UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

FERRAMENTA PARA CÁLCULO DE MÉTRICAS EM SOFTWARES ORIENTADOS A OBJETOS

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

PATRÍCIA REGINA RAMOS DA SILVA SEIBT

BLUMENAU, NOVEMBRO/2001

2001/2-37

FERRAMENTA PARA CÁLCULO DE MÉTRICAS EM SOFTWARES ORIENTADOS A OBJETOS

PATRÍCIA REGINA RAMOS DA SILVA SEIBT

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. Marcel Hugo — Orientador na FURB

____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

____

Prof. Marcel Hugo

____

Prof. Everaldo Arthur Grahl

____

Prof. Dalton Solano dos Reis

AGRADECIMENTO

A minha mãe Ida Teresa Damásio pelo apoio, incentivo ao estudo, carinho e dedicação.

Ao meu marido Alcino Seibt e filho Matheus pelo amor, carinho e compreensão.

Ao professor Marcel Hugo que me orientou neste trabalho com paciência, dedicação e sabedoria.

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

A todos muito obrigado.

SUMÁRIO

AGRADECIMENTO III

LISTA DE FIGURAS VII

LISTA DE QUADROS IX

RESUMO X

ABSTRACT XI

1 INTRODUÇÃO 1

1.1 OBJETIVOS 2

1.2 ORGANIZAÇÃO DO TEXTO 3

2 MÉTRICAS 4

2.1 A ORIGEM DOS SISTEMAS MÉTRICOS 5

2.2 IMPORTÂNCIA DAS MÉTRICAS 5

2.3 OBJETIVOS DA UTILIZAÇÃO DE MÉTRICAS 8

2.4 MÉTRICAS TRADICIONAIS 10

2.4.1 ANÁLISE POR PONTOS DE FUNÇÃO (FPA) 10

2.4.2 COCOMO (CONSTRUCTIVE COST MODEL) 11

2.4.3 LINHAS DE CÓDIGO (LOC - LINES OF CODE) 11

2.4.4 MÉTRICA DE CIÊNCIA DO SOFTWARE 12

2.4.5 MÉTRICA DA COMPLEXIDADE CICLOMÁTICA 13

3 MÉTRICAS PARA ORIENTAÇÃO A OBJETOS (OO) 15

3.1 MÉTRICAS DE ANÁLISE 18

3.1.1 PORCENTAGEM DE CLASSES-CHAVE 18

3.1.2 NÚMEROS DE CENÁRIOS DE UTILIZAÇÃO 19

3.2 MÉTRICAS DE PROJETO 19

3.2.1 CONTAGEM DE MÉTODOS 19
3.2.2 MÉTODOS PONDERADOS POR CLASSE (WMC -WEIGTHED METHODS PER CLASS) 20
3.2.3 RESPOSTA DE UMA CLASSE (RFC – RESPONSE FOR A CLASS) 20
3.2.4 PROFUNDIDADE DA ÁRVORE DE HERANÇA (DTI - DEPTH OF INHERITANCE TREE) 21
3.2.5 NÚMERO DE FILHOS (NOC – NUMBER OF CHILDREN) 21
3.2.6 FALTA DE COESÃO (LCOM - LACK OF COHESION) 22
3.2.7 ACOPLAMENTO ENTRE OBJETOS (CBO – COUPLING BETWEEN OBJECT CLASSES) 24
3.2.8 UTILIZAÇÃO GLOBAL 25
3.3 MÉTRICAS DE CONSTRUÇÃO 26
3.3.1 TAMANHO DO MÉTODO 26
3.3.1.1 QUANTIDADE DE MENSAGENS ENVIADAS 26
3.3.1.2 LINHAS DE CÓDIGO (LOC) 27
3.3.1.3 MÉDIA DO TAMANHO DOS MÉTODOS 28
3.3.2 PERCENTUAL COMENTADO 28
3.3.3 COMPLEXIDADE DO MÉTODO 28
3.3.4 TAMANHO DA CLASSE 29
3.3.4.1 QUANTIDADE DE MÉTODOS DE INSTÂNCIA PÚBLICOS EM UMA CLASSE 29
3.3.4.2 QUANTIDADE DE MÉTODOS DE INSTÂNCIA EM UMA CLASSE 30
3.3.4.3 QUANTIDADE DE ATRIBUTOS POR CLASSE 30
3.3.4.4 MÉDIA DE ATRIBUTOS POR CLASSE 31
3.3.4.5 QUANTIDADE DE MÉTODOS DE CLASSE EM UMA CLASSE 31
3.3.4.6 QUANTIDADE DE VARIÁVEIS DE CLASSE EM UMA CLASSE 31

3.3.5 QUANTIDADE DE CLASSES ABSTRATAS 32
3.3.6 USO DE HERANCA MÚLTIPLA 32
3.3.7 QUANTIDADE DE MÉTODOS SOBRESCRITOS POR UMA SUBCLASSE 33
3.3.8 QUANTIDADE DE MÉTODOS HERDADOS POR UMA SUBCLASSE 33
3.3.9 QUANTIDADE DE MÉTODOS ADICIONADOS POR UMA SUBCLASSE 33
3.3.10 ÍNDICE DE ESPECIALIZAÇÃO 34
4 DESENVOLVIMENTO DO TRABALHO 35
4.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO 35
4.2 MÉTRICAS SELECIONADAS 37
4.3 ESPECIFICAÇÃO DO PROTÓTIPO 38
4.3.1 DIAGRAMA DE CASO DE USO 38
4.3.2 DIAGRAMA DE CLASSES 39
4.3.3 DIAGRAMA DE SEQÜÊNCIA 45
4.4 IMPLEMENTAÇÃO 48
4.4.1 TÉCNICAS E FERRAMENTAS UTILIZADAS 48
4.4.2 OPERACIONALIDADE DA IMPLEMENTAÇÃO 52
4.4.3 ESTUDO DE CASO 59
4.4.4 RESULTADOS OBTIDOS 64
5 CONCLUSÕES 65
5.1 EXTENSÕES 66
ANEXO 1 – CÓDIGO FONTE ANALISADO 67
REFERÊNCIAS BIBLIOGRÁFICAS 86

LISTA DE FIGURAS

Figura 1 – EXEMPLO DE CÁLCULO DE COMPLEXIDADE CICLOMÁTICA 14
Figura 2 – EXEMPLO DE DIAGRAMA DE CLASSES 18
Figura 3 – EXEMPLO DE FALTA DE COESÃO 23
Figura 4 – CLASSES GEOMÉTRICAS COM ATRIBUTOS E MÉTODOS 25
Figura 5 – EXEMPLO DE CONTAGEM DE MENSAGENS ENVIADAS 27
Figura 6 – DIAGRAMA DE SINTAXE DO CABEÇALHO DE *PROCEDURE* 35
Figura 7 – DIAGRAMA DE SINTAXE DA DEFINIÇÃO DE UMA CLASSE 36
Figura 8 – DIAGRAMA DE SINTAXE DA DEFINIÇÃO DE UM ATRIBUTO 36
Figura 9 – DIAGRAMA DE SINTAXE DA DEFINIÇÃO DE UM MÉTODO 37
Figura 10 – DIAGRAMA DE CASO DE USO 39
Figura 11 – DIAGRAMA DE CLASSES 40
Figura 12 – DIAGRAMA DE SEQÜÊNCIA DA ANÁLISE DO FONTE 46
Figura 13 – DIAGRAMA DE SEQÜÊNCIA DO CÁLCULO DAS MÉTRICAS 47
Figura 14 – TELA PRINCIPAL DO PROTÓTIPO 53
Figura 15 – TELA DE ABERTURA DO PROJETO A SER ANALISADO 53
Figura 16 – LOG DAS UNITS ANALISADAS 54
Figura 17 – TELA DE SELEÇÃO DE PROJETOS NA BASE DE DADOS 54
Figura 18 – TELA DE INFORMAÇÕES DO PROJETO 55
Figura 19 – TELA DE FILTRO DAS MÉTRICAS E CLASSES 56
Figura 20 – TELA DE INFORMAÇÕES DAS MÉTRICAS CALCULADAS 56
Figura 21 – TELA DE LEGENDAS DAS MÉTRICAS 57
Figura 22 – TELA DE TOTAIS DO PROJETO 57
Figura 23 – TELA DE DETALHES DAS MÉTRICAS POR CLASSE 58

Figura 24 – TELA DE OPÇÕES DO PROJETO ... 58
Figura 25 – DIAGRAMA DE CLASSES DO SISTEMA DE MATRÍCULA ... 60
Figura 26 – INFORMAÇÕES EXTRAÍDAS DO SISTEMA DE MATRÍCULA ... 61
Figura 27 – TOTAIS DO PROJETO DO SISTEMA DE MATRÍCULA ... 61
Figura 28 – MÉTRICAS DO SISTEMA DE MATRÍCULA ... 62
Figura 29 – MÉTRICAS DA CLASSE TALUNO ... 62
Figura 30 – MÉTRICAS DA CLASSE TGRADUACAO ... 63
Figura 31 – MÉTRICAS DA CLASSE TCURSO ... 63

LISTA DE QUADROS

Quadro 1 – MÉTODO CRIACLASSE..48
Quadro 2 – MÉTODO ANALISA UNIT..49
Quadro 3 – MÉTODO EXTRACTTOKEN..50
Quadro 4 – MÉTODO CALCULADTI..50
Quadro 5 – MÉTODO CALCULACBO..51
Quadro 6 – MÉTODO ARMAZENACBO..52
Quadro 7 – ESTUDO DE CASO..59
Quadro 8 – CÓDIGO FONTE UNIT UPRINCIPAL..67
Quadro 9 – CÓDIGO FONTE UNIT C_CURSO..68
Quadro 10 – CÓDIGO FONTE UNIT C_DISCIPLINA..69
Quadro 11 – CÓDIGO FONTE UNIT C_ALUNO..70
Quadro 12 – CÓDIGO FONTE UNIT C_GRADUAÇÃO..71
Quadro 13 – CÓDIGO FONTE UNIT C_POSGRADUAÇÃO..72
Quadro 14 – CÓDIGO FONTE UNIT C_MATRICULA..73
Quadro 15 – CÓDIGO FONTE UNIT C_DISMATRI..74
Quadro 16 – CÓDIGO FONTE UNIT UALUNO..75
Quadro 17 – CÓDIGO FONTE UNIT UDISCIPLINA..76
Quadro 18 – CÓDIGO FONTE UNIT UCURSO..78
Quadro 19 – CÓDIGO FONTE UNIT C_VETORDISCIPLINA..79
Quadro 20 – CÓDIGO FONTE UNIT C_VETORALUNO..80
Quadro 21 – CÓDIGO FONTE UNIT C_VETORCURSO..82
Quadro 22 – CÓDIGO FONTE UNIT C_VETORMATRICULA..84

RESUMO

Este trabalho descreve o desenvolvimento e a construção de um protótipo de uma ferramenta capaz de analisar o código fonte de um projeto orientado a objetos em Delphi, extraindo as classes seus métodos e atributos para posterior cálculo de métricas para softwares Orientados a Objetos. A ferramenta permite calcular métricas de projeto e de construção, como por exemplo Profundidade da árvore de herança e métodos ponderados por classe.No final deste trabalho é apresentado um estudo de caso contendo um diagrama de classes para o problema apresentado, bem como o código fonte analisado e o cálculo das métricas.

ABSTRACT

This work describes the development and the construction of an archetype of a tool capable to analyze the code source of a guided design the objects in Delphi, extracting the classes its methods and attributes for posterior calculation of metric for Object Oriented softwares. A case study is presented, in the end of this work, containing a class diagram for the presented problem, as well the analyzed source code and the calculation of metrics.

1 INTRODUÇÃO

Conforme Arthur (1994), para gerenciar a produtividade e a qualidade, é necessário saber se ambas estão melhorando ou piorando. Isto implica a necessidade de métricas de software que rastreiem as tendências do desenvolvimento do software. De acordo com Ambler (1998), a gestão de projetos e produtos de software somente atinge determinado nível de eficácia e exatidão se existir medidas que possibilitem gerenciar através de fatos os aspectos econômicos do software.

Medidas, em Engenharia de Software, são chamadas de métricas, que podem ser definidas como métodos de determinar quantitativamente a extensão em que o projeto, o processo e o produto de software têm certos atributos (Fernandes, 1995). Segundo Ambler (1998), métrica é uma medida pura e simples. Ao medir as coisas de maneira consistente fica-se a par do que está sendo feito e qual é a sua eficiência.

As métricas são utilizadas para estimar projetos, melhorar os esforços de desenvolvimento e selecionar as ferramentas, entre outros.

De acordo com DeMarco (1989), para manter o processo de software sob controle deveria ser instituída uma política de medição e documentação meticulosa durante o projeto para assegurar que todos os dados cruciais estejam disponíveis no final. Qualquer aspecto que necessite de controle também necessita de medição. Segundo Fernandes (1995), o objetivo da aplicação da medição é fornecer aos gerentes e engenheiros de software um conjunto de informações tangíveis para planejar, realizar estimativas, gerenciar e controlar os projetos com maior precisão.

Segundo DeMarco (1989), a maioria das pessoas concorda que a produtividade do software é mais baixa do que deveria ser. Ao medir-se alguma coisa, espera-se compreender os fatores que a influenciam e a partir daí descobrir formas de manipulá-las de maneira vantajosa. Este é o objetivo da métrica do software, o estudo dos fatores que influenciam a produtividade através da qualificação dos projetos de desenvolvimento de software.

De acordo com Martin (1994), uma das maiores preocupações na indústria de software hoje é a necessidade de se criar softwares e sistemas corporativos, que são mais complexos, de uma forma mais veloz e a um custo mais baixo. As técnicas baseadas em objetos simplificam o projeto de sistemas complexos. Conforme Ambler (1998), uma das razões, na

maioria das vezes a principal, pela qual as empresas mudam para Orientação a Objetos é que elas querem aperfeiçoar a qualidade das aplicações que estão desenvolvendo, as quais querem desenvolver sistemas melhores, mais baratos e em menos tempo. As métricas fornecem uma medida da qualidade e produtividade do projeto.

Segundo Ambler (1998), algumas métricas que podem ser utilizadas em softwares orientados a objetos são:

a) contagem de métodos;
b) número de atributos de instância;
c) profundidade da árvore de herança;
d) número de filhos;
e) utilização global;
f) quantidade de atributos de classe;
g) tamanho do método;
h) resposta do método;
i) comentários por método.

Outros trabalhos realizaram a pesquisa e aplicação de métricas tradicionais como Fuck (1995) e Possamai (2000). Cardoso (1999), realizou um trabalho sobre métricas para softwares orientados a objetos tais como: número de classes reutilizadas, classes descendentes, classes sucessoras, métodos novos, métodos redefinidos, métodos herdados.

1.1 OBJETIVOS

O objetivo principal deste trabalho foi construir um protótipo de uma ferramenta que permita calcular algumas métricas pré-determinadas para softwares orientados a objetos, a partir da análise do código fonte em Delphi.

Os objetivos específicos do trabalho são:

a) identificar as métricas e elaborar sua forma de cálculo;
b) obter mais informações estatísticas sobre softwares orientados a objetos.

1.2 ORGANIZAÇÃO DO TEXTO

O primeiro capítulo apresenta a introdução e os objetivos pretendidos com a elaboração do trabalho.

O segundo capítulo inicia descrevendo o que são métricas. Apresenta a origem dos sistemas métricos, importância e objetivos. O capítulo reúne os conceitos de algumas métricas tradicionais.

O terceiro capítulo apresenta as métricas para orientação a objetos, suas características e diferenças em relação às métricas tradicionais. Em seguida são apresentadas algumas métricas para orientação a objetos divididas em três categorias: análise, projeto e construção.

O quarto capítulo é apresentado o desenvolvimento do trabalho, incluindo a descrição da especificação e da implementação do protótipo. Também no quarto capítulo são apresentadas as métricas selecionadas para implementação.

O quinto capítulo finaliza o trabalho, apresentando as conclusões, limitações e sugestões para novos trabalhos.

2 MÉTRICAS

Segundo Pressmann (1995), as métricas de software, do ponto de vista de medição, podem ser divididas em duas categorias: medidas diretas e indiretas. Para Cordeiro (2000), pode-se considerar como medidas diretas do processo de engenharia de software o custo e o esforço aplicados no desenvolvimento e manutenção do software e do produto, a quantidade de linhas de código produzidas, velocidade de execução e o total de defeitos registrados durante um determinado período de tempo. As medidas indiretas do produto incluem funcionalidade, qualidade, complexidade, eficiência, confiabilidade e muitos outros aspectos do produto. As medidas indiretas são mais difíceis de serem medidas e avaliadas (Pressmann, 1995).

As métricas de software podem ser divididas também, sob o ponto de vista da aplicação, em três categorias: métricas de produtividade, métricas de qualidade e métricas técnicas. As métricas de produtividade concentram-se na saída do processo de engenharia de software, as métricas de qualidade indicam o quanto o software atende aos requisitos definidos pelo cliente e as métricas técnicas concentram-se nas características do software, tais como complexidade lógica e modularidade (Cordeiro, 2000).

Outra divisão do domínio das métricas vista em Pressmann (1995):

a) métricas orientadas ao tamanho: são medidas diretas do software e do processo por meio do qual ele é desenvolvido. Estas medidas são derivadas a partir de atributos de tamanho do software como linhas de código, esforço, custo, quantidade de documentação;
b) métricas orientadas à função: são medidas indiretas do software e do processo através do qual o software é desenvolvido. Estas métricas concentram-se na funcionalidade ou qualidade do software, um exemplo destas métricas é a técnica de Análise por Pontos de Função;
c) métricas orientadas às pessoas: estas métricas utilizam informações sobre a maneira pela qual as pessoas desenvolvem software e percepções humanas sobre a efetividade das ferramentas e métodos.

2.1 A ORIGEM DOS SISTEMAS MÉTRICOS

As métricas originaram-se da aplicação de cálculos para quantificar indicadores sobre o processo de desenvolvimento de um software, sendo adotadas a partir de 1970. Existem quatro tendências desta tecnologia, vista em Moller (1993):

a) medida de complexidade do código: foi desenvolvido em meados de 1970, os códigos métricos foram fáceis de se obter desde que fossem calculados pelo próprio código automatizado;
b) estimativa do custo de um projeto de software: esta técnica foi desenvolvida em meados de 1970, estimando o trabalho e o tempo gasto para se desenvolver um software, baseando-se além de outros fatores, na quantidade de linhas de código necessárias para a implementação;
c) garantia da qualidade do software: estas técnicas foram melhoradas significativamente entre os anos de 1970 e 1980. Neste caso, se da ênfase à identificação de informações que faltam durante as varias fases do ciclo de vida do software;
d) processo de desenvolvimento do software o projeto de software tornou-se grande e mais complexo sendo que a necessidade de se controlar este processo foi emergencial. O processo incluiu a definição do ciclo de vida do software pela seqüência das fases. Dando ênfase maior no gerenciamento e controle de recursos deste projeto.

Os desenvolvedores de sistemas, a partir do aparecimento destas tendências, começaram então a utilizar as métricas no propósito de melhorar o processo de desenvolvimento do software.

2.2 IMPORTÂNCIA DAS MÉTRICAS

Conforme DeMarco (1989), não se pode controlar o que não se pode medir. Segundo Jacobson (1992), um método necessário para controlar o desenvolvimento de um software é a utilização de métricas. As métricas podem medir cada etapa do desenvolvimento e vários aspectos do produto. Medições e métricas auxiliam a entender o processo usado para se

desenvolver um produto. O processo é medido a fim de melhorá-lo e o produto é medido para aumentar a sua qualidade (Pressman, 1995).

