UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

FERRAMENTA DE ANÁLISE DE ESTRUTURAS BÁSICAS EM LINGUAGEM PASCAL PARA FORNECIMENTO DE MÉTRICAS

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

ROQUE CÉSAR POSSAMAI

BLUMENAU, JUNHO/2000

2000/1-61

FERRAMENTA DE ANÁLISE DE ESTRUTURAS BÁSICAS EM LINGUAGEM PASCAL PARA FORNECIMENTO DE MÉTRICAS

ROQUE CÉSAR POSSAMAI

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

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. Everaldo Artur Grahl — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. Everaldo Artur Grahl

_____

Prof. Maurício Capobianco Lopes

_____

Prof. Wilson Pedro Carli

AGRADECIMENTOS

Ao meu orientador, professor Everaldo Artur Grahl, pelo apoio, empenho e dedicação dispensados, pois sem seu auxílio, não poderia concluir este trabalho.

Aos meus amigos que de uma forma ou de outra sempre me incentivaram e me animaram a seguir adiante nessa difícil caminhada.

À toda a minha família que sempre me incentivou a continuar os estudos e seguir em frente.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Este trabalho tem como objetivo principal a implementação de um protótipo de ferramenta de análise de programas fontes em Pascal, que fornecerá informações sobre métricas de software. As métricas da ciência do software e complexidade ciclomática são alguns dos resultados obtidos pela ferramenta. Adicionalmente, são exibidos fluxogramas e grafos de determinadas partes do sistema analisado.

ABSTRACT

The mainly objective of this work is the development of a prototype of source code analisys system that give informations about software metrics. As results of this prototype, are the metrics of Science Software and Ciclomatic Complexity. This prototype also shows fluxograms and graphs of analised system.

1 INTRODUÇÃO

1.1 ORIGEM

As organizações que utilizam computadores estão começando a adotar, como prioridade, a qualidade do produto de software e a performance e produtividade do time de desenvolvimento. Para que isto seja possível é necessário que sejam adotadas técnicas de gerenciamento e projeto de software. A engenharia de software é o ramo da computação que estuda as formas de gerenciamento, projeto, criação e manutenção de software, sendo que o conceito de qualidade em software está totalmente associado a esta área [MÖL1993].

Diferentes técnicas podem ser aplicadas para minimizar o problema de gerenciamento do desenvolvimento de software. Uma das maneiras pelas quais a engenharia de software consegue efetuar este gerenciamento é através da utilização de métodos quantitativos (métricas de software). Segundo Fernandes [FER1995], toda e qualquer engenharia é fundamentada em medições que são a base para qualquer ciência. Para tratar o software sob uma abordagem de engenharia é preciso entender as características de sua dimensão e a importância da medição para a sua gestão.

A aplicação de métricas de software é uma excelente ferramenta para uma eficiente gerência do desenvolvimento de software e manutenção de processos [MÖL1993].

Ferramentas que mostrem informações sobre o software que está sendo desenvolvido são particularmente úteis, pois estas podem analisar a implementação feita e fornecer relatórios mostrando as principais deficiências existentes no sistema. Eventualmente estas ferramentas fornecem, além de planilhas de custo e estimativas de conclusão do projeto, resultados visuais ao desenvolvedor, como um grafo de controle do fluxo de dados ou um grafo que represente a estruturação do código-fonte, exibindo todas as estruturas condicionais do software. Baseado nas informações que são disponibilizadas por estas ferramentas o desenvolvedor pode mais facilmente encontrar alguma irregularidade no software ou saber em qual aspecto este software está deficiente.

Um dos pontos abrangidos pela área de engenharia de software é relativo à métricas de software, ou seja, todas as informações que podem-se obter *medindo-se* um sistema ou um programa. Se estas informações forem disponibilizadas de maneira apropriada poderão ser

extremamente úteis para o entendimento e validação de um sistema ou programa. Algumas destas informações permitem prever quais as partes de um sistema serão afetadas quando uma determinada parte do programa for alterada.

Neste sentido, a linguagem Pascal foi escolhido como o alvo deste trabalho, devido a mesma ser conhecida como uma boa ferramenta de desenvolvimento de software e por possuir uma linguagem de programação simples e robusta, permitindo o desenvolvimento de sistemas de forma fácil e rápida.

Como ferramenta de desenvolvimento optou-se pelo ambiente Delphi e pela ferramenta de especificação foi utilizado Rational Rose, devido à sua facilidade na especificação através da abordagem UML.

1.2 OBJETIVOS

O principal objetivo deste trabalho será a especificação e implementação de um protótipo de ferramenta de análise de programas fontes em Pascal que fornecerá informações sobre métricas de software. Entre as informações que são disponibilizadas pelo protótipo pode-se citar:

a) lista de tipos, constantes, variáveis, subprogramas e outras informações do programa;
b) disponibilização de duas métricas sobre o código-fonte analisado;
c) grafo representativo dos caminhos lógicos do programa;
d) grafo representativo da utilização de subprogramas e outras entidades que são utilizadas pelo sistema.

1.3 ORGANIZAÇÃO DO TEXTO

No primeiro capítulo encontra-se a introdução juntamente com a motivação e os objetivos deste trabalho.

No segundo capítulo encontra-se uma breve explanação sobre a utilização de métricas de software, juntamente com suas variações.

O terceiro capítulo está focado nas métricas orientadas à código-fonte, juntamente com a explanação de algumas métricas desta categoria e um exemplo simples de sua utilização.

No quarto capítulo é feita uma pequena descrição da linguagem Pascal, e encontra-se também a especificação do protótipo juntamente com um exemplo básico de funcionamento do mesmo.

No quinto capítulo tem-se as conclusões obtidas com a execução deste trabalho e uma relato das limitações do protótipo e sugestões para implementações futuras.

2 MÉTRICAS DE SOFTWARE

Segundo [MCD1993], métrica é uma medição que caracteriza alguns aspectos de uma entidade, seja ela um produto, processo ou companhia.

As disciplinas das ciências naturais e das engenharias formam o tronco principal do conhecimento, transformando observações empíricas em teorias formais, expressando-as em notações matemáticas. Para isto, elas utilizam-se da teoria da mensuração e práticas metrológicas, as quais são utilizadas não somente na ciência mas também na vida diária [MCD1993].

Segundo [SHE1993], a teoria da mensuração pode ser definida como o processo de associação de números ou outras entidade matemáticas, para descrever atributos empíricos de um produto ou evento, através de regras preestabelecidas.

Segundo McDermic [MCD1993], as medições têm sido a base de conduta das pesquisas dos cientistas empíricos. Estas, por sua vez, possuem uma grande importância, pois seus resultados obtidos podem ser catalogados e utilizados para a obtenção de novas verdades, baseadas em leis genéricas e métodos fidedignos provindos do estudo destes resultados. Desta forma, pode-se afirmar que a mensuração é o *coração* do *método científico da observação*, permitido a criação, defesa e rejeição de hipóteses.

A mensuração é, desta forma, parte importante em todos os aspectos da vida diária: comércio, finanças, medicina, comunicações, gerenciamento, educação, leis e demais áreas do conhecimento. As medições são necessárias em todas as fases do ciclo de vida dos produtos e processos, desde a captação das necessidades dos clientes, passando também por todas as fases do ciclo de vida do produto como: especificação, projeto, produção, operação e manutenção. Elas são utilizadas na classificação, caracterização e avaliação dos sistemas e componentes atuais e futuros, formulação precisa de problemas, experimentação de soluções variadas, avaliação de projetos alternativos, garantia da qualidade dos produtos e serviços e a verificação dos custos sobre os mesmos.

Segundo [FER1995], as métricas de software podem ser definidas como métodos de determinar quantitativamente a extensão em que o processo e o produto de software têm certos atributos. Isto inclui uma fórmula para determinar o valor da métrica como também sua

forma de apresentação e as diretrizes de utilização e interpretação de resultados obtidos no contexto do ambiente de desenvolvimento de software.

Segundo [MCD1993], a existência dos itens relacionados a seguir, é necessária em uma aplicação de um sistema de mensuração:

a) descritores: que caracterizam uma entidade existente;
b) prescritores: que definem as propriedades que uma entidade futura deve reunir uma vez implementada;
c) preditores: que são utilizados durante o período de projeto para estimativas e cálculos de propriedades futuras de uma entidade a ser implementada.

2.1 DEFINIÇÃO DE MENSURAÇÃO

Segundo McDermic [MCD1993] a mensuração se aplica como “as operações sobre um objeto que determinam o valor de uma quantidade”.

As operações de mensuração envolvem a coleta e catalogação dos dados observados; no entanto, esta atividade constitui somente parte do processo. Mensuração também engloba a identificação das classes das entidades para as quais a medição se relaciona, definindo precisamente as propriedades selecionadas e descrevendo-as objetiva e repetidamente. Os resultados da medição devem ser independentes dos pontos de vista ou preferências do medidor e não podem ser corrompidos por circunstâncias incidentais, influenciando o resultado.

A definição de mensuração adotada por este trabalho é a mesma de [MCD1993]: “Mensuração é o processo de codificação objetiva e empírica, de algumas propriedades de uma selecionada classe de entidades, em um sistema de símbolos formais, tão bem quanto a descrição das mesmas”.

2.2 A ORGANIZAÇÃO DAS ÁREAS DA MENSURAÇÃO

Metrologia é o campo do conhecimento que preocupa-se com as mensurações [MCD1993].

A metrologia abrange todo o conhecimento teórico e programático acerca das mensurações. A matriz da tabela 1 é uma amostra classificada e organizada deste vasto domínio.

TABELA 1 - Estrutura de representação da abrangência da metrologia

Aspectos/ Esferas	Metrologia Quantitativa	Metrologia Orientada à Objeto	Metrologia Orientada à Assunto	Metrologia Geral
Metrologia Teórica				
Metodologia da Metrologia				
Instrumentação da Metodologia da Metrologia				
Metrologia Legal (Incluindo padrões e qualidade)				

FONTE: McDermic, 1993: 12/4

As linhas da tabela 1 indicam os seguintes aspectos da metrologia, segundo [MCD1993]:

a) Metrologia teórica:
   - utilização da filosofia e método da ciência empírica;
   - definição das fundamentações lógicas da mensuração, incluindo classificação e conceitualização;
   - formação de hipóteses e modelos;
   - definição de testabilidade;
   - conceitualização da teoria geral da experimentação, formação e interpretação de escalas dos dados como evidências;
   - definição da teoria do erro;

b) metodologia da metrologia:
   - definição de estratégias de mensuração;
   - organização, execução e interpretação dos resultados;

c) tecnologia da metrologia: projeto ou escolha racional dos instrumentos de medida ou sistemas de mensuração, para percepção, processamento, armazenamento e exibição dos dados medidos;

d) metrologia legal: sistemas ou procedimentos, regulamentações e leis sobre o desenvolvimento e manutenção das unidades e padrões de medidas, conjuntamente com padrões e outras estruturas de suporte para monitoramento e manutenção destes padrões.

Do mesmo modo, as colunas da tabela representam *áreas de interesse* da metrologia:

a) metrologia quantitativa (orientada à propriedades): determinação do valor de um parâmetro em particular de alguma entidade;
b) metrologia orientada à objetos: caracterização de uma entidade em termos dos parâmetros do modelo selecionado;
c) metrologia orientada à assunto: a caracterização paramétrica das classes de todas as entidades com um domínio específico da ciência, engenharia ou outra disciplina;
d) metrologia geral: o estudo sem referência a qualquer domínio em particular, onde são formados os conceitos mensuráveis, entidades caracterizadas parametricamente, sistemas metrológicos desenvolvidos, etc.

2.3 O PROCESSO DE MENSURAÇÃO

Segundo [MCD1993], o processo de mensuração possui dois aspectos principais: gerenciamento e execução. O gerenciamento da medição é dividido em 3 partes interligadas que são:

a) planejamento da medição: inclui os planos para criar e conduzir a medição tanto como planejar a validação e garantia da qualidade dos resultados;
b) organização da execução da medição: cobrem problemas como a provisão de recursos humanos e treinamento de pessoal, a projeção e obtenção de equipamentos e instrumentos, as acomodações e o ambiente de medição, o projeto de procedimentos e medição, e, a provisão de estruturas administrativas apropriadas com suporte legal;
c) controle e supervisão da execução da medição: estende-se ao gerenciamento de recursos e pessoal utilizados na condução da mensuração, garantia que a medição praticada está de acordo com os planos e revisão da estrutura organizacional para garantir que estes planos serão adequados.

2.4 QUALIDADE DA MENSURAÇÃO

A definição de medição admite a codificação das observações em números e outros símbolos de sistema. A codificação numérica das observações pode ter um grande valor em contextos específicos, no entanto, a relação casual de números para objetos raramente satisfaz as demandas de garantia de integridade e validade universal – qualidade – da medição.

Descrições quantitativas e intuitivas das entidades podem ser inválidas ou abertas à má interpretação: elas podem complicar tanto quanto facilitar a compreensão, retardar tanto quanto promover o desenvolvimento de padrões comerciais e industriais, podendo haver sérios perigos de extravio de informações confidenciais pelos administradores que agem sobre estas medições.

Segundo [MCD1993], a regra da teoria da informação é conceitualizar e formalizar algumas das importantes características desejáveis da mensuração e prover métodos que suportem sua realização:

a) a medição deve ser válida quanto aos propósitos tencionados e deve conceder resultados válidos referentes ao seu propósito. A validade da mensuração pode ser promovida através de uma boa execução e gerenciamento, mas somente poderá ser confirmada empiricamente, logo que os dados estiverem disponíveis;
b) a codificação numérica das observações pela medição não deve ser arbitrária. O propósito da medição reflete em um julgamento subjetivo, mas seus resultados devem refletir unicamente as propriedades em observação;
c) a medição não pode ser obtusa. O ato de medir não deve interferir nas propriedades observadas;
d) a medição deverá ser exata. O erro ou incerteza na medição faz com que a mesma não tenha uma base sólida, podendo ser invalidada;
e) a medição deve ser reproduzível. Ela deve ser estável e objetiva, tal que os mesmos resultados deverão ser obtidos quando a mesma medição for feita com diferentes pessoas, utilizando, possivelmente, diferentes acessórios;
f) a medição deverá ser efetiva na utilização dos recursos. A medição deverá estar focada em prover as informações requisitadas, sem o desperdício de recursos em itens não relevantes. Deverá ser possível conduzir a medição sem a imposição de excessivas perdas de tempo.

g) os resultados medidos deverão ter integridade. Deverá ser possível confirmar e demostrar a imparcialidade da medição, pois caso contrário, não será fácil tirar as conclusões necessárias.

2.5 A ORIGEM DAS MÉTRICAS DE SOFTWARE

Em meados de 1970, surgiram quatro tendências tecnológicas das métricas de software, sendo que estas evoluíram para as métricas atuais [MÖL1993].

As quatro tendências são:

a) Métricas da Complexidade do Código: surgiram em meados de 1970. As métricas de código eram fáceis de obter, pois elas podiam ser extraídas por meios automatizados. Como exemplo deste tipo de métrica pode-se citar a métrica da complexidade ciclomática de McCabe;

b) Estimativa de Custo do Projeto de Software: estas técnicas foram desenvolvidas em meados dos anos 1970, para estimar o esforço de programação que era necessário para desenvolver um software. Estas estimativas eram baseadas nos números de linhas de código necessárias para a implementação entre outros fatores. Como exemplo pode-se citar o modelo COCOMO (Constructive Cost Model);

c) Garantia da Qualidade de Software: as técnicas da garantia da qualidade de software foram aperfeiçoadas significativamente ao fim dos anos 70 e início dos anos 80. A ênfase é dada para as informações que não estão disponíveis durante o ciclo de vida do software;

d) Processo de Desenvolvimento de Software: A medida que os projetos de software tornaram-se maiores e mais complexos, tornou-se necessário o desenvolvimento de novos processos para o controle dos mesmos. Estes processos incluem a definição do ciclo de vida do projeto; com maior ênfase no gerenciamento e controle dos recursos do mesmo.

Segundo Möller [MÖL1993], nos anos 80, estas quatro tendências tecnológicas da engenharia de software impulsionaram o aperfeiçoamento do gerenciamento de projetos de software através de métodos quantitativos. Os pioneiros nesta área iniciaram a aplicação de métricas com o propósito de aperfeiçoar o processo de desenvolvimento de software. A aplicação de métricas para o aperfeiçoamento do processo de desenvolvimento de software foi

também um complemento utilizado em muitos programas de aperfeiçoamento de qualidade implantados na época em várias empresas.

Atualmente, a prática de métricas de software utiliza indicadores globais que discernem entre o aperfeiçoamento do desenvolvimento de software e o processo de manutenção. O processo de aperfeiçoamento auxilia na obtenção dos objetivos organizacionais estabelecidos pelo aperfeiçoamento da qualidade de software e pela produtividade da equipe de desenvolvimento. O uso de métricas pode ser um auxílio valioso para a compreensão dos efeitos das ações que são executadas para o aperfeiçoamento do processo de desenvolvimento de software.

Futuramente, as métricas serão aplicadas mais freqüentemente, visando a prevenção de falhas, fornecendo um mecanismo que forneça informações, que auxiliam na análise dos problemas para que o processo de desenvolvimento possa ser atualizado. A aplicação de métricas associadas com atividades de prevenção à falhas causarão impacto positivo na qualidade e produtividade do processo de desenvolvimento do software.

Outra projeção sobre métricas é baseada na observação das organizações que já têm utilizado métricas por um período considerável. Nestas organizações, um extenso histórico de dados de métricas estão disponíveis para a equipe de desenvolvimento que pode utilizar-se do mesmo para a definição de qualquer projeto.

2.6 DIVISÃO DAS MÉTRICAS DE SOFTWARE

Segundo [KAN1995], as métricas de software podem ser classificadas em três categorias:

a) métricas de produto: são aquelas que descrevem as características do produto, como tamanho, complexidade, caraterísticas de projeto, performance e nível da qualidade;

b) métricas de processo: são aquelas que podem ser utilizadas para o aperfeiçoamento processo de desenvolvimento e manutenção de software. Um exemplo deste tipo de métrica é a eficiência da remoção de defeitos do software durante a fase de desenvolvimento;

c) métricas de projeto: são aquelas que descrevem as características e execução do projeto. Exemplos deste tipo de métrica incluem o número de programadores, custo, agenda e produtividade.

