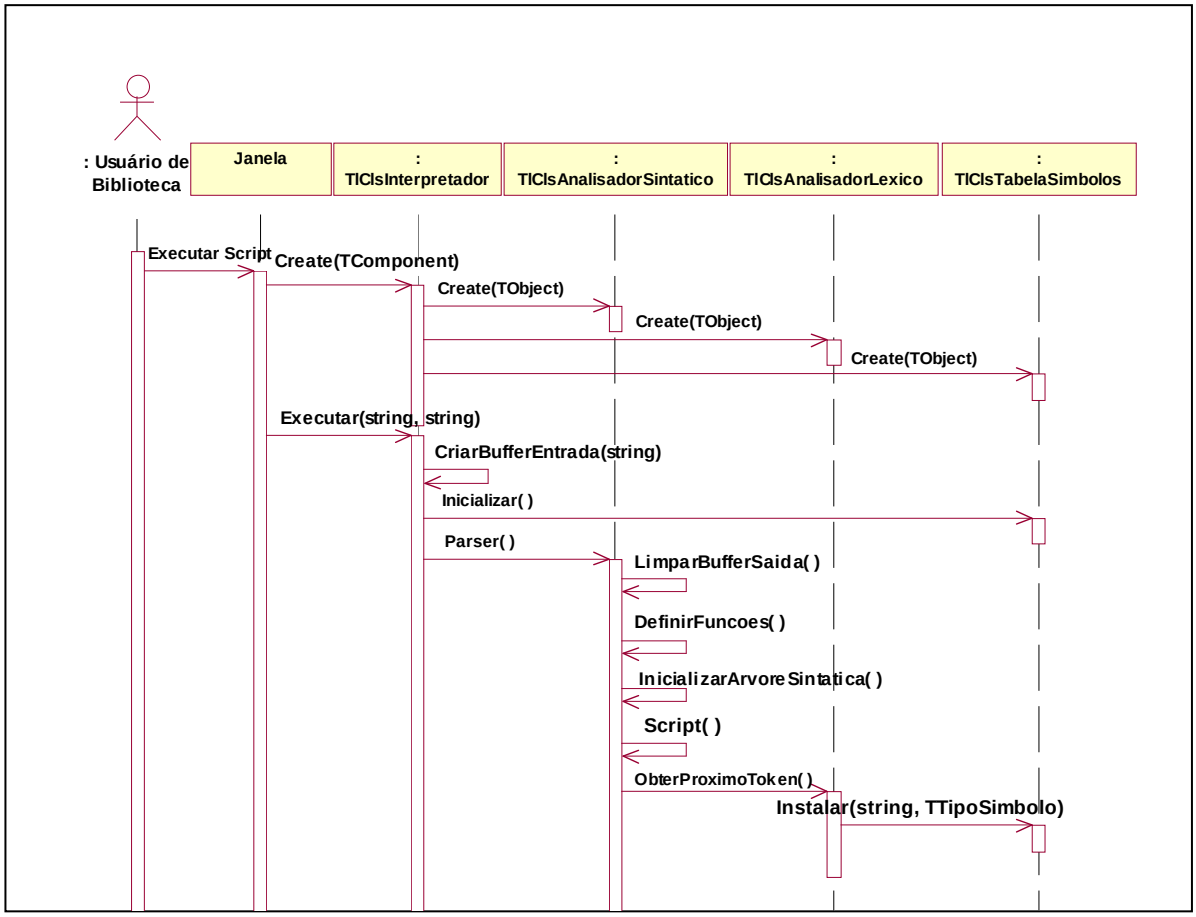
A interação entre as classes pode ser observada através do diagrama de seqüência, apresentado na Figura 3-12. A classe do analisador léxico fornece ao analisador sintático a estrutura de dados correspondente ao próximo *token* obtido a partir do *buffer* de entrada, o qual é alimentado inicialmente pela classe do interpretador. A classe do analisador léxico oferece também um mecanismo que permite marcar “rótulos”, utilizados em comandos que alteram o fluxo de controle. Este recurso será descrito em detalhes na seção 3.3.

A classe da tabela de símbolos provê métodos de instalação e manipulação de símbolos na referida tabela. Cabe ressaltar que o interpretador objeto de discussão desta seção incorpora o tratamento de uma tabela de símbolos com fins meramente didáticos, pois a gramática da linguagem não prevê a necessária utilização desta tabela. Encontra-se fora do escopo deste interpretador a declaração de variáveis e procedimentos, ainda que o mesmo implemente um artifício que permite simular a existência de variáveis internas.

A classe do tratador de erros é notificada na ocorrência de qualquer erro durante o processo de interpretação. Conceitualmente, caso haja possibilidade de tratamento, ou recuperação do erro, os controles necessários para tais fins devem ser embutidos nesta classe. Para o interpretador em questão, o tratamento de erro resume-se ao posicionamento no início do lexema do correspondente *token* que originou o erro, a partir do componente visual para edição do código-fonte.

Após uma descrição geral sobre as características de especificação relacionadas às classes que compõem o módulo interpretador, convém descrever as técnicas que resultaram na linguagem a ser interpretada.

Figura 3-12 – DIAGRAMA DE SEQUÊNCIA DE "EXECUTAR SCRIPT"



Segundo Neto (1987, p. 5) é essencial uma descrição formal das linguagens de programação de maneira completa e isenta de ambigüidades. Seguindo esta premissa, empregou-se a metalinguagem BNF para especificação da gramática que define a sintaxe da linguagem, e, para a descrição de ações semânticas, aplicou-se o método de gramática de atributos.

A definição da gramática resultou da adaptação de um subconjunto da linguagem Pascal, obtido em Aho (1995, p. 326), que especifica as regras estruturais básicas da linguagem, como blocos, enunciados, funções e expressões. Porém, tal subconjunto não apresenta originalmente os requisitos para utilização de um analisador sintático preditivo, proposto pelo trabalho. Para tanto, foram aplicados os seguintes procedimentos a partir das produções originais:

a) eliminada a recursividade à esquerda;

b) incorporado à gramática um analisador sintático de precedência de operadores. A gramática deste analisador é baseada em Silva (2001) e encontra-se devidamente ajustada para análise sintática preditiva, cujas produções são apresentadas no Quadro 3-6;
c) eliminada a ambigüidade do “else-vazio”, através da técnica descrita por Aho (1995, p. 85);
d) aplicada a técnica de fatoração à esquerda (Aho, 1995, p. 79).

Para facilitar a referência, a linguagem especificada pela gramática apresentada nos Quadros 3-5 e 3-6 denominar-se-á LCS (Linguagem de Comandos *Struct*).

Quadro 3-5 – GRAMÁTICA DA LINGUAGEM LCS

(1)	*script*	→	**Script id;** *enunciado_composto* **.**
(2)	*enunciado_composto*	→	**Inicio** *enunciados_opcionais* **Fim**
(3)	*enunciados_opcionais*	→	*lista_de_enunciados* | ∈
(4)	*lista_de_enunciados*	→	*enunciado* *lista_de_enunciados'*
(5)	*lista_de_enunciados'*	→	; *enunciado* *lista_de_enunciados'* | ∈
(6)	*enunciado*	→	*enunciado_composto* | **Se** *expressao* **Entao** *enunciado enunciado'* | **Enquanto** *expressao* **Faca** *enunciado* | *chamada_de_funcao*
(7)	*enunciado'*	→	**Senao** *enunciado* | ∈
(8)	*chamada_de_funcao*	→	**id** *chamada_de_funcao'*
(9)	*chamada_de_funcao'*	→	**(** *lista_de_expressoes* **)** | ∈
(10)	*lista_de_expressoes*	→	*expressao* *lista_de_expressoes'*
(11)	*lista_de_expressoes'*	→	, *expressao* *lista_de_expressoes'* |∈
(12)	*expressao*	→	*CE*

Fonte: Baseado em (Aho, 1995, p. 327)

Quadro 3-6 – DEFINIÇÃO DIRIGIDA PELA SINTAXE PARA ANÁLISE DE EXPRESSÕES

(13)	*CE*	→	*SE* *RE*	*{ RE.i* *{ CE.nptr*	**:=** **:=**	*SE.nptr }* *RE.s }*
(14)	*RE*	→	**=** *SE*	*{ RE.s*	**:=**	*CriarNodo ('=', RE.i, SE.nptr) }*
(15)	*RE*	→	**<>** *SE*	*{ RE.s*	:=	*CriarNodo ('<>', RE.i, SE.nptr) }*
(16)	*RE*	→	**<** *SE*	*{ RE.s*	:=	*CriarNodo ('<', RE.i, SE.nptr) }*
(17)	*RE*	→	**>** *SE*	*{ RE.s*	:=	*CriarNodo ('>', RE.i, SE.nptr) }*
(18)	*RE*	→	**<=** *SE*	*{ RE.s*	:=	*CriarNodo ('<=', RE.i, SE.nptr) }*
(19)	*RE*	→	**>=** *SE*	*{ RE.s*	:=	*CriarNodo ('>=', RE.i, SE.nptr) }*
(20)	*RE*	→	∈	*{ RE.s*	:=	*RE.i }*
(21)	*SE*	→	*T* *SE1*	*{ SE1.i* *{ SE.nptr*	:= :=	*T.nptr }* *SE1.s }*
(22)	*SE1*	→	**+** *T* *SE11*	*{SE11.i* *{ SE1.s*	:= :=	*CriarNodo ('+', SE1.i, T.nptr) }* *SE11.s }*
(23)	*SE1*	→	**-** *T* *SE11*	*{SE11.i* *{ SE1.s*	:= :=	*CriarNodo ('-', SE1.i, T.nptr) }* *SE11.s }*
(24)	*SE1*	→	**ou** *T* *SE11*	*{SE11.i* *{ SE1.s*	:= :=	*CriarNodo ('ou', SE1.i, T.nptr) }* *SE11.s }*
(25)	*SE1*	→	**∈**	*{ SE1.s*	:=	*SE1.i }*
(26)	*T*	→	*F* *T1*	*{ T1.i* *{ T.nptr*	:= :=	*F.nptr }* *T1.s*
(27)	*T1*	→	***** *F* *T11*	*{ T11.i* *{ T1.s*	:= :=	*CriarNodo ('*', T1.i, F.nptr) }* *T11.s }*
(28)	*T1*	→	**/** *F* *T11*	*{ T11.i* *{ T1.s*	:= :=	*CriarNodo ('/', T1.i, F.nptr) }* *T11.s }*
(29)	*T1*	→	**div** *F* *T11*	*{ T11.i* *{ T1.s*	:= :=	*CriarNodo ('div', T1.i, F.nptr) }* *T11.s }*
(30)	*T1*	→	**mod** *F* *T11*	*{ T11.i* *{ T1.s*	:= :=	*CriarNodo ('mod', T1.i, F.nptr) }* *T11.s }*
(31)	*T1*	→	**e** *F* *T11*	 *{ T11.i* *{ T1.s*	 := :=	 *CriarNodo ('e', T1.i, F.nptr) }* *T11.s }*
(32)	*T1*	→	**∈**	*{ T1.s*	:=	*T1.i }*
(33)	*F*	→	*chamada de função*	*{ F.nptr*	:=	*CriarFolha (id, id.entrada) }*
(34)	*F*	→	**@NUM**	*{ F.nptr*	:=	*CriarFolha (num, num.val) }*
(35)	*F*	→	*(CE)*	*{ F.nptr*	:=	*CE.nptr }*
(36)	*F*	→	**nao** *F*	*{ F.nptr*	:=	*CriarNodo ('nao', F.nptr, nil) }*
(37)	*F*	→	**+** *F*	*{ F.nptr*	:=	*CriarNodo ('+', F.nptr, nil) }*
(38)	*F*	→	**-** *F*	*{ F.nptr*	:=	*CriarNodo ('-', F.nptr, nil) }*

Fonte: Baseado em (Silva, 2001)

Como a linguagem concebida caracteriza-se em uma linguagem de comandos, torna-se mais usual o emprego do termo "*script*" ao invés de "programa", embora sob o enfoque das técnicas de compiladores ambos possuam o mesmo conceito.

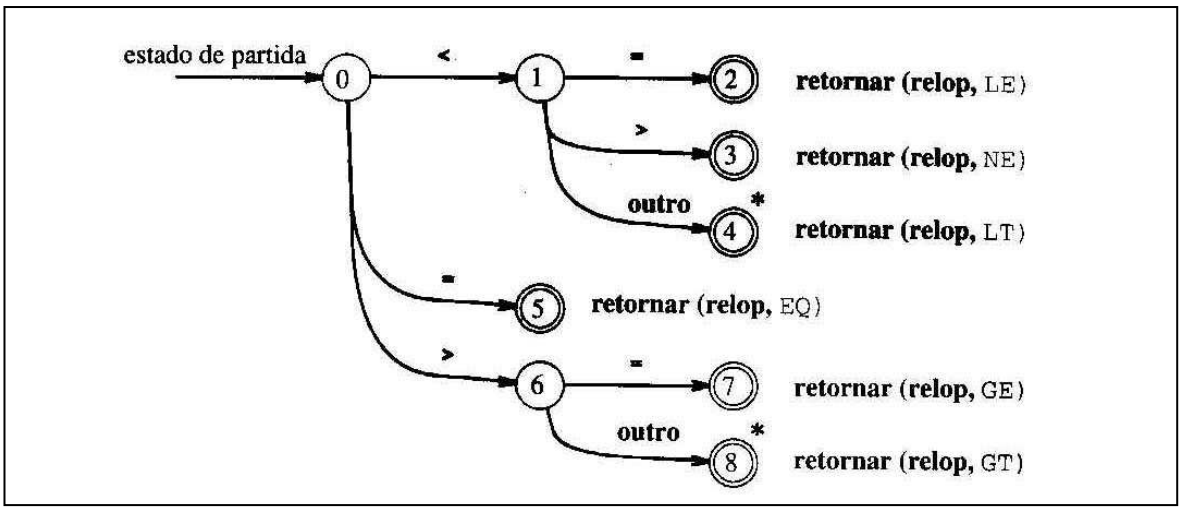
Para especificar alguns padrões léxicos utilizou-se a notação das expressões regulares, que serviram como base para a confecção dos diagramas de transições. A técnica de caminhamento através dos diagramas de transições foi convertida em um algoritmo, conforme os procedimentos descritos por Aho (1995, p. 48). O Quadro 3-7 apresenta as definições regulares aplicadas para atender aos requisitos da linguagem.

Quadro 3-7 – DEFINIÇÕES REGULARES

```
(1) relop   → <|<=|=|<>|>|>=
(2) id      → letra (letra|digito)*
(3) num     → digito+(.digito+)?(E(+|-)?digito+)?
(4) delim   → branco|tabulação|avanço de linha
(5) ws      → delim+
(6) artop   → +|-|*|/
(7) simb    → (|)|;|.|,
(8) literal → "(caracter)*"
```

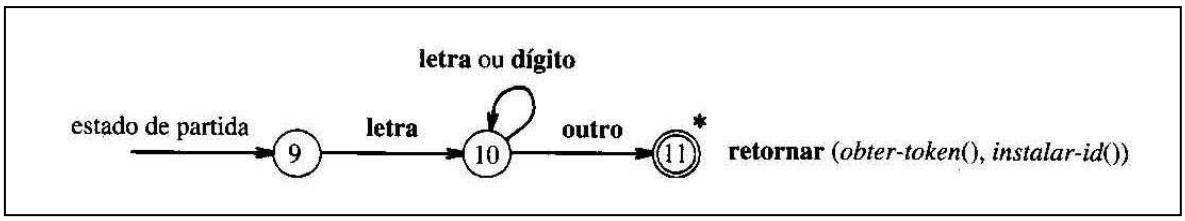
Para guiar a implementação do analisador léxico foram especificados os diagramas de transições apresentados nas Figuras 3-13, 3-14 e 3-15.