Para Cordeiro (2000), medidas são necessárias para analisar qualidade e produtividade do processo de desenvolvimento e manutenção bem como do produto de software construído. Métricas técnicas são necessárias para qualificar a performance técnica dos produtos do ponto de vista do desenvolvedor. Por outro lado métricas funcionais são necessárias para qualificar a performance dos produtos pela perspectiva do usuário. Medidas funcionais devem ser independentes das decisões do desenvolvimento técnico e implementação e podem ser utilizadas para comparar a produtividade de diferentes técnicas e tecnologias.

Segundo Pressman (1995), o software é medido por várias razões:

a) indicar a qualidade do produto;
b) avaliar a produtividade das pessoas que produzem o produto;
c) avaliar os benefícios em termos de produtividade e qualidade derivados de novos métodos e ferramentas de software;
d) formar uma linha básica de estimativas;
e) ajudar a justificar os pedidos de novas ferramentas ou treinamento adicional.

Conforme Fernandes (1995), a gestão de projetos e de produtos de software somente atinge determinado nível de eficácia e exatidão se houver métricas e medidas que possibilitem gerenciar através de fatos e gerenciar os aspectos econômicos do software, que geralmente são negligenciados em organizações de desenvolvimento.

Segundo Cardoso (1999), a realidade da informática atual aponta para o sério problema da má qualidade nos produtos desenvolvidos. A única preocupação das equipes de desenvolvimento seria o cumprimento dos prazos. Com esta atitude, é entregue ao cliente apenas as funções básicas do seu sistema, com os prováveis defeitos, que após seriam novamente avaliados pelos desenvolvedores e muitas vezes o cliente tendo que pagar adicionalmente um produto que já havia sido concluído.

De acordo com Fernandes (1995), se os conceitos de engenharia para o desenvolvimento de software fossem adotados, seria possível medir a qualidade e produtividade dos projetos/processos e do produto, compará-los com metas preestabelecidas,

verificar tendências e fundamentar o aperfeiçoamento contínuo do processo. A adoção desses conceitos requer:

a) um processo de software definido em termos de políticas de desenvolvimento, procedimentos, padrões, métodos, técnicas e ferramentas;
b) medições relativas a atributos do projeto, como prazos, recursos, custo e esforço;
c) medições relativas a atributos do processo, como cobertura de testes, nível de detecção e remoção de defeitos, nível de complexidade do projeto, produtividade;
d) medições relativas a atributos do produto, como confiabilidade, nível de detecção de defeitos, índice de manutenção, tamanho do software;
e) medições relativas à satisfação do cliente;
f) sistema de garantia da qualidade, incluindo gerencia de configuração, planos de qualidade, inspeção formal de software, técnicas de testes de sistemas e assim sucessivamente.

Portanto, segundo Fernandes (1995), a medição do software tem a importância de fornecer aos responsáveis pelo seu desenvolvimento, as informações que permitam ao gerente planejar o seu projeto de forma adequada controlando todo o trabalho com maior exatidão e tornando seu conceito em relação ao sistema mais seguro e confiável do ponto de vista do cliente. A realidade aponta para a necessidade de medições, visto que:

a) as estimativas de prazos, recursos, esforço e custo são realizadas com base no julgamento pessoal do gerente de 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 não é mensurada;
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 (clientes).

2.3 OBJETIVOS DA UTILIZAÇÃO DE MÉTRICAS

Segundo Fuck (1995), a utilidade das métricas deve ser traçada desde o início da implantação de métricas para avaliação de software. Há várias características importantes associadas com o emprego das métricas de software. Sua escolha requer alguns pré-requisitos, visto em Fernandes (1995):

a) os objetivos que se pretende atingir com a utilização das métricas;
b) as métricas devem ser simples de entender e de serem utilizadas para verificar atendimento de objetivos e para subsidiar processos de tomadas de decisão;
c) as métricas devem ser objetivas, visando reduzir ou minimizar a influência do julgamento pessoal na coleta, cálculo e análise dos resultados;
d) as métricas devem ser efetivas no custo. O valor da informação que é obtido como resultado das medições deve exceder o custo de coletar, armazenar e calcular as métricas;
e) as métricas selecionadas devem propiciar informação que possibilite avaliar acertos (ou não) de decisões e ações realizadas no passado, evidenciar a ocorrência de eventos presentes que subsidiem decisões tempestivas, bem como prever a possibilidade de ocorrência de eventos futuros.

Conforme Ambler (1998), as métricas podem ser utilizadas para:

a) estimar projetos: baseado em experiências anteriores pode-se utilizar métricas para estimar o tempo, o esforço e o custo de um projeto;
b) melhorar os esforços de desenvolvimento: as métricas podem ser utilizadas para identificar os pontos fracos do projeto, proporcionando a oportunidade de resolver um problema antes que fique fora do alcance;
c) selecionar as ferramentas;
d) melhorar a abordagem de desenvolvimento.

De acordo com Fernandes (1995), em uma organização que se dedica ao desenvolvimento de software, seja como atividade-fim seja como de suporte para uma empresa, há vários objetivos que se busca atingir, dependendo do estágio de maturidade em que se encontram essas atividades. Alguns dos objetivos perseguidos geralmente se enquadram 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, tais como novas ferramentas;
k) determinar tendências relativas a certos atributos do processo.

Para Fernandes (1995), um dos aspectos que deve ser observado quando da implementação de iniciativas de utilização de métricas é quanto a sua utilidade no contexto de um projeto ou do ambiente como um todo, além dos tipos e categorias de métricas, usuários das métricas, pessoas para as quais os resultados das métricas são destinados e os seus níveis de aplicação.

Segundo Fuck (1995), o processo de medição e avaliação requer um mecanismo para determinar quais os dados que devem ser coletados e como os dados coletados devem ser interpretados. O processo requer um mecanismo organizado para a determinação do objetivo da medição. A definição de tal objetivo abre caminho para algumas perguntas que definem um conjunto específico de dados a serem coletados. Os objetivos da medição e da avaliação são conseqüências das necessidades da empresa, que podem ser a necessidade de avaliar determinada tecnologia, a necessidade de entender melhor a utilização dos recursos para melhorar a estimativa de custos, a necessidade de avaliar a qualidade do produto para poder

determinar sua implementação, ou a necessidade de avaliar as vantagens e desvantagens de um projeto de pesquisa.

Portanto, segundo Fernandes (1995), o objetivo primário de se realizar medições no tocante ao desenvolvimento de software é obter níveis cada vez maiores de qualidade, considerando o projeto, o processo e o produto, visando à satisfação plena dos clientes ou usuários a um custo economicamente compatível.

2.4 MÉTRICAS TRADICIONAIS

Nesta seção serão apresentados alguns exemplos de métricas tradicionais. O assunto métricas de software é muito extenso para ser abordado adequadamente em um único trabalho, para maiores informações sobre métricas tradicionais, consulte Fuck (1995).

2.4.1 ANÁLISE POR PONTOS DE FUNÇÃO (FPA)

Segundo Fernandes (1995), esta técnica foi proposta por Albrecht e Gaffney e depois por Carpers Jones. 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.

Segundo Ambler (1998), embora os pontos de função sejam uma abordagem útil para a comparação de aplicações, até mesmo para aplicações que são desenvolvidas utilizando diferentes técnicas e ferramentas, podem ser caros e difíceis de colecionar. A contagem de

pontos é uma habilidade que poucas pessoas possuem, e é um processo que consome tempo até mesmo para pessoas que têm experiência nisto.

2.4.2 COCOMO (CONSTRUCTIVE COST MODEL)

Segundo Fuck (1995), o modelo COCOMO (*Constructive Cost Model* – Modelo Construtivo de Custos) é calculado a partir do número de linhas de código fonte entregues ao usuário. Este modelo foi desenvolvido por Barry Boehm e resulta estimativas de esforço, prazo, custo e tamanho da equipe para um projeto de software. O COCOMO é um conjunto de submodelos hierárquicos os quais podem ser divididos em submodelos básicos, intermediários ou detalhados.

2.4.3 LINHAS DE CÓDIGO (LOC - LINES OF CODE)

Segundo Arigoflu (1993), o sistema LOC (Linhas de Código), é a técnica de estimativa de mais antiga. Ela pode ser aplicada para estimar o custo do software ou para 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, no entanto outros não. 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. Outro problema da técnica LOC, segundo Pressman (1995), é que esta técnica é fortemente ligada à linguagem de programação utilizada, impossibilitando a utilização de dados históricos para projetos que não utilizam a mesma linguagem.

As vantagens do sistema LOC são (Fuck, 1995 e Possamai, 2000):

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 execução de uma tarefa. Assim sendo, um programa que utilize componentes está mais bem projetado, mas a medição deste tipo de programa através desta métrica não é eficiente;
c) difíceis de estima no inicio 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.

Com estas colocações, nota-se que a métrica LOC não é uma métrica a ser utilizada por si só, ela 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.

2.4.4 MÉTRICA DE CIÊNCIA DO SOFTWARE

Segundo Shepperd (1993), Halstead identificou a Ciência do Software - originalmente chamada de Física do Software - como uma das primeiras manifestações sobre 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 associados a operandos.

Segundo Shepperd (1993), a ciência do Software atraiu consideravelmente o interesse das pessoas em meados de 1970 por ser uma novidade na metrificação do software. Além disso, as entradas básicas do software são todas facilmente extraídas. Após o entusiasmo inicial da Ciência do Software, foram encontrados sérios problemas. Os motivos podem ser relatados em função da dificuldade que os pesquisadores encontraram na comparação dos

trabalhos e evolução da métrica. Outro motivo seria a não associação correta entre o esforço requerido para manipulação do programa e o tempo exigido para conceber o programa e também por tratar um sistema como um simples módulo.

2.4.5 MÉTRICA DA COMPLEXIDADE CICLOMÁTICA

Segundo Shepperd (1993), este método foi proposto por McCabe, que 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 softwares em módulos.

Segundo Jacobson (1992), a idéia é desenhar num grafo a seqüência que um programa pode tomar com todos os possíveis caminhos. A complexidade calculada fornecerá um número designando o quão complexo é um programa (ou seqüência). Segundo Shepperd (1993), 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:

$$\mathbf{v(G) = E - n + 2},$$

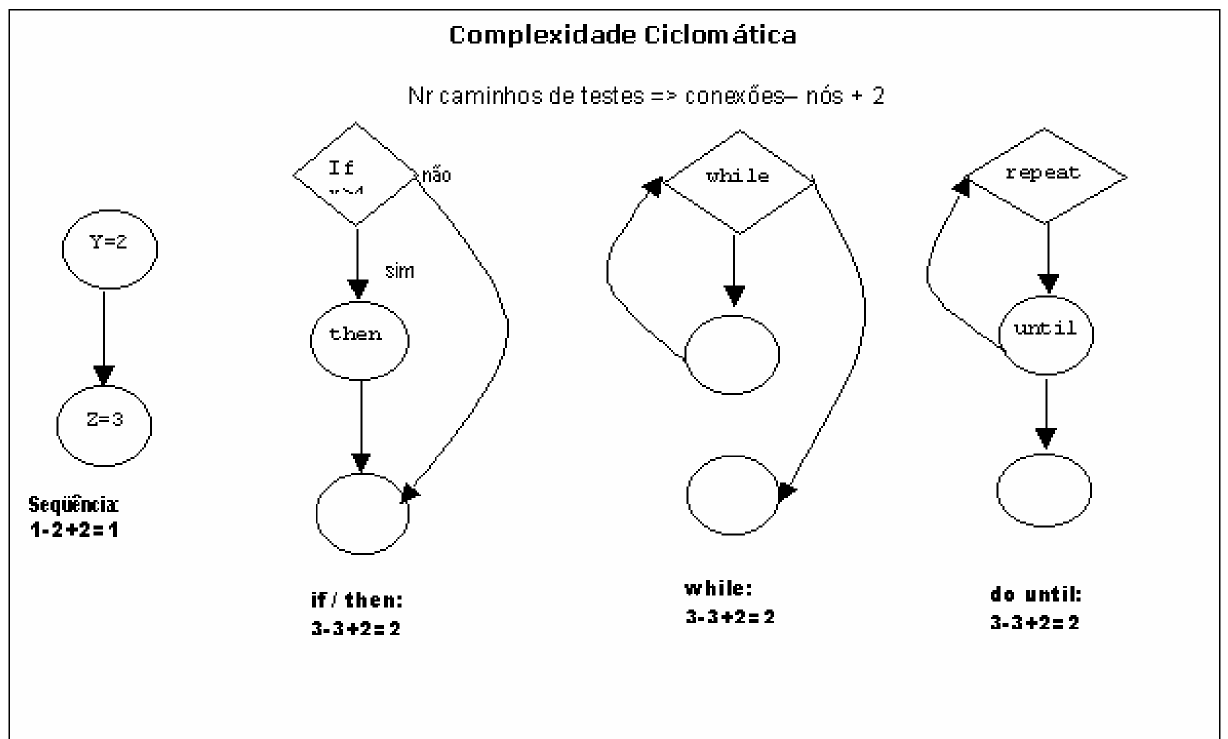
onde, **E:** é o número de arestas e **N**: é o número de nós.

A visão simplista da métrica de McCabe pode ser questionada em vários pontos. Primeiro, ele tinha uma preocupação especial 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. Conseqüentemente, a métrica não é sensível à complexidade, contribuindo assim na formação de declarações de seqüências lineares.

Ainda segundo Shepperd (1993), a complexidade ciclomática 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 de um 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.

Figura 1 – EXEMPLO DE CÁLCULO DE COMPLEXIDADE CICLOMÁTICA



3 MÉTRICAS PARA ORIENTAÇÃO A OBJETOS (OO)

Segundo Rocha (2001), embora exista farta literatura sobre como aplicar os métodos OO, nenhum deles trata em detalhe os aspectos relativos à qualidade. A razão é simples: o desenvolvimento de software utilizando esse enfoque ainda não dispõe de métricas precisas e bem entendidas que possam ser utilizadas para avaliar produtos e processos de desenvolvimento OO. Valores para medidas de atributos e formas de prevenção e correção de defeitos ainda não estão estabelecidos.

O desenvolvimento de software utilizando o paradigma de orientação a objetos (OO) surge como uma possibilidade para a melhoria da qualidade e produtividade, pois o enfoque OO permite modelar o problema em termos de objetos capazes de diminuir a distância entre o problema do mundo real e sua abstração (Rocha, 2001).

A orientação a objetos requer uma abordagem diferente tanto no projeto e implementação, como também nas métricas de software, uma vez que a tecnologia OO usa objetos e não algoritmos como blocos de construção fundamentais (Rosenberg, 1998). Para Rocha (2001), dadas as diferenças entre as duas visões, OO e funcional, é comum constatar que as métricas de software desenvolvidas para serem aplicadas aos métodos tradicionais de desenvolvimento não são facilmente mapeadas para os conceitos OO.

Segundo Rosenberg (1998), as métricas OO devem avaliar os seguintes atributos:

a) eficiência: verificar se as construções estão projetadas eficientemente;
b) complexidade: verificar se as construções estão sendo utilizadas visando reduzir a complexidade;
c) entendimento;
d) reusabilidade;
e) testabilidade/manutenção: verificar se a estrutura é fácil de ser testada e/ou modificada.

As métricas para software OO são diferentes devido a: localização, encapsulamento, informação oculta, herança e técnicas de abstração dos objetos.

Segundo Jacobson (1992), as métricas OO podem ser divididas em duas categorias: métricas relacionadas com processos e métricas relacionadas com produtos.

As métricas relacionadas com processo são utilizadas para medir o progresso e o status do processo de desenvolvimento do software, consistem em medir coisas tais como: homem/meses, cronogramas ou número de falhas encontradas durante os testes. Segundo Jacobson (1992), para aprender a manipular e administrar um processo de desenvolvimento Orientado a Objetos é importante iniciar a coleta de dados destas medições tão metodicamente quanto possível. Abaixo alguns exemplos de métricas relacionadas com processo que podem ser coletadas quando se utiliza a Orientação a Objetos:

a) tempo total de desenvolvimento;
b) tempo de desenvolvimento em cada processo e sub-processo;
c) tempo gasto modificando modelos de processos anteriores;
d) tempo gasto em todos os tipos de sub-processos como: especificação dos casos de uso, desenho dos casos de uso, desenho do bloco, teste do bloco e do caso de uso para cada objeto;
e) número de diferentes tipos de falhas encontrados durante revisões;
f) número de mudanças propostas nos modelos anteriores;
g) custo da garantia de qualidade;
h) custo para introduzir novas ferramentas e processo de desenvolvimento.

Estas medições podem formar uma base para o planejamento do desenvolvimento de projetos futuros. Por exemplo, conhecendo o tempo médio gasto para especificar um caso de uso, pode-se prever o tempo necessário para especificar todos os casos de uso. Estas medições, entretanto deveriam sempre vir acompanhadas por uma indicação de exatidão da medição (tal como desvio padrão), caso contrário, não se tem senso de exatidão da previsão. Deve-se observar também que estas medições podem variar muito entre diferentes processos, organizações, aplicações e equipes. Portanto, é perigoso tirar conclusões genéricas sobre dados existentes sem considerar as circunstâncias (Jacobson, 1992).

As métricas relacionadas com produtos são aquelas que são utilizadas para controlar a qualidade do produto final (o software). Elas tradicionalmente são aplicadas ao software ainda em construção para medir sua complexidade e prever propriedades do produto final.

Conforme Jacobson (1992), métricas tradicionais de produto podem ser utilizadas para algumas aplicações OO. Entretanto a métrica mais comum, linhas de código, é a menos interessante para softwares OO, pois, às vezes o menor código escrito é o mais reutilizado e, muitas vezes dá maior qualidade ao produto. Para obter uma noção para o código os exemplos de métricas abaixo são os mais apropriados para softwares orientados a objetos:

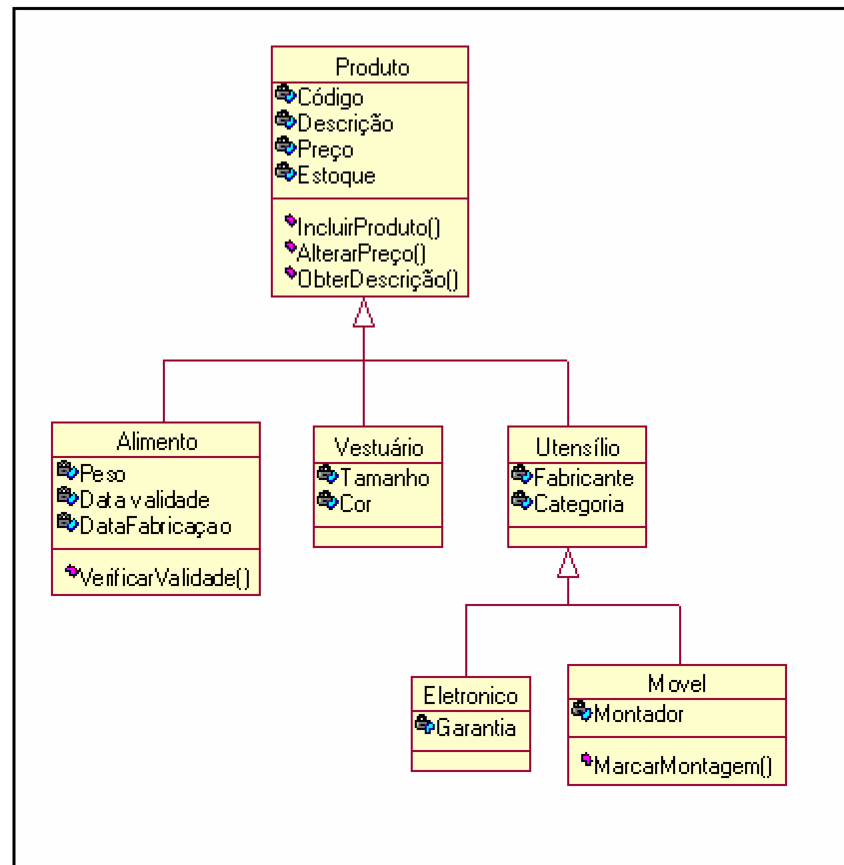
a) quantidade total de classes;
b) quantidade total de operações;
c) quantidade de classes reutilizadas e número de classes desenvolvidas;
d) quantidade de operações reutilizadas e número de operações desenvolvidas;
e) quantidade de mensagens enviadas.