Ainda em métricas de software há uma ramificação que é especializada em métricas da qualidade do software. Esta ramificação foca aspectos de qualidade do produto, processo e projeto. Geralmente as métricas de software são mais associadas às métricas de processo e produto do que às métricas de projeto. A principal aplicação das métricas da qualidade de software é investigar o relacionamento entre métricas de processo, características de projeto e qualidade do produto final [KAN1995].

2.7 APLICAÇÃO DE MÉTRICAS EM EMPRESAS

Como afirmado anteriormente, uma métrica caracteriza aspectos de uma entidade qualquer, então a mesma pode ser utilizada para efetuar-se caracterizações de qualquer natureza. Devido à esta facilidade a mesma também é utilizada pelas empresas. Segundo [FER1995], é comum ouvir-se a afirmação de que não se pode gerenciar o que não se pode medir. As métricas ajudam na definição de objetivos, monitoramento de processos e solução de problemas.

É importante que o processo de definição de métricas seja compreendido por toda a organização, como é importante obter o engajamento necessário por parte do todo. Muitos integrantes na organização serão envolvidos na definição de quais métricas serão utilizadas e como os objetivos serão determinados. Isto fará com que os mesmos sintam-se responsáveis pela execução da mensuração e obtenção dos resultados.

Vários indivíduos e equipes podem participar do processo de definição de métricas e objetivos, mas as métricas e objetivos adotados serão escolhidos pela gerência. Esta escolha deverá ser coesiva, realística e aceita por toda a organização. As métricas escolhidas deverão ser fáceis de lidar e não poderão representar uma perda de tempo nas atividades cotidianas.

Quando uma organização resolve utilizar-se de métricas, os resultados somente serão visualizados pela mesma a longo prazo. O principal motivo da existência de métricas numa empresa é pela possibilidade de medir performance e pela possibilidade desta medição permitir um melhoramento desta performance. Um fator importante deve ser levado em

consideração: os resultados obtidos não serão válidos se forem modificados freqüentemente o conjunto de métricas ou a forma como as mesmas são utilizadas.

2.8 A UTILIZAÇÃO DE MÉTRICAS DE SOFTWARE

Segundo [FER1995], em um organização que se dedica ao desenvolvimento de software, seja como atividade fim, seja como de suporte para uma empresa, há vários objetivos que se buscam atingir, dependendo do estágio de maturidade em que se encontram estas atividades.

Alguns destes objetivos podem se enquadrar na seguinte relação:

a) melhorar a qualidade do planejamento do projeto;
b) melhorar a qualidade do processo de desenvolvimento;
c) melhorar a qualidade do produto resultante do processo;
d) aumentar a satisfação dos usuários e clientes do software;
e) reduzir os custos de retrabalho no processo;
f) reduzir os custos de falhas externas;
g) aumentar a produtividade do desenvolvimento;
h) aperfeiçoar continuamente os métodos de gestão do projeto;
i) aperfeiçoar continuamente o processo e o produto;
j) avaliar o impacto de atributos no processo de desenvolvimento com novas ferramentas;
k) determinar tendências relativas a certos atributos no processo.

Segundo [SHE1993], a questão primordial no âmbito das métricas é o estabelecimento das metas da medição. Portanto esta meta engloba os seguintes requisitos:

a) um objeto de interesse o qual seria diferente ao produto ou processo;
b) um propósito como o entendimento, caracterização ou melhoria;
c) uma perspectiva que identifica quem é o interessado pelos resultados, por exemplo, o gerente, os desenvolvedores de software ou até mesmo o cliente;
d) uma descrição do ambiente para fornecer um contexto próprio para qualquer estado.

Segundo [YOU1995], se as pessoas envolvidas no desenvolvimento de um sistema estiverem unidas em adquirir novos conhecimentos para melhoria do processo do sistema, o conceito de métricas de software será percebido pelo pessoal técnico como um fenômeno positivo; porém deve ficar claro desde o começo que o objetivo do esforço de métrica não é punir as pessoas que cometem erros, mas sim melhorar o processo. Desse modo, quando um gerente de projetos utiliza indevidamente uma métrica de software para reprimir alguém que apresentou um desempenho inferior, todas as pessoas da equipe serão atingidas, e como conseqüência, poderão não mais colaborar com o objetivo das métricas.

Segundo [FER1995], a importância da utilização de métricas para as empresas de software está na realidade atual, que aponta para a necessidade das mesmas, baseado nos seguintes fatores:

a) as estimativas de prazos, recursos, esforço e custo são realizadas com base no julgamento pessoal do gerente do projeto;
b) a estimativa do tamanho do software não é realizada;
c) a produtividade da equipe de desenvolvimento não é mensurada;
d) a qualidade dos produtos intermediários do processo não é medida;
e) a qualidade do produto final não é medida;
f) o aperfeiçoamento da qualidade do produto ao longo de sua vida útil não é medido;
g) os fatores que impactam a produtividade e a qualidade não são determinados;
h) a qualidade do planejamento dos projetos não é medida;
i) os custos de não conformidade ou da má qualidade não são medidos;
j) a capacidade de detecção de defeitos introduzidos durante o processo não é medida;
k) não há ações sistematizadas no sentido de aperfeiçoar continuamente o processo de desenvolvimento e de gestão do software;
l) não há avaliação sistemática da satisfação dos usuários e clientes.

Deste modo, pode-se observar que a medição de software tem a importância de fornecer aos responsáveis pelo seu desenvolvimento, as informações que permitem que seu gerente possa planejar o projeto de forma adequada, controlando todo o trabalho de desenvolvimento com maior exatidão. Isto possibilita a construção de um software confiável e

menos propenso à erros, promovendo a satisfação do cliente e uma boa imagem da empresa desenvolvedora.

Para a implantação de métricas na empresa é necessário que seja traçado o caminho pelo qual a mesma será implantada. Segundo [FER1995], a seguir estão listadas diretrizes básicas para a implantação de um sistema de métricas em uma empresa:

a) definir os objetivos do projeto;
b) atribuir responsabilidades pois o resultado de avaliações e retorno do processo de medição em termos quantitativos, pode não ser tão rápido como todos esperam;
c) fazer pesquisas, sempre tentando encontrar novos tipos de métricas para avaliação;
d) definir as métricas iniciais a serem coletadas, devendo utilizar-se as métricas que se identificam com cada fase do ciclo de vida do software;
e) fazer com que as pessoas entendam a utilidade das avaliações do processo de desenvolvimento de sistemas, divulgando a idéia de como as métricas podem ser utilizadas para este fim;
f) obter ferramentas para a coleta de dados e análise de dados automáticas;
g) estabelecer um programa de treinamento em metrificação de software: muitas vezes os integrantes da equipe não tiveram ainda noções sobre este tipo de metodologia de trabalho, portanto, se não houver treinamento, as pessoas podem ficar desmotivadas sem compreender a utilidade do trabalho empregado para a coleta de dados;
h) publicar casos de sucesso na utilização de métricas e encorajar o intercâmbio de idéias.

É importante comunicar a noção de que as métricas de software não são passivas, mas são uma ferramenta para ação. Dessa forma as métricas podem ser usadas para identificar o que está errado no projeto, antes que um problema sério se desenvolva.

Segundo [MÖL1993], quando uma empresa de software tenta implantar um sistema de métricas, pode encontrar barreiras na utilização dos novos conceitos que esta etapa introduz na companhia. Algumas destas barreiras podem ser vistas a seguir:

a) as métricas restringem a criatividade do processo: as métricas podem ser vistas como um tempo perdido para avaliar processos. Esta idéia deve ser superada através de exemplos de métricas e experiências. As métricas são necessárias para

ajudar a modificar o desenvolvimento de software, fazendo com que o processo seja continuamente melhorado;

b) as métricas foram criadas para dar mais trabalho: manter os registros de dados métricos causam um trabalho adicional. Por este motivo, deve-se planejar, economizando assim tempo e dinheiro, sempre levando em consideração todo o ciclo de vida de desenvolvimento de um software;
c) não há objetivos definidos: quando são introduzidas métricas em um projeto, pode ocorrer pouca concordância entre os membros da equipe. Isto ocorre porque não há uma fundamentação necessária para a introdução destes métodos. Os benefícios dos métodos quantitativos devem ser claramente explicados para toda a organização;
d) medo do início da medição: muitas pessoas têm medo de analisar os seus retrospectos de tempos de escola. A organização deve sempre considerar a melhora da qualidade no processo de desenvolvimento do software e, não deve punir as pessoas pelos seus erros. A utilização de métricas traz a melhoria para o processo de desenvolvimento e portanto, o que aconteceu no passado, terá pouca probabilidade de ocorrer novamente;
e) não há necessidade de melhorias: muitas pessoas verificarão que com a implantação de melhoria na qualidade e produtividade é que surgem os maiores problemas. Eles defendem que o sistema atual pode limitar a aprovação de um programa de métricas. Isto deve ser descartado, porque a melhoria no processo é sempre possível e pensando assim, obtém-se os maiores benefícios.

A necessidade de mudança e melhoria deve ser um sentimento das pessoas. Isto ocorre na vida diária de todos, então, por que não passar este sentimento para o mundo dos negócios? As métricas são ferramentas que auxiliam nos processo de melhoria, no entanto, a mudança deve ser consciente em todo o grupo de pessoas da organização. Todos devem compreender perfeitamente os benefícios e as dificuldades na utilização de métricas no processo de desenvolvimento de um sistema.

2.9 TIPOS DE MÉTRICAS DE SOFTWARE

As métricas de software possuem várias aplicações, sendo que existem métricas que são direcionadas a uma fase específica do ciclo de vida do desenvolvimento do software. Desse modo, existem métricas que são utilizadas na fase de projeto para estimar os custos do desenvolvimento deste software e outras métricas são destinadas a verificar o comportamento da equipe em relação à codificação do sistema como defeitos e custos de retrabalho.

O enfoque atual das empresas em relação à utilização de métricas de software é visando a qualidade do produto final, na tentativa de garantir que o mesmo atenda as necessidades do cliente e seja o mais eficiente possível. As métricas da qualidade são outro tipo de métricas de software.

Como pode-se notar, o assunto *métricas de software* é muito extenso para ser abordado adequadamente em um único trabalho. Para maiores informações acerca destas métricas, consulte o autor [FUC1995], conforme a bibliografia deste trabalho. Tendo em vista esta limitação, será dado um enfoque às métricas de código-fonte. Estas métricas podem ser aplicadas tanto na fase de desenvolvimento do software, fornecendo estimativas para o projeto, quanto na fase de manutenção, fornecendo informações para auxiliar a equipe de desenvolvimento nesta tarefa.

Na fase de manutenção, geralmente, há a necessidade de um reestudo da rotina desenvolvida, de modo que o programador a efetuar a manutenção, faça-a da maneira correta. Quando o sistema desenvolvido possui uma documentação e especificação adequada, esta não é uma tarefa muito difícil, no entanto, não é comum encontrar empresas que lutem para manter seus sistemas bem documentados e especificados. Nestas situações, a existência de uma ferramenta que forneça informações sobre o código-fonte a ser mantido, é de grande valia, pois o programador poderá utilizar-se da mesma para estudar e entender o sistema. Isto também permitirá que o mesmo possa visualizar o escopo do impacto de sua manutenção, isto é, o mesmo poderá analisar quais as partes do sistema serão afetadas com a manutenção.

No próximo tópico serão abordadas, apenas para conhecimento teórico, duas das tradicionais métricas de software. Deve-se notar que as métricas que estão contempladas no protótipo são aquelas que estão relacionadas no terceiro capítulo deste trabalho.

2.9.1 ANÁLISE POR PONTOS DE FUNÇÃO (FPA)

Segundo [ARI1993], esta técnica foi proposta por Albrecht e Gaffney. A contagem de pontos de função tem como objetivos:

a) medir o que o cliente/usuário requisitou e recebeu;
b) medir independentemente da tecnologia utilizada para implementação;
c) propiciar uma métrica de tamanho para apoiar análises da qualidade e produtividade;
d) propiciar um veículo para estimativa de software;
e) propiciar um fator de normalização para comparação entre softwares.

Neste método, o tamanho de um software é calculado por componentes os quais fornecem informações como: tamanho de processamento e complexidade técnica dos fatores de ajuste. Este método utiliza como unidades de medida, os aspectos externos do software requisitados pelo usuário. O cálculo é efetuado através dos pontos de função, que são os dados ou transações do sistema.

2.9.2 COCOMO (CONSTRUCTIVE COST MODEL)

Segundo [FER1995], este método foi desenvolvido por Barry Boehm e refere-se à um modelo paramétrico utilizado para estimativas de esforço, prazo, custo e tamanho da equipe para um projeto de software. O COCOMO é um conjunto de submodelos hierárquicos divididos em submodelos básicos, intermediários ou detalhados.

3 MÉTRICAS PARA CÓDIGO FONTE

Ao longo dos anos, um grande esforço foi gasto no desenvolvimento de métricas para obter a complexidade do código-fonte. Muitas das técnicas incorporam facilmente propriedades do próprio código-fonte, como número de operadores e operandos, a complexidade do grafo de controle de fluxo, o número de parâmetros e variáveis globais nas rotinas, o números de camadas (níveis) e o modo de intercomunicação no grafo de controle. Outras técnicas analisam a complexidade do problema a ser resolvido e a forma como o mesmo foi representado computacionalmente. Como complexidade do problema, entenda-se a quantidade de variáveis externas ao sistema que serão fundamentais para o funcionamento do mesmo, quantidade de funções que serão executadas pelo sistema e o tipo de saída que é fornecida [FAR1985].

Um exemplo é a complexidade de um sistema de apoio à tomada de decisões, cuja complexidade não será semelhante à complexidade de um sistema de contas a pagar, pois os controles e resultados que ambos devem emitir são totalmente diferentes. Estas métricas auxiliam na elaboração de uma planilha de custos que será utilizada no desenvolvimento e manutenção do software.

As métricas de software orientadas ao código-fonte, fornecem informações sobre o produto de software, pois através dos resultados obtidos por estas métricas poderá ser estimado o desenvolvimento do software, diagnosticar problemas de codificação, estudar e entender o produto. No decorrer deste capítulo serão apresentadas algumas métricas de software orientadas ao código-fonte.

Algumas destas métricas serão contempladas no protótipo a ser desenvolvido conjuntamente com o trabalho, sendo que neste trabalho, foram “criadas” novas métricas de código-fonte. Estas métricas levam em consideração, conselhos da Engenharia de Software quanto à codificação de programas, sendo que as mesmas foram implementadas no protótipo.

3.1 A MÉTRICA LOC (LINHAS DE CÓDIGO)

Segundo [ARI1993], o sistema LOC, foi a primeira métrica de software utilizada. Ela pode ser aplicada para estimar o custo do software ou especificar igualdades de analogias. Há muitas discussões e especulações sobre esta técnica. Primeiramente, a definição de linhas de

código não é muito clara. Um exemplo simples seria o caso de ser colocado ou não, um comentário ou uma linha em branco como LOC. Alguns autores consideram estes comentários afirmando que estes geralmente são informados para facilitar a manutenção do código, mas no entanto, outros autores não consideram um comentário como LOC, pois o mesmo não está presente no executável (produto final). No caso de programas recursivos, esta técnica falha, porque a recursividade torna o programa mais curto. O sistema LOC é uma técnica genérica e superficial

Podem ser feitas algumas medições experimentais, como o custo por LOC, os LOCs por dia e por programador, convertendo-se os resultados do LOC em custos unitários como tempo, número de pessoas e dinheiro.

Segundo [SHE1993], em função da simplicidade desta métrica, ela foi globalmente rejeitada pela razão principal de poder facilmente ser modificada ou manipulada pelo programador. Isto acontece porque geralmente os programadores organizam os seus programas de formas diferentes. Uma sugestão, é que a métrica LOC seja respeitada por ser uma métrica básica, a partir da qual todas as outras métricas são comparadas.

Segundo [CAP1995], as vantagens do sistema LOC são:

a) é fácil de ser obtido;
b) é utilizado por muitos modelos de estimativa de software como valor básico de entrada;
c) existe farta literatura e dados sobre este sistema de métrica.

As desvantagens são:

a) dependência de linguagem: não é possível comparar diretamente projetos que foram desenvolvidos em linguagens diferentes. Como exemplo, pode-se verificar a quantidade de tempo gasto para gerar uma instrução em uma linguagem de alto nível comparado com uma linguagem de baixo nível;
b) penalizam programas bem projetados: quando um programa é bem projetado o mesmo utiliza poucos comandos para a execução de uma tarefa. Assim sendo, um programa que utilize componentes está melhor projetado, mas a medição deste tipo de programa através desta métrica não é eficiente;

c) difíceis de estimar no início do projeto de software: é praticamente impossível estimar o LOC necessário para um sistema saindo da fase de levantamento de requisitos ou da fase de modelagem.

Após estas explanações, nota-se que a métrica LOC não é uma métrica a ser utilizada por si só. A mesma deveria ser utilizada em conjunto com outras métricas, efetuando um comparativo de resultados. Deste modo uma métrica poderia completar a outra, fornecendo informações que são pertinentes às características de cada uma.

3.2 MÉTRICA DA CIÊNCIA DO SOFTWARE

Segundo [FAR1985], Halstead identificou a Ciência do Software – originalmente chamada Física do Software – como uma das primeiras manifestações sobre a métrica de código baseada num modelo de complexidade do software. A idéia principal deste modelo é a compreensão de que o software é um processo de manipulação mental dos símbolos de seus programas. Estes símbolos podem ser caracterizados como operadores (em um programa executável verbos como: IF, DIV, READ, ELSE e os operadores, propriamente ditos) ou operandos (variáveis e constantes), visto que a divisão de um programa pode ser considerada como uma seqüência de operadores (que definem o fluxo da execução do código), associados a operandos.