Figura 3-13 – DIAGRAMA DE TRANSIÇÕES PARA OPERADORES RELACIONAIS



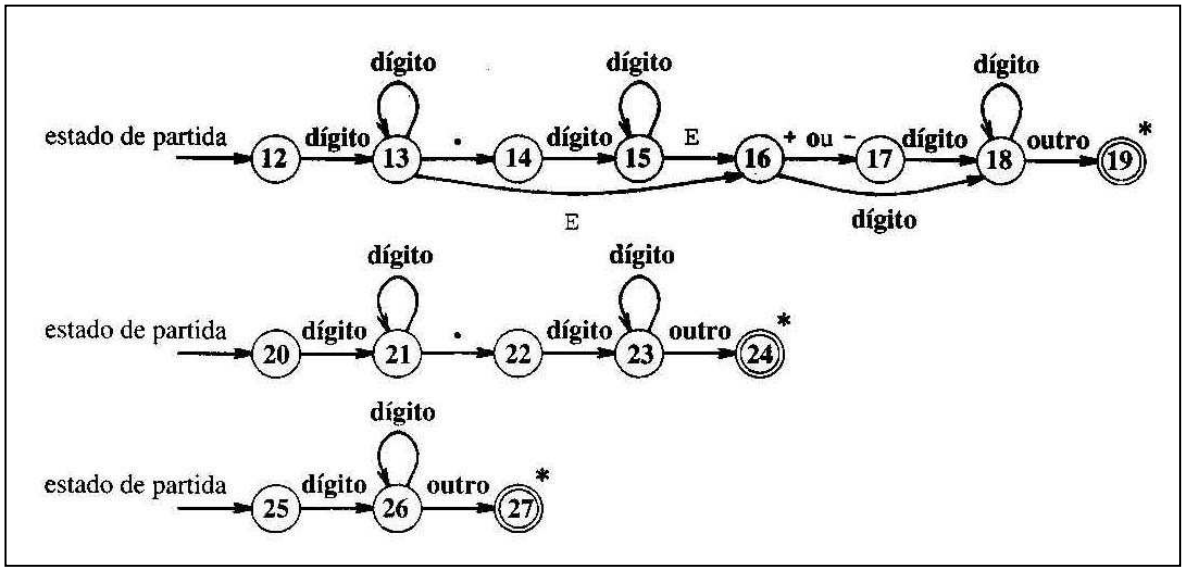
Fonte: (Aho, 1995, p. 64)

Figura 3-14 – DIAGRAMA DE TRANSIÇÕES PARA IDENTIFICADORES E PALAVRAS-CHAVE



Fonte: (Aho, 1995, p. 46)

Figura 3-15 – DIAGRAMA DE TRANSIÇÕES PARA NÚMEROS SEM SINAL EM PASCAL



Fonte: (Aho, 1995, p. 47)

Os comandos da linguagem LCS foram generalizados na forma de funções, ou seja, todos retornam um valor, ainda que nulo. Dessa forma, a geração do código-fonte consiste basicamente na chamada de comandos de alto-nível que alimentam um arquivo texto com as cadeias de caracteres que deverão corresponder à linguagem desejada. A produção 8 da gramática apresentada no Quadro 3-5 especifica uma chamada de função interna. A relação parcial das funções disponíveis na linguagem LCS é apresentada no Anexo I. Estão previstas na linguagem LCS as ações semânticas correspondentes ao tratamento dos tipos de dados apresentados no Quadro 3-8.

Quadro 3-8 – TIPOS DE DADOS DA LINGUAGEM LCS

tpdCadeia	aceita caracteres alfanuméricos, de forma análoga ao tipo *string* da linguagem *Pascal*;
tpdNumero	aceita números inteiros e de ponto flutuante;
tpdLogico	aceita valores lógicos (verdadeiro, falso);
tpdDinamico	aceita valores de qualquer tipo de dado;
tpdNulo	tipo nulo utilizado em funções que não retornam valor válido.

3.3 IMPLEMENTAÇÃO

Nesta seção serão descritas as técnicas e ferramentas utilizadas na implementação das especificações apresentadas na seção anterior. Ainda, a operacionalidade da implementação é ilustrada através de um estudo de caso.

Para implementação deste trabalho utilizou-se a linguagem *Object Pascal*, disponível no ambiente de desenvolvimento Borland Delphi versão 5.0.

Empregou-se ainda, na construção do trabalho, uma biblioteca de classes desenvolvida pela Mult Sistemas Ltda, a partir de especializações de classes da VCL.

3.3.1 TÉCNICAS E FERRAMENTAS UTILIZADAS

No intuito de manter a relação entre a especificação e a implementação, esta seção também será abordada em cinco etapas: repositório de dados do CA ERwin, repositório de dados complementar, coordenador de repositório, engenharia reversa e interpretador.

3.3.1.1 REPOSITÓRIO DE DADOS CA ERWIN

A VCL possui uma classe abstrata de persistência, denominada *TDataSet*. Especializações desta classe permitem o acesso a fontes de dados distintas, como bancos de dados com suporte a SQL, gerenciadores de arquivos, arquivos texto, entre tantos outros disponíveis no mercado.

Para implementação deste trabalho foram utilizadas duas especializações de *TDataSet*, também disponíveis na VCL, que são: *TTable* e *TQuery*. A classe *TQuery* apresenta uma considerável flexibilidade na extração de conjuntos de dados, pois suporta a linguagem SQL.

Dessa forma, quase que a totalidade das situações foram resolvidas através desta classe. A classe *TTable* possui um método de acesso mais simples, pois utiliza uma única tabela por vez. Pela sua simplicidade, empregou-se esta classe em algumas situações de edição de registros. Ambas as classes suportam bancos de dados criados pelo SGDB Sybase SQL Anywhere, utilizado no trabalho.

As classes de persistência encontram-se agrupadas em uma classe da VCL que permite a organização visual dos descendentes de *TDataSet*, denominada *TDataModule*. Todos os métodos de acesso às tabelas do repositório de dados do CA Erwin estão implementados na classe descendente denominada *TpDtmErwin*, que contém os fragmentos de código apresentado no Quadro 3-9. O identificador *fSQL* corresponde à instância de uma classe que publica métodos primitivos de manipulação da cláusula SQL.

Quadro 3-9 – FRAGMENTOS DE CÓDIGO DA CLASSE *TpDtmERwin*

```
procedure TpDtmERwin.DefinirSelecoes;
begin
  with fSQL do
  begin
    Definir (qrERW_DmnDominios,
         'select * from dba.ERW_DMN '+
         'where DIAG_ID = :DIAG_ID and DMN_INT <> 2');
    ...
  end;
end;

function TpDtmErwin.SelecionarDominiosPrimitivos (aDIAG_ID:
                                     integer): boolean;
begin
  result := fSQL.Aplicar (qrERW_DmnDominios, '', [aDIAG_ID]);
end;
```

3.3.1.2 REPOSITÓRIO DE DADOS COMPLEMENTAR

Da mesma forma que o repositório de dados do ERwin, as classes de persistência correspondentes ao repositório complementar também encontram-se agrupadas em uma especialização da classe *TDataModule*, denominada *TpDtmStruct*.

Estas classes publicam todos os métodos para manipulação das tabelas do repositório complementar. Um método exemplo é apresentado no Quadro 3-10.

Quadro 3-10 - FRAGMENTOS DE CÓDIGO DA CLASSE *TpDtmStruct*

```
function TDmDef.PosicionarDominio (aNmDmn: String): boolean;
begin
  with TabDmn do
  begin
    IndexName := 'pkDmn';
    Open;
    result := FindKey ([aNmDmn]);
  end;
end;
```

3.3.1.3 COORDENADOR DE REPOSITÓRIO

A implementação do coordenador de repositório consiste em um processo natural a partir da especificação das classes. Porém, existem situações que requerem tratamentos mais especializados, as quais serão descritas a seguir.

Como exposto anteriormente, a classe *TlClsColecaoRepositorio*, consiste em uma classe abstrata que generaliza o tratamento a coleções de componentes, ou objetos do repositório. Basicamente, a referida classe possui métodos que controlam o mecanismo de instanciação e de manipulação dos objetos da classe *TlClsObjetoRepositorio*.

Por questões de performance, os objetos correspondentes aos elementos do diagrama não podem ser instanciados de uma única vez na carga do mesmo. Para alternar entre os componentes do diagrama, o usuário interage através de um componente do tipo "*tree-view*", que apresenta uma estrutura de itens hierarquicamente organizados. A partir desta interação ocorre a chamada ao método de instanciação da coleção de objetos. Através do uso de uma propriedade (*property*), implementou-se uma forma abstrata de verificar se o objeto selecionado pelo usuário já encontra-se instanciado. Internamente, tal controle é realizado através da variável lógica *bCargaOk.*

A ferramenta *Struct* permite que determinados componentes do projeto sejam desativados, ou seja, não apresentem-se disponíveis ao usuário para seleção. Este recurso torna-se interessante quando aplicado em diagramas com um vasto número de componentes. O controle interno é implementado através da instância *fAtivos* da classe *TList* da VCL. Esta classe publica métodos de manipulação de uma lista de ponteiros. Através de uma opção

denominada gerenciador de objetos do projeto, são determinados os objetos para inclusão nesta lista.

Todo objeto do repositório, ou de outra forma, toda instância de *TlClsObjRepositorio*, que têm suas características alteradas é incluída na lista *fAlterados*. Dessa forma, sugere-se a pré-seleção destes componentes quando da ativação do assistente de execução de *scripts*.

Existe uma outra lista importante, que agrupa ponteiros para os objetos efetivamente selecionados para execução de *scripts*. Esta lista, apontada pela variável *fGerados*, serve como base para o método *Movimentar*, o qual é invocado por comandos de alto nível da linguagem, tais como *Tabela_Primeira*, *Tabela_Proxima*, ou *Tabela_Ultima*.

A classe *TlClsObjetoRepositorio* possui métodos abstratos de leitura e gravação de seus atributos a partir de chamadas a objetos da camada de persistência, além de métodos abstratos de consistência, exclusão e exportação.

Através das classes *TlClsColecaoRepositorio* e *TlClsObjetoRepositorio*, a implementação de novos tipos de componentes especificados no repositório torna-se extremamente direcionada.

3.3.1.4 ENGENHARIA REVERSA

O processo de engenharia reversa é parametrizado através das propriedades apresentadas no Quadro 3-11.

Quadro 3-11 – PARÂMETROS DA ENGENHARIA REVERSA

Titulo	nome do diagrama resultante do processo
NomeDirBase	nome do diretório padrão para localização dos arquivos-fonte Pascal
NomeArqLog	nome do arquivo de *log* alimentado pelo processo
ListaDir	lista de diretórios que devem ser ignorados
Gerar	indica se os objetos do modelo devem ser gerados fisicamente

A engenharia reversa implementada consiste em quatro etapas:

a) identificação das estruturas do código-fonte;

b) associação lógica das estruturas;

c) geração física dos elementos do modelo de dados;
d) geração do dicionário de domínios.

O processo de engenharia reversa é desencadeado através da chamada ao método *Iniciar* da classe *TlClsERPascal*. Após inicializar o arquivo de *log* do processo, é invocado um método que instancia e persiste a classe *TlObjDiagrama*.

Em seguida, o método *ProcessarDiretórios* lê todos os subdiretórios a partir de um diretório padrão. A cada subdiretório encontrado é feita uma chamada ao método *ProcessarFileBlocks*, que dispara métodos de montagem de três listas: *fileblocks*, estruturas do tipo *record* e índices, cujos métodos serão descritos a seguir.

O método *MontarListaFileBlocks* localiza todos os arquivos com a extensão "*.pas*" a partir do diretório corrente, a fim de identificar as *units* Pascal. Para cada arquivo localizado, realiza uma chamada ao método *PesquisarFileBlock*, que carrega o conteúdo do arquivo fonte em um *buffer* auxiliar e, através da identificação de palavras-chave, busca pelo procedimento de instanciação da classe do *fileblock*. Caso encontre, instancia uma classe *TFB* e alimenta suas propriedades com as características obtidas a partir do enunciado do Quadro 3-1, como nome lógico, nome físico e sigla. Extrai ainda a importante informação do tipo de registro associado ao *fileblock*. Por fim, o método inclui a instância na lista de *fLsFB*.

O método *MontarListaRecords* pesquisa os arquivos com a extensão padrão de fontes do Pascal. A cada arquivo localizado é realizada uma chamada ao método *PesquisarRecord* que busca através de palavras-chave todas as estruturas do tipo *Record* presentes no arquivo-fonte. Cada estrutura gera uma instância de *TRecord*, que recebe o nome do identificador e o nome da *unit* corrente. Em seguida são extraídas as informações de campos, instanciando e inicializando a classe *TCampo*. Após o processamento dos campos, a instância de *TRecord* é incluída na lista *fLsRec*.

O método *MontaListaIndices* pesquisa as *units* Pascal, identificando as palavras-chave que correspondem ao método de composição de chaves de índice (*MontaChave*), apresentado no Quadro 3-4. Ao encontrar a declaração do método, instancia e inicializa a classe *TMontaChave*, incluindo o ponteiro correspondente na lista *fLsInd*.

Após a identificação das estruturas passíveis de engenharia reversa, é realizada a associação entre estas estruturas, implementada pelo método *AssociarEstruturas*. Cada classe de componente da engenharia possui uma propriedade lógica *Associa*, que indica se aquela estrutura já foi associada. Para cada estrutura não associada de *fileblock* presente na lista *fLsFB*, é realizada uma pesquisa na lista *fLsRec* pelo tipo de registro obtido em *TipoRecord*. Caso seja encontrado, é feita a pesquisa a fim de obter o índice, a partir da lista *fLsInd*. O índice para as tabelas é opcional. Em seguida, é invocado o método *CriarEntidade* que instancia e armazena os objetos do coordenador de repositório para criar fisicamente todos os componentes relacionados.

Após a associação dos componentes, é necessário criar uma estrutura de domínios correspondentes aos tipos de campos extraídos da definição *Record*, ilustrado no Quadro 3-2. Para diferenciar os tipos gerados de tipos pré-existentes, concatena-se o prefixo “er_” ao nome do domínio.

Ao final do processo, o arquivo de *log* armazena uma relação dos objetos de repositório gerados e das estruturas não associadas.

3.3.1.5 INTERPRETADOR

A implementação do interpretador para a linguagem LCS envolve os componentes já descritos na especificação do recurso. Nesta seção serão apresentados os pontos de aplicação das especificações, bem como as estruturas de dados, métodos e técnicas aplicadas na implementação.

Como já mencionado, a classe *TlClsInterpretador* inicializa os componentes necessários ao processo de interpretação, ou seja, instancia as classes correspondentes às fases de análise léxica (*TlClsAnalisadorLexico*), análise sintática (*TlClsAnalisadorSintatico*), associadas às classes da tabelas de símbolos (*TlClsTabelaSimbolos*) e do tratador de erros (*TlClsTratadorErros*).

O interpretador desencadeia o processo através do método *Executar*, que recebe como parâmetro o nome do arquivo de *script* inicial. Este método alimenta o *buffer* de entrada do analisador léxico a partir da leitura do arquivo, invoca um método de inicialização da classe da tabela de símbolos e, por fim, efetua uma chamada de execução do *parser*.

Através do mecanismo de tratamento de exceções (bloco *try..except*) presente na linguagem *Object Pascal*, o interpretador detecta a ocorrência de um erro e realiza uma chamada à classe do tratador de erros, conforme o fragmento de código apresentado no Quadro 3-12. A classe do tratador de erros alimenta uma estrutura de dados que contém informações relevantes para o posicionamento no ponto exato do código do *script* onde foi detectado o erro. Uma classe da camada de apresentação lê a estrutura gerada pelo tratador de erros e apresenta o *script* ao usuário, através de um editor de *scripts* disponível na ferramenta. Seleciona ainda o último lexema obtido quando da ocorrência do erro, ou seja, o símbolo *lookahead* do analisador sintático e apresenta uma mensagem significativa de identificação do erro. Caso não exista tratamento para a exceção gerada, o método *Executar* do tratador de erros retorna valor falso e a mesma exceção é propagada.