Segundo Rocha (2001), muitas métricas desenvolvidas para serem utilizadas no paradigma tradicional são inadequadas para avaliar sistemas OO, embora algumas possam ser utilizadas para avaliar a qualidade de métodos nas classes. Por isto novas métricas têm sido propostas para atender a orientação a objetos.

A seguir serão explicadas algumas métricas para Orientação a Objetos. Estas métricas estão divididas em três categorias: métricas de análise, métricas de projeto e métricas de construção. Segundo Ambler (1998), estas métricas podem ser utilizadas para ajudar a melhorar os esforços de desenvolvimento. Elas podem identificar áreas com problemas na aplicação antes que elas apareçam como um erro detectado pelo usuário. As métricas de projeto e construção, além de medir aspectos importantes do software, são fáceis de automatizar, tornando-as mais fáceis de coletar (Ambler, 1998).

A figura 2 apresenta um exemplo de diagrama de classes que servirá para ilustrar algumas das métricas explicadas a seguir. Este diagrama de classes define criação de produtos: um produto pode ser do tipo alimento, vestuário ou utensílio; um alimento precisa ter o controle de peso e validade; já um vestuário possui o controle de tamanho e cor; um utensílio pode ser to tipo eletrônico ou móvel.

Figura 2 – EXEMPLO DE DIAGRAMA DE CLASSES



3.1 MÉTRICAS DE ANÁLISE

Segundo Ambler (1998), as métricas de análise são utilizadas para medir a qualidade dos esforços de análise.

3.1.1 PORCENTAGEM DE CLASSES-CHAVE

Segundo Lorenz (1994), classes-chave são as classes que são de importância central no negócio. Conforme Ambler (1998), embora o número de classes-chave seja utilizado para estimar o tamanho de um projeto, também pode indicar se a análise terminou ou não. Entre todas as classes identificadas através do CRC (colaborador de responsabilidade de classe) ou da diagramação de classe, a porcentagem de classes-chave deve ficar entre 30% e 50%. Se a porcentagem for baixa é sinal que ainda há muita análise a fazer.

3.1.2 NÚMEROS DE CENÁRIOS DE UTILIZAÇÃO

Segundo Lorenz (1994), o número de cenários de utilização é um indicador do tamanho da aplicação a ser desenvolvida.

Para Ambler (1998), o número de cenários de utilização identificados também pode ser usado para verificar se análise foi completada ou não. Pequenas aplicações (poucos meses de desenvolvimento) possuem entre cinco e dez cenários de utilização, aplicações de tamanho médio (menos de um ano de desenvolvimento) têm entre 20 e 30, e projetos grandes têm, tipicamente, 40 ou mais.

3.2 MÉTRICAS DE PROJETO

Conforme Andrade (1997), do vasto conjunto de métricas existentes há várias que podem ser avaliadas em nível de projeto, possibilitando seu uso para uma melhor orientação e acompanhamento do desenvolvimento durante sua realização. Segundo Lorenz (1994), as métricas de projeto permitem avaliar as características dos componentes do software.

Segundo Cardoso (1999), essas métricas destinam-se a ajudar a estabelecer comparações entre vários sistemas e a criar as bases de futuras recomendações para um novo projeto que poderão eventualmente evoluir para normas em nível organizacional.

3.2.1 CONTAGEM DE MÉTODOS

Para Ambler (1998), a contagem de métodos é uma métrica muito fácil de se coletar. As classes com grande quantidade de métodos provavelmente são aplicações específicas ao passo que classes com menos métodos possuem uma tendência a ser mais reutilizáveis. Embora isto não seja sempre verdade, é válido ter uma contagem de métodos para todas as classes da aplicação, e então verificar as que possuem uma contagem muito alta comparadas às outras. Considerando estas classes, questiona-se se podem ou não ser aplicadas em outros projetos, se sim, então se deve pensar numa maneira de torná-las mais reutilizáveis possível.

Segundo Rosenberg (1998), o número de métodos de uma classe não deveria passar de 20, porem é aceitável que este número chegue a 40.

Referindo-se a figura 2, a quantidade de métodos da classe *Produto* é igual a dois, para a classe item a quantidade de métodos é igual a três. Por ser ilustrativo, as classes do exemplo da figura 2 não possuem todos os métodos que seriam necessários para sua implementação.

3.2.2 MÉTODOS PONDERADOS POR CLASSE (WMC - WEIGTHED METHODS PER CLASS)

De acordo com Rosenberg (1998), esta métrica é a medida da complexidade individual de uma classe. É uma contagem dos métodos implementados numa classe e a soma de suas complexidades (a complexidade do método é medida pela complexidade ciclomática).O número de métodos e sua complexidade são indicadores de quanto tempo e esforço serão necessários para desenvolver e manter a classe.

Conforme Chidamber (1994), quanto maior o número de métodos numa classe, maior será o potencial de impacto nos filhos, uma vez que os filhos herdam os métodos definidos na classe pai. Classes que possuem métodos maiores são geralmente feitas para aplicações mais especificas limitando a possibilidade de reutilização.

3.2.3 RESPOSTA DE UMA CLASSE (RFC – RESPONSE FOR A CLASS)

Segundo Chidamber (1994), a RFC é definida pelo número de diferentes métodos que podem ser chamados em resposta a uma mensagem para um objeto da classe ou por algum método na classe, incluindo todos os métodos acessados dentro da hierarquia da classe. Esta métrica contempla a combinação da complexidade da classe através do número de métodos e a comunicação total com outras classes.

Se um grande número de métodos pode ser chamado em resposta a uma mensagem, o teste e a depuração da classe tornam-se mais complicados, uma vez que ela requer um maior nível de entendimento. Além disso, quanto maior o número de métodos que podem ser chamados pela classe, maior é a sua complexidade (Rosenberg, 1998).

Para Rosenberg (1998), um número aceitável para esta métrica seria 100, entretanto, para a maioria das classes este número é menor ou igual a 50.

Na figura 2, a RFC para classe *Móve*l, é o número de métodos que podem ser chamados em resposta a mensagens dela própria e de suas superclasses (*Utensílio* e *Produto*). Portanto a RFC da classe *Móvel* é igual a cinco, sendo a soma dos seus métodos e dos métodos de seus pais.

3.2.4 PROFUNDIDADE DA ÁRVORE DE HERANÇA (DTI - DEPTH OF INHERITANCE TREE)

A profundidade da hierarquia da herança é o número máximo de passos da classe nó até a raiz da árvore e é medida pelo número de classes ancestrais (Rosenberg, 1998).

Segundo Ambler (1998), esta métrica indica as dificuldades na maneira de utilizar o conceito de herança. Árvores muito profundas constituem projetos de maior complexidade, uma vez que um número maior de métodos e classes está envolvido, pois quanto mais profunda uma classe na hierarquia, maior o número de métodos que ela provavelmente herda, tornando-a mais difícil de entender e, portanto de manter e incrementar.

Segundo Rosenberg (1998), DTI igual a zero indica uma classe raiz, um alto percentual de DTI entre dois e três indica um alto grau de reutilização, entretanto se a maioria dos ramos da árvore forem pouco profundos (DTI < 2) isto pode representar uma pobre exploração das vantagens do desenvolvimento OO e da herança. Por outro lado uma árvore muito profunda (DTI > 5) pode ser perigosa, pois aumenta a complexidade do projeto.Para Ambler (1998), se a profundidade de uma herança for maior do que cinco, é preciso reavaliar o projeto.

Na figura 2, a classe Produto é a raiz da árvore e tem uma DTI igual a zero, as classes Móvel e Eletrônico têm DTI igual a três.

3.2.5 NÚMERO DE FILHOS (NOC – NUMBER OF CHILDREN)

Segundo Rosenberg (1998), o número de filhos consiste do número de subclasses imediatamente subordinadas a uma classe na hierarquia. Esta medida está relacionada com a profundidade da árvore de herança, porém a contraria. Embora se queira evitar a complexidade originada da profundidade da árvore de herança, geralmente é melhor se ter

profundidade do que comprimento, pois promove a reutilização ao longo da herança (Ambler, 1998).

O número de filhos é um indicador do potencial de influência que uma classe tem no projeto e no sistema. Um alto número de filhos promove maior reutilização, visto que a herança é uma forma de reutilização. Porém quanto maior o número de filhos, maior a probabilidade de abstração imprópria do pai e poder ser um caso de mau uso da subclasse (Chidamber, 1994).

Segundo Ambler (1998), um modo do número de filhos ser utilizado para avaliar a qualidade do uso da herança é observar onde na hierarquia de classes as suas classes estão localizadas. As classes mais altas geralmente possuem mais subclasses do que as localizadas mais abaixo, pois possuem menos códigos específicos de aplicações e, portanto, são mais reutilizáveis.

Pode-se observar na figura 2 que o número de filhos da classe *Utensílio* é igual a dois, já para a classe *Alimento* o NOC é igual a zero, pois está classe um nó terminal na estrutura da árvore.

3.2.6 FALTA DE COESÃO (LCOM - LACK OF COHESION)

De acordo com Rosenberg (1998), a falta de coesão é definida pelo número de diferentes métodos dentro de uma classe que referenciam uma determinada variável de instância. A falta de coesão mede as diferenças de métodos em uma classe pelos atributos ou variáveis de instância.

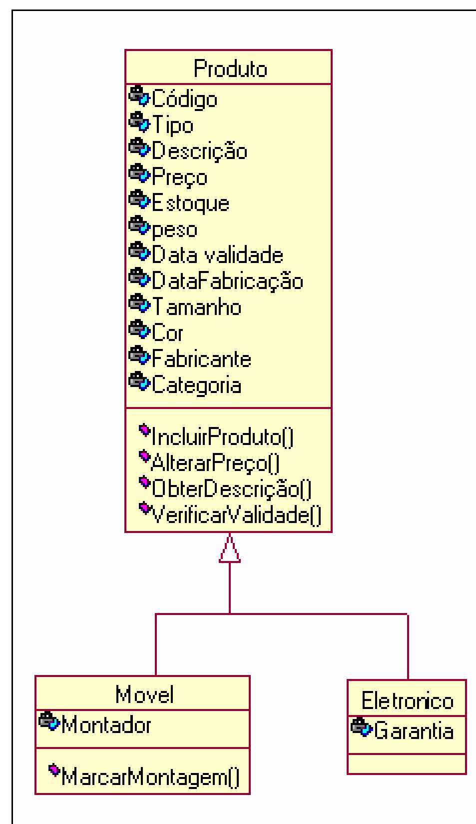
Para Chidamber (1994), um módulo altamente coeso deveria manter-se sozinho. Alta coesão indica uma boa subdivisão das classes e implica em simplicidade e alta reutilização. A falta de coesão ou coesão baixa aumenta a complexidade, aumentado com isso a possibilidade de erros durante o processo de desenvolvimento. Classes com coesão baixa poderiam, provavelmente ser divididas em duas ou mais subclasses com coesão maior.

Segundo Rosenberg (1998), existem pelo menos duas maneiras de medir a coesão:

a) calcular para cada atributo de uma classe o percentual de métodos que o utilizam, obter a média destes percentuais e subtraí-la de 100%. Percentuais baixos significam uma maior coesão entre atributos e métodos na classe;
b) outra maneira é contar o número de conjuntos disjuntos produzidos pela intersecção dos conjuntos de atributos de diferentes classes.

Na figura 3, as classes Alimento, Utensílio e Vestuário filhas de Produto na figura 2 foram eliminadas e os atributos e métodos das mesmas foram inseridos na classe Produto. Este exemplo mostra um projeto com uma falta de coesão, pois muitos métodos e atributos da classe Produto não farão sentido e nem serão utilizados pelos objetos que serão criados a partir desta classe. Isto implica que novas abstrações serão necessárias, pois objetos similares precisariam ser agrupados pela criação de classes filhas deles. Portanto, o projeto da figura 2 é melhor neste aspecto do que o da figura 3.

Figura 3 – EXEMPLO DE FALTA DE COESÃO



3.2.7 ACOPLAMENTO ENTRE OBJETOS (CBO – COUPLING BETWEEN OBJECT CLASSES)

Segundo Rosenberg (1998), o acoplamento entre objetos é definido pelo número de outras classes acopladas a uma classe. Duas classes são acopladas quando métodos declarados em uma classe utilizam métodos ou variáveis de instância em outra classe. O acoplamento é medido pelo número de classes distintas não herdadas da hierarquia de classes das quais a classe depende.

Segundo Rosenberg (1998), as classes (objetos) podem ser acopladas de três maneiras:

a) quando uma mensagem é passada entre os objetos;
b) quando os métodos declarados em uma classe utilizam atributos ou métodos de outras classes;
c) através da herança que introduz um significativo acoplamento entre as superclasses e suas subclasses.

Conforme Chidamber (1994), o acoplamento excessivo entre objetos é danoso para projetos modulares e impede a reutilização. Quanto mais independente for uma classe mais facilmente ela será reutilizada em outra aplicação. Para melhorar a modularidade e o encapsulamento o acoplamento entre classes deveria ser o mínimo possível, pois um alto nível de acoplamento torna a manutenção mais difícil, pois a sensibilidade a mudanças em outras partes do projeto é aumentada.

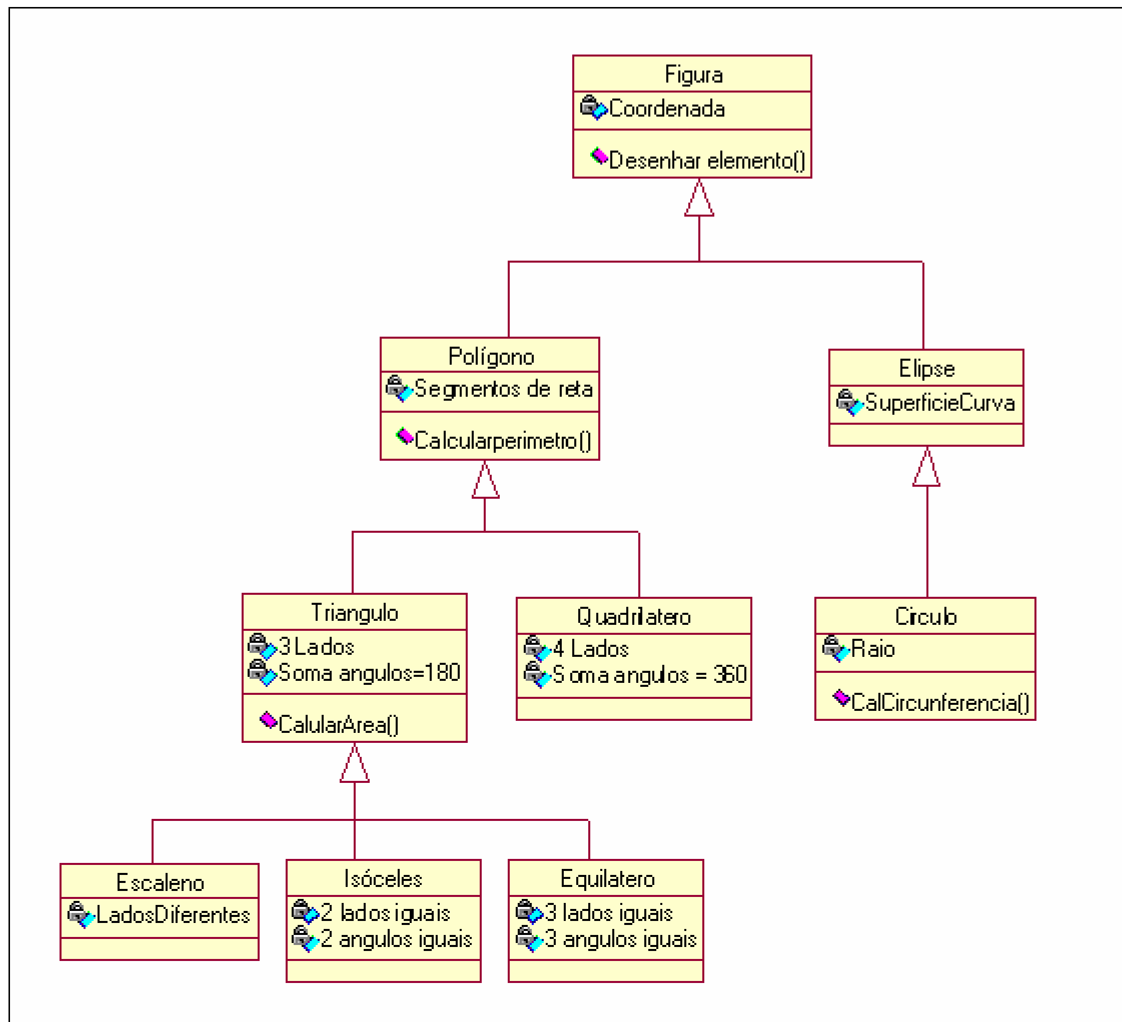
A medição do acoplamento é útil para determinar o quão complexos serão os testes das várias partes do sistema. Um maior acoplamento entre classes requer testes mais rigorosos (Rosenberg, 1998).

Segundo Rosenberg (1998), o valor desta métrica deveria ser menor ou igual a cinco, uma vez que um número CBO muito alto indica que as classes podem ser difíceis de entender, manter e/ou reutilizar.

Na figura 4, para a classe Polígono todo acoplamento é dentro da classe, pois o método CalcularPerimetro apenas acessaria o valor de cada segmento de reta, então o CBO é igual a zero. Ao observar na classe Figura o método DesenharElemento ele se relaciona a algo

externo à classe (sistema de desenho da interface), o que indica um alto acoplamento desta classe com uma classe externa.

Figura 4 – CLASSES GEOMÉTRICAS COM ATRIBUTOS E MÉTODOS



Fonte: Rosenberg (1998)

3.2.8 UTILIZAÇÃO GLOBAL

Conforme Lorenz (1994), a métrica da utilização global é uma contagem de quantas variáveis globais estão sendo utilizadas na aplicação. Segundo Ambler (1998), o ideal é reduzir o uso de variáveis globais, pois estas aumentam o acoplamento da aplicação, tornando-a mais difícil de manter e de aumentar.

3.3 MÉTRICAS DE CONSTRUÇÃO

Segundo Ambler (1998), além de medir a qualidade do projeto as métricas podem ser aplicadas para melhorar a qualidade do código. As métricas a seguir podem ser utilizadas como base para este fim. Porém um dos problemas com as métricas de construção é que os codificadores com uma mentalidade *hacker* não gostam de outras pessoas avaliando o seu trabalho.

3.3.1 TAMANHO DO MÉTODO

Segundo Rosenberg (1998), o tamanho de um método pode ser medido de várias maneiras. Estas incluem contagem de todas as linhas físicas de código, o número de comandos, o número de linhas em branco e o número de linhas comentadas, entre outras. As métricas abaixo lidam com a quantificação de um método individual.

3.3.1.1 QUANTIDADE DE MENSAGENS ENVIADAS

Segundo Lorenz (1994), esta métrica conta o número de mensagens enviadas por um método. Um método de um objeto é chamado através de uma mensagem, esta pode chamar um método para executar cálculos, acessar e alterar as variáveis de instância ou enviar mensagens para outros objetos como parte de suas respostas.

Esta métrica quantifica o tamanho do método de um modo relativamente imparcial. Outras métricas, tais como linhas de código (LOC), não levam em consideração fatores como estilo de código quando medem o tamanho de um método.