Esta métrica mostra os seguintes contadores:
N1 – Número total de operadores em um programa.
N2 – Número total de operandos em um programa.
n1 – Número de operadores únicos
n2 – Número de operandos únicos
operadores – Instruções/Sinais
operandos – Variáveis/Constantes

O vocabulário do programa, n, é dado por:
n = n1 + n2;

O tamanho do programa em símbolos, N, é dado por:
N = N1 + N2

Segundo [FAR1985], Halstead sugeriu que a manipulação de um símbolo – um operador ou operando – requer classificações de um dicionário mental, que consiste em um vocabulário *n* do programa inteiro. Ele também sugeriu que esta ação seria realizada por meio de um mecanismo de pesquisa binária. O número de comparações mentais, ou acesso ao dicionário, faz entender que parte do programa pode ser calculado pelo tamanho do vocabulário e total do número de símbolos sendo usados. Isto chama-se volume (V) do

$$V = N \times \log_2 n$$

programa representado por:

Como um programa pode ser feito de diferentes maneiras, pode-se tirar proveito do volume de uma implementação isolada comparando com soluções teóricas ótimas, que tem a medida do volume mínimo, V*. Isto é conhecido como o nível do programa, λ:

$$\lambda = V^* / V$$

Outra métrica seria aquela que indica o aumento da complexidade, D, sendo esta o inverso do nível do programa:

$$D = 1 / \lambda$$

Para se estimar o nível de abstração do programa, ^λ aplica-se o seguinte cálculo:

$$\hat{\lambda} = (2 / n1) \times (n2 / N2)$$

Igualmente pode-se utilizar uma métrica de estimativa da complexidade ^D:

$$\hat{D} = (n1 / 2) \times (N2 / n2)$$

O termo (n1 / 2) aumentará com o uso de operadores adicionais e, por esta razão, adiciona complexidade ao código. O divisor 2 é porque esta é a possibilidade mínima do número de operadores exigidos para uma implementação em um algoritmo, por exemplo: uma chamada de função e o argumento da função. O termo (N2 / n2) é a média utilizada por operando.

Como o nível de complexidade de um programa é dado pela comparação do número de discriminações elementares mentais (EMD), e que o volume é dado pelo número de total de comparações, é possível produzir uma métrica para o esforço exigido E, para a manipulação de um programa.

$E = \hat{}D * V$

Neste caso, a complexidade multiplicada pelo volume, resulta o esforço necessário de pessoas para realizar os trabalhos. Esta é uma medida que pode não mostrar a realidade do desenvolvimento de um programa, porque apenas inclui fatores relacionados ao código propriamente dito. Fatores ambientais poderiam também estar incluídos neste cálculo, para que esta não seja apenas mecânico.

Segundo [SHE1993], Halstead sugeriu que o tempo T, necessário para a geração de um programa pode ser calculado pela utilização do número Stroud S (pesquisador de psicologia cognitiva) que é o número de EMDs que o cérebro é capaz de fazer por segundo. Pela estimativa original de Stroud, o fator S estaria no intervalo de 5 – 20. Halstead, utilizou o valor S igual a 18, permitindo-o prever T em segundos como segue:

$T = E / 18;$

Halstead derivou também uma equação de estimativa para o tamanho do programa ^N. Ela somente se baseia nos contadores n1 e n2 que, desse modo, são avaliados antes do programa estar completo. Portanto tem-se:

$$\hat{} N = n1 \times \log_2 n1 + n2 \times \log_2 n2$$

A ciência do software atraiu consideravelmente o interesse das pessoas em meados de 1970 por ser uma novidade nas métricas de software. Além disso, as entradas básicas desta métrica são facilmente extraídas.

Após o entusiasmo inicial da Ciência do Software, sérios problemas foram encontrados. Os motivos podem ser relatados em função da dificuldade que os pesquisadores encontraram na comparação dos trabalhos de evolução da métrica, outro motivo seria a não associação correta entre esforço requerido para manipulação do programa e o tempo exigido para conceber o programa e também por tratar um sistema como um simples módulo.

3.3 MÉTRICA DA COMPLEXIDADE CICLOMÁTICA

Segundo [SHE1993], este método foi promovido por McCabe, pois ele estava particularmente interessado em descobrir o número de caminhos criados pelos fluxos de

controle em um módulo do software, desde que fosse relacionado à dificuldade de testes e na melhor maneira de dividir um software em módulos.

Os programas são representados por grafos dirigidos, representando o fluxo de controle. De um grafo G, pode ser extraído a complexidade ciclomática v(G). O número de caminhos dentro de um grafo pode ser dado como: o conjunto mínimo de caminhos os quais podem ser utilizados para a construção de outros caminhos através do grafo. A complexidade ciclomática é também equivalente ao número de decisões adicionais dentro de um programa:

v(G) = E – n + 2p,

onde:

E: é o número de arestas

N: é o número de nós

p: é o número de componentes conectados, isto é, é o número de entradas e saídas existentes no grafo do fluxo de execução.

A visão simplista da métrica de McCabe pode ser questionada em vários pontos. Primeiro, ele tinha uma especial preocupação com os programas escritos em FORTRAN, onde o mapeamento do código-fonte, para um grafo de fluxo do programa era bem definido, sendo que isto não seria o caso de outras linguagens como Ada. A segunda oposição é que v(G) = 1, seria verdadeiro em uma seqüência linear de código de qualquer tamanho. Consequentemente, a métrica não é sensível à complexidade, contribuindo assim na formação de declarações de seqüências lineares.

Este argumento seria um caso específico e que há outros pontos no contexto de desenvolvimento que a métrica ignora para a tomada de decisões. Todas as decisões têm um peso igual, considerando qualquer nível de utilidade da informação ou relacionamento com outras decisões. Em outras palavras, McCabe prefere abreviar e então verificar a estrutura do software.

Outro problema da métrica da complexidade ciclomática é o procedimento inconsistente quanto à medida da modularidade de um software. Isto pode ser demonstrado pelo aumento da complexidade ciclomática na adição de módulos extras, mas diminui com o fator de duplicidade de código. Todos os outros aspectos referentes à modularidade são

desconsiderados. O que é contrário à teoria correta da modularização, causando problemas ao objetivo de McCabe no auxílio para a seleção de uma arquitetura de software eficaz.

Esta métrica possui uma séria dificuldade em relação a outros aspectos do software. Não se leva em consideração aqui os dados com os quais um sistema trabalhará, como também a complexidade funcional do sistema, sendo que hoje, a funcionalidade é um requisito altamente necessário para que uma métrica possa avaliar um software corretamente. O funcionamento do sistema faz com que o usuário tenha atração pela ferramenta de trabalho. Se o sistema é complicado, ele pode perder o interesse e continuar realizando seus trabalhos manualmente.

Segundo [SHE1993], v(G) é sensível ao número de subrotinas dentro de um programa, por este motivo, McCabe sugere que este aspecto seja tratado como componentes não relacionados dentro do grafo de controle. Este ponto teria um resultado interessante pois aumentaria a complexidade do programa globalmente, visto que ele é dividido em vários módulos que se imagina serem sempre simples.

3.4 UTILIZAÇÃO DAS MÉTRICAS DE CÓDIGO FONTE

Neste item será dado um enfoque na compreensão das métricas de software para código-fonte, bem como sugestões de novas métricas para uma futura implementação. A melhor maneira de entender uma métrica se software é visualizar a aplicação da mesma, ou os resultados de sua aplicação em um programa fonte.

Assim, serão exibidos os resultados de algumas métricas aplicadas à um código-fonte exemplo. Este fonte exemplo tenta reunir algumas das características básicas para a aplicação das métricas de código-fonte, apenas com o intuito didático.

Deve-se notar que este programa exemplo é um fonte feito em *Pascal*, identificando o código-fonte central, que é o responsável pela execução do mesmo. Tendo em vista a simplicidade do mesmo, não foram utilizados conceitos de modularização de software, como pode ser visto na figura 1.

Baseado na figura 1, pode-se aplicar as seguintes métricas: Linhas de Código (LOC), Ciência do Software e Complexidade Ciclomática. Para a métrica de Linhas de Código,

obtém-se dois valores: contando-se comentários e linhas em branco: **20**; contando-se somente linhas de comando: **16.**

FIGURA 1 - Exemplo de programa-fonte em *Pascal*

```
PROGRAM Metrics;

VAR
   i, x, y, z: INTEGER;
BEGIN
   { Leitura dos valores }
   READ(x);
   READ(y);
   { Verifica se x é maior que y  ou vice-versa}
   IF x > y THEN
      z := x / y
   ELSE
      IF y > x THEN
         z := y / x
      ELSE
         z := y + x;
   { Imprime a mensagem }
   FOR i := 1 TO z DO
      WRITELN('Alô Mundo!');
END.
```

Os operadores e operandos foram diferenciados de maneira a serem identificados mais facilmente, sendo que os operadores estão em negrito e os operandos estão sublinhados.

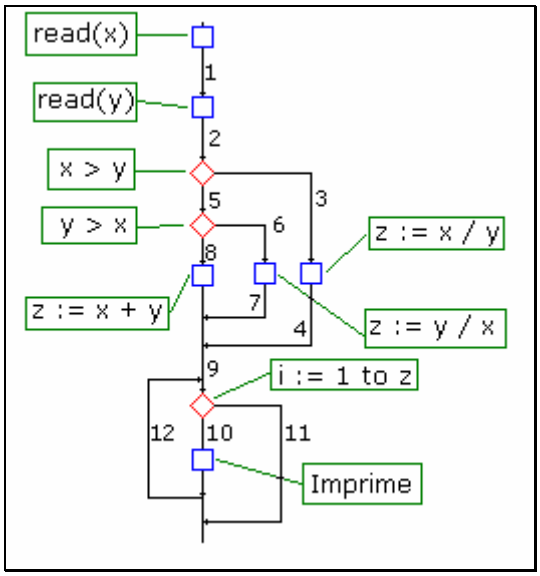
Para a métrica da Ciência do Software obtém-se os seguintes resultados, conforme descrito na tabela 2:

TABELA 2 - Resultados da aplicação da métrica da Ciência do Software.

Variáveis / Métricas	Resultados
Operadores únicos (n1)	8,00
Operandos únicos (n2)	4,00
Total de operadores (N1)	13,00
Total de operandos (N2)	17,00
Vocabulário do programa (n)	12,00
Tamanho do programa em símbolos (N)	30,00
Volume do programa (V)	107,55
Estimativa da Complexidade (^D)	17,00
Esforço exigido (E)	1828,33
Tempo de assimilação (Stroud) (S)	101,57
Estimativa do tamanho do programa (^N)	32,00

Para a métrica da Complexidade Ciclomática, primeiramente precisa-se desenhar o grafo do fluxo de controle do programa, como mostra o desenho da figura 2:

FIGURA 2 - Grafo representativo do programa exemplo



Baseado no desenho obtido da análise do programa-fonte, obtém-se os seguintes resultados, conforme descrito na tabela 3:

TABELA 3 - Resultado da métrica da Complexidade Ciclomática

Variáveis / Métricas	Resultados
Número de arestas (E)	12
Número de nós (N)	9
Entradas e Saídas (Componentes Conectados) (p)	2
Complexidade Ciclomática (v(G) = E – n + 2p)	4

4 DESENVOLVIMENTO DO PROTÓTIPO

4.1 A LINGUAGEM PASCAL

O Pascal é uma linguagem, que possui características como legibilidade, flexibilidade, fácil compreensão, entre outros itens. Devido a estes e outros fatores esta linguagem é grandemente utilizada com fins didáticos. Para uma melhor compreensão desta linguagem serão descritas algumas caraterísticas da mesma [OSL1997].

Para desenvolver um programa que seja executável em um computador é necessário ter em mente que o trabalho de gerenciamento de dados ocorre em três níveis: entrada, processamento e saída de dados. Este trabalho ocorre através da execução de comandos que são ordenados de forma lógica para permitir a execução de uma determinada ação.

Os dados são representados pelas informações a serem processadas por um computador. O Pascal fornece ao programador um conjunto de tipos de dados predefinidos, podendo estes ser dos tipos numéricos (inteiros e reais), caracteres e lógicos. Os tipos de dados inteiros são caracterizados por dados numéricos que não possuem fração, ao contrário dos dados reais, que são dados numéricos com frações. Os tipos de dados caracteres são seqüências de números, letras ou símbolos. Estes dados normalmente são representados entre apóstrofos, o que identifica este tipo de dado. Os tipos de dados lógicos podem representar apenas duas informações: verdadeiro ou falso.

Em programação, variável é uma região de memória do computador, previamente identificada que tem por finalidade armazenar temporariamente os dados de um programa. Cada variável possui um tipo de dado, sendo que esta relação é intrínseca, isto é, uma variável do tipo numérico não pode conter um dado caractere. Para utilizar-se de variáveis, é necessário declará-las primeiro.

As constantes são valores fixos que não são modificadas durante a execução do programa. Sua principal utilização é identificar algum dado específico, como um número ou um texto qualquer.

Tanto as variáveis como as constantes podem ser utilizadas na elaboração de cálculos matemáticos com a utilização de operadores aritméticos. Os operadores aritméticos são

classificados em duas categorias, sendo binários ou unários. São binários quando são utilizando em operações de exponenciação, multiplicação, divisão, adição e subtração. São unários quando modificam o valor de um dados através da mudança de seu sinal.

A tomada de decisão é basicamente feita através da estrutura *if...then*. Este estrutura toma uma decisão e efetua um desvio no processamento do programa, dependendo da condição atribuída à mesma. Esta condição deverá sempre resultar um um valor lógico (verdadeiro ou falso). Para a construção desta condição, utiliza-se, normalmente, operadores relacionais (=, <>, >, <, >=, =<). Uma alternativa à utilização do estrutura IF é a estrutura CASE, onde pode-se agrupar várias ações relacionadas à um único operando, o que torna o código mais legível e compreensível.

Existem estruturas que permitem repetir instruções um determinado número de vezes. Isto pode ser feito utilizando-se as estruturas de repetição como: *while*, *for* e *repeat*. Para utilizar a estrutura *while*, deve-se construir uma condição que, uma vez satisfeita, faça com que o fluxo de execução do programa entre na repetição. Esta será feita até que a condição seja falsa. A estrutura *for*, executa a repetição um número finito de vezes e a estrutura *repeat* possui praticamente a mesma funcionalidade da estrutura *while*.

As subrotinas são utilizadas para facilitar a construção de um programa, modularizando-o. Assim, quando um programa torna-se muito extenso é necessário “separá-lo” para que sua manutenção seja possível. Esta separação é feita através da criação de subrotinas que são “pedaços” do código do programa que são executadas separadamente do programa principal. As subrotinas subdividem-se em *procedures* e *functions*. As *procedures* são subrotinas que executam uma séria de comandos não retornando nenhum valor adicional. As *functions* são subrotinas que retornam informações quando as mesmas são executadas.

Units são bibliotecas de funções e procedimentos as quais podem ser criadas pelo programador. São conjuntos de rotinas que podem ser utilizadas pelo programador, para auxiliá-lo na construção do seu programa. São rotinas compiladas separadamente do programa principal.

Todo programa em Pascal é subdividido em três áreas distintas: cabeçalho, área de declarações e corpo do programa. O cabeçalho do programa é utilizado para a identificação do

mesmo. A área de declarações é utilizada para validar o uso de qualquer tipos de identificador, sendo que estes podem ser: variáveis, constantes, tipos e subrotinas. O corpo do programa é onde está o programa em Pascal propriamente dito. O corpo do programa inicia através de um *begin* e é finalizado por um *end* seguido do ponto final (.).

Para maiores informações sobre a linguagem Pascal, pode-se utilizar [OLI1996] e [OSL1997].

4.2 ESPECIFICAÇÃO DO PROTÓTIPO

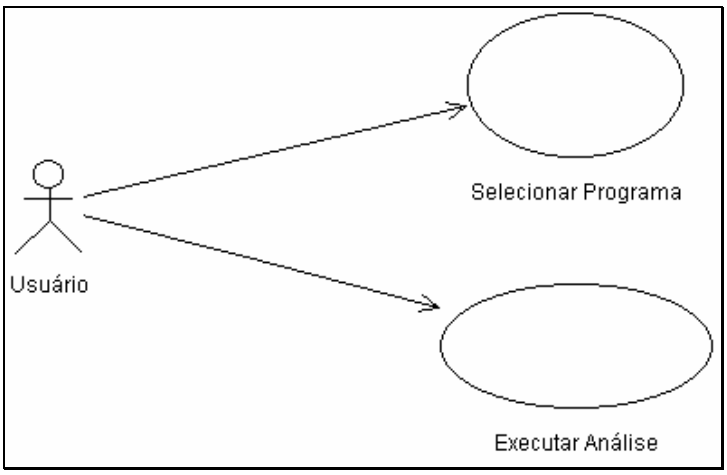
O objetivo do protótipo é fornecer ao usuário algumas métricas de código-fonte, extraídas através da análise de programas codificados no ambiente Delphi. Baseado nestes resultados o usuário poderá estudar um determinado programa e entendê-lo. Para tanto, a abordagem adotada para a especificação foi a orientação a objetos através da UML que segundo [FUR1998], é uma linguagem de modelagem unificada que trata de assuntos inerentes a sistemas complexos e de missão crítica que é suficiente para modelar qualquer tipo de aplicação de tempo real, cliente/servidor ou outros tipos de softwares. A UML pode ser usada para:

a) mostrar as fronteiras de um sistema e suas funções principais utilizando atores e casos de uso;
b) ilustrar a realização de casos de uso com diagramas de interação;
c) representar uma estrutura estática de um sistema utilizando diagramas de classes;
d) modelar o comportamento de objetos através de diagramas de transição de estado;
e) revelar a arquitetura de implementação física com diagramas de componentes e de implantação;

Na figura 3, é exibida o USE CASE do protótipo que pode ser analisado da seguinte maneira:

a) Selecionar Programa: é a ação executada pelo usuário para selecionar o programa que será analisado;
b) Executar Análise: é a ação feita pelo usuário que permitirá o início da análise do programa, e todos os fontes relacionados ao mesmo.

FIGURA 3 - USE CASE do protótipo



A seguir é exibido o diagrama de classes do protótipo dividido em duas partes, sendo que a parte representada pela figura 4 é relativa à analise de código do programa, e, a parte representada pela figura 5, é relativa à representação dos resultados obtidos durante a análise do código-fonte.