Quadro 3-12 – MÉTODO DE ATIVAÇÃO DO INTERPRETADOR

```
function TlClsInterpretador.Executar (aNmArq: string): boolean;
begin
  try
    CriarBufferEntrada (aNmArq);
    TabelaSimbolos.Inicializar;
    AnalisadorSintatico.Parser;
  except
    on E : Exception do
    begin
      if not fTratadorErros.Executar (E) then
        raise;
    end;
  end;
end;
```

Seguindo o fluxo de execução do método *Executar*, terá início a abordagem sobre a classe do analisador sintático, que publica o método *Parser*. Este método inicializa algumas estruturas de controle da análise, descritas oportunamente, e solicita o primeiro *token*, através da chamada a função *ObterProximoToken*, publicada pela classe do analisador léxico. Esta função retorna a estrutura de dados apresentada no Quadro 3-13.

Para implementação do analisador léxico adotou-se um enfoque sistemático descrito por Aho (1995, p. 47), que converte uma seqüência de diagramas de transições em um algoritmo. Os diagramas utilizados na implementação são apresentados pela Figura 3-13, Figura 3-14 e Figura 3-15, cujos estados encontram-se numerados de forma seqüencial.

Quadro 3-13 – ESTRUTURA DE DADOS *TTOKEN*

```
TToken  = record
  fClasse      : TClasseToken;
  sLexema      : string;
  fVlLexico    : variant;
  iBuffer      : integer;
  iLinha       : integer;
end;
```

A cada estado é atribuído um segmento de código. Se existirem lados deixando um estado, então o código correspondente obtém um caractere e seleciona um lado para seguir, se possível, através da comparação com os rótulos dos lados disponíveis a partir do estado atual. A função *ObterProximoCaractere* é invocada para ler o próximo caractere a partir do *buffer* de entrada, avançar o apontador *iBufEnt* e retornar o caractere lido. Se existir um lado rotulado pelo caractere lido, ou por uma classe de caracteres que o contenha, o controle é transferido para o código do estado apontado por aquele lado. Caso não exista este lado, o estado corrente não é um daqueles que indica que um *token* foi encontrado, e, nesse caso, a função *Falhar* é invocada. Esta função retorna o apontador para a posição de início, previamente salva em *iInicioLexema*, e define o próximo diagrama de transições pelo qual o *token* será avaliado. Caso o diagrama em questão seja o último disponível, *Falhar* levanta uma exceção que será tratada pela classe responsável.

Para determinar o estado de partida do próximo diagrama de transições utiliza-se um enunciado "*case*" da linguagem *Object Pascal*. Antes de iniciar as tentativas para detecção do estado correspondente ao caractere processado, o código do analisador léxico associado ao estado 0 remove caracteres de controle e blocos de comentário, incrementando o contador de linhas.

Quando um estado de aceitação é atingido, a estrutura de dados *TToken* é atualizada e retornada pela função principal do analisador léxico *ObterProximoToken*. Cada *token* é classificado de acordo com o tipo *TClasseToken* (conforme Quadro 3-14), atribuído à variável *fClasse*. A estrutura de dados do *token* contém ainda o lexema correspondente (*sLexema*), um valor léxico atribuído ao *token* (*fVlLexico*), o índice que aponta para o início do *token* no *buffer* de entrada (*iBufEnt*) e a linha de ocorrência do *token* dentro do *script* (*iLinha*).

Inicialmente a estrutura de dados do *token* possui uma classe não significativa (*ctkNula*). Qualquer alteração na classe do *token*, imediatamente força o término do laço que processa o *buffer* de entrada, que pode ocorrer a partir dos seguintes eventos: reconhecimento de um *token*, fim do arquivo de *script* atingido ou exceção interna do interpretador.

Quadro 3-14 – TIPO ENUMERADO CORRESPONDENTE AO *TOKEN*

```
TClasseToken    = (ctkID, ctkLITERAL, ctkNUM, ctkSIMB,
                   ctkEOF, ctkNula,

                   // operadores relacionais
                   ctkRELOPIGUAL, ctkRELOPDIFERENTE,
                   ctkRELOPMENOR, ctkRELOPMAIOR,
                   ctkRELOPMENORIGUAL, ctkRELOPMAIORIGUAL,

                   // operadores aritmeticos
                   ctkARTOPADICAO, ctkARTOPSUBTRACAO,
                   ctkARTOPMULTIPLICACAO, ctkARTOPDIVISAO,
                   ctkARTOPDIV, ctkARTOPMOD,

                   // operadores logicos
                   ctkLOGOPOU, ctkLOGOPE, ctkLOGOPNOT);
```

Os estados de aceitação do diagrama apresentado na Figura 3-13 identificados pelos números 2, 3, 4, 5, 7 e 8, atribuem a classe do *token* correspondente ao operador identificado. Quando alcançado o estado de aceitação 11 do diagrama da Figura 3-14, o analisador instala o *lexema* na tabela de símbolos e atribui a classe *"ctkID"* ao *token*. O estado 19 do diagrama da Figura 3-15 atribui ao valor léxico do *token* (*fVlLexico*) o resultado da conversão do *lexema* (*sLexema*) para um valor de ponto flutuante e retorna a classe *ctkNUM*.

A classe do analisador léxico especifica ainda uma estrutura de dados que armazena atributos relacionados ao contexto do próprio analisador, apresentados no Quadro 3-15. Esta estrutura permite que sejam definidos rótulos ou *labels*, para utilização através do comando "*Enquanto*" que altera o fluxo de controle da linguagem LCS, gerando uma estrutura de laço através da restauração do contexto sempre que a expressão lógica for satisfeita.

Retornando ao escopo do analisador sintático, a estrutura do *token* fornecida pelo analisador léxico é atribuída ao símbolo *lookahead*, implementado como um atributo privado *fLookAhead* do tipo *TToken*. Para facilitar o processamento do símbolo *lookahead*, a classe

do analisador sintático implementa alguns métodos destinados a reconhecer classes de *tokens*, terminais, funções e analisar o primeiro terminal das possíveis produções.

Quadro 3-15 – ESTRUTURA DE CONTEXTO DO ANALISADOR LEXICO

```
PRotulo = ^TRotulo;
TRotulo = record
  iLinhaAtual   : integer;
  iInicioLexema : integer;
  iBufEnt       : integer;
  fLookAhead    : TToken;
end;
```

Após inicializar o atributo *fLookAhead* com a estrutura do primeiro *token* reconhecido, o analisador sintático executa uma chamada ao procedimento correspondente ao símbolo de partida da gramática, ou seja, o não-terminal "*Script*".

A gramática especificada para a linguagem LCS é processável por um analisador sintático de descendência recursiva que não necessita de retrocesso, ou analisador sintático preditivo. Dessa forma, o analisador deve conhecer, a partir do terminal mais à esquerda das produções, qual alternativa é a única que deriva a cadeia começando pelo símbolo corrente de entrada (Aho, 1995, p. 81).

Antes de invocar o método correspondente ao símbolo inicial da gramática (*Script*), o analisador sintático executa uma chamada ao método *DefinirFuncoes,* o qual instala todas as funções internas da linguagem LCS na tabela de símbolos. Para cada função é registrada a palavra reservada correspondente ao nome da mesma, os tipos de dados dos argumentos de entrada e o tipo de dado do retorno, além do ponteiro para o método de avaliação da função.

A classe *TlClsTabelaSimbolos* herda de *TStringList* o comportamento necessário para manipulação de listas de *strings*, que, no caso corresponderão aos lexemas. Paralelamente, esta classe mantém uma lista de ponteiros implementada através de uma instância *fSimbolos* da classe *TList*. Cada item da lista aponta para a estrutura básica de dados do símbolo apresentada no Quadro 3-16.

Com o propósito de instalar símbolos na tabela, a classe *TpClsTabelaSimbolos* publica dois métodos: *Instalar* e *InstalarFuncao*.

Quadro 3-16 – ESTRUTURA BÁSICA DE UMA ENTRADA NA TABELA DE SÍMBOLOS

```
TpSimbolo = ^TrSimbolo;
TrSimbolo = record
  fTipo   : TTipoSimbolo;
  sNome   : string;
  pAux    : pointer;
end;
```

O método *Instalar* recebe como parâmetros o lexema do identificador e o tipo de símbolo que deve ser instalado, conforme o Quadro 3-17. Inicialmente, verifica a pré-existência do lexema passado por parâmetro. Caso não encontre, aloca e inicializa a estrutura apontada por *TpSimbolo*, incluindo-o na lista *fSimbolos*, e adiciona o lexema na instância da classe da tabela de símbolos. Este método é invocado diretamente pelo analisador léxico, quando um identificador é reconhecido.

Quadro 3-17 – TIPOS POSSÍVEIS DE SÍMBOLOS

```
TTipoSimbolo = (tsbFuncao,
                tsbIdentificador,
                tsbNaoAvaliado);
```

O método *InstalarFuncao* recebe como parâmetros o lexema, um *array* de tipos de dados dos argumentos, um tipo de dado de retorno e um ponteiro para o método de avaliação da função. Na tabela de símbolos uma função corresponde a um símbolo do tipo *TpSimbolo* que através de *pAux*, aponta para uma estrutura de dados complementar, apresentada no Quadro 3-18. Esta estrutura contém os seguintes campos: o ponteiro *pArgIni* para a estrutura do primeiro argumento da função, apresentada no Quadro 3-19; o tipo de dado de retorno da função, conforme o Quadro 3-20, e o ponteiro para um método de avaliação da função.

Quadro 3-18 – ESTRUTURA DE DADOS DE UM SÍMBOLO DO TIPO FUNÇÃO

```
TpFuncao = ^TrFuncao;
TrFuncao = record
  pArgIni   : TpArgumento;
  fTpRet    : TTipoDado;
  pRotina   : TRotinaAux;
end;
```

Quadro 3-19 – ESTRUTURA DE DADOS DE UM ARGUMENTO DE UMA FUNÇÃO

```
TpArgumento = ^TrArgumento;
TrArgumento = record
  fTipo : TTipoDado;
  pProx : TpArgumento;
  pNo   : TpNoArvore;
end;
```

Quadro 3-20 –ESTRUTURA DE TIPOS DE DADOS LCS

```
TTipoDado = (tpdCadeia, tpdNumero, tpdLogico,
             tpdDinamico, tpdNulo);
```

Inicialmente, *InstalarFuncao* invoca o método *Instalar* passando como parâmetros o lexema e o tipo *tsbFuncao*. Caso a instalação do lexema ocorra com sucesso, a estrutura apontada por *TpFuncao* é alocada e inicializada, bem como uma lista encadeada de argumentos da função. Cada nó da lista de argumentos, correspondente a estrutura apontada por *TpArgumento*, possui um tipo de dado (*fTipo*), o ponteiro para o próximo nó da lista (*pProx*) e um ponteiro para o nó da árvore sintática, utilizado na função de avaliação e descrito com detalhes posteriormente. O método *InstalarFuncao* é invocado pela classe do analisador sintático antes de iniciar o processo de *parser*.

A classe da tabela de símbolos implementa também os métodos de pesquisa de identificadores: *Buscar* e *BuscarFuncao.* Ambos recebem como parâmetro a cadeia do lexema e retornam o ponteiro para a estrutura do símbolo e o ponteiro para estrutura da função, respectivamente.

Por fim, a classe *TlClsTabelaSimbolos* implementa os métodos de desinstalação de símbolos: *Remover* e *RemoverFuncao*. O método *Remover* libera as estruturas e remove o item das listas, cujo índice é passado por parâmetro. O método *RemoverFuncao* recebe um ponteiro do tipo *TpFuncao* e libera as estruturas complementares do símbolo do tipo função.

A seguir serão detalhadas as técnicas para implementação dos métodos correspondentes aos não-terminais da gramática da linguagem LCS, apresentada no Quadro 3-5. O nome dos métodos corresponde ao nome do não-terminal, sendo que, por questões de sintaxe, o símbolo ‘ (linha) foi substituído pelo caractere 1 (um).

A árvore sintática é implementada através da instância *fArvore* da classe *TList*. Cada item da lista corresponde a um ponteiro *TpNoArvore* para a estrutura de nó da árvore sintática, apresentada no Quadro 3-21. Cabe mencionar que a implementação da árvore sintática para o interpretador da linguagem LCS deu-se por propósitos exclusivamente didáticos, visto que a execução das regras semânticas é realizada durante o processo de análise sintática.

Quadro 3-21 – ESTRUTURA DO NÓ DA ARVORE SINTÁTICA

```
TpNoArvore = ^TrNoArvore;
TrNoArvore = record
  fRotulo       : TClasseToken;
  pNoEsq        : TpNoArvore;
  pNoDir        : TpNoArvore;
  iIDEntrada    : integer;
  fValor        : variant;
  fTipo         : TTipoDado;
end;
```

Para tratamento da árvore sintática, a classe do analisador implementa os seguintes métodos: *CriarNo*, *AvaliarNo*, *CriarFolha* e *InicializarArvore Sintatica*.

O método *CriarNo* recebe como parâmetro um rótulo do tipo *TClasseToken*, bem como os ponteiros *TpNoArvore* que figurarão à esquerda e à direita do nó. O método aloca e inicializa a estrutura apontada por *TpNoArvore*, incluindo este ponteiro na lista *fArvore*. Em seguida executa uma chamada ao método *AvaliarNo* passando como parâmetro o ponteiro para o nó recém-criado. Por fim, este ponteiro é retornado.

O método *AvaliarNo* recebe um ponteiro *TpNoArvore* como parâmetro. Inicialmente, o método realiza a ação semântica de comparar os tipos dos nós à esquerda e à direita, considerando que estejam associados. Caso os tipos não sejam idênticos, uma exceção é levantada indicando tipos incompatíveis na expressão. Caso contrário, é utilizado um enunciado *case* para definir qual a operação que deve ser aplicada, de acordo com o tipo de operador correspondente ao rótulo do nó. Os operadores são classificados em aritméticos, relacionais e lógicos. A avaliação da expressão correspondente ao operador resulta em um valor booleano que é armazenado no campo *fValor* da estrutura do nó em questão. Ao final, o tipo do nó à esquerda é assumido pelo nó avaliado.

O método *CriarFolha* recebe como parâmetros um rótulo *TClasseToken* e um valor do tipo *variant*. A execução do método identifica o tipo de rótulo, atribuindo o tipo e o valor léxico correspondente. Se o rótulo corresponder a um identificador (*ctkID*), assume-se que o parâmetro valor contém um ponteiro *TpNoArvore*, ou seja, é um nó resultado da avaliação de uma função. Esta situação é gerada pela produção 33 da gramática da linguagem LCS, apresentada no Quadro 3-5.

O tipo *variant* disponível na linguagem *Object Pascal* permite a manipulação de variáveis cujo tipo é determinado somente em tempo de execução. A partir da atribuição de um valor, o tipo *variant* assume o tipo da expressão do lado direito da atribuição. O tipo *variant* serviu à implementação do interpretador como um método abstrato de avaliação de expressões, isentando a criação de um *buffer* especial para cada tipo de dado.

3.3.2 OPERACIONALIDADE DA IMPLEMENTAÇÃO

A operacionalidade da implementação é ilustrada através da adaptação de um estudo de caso real, baseado em um produto de *home-office banking* desenvolvido pela empresa Mult Sistemas.

O termo *home banking* tem sido utilizado para classificar um software de comunicação fornecido pelas instituições bancárias, que permite ao cliente comandar operações, antes somente possíveis quando efetuadas através da agência. Estas operações são denominadas transações bancárias, sendo geralmente categorizadas em consultas, transferências de fundos e pagamento eletrônico, as quais são funcionais tanto para pessoas físicas, como para empresas. Ao software que além das funcionalidades citadas, agrega a possibilidade de automatização da cobrança bancária de empresas dá-se a denominação *home-office banking*.