A linguagem utilizada influencia nesta métrica, pois quando se codifica numa linguagem híbrida, como C++, pode-se escrever métodos que não são orientados a objeto. Este código não seria contado no número de mensagens enviadas, mas relaciona-se com o tamanho do método. Por exemplo, não se pode ignorar 100 linhas de código não orientado a objetos e contar somente um par de mensagens enviadas.

A figura 5 mostra dois pedaços de código Smalltalk que fazem exatamente a mesma coisa, porém com estilos de codificação diferentes. Se estes dois pedaços de código forem

comparados baseando-se na contagem de linhas físicas o primeiro caso teria um valor maior. Contando as mensagens a comparação entre os dois é mais exata.

Figura 5 – EXEMPLO DE CONTAGEM DE MENSAGENS ENVIADAS

Código	Msg enviadas	LOC
`(invoice lineItens) do: [each| totalSale := totalSale + each price. ].`	4	4
`Invoice lineItens do : [:each| Total := total + each price.].`	4	2

```
(invoice lineItens) do: [each|
   totalSale :=
         totalSale + each price.
].
```

```
Invoice lineItens do : [:each|
    Total := total + each price.].
```

Fonte: Lorenz (1994)

Segundo Lorenz (1994), se a quantidade de mensagens enviadas pelo método ultrapassar nove pode ser um indicativo de que o código está orientado a função e/ou pobre na alocação de responsabilidades.

3.3.1.2 LINHAS DE CÓDIGO (LOC)

Conforme Lorenz (1994), a métrica linhas de código conta o número de linhas físicas de código ativo que estão em um método. Ela quantifica o tamanho do método sem levar em consideração fatores como estilo de codificação. De acordo com Ambler (1998), para se criar métodos que sejam fáceis de manter, eles devem ser pequenos. Além disso, quando o método é grande existe um bom indicador de que seu código é na realidade, orientado a função, ao contrário de orientado a objeto. Os objetos conseguem as coisas através da colaboração de outros objetos e não fazendo tudo sozinhos. Isto resulta em métodos curtos e não em métodos longos.

Segundo Lorenz (1994), a forma que o programador escolhe para identar linhas, ir para uma linha nova ou utilização de nomes de variáveis muito grandes irão afetar esta medida. Pois a escolha do estilo (para redigibilidade) quebra uma mensagem em diferentes linhas físicas o que induz aumento na contagem de linhas de código.

Para Ambler (1998), deve-se esperar menos de 10 linhas para código Smalltalk e 30 linhas para C++. Os métodos devem ser pequenos, porém bem comentados, e devem fazer apenas uma única coisa.

3.3.1.3 MÉDIA DO TAMANHO DOS MÉTODOS

Segundo Lorenz (1994), através do projeto os métodos deveriam ser pequenos em média. Pela observação do número de métodos, pode-se obter um sentimento da qualidade do projeto como um todo. Se o tamanho médio dos métodos for muito alto indica uma grande probabilidade que código orientado a função está sendo escrito. Ainda segundo Lorenz (1994), para Smalltalk a média do tamanho dos métodos deveria ser inferior a seis linhas de código e por volta de 18 linhas de código pra C++.

O tamanho médio dos métodos pode ser obtido pela média de linhas de código por métodos (total de linhas de código dividido pelo total de métodos) ou pela média de mensagens enviadas (total de mensagens enviadas dividido pelo total de métodos).

3.3.2 PERCENTUAL COMENTADO

Segundo Ambler (1998), uma métrica muito útil para estimar a qualidade do código é o percentual de linhas comentadas por método. Para Ambler (1998), ter poucas linhas de comentários indica que outro programador terá dificuldades em compreender o código, porém comentários demais podem indicar que muito tempo está sendo perdido para documentar os métodos.

O percentual de linhas comentadas é obtido pela divisão do total de linhas comentadas pelo total de linhas de código. Segundo Rosenberg (1998), este percentual deveria ficar entre 20 e 30%, pois os comentários auxiliam na manutenção e reusabilidade do código.

3.3.3 COMPLEXIDADE DO MÉTODO

Segundo Lorenz (1994), existem muitos trabalhos feitos na área de complexidade de código, a maioria destes trabalhos enfoca fatores tais como números de pontos de decisões feitas no código de uma função, representada por construções *if-then-else* e outras. Métricas tradicionais de complexidade não levam em consideração as diferenças dos projetos OO.

Existem diferenças básicas no código OO que podem fazer estas medidas menos proveitosas: métodos orientados a objetos são curtos; sistemas orientados a objetos bem desenhados não utilizam declarações *case*. Conforme Lorenz (1994) bons projetos OO farão as métricas históricas de complexidade menos utilizáveis, tornando necessário utilizar outras métricas que tenham sentido para código OO, como suplemento para as métricas tradicionais. Lorenz (1994) utilizou os seguintes pesos para computar a complexidade do método:

a) chamadas a API: 5.0;
b) atribuições: 0,5;
c) expressões binárias (Smalltalk) ou operadores matemáticos (C++): 2,0;
d) mensagens enviadas (Smalltalk) ou mensagens com parâmetros (C++): 3,0;
e) expressões aninhadas: 0,5;
f) parâmetros: 0,3;
g) chamadas primitivas: 7,0;
h) variáveis temporárias: 0,5;
i) mensagens sem parâmetros: 1,0.

Segundo Lorenz (1994), o valor esperado para esta métrica é 65. Este valor originou-se da avaliação de um número significativo de projetos ao longo dos anos e julgamento baseado na experiência do autor.

A média da complexidade por método é obtida pelo número de complexidades dividido pelo número de métodos.

3.3.4 TAMANHO DA CLASSE

O tamanho da classe pode ser medido de várias maneiras, as métricas explicadas a seguir podem ser utilizadas para quantificar o tamanho de uma classe.

3.3.4.1 QUANTIDADE DE MÉTODOS DE INSTÂNCIA PÚBLICOS EM UMA CLASSE

Segundo Lorenz (1994), esta é uma boa medida da responsabilidade total da classe. Os métodos públicos são aqueles serviços que estão disponíveis como serviços para outras

classes. Métodos públicos são indicadores do trabalho total feito por uma classe, uma vez que eles são serviços utilizados por outras classes.

3.3.4.2 QUANTIDADE DE MÉTODOS DE INSTÂNCIA EM UMA CLASSE

Segundo Lorenz (1994), esta métrica conta todos os métodos de instância de uma classe. Este número relaciona-se com o total de colaboração utilizado. Classes maiores podem estar tentando fazer muito trabalho sozinhas, além disso, elas são mais complexas e difíceis de manter. Classes menores tendem a ser mais reutilizáveis, uma vez que elas proporcionam um conjunto coeso de serviços em vez de um conjunto misturado de aptidões. Muitos métodos numa classe (não incluindo os herdados) são um aviso que um tipo de objeto está com responsabilidade demais. Bons projetos OO deveriam distribuir inteligência e trabalho pesado cooperando entre vários objetos. Esta métrica deveria focar-se em métodos públicos, uma vez que estes indicam o trabalho total que uma classe está designada a fazer. Os métodos privados são simplesmente o modo como a classe vai fazer o seu trabalho e alimentar os métodos públicos.

Segundo Lorenz (1994), o valor recomendado para esta métrica é de 40 para classes de interface com o usuário e 20 para as demais.

Através desta métrica pode-se obter outra que é a média de métodos de instância por classe. Esta média é obtida pelo total de métodos instanciados dividido pelo total de métodos.

3.3.4.3 QUANTIDADE DE ATRIBUTOS POR CLASSE

Conforme Lorenz (1994), ao contar a quantidade de atributos das classes, adquire-se um indicador da qualidade do projeto. O fato de uma classe ter muitos atributos indica que a classe tem muitos relacionamentos com outros objetos do sistema. Estes podem ser objetos simples como um número inteiro, ou objetos complexos como um endereço ou um cliente.

Segundo Ambler (1998), as classes com mais de três ou quatro atributos estão com freqüência mascarando um problema de acoplamento na aplicação. Conforme Lorenz (1994), para classes de interface com o usuário este número pode chegar a nove, pois estas classes necessitam de mais atributos para lidar com componentes de telas.

3.3.4.4 MÉDIA DE ATRIBUTOS POR CLASSE

Segundo Lorenz (1994), ao longo do projeto a média de atributos é um indicador do tamanho da classe. Ela é obtida pelo total de atributos dividido pelo total de métodos. Os atributos afetam as requisições de armazenamento do sistema em tempo de execução. Muitos atributos, em geral, indicam a possibilidade das classes estarem fazendo mais do que deveriam. Devido a este fato, as classes podem ter relacionamentos demais com outros objetos.

3.3.4.5 QUANTIDADE DE MÉTODOS DE CLASSE EM UMA CLASSE

Segundo Lorenz (1994), as próprias classes são objetos que podem fornecer serviços que são globais para as suas instâncias. Isto faz sentido para manusear valores constantes comuns e inicializar instâncias, mas não deveria ser o principal meio de fazer o trabalho. O número de métodos disponíveis para a classe e não para suas instâncias afetam o tamanho da mesma. Este número deveria ser relativamente menor comparado com o número de métodos de instância.

A quantidade de métodos de classe pode indicar o total de métodos comuns que são manuseados por todas as instâncias. Também pode indicar projetos pobres, se os serviços estão sendo melhor manipulados por instâncias individuais, do que pela própria classe.

Lorenz (1994) utiliza dois tipos de entradas para esta métrica:

a) absoluto: as classes raramente necessitam mais do que quatro métodos de classe;
b) relativo a quantidade de métodos de instância: este valor é obtido com a combinação da entrada desta métrica com a entrada esperada da quantidade de métodos de instância (quatro versus vinte), o que resulta em 20% este valor é utilizado para identificar possíveis anormalidades.

3.3.4.6 QUANTIDADE DE VARIÁVEIS DE CLASSE EM UMA CLASSE

Conforme Lorenz (1994), variáveis de classes são globais e proporcionam objetos comuns para todas as instâncias de uma classe. As variáveis de classe são utilizadas freqüentemente para proporcionar valores constantes customizáveis, que costumam afetar

toda a funcionalidade das instâncias. Elas podem também coordenar informações através de todas as instâncias, como a determinação de um único valor para o número de uma transação.

A média das variáveis de classe é obtida pelo total de variáveis de classe dividido pelo total de classes. Algumas classes podem ter mais variáveis de classe, mas, em geral, elas deveriam ter menos variáveis de classe do que atributos. Segundo Lorenz (1994), uma classe deveria ter em torno de três variáveis de classe.

3.3.5 QUANTIDADE DE CLASSES ABSTRATAS

Uma classe abstrata existe para facilitar a reutilização de métodos e dados entre suas subclasses. Ela não possui instâncias em tempo de execução, mas representa uma generalização de conceitos incorporados por uma coleção de subclasses. Embora uma superclasse não precise ser abstrata, a experiência mostra que existe um certo número de classes abstratas em projetos bem sucedidos. A quantidade de classes abstratas é uma indicação do uso bem sucedido da herança e o do esforço que foi gasto observando conceitos gerais no domínio do problema. Projetos bem desenhados possuem, geralmente mais de 10% de classes abstratas (Lorenz, 1994).

3.3.6 USO DE HERANCA MÚLTIPLA

Segundo Lorenz (1994), algumas linguagens como C++, permitem que uma classe herde condições e funcionalidades de múltiplas superclasses. Alguns podem argumentar que isto é necessário para modelar o mundo real corretamente. Algumas complicações resultam do uso de herança múltipla, tais como: conflito de nome e compreensão do programador.

A indústria tem geralmente concordado que a herança múltipla não é necessária, uma vez que existem vários meios apropriados para modelar seu negócio e a herança múltipla pode trazer problemas. O uso de herança múltipla não é recomendado, porém em linguagens que a suportam seu uso pode ser medido e reportado para garantir uma decisão consciente de utilizá-la (Lorenz, 1994).

3.3.7 QUANTIDADE DE MÉTODOS SOBRESCRITOS POR UMA SUBCLASSE

Segundo Lorenz (1994), uma subclasse pode definir um método com o mesmo nome de sua superclasse. Isto é chamado de sobrescrever o método, pois uma mensagem agora irá causar a execução do novo método, em vez do método da superclasse. Muitos métodos sobrescritos indicam um problema de projeto, pois as subclasses deveriam ser uma especialização de suas superclasses, elas deveriam ampliar os serviços das superclasses.

A média de métodos sobrescritos por classe pode ser obtida pelo total de métodos sobrescritos dividido pelo total de classes.

Ainda segundo Lorenz (1994), o número de métodos sobrescritos por uma subclasse deveria ser em torno de três. Este número deveria diminuir nos níveis mais baixos da árvore de herança.

3.3.8 QUANTIDADE DE MÉTODOS HERDADOS POR UMA SUBCLASSE

Segundo Lorenz (1994), herança é um conceito muito importante no desenvolvimento de software orientado a objetos. Subclasses naturalmente herdam de suas superclasses funcionalidades na forma de métodos e dados na forma de atributos. Este é o esquema normal da hierarquia de classes. O número de métodos herdados de superclasses indica a eficácia da subclasse pela especialização. O percentual de métodos herdados deveria ser alto. A entrada desta métrica é o inverso da métrica quantidade de métodos sobrescritos, e um percentual baixo indica subclasses pobres.

3.3.9 QUANTIDADE DE MÉTODOS ADICIONADOS POR UMA SUBCLASSE

As subclasses deveriam definir métodos novos, ampliando a funcionalidade da superclasse, pois uma classe sem métodos é questionável. A quantidade de métodos novos deveria ir diminuindo conforme se desce os degraus da hierarquia (Lorenz, 1994).

Segundo Lorenz (1994), uma subclasse deve ter pelo menos um método novo para justificar a sua existência. A quantidade de métodos novos vai diminuindo conforme vai se descendo os degraus da hierarquia da árvore de herança. Portanto para classes que estão no nível um da hierarquia o número de métodos novos poderia ser em torno de 20, já no nível seis este número deveria ser baixo, por volta de quatro.

3.3.10 ÍNDICE DE ESPECIALIZAÇÃO

Conforme Lorenz (1994), especialização pura implica em adicionar mais funcionalidades a uma subclasse completando as funcionalidades existentes. Na prática, especialização de subclasses inclui:

a) adicionar novos métodos;
b) adicionar funções a métodos existentes, embora ainda chame o método da superclasse;
c) sobrescrever métodos com uma função totalmente nova;
d) excluir métodos através da sobrescrita dos mesmos por outros sem funções.

Pela observação de uma combinação de métricas, pode-se determinar a qualidade das subclasses. Neste caso, a qualidade é definida como alta para subclasses por especialização e baixa para subclasses por implementação.

Subclasse por especialização significa uma extensão das capacidades das superclasses, criando um novo tipo de objeto. Este desejável tipo de subclasse é caracterizado por: baixo número de métodos sobrescritos; número decrescente de métodos adicionados e nenhum ou poucos métodos excluídos. Subclasse por implementação é o uso conveniente de alguns métodos e/ou atributos das superclasses por uma subclasse. Ou seja, a subclasse foi criada apenas para aproveitar alguns recursos e não porque ela é do mesmo tipo da superclasse.

A fórmula desta métrica é: (número de métodos sobrescritos * nível da classe na árvore de herança)/total de métodos. Lorenz (1994), utiliza 15% como sinal de anormalidade, o qual resulta em um máximo de três métodos sobrescritos no primeiro nível da árvore de herança. O nível da árvore na árvore de herança é igual à Profundidade da arvore de herança mais um.

4 DESENVOLVIMENTO DO TRABALHO

O presente trabalho resultou na criação de um protótipo de software que possibilita analisar código fonte orientado a objeto em Delphi e fornecer algumas das métricas estudadas.

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

4.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O objetivo do desenvolvimento deste trabalho foi criar uma ferramenta capaz de fornecer métricas pré-definidas a partir da análise de código fonte de programas codificados em Delphi. As informações para cálculo das métricas são obtidas através da extração das classes de seus métodos e de seus atributos através da análise das *units* de um projeto em Delphi. Após a extração destes dados são calculadas algumas das métricas estudadas.

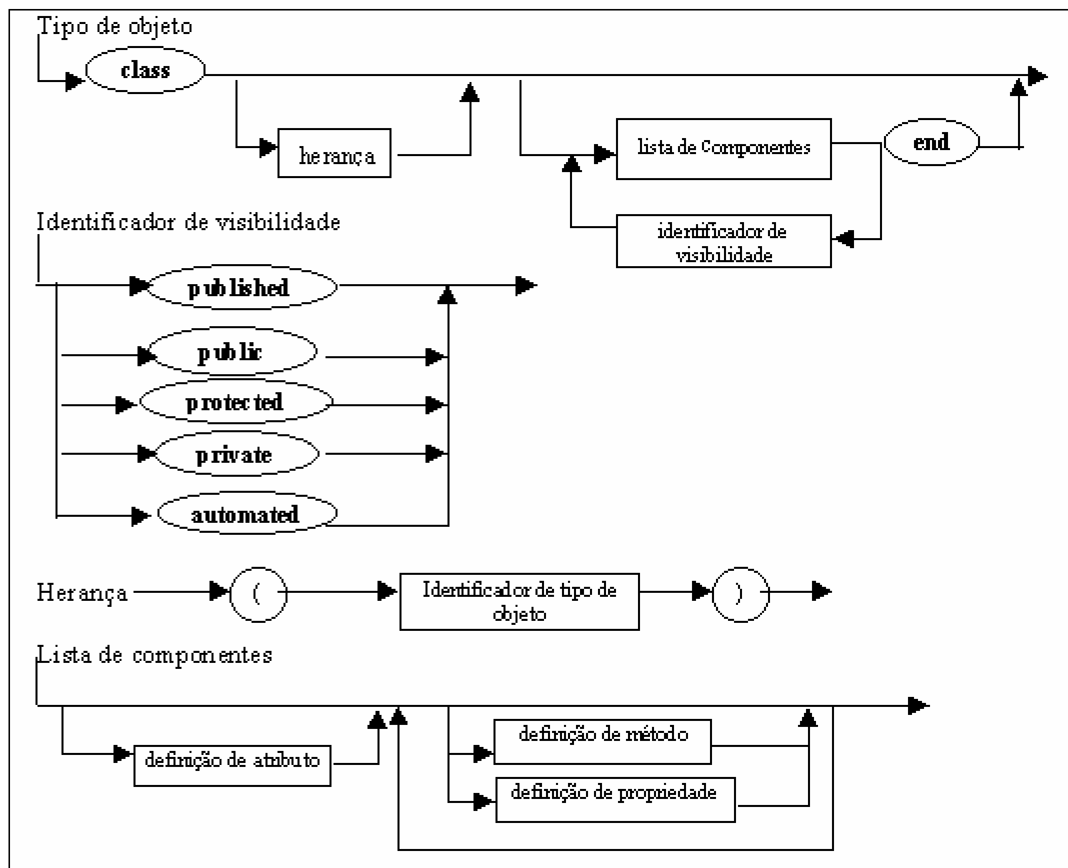
Para extrair e identificar do código fonte as informações das classes foram utilizados os diagramas de sintaxe existentes em Borland (1997). Estes diagramas representam a definição formal da linguagem *Object Pascal,* usada no ambiente Delphi. Para interpretar um diagrama de sintaxe deve-se seguir as setas. Freqüentemente é possível se ter mais de um caminho. A figura 6 apresenta o diagrama de sintaxe do cabeçalho de *procedure*. Neste diagrama a lista de parâmetros é opcional. Os nomes que estão nos retângulos representam as construções, sendo cada um deles representado por outro diagrama de sintaxe. O conteúdo das circunferências representa palavras reservadas, operadores e pontuação da linguagem, sendo estes os termos encontrados no código fonte.