FIGURA 4 - Diagrama de Classes do protótipo relativo a Análise de código

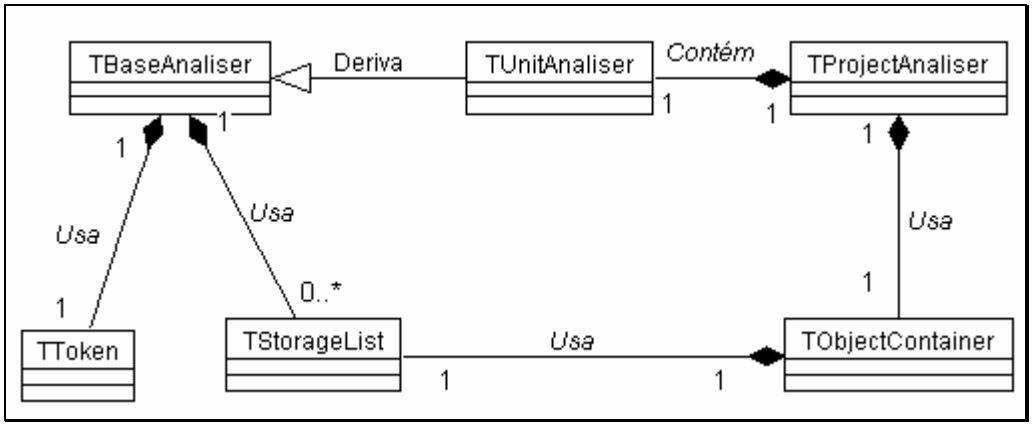
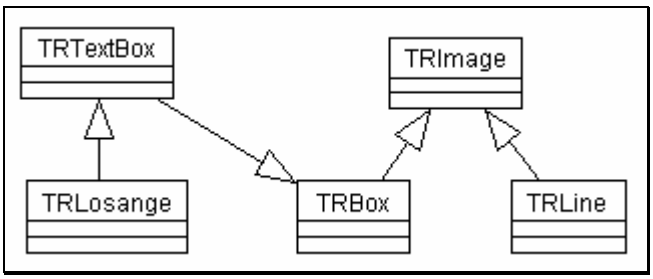
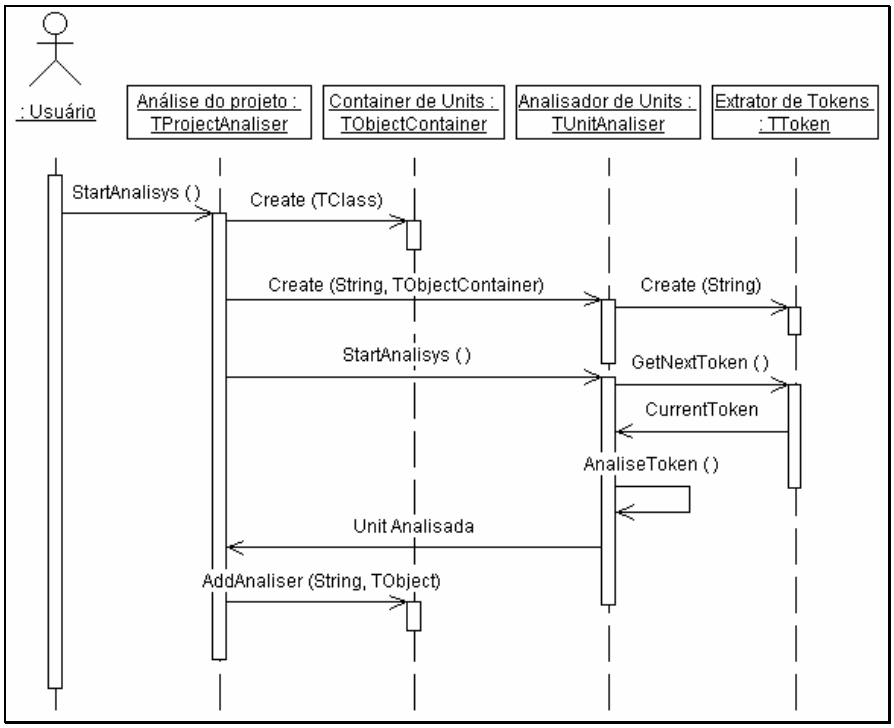


FIGURA 5 - Diagrama de Classes do protótipo relativo a saída ao usuário (gráficos)



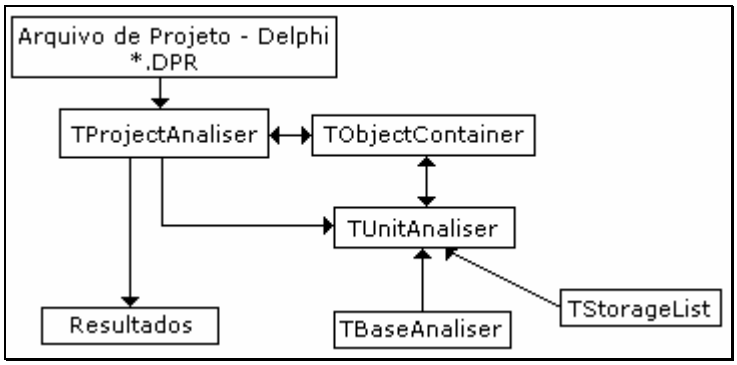
Do mesmo modo, pode-se exibir o diagrama de seqüência, conforme a figura 6, representando a ação de solicitação de análise feita pelo usuário ao protótipo:

FIGURA 6 - Diagrama de Seqüência da Análise de um projeto.



O diagrama de seqüência exibido na figura 6, representa, segundo as convenções da UML, a etapa da análise do programa-fonte efetuada pelo protótipo, ou, de acordo com o USE CASE, do protótipo, o caso de uso *Executar Análise*. Este USE CASE resume-se nas seguintes ações: a solicitação é enviada pelo usuário, sendo que esta acarretará na instanciação da classe TProjectAnaliser. Esta classe, após instanciada estará apta a efetuar a análise dos programas-fontes e fornecer os resultados. Esta simples solicitação desencadeia inúmeros eventos que são tratados pelos objetos associados a classe TProjectAnaliser, conforme exibido no diagrama da figura 7. A seguir são descritas as classes utilizadas no protótipo. Optou-se por apresentar individualmente cada classe em função destas classes possuírem uma grande quantidade de propriedades e métodos tornando o diagrama de classes geral muito complexo. Desta maneira, pode-se exibir o esquema representado na figura 7, que mostra o relacionamento existente entre as classes, no que se refere à analise do código-fonte.

FIGURA 7 - Esquema de execução da análise de um projeto.



Como pode ser observado, após a seleção de um arquivo de projeto a análise é efetuada. Para tanto, são instanciados os seguintes itens:

a) TProjectAnaliser, o analisador do projeto como um todo;
b) TUnitAnaliser, o analisador da unit referenciada;
c) TObjectContainer, responsável pelo armazenamento das informações;
d) TStorageList, armazenamento de listas de valores;
e) TBaseAnaliser, classe base para TUnitAnaliser.

A seguir serão detalhadas cada uma das classes do projeto, sendo que para a representação dos elementos das classes, foi adotada a seguinte notação:

a) a primeira parte da figura representa o nome da classe;

b) a segunda parte da figura representa os atributos da classe;

c) a terceira parte da figura representa os métodos da classe;

Certos atributos são privados, isto é, não podem ser utilizados diretamente pelo programador. Para tanto, existem as *propriedades*, que são maneiras diferentes de acesso aos atributos privados. Na figura abaixo as propriedades são representadas pelo seu nome e pelo atributos aos quais têm acesso. Por exemplo: *ProjectName: String read FProjectName*. Isto indica que a propriedade *ProjectName* acessa o atributo *FProjectName*. Estas propriedades não são exibidas das tabelas de atributos e métodos das classes.

Classe TProjectAnaliser: Esta classe é a responsável pela análise do projeto como um todo. Esta classe dispara a execução da análise dos módulos do projeto, sendo que todos os resultados poderão ser obtidos utilizando-se como referência esta classe. Essas análises são representadas pelos objetos TUnitAnaliser que são instanciados e executados para cada arquivo fonte que é utilizado pelo projeto analisado. A representação desta classe pode ser visualizada através da figura 8 e da tabela 4.

FIGURA 8 - Classe TProjectAnaliser

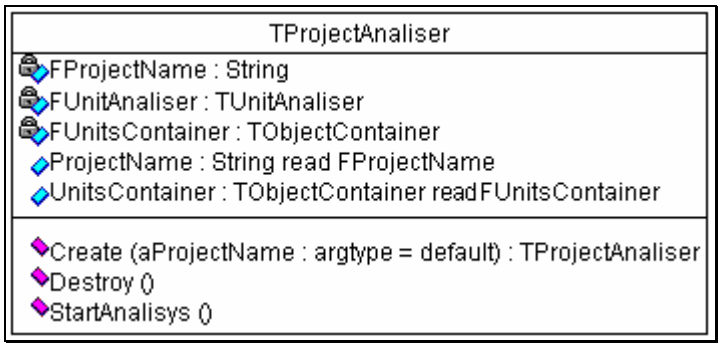


TABELA 4 - Atributos e métodos da classe TProjectAnaliser

Atributo	Valor
FProjectName	contém o nome do projeto a ser analisado
FUnitAnaliser	contém uma instância do objeto TUnitAnaliser
FUnitsContainer	contém uma instância do objeto FUnitsContainer
Método	**Ação**
Create	construtor da classe
Destroy	destrutor da classe
StartAnalisys	inicia a análise propriamente dita, do programa-fonte

Classe TToken: a classe TToken é a responsável pela extração dos tokens do programa-fonte. Esta classe é associada à classe TBaseAnaliser que utiliza-a para retirar os

tokens do programa-fonte. Ao extrair um token, é feito um reconhecimento prévio do mesmo, isto é, quando é extraído um token, a classe TToken já define a característica básica do mesmo, indicando se este token é uma cadeia de caracteres, um número, um identificador, um operador matemático, um operador relacional, entre outros. Baseado nesta identificação prévia a classe TBaseAnaliser pode dar seqüência à análise.

A especificação desta classe, bem como seus atributos e métodos podem ser visualizados na figura 9 e tabela 5.

FIGURA 9 - Classe Ttoken

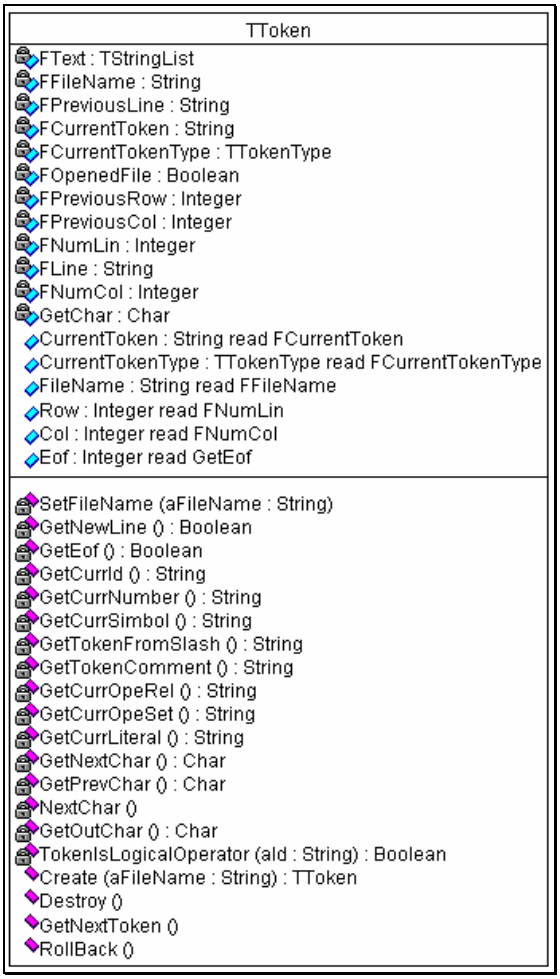


TABELA 5 - Atributos e métodos da classe TToken

Atributo	Valor
FText	contém o texto do arquivo a ser analisado
FFileName	nome do arquivo analisado
FPreviousLine	possui o texto da linha anterior (utilizada por RollBack)
FCurrentToken	possui o token atualmente extraído
FCurrentTokenType	possui o tipo do token extraído e este tipo poderá ser: (ttId,

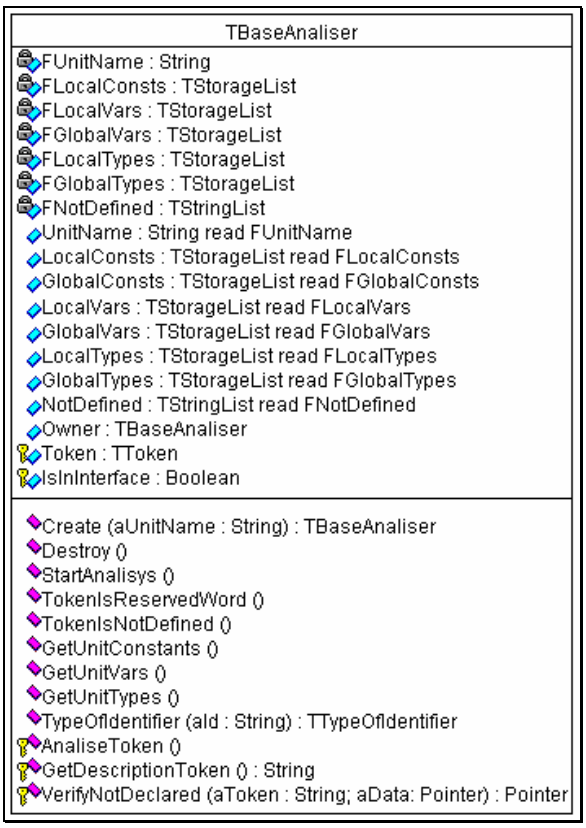
	ttNum, ttLiteral, ttSimbol, ttMath, ttRelat, ttAttrib, ttLogical, ttComm, ttBof, ttEof, ttError, ttNone)
FOpenedFile	identifica se o arquivo associado já foi aberto pelo extrator
FPreviousRow	possui o número da linha do último token extraído
FpreviousCol	possui o número da coluna do último token extraído
FNumLin	possui o número da linha atualmente em análise
FLine	possui o texto da linha atualmente em análise
FNumCol	possui a coluna da linha atual onde está a análise
GetChar	retorna o próximo caracter do token analisado
Método	Ação
SetFileName	atribui o nome do arquivo a FFileName
GetNewLine	retorna a próxima linha a ser analisada
GetEof	indica se é o fim do arquivo
GetCurrId	extrai o token atual como identificador
GetCurrNumber	extrai o token atual como número
GetCurrSimbol	extrai o token atual como símbolo
GetTokenFromSlash	verifica se o token atual é início de comentário ou símbolo
GetTokenComment	extrai o token atual como comentário
GetCurrOpeRel	extrai o token atual como operador relacional
GetCurrOpeSet	extrai o token atual como operador de atribuição
GetCurrLiteral	extrai o token atual como uma cadeia de caracteres
GetNextChar	retorna o próximo caracter a ser lido
GetPrevChar	retorna o caracter anteriormente lido
NextChar	avança a coluna em um caracter
GetOutChar	retorna o caracter para a montagem do token
TokenIsLogicalOperator	verifica se o token é um operador lógico
Create	construtor da classe
Destroy	destrutor da classe
GetNextToken	extrai o próximo token
RollBack	volta em um token a análise do arquivo atual

Classe TBaseAnaliser: Esta classe é uma classe base para a análise do programa-fonte propriamente dito. Esta classe instancia a classe TToken e inicia o reconhecimento de todos os identificadores existentes no código-fonte.

Sendo uma classe base, a mesma possui vários métodos que são virtuais, isto é, podem ser sobrescrevidos pelas classes descendentes. A classe descendente de TBaseAnaliser é a classe TUnitAnaliser que será descrita mais adiante.

A especificação da classe pode ser vista na figura 10. A tabela 6 exibe as propriedades e métodos da classe TToken.

FIGURA 10 - Classe TBaseAnaliser



As propriedades da classe TBaseAnaliser são as seguintes:

TABELA 6 - Atributos e métodos da classe TToken

Atributo	Valor
FUnitName	nome do programa-fonte a ser analisado
FLocalConsts	lista de constantes locais
FGlobalConsts	lista de constantes globais
FLocalVars	lista de variáveis locais
FGlobalVars	lista de variáveis globais
FLocalTypes	lista de tipos locais
FGlobalTypes	lista de tipos globais
FNotDefined	lista de identificadores não definidos ou não reconhecidos
Owner	endereço de memória da classe que iniciou a análise do código-fonte. Geralmente este atributo possui valor quando a análise que está sendo efetuada é de alguma subrotina do programa-fonte que está sendo analisado
Token	instância do objeto TToken para a extração dos tokens
IsInInterface	verifica se a análise encontra-se atualmente na interface ou implementação
Método	**Ação**
Create	construtor da classe

Destroy	destrutor da classe
StartAnalisys	inicia a análise propriamente dita
TokenIsReservedWord	verifica se o token retornado é uma palavra reservada
TokenIsNotDefined	verifica se o token retornado é um identificador não reconhecido
GetUnitConstants	obtém as constantes da unit analisada
GetUnitVars	obtém as variáveis da unit analisada
GetUnitTypes	obtém os tipos da unit analisada
IsConstant	verifica se o identificador atual é uma constante
IsVariable	verifica se o identificador atual é uma variável
TypeOfIdentifier	retorna o tipo do identificador que pode ser: (tiReservedWord, tiConstant, tiVariable, tiType, tiSubRoutine, tiNotDefined)
AnaliseToken	efetua a análise do token extraído e dispara os processos necessários para o tratamento do mesmo
GetDescriptionToken	retorna o token de forma que o mesmo possa ser utilizado para a montagem de mensagens a serem enviadas ao usuário final
VerifyNotDeclared	verifica se o token é um identificador não reconhecido.

Classe TUnitAnaliser: a classe TUnitAnaliser é uma generalização da classe TBaseAnaliser, pois além de utilizar todos os métodos e atributos da classe pai, necessita de alguns aperfeiçoamentos para a análise das subrotinas que eventualmente podem ser encontradas em qualquer programa-fonte a ser analisado. Desta forma, esta classe é a responsável direta pela análise do código-fonte do do programa em Pascal, sendo que a mesma é utilizada pela classe TProjectAnaliser para efetuar esta análise. A especificação pode ser vista na figura 11 e as propriedades e métodos podem ser vistos na tabela 7.

FIGURA 11 - Classe TUnitAnaliser

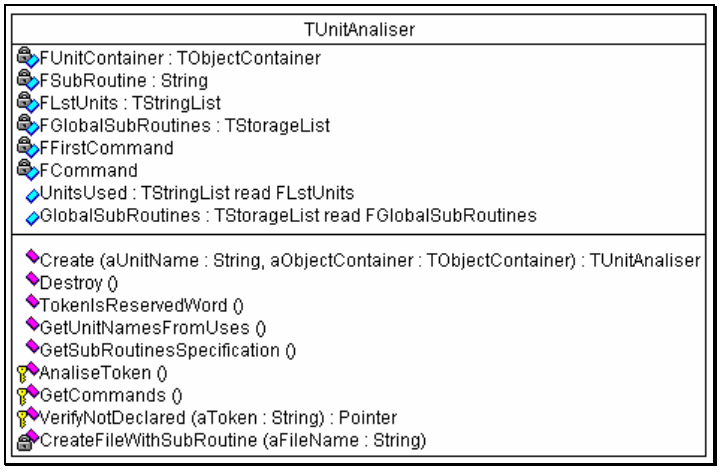


TABELA 7 - Atributos e métodos da classe TUnitAnaliser.