Este tipo de produto possui dois usuários finais: o banco e o cliente. Ao banco basicamente interessa um alto grau de flexibilidade, principalmente na confecção das transações que serão disponibilizadas. Ao cliente, são relevantes os aspectos de facilidade de operação e agilidade no retorno das transações solicitadas.

A primeira versão do produto em questão foi lançada comercialmente no ano de 1994, tendo sido codificada na linguagem *Borland Pascal 6.0* para a plataforma DOS, amparada por

uma biblioteca de desenvolvimento que utilizava o gerenciador de arquivos *Turbo Power BTree-Filer*.

Atualmente, a empresa objetiva uma evolução tecnológica do produto de *home-office banking*, em dois aspectos: sistema operacional e base de dados. Com relação ao ambiente operacional, deve ser empregada a linguagem *Object Pascal* para o ambiente *Microsoft Windows.*

Com relação à base de dados, a empresa objetiva a utilização do SGBD *Interbase*, porém mantendo algumas tabelas no gerenciador de arquivos *BTree-Filer*. Tal decisão é justificada pelo fato de que os arquivos do gerenciador correspondem a tabelas de parâmetros alimentadas pelo banco, as quais fornecem a flexibilidade desejada de atualizar remotamente determinados recursos nas instalações dos clientes, como por exemplo, a tabela de agências bancárias, ou as definições do formato das transações. Ao SGBD Interbase, caberá o armazenamento dos dados alimentados pelo cliente, como os cadastros de contas, os parâmetros de cobrança bancária, entre outros. Como resultado de uma avaliação dos recursos da versão legada, foi identificada a possibilidade de reaproveitamento da definição dos arquivos.

Apresentado o cenário do estudo de caso, será descrita uma forma de solução do problema exposto utilizando-se a ferramenta *Struct*.

Inicialmente, serão introduzidos aspectos relacionados à interface do sistema. Para tanto, a Figura 3-16 apresenta a identificação dos elementos de interface presentes na janela principal. A Figura 3-17 ilustra ainda a explosão de cada item do menu principal.

Os diagramas manipulados pelo *Struct* são organizados em “projetos”. Cada projeto possui um conjunto de propriedades associadas, ilustradas na Figura 3-18 que permitem a criação de um ambiente específico para cada sistema. Pode-se trabalhar com vários projetos abertos ao mesmo tempo, alternando facilmente entre eles, através da opção “Janelas”.

Figura 3-16 – JANELA PRINCIPAL DA FERRAMENTA *STRUCT*

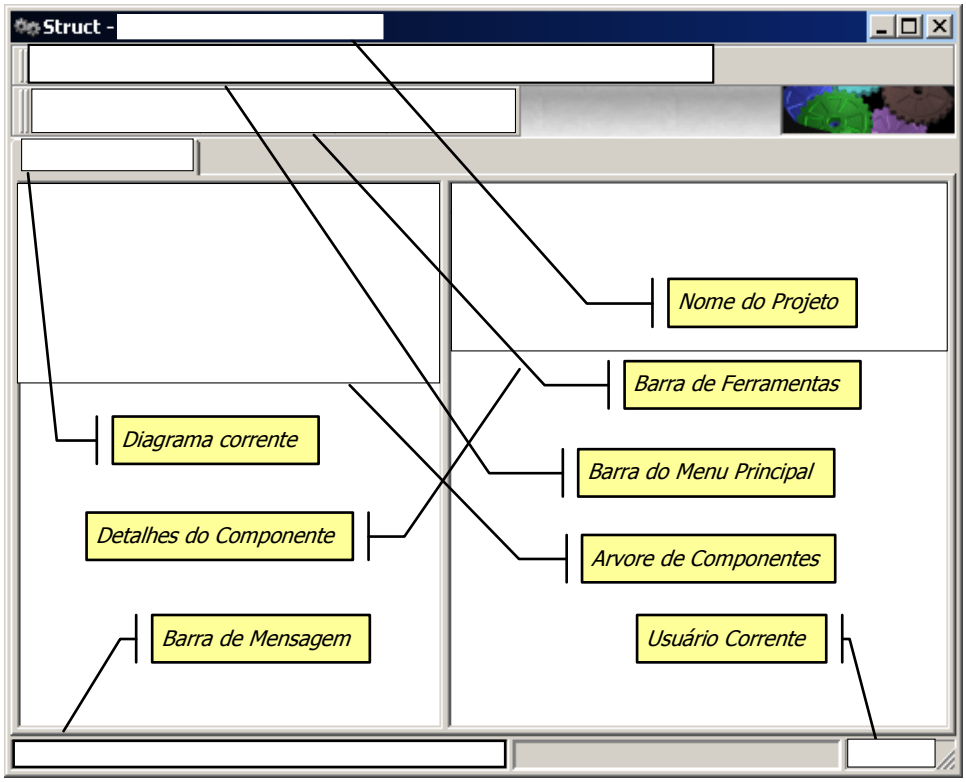


Figura 3-17 – OPÇÕES DO MENU PRINCIPAL DO *STRUCT*

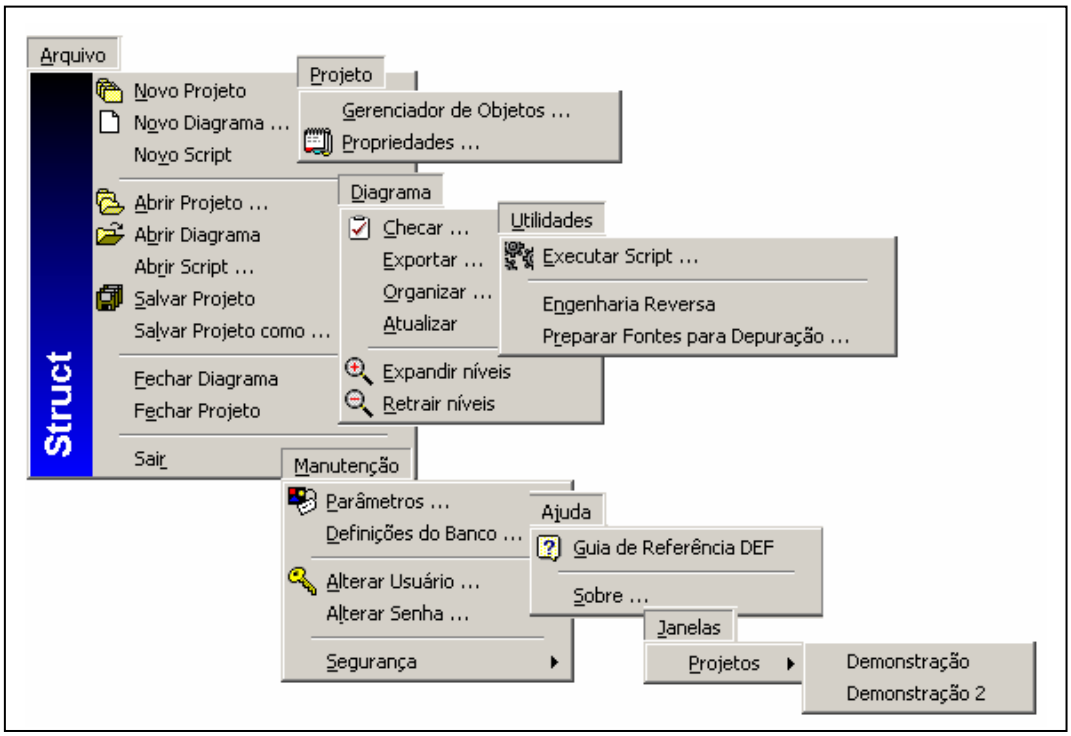
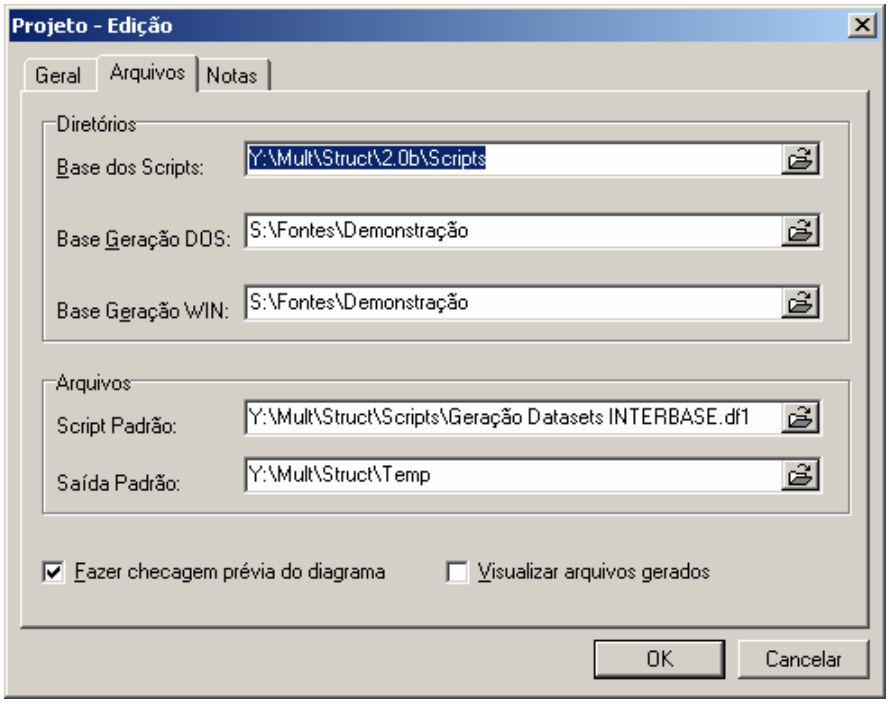


Figura 3-18 – DIÁLOGO DE PROPRIEDADES DO PROJETO

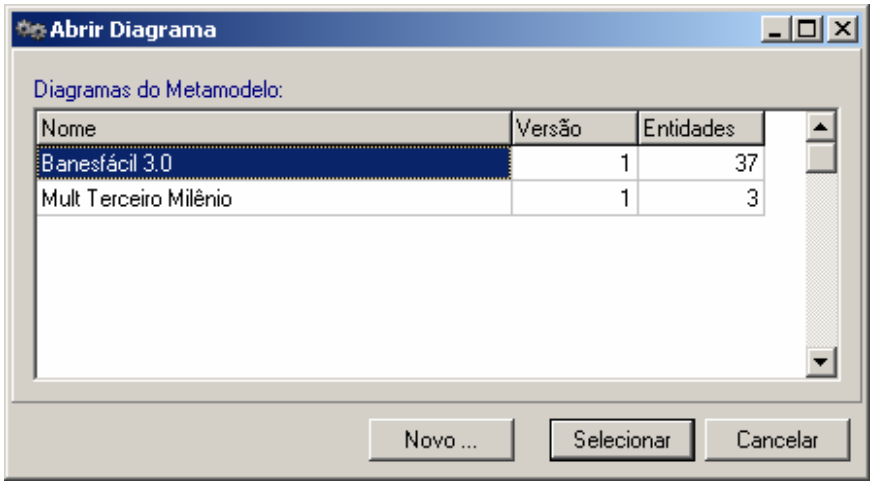


Para obter acesso ao sistema, deve-se informar um nome do usuário e uma senha, os quais devem ser previamente registrados no sistema através das opções do menu “Manutenção | Segurança”. Cada usuário deve ser alocado em um grupo, o qual pode possuir permissões de administrador. O administrador da ferramenta pode realizar manutenção na base de dados, através da inclusão, alteração e exclusão de componentes. Aos usuários não-administradores somente é permitida a consulta aos dados destes componentes.

Após a validação do nome do usuário e da senha de acesso a ferramenta estabelece a conexão com o banco de dados do repositório. Em seguida, inicia a carga automática dos projetos ativos na última sessão ou solicita a criação de um novo projeto, a qual é realizada através do preenchimento do diálogo da Figura 3-18 e da seleção de um diagrama inicial para trabalho, através da janela ilustrada na Figura 3-19.

A grande maioria dos componentes do repositório de dados podem ser mantidos através da ferramenta *Struct*, sem a necessidade de utilização do ERwin.

Figura 3-19 – SELEÇÃO DE DIAGRAMAS DO REPOSITÓRIO



Após a seleção de um diagrama, a ferramenta carrega seus componentes básicos, como domínios, áreas de assunto e entidades, agrupando-os hierarquicamente em um controle visual, identificado na Figura 3-16 como árvore de componentes.

Realizados os procedimentos operacionais descritos, o *Struct* está disponível para uso, o qual será demonstrado com o prosseguimento do estudo de caso.

Inicialmente, aplicou-se a rotina de engenharia reversa disponível na ferramenta, acessada através da combinação das seleções “Utilidades | Engenharia Reversa | Fontes Pascal”. O diálogo para informação dos parâmetros da rotina é apresentado na Figura 3-20. O diagrama destino para a engenharia foi criado previamente através da opção “Arquivo | Novo Diagrama”.

Concluído o processo, as definições extraídas a partir dos arquivos-fonte originam componentes do diagrama informado, os quais podem ser visualizados automaticamente a partir do controle de árvore de componentes da janela principal.

Em seguida, utilizou-se a ferramenta ERwin para a inclusão dos relacionamentos, pois não existe possibilidade de extração destes a partir do código-fonte. Foram ainda realizados alguns ajustes visuais, como reposicionamento das entidades, alteração de cores e organização destas entidades em áreas de assunto (*subject area*). O diagrama resultante destas alterações é apresentado na Figura 3-21.

Figura 3-20 – DIÁLOGO DA ROTINA DE ENGENHARIA REVERSA

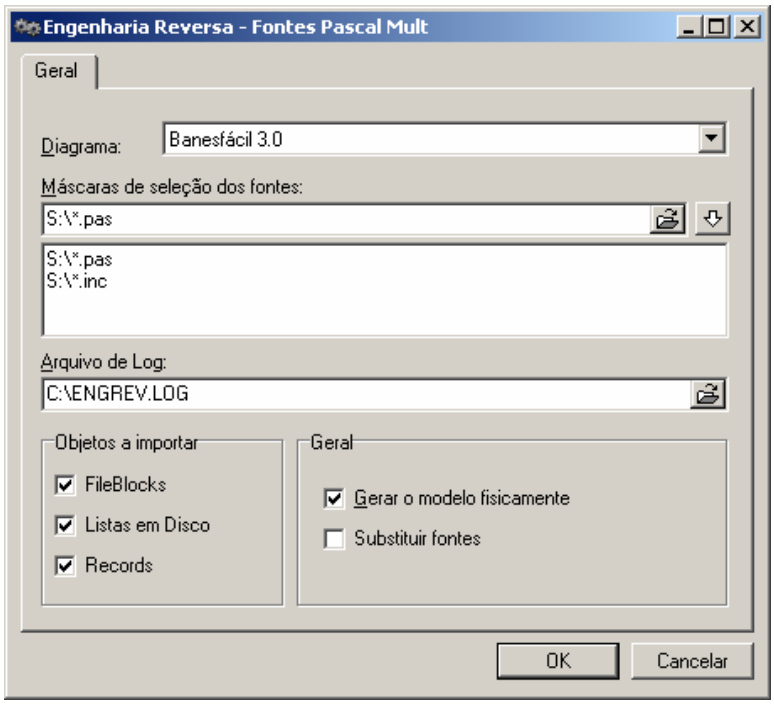
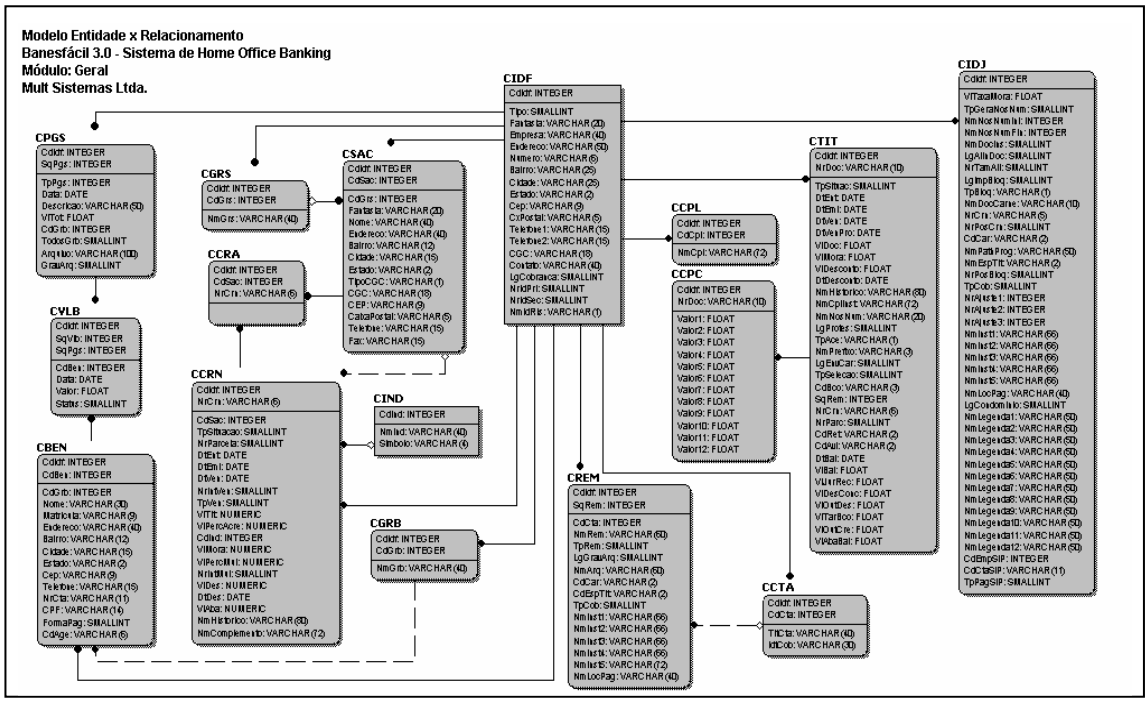