Figura 6 – DIAGRAMA DE SINTAXE DO CABEÇALHO DE *PROCEDURE*

Fonte: Borland (1997)

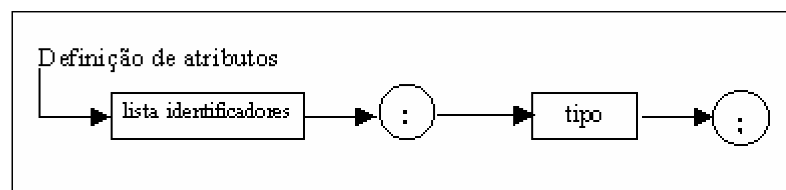
A figura 7 mostra o diagrama de sintaxe da definição de uma classe em *Object Pascal*. Os diagramas de sintaxe da figura 8 e da figura 9 representam as definições dos atributos e dos métodos respectivamente. Maiores informações sobre os diagramas de sintaxe utilizados na especificação e implementação do protótipo podem ser encontradas em Borland (1997).

Figura 7 – DIAGRAMA DE SINTAXE DA DEFINIÇÃO DE UMA CLASSE



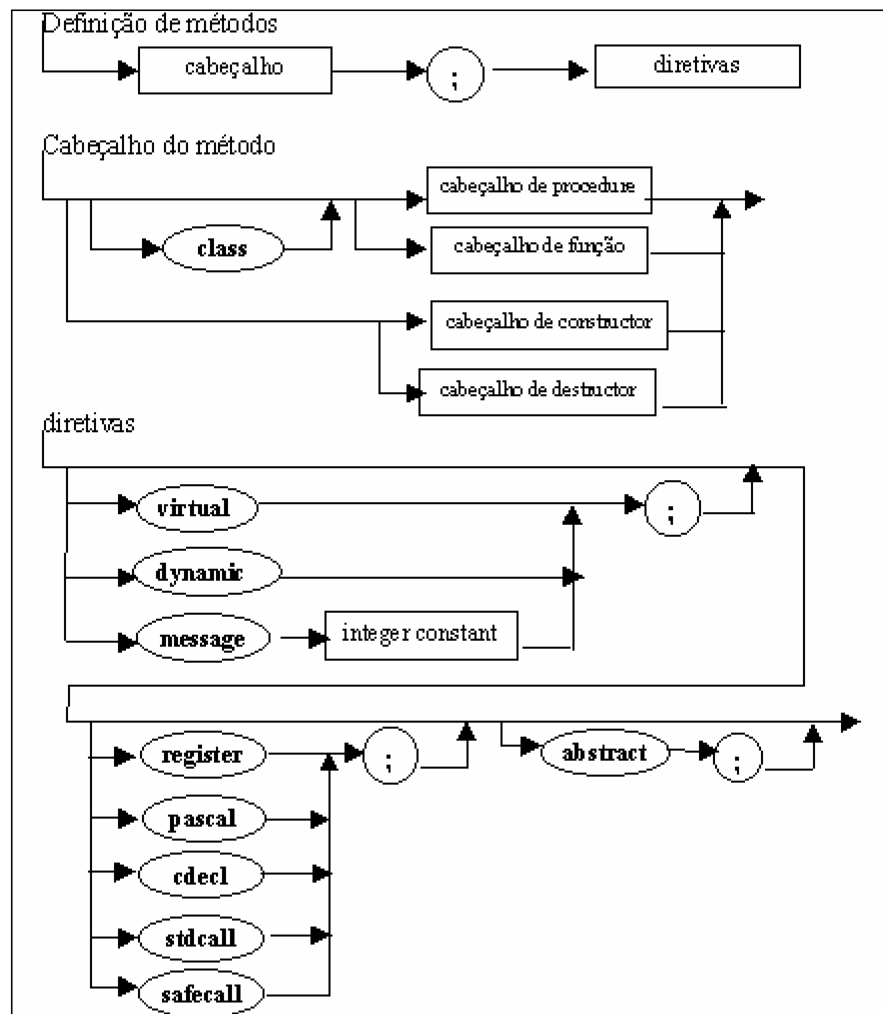
Fonte: Borland (1997)

Figura 8 – DIAGRAMA DE SINTAXE DA DEFINIÇÃO DE UM ATRIBUTO



Fonte: Borland (1997)

Figura 9 – DIAGRAMA DE SINTAXE DA DEFINIÇÃO DE UM MÉTODO



Fonte: Borland (1997)

4.2 MÉTRICAS SELECIONADAS

As métricas de projeto e de construção são as mais indicadas para se obter através da análise do código fonte, pois a maioria das informações necessárias para o cálculo destas métricas pode ser obtida através de análise automática do código fonte. As métricas a seguir foram as implementadas pelo protótipo:

a) profundidade da árvore de herança;

b) número de filhos;

c) contagem de métodos;

d) quantidade de atributos por classe;

e) resposta de uma classe;
f) média do tamanho dos métodos;
g) percentual comentado;
h) acoplamento entre objetos;
i) falta de coesão;
j) quantidade de métodos de instância em uma classe;
k) quantidade de métodos de classe em uma classe;
l) quantidade de métodos sobrescritos por uma classe;
m) índice de especialização;
n) quantidade de métodos herdados por uma subclasse;
o) quantidade de métodos adicionados por uma subclasse;
p) métodos ponderados por classe;
q) média de atributos por classe;
r) média de métodos por classe;
s) média de métodos públicos por classe.

4.3 ESPECIFICAÇÃO DO PROTÓTIPO

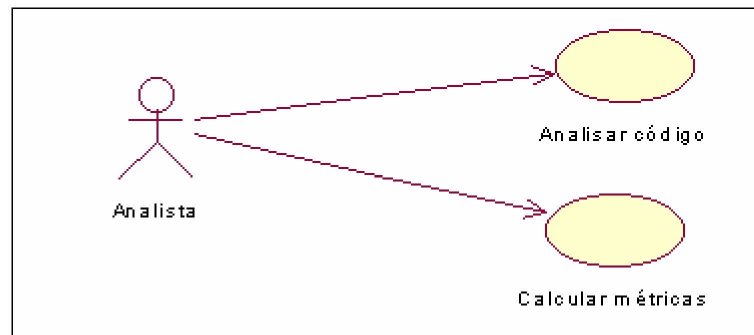
Para a especificação do protótipo foi utilizado a UML, que é apresentado através do diagrama de caso de uso, do diagrama de classes e do diagrama de seqüência. Estes diagramas serão apresentados a seguir e foram construídos na ferramenta CASE *Rational Rose*.

4.3.1 DIAGRAMA DE CASO DE USO

Para o software foi elaborado o diagrama de casos de uso apresentado na figura 10, que se refere à interação do analista para análise do código fonte e cálculo das métricas:

a) Analisar Código: é a ação executada pelo analista após a seleção do projeto para analisar o código fonte das *units* do mesmo e extrair as informações das classes, seus atributos e métodos.
b) Calcular métricas: é a ação executada pelo analista para calcular as métricas a partir das informações extraídas anteriormente.

Figura 10 – DIAGRAMA DE CASO DE USO



4.3.2 DIAGRAMA DE CLASSES

No desenvolvimento do protótipo foram identificadas cinco classes (figura 11) que são utilizadas para armazenar as informações extraídas do código fonte e posteriormente auxiliar na obtenção das métricas.

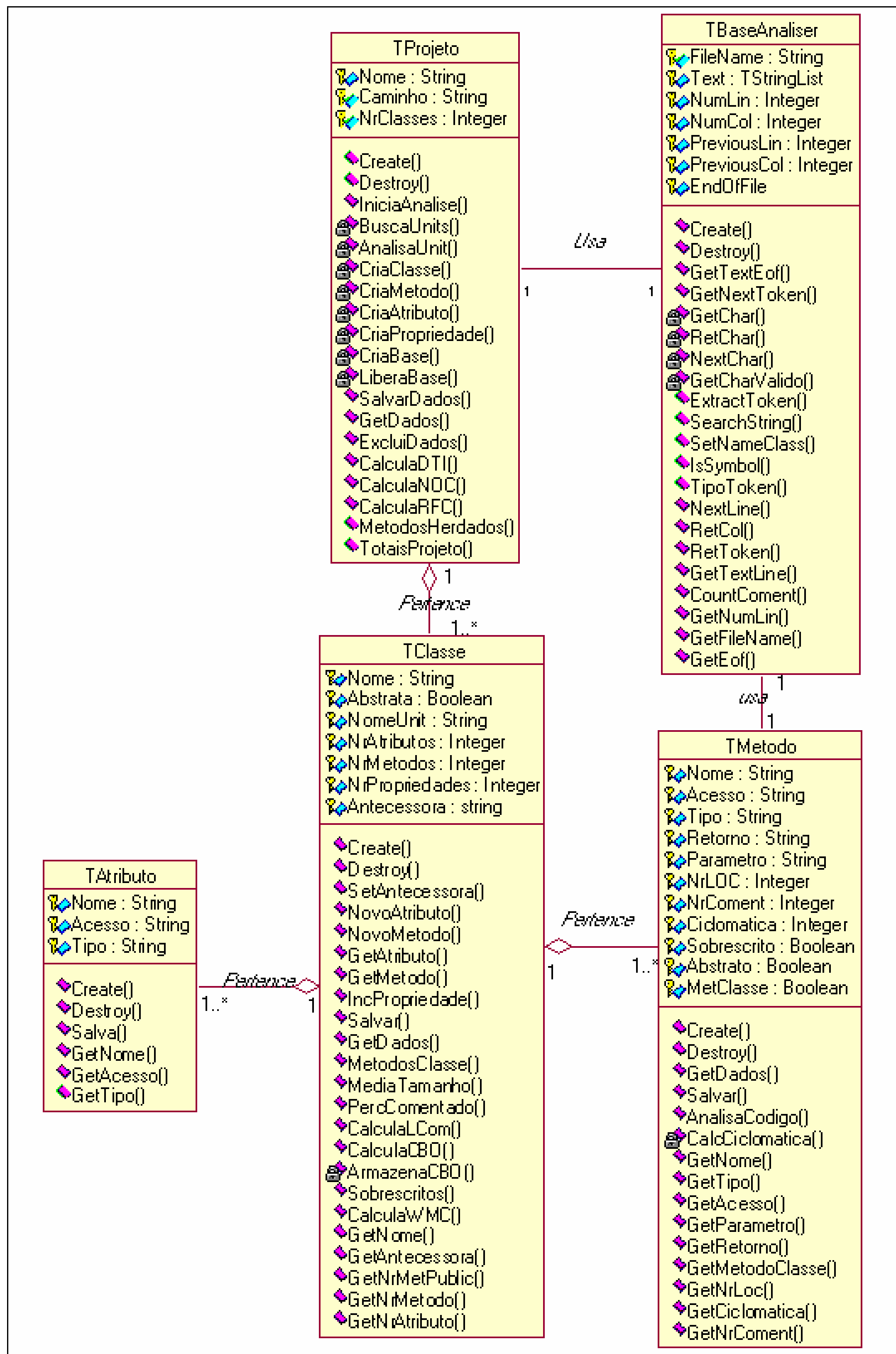
A classe TProjeto, apresentada na figura 11, é responsável por iniciar a análise do código fonte das *units* do projeto selecionado e a partir desta análise criar as classes TClasse, TMetodo e TAtributo. Esta classe também é responsável pela leitura e atualização da base de dados e pelo fornecimento de algumas métricas de projeto, que necessitam da análise das classes utilizadas por outras classes. Ela contém os seguintes atributos:

a) Nome: contém o nome do projeto;
b) Caminho: contém o caminho onde se encontra o projeto;
c) NrClasses: contém o número de classes que o projeto possui.

Existem ainda os métodos da classe que são:

a) Create: operação responsável por criar a classe TProjeto;
b) Destroy: libera o espaço em memória que a classe está utilizando;
c) IniciaAnalise: operação responsável pela análise do código fonte para extração das definições de classes, seus métodos e atributos;
d) BuscaUnits: função privada responsável por criar a lista das *units* pertencentes ao projeto;
e) AnalisaUnit: função privada responsável pela análise da *unit* propriamente dita;
f) CriaClasse: função privada responsável pela criação de uma classe para o projeto;

Figura 11 – DIAGRAMA DE CLASSES



g) CriaAtributo: função privada cria um novo atributo para a classe analisada;
h) CriaMetodo: função privada que cria um novo método para a classe analisada;
i) CriaBase: abre a base de dados e cria as tabelas para pesquisa e atualização de projetos já analisados;
j) LiberaBase: libera a base de dados;
k) SalvarDados: grava na base de dados as informações do projeto analisado;
l) GetDados: operação responsável por pesquisar as informações de um determinado projeto na base de dados;
m) ExcluiDados: operação responsável pela exclusão de todas as informações do projeto da base de dados;
n) CalculaDTI: função que calcula a profundidade da árvore de herança (DTI) de uma classe, retrocedendo na árvore até chegar na raiz, ou seja, até a classe não possuir mais antecessora. Só verifica as classes implementadas no projeto, não considerando os objetos da linguagem;
o) CalculaNOC: calcula o número de filhos de determinada classe, contando o número de classes em que a mesma aparece como antecessora;
p) CalculaRFC: retorna a quantidade de métodos públicos da classe e de seus pais;
q) MetodosHerdados: retorna a quantidade de métodos herdados pela classe;
r) TotaisProjeto: calcula os totais do projeto.

A classe TBaseAnaliser, apresentada na figura 11, é a que auxilia na análise do código fonte. Ela faz a busca e extração de palavras reservadas, identificadores e símbolos especiais. Esta classe possui os seguintes atributos:

a) FileName: possui o nome do arquivo que está sendo analisado;
b) Text: contém o texto do arquivo a ser analisado;
c) NumLin: possui o número da linha atualmente em análise;
d) NumCol: possui o número da coluna atualmente em análise;
e) PreviousLin: possui o número da linha do último *token* extraído;
f) PreviousCol: possui o número da coluna do último *token* extraído;
g) EndofFile: indica se a leitura chegou ao final de arquivo;

Os métodos desta classe são listados a seguir:

a) Create: operação responsável por criar a classe TBaseAnaliser;
b) Destroy: libera o espaço em memória que a classe está utilizando;
c) GetChar: função privada que retorna o caracter que está na linha e coluna passadas pelo parâmetro;
d) RetChar: função privada que retrocede um caracter no texto;
e) NextChar: função privada que avança um caracter no texto;
f) GetTextEof: função que retorna o número da ultima linha do texto;
g) GextNextToken: função que extrai uma *String* até encontrar o delimitador identificado;
h) ExtractToken: extrai o próximo *token* válido;
i) GetCharValido: função privada que retorna um caracter diferente de espaços; pode avançar ou retroceder no texto;
j) SearchString: procura determinada *string* a partir da linha especificada pelo parâmetro;
k) SetClassName: retorna a especificação de uma classe;
l) IsSymbol: verifica se o próximo caracter no texto é um símbolo especial;
m) TipoToken: verifica se o *token* é uma palavra reservada, retornando seu tipo;
n) NextLine: avança para a próxima linha do texto;
o) RetCol: retrocede uma coluna no texto a partir das posições atuais;
p) RetToken: retorna ao *token* extraído anteriormente;
q) CountComent: retorna a quantidade de linhas comentadas no intervalo especificado pelo parâmetro;
r) GetNumLin: retorna o número da linha atualmente em análise;
s) GetFileName: retorna o nome do arquivo que está sendo analisado.

A classe TClasse, apresentada na figura 11, contém as informações das classes que são encontradas no código fonte, ela é instanciada a partir da classe TProjeto e possui os seguintes atributos:

a) Nome: contém o nome da classe;
b) Abstrata: indica se a classe é abstrata;
c) NomeUnit: contém o nome da unit onde a classe é implementada;
d) NrAtributos: contém a quantidade de atributos da classe;

e) NrMetodos: contém a quantidade de métodos da classe;
f) NrPropriedades: contém a quantidade de propriedades da classe.

Existem ainda os métodos da classe TClasse que são:

a) Create: operação responsável por criar a classe TClasse;
b) Destroy: libera o espaço em memória que a classe está utilizando;
c) SetAntecessora: operação que atualiza o nome da classe antecessora;
d) NovoAtributo: cria um novo atributo para a classe;
e) NovoMetodo: cria um novo método para a classe;
f) GetMetodo: busca as informações de um método especificado pelo parâmetro;
g) GetAtributo: busca as informações de um atributo especificado pelo parâmetro;
h) IncPropriedade: incrementa a quantidade de propriedades da classe;
i) Salvar: salva os dados da classe na base de dados;
j) GetDados: busca as informações de TClasse da base de dados;
k) MetodosClass: retorna a quantidade de métodos de classe da classe;
l) MediaTamanho: retorna a média de tamanho dos métodos da classe;
m) PercComentado: retorna o percentual de linhas comentadas em relação às linhas de código;
n) CalculaLCom: calcula a falta de coesão dos métodos da classe. Para cada atributo da classe é calculado o percentual de métodos que o utilizam, em seguida calcula-se a média destes percentuais e subtrai de 100;
o) CalculaCBO: calcula o acoplamento entre os objetos. Verifica se os atributos, as variáveis utilizadas pelos métodos, os tipos de retorno e parâmetros dos métodos ou variáveis dos métodos são classes. A quantidade de classes distintas resulta no acoplamento;
p) ArmazenaCBO: operação privada utilizada para auxiliar no cálculo do acoplamento, armazena em uma lista o *token* passado se ele for uma classe;
q) Sobrescritos: retorna a quantidade de métodos sobrescritos pela classe;
r) CalculaWMC: calcula os métodos ponderados por classe (WMC), ou seja, a soma das complexidades ciclomáticas dos métodos da classe;
s) GetNome: retorna o nome da classe;
t) GetAntecessora: retorna o nome da classe antecessora;

u) GetNrAtributo: retorna a quantidade de atributos que a classe possui;
v) GetNrMetodo: retorna a quantidade de métodos que a classe possui;
w) GetNrMetPublic: retorna a quantidade de métodos públicos que a classe possui.

A classe TAtributo, apresentada na figura 11, contém as informações dos atributos das classes encontradas no projeto. Esta classe contém os seguintes atributos:

a) Nome: contém o nome do atributo;
b) Acesso: indica se o atributo é *Public*, *Protected* ou *Private*;
c) Tipo: indica o tipo do atributo.

Já os métodos dessa classe são:

a) Create: operação responsável por criar a classe TAtributo;
b) Destroy: libera o espaço em memória que a classe TAtributo está utilizando;
c) Salvar: salva os dados do atributo na base de dados;
d) GetNome: retorna o nome do atributo;
e) GetAcesso: retorna o acesso do atributo;
f) GetTipo: retorna o tipo do atributo.

A classe TMetodo, apresentada na figura 11, contém as informações dos métodos das classes, extraídas durante a análise do código fonte. Os atributos desta classe são:

a) Nome: contém o nome do método;
b) Acesso: indica se o atributo é *Public*, *Protected* ou *Private*;
c) Tipo: indica se o método é *Procedure*, *Function*, *Constructor* ou *Destructo*r;
d) Retorno: indica o tipo de retorno caso o método seja uma *Function*;
e) Parametro: contém os parâmetros do método;
f) NrLoc: contém o número de linhas de código do método;
g) NrComent: contém o número de linhas de comentário do método;
h) Ciclomatica: contém o valor da complexidade ciclomática do método;
i) Abstrato: indica se o método é abstrato;
j) Sobrescrito: indica se o método é sobrescrito;
k) MetodoClasse: indica se o método é um método classe;

Os métodos da classe TMetodo são:

a) Create: operação responsável por criar a classe TMetodo;
b) Destroy: libera o espaço em memória que a classe TMetodo está utilizando;
c) GetDados: busca as informações do método na base de dados;
d) Salvar: salva os dados do método na base de dados;
e) AnalisaCodigo: operação responsável pela análise do código fonte do método, para cálculo do seu tamanho e de sua complexidade ciclomática;
f) CalcCiclomatica: função privada que calcula a complexidade ciclomática do método;
g) GetNome: retorna o nome do método;
h) GetTipo: retorna o conteúdo do atributo do método;
i) GetAcesso: retorna o conteúdo do atributo acesso;
j) GetParametro: retorna os parâmetros do método;
k) GetRetorno: retorna o conteúdo do atributo Retorno;
l) GetMetodoClasse: retorna o conteúdo do atributo MetodoClasse;
m) GetNrLoc: retorna o conteúdo do atributo NrLoc;
n) GetNrComent: retorna o conteúdo do atributo NrComent;
o) GetCiclocmatica: retorna o conteúdo do atributo Ciclomatica.