Atributo	Valor
FUnitContainer	instância da classe TObjectContainer.

FLstUnits	lista dos *uses* feitos pela unit atualmente analisada
FGlobalSubRoutines	lista de SubRotinas presentes na unit analisada
FSubRoutine	contém o nome da subrotina analisada
FFirstCommand	contém o primeiro endereço de memória com a estrutura do primeiro comando analisado para a construção dos gráficos de resultado
FCommand	contém o último endereço de memória apontando para a estrutura dos comandos
Método	Ação
Create	construtor da Classe
Destroy	destrutor da Classe
TokenIsReservedWord	verifica se o token é uma palavra reservada. Este método é uma especialização do método já presente na classe pai (TBaseAnaliser)
GetUnitNameFromUses	extrai todos os nomes das units nas quais é feito o *uses*
GetSubRotineSpecification	extrai todo o corpo da subrotina para sua posterior análise
AnaliseToken	analisa o token atualmente extraído. Este método é uma especialização do método já existente na classe pai (TBaseAnaliser)
VerifyNotDeclared	verifica se o token declarado é um identificador não reconhecido. Este método também é uma especialização do mesmo método da classe pai (TBaseAnaliser)
CreateFileWithSubRoutine	cria um arquivo temporário com o código-fonte da subrotina a ser analisada. Isto é necessário, pois as subrotinas possuem praticamente todas as características que encontram-se em um programa-fonte qualquer. Neste caso, é necessária uma análise completa da subrotina.

Classe TObjectContainer: a classe TObjectContainer pode ser encarada como um repositório de objetos. Internamente, esta classe utiliza-se de uma lista em memória para armazenar os objetos instanciados. A classe TProjectAnaliser e TUnitAnaliser utilizam-se desta classe para o armazenamento de todos os objetos TUnitAnaliser cuja análise está finalizada. O principal objetivo é o reaproveitamento de análises já efetuadas em outros programas fontes de um mesmo projeto.

Por exemplo, se as units A e B utilizam-se de C, a unit C somente será analisada uma única vez, pois após sua análise a mesma será adicionada à lista de units analisadas. Assim, se a unit A for analisada primeiramente, desta análise resultará a análise da unit C, pois a mesma é necessária por ser utilizada pela unit A. Quando a unit B for analisada a unit C já estará pronta, pois a mesma encontra-se, juntamente com a unit A, na lista de unit analisadas. Esta lista é mantida pela classe TProjectAnaliser através do atributo FUnitsContainer. A figura 12

representa a classe TObjectContainer, sendo que suas propriedades e métodos são apresentadas na tabela 8.

FIGURA 12 - Classe TObjectContainer

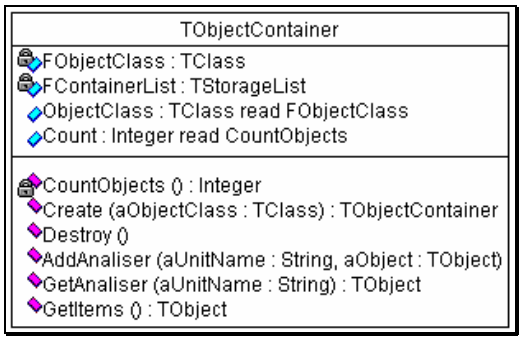


TABELA 8 - Atributos e métodos da classe TObjectContainer

Atributo	Valor
FObjectClass	contém a classe que identifica o tipo de objeto armazenado
FContainerList	lista em memória para o armazenamento dos objetos
Método	**Ação**
CountObjects	retorna a quantidade de objetos armazenados na lista
Create	construtor da classe
Destroy	destrutor da classe
AddAnaliser	adiciona o objeto (TUnitAnaliser) à lista de objetos
GetAnaliser	retorna o objeto (TUnitAnaliser) da lista de objetos
GetItems	retorna o objeto baseado em seu índice numérico

Classe TStorageList: esta classe é uma especialização da classe TStringList da linguagem Delphi, permitindo o cadastro de qualquer item de memória, atribuindo-lhe um identificador, como por exemplo, um nome. Esta classe é extremamente útil à análise pois é através da mesma que são mantidas as listas em memória das informações lidas dos programas-fonte.

Todas as classes especificadas até o momento, possuem pelo menos um atributo do tipo TStorageList. Quando qualquer dado é armazenado neste objeto, o mesmo verifica se o objeto já está inserido; se não estiver, procede com a inserção normalmente, caso contrário ignora o pedido de inserção. Para a recuperação dos valores basta informar o nome do objeto que será devolvido o seu endereço de memória. Sua especificação pode ser vista na figura 13, juntamente com seus métodos e propriedades na tabela 9.

FIGURA 13 - Classe TStorageList

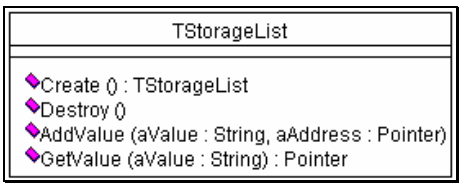


TABELA 9 - Atributos e métodos da classe TStorageList

Método	Ação
Create	Construtor da classe
Destroy	Destrutor da classe
AddValue	Adiciona o objeto à lista.
GetValue	Retorna o objeto da lista.

Classe TRImage: esta é a classe base responsável pelo desenho dos objetos das saídas gráficas. A especificação desta classe pode ser vista na figura 14 e na tabela 10.

FIGURA 14 - Classe TRImage

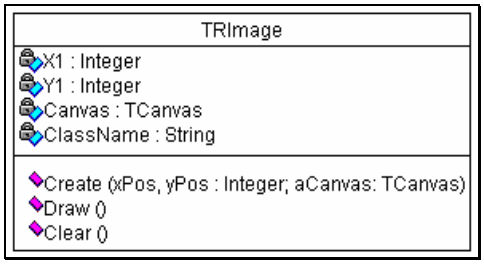


TABELA 10 - Atributos e métodos da classe TRImage

Atributo	Valor
X1	contém o ponto X1 (X1 e Y1 formam o ponto inicial da imagem)
Y1	contém o ponto Y1 (X1 e Y1 formam o ponto inicial da imagem)
Canvas	contém o endereço da área da janela onde a figura será desenhada
ClassName	nome identificador da classe
Método	**Ação**
Create	construtor da classe
Draw	desenha o objeto
Clear	apaga o objeto da tela

Classe TRBox: esta classe é uma generalização de TRImage e desenha uma caixa na tela. Esta caixa pode ser dimensionada conforme a necessidade, modificando-se os atributos de posicionamento. A especificação desta classe pode ser vista na figura 15 e tabela 11.

FIGURA 15 - Classe TRBox

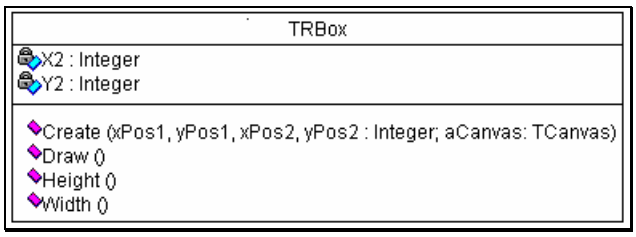


TABELA 11 - Atributos e métodos da classe TRBox

Atributo	Valor
X2	contém o ponto X2 (X2 e Y2 formam o ponto final da imagem)
Y2	contém o ponto Y2 (X2 e Y2 formam o ponto final da imagem)
Método	**Ação**
Create	construtor da classe
Draw	desenha o objeto
Height	indica a altura do objeto
Width	indica a largura do objeto

Classe TRLine: esta classe é uma generalização de TRImage e é responsável pela interligação dos componentes do desenho. A ligação é feita quando são informados os dois objetos que deverão estar interligados. Esta classe traça uma linha de um objeto até o outro, formando uma união entre estes objetos. A especificação desta classe pode ser vista na figura 16 e tabela 12.

FIGURA 16 - Classe TRLine

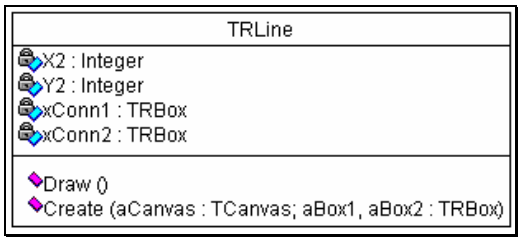


TABELA 12 - Propriedades e métodos da classe TRLine

Atributo	Valor
X2	contém o ponto X2 (X2 e Y2 formam o ponto final da imagem)
Y2	contém o ponto X2 (X2 e Y2 formam o ponto final da imagem)
xConn1	objeto de inicio da conexão
xConn2	objeto de fim da conexão
Método	**Ação**
Create	construtor da classe
Draw	desenha a linha de união

Classe TRTextBox: esta classe é uma generalização da classe TRBox e desenha uma caixa com um texto. A especificação desta classe pode ser vista na figura 17 e tabela 13.

FIGURA 17 - Classe TRTextBox

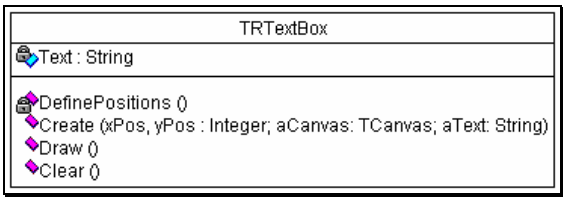


TABELA 13 - Propriedades e métodos da classe TRTextBox

Propriedade	Valor
Text	texto a ser exibido dentro da caixa
Método	Ação
Create	construtor da classe
Draw	desenha a linha de união
Clear	apaga o desenho da tela
DefinePositions	calcula as posições do desenho na tela

Classe TRLosange: esta classe é uma generalização de TRTextBox, sendo que a mesma desenha um losango na tela. Este losango representa o símbolo de decisão em um fluxograma, sendo que o mesmo é utilizado por várias estruturas. A especificação desta classe pode ser melhor visualizada através da figura 18 e tabela 14.

FIGURA 18 - Classe TRLosange

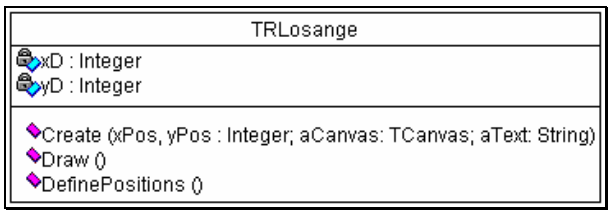


TABELA 14 - Propriedades e métodos da classe TRLosange

Propriedade	Valor
xD	possui a posição central da largura da imagem
yD	possui a posição centra da altura da imagem
Método	Ação
Create	construtor da classe
Draw	desenha a linha de união
DefinePositions	calcula as posições do desenho na tela

4.3 LÓGICA GERAL

Nesta seção será explanado o funcionamento da análise do código-fonte de um programa em Pascal, segundo o protótipo. Esta explanação será mais compacta e sucinta, o que não interfere no entendimento da mesma, pois serão excluídos os detalhes de programação pertinentes ao protótipo.

A análise do código-fonte de um programa em Pascal segue, basicamente, as seguintes etapas:

a) seleção do programa a ser analisado;
b) início da análise do programa através da análise dos arquivos de código-fonte dependentes;
c) armazenamento dos dados utilizados para a análise.

Partindo do pressuposto de que o projeto analisado já está selecionado tem-se as seguintes seqüências de comandos, conforme a figura 19:

FIGURA 19 - Descrição dos passos efetuados pelo protótipo para a realização da análise

1) Instancia-se a classe TProjectAnaliser, passando como parâmetro o nome do projeto selecionado. Ao instanciar-se a classe TProjectAnaliser, a mesma cria uma instância das classes TUnitAnaliser e TObjectContainer. Durante a criação do objeto TUnitAnaliser, o mesmo instancia a classe TToken para a extração dos tokens do projeto selecionado. Deve-se notar que neste instante, todos os objetos necessários, como as listas de valores, também são instanciadas pelas respectivas classes;
2) Inicia-se a análise do projeto através do método StartAnalisys, que inicia a análise do programa-fonte associado ao projeto, executando o método StartAnalisys de TUnitAnaliser;
3) TUnitAnaliser.StartAnalisys inicia o processamento do programa-fonte, executando o método AnaliseToken para cada token extraído;
4) O método AnaliseToken, verifica se este token é uma palavra reservada ou um símbolo não definido. Se for uma palavra reservada, é chamado o método TokenIsReservedWord, caso contrário chama-se o método TokenIsNotDefined.
5) O método TokenIsReservedWord verifica se a palavra reservada é uma das palavras que contenham instruções inerentes à análise como: VAR, USES, CONST, TYPE, etc.
6) Caso esta palavra reservada seja VAR, é chamado o método GetUnitVars que extrairá as variáveis adicionando-as na lista de variáveis da unit. Neste ponto é feita a verificação do atributo IsInInterface, que indica se as variáveis declaradas são públicas ou privadas.
7) Caso a palavra reservada seja USES, é chamado o método GetUnitNamesFromUses, que armazenará na lista de units utilizadas, efetuando automaticamente a sua análise. Deve-se dar especial atenção à este método, pois para cada unit encontrada, é iniciado uma análise, pois as informações utilizadas na unit atual poderão estar presentes na unit indicada no USES. Deste modo, a análise desta unit somente prosseguirá ao fim da análise das units encontradas na cláusula USES.
8) Caso a palavra reservada seja CONST, é chamado o método GetUnitConstants, que extrairá todas as constantes declaradas por esta cláusula. O mesmo procedimento

```
    é adotado para a palavra TYPE.

9)  Quando palavra reservada é FUNCTION ou PROCEDURE, o processamento é diferente:
    primeiro, certifica-se de que esta não é somente a declaração da subrotina. Caso
    não seja somente e declaração, todo o corpo da rotina é extraído e gravado em um
    arquivo temporário. Após isto, é instanciada uma classe TUnitAnaliser para
    verificar este arquivo temporário. Isto permite que a subrotina tenha uma
    análise adequada, da mesma forma como é feito com o código-fonte da unit.

10) O método TokenIsNotDefined, que é chamado quando o identificador não é uma
    palavra reservada, verifica se este identificador é uma variável, constante,
    tipo, ou subrotina que já tenha sido analisada nesta unit ou nas units as quais
    tenha feito uso (USES). Caso o identificador não seja reconhecido o mesmo é
    armazenado na lista de palavras não reconhecidas do analisador.

11) O método TokenIsNotDefined, procura em todas as listas de identificadores para
    encontrar o tipo do mesmo. Quando encontrado, incrementa o contador de
    utilização deste identificador. Caso este identificador seja uma subrotina,
    adiciona o local ao qual a mesma está sendo referenciada na sua UseList (lista
    de usos). Esta UseList é utilizada para a formação do grafo de utilização da
    subrotina.
```

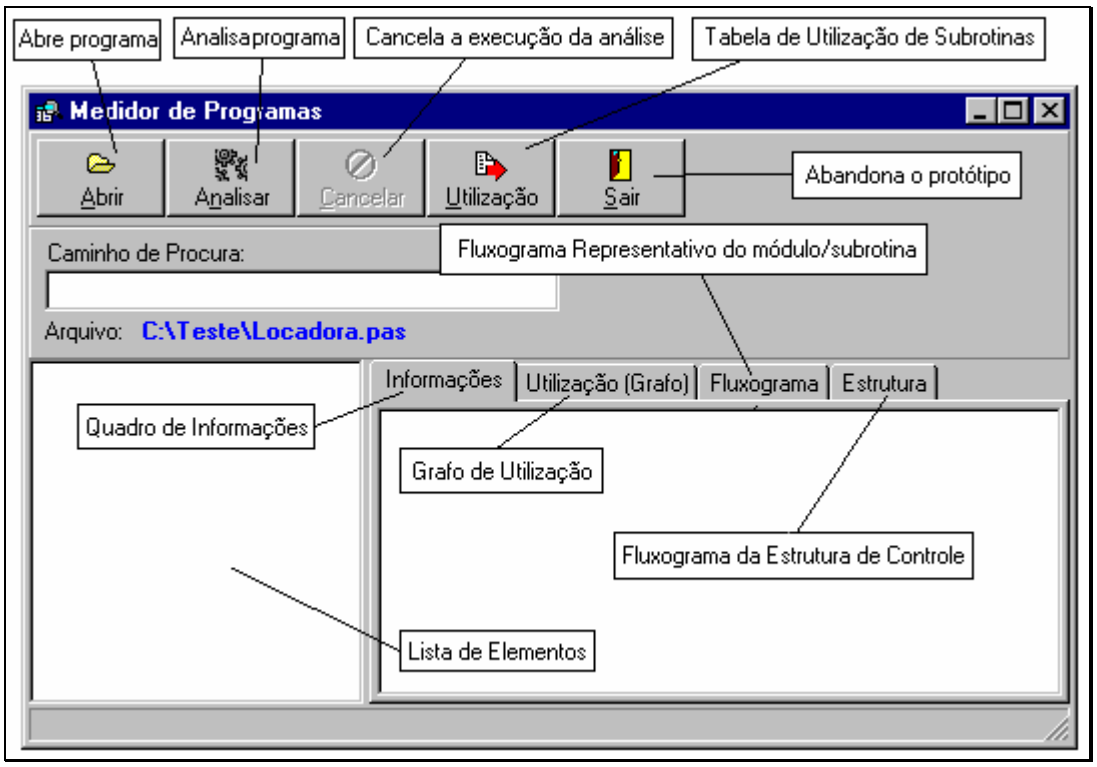
Esta é uma descrição reduzida de como o protótipo efetua a análise do código-fonte. Para uma melhor compreensão do funcionamento deste protótipo, aconselha-se verificar o código-fonte responsável pela análise do código conforme presente no anexo 1.

4.4 DESCRIÇÃO DAS TELAS

Para efetuar a análise de um projeto através do protótipo, procede-se da seguinte maneira: Após executar o protótipo, será exibida uma tela conforme exibido na figura 20. A partir disto, seleciona-se o programa ao qual deseja-se efetuar a análise, sendo que este projeto deverá ser um arquivo de programa Pascal (.PAS). Para tanto deve-se clicar no botão *Abrir*. Após selecionar o programa, será habilitado o botão *Analisar*. Isto indica que o programa está pronto para efetuar a análise.