Figura 3-21 – DER AJUSTADO APÓS A ENGENHARIA REVERSA



Retornando-se a ferramenta *Struct*, podem ser alimentadas as informações complementares para cada componente do diagrama, as quais são direcionadas para geração de código-fonte. A Figura 3-22 apresenta os atributos expandidos da entidade “Agências Bancárias”, sendo que a Figura 3-23 apresenta o diálogo de edição do atributo “C.E.P”.

Figura 3-22 – DIAGRAMA RESULTADO COM EXPLOSÃO DE ATRIBUTOS

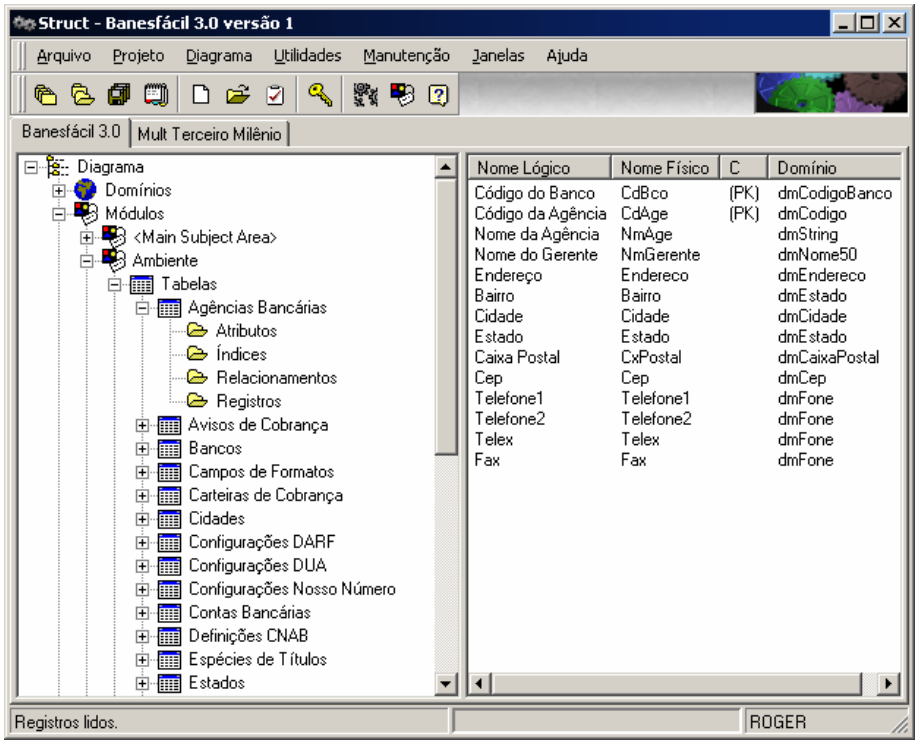
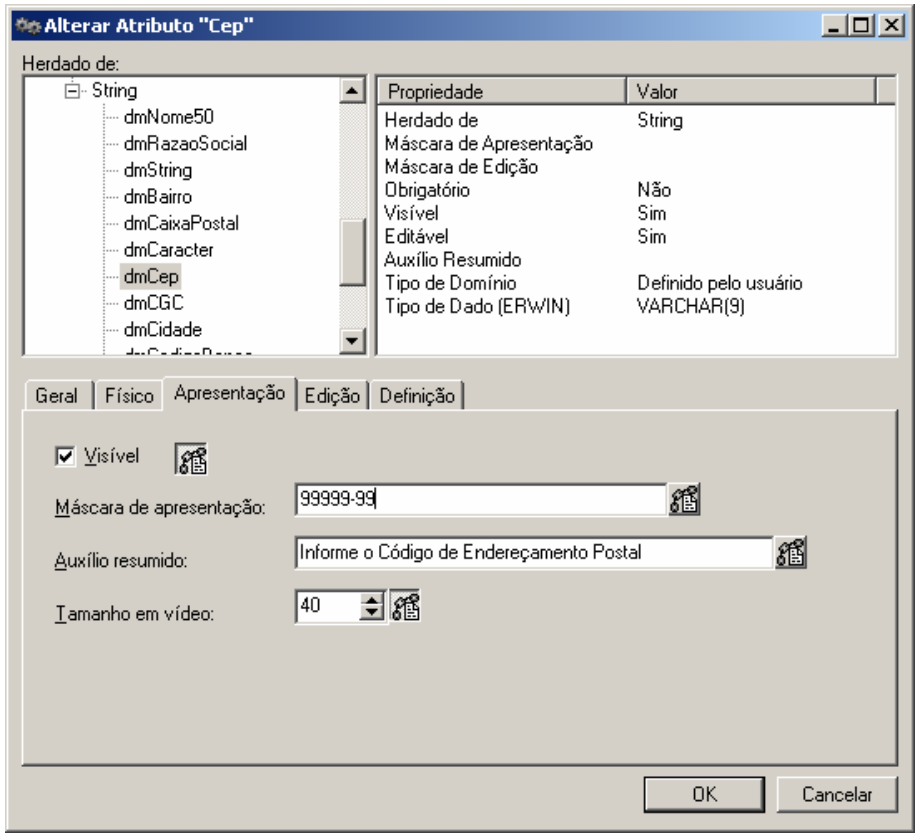


Figura 3-23 – DIÁLOGO DE EDIÇÃO DE ATRIBUTOS



Cada atributo herda definições do domínio associado. Esta característica pode ser ativada ou não, através do botão presente ao lado dos campos de edição.

Assim como os atributos, os demais componentes também apresentam características complementares editadas em conjunto com as características originárias do ERwin.

A partir da carga de um projeto contendo o diagrama em questão, seleciona-se a opção “Utilidades | Executar Script”, a qual apresenta o assistente cujas etapas são visualizadas na Figura 3-24 e na Figura 3-25. A primeira etapa solicita a seleção dos objetos que devem ser considerados na execução do *script*. A partir da segunda etapa são informados os arquivos de *script* e saída iniciais, além de algumas preferências para geração.

Caso tenha-se optado por realizar uma checagem prévia do diagrama, a ferramenta executa uma série de verificações para todos os objetos selecionados para execução do *script*, caso contrário, automaticamente tem início a interpretação do *script* inicial informado. As críticas geradas durante a checagem prévia são apresentadas na janela da Figura 3-26.

Figura 3-24 – ETAPA 1 DO ASSISTENTE PARA EXECUTAR *SCRIPT*

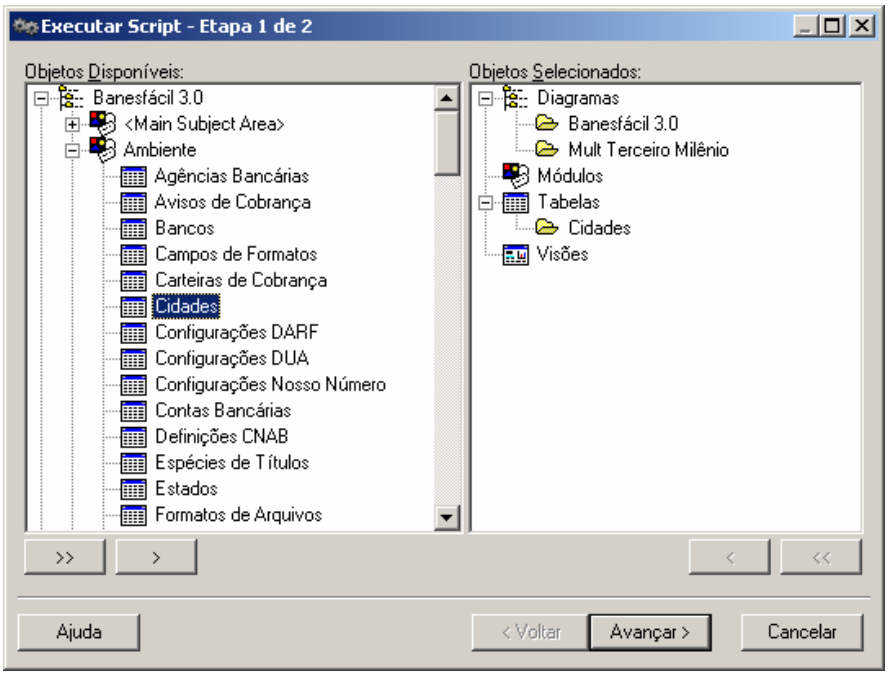


Figura 3-25 - ETAPA 2 DO ASSISTENTE PARA EXECUTAR *SCRIPT*

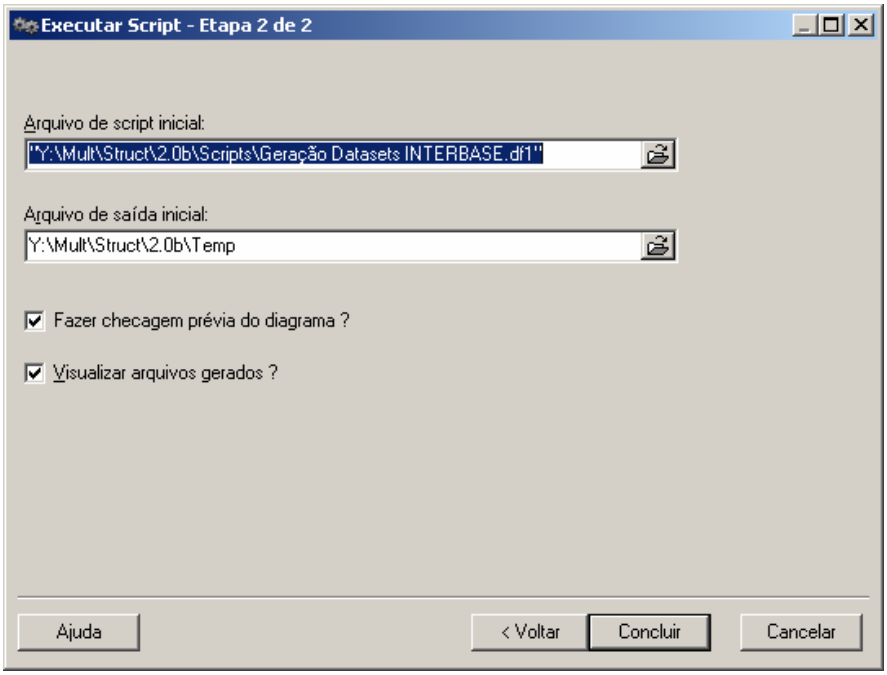
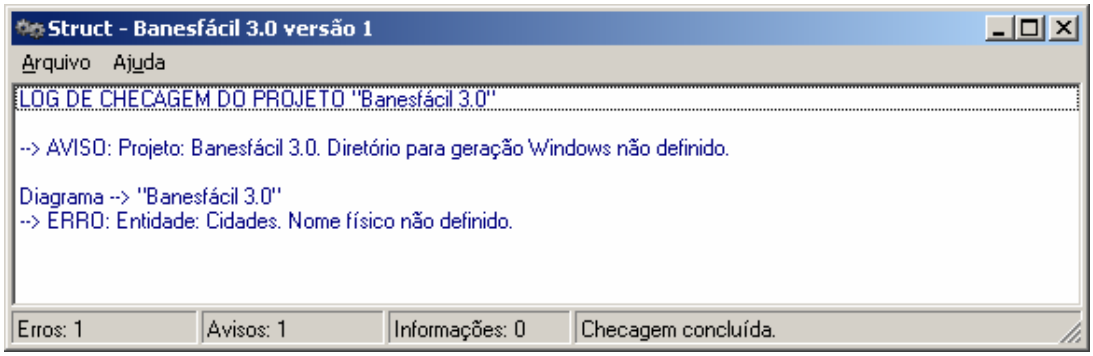


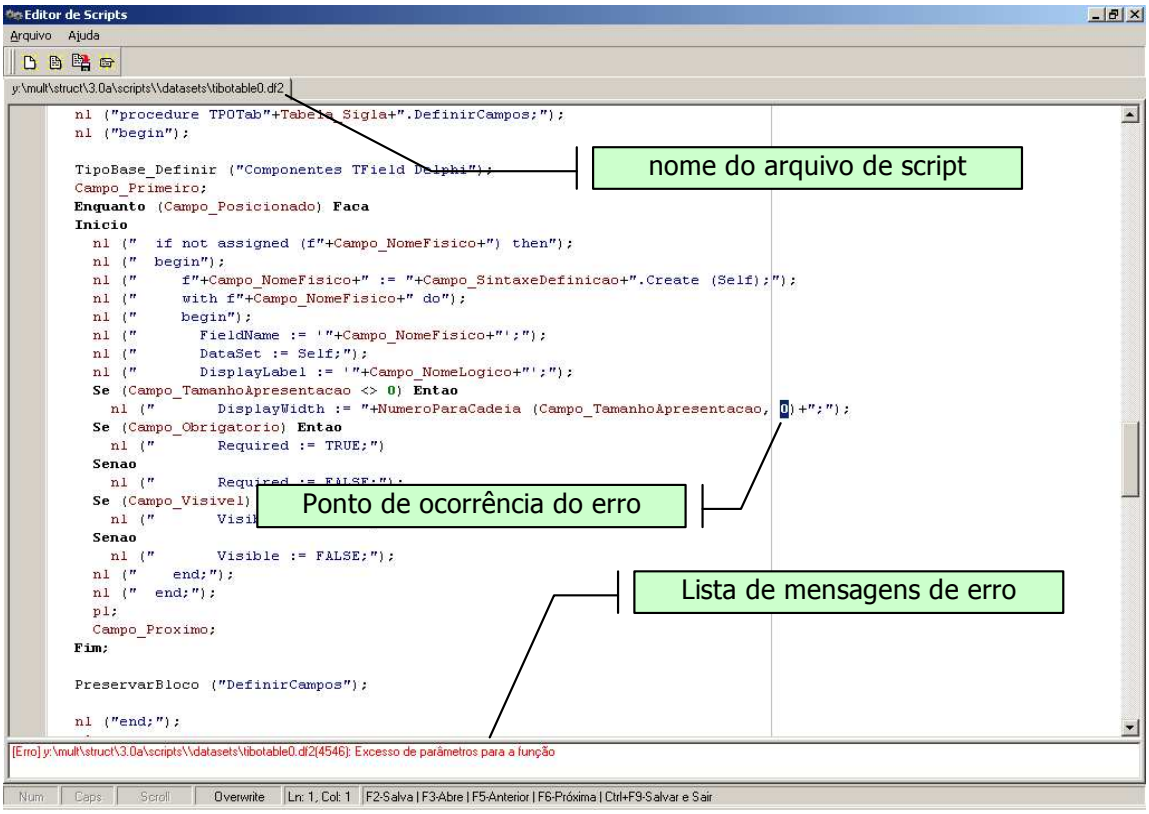
Figura 3-26 – JANELA DE OCORRÊNCIAS DO *LOG*



Para contemplar ao requisito do estudo de caso que propõe a geração da classe de persistência para tabelas *Interbase* e *BTree-Filer*, foram codificados dois *scripts*, cujo código é apresentado no Anexo II. Foram criados ainda dois módulos para agrupar as tabelas de acordo com o tipo de base de dados. Dessa forma, a partir da seleção do objeto “Tabelas *Interbase*” automaticamente todas as tabelas associadas são incluídas na lista para geração.