4.3.3 DIAGRAMA DE SEQÜÊNCIA

Na elaboração do protótipo identificou-se que o diagrama de seqüência possui duas fases, cada uma correspondente a um caso de uso. A fase 01, na figura 12, indica a seqüência de passos que são realizados para extração das informações do projeto (classes, métodos e atributos), a partir da análise do código fonte.

A fase 02, apresentada na figura 13, identifica a seqüência de passos que são realizados para o cálculo e obtenção das métricas.

Figura 12 – DIAGRAMA DE SEQÜÊNCIA DA ANÁLISE DO FONTE

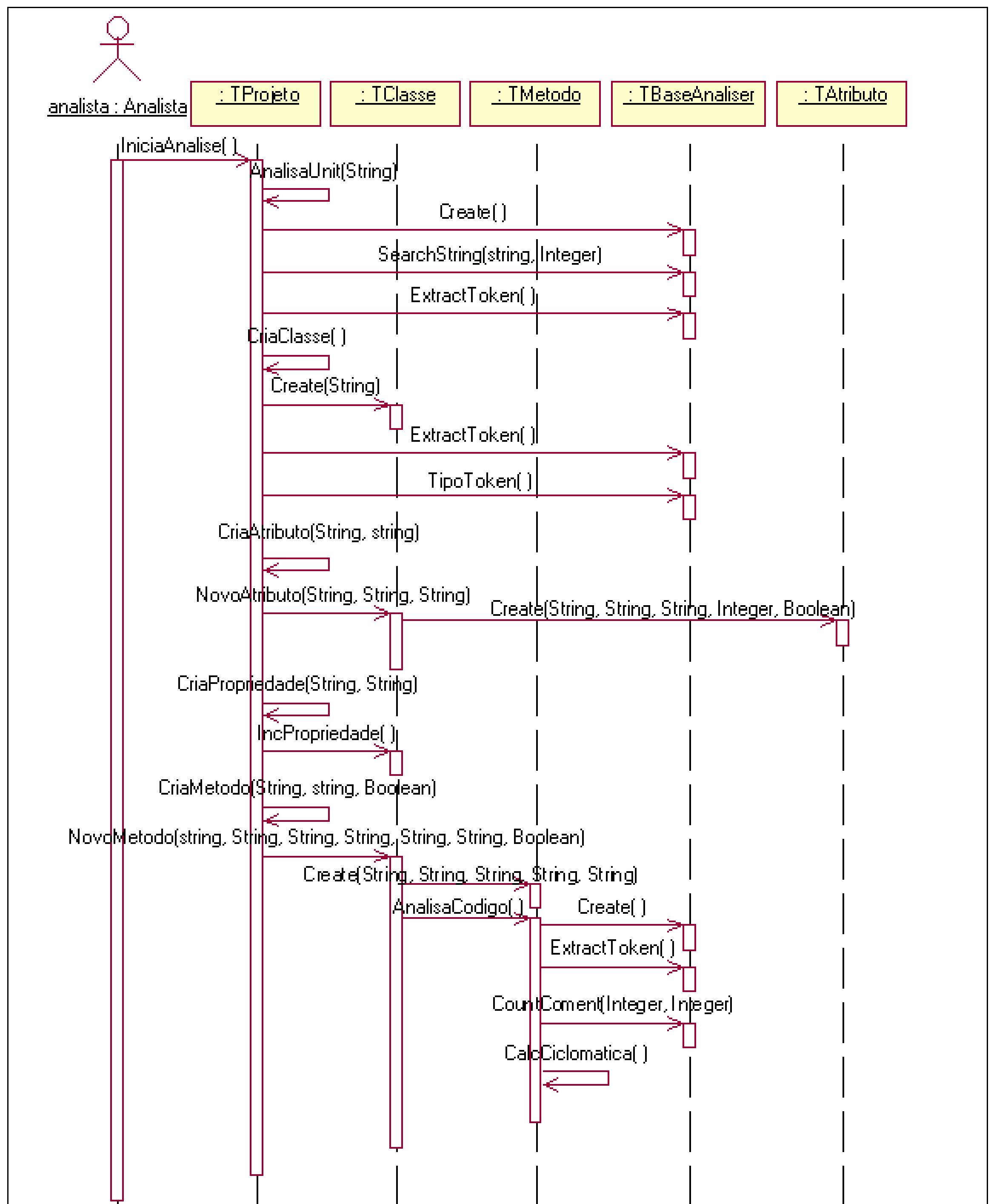
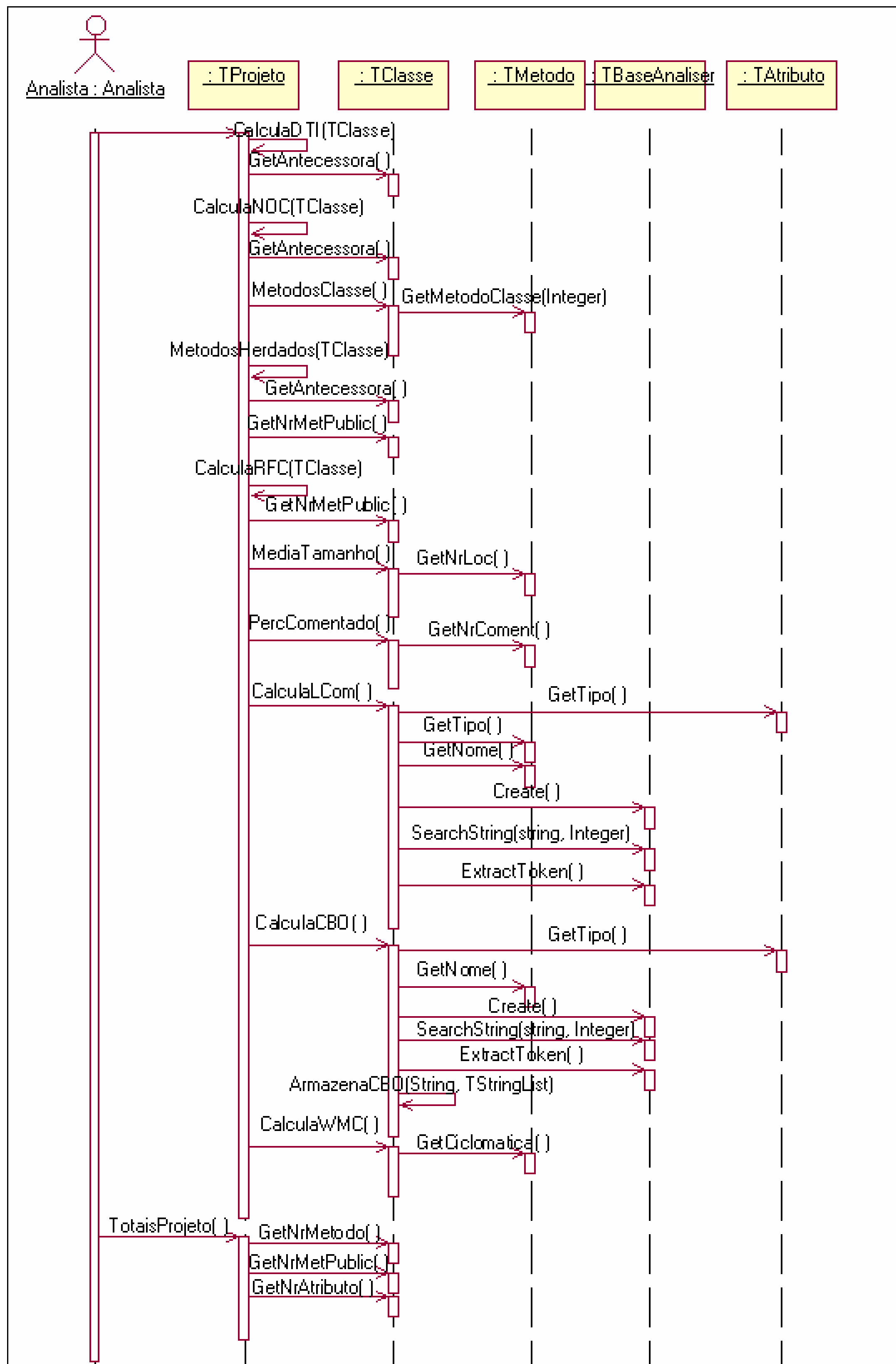


Figura 13 – DIAGRAMA DE SEQÜÊNCIA DO CÁLCULO DAS MÉTRICAS



4.4 IMPLEMENTAÇÃO

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

4.4.1 TÉCNICAS E FERRAMENTAS UTILIZADAS

O protótipo foi implementado no ambiente de desenvolvimento Delphi 5.0, onde foram empregados os conceitos de orientação a objetos para desenvolver as cinco classes responsáveis pela extração das classes do código fonte e cálculo das métricas.

Ao iniciar a análise do código fonte, a classe TProjeto é instanciada e todas as *units* existentes no projeto selecionado serão analisadas. Para tanto, é instanciada a classe TBaseAnaliser que realiza as operações de extração e reconhecimento de *tokens*, o código fonte da *unit* é armazenado temporariamente em uma *TStringList* (componente da linguagem). Após a análise do código fonte a classe TBaseAnaliser é liberada da memória. Ao encontrar uma definição de classe é chamado o método CriaClasse que irá criar a classe e procurar as definições de seus métodos e atributos. Este método é apresentado no quadro 1.

Quadro 1 – MÉTODO CRIACLASSE

```
procedure TProjeto.CriaClasse;
var
  Token, Acesso: string;
begin
  inc(NrClasses);
  ClasseAux := TClasse.Create(Source.SetNameClass, Source.GetFileName);
  { Heranca }
  if Source.IsSymbol('(', True) then begin
    ClasseAux.SetAntecessora(Source.GetNextToken(')'));
    end
  else
    Source.RetCol;

  Acesso := 'PUBLIC';
  Token := Source.ExtractToken;
  repeat
    Case Source.TipoToken(Token) of
      trwMethod: CriaMetodo(Token, Acesso, False);
      trwVisibility: Acesso:= Token;
      trwNada: CriaAtributo(Token, Acesso);
      trwProperty: CriaPropriedade(Token, Acesso);
      trwMetClass: begin
         Token := Source.ExtractToken;
         CriaMetodo(Token, Acesso, True)
         end;
    end;
    Token := Source.ExtractToken;
  until UpperCase(Token) = 'END';
 SClasses.AddObject(ClasseAux.GetNome,ClasseAux);
 Source.NextLine;
end;
```

O quadro 2 apresenta o método AnalisaUnit da classe TProjeto que procura definições de classe numa *unit*.

Quadro 2 – MÉTODO ANALISA UNIT

```
function TProjeto.AnalisaUnit(PUnit: string): string;
var
  i,
  LinImp,    // linha onde começa implementation - posionar nesta linha p/ contar loc
  LinAtu: integer;
  Token : string;
begin
  try
    Source := TBaseAnaliser.Create(PUnit);
    if Source.GetEof then
      Raise Exception.Create('Unit não encontrada: ' + PUnit);

    LinImp := 0;
    if Source.SearchString('TYPE', 0) then begin
       // procurar as classes seus métodos e atributos

       repeat
         Token := Source.ExtractToken;
         if (UpperCase(Token) = 'CLASS') then begin
           for i:= 1 to (Length(Token) -1) do
              Source.RetCol;
           if  Source.IsSymbol('=',False) then
             CriaClasse;
           end;
         LinAtu := Source.GetNumLin;
         if UpperCase(Token) = 'IMPLEMENTATION' then
           LinImp := LinAtu;
       until (LinAtu >= Source.GetTextEof) or (LinImp > 0) or (Source.GetEof);
       end;
    Source.Free;
    Result := 'Unit analisada: '+ PUnit +#13#10;

  except
    if Source.GetEof then
      Result := Exception(ExceptObject).Message +#13#10;
    if Assigned(Source) then
      Source.Free;
    end;

end;
```

As informações extraídas através da análise do código fonte podem ser salvas em uma base de dados. Esta base de dados consiste de tabelas em *Paradox* e as informações gravadas correspondem à definição dos atributos existente no Diagramas de Classes para as classes: TProjeto, TClasse, TMetodo e TAtributo. A classe TBaseAnaliser não é armazenada na base de dados, pois não é uma classe persistente.

No quadro 3, é apresentado o método ExtractToken da classe TBaseAnaliser, este método retorna um *token* extraído do texto. *Token* são as menores unidades de um texto que possuem significado, e podem ser: símbolo especial da linguagem, identificadores, palavras reservadas.

Quadro 3 – MÉTODO EXTRACTTOKEN

```
function TBaseAnaliser.ExtractToken: string;
var
  aux,s: string;
begin
  PreviousCol := NumCol;
  PreviousLin := NumLin;

  if NumCol = 1 then
    RetCol;

  { procura primeiro simbolo se for especial retorna
   se nao monta palavra até achar um especial ou espaço}
  aux := GetCharValido(True);
  if aux = #39 then begin
    repeat
      GetNextToken(#39);
      aux := GetCharvalido(True);
    until (aux <> #39);
    end;

  // se o simbolo encontrado for um comentário ignora o que vem depois
  while (aux = '{') or (aux ='/') do begin
    if aux = '{' then
      GetNextToken('}');
    if aux = '/'  then begin
       aux := GetCharValido(True); // se acha uma barra de comentário deve achar a próxima
       if aux ='/' then begin
         NextLine;
         NumCol := 0;
         end;
       end;
    aux := GetCharValido(True);
    end;

  if not (TipoToken(Aux) = trwSymbol)  then begin
    s:= '';
    while (s <> ' ') and not (TipoToken(s) = trwSymbol)  and (not EndofFile) do begin
      aux:= aux + s;
      if NumCol = Length(Text[NumLin]) then begin
        s:= '';
        break;
        end;
      NextChar(NumLin, NumCol);
      s := GetChar(NumLin,NumCol);
      end;
    if (s <> '') and (s <> ' ') then
      RetCol;
    end;
  Result := Trim(aux)

end;
```

Para o cálculo de algumas métricas de uma classe é preciso percorrer toda a árvore do projeto, como no caso da profundidade da árvore de herança. O quadro 4 apresenta o método CalculaDTI que é executado pela classe projeto para cada uma das classes do mesmo.

Quadro 4 – MÉTODO CALCULADTI

```
function TProjeto.CalculaDTI(Classe: TClasse): integer;
var
  idx, dti: integer;
  ClasseAux : TClasse;
  Antecessora: string;
begin
```

```
  { Calcula a profundidade da arvore de heranca da classe, vai voltando a arvore até que
  chegar na raiz ou seja a classe nâo possui antecessora , este método só verifica as classes
  implementadas no projeto, nâo considerando os objetos da linguagem }
  DTI := 0;
  if Classe.GetAntecessora <> '' then begin
     Antecessora := Classe.GetAntecessora;
     repeat
       idx:= SClasses.IndexOf(Antecessora);
       if idx >= 0 then begin
         ClasseAux := TClasse(SClasses.Objects[idx]);
         Antecessora := ClasseAux.GetAntecessora;
         inc(Dti);
         end;
     until (Antecessora = '') or (idx < 0);
     end;
  Result := Dti;
end;
```

Para algumas métricas é necessário varrer o código fonte da classe. É o caso do Acoplamento entre Objetos, para se obter esta métrica é necessário verificar quais atributos, parâmetros de métodos e variáveis de métodos são classes. O quadro 5 mostra o método CalculaCBO implementado na classe TClasse que realiza este cálculo.

Quadro 5 – MÉTODO CALCULACBO

```
function TClasse.CalculaCBO: integer;
var
  LstAcopladas: TStringList;
  i: integer;
  Token: string;
begin
  CarregaFonte;
  LstAcopladas := TStringList.create;
  if Antecessora <>'' then
    lstAcopladas.Add(Antecessora);
  for i:= 0 to pred(NrAtributos) do  // verifica atributos
    ProcuraCBO(Atributos[i].GetTipo,LstAcopladas);
   // verificar se tipo de retorno, parametros ou variaveis dos métodos são classes
   for i:= 0 to pred(NrMetodos) do begin
     if not Metodos[i].GetAbstrato then begin
       Source.SearchString(Nome+'.'+Metodos[i].GetNome,0);
       repeat
         Token := Source.ExtractToken;
         if Token = ':' then begin
           repeat
             Token := Source.ExtractToken;
             if Source.TipoToken(Token) = trwNada then
               ProcuraCBO(Token, LstAcopladas);
             if Token = '[' then
               Source.GetNextToken(']');
             until (Token = ';') or (Token=')');
            end; // token = :
         until (UpperCase(Token) = 'BEGIN');
       end; // if
     end; // for

  Result := LstAcopladas.Count;
  LstAcopladas.Destroy
end;
```

O método ArmazenaCBO, apresentado no Quadro 6, é uma função privada auxiliar que armazena as classes acopladas.

Quadro 6 – MÉTODO ARMAZENACBO

```
procedure TClasse.ArmazenaCBO(PStr: string; var PAcoplamento: TStringList);
var
  s, s1 : string;
  i,j: integer;
  FimStr: Byte;
begin
  s := Pstr;
  if pos('[',s) > 0 then  begin
    i:= pos('[',s);
    j:= pos(']',s);
    Delete(s,i, j-i +1);
    end;
  Trim(s);
  repeat
     FimStr := Pos(' ', s);
     if FimStr = 0 then
       s1:= s
     else  begin
       s1 := Copy(s,1,FimStr -1);
       s:= Copy(s, FimStr + 1, length(s) - FimStr);
     end;
     if not (Source.TipoDados(s1)) and  (trim(s1) <> '') and
        (PAcoplamento.Indexof(s1) = -1) and (Source.TipoToken(s) = trwnada) then
           PAcoplamento.Add(s1);
   until FimStr = 0;
end;
```

4.4.2 OPERACIONALIDADE DA IMPLEMENTAÇÃO

A seguir serão apresentadas algumas telas do protótipo com suas respectivas funcionalidades. Com intuito de facilitar a demonstração e compreensão, será realizada a análise e o cálculo das métricas a partir do código fonte do próprio protótipo desenvolvido.

Ao iniciar o programa será apresentada a tela principal do programa ao usuário como demonstra a figura 14.

Para efetuar o cálculo das métricas de um projeto o usuário deverá selecionar um projeto. Esta seleção pode ser feita de duas formas. A primeira forma é selecionar um projeto novo para análise, para isto o usuário deverá selecionar a opção Projeto e posteriormente Novo. Feito isto aparecerá uma janela para seleção de um projeto em Delphi (arquivos com extensão “dpr”) conforme figura 15, para que o usuário escolha o projeto que deseja analisar. Escolhido o projeto é iniciada a análise do código fonte de onde são extraídas as classes e seus respectivos métodos e atributos.