Ao clicar no botão *Analisar* o protótipo iniciará a análise de todos os programas que estão relacionados ao arquivo selecionado.

FIGURA 20 - Tela principal do protótipo

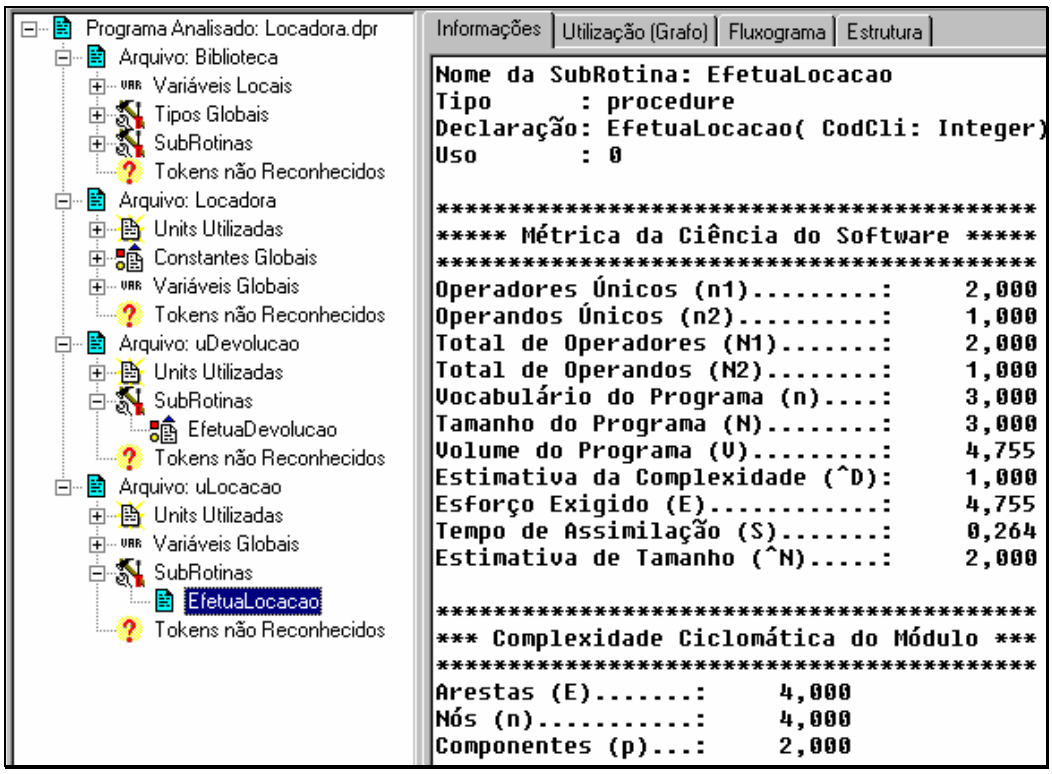


Ao término da análise, os resultados são exibidos na lista de elementos do projeto, conforme a figura 21. A lista de elementos exibe todos os elementos que podem ser extraídos da análise do código-fonte (units, variáveis, constantes, subrotinas, tokens não reconhecidos, etc.) sendo que para cada elemento selecionado, o quadro de informações é atualizado. Da mesma maneira, dependendo do elemento selecionado, são exibidos ou não o grafo de utilização, o fluxograma e a estrutura de controle.

Além desta lista de elementos, pode-se visualizar informações sobre os elementos selecionados através do quadro de informações. O quadro de informações fornece dados sobre o elemento que foi analisado como o número de vezes que uma variável foi utilizada, seu tipo, valor de inicialização, etc. Além destas informações triviais, o quadro de utilização fornece os resultados das métricas da Ciência de Software e da Complexidade Ciclomática, como pode ser visualizado na figura 21:

Os resultados exibidos nas figuras que seguem foram obtidos através da análise de programas-fontes que encontram-se listados no anexo 2. Através do anexo 2 será possível um melhor entendimento e também uma confrontação de opiniões.

FIGURA 21 - Resultado da análise



Adicionalmente, podem ser visualizados os grafos de utilização e o fluxograma, conforma a figura 22 e 23. Estas opções estão disponíveis somente para os elementos presentes nas subrotinas e nos arquivos analisados, visto que os mesmos são informações que são retiradas diretamente do programa-fonte. O grafo de utilização exibe, graficamente, o escopo de utilização de uma determinada subrotina. O fluxograma tenta fornecer de forma visual o grafo do fluxo de controle do programa/subrotina analisado.

FIGURA 22 - Grafo de utilização da procedure *Imprime*.

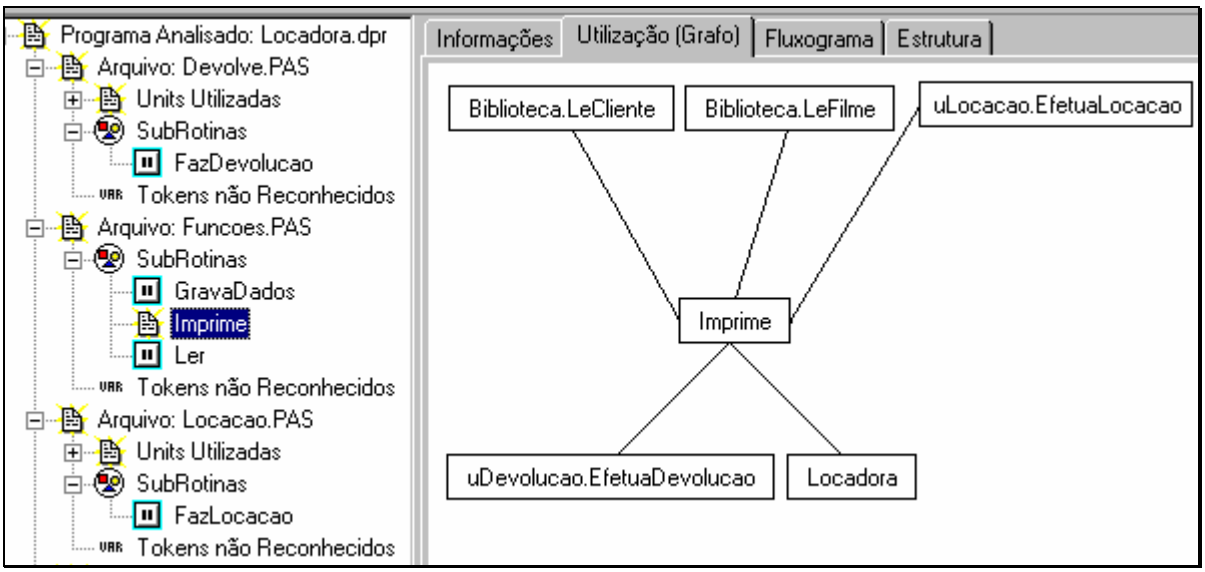
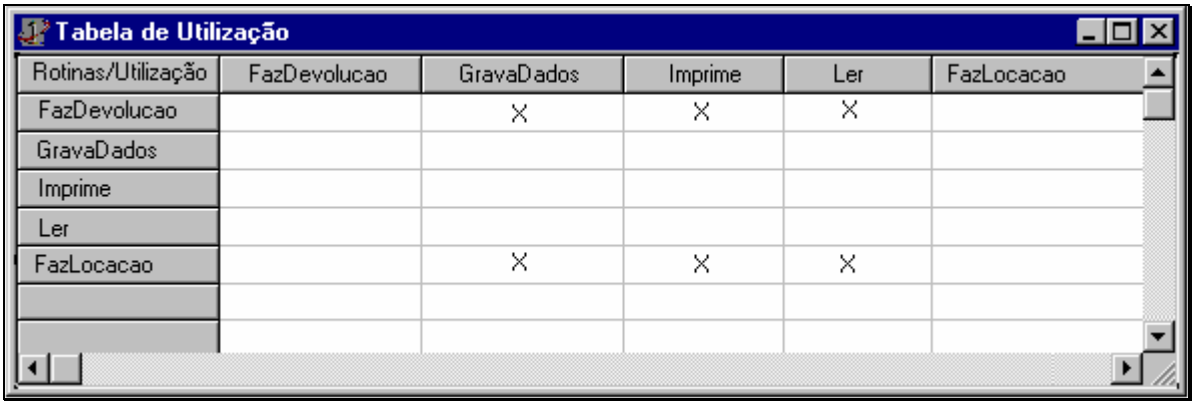


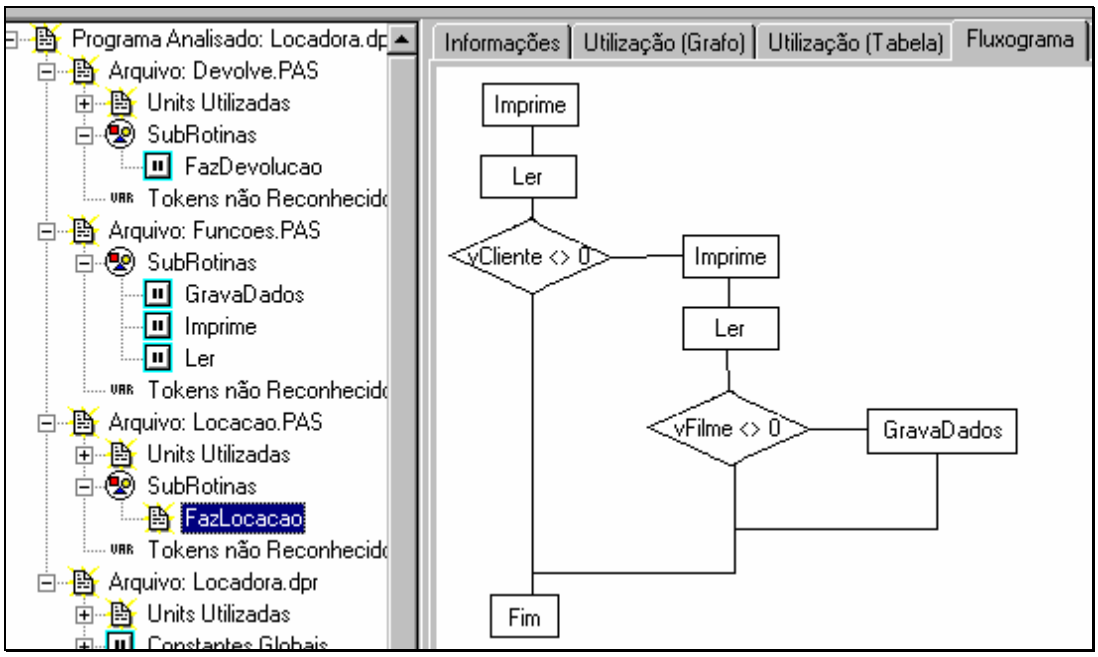
FIGURA 23 - Tabela de Utilização das SubRotinas



A figura 23 representa a tela de utilização das subrotinas. Esta tela é uma variação do grafo de utilização de subrotina, com a exceção de que esta tela é disponibilizada, exibindo todas as subrotinas detectadas pela análise que são utilizadas no sistema. Esta tabela tenta dar uma idéia da hierarquia de utilização das subrotinas dentro do sistema, por exemplo, a subrotina *FazLocacao* utiliza-se das seguintes subrotinas: *Imprime*, *Ler* e *GravaDados*.

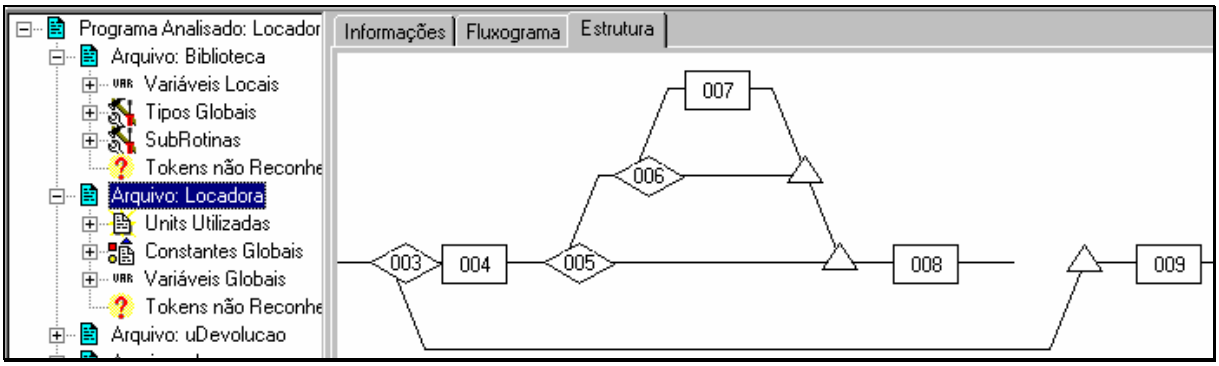
A figura 24 exibe o tradicional fluxograma de execução da subrotina.

FIGURA 24 - Fluxograma da subrotina FazLocacao



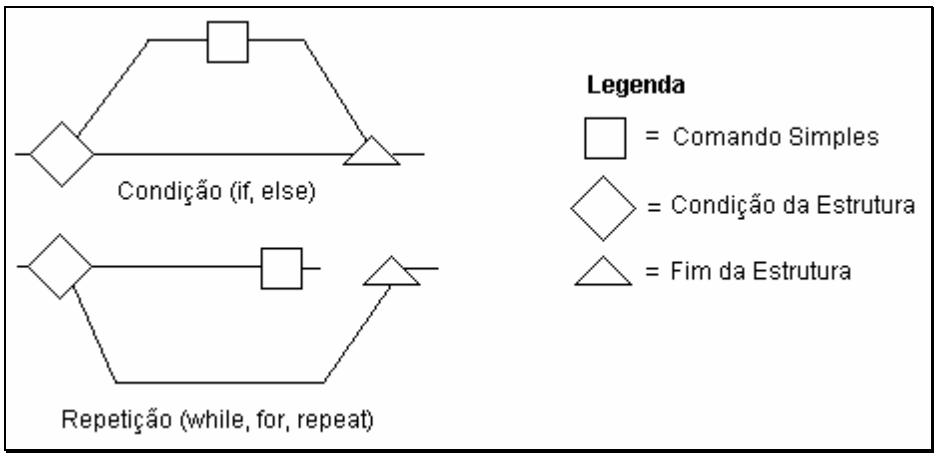
A figura 25, mostra como é disponibilizado o grafo de estrutura. Este grafo funciona como uma visualização do “esqueleto” do software/subrotina, permitindo analisar seu fluxo de controle do mesmo. Através do mesmo é possível detectar eventuais problemas do fluxo de processamento das informações. Não são representados as seguintes estruturas: *while*, *repeat*, *for* e *case*.

FIGURA 25 - Grafo representativo da estrutura da subrotina



A estrutura, exibe maiores informações acerca do programa-fonte como: comandos de condição e de repetição: foi convencionado que os comandos condicionais e de repetição seguiriam formato conforme exibidos na figuras 26.

FIGURA 26 - Notação utilizada para a exibição do gráfico de estrutura



5 CONCLUSÃO

5.1 CONSIDERAÇÕES FINAIS

A utilização de métricas de código-fonte somente se tornará viável a partir do momento que existir uma ferramenta adequada que forneça de maneira rápida, simples e correta, as informações sobre o mesmo. No entanto, ferramentas desta categoria são muito difíceis de serem encontradas, pois as mesmas devem levar em consideração aspectos relativos à linguagem ao qual a mesma se destina.

No caso do protótipo deste trabalho, o mesmo está intimamente ligado (dentro de suas limitações) à linguagem Pascal, sendo que este não terá utilização nenhuma para a análise de código-fonte de qualquer outra linguagem, devido às particularidades da linguagem hospedeira. Esta é uma das principais causas da escassez de ferramentas desta natureza. As linguagens de programação atuais sofrem constantes modificações e as ferramentas de análise de código-fonte, muitas vezes não conseguem acompanhar estas mudanças.

No protótipo desenvolvido, pode-se verificar que os resultados disponibilizados poderão ser de grande utilidade para o programador que está trabalhando no sistema. Se este programador ainda não está familiarizado com algum módulo, basta que o mesmo efetue uma análise e estude os resultados.

Outra forma de facilitar o entendimento é a utilização do fluxograma exibido pelo protótipo. Um fluxograma exibe, visualmente, o funcionamento do programa/subrotina. Isto facilita a compreensão de um determinado trecho de código.

A utilização de métricas de código-fonte não pode ser considerada como a melhor maneira de compreensão de um determinado sistema. O conhecimento do negócio ainda é fundamental, juntamente com o conhecimento de aspectos da linguagem no qual o mesmo foi concebido.

No entanto, uma análise de um programa-fonte, fornecido por uma ferramenta desta natureza, pode auxiliar sensivelmente a compreensão do mesmo. Neste ponto, pode-se observar as particularidades de implementação adotadas pelo programador que construiu o software. Como é sabido, cada pessoa (programador/analista) possui uma maneira própria de

codificação. Quando uma segunda pessoa inicia uma manutenção de sistemas prontos, há uma certa dificuldade inicial para a compreensão da lógica adotada na construção do mesmo, principalmente se o criador do software não adotou algumas recomendações básicas para a codificação de sistemas, definidas pela Engenharia de Software. Neste momento pode-se visualizar a utilidade desta ferramenta, sendo que o tempo gasto para a compreensão do sistema poderá ser bem menor devido às informações disponibilizadas pela mesma.

As declarações feitas anteriormente podem não representar a realidade quando um sistema é bem projetado, especificado e documentado. Neste caso, quando existe uma boa especificação e documentação do software, o trabalho de compreensão será facilitado, pois a documentação servirá como um guia para a pessoa que está mantendo o software. Como esta situação não é uma regra, mas infelizmente uma rara exceção, a utilização destas ferramentas pode ser uma valiosa contribuição para o entendimento do software. Até mesmo nos casos onde são encontrados sistemas bem documentados, muitas vezes esta documentação está desatualizada ou em desacordo com as atuais especificações do software.

De uma maneira ou de outra, a utilização de ferramentas de análise de código-fonte sempre será um auxílio extra para uma equipe de desenvolvimento. Através dos resultados obtidos, é possível verificar se o software está bem construído e é possível verificar quais as partes do software que serão afetadas com uma eventual mudança em uma determinada subrotina.

Ferramentas desta natureza podem, inclusive, serem utilizadas para efeitos didáticos. É muito mais fácil para um estudante poder visualizar o impacto que alguma alteração feita no código-fonte provocará no sistema. Da mesma maneira é mais simples visualizar a lógica de construção do programa através de um grafo que represente o fluxo de informações de uma determinada parte do software.