Ao detectar-se um erro, o processo de interpretação é automaticamente interrompido e o arquivo de *script* é carregado no editor apresentado na Figura 3-27.

Figura 3-27 – JANELA DO EDITOR DE *SCRIPTS* REPORTANDO UM ERRO



A conclusão com sucesso do processo de interpretação apresenta a janela da Figura 3-28. Pode-se visualizar os arquivos resultantes da interpretação, conforme a janela apresentada na Figura 3-29.

Figura 3-28 – JANELA DE INTERPRETAÇÃO COM SUCESSO

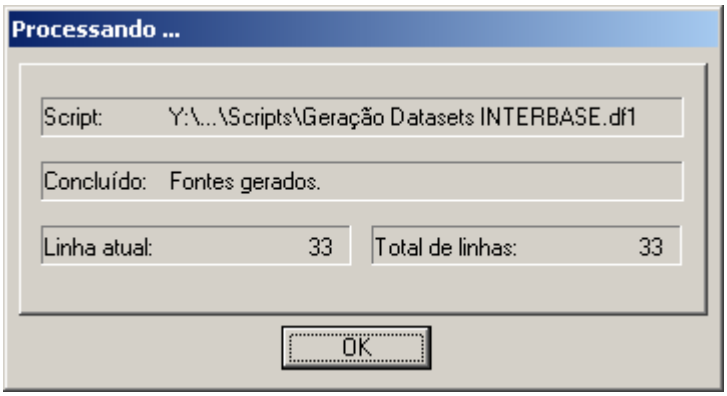
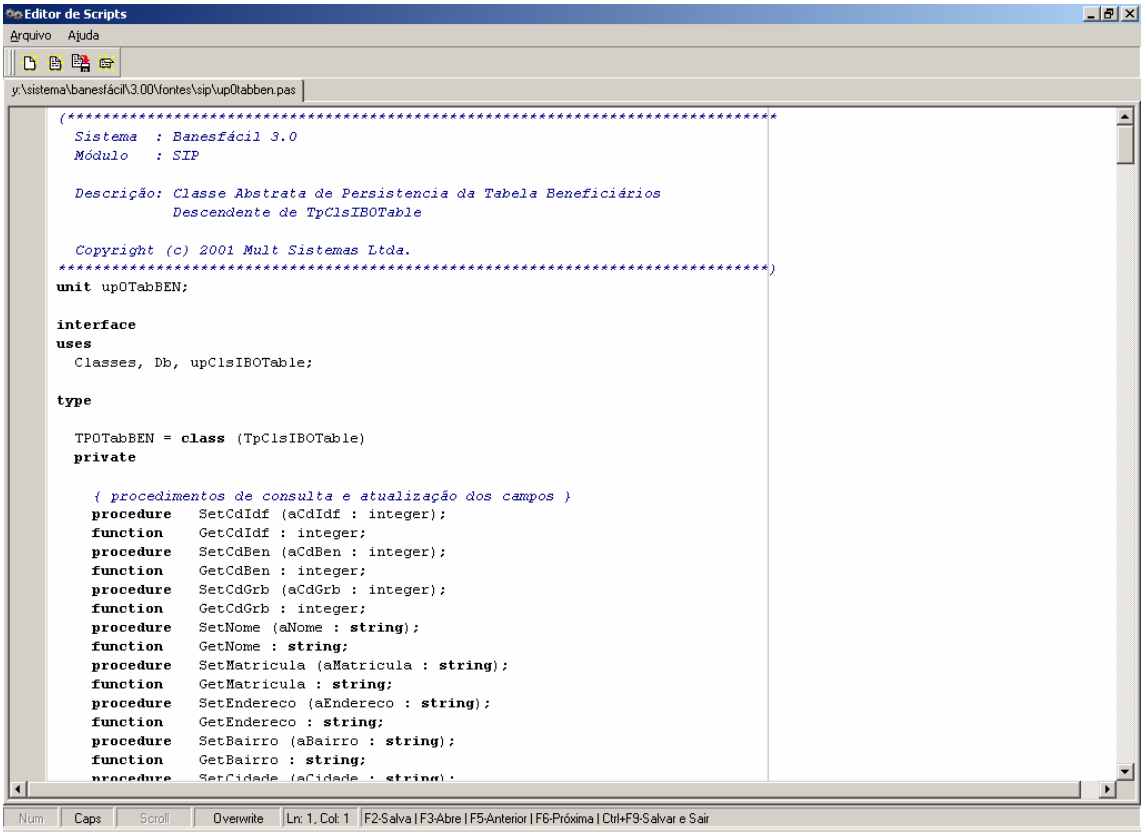


Figura 3-29 – VISUALIZAÇÃO DE UM ARQUIVO-FONTE GERADO



O arquivo de código-fonte gerado a partir da execução do *script* para tabelas *Interbase* pode ser verificado no anexo III. Porém, a geração de código-fonte é uma das aplicações da ferramenta *Struct*. Através das funções de tratamento do arquivo de saída, apresentadas no anexo I, podem ser gerados arquivos texto em geral, como documentação de sistemas, ou arquivos HTML. O anexo IV ilustra esta possibilidade adicional através de um modelo de *script* para geração de arquivos de documentação de tabelas, sendo que anexo V apresenta o resultado da execução do mesmo.

3.4 DEMAIS FASES DO CICLO DE VIDA

Nesta seção serão descritas brevemente as demais fases do ciclo de vida adotado pelo presente trabalho, que são teste beta, lançamento e manutenção.

O teste beta do produto foi realizado através de um ambiente de testes, originado a partir da duplicação do ambiente de produção da Mult Sistemas. Além de validar a ferramenta

em questão, houve a necessidade de validar também os sistemas envolvidos, pois estes compilavam o código-fonte gerado pelo produto deste trabalho.

O lançamento ocorreu após a certificação de que todos os recursos incorporados ao produto haviam sido validados. Na prática, o lançamento envolveu a aplicação da engenharia reversa a partir dos arquivos-fonte do ambiente de produção, a publicação do produto para utilização pela equipe de desenvolvimento e o início do treinamento de pessoal.

A manutenção representa a fase atual do ciclo de vida da ferramenta *Struct*, que envolve basicamente ajustes e pequenos melhoramentos. À medida em que surgirem necessidades de novos recursos, retorna-se a fase de análise de requisitos, respeitando o modelo de desenvolvimento cíclico.

3.5 RESULTADOS E DISCUSSÃO

Durante a fase inicial de desenvolvimento do trabalho, identificou-se uma deficiência no planejamento das atividades alocadas à construção do interpretador e à especificação da linguagem LCS. Houve uma tentativa de construir os componentes referidos utilizando uma forma empírica, desconsiderando o emprego de fundamentação científica. Através desta tentativa, pode-se identificar uma série de dificuldades, principalmente relacionadas à especificação da linguagem e a estruturação do interpretador. A medida em que aumentava o grau de complexidade do algoritmo, surgiam situações não previstas e de difícil tratamento.

Constatada a inviabilidade do método exposto, iniciou-se um estudo das técnicas relacionadas à área de compiladores, no sentido de identificar aspectos relevantes a um interpretador de uma linguagem de comandos e abstrair aspectos não condizentes com tal objetivo. Este processo foi auxiliado pelo Prof. José Roque Voltolini da Silva, o qual expôs uma visão geral do escopo de um interpretador na área de compiladores e forneceu a gramática de atributos para análise de expressões com precedência de operadores, cujas produções complementaram a gramática da linguagem LCS.

Com exceção da análise semântica, cada fase do processo de compilação foi especificada através de uma classe distinta. Embora esta forma de implementação não seja comum, vislumbrou-se o propósito didático de tornar explícita a independência e identificar claramente as interações entre as fases do processo de compilação. Através da orientação a

objetos, aplicou-se o conceito de encapsulamento dos atributos internos e da publicação dos métodos de escopo mais abrangente. Durante a manutenção do código-fonte das classes componentes do interpretador, detectou-se um alto grau de legibilidade, fator pelo qual acredita-se ter alcançado os propósitos desejados com esta forma de especificação.

4 CONCLUSÕES

O objetivo do presente trabalho foi atingido, o qual consistia no desenvolvimento de uma ferramenta CASE que possibilitasse a geração de código-fonte a partir das definições contidas no repositório de dados da ferramenta de modelagem ERwin, complementadas pelo repositório de dados da ferramenta desenvolvida pelo trabalho.

Para desenvolvimento da ferramenta *Struct* integrou-se as seguintes tecnologias otimizadoras: ferramentas CASE, programação visual, geradores de código, repositório e coordenador de repositório, tecnologia cliente-servidor, bibliotecas de classes e análise e projeto orientados a objetos. As demais tecnologias presentes na fundamentação teórica não foram incorporadas a ferramenta, porém serviram para ilustrar outras possibilidades de potencializar a integração. Pôde-se comprovar através da efetiva combinação das tecnologias citadas, um ganho substancial de qualidade e velocidade no desenvolvimento, conforme proposto por Martin (1994).

Através do presente trabalho comprovou-se ainda a importância do emprego das técnicas de compiladores no desenvolvimento de quaisquer ferramentas com propósito semelhante ao interpretador em questão, visto que as aplicações são inúmeras. Além disso, de forma mais ampla, constatou-se a necessidade da utilização de conhecimento científico na construção de programas. Apontando neste sentido, encontra-se a tecnologia otimizadora dos métodos matemáticos formais, descrita brevemente na seção 2.1.11.

O estudo relatado teve como efeito positivo, além de viabilizar a construção do interpretador, a agregação de elementos adicionais das técnicas de compiladores, não considerados essenciais à construção do interpretador em questão. Tal atitude visa futuras extensões do presente trabalho, além de terem servido ao propósito didático.

O trabalho empregou a estruturação das classes em camadas através da arquitetura tipo-classe, exposta por Ambler (1998, p. 88). Esta medida adicionou uma substancial capacidade de manutenibilidade e portabilidade ao software. Como a ferramenta CASE em questão é aplicada a um ambiente real, adotando um ciclo de vida dinâmico, pôde-se constatar na prática a facilidade de manutenção resultante do emprego da técnica de arquitetura tipo-classe, combinada com a aplicação integral das técnicas de orientação a objetos.

Durante a implementação do mecanismo de atualização do repositório de dados, foi identificada a dificuldade de persistência de objetos através de bancos de dados relacionais. Cada classe de um componente do repositório é instanciada a partir de valores de atributos lidos de uma tabela. Constatou-se alguns problemas decorrentes desta prática, dentre os quais: comprometimento da performance da aplicação, através do *overhead* gerado pela instanciação da classe; e perda de sincronismo entre o conteúdo do registro do banco de dados e a estrutura de dados instanciada em memória.

Ambler (1998, p. 318) propõe técnicas de superação da diferença entre o conceito relacional e de objetos, contudo, estas técnicas não puderam ser aplicadas integralmente na construção da camada de persistência da ferramenta CASE. Tal impossibilidade deve-se ao fato da ferramenta *Struct* manipular o banco de dados correspondente ao repositório de dados do ERwin, cuja estrutura não pôde ser modificada.

Atualmente, o *Struct* é aplicado ao ambiente real de produção da empresa Mult Sistemas Ltda. Através da ferramenta são geradas estruturas de código para um gerenciador de arquivos utilizado por produtos legados, além de classes especializadas da camada de persistência para novos produtos. Dessa forma, a utilização da ferramenta garante a compatibilidade, com relação à base de dados, das duas linhas de produtos.

Com relação a ferramenta de software resultado deste trabalho, o orientador do estágio na empresa, Sr. Ricardo de Freitas Becker, expõe o seguinte parecer: “A Mult Sistemas é uma empresa de desenvolvimento de sistemas aplicativos voltados para gestão de negócios. Nosso desafio diário é disponibilizar para os clientes soluções verdadeiras, que consigam superar todas as suas expectativas dentro do menor período de tempo possível. Mas a dinâmica empresarial atual impõe dificuldades crescentes a esse objetivo. Enquanto a competitividade de um mundo globalizado exige uma velocidade sempre maior e maior para as soluções de negócio, a constante oferta de novas tecnologias força uma situação onde muito rapidamente os sistemas caem em desinteresse comercial devido apenas ao modismo do momento. E é neste contexto que o *Struct* assume uma importância fundamental, representando o principal elo entre o passado e o futuro. A geração de código-fonte para diferentes tecnologias a partir do mesmo repositório é a chave para propiciar a continuidade de muito do conhecimento existente nos sistemas legados da empresa, que foram, por sua vez, construídos durante anos de evolução e aprimoramento constante. A produtividade alcançada com a eliminação das

tarefas redundantes é a chave para a redução do tempo de resposta para as novas necessidades. E, finalmente, as perspectivas abertas em função de ser um produto interno, construído com fundamentos tecnológicos avançados, garantem a flexibilidade da evolução e a aderência do sistema às necessidades e surpresas que ainda não conseguimos sequer imaginar.”

4.1 EXTENSÕES

Como proposta para trabalhos futuros, sugere-se inicialmente a incorporação da possibilidade de geração de código-fonte para as camadas de negócio e interface. Para tanto, pode ser empregada a seguinte seqüência de atividades:

a) desenvolver estruturas de negócio e interface generalizadas em uma biblioteca de classes;
b) agregá-las ao repositório de dados da ferramenta;
c) publicar comandos de acesso a tais estruturas na linguagem LCS;
d) codificar os *scripts* que especializem as referidas classes da biblioteca.

Como extensão à linguagem LCS sugere-se as seguintes implementações:

a) incorporar a manipulação de variáveis, através da instalação do identificador na tabela de símbolos já estruturada;
b) permitir a geração de código intermediário, viabilizando a criação de rotinas do usuário;
c) realizar estudos para otimização do interpretador, através da adoção de algumas técnicas apresentadas por Aho (1995).

Por fim, sugere-se a integração com o repositório de dados da ferramenta Rational Rose, no sentido de absorver as definições dos componentes do modelo de classes para geração do modelo de dados a partir do ERwin. Tal recurso pode ser embasado nas técnicas de persistência de objetos apresentadas por Ambler (1998, p. 297).

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, Alfred V; SETHI, Ravi; ULLMAN, Jeffrey D. **Conceitos de compiladores**. Rio de Janeiro: LTC – Livros Técnicos e Científicos, 1995.

AMBLER, Scott W. **Análise e projeto orientados a objeto**, volume II: seu guia para desenvolver sistemas robustos com tecnologia de objetos. Rio de Janeiro: Infobook, 1998.

FISHER, Alan S. **CASE**: utilização de ferramentas para desenvolvimento de software. Rio de Janeiro: Campus, 1990.