Figura 14 – TELA PRINCIPAL DO PROTÓTIPO

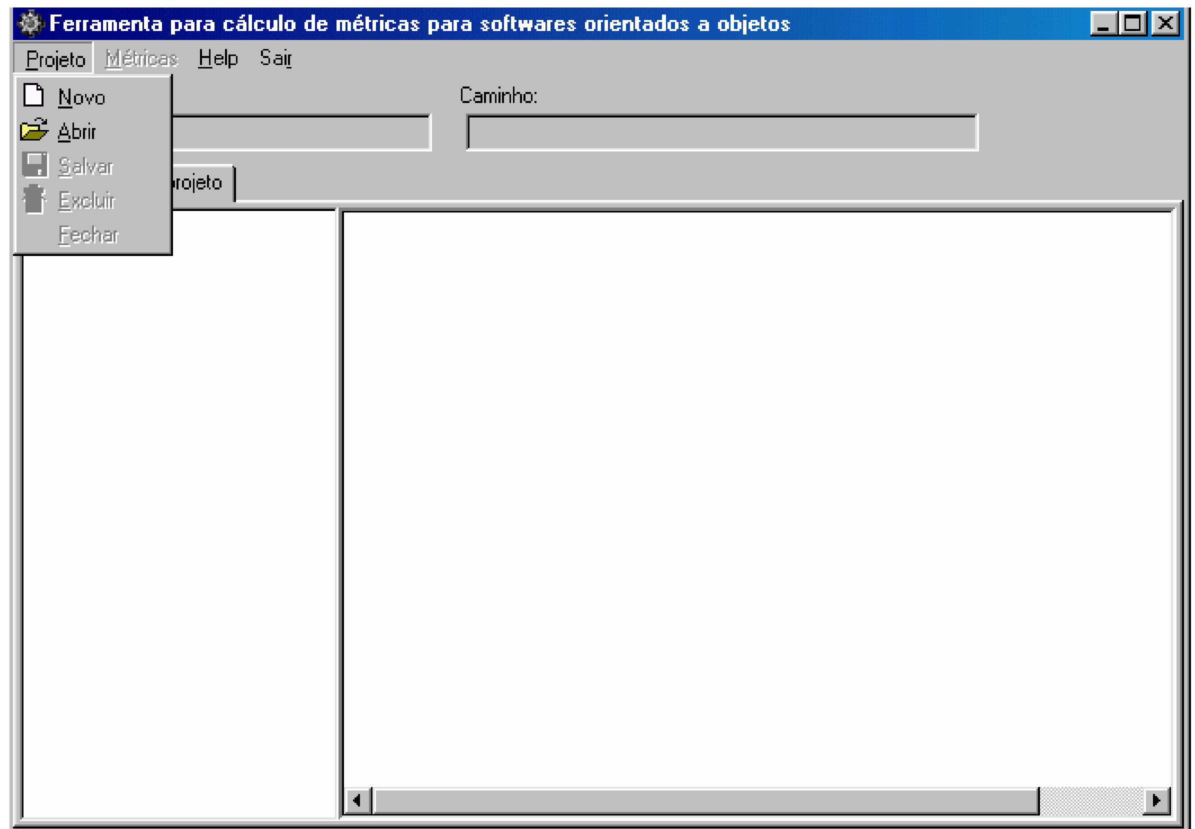
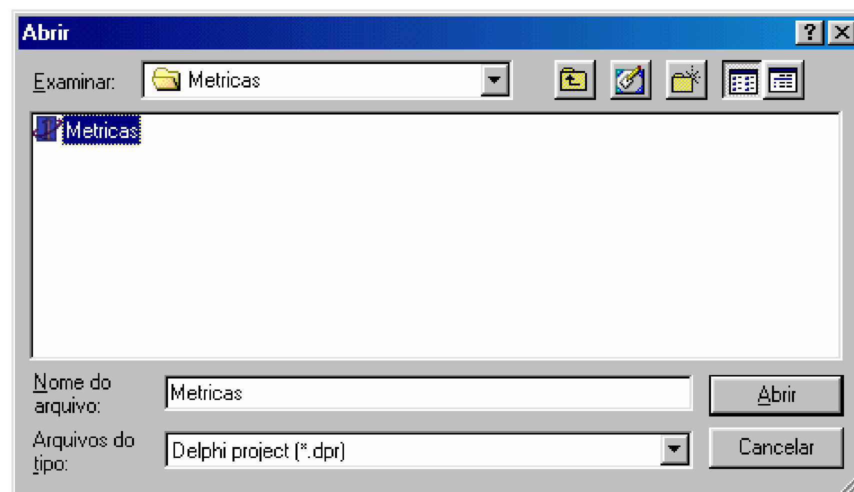
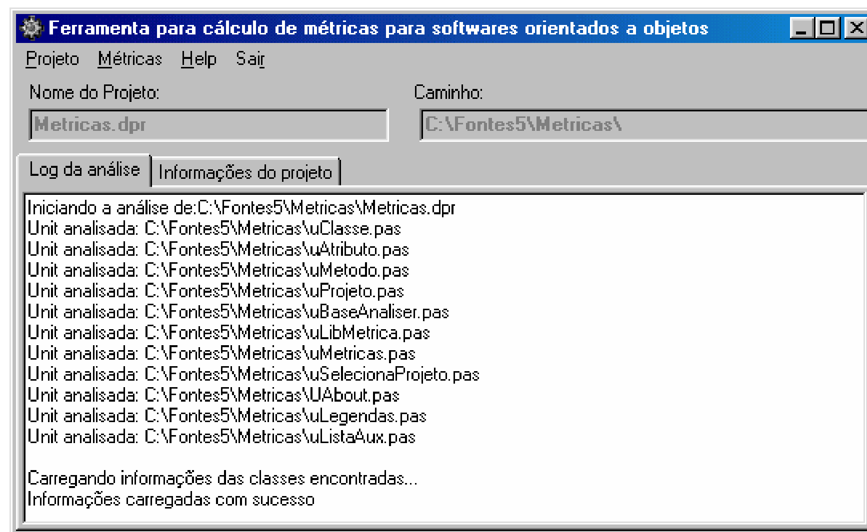


Figura 15 – TELA DE ABERTURA DO PROJETO A SER ANALISADO



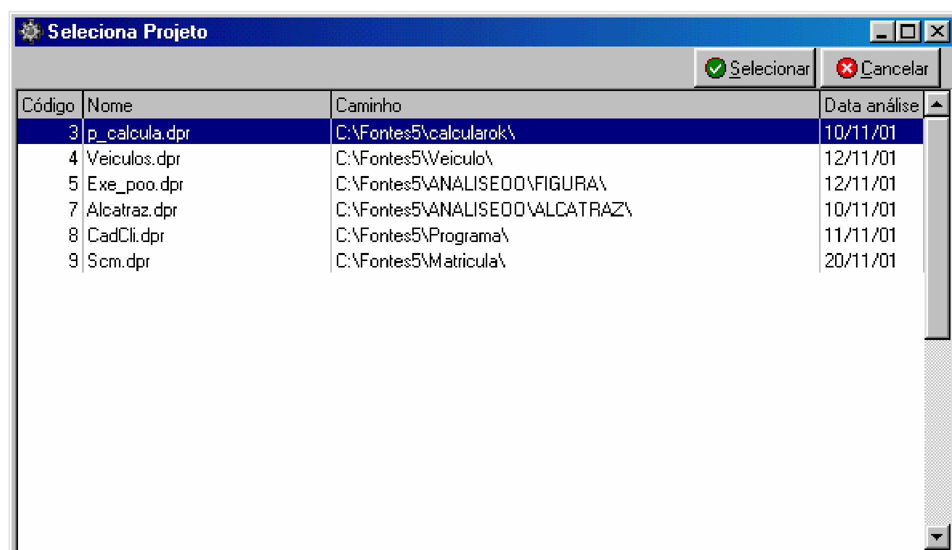
Durante a análise do código fonte é alimentado um *log* que contém o nome de todas as *units* encontradas no arquivo analisado e seu status após a análise, como mostra a figura 16.

Figura 16 – LOG DAS UNITS ANALISADAS



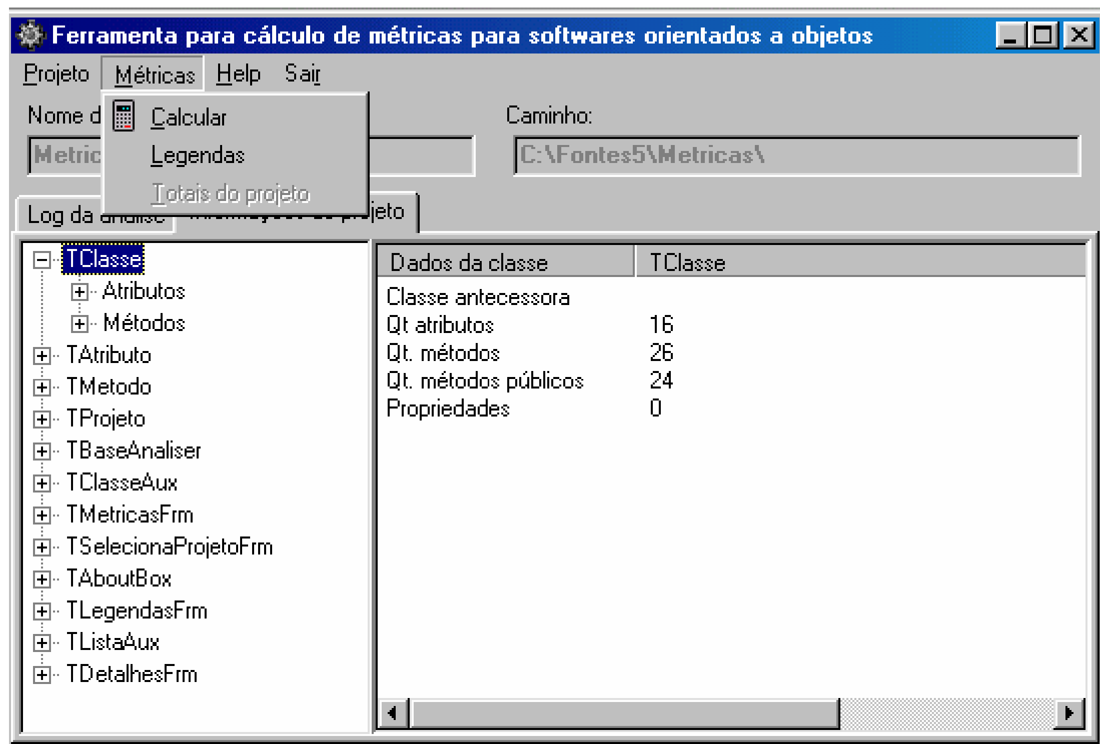
Outra maneira de selecionar um projeto para o cálculo de suas métricas é abrir um projeto analisado previamente e armazenado na base de dados. Para tanto o usuário deverá selecionar a opção Projeto e posteriormente Abrir. O protótipo mostrará a tela da figura 17, que lista todos os projetos analisados anteriormente e armazenados na base de dados.

Figura 17 – TELA DE SELEÇÃO DE PROJETOS NA BASE DE DADOS



Selecionado o projeto, é habilitada a opção Métricas e as informações do projeto que foram extraídas são listadas como mostra a figura 18.

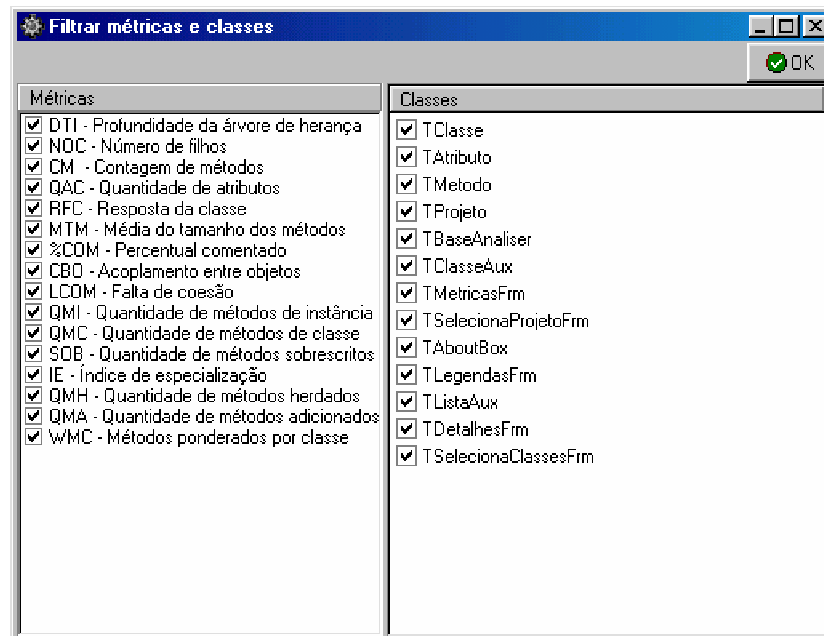
Figura 18 – TELA DE INFORMAÇÕES DO PROJETO



Ao selecionar a Opção métricas e em seguida calcular é apresentada a tela da figura 19. Esta tela consiste de um filtro que permite que o usuário escolha quais métricas ele deseja calcular e para quais classes.

Em seguida será realizado o cálculo das métricas. Algumas métricas são extraídas pela contagem dos atributos existentes, por exemplo, a Quantidade de Atributos por classe. Outras necessitam que se percorra o código fonte dos métodos das classes para extrair as informações, exemplos destas métricas são: Acoplamento entre Objetos e Falta de Coesão. Uma terceira forma que o protótipo obtém algumas métricas é percorrendo todas as classes do projeto, isto é necessário no cálculo da Profundidade da Árvore de Herança, por exemplo.

Figura 19 – TELA DE FILTRO DAS MÉTRICAS E CLASSES



Após o cálculo das métricas os resultados são apresentados como mostra a figura 20. Esta tela lista todas as métricas calculadas para as classes selecionadas do projeto.

Figura 20 – TELA DE INFORMAÇÕES DAS MÉTRICAS CALCULADAS

Ferramenta para cálculo de métricas para softwares orientados a objetos

Projeto Métricas Help Sair

Nome do Projeto: Metricas.dpr

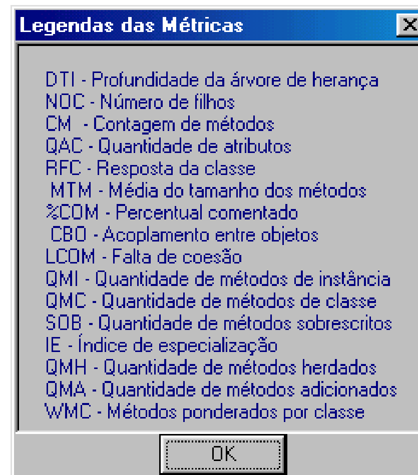
Caminho: C:\Fontes5\Metricas\

Log da análise | Informações do projeto | Métricas | Detalhes

Classe	DTI	NOC	CM	QAC	RFC	MTM	%COM	LCOM	CBO	QMI	QMC	SOB	IE	QMH	QMA	WMC
TClasse	0	0	26	16	24	12.62	6.40	83.65	6	26	0	1	0.04	0	25	72
TAtributo	0	0	6	3	6	6.50	0.00	50.00	1	6	0	1	0.17	0	5	12
TMetodo	0	0	17	13	16	11.59	7.11	76.47	3	17	0	1	0.06	0	16	44
TProjeto	0	0	22	8	14	20.27	5.16	73.86	3	22	0	1	0.05	0	21	67
TBaseAnaliser	0	0	23	9	19	10.83	10.44	76.33	3	23	0	1	0.04	0	22	66
TClasseAux	0	0	0	3	0	0.00	0.00	0.00	0	0	0	0	0.00	0	0	0
TMetricasFrm	0	0	24	41	18	17.67	4.01	93.80	25	24	0	0	0.00	0	24	56
TSelecionaProjetoFrm	0	0	5	12	5	4.00	0.00	91.67	11	5	0	0	0.00	0	5	10
TAboutBox	0	0	0	7	0	0.00	0.00	0.00	4	0	0	0	0.00	0	0	0
TLegendasFrm	0	0	1	4	1	3.00	0.00	100....	6	1	0	0	0.00	0	1	2
TListaAux	0	0	5	1	3	12.20	0.00	0.00	1	5	0	1	0.20	0	4	11
TDetalhesFrm	0	0	3	1	2	11.00	0.00	33.33	6	3	0	0	0.00	0	3	6

A figura 21 é a tela apresentada quando se seleciona a opção Métricas e em seguida Legendas.Esta tela contém todas as legendas das métricas listadas.

Figura 21 – TELA DE LEGENDAS DAS MÉTRICAS



Após o cálculo das métricas ficara habilitada a opção Métricas – Totais do projeto. Ao selecionar esta opção aparecerá a tela da figura 22 listando algumas métricas que são por projeto.

Figura 22 – TELA DE TOTAIS DO PROJETO

Totais do Projeto: Metricas.dpr

Métrica	Valor
Quantidade de classes	12
Média de métodos por classe	11.00
Média de métodos públicos por classe	9.00
Média de atributos por classe	9.83

Se o usuário desejar ver os detalhes das métricas de uma determinada classe, ele precisa selecionar a classe desejada e clicar duas vezes sobre a mesma, isto mostrará a tela apresentada na figura 23. Esta tela contém todas as métricas calculadas para a classe selecionada.

Figura 23 – TELA DE DETALHES DAS MÉTRICAS POR CLASSE

Ferramenta para cálculo de métricas para softwares orientados a objetos

Projeto Métricas Help Sair

Nome do Projeto: Metricas.dpr

Caminho: C:\Fontes5\Metricas\

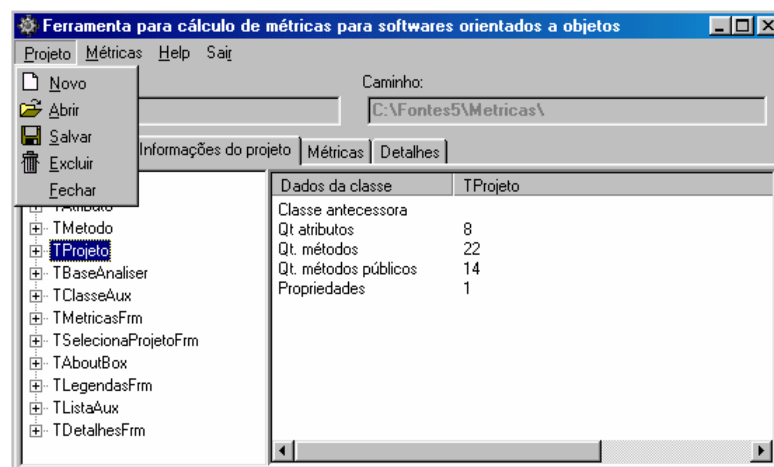
Log da análise | Informações do projeto | Métricas | Detalhes

Classe: TClasse

Métrica	Valor
Profundidade da árvore de herança (DTI)	0
Número de filhos (NOC)	0
Contagem de métodos (CM)	19
Quantidade de atributos (QAC)	12
Resposta da classe (RFC)	17
Média do tamanho dos métodos (MTM)	16.16
Percentual comentado (%COM)	6.84
Falta de coesão (LCOM)	73.68
Acoplamento entre objetos (CBO)	6
Quantidade de métodos de instância (QMI)	19
Quantidade de métodos de classe (QMC)	0
Quantidade de métodos sobrescritos (SOB)	1
Indice de especialização (IE)	0.05
Quantidade de métodos herdados (QMH)	0
Quantidade de métodos adicionados (QMA)	18
Métodos ponderados (WMC)	58

Depois de analisar um projeto o usuário pode salvar suas informações na base de dados, para isto basta selecionar em Projeto a opção Salvar. Um projeto gravado na base de dados pode também ser excluído da mesma através da opção Projeto – Excluir. A opção Projeto – Fechar libera as classes instanciadas durante a análise do projeto da memória, sem excluí-lo da base de dados se ele estiver salvo. A figura 24 apresenta estas três opções.