A utilização e manutenção das métricas de software é um passo em busca de uma maior qualidade do mesmo, mas a utilização desta somente será concretizada a partir do momento que houver mais preocupação com o projeto, especificação e documentação do software existente. Como pode-se notar, é uma questão de *cultura profissional*. Atualmente, os consumidores de tecnologia estão cada vez mais exigentes no que se refere à qualidade do software. Em razão disto, as empresas, naturalmente, estão buscando soluções para a

obtenção da mesma e gradativamente estarão voltando-se para as práticas de planejamento, especificação, codificação, documentação e manutenção recomendadas pela engenharia de software.

5.2 SUGESTÕES

Como sugestões para trabalhos futuros ou para a extensão deste, tem-se:

a) utilização de uma técnica formal para a construção de um analisador sintático adequado à linguagem Pascal, o que facilitaria e daria mais agilidade a análise;
b) definição de uma tabela de palavras reservadas;
c) extensão da análise para as demais estruturas condicionais da linguagem;
d) extensão da analise para o ambiente Delphi.

ANEXO 1 - LISTAGEM DO CÓDIGO-FONTE DO PROTÓTIPO RELATIVO À ANALISE DO CÓDIGO-FONTE

Listagem do código-fonte da Unit uBaseAnaliser.pas

```
unit uBaseAnaliser;

interface
uses Classes, uToken, uObjectContainer, uBaseDefinitions;

type
   TBaseAnaliser = Class
   private
      FUnitName    : String;
      FLocalConsts : TStorageList;
      FGlobalConsts: TStorageList;
      FLocalVars   : TStorageList;
      FGlobalVars  : TStorageList;
      FLocalTypes  : TStorageList;
      FGlobalTypes : TStorageList;
      FNotDefined  : TStringList;
      function GetLineCount: Integer;
   protected
      Token: TToken;
      IsInInterface: Boolean; { Indica se a análise está na Seção Interface }
      procedure AnaliseToken; virtual;
      function GetDescriptionToken: String;
      { Verifica se o token é NotDeclared mesmo }
      function  VerifyNotDeclared(const  aToken:  String;  aData:  Pointer):  Pointer;
virtual;
      procedure FinalizeAnalisys; virtual;
   public
      Owner: TBaseAnaliser;
      constructor Create(const aUnitName: String);
      destructor Destroy; override;
      procedure StartAnalisys;
      procedure TokenIsReservedWord; virtual;
      procedure TokenIsNotDefined;
      procedure GetUnitConstants;
      procedure GetUnitVars;
      procedure GetUnitTypes;
      function TypeOfIdentifier(const aId: String): TTypeIdentifier;
      property UnitName    : String read FUnitName;
      property LocalConsts : TStorageList read FLocalConsts;
      property GlobalConsts: TStorageList read FGlobalConsts;
      property LocalVars   : TStorageList read FLocalVars;
      property GlobalVars  : TStorageList read FGlobalVars;
      property LocalTypes  : TStorageList read FLocalTypes;
      property GlobalTypes : TStorageList read FGlobalTypes;
      property NotDefined  : TStringList read FNotDefined;
      property LineCount   : Integer read GetLineCount;
   end;

implementation
uses SysUtils, uLib;

constructor TBaseAnaliser.Create(const aUnitName: String);
begin
   inherited Create;
```

```
   Owner := nil;

   FLocalConsts := TStorageList.Create;
   FGlobalConsts := TStorageList.Create;

   FLocalVars := TStorageList.Create;
   FGlobalVars := TStorageList.Create;

   FLocalTypes := TStorageList.Create;
   FGlobalTypes := TStorageList.Create;

   FNotDefined := TStringList.Create;
   FNotDefined.Sorted := True;

   FUnitName := GetLocalOfFileName(aUnitName);
   Token := TToken.Create(FUnitName);

   { Caso a unit analisada não possua interface <programa principal> todas
     as declarações serão consideradas globais }
   IsInInterface := True;
end;

destructor TBaseAnaliser.Destroy;
var
   i: Integer;
begin
   Token.Free;
   FNotDefined.Free;

   for i := 0 to Pred(FLocalConsts.Count) do
      Dispose(FLocalConsts.GetValue(FLocalConsts.Strings[i]));
   for i := 0 to Pred(FGlobalConsts.Count) do
      Dispose(FGlobalConsts.GetValue(FGlobalConsts.Strings[i]));

   for i := 0 to Pred(FLocalVars.Count) do
      Dispose(FLocalVars.GetValue(FLocalVars.Strings[i]));
   for i := 0 to Pred(FGlobalVars.Count) do
      Dispose(FGlobalVars.GetValue(FGlobalVars.Strings[i]));

   for i := 0 to Pred(FLocalTypes.Count) do
      Dispose(FLocalTypes.GetValue(FLocalTypes.Strings[i]));
   for i := 0 to Pred(FGlobalTypes.Count) do
      Dispose(FGlobalTypes.GetValue(FGlobalTypes.Strings[i]));

   inherited;
end;

procedure TBaseAnaliser.StartAnalisys;
begin
   { Pega o primeiro token }
   while not Token.Eof do
   begin
      AnaliseToken;
      Token.GetNextToken;
   end;
   FinalizeAnalisys;
end;

procedure TBaseAnaliser.FinalizeAnalisys;
begin
end;

procedure TBaseAnaliser.AnaliseToken;
begin
   if Token.CurrentTokenType = ttId then
```

```
      case TypeOfIdentifier(Token.CurrentToken) of
         tiReservedWord: TokenIsReservedWord;
      else
         TokenIsNotDefined;
      end;
end;

function TBaseAnaliser.TypeOfIdentifier(const aId: String): TTypeIdentifier;
begin
   Result := tiNotDefined;
   { Se for uma palavra reservada, chama a rotina de tratamento de
     palavras reservadas }
   if IsReservedWord(aId) then
      Result := tiReservedWord;
end;

procedure TBaseAnaliser.TokenIsReservedWord;
begin
   { Analisa as palavras reservadas básicas }
   if SameText(Token.CurrentToken, 'Const') then
      GetUnitConstants
   else if SameText(Token.CurrentToken, 'Var') then
      GetUnitVars
   else if SameText(Token.CurrentToken, 'Type') then
      GetUnitTypes
   else if SameText(Token.CurrentToken, 'Implementation') then
      IsInInterface := False;
end;

function TBaseAnaliser.GetDescriptionToken: String;
begin
   Result := Token.CurrentToken;
   if Token.CurrentTokenType = ttid then
      Result := ' ' + Result;
end;

procedure TBaseAnaliser.TokenIsNotDefined;
var
   aValue: Pointer;
begin
   { Verifica somente os tokens que são identificadores ..por enquanto.. }
   if Token.CurrentTokenType <> ttId then
      exit;

   aValue := VerifyNotDeclared(Token.CurrentToken, nil);
   if SameText(Token.CurrentToken, RemoveExtension(FUnitName)) then
      aValue := Self;

   { Caso o token não seja encontrado em nenhum dos casos acima... }
   if (aValue = nil) and (FNotDefined.IndexOf(Token.CurrentToken) = -1) then
      FNotDefined.Add(Token.CurrentToken);
end;

function  TBaseAnaliser.VerifyNotDeclared(const  aToken:  String;  aData:  Pointer):
Pointer;
var
   aBase : TBaseAnaliser;
begin
   Result := nil;
   aBase := Self;
   while (aBase <> nil) and (Result = nil) do
   begin
      { Verifica se o token encontrado é algo já reconhecido }
      Result := aBase.LocalConsts.GetValue(aToken);
      if Result <> nil then
```

```
      begin
         Inc(TConst(Result)^.ConstCount);
         break;
      end;

      Result := aBase.GlobalConsts.GetValue(aToken);
      if Result <> nil then
      begin
         Inc(TConst(Result)^.ConstCount);
         break;
      end;

      Result := aBase.LocalVars.GetValue(aToken);
      if Result <> nil then
      begin
         Inc(TVar(Result)^.VarCount);
         break;
      end;

      Result := aBase.GlobalVars.GetValue(aToken);
      if Result <> nil then
      begin
         Inc(TVar(Result)^.VarCount);
         break;
      end;

      aBase := aBase.Owner;
   end;
end;

procedure TBaseAnaliser.GetUnitConstants;
var
   aConst: TConst;
   aName,
   aAux: String;
begin
   Token.GetNextToken;
   while (not IsReservedWord(Token.CurrentToken)) and (not Token.Eof) do
   begin
      New(aConst);
      aName := Token.CurrentToken;
      aAux := '';
      Token.GetNextToken;
      if Token.CurrentToken = ':' then
      begin
         Token.GetNextToken;
         while (Token.CurrentToken <> '=') and (not Token.Eof) do
         begin
            aAux := aAux + GetDescriptionToken;
            Token.GetNextToken;
         end;
      end;
      aConst^.ConstType := aAux;
      Token.GetNextToken;
      aAux := '';
      while (Token.CurrentToken <> ';') and (not Token.Eof) do
      begin
         aAux := Token.CurrentToken;
         Token.GetNextToken;
      end;
      aConst^.ConstValue := aAux;

      { Adiciona a constante pesquisada na lista de constantes correspondente.
        a lista será escolhida através da variável IsInInterface, que indicará
        qual a atual seção da unit que está sendo tratada }
```

```
      aConst^.ConstCount := 0;
      if IsInInterface then
         FGlobalConsts.AddValue(aName, aConst)
      else
         FLocalConsts.AddValue(aName, aConst);

      { Pega o próximo token }
      Token.GetNextToken;
   end;
   Token.RollBack;
end;

procedure TBaseAnaliser.GetUnitVars;
var
   aVar: TVar;
   aAux: String;
   aListVars: TList;
   i: Integer;
begin
   aListVars := TList.Create;
   Token.GetNextToken;
   while (not IsReservedWord(Token.CurrentToken)) and (not Token.Eof) do
   begin
      New(aVar);
      aVar^.VarName := Token.CurrentToken;
      aAux := '';
      Token.GetNextToken;
      { Caso exista declarações de variáveis separadas por ',' estas são
        empilhadas para que todas tenhas os seus tipos definidos corretamente ao
        final da sequencia }
      while (Token.CurrentToken <> ':') and (not Token.Eof) do
      begin
         if Token.CurrentToken = ',' then
         begin
            aListVars.Add(aVar);
            New(aVar);
         end;
         Token.GetNextToken;
         aVar^.VarName := Token.CurrentToken;
         Token.GetNextToken;
      end;
      aListVars.Add(aVar);

      { Pega a definição do tipo das variáveis }
      if Token.CurrentToken = ':' then
      begin
         Token.GetNextToken;
         while (Token.CurrentToken <> '=') and (Token.CurrentToken <> ';') and
               (not Token.Eof) do
         begin
            aAux := aAux + GetDescriptionToken;
            Token.GetNextToken;
         end;
      end;

      for i := 0 to Pred(aListVars.Count) do
         TVar(aListVars.Items[i])^.VarType := aAux;

      aAux := '';
      if Token.CurrentToken <> ';' then
         Token.GetNextToken;
      while (Token.CurrentToken <> ';') and (not Token.Eof) do
      begin
         aAux := Token.CurrentToken;
         Token.GetNextToken;
```

```
      end;
      aVar^.VarValue := aAux;

      { Adiciona a constante pesquisada na lista de constantes correspondente.
        a lista será escolhida através da variável IsInInterface, que indicará
        qual a atual seção da unit que está sendo tratada }
      if IsInInterface then
         for i := 0 to Pred(aListVars.Count) do
         begin
            TVar(aListVars.Items[i])^.VarCount := 0;
            FGlobalVars.AddValue(TVar(aListVars.Items[i])^.VarName,
aListVars.Items[i]);
         end
      else
         for i := 0 to Pred(aListVars.Count) do
         begin
            TVar(aListVars.Items[i])^.VarCount := 0;
            FLocalVars.AddValue(TVar(aListVars.Items[i])^.VarName,
aListVars.Items[i]);
         end;

      { Libera a lista de variáveis }
      aListVars.Clear;

      { Pega o próximo token }
      Token.GetNextToken;
   end;
   Token.RollBack;
   aListVars.Free;
end;

procedure TBaseAnaliser.GetUnitTypes;
var
   aType: TType;
   aDesc: String;
   aGetInherited: Boolean;
begin
   Token.GetNextToken;
   while (not IsReservedWord(Token.CurrentToken)) and (not Token.Eof) do
   begin
      New(aType);
      aType^.TypeName := Token.CurrentToken;
      aDesc := '';
      Token.GetNextToken;
      { Sempre depois da declaração do nome do novo tipo segue-se o sinal de '='
        mas mesmo assim, segue o 'if' abaixo }
      if Token.CurrentToken = '=' then
         Token.GetNextToken;
      while (Token.CurrentToken <> ';') and (not Token.Eof) do
      begin
         aDesc := aDesc + GetDescriptionToken;
         { Se for um record, percorre todos os campos, ignorando-os, pois somente
           será necessário saber que a variável é um record, nada maia }
         if SameText(Token.CurrentToken,'record') then
            while (not SameText(Token.CurrentToken, 'end')) and (not Token.Eof) do
               Token.GetNextToken;
         { Se for uma classe, pega somente a herança, caso exista, e ignora as
           demais declarações (procedures, function, properties ...), pois
           atualmente só nos interessa o tipo da variável }
         if SameText(Token.CurrentToken,'class') then
         begin
            aGetInherited := False;
            while (not SameText(Token.CurrentToken, 'end')) and (not Token.Eof) do
            begin
               Token.GetNextToken;
```

```
               if (Token.CurrentToken = '(') and (not aGetInherited) then
               begin
                  aDesc := aDesc + GetDescriptionToken;
                  Token.GetNextToken;
                  aDesc := aDesc + GetDescriptionToken;
                  Token.GetNextToken;
                  aDesc := aDesc + GetDescriptionToken;
                  aGetInherited := True;
               end;
            end;
         end;

         Token.GetNextToken;
      end;
      aType^.TypeDesc := aDesc;
      if IsInInterface then
         FGlobalTypes.AddValue(aType^.TypeName, aType)
      else
         FLocalTypes.AddValue(aType^.TypeName, aType);

      Token.GetNextToken;
   end;
   Token.RollBack;
end;

function TBaseAnaliser.GetLineCount: Integer;
begin
   Result := Token.LineCount;
end;


end.
```