GANE, Chris. **CASE**: o relatório Gane. Rio de Janeiro: LTC – Livros Técnicos e Científicos, 1990.

HUNTER, Robin. **Compiladores sua concepção e programação em Pascal**. Lisboa: Presença, 1986.

MARTIN, James. **Princípios de análise e projeto baseados em objetos**. Rio de Janeiro: Campus, 1994.

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

POMPILHO, S. **Análise essencial.** Rio de Janeiro: Infobook, 1994.

PRESSMAN, Roger S. **Engenharia de software**. São Paulo: Makron Books, 1995.

RENAUD, Paul E. **Introdução aos sistemas cliente/servidor**: um guia prático para profissionais de sistemas. Rio de Janeiro: Infobook, 1994.

REZENDE, Denis Alcides. **Engenharia de software e sistemas de informações.** Rio de Janeiro: Brasport, 1999.

SCHIMT, Héldio. **Implementação de produto cartesiano e métodos de passagem de parâmetros no ambiente FURBOL**. 1999. 86 f. Trabalho de Conclusão de Curso

(Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

SILVA, José Roque Voltolini da. **Proposta de uma gramática para um analisador sintático de precedência de operadores.** Artigo não publicado, 2001. Universidade Regional de Blumenau.

ANEXO I – VISÃO GERAL DAS FUNÇÕES DA LINGUAGEM LCS

Funções de Manipulação da Memória

funcao MemInic (*aPosicao*: **tpdNumero**, *aValor*: **tpdDinamico**): **tpdNulo**;

Atribui o valor *aValor* à posição indexada por *aPosicao* da memória;

funcao MemVal (*aPosicao*: **tpdNumero**): **tpdNumero;**

Retorna o valor correspondente à posição indexada por *aPosicao* da memória;

Funções de Apresentação de Diálogos

funcao Aviso (*aMensagem*: **tpdCadeia**): **tpdNulo**;

Apresenta a cadeia *aMensagem* através de uma janela de aviso;

funcao Erro (*aMensagem*: **tpdCadeia**): **tpdNumero;**

Aborta a interpretação do *script* corrente e apresenta a cadeia *aMensagem* através de uma janela de erro;

funcao ConfirmarSimNao (*aMensagem*: **tpdCadeia**): **tpdLogico;**

Apresenta uma janela contendo *aMensagem* para seleção de uma opção lógica, retornando o valor correspondente;

Funções de Conversão de Tipos

funcao NumeroParaCadeia (*aNumero*: **tpdNumero**): **tpdCadeia**;

Converte *aNumero* para o tipo *cadeia*. Em caso de erro na conversão, aborta o processamento;

funcao DinamicoParaCadeia (*aDinamico*: **tpdDinamico**): **tpdCadeia;**

Converte *aDinamico* para o tipo *cadeia*. Em caso de erro na conversão, aborta o processamento;

funcao LogicoParaCadeia (*aLogico*: **tpdDinamico**): **tpdCadeia;**

Converte *aLogico* para o tipo *cadeia*. Em caso de erro na conversão, aborta o processamento;

Funções de Tratamento do Arquivo de Saída

funcao Saida (*aNome*: **tpdCadeia;** *aTipoGravacao*: **tpdNumero**): **tpdNulo**;

Assume *aNome* como o arquivo de saída corrente. aTipoGravação indica o procedimento a ser adotado quando o arquivo de saída já existir, da seguinte forma: 0, sobrepõe; 1, acrescenta ao final; e 2, solicita confirmação para atualizá-lo. Em caso de erro na criação do arquivo, aborta o processamento;

funcao pl: **tpdNulo;**

Envia um avanço de linha no arquivo de saída;

funcao nl (*aCadeia*: **tpdCadeia**): **tpdNulo;**

Transfere *aCadeia* para o arquivo de saída, seguido de um avanço de linha;

funcao ml (*aCadeia*: **tpdCadeia**): **tpdNulo;**

Transfere *aCadeia* para o arquivo de saída;

Funções de Ambiente

funcao AmbSaidaAtual: **tpdCadeia;**

Retorna o nome do arquivo de saída corrente. Caso não exista, retorna uma cadeia vazia;

funcao AmbData: **tpdCadeia;**

Retorna a data atual do sistema;

funcao AmbHora: **tpdCadeia;**

Retorna a hora atual do sistema;

funcao AmbScriptAtual: **tpdCadeia;**

Retorna o nome do arquivo de *script* atualmente interpretado;

funcao AmbDirScripts: **tpdCadeia;**

Retorna o nome do diretório padrão para arquivos de *script*;

funcao AmbDirGeracaoDOS: **tpdCadeia;**

Retorna o nome do diretório padrão para geração de arquivos de saída para a linha de produtos legados (ambiente DOS);

funcao AmbDirGeracaoWIN: **tpdCadeia;**

Retorna o nome do diretório padrão para geração de arquivos de saída da linha de produtos novos (ambiente Windows);

Funções de Manipulação de Cadeias

funcao StrMaiusculas (*aCadeia*: **tpdCadeia**): **tpdCadeia;**

Retorna uma cadeia contendo todas as letras de *aCadeia* convertidas para maiúsculas;

funcao StrMinusculas (*aCadeia*: **tpdCadeia**): **tpdCadeia;**

Retorna uma cadeia contendo todas as letras de *aCadeia* convertidas para minúsculas;

funcao StrRemoverEspacos (*aCadeia*: **tpdCadeia;** *aSentido:* **tpdNumero**): **tpdCadeia;**

Retorna uma cadeia sem os espaços em branco de *aCadeia*, na seguinte forma indicada por *aSentido*: 0, somente espaços à direita; 1, somente espaços à esquerda; ou 2, ambos os lados;

funcao StrAlinhar (*aCadeia*: **tpdCadeia;** *aSentido:* **tpdNumero;** *aTamanho:* **tpdNumero**): **tpdCadeia;**

Retorna uma cadeia resultante da concatenação de *aCadeia* com espaços em branco, visando criar um efeito de alinhamento, o qual pode ser obtido somente em fontes de tamanho fixo. *aSentido* indica o sentido do alinhamento, na forma: 0, direita; 1, esquerda; ou 2, centralizado. *aTamanho* define o tamanho máximo da cadeia gerada, ou seja, o limite para preenchimento com espaços;

funcao StrReplicar (*aCadeia*: **tpdCadeia;** *aVezes:* **tpdNumero)**: **tpdCadeia;**

Retorna uma cadeia resultado da concatenação de *aVezes* ocorrências de *aCadeia;*

funcao StrExtrair (*aCadeia*: **tpdCadeia;** *aDel1*, *aDel2:* **tpdCadeia**): **tpdCadeia;**

Retorna uma cadeia resultante da remoção de uma sub-cadeia de *aCadeia* delimitada por *aDel1*(inferior) e *aDel2* (superior). Caso as sub-cadeias limites não sejam encontradas, retorna *aCadeia* original;

funcao StrEliminarSet (*aCadeia*: **tpdCadeia)**: **tpdCadeia;**

Retorna uma cadeia resultante da eliminação dos caracteres componentes de *aCadeia*;

funcao StrNula (*aCadeia*: **tpdCadeia)**: **tpdLogico;**

Retorna um valor booleano indicando se a *aCadeia* é vazia, ou nula.

funcao StrChar (*aCadeia*: **tpdCadeia)**: **tpdCadeia;**

Retorna o código ASCIIZ do primeiro caractere de *aCadeia*. Caso *aCadeia* esteja vazia, retorna 0;

funcao StrExtrairDiretorio (*aNomeArquivo*: **tpdCadeia)**: **tpdCadeia;**

Retorna o diretório expresso por *aNomeArquivo*;

funcao StrExtrairExtensão (*aNomeArquivo*: **tpdCadeia)**: **tpdCadeia;**

Retorna a extensão a partir de *aNomeArquivo*;

funcao StrExtrairNome (*aNomeArquivo*: **tpdCadeia)**: **tpdCadeia;**

Retorna somente o nome do arquivo a partir de *aNomeArquivo*;

Funções de Manipulação do Repositório (Relação Parcial)

funcao Tabela_Primeira: **tpdNulo;**

Posiciona o apontador do objeto "tabela" no início da lista de selecionadas.

funcao Tabela_Proxima: **tpdNulo;**

Incrementa o apontador do objeto "tabela".

funcao Tabela_Anterior: **tpdNulo;**

Decrementa o apontador do objeto "tabela".

funcao Tabela_Ultima: **tpdNulo;**

Posiciona o apontador do objeto "tabela" no último índice da lista de selecionadas.

funcao Tabela_Posicionada: **tpdLogico;**

Retorna verdadeiro se o apontador indicar um objeto "tabela" válido.

funcao Tabela_NomeFisico: **tpdCadeia;**

Retorna o nome físico da tabela posicionada.

funcao Tabela_NomeLogico: **tpdCadeia;**

Retorna o nome lógico da tabela posicionada.

funcao Tabela_Sigla: **tpdCadeia;**

Retorna a sigla da tabela posicionada.

funcao Tabela_NumeroCampos: **tpdNumero;**

Retorna o número de campos associados a tabela posicionada.

ANEXO II – *SCRIPT* DE GERAÇÃO DA CLASSE DE PERSISTÊNCIA PARA O *INTERBASE*

```
{
   Sistema : Struct
   Recurso : Script de Especialização da classe TpClsIBOTable
             Instanciação dos atributos, publicação dos índices e
             tratamentos gerais

   Copyright (c) 2001 Mult Sistemas Ltda
}
Script TIBOTable0;

Inicio
  Tabela_PosicionarSistema;

  Saida (AmbDirGeracaoWIN +
         "\" + Sistema_Diretorio + "\" + "up0Tab" + StrMaiusculas
         (Tabela_Sigla) + ".pas", 0);

  {cabeçalho da unit}

  nl ("("+StrReplicar ("*", 80));
  nl ("  Sistema  : " + Diagrama_Nome);
  nl ("  Módulo   : " + Sistema_Nome);
  nl (" ");
  nl ("  Descrição: Classe Abstrata de Persistencia da Tabela " +
        Tabela_NomeLogico);
  nl ("             Descendente de TpClsIBOTable");
  pl;
  nl ("  Copyright (c) 2001 Mult Sistemas Ltda.");
  nl (StrReplicar ("*", 80)+")");

  {declarações da interface}

  nl ("unit up0Tab" + StrMaiusculas (Tabela_Sigla) + ";");
  nl (" ");
  nl ("interface");
  nl ("uses");
  nl ("  Classes, Db, upClsIBOTable;");
  nl (" ");

  nl ("type");
  PreservarBloco ("TiposDaInterface");
  pl;
  nl ("  TP0Tab"+Tabela_Sigla+" = class (TpClsIBOTable)");
  nl ("  private");
  pl;

  TipoBase_Definir ("Variáveis Internas Delphi");
```

```
nl ("    { procedimentos de consulta e atualização dos campos }");

Campo_Primeiro;
Enquanto (Campo_Posicionado) Faca
Inicio
  Campo_PosicionarDominio;
  ml ("    procedure   Set" + Campo_NomeFisico + " (a"+Campo_NomeFisico +
      " : ");
  nl (StrExtrair (Campo_SintaxeDefinicao, "[", "]") + ");");
  ml ("    function    Get" + Campo_NomeFisico + " : ");
  nl (StrExtrair (Campo_SintaxeDefinicao, "[", "]") + ";");
  Campo_Proximo;
Fim;

pl;
nl ("  protected");
pl;
nl ("    { definição dos campos fixos da tabela }");
nl ("    procedure   DefinirCampos; override;");
nl ("    procedure   DefinirComplementos; override;");
pl;
nl ("  public");
nl ("    { campos fixos da tabela }");

TipoBase_Definir ("Componentes TField Delphi");
Campo_Primeiro;
Enquanto (Campo_Posicionado) Faca
Inicio
  ml ("    " + StrAlinhar ("f" + Campo_NomeFisico, 0, 20) + " : ");
  ml (StrAlinhar (Campo_SintaxeDefinicao + ";", 0, 25));
  nl ("{" + Campo_NomeLogico + "}");
  Campo_Proximo;
Fim;

pl;
nl ("    { métodos de pesquisa de índices }");

Indice_Primeiro;
Enquanto (Indice_Posicionado) Faca
Inicio
  nl ("    function    Achar" + Indice_NomeFisico + " (const KeyValues:
     array of const): boolean;");
  Indice_Proximo;
Fim;

pl;
nl ("    { descendidos automaticamente }");
nl ("    constructor Create (aOwner : TComponent); override;");
nl ("    destructor  Destroy; override;");
pl;
nl ("    { cria os índices para a tabela conforme definido
           originalmente}");
nl ("    procedure   DefinirIndices; override;");
pl;
```

```
nl ("    { valores dos campos fixos da tabela }");

TipoBase_Definir ("Variáveis internas Delphi");
Campo_Primeiro;
Enquanto (Campo_Posicionado) Faca
Inicio
  Campo_PosicionarDominio;
  ml ("    property    ");
  ml (StrAlinhar (Campo_NomeFisico, 0, 20) + " : ");
  ml (StrAlinhar (StrExtrair (Campo_SintaxeDefinicao, "[", "]"), 0, 30));
  ml (StrAlinhar (" read Get" + Campo_NomeFisico, 0, 30));
  nl (" write Set" + Campo_NomeFisico + ";");
  Campo_Proximo;
Fim;

nl ("  end;");
pl;

{declarações da implementation}

nl ("implementation");
nl ("uses");
nl ("  SysUtils;");
pl;
nl ("constructor Tp0Tab"+Tabela_Sigla+".Create (aOwner : TComponent);");
nl ("begin");
nl ("  inherited Create (aOwner);");
Se (StrNula (Sistema_DBAlias)) Entao
  nl ("  DatabaseName := '"+Sistema_DBAlias+"';");
nl ("  TableName := '"+Tabela_NomeFisico+"';");
nl ("end;");
pl;
nl ("destructor Tp0Tab"+Tabela_Sigla+".Destroy;");
nl ("begin");
nl ("  inherited Destroy;");
nl ("end;");
pl;
nl ("procedure TP0Tab"+Tabela_Sigla+".DefinirCampos;");
nl ("begin");

TipoBase_Definir ("Componentes TField Delphi");
Campo_Primeiro;
Enquanto (Campo_Posicionado) Faca
Inicio
  nl ("  if not assigned (f"+Campo_NomeFisico+") then");
  nl ("  begin");
  nl ("    f"+Campo_NomeFisico+" := "+Campo_SintaxeDefinicao+".Create
      (Self);");
  nl ("    with f"+Campo_NomeFisico+" do");
  nl ("    begin");
  nl ("      FieldName := '"+Campo_NomeFisico+"';");
  nl ("      DataSet := Self;");
  nl ("      DisplayLabel := '"+Campo_NomeLogico+"';");
  Se (Campo_TamanhoApresentacao <> 0) Entao
    nl ("      DisplayWidth := "+NumeroParaCadeia
```

```
        Campo_TamanhoApresentacao)+";");
  Se (Campo_Obrigatorio) Entao
    nl ("       Required := TRUE;")
  Senao
    nl ("       Required := FALSE;");
  Se (Campo_Visivel) Entao
    nl ("       Visible := TRUE;")
  Senao
    nl ("       Visible := FALSE;");
  nl ("    end;");
  nl ("  end;");
  pl;
  Campo_Proximo;
Fim;

PreservarBloco ("DefinirCampos");

nl ("end;");
pl;
nl ("procedure TP0Tab"+Tabela_Sigla+".DefinirComplementos;");
nl ("begin");
nl ("  sNmLogico := '"+Tabela_NomeLogico+"';");
nl ("  sSigla    := '"+Tabela_Sigla+"';");
nl ("end;");
pl;
nl ("("+StrReplicar ("*", 80));
nl (StrAlinhar ("PROCEDIMENTOS DE CONSULTA E GRAVAÇÃO DE CAMPOS", 2,
    80));
nl (StrReplicar ("*", 80)+")");

TipoBase_Definir ("Variáveis internas Delphi");

Campo_Primeiro;
Enquanto (Campo_Posicionado) Faca
Inicio
  Campo_PosicionarDominio;
  pl;
  ml ("procedure Tp0Tab"+Tabela_Sigla+".Set"+Campo_NomeFisico+
      " (a"+Campo_NomeFisico+": ");
  nl (StrExtrair (Campo_SintaxeDefinicao, "[", "]")+");");
  nl ("begin");
  nl ("  if assigned (f"+Campo_NomeFisico+") then");
  nl ("    f"+Campo_NomeFisico+".value := a"+Campo_NomeFisico+";");
  nl ("end;");
  pl;
  ml ("function TP0Tab"+Tabela_Sigla+".Get"+Campo_NomeFisico+": ");
  nl (StrExtrair (Campo_SintaxeDefinicao, "[", "]")+";");
  nl ("begin");
  nl ("  if assigned (f"+Campo_NomeFisico+") then");
  nl ("    result := f"+Campo_NomeFisico+".value");
  nl ("  else");
  nl ("    result := "+Campo_SintaxeNulo+";");
  nl ("end;");
  Campo_Proximo;
Fim;
```

```
pl;
nl ("("+StrReplicar ("*", 80));
nl (StrAlinhar ("MÉTODOS DE DEFINIÇÃO E PESQUISA DE ÍNDICES", 2, 80));
nl (StrReplicar ("*", 80)+")");
pl;

{métodos gerais de definição de índices}

nl ("procedure Tp0Tab"+Tabela_Sigla+".DefinirIndices;");
nl ("begin");
nl ("  with IndexDefs do");
nl ("  begin");
nl ("    Clear;");

Indice_Primeiro;
Enquanto (Indice_Posicionado) Faca
Inicio
  Se (StrNula (Indice_Numero)) Entao
  Inicio
    ml ("    Add ('"+Indice_NomeFisico+"', '");
    CampoIndice_Primeiro;
    Enquanto (CampoIndice_Posicionado) Faca
      Inicio
      ml (CampoIndice_NomeFisico);
      CampoIndice_Proximo;
      Se (CampoIndice_Posicionado) Entao
        ml (";");
    Fim;
    ml ("',");
    Se (Indice_Tipo = "PK") Entao
      nl (" [ixPrimary, ixUnique]);")
    Senao
      Se (Indice_Tipo = "AK") Entao
        nl (" []);")
      Senao
        nl (" [ixCaseInsensitive]);");
  Fim;
  Indice_Proximo;
Fim;
nl ("  end;");

nl ("end;");

Indice_Primeiro;
Enquanto (Indice_Posicionado) Faca
Inicio
  Se (StrNula (Indice_Numero)) Entao
    Inicio
    pl;
    nl ("function Tp0Tab"+Tabela_Sigla+".Achar"+Indice_NomeFisico+
      " (const KeyValues: array of const): Boolean;");
    nl ("begin");
    nl ("  IndexName := '"+Indice_NomeFisico+"';");
    nl ("  result := FindKey (KeyValues);");
```

```
      nl ("end;");
    Fim;
    Indice_Proximo;
  Fim;

  pl;
  nl ("end.");

Fim.
```

ANEXO III – CÓDIGO-FONTE RESULTANTE DA EXECUÇÃO DO *SCRIPT*

```
(*************************************************************************
  Sistema  : Banesfácil 3.0
  Módulo   : SIP

  Descrição: Classe Abstrata de Persistencia da Tabela Grupos de
             Beneficiários
             Descendente de TpClsIBOTable

  Copyright (c) 2001 Mult Sistemas Ltda.
*************************************************************************)
unit up0TabGRB;

interface
uses
  Classes, Db, upClsIBOTable;

type

  TP0TabGRB = class (TpClsIBOTable)
  private

    { procedimentos de consulta e atualização dos campos }
    procedure   SetCdIdf (aCdIdf : integer);
    function    GetCdIdf : integer;
    procedure   SetCdGrb (aCdGrb : integer);
    function    GetCdGrb : integer;
    procedure   SetNmGrb (aNmGrb : string);
    function    GetNmGrb : string;

  protected

    { definição dos campos fixos da tabela }
    procedure   DefinirCampos; override;
    procedure   DefinirComplementos; override;

  public
    { campos fixos da tabela }
    fCdIdf              : TIntegerField;      {Código da Identificação}
    fCdGrb              : TIntegerField;      {Código do Grupo de
                                               Beneficiários}
    fNmGrb              : TStringField;       {Nome do Grupo}
```

```
    { métodos de pesquisa de índices }
    function    AcharpkGRB (const KeyValues: array of const): boolean;
    function    AcharfkGRBIdent (const KeyValues: array of const): boolean;

    { descendidos automaticamente }
    constructor Create (aOwner : TComponent); override;
    destructor  Destroy; override;

    { cria os índices para a tabela conforme definido originalmente }
    procedure   DefinirIndices; override;

    { valores dos campos fixos da tabela }
    property    CdIdf               : integer
                        read GetCdIdf               write SetCdIdf;
    property    CdGrb               : integer
                        read GetCdGrb               write SetCdGrb;
    property    NmGrb               : string
                        read GetNmGrb               write SetNmGrb;
  end;

implementation
uses
  SysUtils;

constructor Tp0TabGRB.Create (aOwner : TComponent);
begin
  inherited Create (aOwner);
  TableName := 'CGRB';
end;

destructor Tp0TabGRB.Destroy;
begin
  inherited Destroy;
end;

procedure TP0TabGRB.DefinirCampos;
begin
  if not assigned (fCdIdf) then
  begin
     fCdIdf := TIntegerField.Create (Self);
     with fCdIdf do
     begin
       FieldName := 'CdIdf';
       DataSet := Self;
       DisplayLabel := 'Código da Identificação';
       Required := FALSE;
```

```
      Visible := TRUE;
    end;
  end;

  if not assigned (fCdGrb) then
  begin
     fCdGrb := TIntegerField.Create (Self);
     with fCdGrb do
     begin
       FieldName := 'CdGrb';
       DataSet := Self;
       DisplayLabel := 'Código do Grupo de Beneficiários';
       Required := FALSE;
       Visible := TRUE;
    end;
  end;

  if not assigned (fNmGrb) then
  begin
     fNmGrb := TStringField.Create (Self);
     with fNmGrb do
     begin
       FieldName := 'NmGrb';
       DataSet := Self;
       DisplayLabel := 'Nome do Grupo';
       Required := FALSE;
       Visible := TRUE;
    end;
  end;

end;

procedure TP0TabGRB.DefinirComplementos;
begin
  sNmLogico := 'Grupos de Beneficiários';
  sSigla    := 'GRB';
end;

(*************************************************************************
                 PROCEDIMENTOS DE CONSULTA E GRAVAÇÃO DE CAMPOS
*************************************************************************)

procedure Tp0TabGRB.SetCdIdf (aCdIdf: integer);
begin
  if assigned (fCdIdf) then
    fCdIdf.value := aCdIdf;
end;
```

```
function TP0TabGRB.GetCdIdf: integer;
begin
  if assigned (fCdIdf) then
    result := fCdIdf.value
  else
    result := 0;
end;

procedure Tp0TabGRB.SetCdGrb (aCdGrb: integer);
begin
  if assigned (fCdGrb) then
    fCdGrb.value := aCdGrb;
end;

function TP0TabGRB.GetCdGrb: integer;
begin
  if assigned (fCdGrb) then
    result := fCdGrb.value
  else
    result := 0;
end;

procedure Tp0TabGRB.SetNmGrb (aNmGrb: string);
begin
  if assigned (fNmGrb) then
    fNmGrb.value := aNmGrb;
end;

function TP0TabGRB.GetNmGrb: string;
begin
  if assigned (fNmGrb) then
    result := fNmGrb.value
  else
    result := '';
end;

(*************************************************************************
                    MÉTODOS DE DEFINIÇÃO E PESQUISA DE ÍNDICES
*************************************************************************)

procedure Tp0TabGRB.DefinirIndices;
begin
  with IndexDefs do
  begin
    Clear;
    Add ('pkGRB', 'CdIdf;CdGrb', [ixPrimary, ixUnique]);
```

```
    Add ('fkGRBIdent', 'CdIdf', [ixCaseInsensitive]);
  end;
end;

function Tp0TabGRB.AcharpkGRB (const KeyValues: array of const): Boolean;
begin
  IndexName := 'pkGRB';
  result := FindKey (KeyValues);
end;

function Tp0TabGRB.AcharfkGRBIdent (const KeyValues: array of const):
   Boolean;
begin
  IndexName := 'fkGRBIdent';
  result := FindKey (KeyValues);
end;

end.
```

ANEXO IV – *SCRIPT* DE GERAÇÃO DE DOCUMENTAÇÃO PARA TABELAS

```
{
   Sistema : Struct
   Recurso : Script de Documentação de Tabelas
   Copyright (c) 2001 Mult Sistemas Ltda
}
Script Documentacao;

Inicio
  Diagrama_Primeiro;

  Enquanto (Diagrama_Posicionado) Faca
  Inicio
    Se (ConfirmarSimNao ("Deseja gerar documentação para o diagrama "
                      +Diagrama_Nome+" ?")) Entao
    Inicio
      Saida (StrExtrairDiretorio (AmbSaidaAtual) + "\" +
             StrRemoverEspacos (Diagrama_Nome, 2) + ".txt", 0);

      nl (StrAlinhar ("DOCUMENTAÇÃO DO SISTEMA "+Diagrama_Nome, 2, 100));
      nl (StrAlinhar ("MULT SISTEMAS LTDA", 2, 100));
      nl (StrReplicar ("=", 100));

      MemInic (0, 0);
      Tabela_Primeira;
      Enquanto (Tabela_Posicionada) Faca
      Inicio

        MemInic (0, MemVal(0) + 1);    {incrementa contador de tabelas}

        pl;
        nl (NumeroParaCadeia (MemVal (0))+" - "+Tabela_NomeLogico);
        pl;
        nl ("  a) D A D O S   G E R A I S");
        pl;
        nl (StrAlinhar ("     Nome Físico", 0, 35)+": "+Tabela_NomeFisico);
        nl (StrAlinhar ("     Sigla", 0, 35)+": "+Tabela_Sigla);
        nl (StrAlinhar ("     Número de Campos", 0, 35)+": "
            + NumeroParaCadeia (Tabela_NumeroCampos));
        nl (StrAlinhar ("     Número de Índices", 0, 35)+": "
            + NumeroParaCadeia (Tabela_NumeroIndices));
        nl (StrAlinhar ("     Número de Relacionamentos", 0, 35)+": "
```

```
   + NumeroParaCadeia (Tabela_NumeroRelactos));
pl;

{leitura dos campos}

nl ("  b) C A M P O S");
pl;
ml (StrReplicar (" ", 3));
ml (StrAlinhar (" Nome Físico", 0, 20));
ml (StrAlinhar (" Nome Lógico", 0, 40));
ml (StrAlinhar (" Domínio", 0, 20));
nl (StrAlinhar (" Tipo Pascal", 0, 20));
ml (StrReplicar (" ", 4));
nl (StrReplicar ("-", 100));

TipoBase_Definir ("Variáveis internas Delphi");

Campo_Primeiro;
Enquanto (Campo_Posicionado) Faca
Inicio
  Campo_PosicionarDominio;
  ml (StrReplicar (" ", 4));
  ml (StrAlinhar (Campo_NomeFisico, 0, 20));
  ml (StrAlinhar (Campo_NomeLogico, 0, 40));
  ml (StrAlinhar (Dominio_Nome, 0, 20));
  nl (StrAlinhar (Campo_SintaxeDefinicao, 0, 20));
  Campo_Proximo;
Fim;

{leitura dos índices}

pl;
nl ("  c) I N D I C E S");
pl;
ml (StrReplicar (" ", 3));
ml (StrAlinhar (" Nome Físico", 0, 20));
ml (StrAlinhar (" Nome Lógico", 0, 40));
nl (StrAlinhar (" Tipo", 0, 20));
ml (StrReplicar (" ", 4));
nl (StrReplicar ("-", 100));

Indice_Primeiro;
Enquanto (Indice_Posicionado) Faca
Inicio
  ml (StrReplicar (" ", 4));
  ml (StrAlinhar (Indice_NomeFisico, 0, 20));
  ml (StrAlinhar (Indice_NomeLogico, 0, 40));
```

```
          Se (Indice_Unico) Entao
            nl (StrAlinhar ("PK", 0, 20))
          Senao
            nl (StrAlinhar ("AK", 0, 20));
          Indice_Proximo;
        Fim;
        pl;
        pl;

        Tabela_Proxima;
      Fim;

      pl;
      nl ("Tabela(s) relacionada(s): "+NumeroParaCadeia (MemVal(0)));
      nl (StrReplicar ("=", 100));
      nl ("Gerado em "+AmbData+ " às "+AmbHora +" por Struct");
      pl;
      pl;
      pl;
    Fim;

    Diagrama_Proximo;
  Fim;

Fim.
```

ANEXO V – ARQUIVO DE DOCUMENTAÇÃO RESULTANTE DA EXECUÇÃO DO *SCRIPT*

DOCUMENTAÇÃO DO SISTEMA Banesfácil 3.0

MULT SISTEMAS LTDA

==========

1 - Beneficiários

a) D A D O S G E R A I S

Nome Físico	: CBEN
Sigla	: BEN
Número de Campos	: 15
Número de Índices	: 3
Número de Relacionamentos	: 3

b) C A M P O S

Nome Físico	Nome Lógico	Domínio	Tipo Pascal
CdIdf	Código da Identificação	dmCodigo	integer
CdBen	Código do Beneficiário	dmInteger	integer
CdGrb	Código do Grupo de Beneficiários	dmInteger	integer
Nome	Nome	dmString	string
Matricula	Número de Matrícula	dmString	string
Endereco	Endereço	dmString	string
Bairro	Bairro	dmString	string
Cidade	Cidade	dmString	string
Estado	Estado	dmString	string
Cep	Cep	dmString	string
Telefone	Telefone	dmString	string
NrCta	Número da Conta	dmString	string
CPF	C.P.F.	dmString	string
FormaPag	Forma de Pagamento	dmSmallint	integer
CdAge	Código da Agência	dmString	string

c) I N D I C E S

Nome Físico	Nome Lógico	Tipo
pkBEN	pkCBEN	PK
fkBENIdent	Foreign Key 74	AK
XIF80CBEN	Foreign Key 80	AK

2 - Sacados

a) D A D O S G E R A I S

Nome Físico	: CSAC
Sigla	: SAC
Número de Campos	: 15
Número de Índices	: 3
Número de Relacionamentos	: 5

b) C A M P O S

Nome Físico	Nome Lógico	Domínio	Tipo Pascal
CdIdf	Código da Identificação	dmCodigo	integer
CdSac	Código	dmCodigo	integer
CdGrs	Código do Grupo de Sacados	dmCodigo	integer
Fantasia	Fantasia	dmFantasia	string
Nome	Razão Social	dmRazaoSocial	string
Endereco	Endereço	dmEndereco	string
Bairro	Bairro	dmBairro	string
Cidade	Cidade	dmCidade	string
Estado	Estado	dmEstado	string
TipoCGC	Tipo CGC x CPF	dmCaracter	string
CGC	CNPJ(CGC)/CPF	dmCGC	string
CEP	CEP	dmCep	string
CaixaPostal	Caixa Postal	dmCaixaPostal	string
Telefone	Telefone	dmFone	string
Fax	Fax	dmFone	string

c) I N D I C E S

Nome Físico	Nome Lógico	Tipo
pkSAC	pkCSAC	PK
fkSACIdent	Foreign Key 53	AK
fkSACIdentGrp	Foreign Key 76	AK

Tabela(s) relacionada(s): 2

==========

Gerado em 20/07/01 às 03:06:17 por Struct