Listagem do código-fonte da Unit uUnitAnaliser.pas

```
      unit uUnitAnaliser;

      interface
      uses Classes, uObjectContainer, uDefinitions, uBaseAnaliser;

      type
         TUnitAnaliser = Class(TBaseAnaliser)
         private
            FSubRoutine  : String;
            FUnitContainer: TObjectContainer;
            FLstUnits: TStringList;
            FGlobalSubRoutines: TStorageList;
            FOperators: TStorageList;
            FFirstCommand,
            FCommand: TCommand;
            FCounter: Integer;
            procedure CreateFileWithSubRoutine(aSubRoutineName, aFileName: String);
            procedure AddCommand(tcType: TCommandType);
            function  GetFather(aCommand: TCommand): TCommand;
            procedure AddOperator(aToken: String);
         protected
            procedure AnaliseToken; override;
            procedure GetCommands;
            function    VerifyNotDeclared(const  aToken:  String;  aData:  Pointer):
Pointer; override;
            procedure FinalizeAnalisys; override;
         public
            constructor    Create(const    aUnitName:    String;    aObjectContainer:
TObjectContainer);
```

```
            destructor Destroy; override;
            procedure TokenIsReservedWord; override;
            procedure GetUnitNamesFromUses;
            procedure GetSubRoutinesSpecification;
            property SubRoutine       : String read FSubRoutine;
            property UnitsUsed        : TStringList read FLstUnits;
            property GlobalSubRoutines: TStorageList read FGlobalSubRoutines;
            property FirstCommand: TCommand read FFirstCommand;
            property Command: TCommand read FCommand;
            property Operators: TStorageList read FOperators;
         end;

         TRecSubRoutines = packed record
            SubRoutineName,
            SubRoutineType,
            Declaration   : String;
            UseList       : TStringList;
            Analiser      : TUnitAnaliser;
            RoutineCount  : Integer;
         end;
         TSubRoutine = ^TRecSubRoutines;

      implementation
      uses SysUtils, uToken, uBaseDefinitions, uLib;

      constructor  TUnitAnaliser.Create(const  aUnitName:  String;  aObjectContainer:
TObjectContainer);
      begin
         FUnitContainer := aObjectContainer;

         FLstUnits := TStringList.Create;
         FLstUnits.Sorted := True;

         FGlobalSubRoutines := TStorageList.Create;
         FOperators := TStorageList.Create;

         inherited Create(aUnitName);
         { Este object não deverá ser liberado por esta classe, visto que o mesmo
           é passado como parâmetro, assim sendo, deverá ser liberado na unit que
           instanciou esta classe }

         FFirstCommand := nil;
         FCommand := nil;  //Inicializa os contenedores de comandos
         FCounter := 1;
      end;

      destructor TUnitAnaliser.Destroy;
      var
         i: Integer;
      begin
         FLstUnits.Free;
         for i := 0 to Pred(FGlobalSubRoutines.Count) do
            Dispose(FGlobalSubRoutines.GetValue(FGlobalSubRoutines.Strings[i]));
         FGlobalSubRoutines.Free;
         for i := 0 to Pred(FOperators.Count) do
            Dispose(FOperators.GetValue(FOperators.Strings[i]));
         FOperators.Free;

         FFirstCommand.Free;
         inherited;
      end;

      procedure TUnitAnaliser.AnaliseToken;
      begin
         case TypeOfIdentifier(Token.CurrentToken) of
```

```
            tiReservedWord: TokenIsReservedWord;
            else
               { Armazena o comando }
               GetCommands;
         end;
      end;

      procedure TUnitAnaliser.FinalizeAnalisys;
      var
         xCommand: TCommand;
      begin
         if FCommand <> nil then
         begin
            xCommand := TCommand.Create;
            xCommand.Counter := FCounter;
            xCommand.Prior := FCommand;
            xCommand.CommandType := tiCode;
            xCommand.Command := 'Fim';

            FCommand.Next := xCommand;
            FCommand := xCommand;
         end;
      end;

      procedure TUnitAnaliser.TokenIsReservedWord;
      begin
         { Analisa as palavras reservadas básicas }
         if SameText(Token.CurrentToken, 'Const') then
            GetUnitConstants
         else if SameText(Token.CurrentToken, 'Var') then
            GetUnitVars
         else if SameText(Token.CurrentToken, 'Type') then
            GetUnitTypes
         else if SameText(Token.CurrentToken, 'Implementation') then
            IsInInterface := False
         { e as palavras reservadas que são de responsabilidade desta classe }
         else if SameText(Token.CurrentToken, 'Uses') then
            GetUnitNamesFromUses
         else if (SameText(Token.CurrentToken, 'Procedure')) or
                 (SameText(Token.CurrentToken, 'Function')) then
            GetSubRoutinesSpecification
         // Pensar em outra forma de verificação para os if's abaixo... talvez uma
lista...
         else if (SameText(Token.CurrentToken, 'Begin')) then
            AddCommand(tiBegin)
         else if (SameText(Token.CurrentToken, 'If')) then
            AddCommand(tiIf)
         else if (SameText(Token.CurrentToken, 'While')) then
            AddCommand(tiWhile)
         else if (SameText(Token.CurrentToken, 'For')) then
            AddCommand(tiFor)
         else if (SameText(Token.CurrentToken, 'Repeat')) then
            AddCommand(tiRepeat)
         else if (SameText(Token.CurrentToken, 'End')) then
            AddCommand(tiEnd)
      end;

      procedure TUnitAnaliser.AddCommand(tcType: TCommandType);
      var
         vAux: TCommand;
      begin
         if tcType = tiBegin then
         begin
            { Esta solicitação vem do método TokenIsReservedWord, pois a palavra
begin precisa estar lá }
```

```
            if (FCommand <> nil) and (FCommand.HasBegin) and (not FCommand.HasEnd)
then
               Inc(FCommand.CountBegin);
            exit;
         end;
         if tcType = tiEnd then
         begin
            if FCommand <> nil then
            begin
               if FCommand.CountBegin = 0 then
                  FCommand.HasEnd := True
               else Dec(FCommand.CountBegin);
            end;
            exit;
         end;

         vAux := TCommand.Create;
         vAux.Counter := FCounter;
         Inc(FCounter);
         vAux.CommandType := tcType;
         case tcType of
            tiIf,
            tiWhile,
            tiFor   : vAux.NeedChild := True;
            tiRepeat: vAux.HasEval := True;
            tiElse  : begin
                      vAux.NeedChild := True;
                      vAux.HasEval := True;
                      end;
            tiCode  : vAux.Command := Token.CurrentToken;
         end;

         { Adiciona o operador à lista de operadores }
         if (tcType <> tiCode) then
            AddOperator(Token.CurrentToken);

         while (not vAux.HasEval) and (not Token.Eof) do
         begin
            Token.GetNextToken;
            if (SameText(Token.CurrentToken, 'Then')) or
               (SameText(Token.CurrentToken, 'Do')) or
               (SameText(Token.CurrentToken, 'Of')) or
               (Token.CurrentToken = ';') then
               vAux.HasEval := True
            else
               vAux.Command := vAux.Command + Self.GetDescriptionToken;

            if TypeOfIdentifier(Token.CurrentToken) <> tiReservedWord then
               TokenIsNotDefined;
            { Procede com a verificação normal dos tokens extraídos }
            if Token.CurrentTokenType in [ttMath, ttRelat, ttLogical] then
               AddOperator(Token.CurrentToken);
         end;

         { Verifica se o próximo token é um BEGIN }
         Token.GetNextToken;
         if SameText(Token.CurrentToken, 'Begin') then
            vAux.HasBegin := True
         else Token.RollBack;

         if FCommand = nil then
         begin
            FFirstCommand := vAux;
            FCommand := vAux
         end
```

```
         else if (FCommand.HasEnd) or
                 ((not FCommand.HasBegin) and (not FCommand.NeedChild)) then
         begin
            FCommand.Next := vAux;
            vAux.Prior := FCommand;
            vAux.Father := FCommand.Father;
            FCommand := vAux;
         end
         else if (FCommand.HasBegin) or (FCommand.NeedChild) then
         begin
            if FCommand.LastChild = nil then
            begin
               FCommand.Child := vAux;
               FCommand.LastChild := vAux;
            end
            else begin
               FCommand.LastChild.Next := vAux;
               vAux.Prior := FCommand.LastChild;
               FCommand.LastChild := vAux;
            end;
            FCommand.NeedChild := False;
            vAux.Father := FCommand;
         end;

         if (vAux.NeedChild) or ((vAux.HasBegin) and (not vAux.HasEnd)) then
            FCommand := vAux;

         if (not FCommand.NeedChild) and
            ((not FCommand.HasBegin) or
            (FCommand.HasBegin and FCommand.HasEnd)) and
            (FCommand.Father <> nil) then
         begin
            FCommand := GetFather(FCommand.Father);
         end;
      end;

      procedure TUnitAnaliser.GetCommands;
      begin
         { Para fazer parte do fluxograma, não precisa incluir símbolos isolados ou
           comentários }
         if  Token.CurrentTokenType  in  [ttSymbol,  ttMath,  ttRelat,  ttAttrib,
ttLogical, ttComm] then
         begin
            if  Token.CurrentTokenType  in  [ttMath,  ttRelat,  ttAttrib,  ttLogical]
then
               AddOperator(Token.CurrentToken);
            exit;
         end;

         { Verifica o tipo do token, e executa a ação necessária }
         TokenIsNotDefined;

         AddCommand(tiCode);
      end;

      { Retorna o último pai que aceita filhos ou que está no fim da lista }
      function TUnitAnaliser.GetFather(aCommand: TCommand): TCommand;
      begin
         if (aCommand.Father = nil) or (aCommand.Father.HasBegin) then
            Result := aCommand
         else
            Result := GetFather(aCommand.Father);
      end;
```

```
      function   TUnitAnaliser.VerifyNotDeclared(const   aToken:   String;   aData:
Pointer): Pointer;
      var
         aBase: TUnitAnaliser;
         i: Integer;
         aUses: String;
      begin
         { Antes de verificar o token aqui, verifica pelo método da classe pai }
            Result := inherited VerifyNotDeclared(aToken, nil);
         if aData = nil then
            aData := Self;

         aBase := Self;
         while (aBase <> nil) and (Result = nil) do
         begin
            { Verifica se o token encontrado é algo já reconhecido }
            Result := aBase.FGlobalSubRoutines.GetValue(aToken);
            if (Result <> nil) and (not SameText(aToken,FSubRoutine)) then
            begin
               aUses := TUnitAnaliser(aData).UnitName;
               if TUnitAnaliser(aData).Owner <> nil then
               begin
                  aUses := TUnitAnaliser(aData).Owner.UnitName;
                  aUses      :=      RemoveExtension(aUses)      +      '.'      +
TUnitAnaliser(aData).SubRoutine;
               end
               else aUses := RemoveExtension(aUses);
               TSubRoutine(Result)^.UseList.Add(aUses);
               Inc(TSubRoutine(Result)^.RoutineCount);
               break;
            end;

            aBase := TUnitAnaliser(aBase.Owner);
         end;

         { Verifica se a subrotina está declarada em outra unit }
         if Result = nil then
         begin
            for i := 0 to Pred(FLstUnits.Count) do
            begin
               Result                                                          :=
TUnitAnaliser(FUnitContainer.GetAnaliser(FLstUnits.Strings[i])).GlobalSubRoutines.G
etValue(aToken);
               if Result <> nil then
               begin
                  AddOperator(aToken);
                  Break;
               end;
            end;
         end;

         { Caso seja uma unit, verifica se esta subrotina está declarada em outras
           units através da lista de units do seu 'Owner' }
         if (Result = nil) and (Owner <> nil) then
         begin
            for i := 0 to Pred(TUnitAnaliser(Owner).FLstUnits.Count) do
            begin
               Result                                                          :=
TUnitAnaliser(FUnitContainer.GetAnaliser(TUnitAnaliser(Owner).FLstUnits.Strings[i])
).VerifyNotDeclared(aToken, aData);
               if Result <> nil then
                  Break;
            end;
         end;
```

```
         { Verifica a lista de identificadores que eventualmente podem não estar
           declarados, para adicionar o mesmos à lista de operadores. Neste caso
           poderão ser functions ou procedure do próprio Object Pascal ou Delphi.
           Esta lista é carregada no início da análise, juntamente com a lista de
           palavras reservadas }
         if (Result = nil) then
            if IsNonDeclaredOperator(Token.CurrentToken) then
            begin
            Result := self;  //Adiciona algum valor à variável para que o item não
entre em identificadores não reconhecidos já que o mesmo faz parte da lista.
            if Result <> nil then
               AddOperator(aToken);
            end;
      end;

      procedure TUnitAnaliser.GetUnitNamesFromUses;
      var
         xUnitAnaliser: TUnitAnaliser;
         aUnitName: String;
      begin
         Token.GetNextToken;
         { Varre toda a lista de arquivos do USES, até encontrar um ';' }
         while (Token.CurrentToken <> ';') and (not Token.Eof) do
         begin
            { Se for um identificador, então é uma unit. }
            if Token.CurrentTokenType = ttId then
            begin
               { Verifica se o Fonte especificado já está analisado }
               aUnitName := Token.CurrentToken + '.PAS';
               try
                  xUnitAnaliser := TUnitAnaliser.Create(aUnitName, FUnitContainer);
                  FLstUnits.Add(xUnitAnaliser.UnitName);
                  if FUnitContainer.GetAnaliser(xUnitAnaliser.UnitName) = nil then
                  begin
                     { Inicia a análise do programa }
                     xUnitAnaliser.StartAnalisys;
                     { Adiciona o analisador no Container de Objetos }
                     FUnitContainer.AddAnaliser(xUnitAnaliser.UnitName,
xUnitAnaliser);
                  end else
                     { Não é necessário efetuar a análise, a mesma já foi feita }
                     xUnitAnaliser.Free;
               except
               end;
            end;
            Token.GetNextToken;
         end;
      end;

      procedure  TUnitAnaliser.CreateFileWithSubRoutine(aSubRoutineName,  aFileName:
String);
      var
         vQtdBegin: Integer;
         aTextFile: String;
         aFile    : TextFile;
         vFirstBegin: Boolean;
      begin
         vFirstBegin := True;
         vQtdBegin := 0;
         aTextFile := aSubRoutineName + #59 + #13;

         //Caso o primeiro token seja um '(' então irá trocá-lo por var
         if Token.CurrentToken = '(' then
         begin
```

```
            Token.GetNextToken;
            aTextFile := aTextFile + 'var ';
         end;
         if Token.CurrentToken = ':' then
         begin
            Token.GetNextToken;
            Token.GetNextToken;
         end;

         AssignFile(aFile, aFileName);
         Rewrite(aFile);
         try
            while ((vQtdBegin > 0) or (vFirstBegin)) and (not Token.Eof) do
            begin
               if (Token.CurrentToken = ')') and (vFirstBegin) then
                  while (not Token.Eof) and (Token.CurrentToken <> ';') do
                     Token.GetNextToken;
               if SameText(Token.CurrentToken,'Begin') then
               begin
                  Inc(vQtdBegin);
                  vFirstBegin := False;
               end;
               if SameText(Token.CurrentToken, 'End') then
                  Dec(vQtdBegin);
               aTextFile := aTextFile + GetDescriptionToken;
               Token.GetNextToken;
            end;
         finally
            { Grava o arquivo em disco }
            WriteLn(aFile, aTextfile);
            CloseFile(aFile);
         end;
      end;

      procedure TUnitAnaliser.GetSubRoutinesSpecification;
      var
         aSubRoutine: TSubRoutine;
         aDeclaration,
         aName, aType: String;
      begin
         aType := Token.CurrentToken; //Tipo da SubRotina

         { O próximo token deverá ser o nome da subrotina; }
         Token.GetNextToken;
         aName := Token.CurrentToken;
         Token.GetNextToken;

         { Verifica se a rotina já está declarada }
         if FGlobalSubRoutines.IndexOf(aName) = -1 then
         begin
            New(aSubRoutine);
            aSubRoutine^.SubRoutineType := aType;
            aSubRoutine^.SubRoutineName := aName;
            aSubRoutine^.UseList := TStringList.Create;
            aSubRoutine^.Analiser := nil;
            aSubRoutine^.RoutineCount := 0;
         end else begin
            { Se a rotina já está declarada, significa que os códigos que seguem
são
              da implementação da mesma }
            aSubRoutine := TSubRoutine(FGlobalSubRoutines.GetValue(aName));
            try
               { Após criar um arquivo com a implementação desta subrotina,
                 pode-se efetuar uma análise da mesma }
               CreateFileWithSubRoutine(aName, 'MEPROV.PAS');
```

```
               aSubRoutine^.Analiser      :=      TUnitAnaliser.Create('MEPROV.PAS',
FUnitContainer);
               aSubRoutine^.Analiser.Owner := Self;
               aSubRoutine^.Analiser.FSubRoutine := aSubRoutine^.SubRoutineName;
               aSubRoutine^.Analiser.StartAnalisys;
            finally
               DeleteFile('MEPROV.PAS');
            end;
         end;

         if aSubRoutine.Analiser = nil then
         begin
            aDeclaration := aName;
            while (Token.CurrentToken <> ';') and (not Token.Eof) do
            begin
               if Token.CurrentToken = '(' then
                  while (Token.CurrentToken <> ')') and (not Token.Eof) do
                  begin
                     aDeclaration := aDeclaration + GetDescriptionToken;
                     Token.GetNextToken;
                  end;
               aDeclaration := aDeclaration + GetDescriptionToken;
               Token.GetNextToken;
            end;
            aSubRoutine^.Declaration := aDeclaration;
            if IsInInterface then
               FGlobalSubRoutines.AddValue(aName, aSubRoutine);
         end;
      end;

      procedure TUnitAnaliser.AddOperator(aToken: String);
      var
         xOperator: TOperator;
      begin
         xOperator := FOperators.GetValue(Token.CurrentToken);
         if xOperator = nil then
         begin
            New(xOperator);
            xOperator^.OperatorName  := Token.CurrentToken;
            xOperator^.OperatorCount := 1;
            FOperators.AddValue(Token.CurrentToken, xOperator)
         end
         else Inc(xOperator^.OperatorCount);
      end;

end.
```

ANEXO 2 - LISTAGEM DO CÓDIGO-FONTE DO PROGRAMA EXEMPLO

Listagem do Programa LOCADORA.DPR.

```
program Locadora;

uses Funcoes, Locacao, Devolve;

const ENTRADA = 'Sistema de Locadora';

var
   Opcao: Byte;

begin
   Opcao = -1;

   while Opcao <> 0 do
   begin
      Imprime(Entrada);
      Imprime('Escolha uma op‡Æo: ');
      Ler(Opcao);

      if opcao = 1 then
      begin
         FazLocacao;
         EmiteNota;
      end else
         if opcao = 2 then
            FazDevolucao;
   end;
end.
```

Listagem do código-fonte da unit FUNCOES.PAS

```
unit Funcoes;

interface

procedure Imprime(vStr: String);
procedure Ler(vNum: Byte);
procedure GravaDados(vCliente, vFilme: Byte; vOp: Char);
{ A procedure abaix nÆo ser  implementada...
  procedure EmitaNota }

implementation

procedure Imprime(vStr: String);
begin
   writeln(vStr);
end;

procedure Ler(vNum: Byte);
begin
```

```
   readln(vNum);
end;

procedure GravaDados(vCliente, vFilme: Byte; vOp: Char);
begin
   if vOp = 'D' then
      { Grava no arquivo a Devolu‡Æo }
      ;
   else
      { Insere no arquivo de dados a Loca‡Æo }
      ;
end;
```

Listagem do código-fonte da unit LOCACAO.PAS

```
unit Locacao;

interface
procedure FazLocacao;

implementation
uses Funcoes;

procedure FazLocacao;
var
   vCliente, vFilme: Byte;
begin
   Imprime('Informe o C¢digo do Cliente: ');
   Ler(vCliente);
   if vCliente <> 0 then
   begin
      Imprime('Informe o C¢digo do Filme: ');
      Ler(vFilme);
      if vFilme <> 0 then
         GravaDados(vCliente, vFilme, 'D');
   end;
end;
```

Listagem do código-fonte da unit DEVOLVE.PAS

```
unit Devolve;

interface
procedure FazDevolucao;

implementation
uses Funcoes;

procedure FazDevolucao;
var
   vFilme: Byte;
begin
   Imprime('Informe o C¢digo do Filme: ');
   Ler(vFilme);
   if vFilme <> 0 then
      GravaDados(vCliente, vFilme, 'L');
end;
```

REFERÊNCIAS BIBLIOGRÁFICAS

[ARI1993] ARIGOFLU, Ali. A methodology for cost estimation. **Software engineering notes**. Vol 18 Nro 2, 1993. Pg 96-105.

[CAP1995] CAPUTO, Geraldo, et al. Um estudo sobre métricas de produtividade no desenvolvimento de sistemas. **Instituto Brasileiro de Pesquisa em Informática**, 1995.

[FAR1985] FARLEY, Richard E. **Software engineering concepts**. California, 1995.

[FER1995] FERNANDES, Agnaldo Aragon. **Gerência de software através de métricas**. São Paulo : Atlas, 1995.

[FUC1995] FUCK, Mônica Andréa. **Estudo e aplicação de métricas da qualidade do processo de desenvolvimento de aplicações em banco de dados**. Blumenau, 1995. Monografia (Bacharelado em Ciências da Computação) Centro de Ciências Exatas e Naturais, FURB.

[FUR1998] FURLAN, José Roberto. **Modelagem de objetos através da UML**. Rio de Janeiro : Makron Books, 1998.

[KAN1995] KAN, Stephen H. **Metrics and models in software quality engineering**. Massachutes : Addison Wesley, 1995.

[MCD1993] MCDERMIC, John. **Software Engineer's Reference Book**. CRC : Flórida. 1993.

[MÖL1993] MÖLLER, K.H.; Paulish, D.J. **Software metrics**. Munich : IEEE Press, 1993.

[MAF1992] MAFFEO, Bruno. **Engenharia de software e especificação de sistemas**. Rio de Janeiro : Campus, 1992.

[OLI1996] OLIVEIRA, Adelize G. **Explorando Delphi 2**. Florianópolis : Visual Books, 1996.

[OSL1997] OSLER, Dan. Grobsman, Steve. **Aprenda em 21 dias Delphi 2**. Rio de Janeiro : Campus, 1997.

[SHE1993] SHEPPERD, Martin. **Derivation and validation of software metrics**. Oxford : Claredon. 1993.

[YOU1995] YOURDON, Edward. **Decline & fall of the american programers**. New Jersey : Yourdon Press.1995.