Figura 24 – TELA DE OPÇÕES DO PROJETO



4.4.3 ESTUDO DE CASO

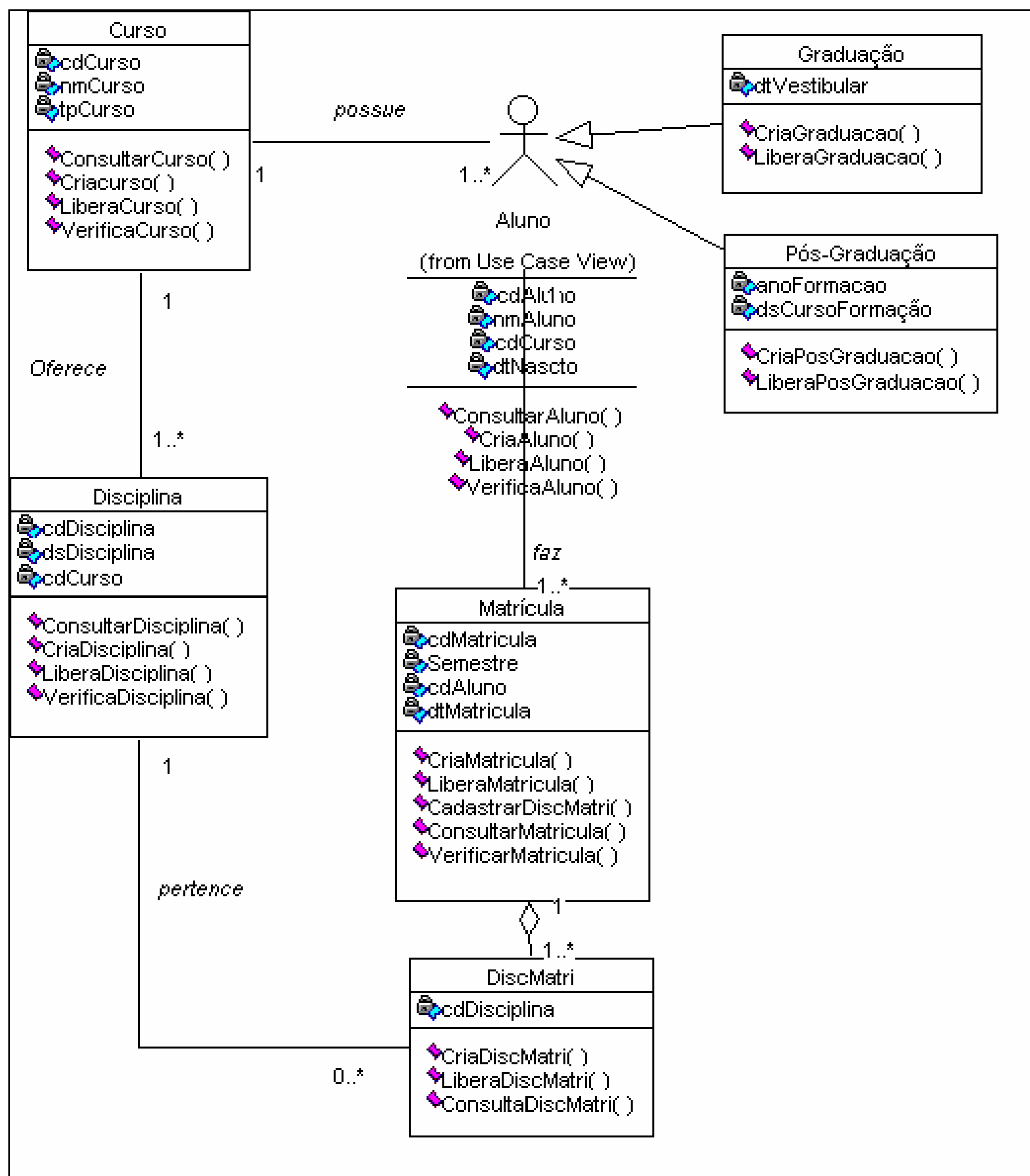
Para exemplificar a utilização do protótipo foi analisado o código fonte do projeto de um sistema fictício de Controle de Matrículas de uma Universidade. Este sistema foi desenvolvido como trabalho final da disciplina Tópicos Especiais III – Análise e Programação Orientada a Objetos, durante o primeiro semestre de 2000.

O quadro 1 apresenta o enunciado do estudo de caso do Sistema de Controle de Matrícula e a figura 25 seu diagrama de classes. O código fonte analisado encontra-se no Anexo 1.

Quadro 7 – ESTUDO DE CASO

O sistema fictício consiste em controlar as matrículas de alunos de graduação e pós-graduação de uma universidade. Os alunos de graduação possuem os seguintes dados de cadastro: código do aluno, nome, data de nascimento, curso e data de aprovação no vestibular. Já os alunos de pós-graduação possuem código, nome, data de nascimento, curso que está sendo freqüentado, ano e curso de formação. A cada início de semestre, o aluno precisa fazer a sua matrícula, escolhendo as disciplinas que irá cursar naquele semestre. A matrícula é composta do código do aluno, número do semestre e data da matrícula.

Figura 25 – DIAGRAMA DE CLASSES DO SISTEMA DE MATRÍCULA



A figura 26 apresenta as informações das classes extraídas a partir da análise do código fonte do sistema de matrícula. Os totais do projeto são apresentados na figura 27. As métricas calculadas por classe para este sistema são apresentadas na figura 28.

Na figura 29 são apresentadas as métricas da classe TAluno e na figura 30 as métricas da classe TGraduacao, que é filha de TAluno. A figura 31 apresenta os detalhes das métricas da classe TCurso.

Figura 26 – INFORMAÇÕES EXTRAÍDAS DO SISTEMA DE MATRÍCULA

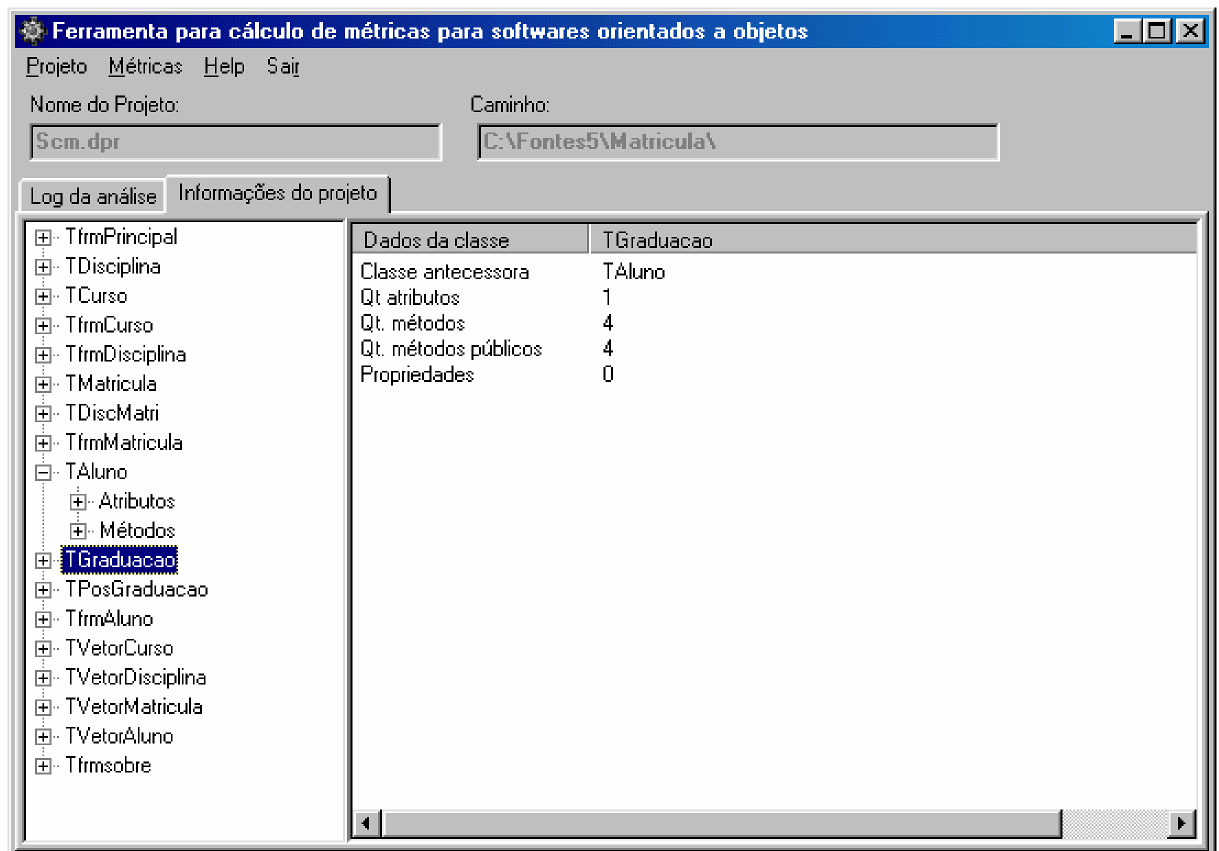


Figura 27 – TOTAIS DO PROJETO DO SISTEMA DE MATRÍCULA

Totais do Projeto: Scm.dpr

Métrica	Valor
Quantidade de classes	17
Média de métodos por classe	4.53
Média de métodos públicos por classe	4.53
Média de atributos por classe	6.47

Figura 28 – MÉTRICAS DO SISTEMA DE MATRÍCULA

Ferramenta para cálculo de métricas para softwares orientados a objetos

Projeto Métricas Help Sair

Nome do Projeto: Scm.dpr

Caminho: C:\Fontes5\Matricula\

Log da análise | Informações do projeto | Métricas | Detalhes

Classe	DTI	NOC	CM	QAC	RFC	MTM	%COM	LCOM	CBO	QMI	QMC	SOB	IE	QMH	QMA	WMC
TfrmPrincipal	0	0	8	13	8	4.63	16.22	92.31	8	8	0	0	0.00	0	8	16
TDisciplina	0	0	4	3	4	6.00	16.67	41.67	0	4	0	0	0.00	0	4	8
TCurso	0	0	4	3	4	11.00	47.73	41.67	0	4	0	0	0.00	0	4	8
TfrmCurso	0	0	4	11	4	8.75	0.00	86.36	7	4	0	0	0.00	0	4	11
TfrmDisciplina	0	0	4	11	4	10.25	0.00	86.36	7	4	0	0	0.00	0	4	12
TMatricula	0	0	5	6	5	10.60	5.66	46.67	2	5	0	0	0.00	0	5	13
TDiscMatri	0	0	3	1	3	4.00	16.67	33.33	0	3	0	0	0.00	0	3	6
TfrmMatricula	0	0	5	16	5	10.60	0.00	78.75	9	5	0	0	0.00	0	5	14
TAluno	0	2	4	4	4	4.00	0.00	75.00	1	4	0	0	0.00	0	4	8
TGraduacao	1	0	4	1	8	7.25	13.79	50.00	2	4	0	3	1.50	4	1	8
TPosGraduacao	1	0	4	2	8	7.25	0.00	50.00	2	4	0	3	1.50	4	1	8
TfrmAluno	0	0	5	23	5	9.60	0.00	83.48	9	5	0	0	0.00	0	5	14
TVetorCurso	0	0	5	2	5	14.40	31.94	10.00	2	5	0	0	0.00	0	5	14
TVetorDisciplina	0	0	5	2	5	14.40	31.94	10.00	2	5	0	0	0.00	0	5	14
TVetorMatricula	0	0	6	2	6	11.67	24.29	8.33	3	6	0	0	0.00	0	6	16
TVetorAluno	0	0	6	2	6	10.50	7.94	8.33	3	6	0	0	0.00	0	6	17
Tfrmsobre	0	0	1	8	1	3.00	0.00	100.00	4	1	0	0	0.00	0	1	2

Figura 29 – MÉTRICAS DA CLASSE TALUNO

Ferramenta para cálculo de métricas para softwares orientados a objetos

Projeto Métricas Help Sair

Nome do Projeto: Scm.dpr

Caminho: C:\Fontes5\Matricula\

Log da análise | Informações do projeto | Métricas | Detalhes

Classe: TAluno

Métrica	Valor
Profundidade da árvore de herança (DTI)	0
Número de filhos (NOC)	2
Contagem de métodos (CM)	4
Quantidade de atributos (QAC)	4
Resposta da classe (RFC)	4
Média do tamanho dos métodos (MTM)	4.00
Percentual comentado (%COM)	0.00
Falta de coesão (LCOM)	75.00
Acoplamento entre objetos (CBO)	1
Quantidade de métodos de instância (QMI)	4
Quantidade de métodos de classe (QMC)	0
Quantidade de métodos sobrescritos (SOB)	0
Indice de especialização (IE)	0.00
Quantidade de métodos herdados (QMH)	0
Quantidade de métodos adicionados (QMA)	4
Métodos ponderados (WMC)	8

Figura 30 – MÉTRICAS DA CLASSE TGRADUACAO

Ferramenta para cálculo de métricas para softwares orientados a objetos

Projeto Métricas Help Sair

Nome do Projeto: Scm.dpr

Caminho: C:\Fontes5\Matricula\

Log da análise | Informações do projeto | Métricas | Detalhes

Classe: TGraduacao

Métrica	Valor
Profundidade da árvore de herança (DTI)	1
Número de filhos (NOC)	0
Contagem de métodos (CM)	4
Quantidade de atributos (QAC)	1
Resposta da classe (RFC)	8
Média do tamanho dos métodos (MTM)	7.25
Percentual comentado (%COM)	13.79
Falta de coesão (LCOM)	50.00
Acoplamento entre objetos (CBO)	2
Quantidade de métodos de instância (QMI)	4
Quantidade de métodos de classe (QMC)	0
Quantidade de métodos sobrescritos (SOB)	3
Indice de especialização (IE)	1.50
Quantidade de métodos herdados (QMH)	4
Quantidade de métodos adicionados (QMA)	1
Métodos ponderados (WMC)	8

Figura 31 – MÉTRICAS DA CLASSE TCURSO

Ferramenta para cálculo de métricas para softwares orientados a objetos

Projeto Métricas Help Sair

Nome do Projeto: Scm.dpr

Caminho: C:\Fontes5\Matricula\

Log da análise | Informações do projeto | Métricas | Detalhes

Classe: TCurso

Métrica	Valor
Profundidade da árvore de herança (DTI)	0
Número de filhos (NOC)	0
Contagem de métodos (CM)	4
Quantidade de atributos (QAC)	3
Resposta da classe (RFC)	4
Média do tamanho dos métodos (MTM)	11.00
Percentual comentado (%COM)	47.73
Falta de coesão (LCOM)	41.67
Acoplamento entre objetos (CBO)	0
Quantidade de métodos de instância (QMI)	4
Quantidade de métodos de classe (QMC)	0
Quantidade de métodos sobrescritos (SOB)	0
Indice de especialização (IE)	0.00
Quantidade de métodos herdados (QMH)	0
Quantidade de métodos adicionados (QMA)	4
Métodos ponderados (WMC)	8

4.4.4 RESULTADOS OBTIDOS

Após o cálculo das métricas pode-se avaliar os resultados obtidos na figura 27 e chegar a algumas conclusões sobre a qualidade do projeto analisado.

Apesar de serem apresentadas no diagrama de classes (figura xx) sete classes, durante a análise foram encontradas dezessete. Esta diferença é porque o analisador extrai todas as definições de classes do código fonte, incluindo as classes de interface e de dados que não estão na especificação do projeto.

As classes de interface como por exemplo TfrmPrincipal e TfrmAluno, possuem uma quantidade maior de atributos. O acoplamento nestas classes também é maior, pois são utilizados diversos componentes. Notou-se também que a há falta de coesão é alta para estas classes, devido a estas classes possuírem muitos atributos e os métodos não irão tratar a maioria deles. A quantidade de métodos nestas classes também é maior, pois as classes de interface são geralmente aplicações especificas.

Já nas classes que estavam descritas no diagrama de classes, por exemplo Tcurso, Talunoe e Tdisciplina, pode-se notar que a quantidade e atributos e métodos é menor. O acoplamento nestas classes também é menor.

Nas classes TvetorCurso e TvetorAluno, que são classes de dados, a coesão entre métodos e atributos mostrou-se alta. A classe Taluno apresentou um percentual de falta de coesão alto (75%) o que pode indicar problemas na construção desta classe.

As métricas de construção podem ser utilizadas para melhorar a qualidade do código, um exemplo de utilização seria verificar se as classes estão sendo bem comentadas. Os comentários auxiliam na manutenção, entendimento e reutilização. A maioria das classes analisadas neste estudo de caso possui comentários, entretanto a classe Taluno não possui comentário nenhum o que, provavelmente irá dificultar sua manutenção no futuro.

Ao observar os resultados da classe tfrmSobre, pode-se notar que a falta de coesão chegou a 100% isto é devido ao fato de nenhum atributo da classe ter sido referenciado pelos métodos.

5 CONCLUSÕES

O objetivo principal deste trabalho, que foi desenvolver um protótipo de software para calcular algumas métricas para softwares orientados a objeto a partir da análise do código fonte, foi atendido.

Para possibilitar o cálculo das métricas, é necessário que sejam extraídas do código fonte as definições das classes e de seus métodos e atributos. A extração destas informações foi realizada baseando-se nos diagramas de sintaxe da linguagem *Object Pascal*. Estes diagramas contêm a especificação formal das construções desta linguagem.

A Orientação a Objetos provê muitos benefícios como reutilização, decomposição do problema em objetos de fácil entendimento e ajuda a futuras modificações, entre outros. Devido a estes fatores a Orientação a Objetos surge como uma possibilidade para melhoria da qualidade e produtividade do software.

Porém, não basta, apenas, identificar que atributos determinam a qualidade do software, mas também que procedimentos adotar para controlar seu processo de desenvolvimento, de forma a atingir o nível de qualidade desejado. Esse processo é realizado através da aplicação de métricas de qualidade, que são medidas ou avaliações das características de qualidade do produto. O uso de medidas, de uma forma organizada e projetada, possui efeito benéfico, tornando os desenvolvedores mais conscientizados da relevância do gerenciamento e dos compromissos para com a qualidade.

O protótipo desenvolvido pode auxiliar os gerentes e desenvolvedores a avaliar a qualidade e produtividade de seu software, através da análise das métricas calculadas pelo mesmo. Também pode auxiliar no desenho de diagramas de classes para projetos já implementados que não possuem documentação, uma vez que o protótipo disponibiliza todas as informações das classes implementadas no projeto analisado.

O protótipo desenvolvido pode ser utilizado em disciplinas engenharia de software, para fins de ensino de métricas de software orientados a objetos.

Sobre as limitações existentes no protótipo pode-se destacar:

a) o protótipo só poderá ser utilizado para a análise de projetos codificados no ambiente de desenvolvimento Delphi, devido às particularidades da linguagem hospedeira;
b) serão analisadas apenas as units especificadas no arquivo de extensão “dpr” selecionado.

5.1 EXTENSÕES

Para extensões deste trabalho sugere-se:

a) inclusão de gráficos com comparativos entre as métricas obtidas e as entradas esperadas;
b) disponibilizar um maior número de métricas no protótipo implementado.

ANEXO 1 – CÓDIGO FONTE ANALISADO

Neste anexo é apresentado o código fonte de algumas das units do projeto utilizado como estudo de caso.

Quadro 8 – CÓDIGO FONTE UNIT UPRINCIPAL

```
unit UPrincipal;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Menus, C_Curso, C_VetorCurso, C_Disciplina, C_VetorDisciplina,
  C_VetorMatricula, C_Matricula, StdCtrls, C_VetorAluno;

type
  TfrmPrincipal = class(TForm)
    MMMenu: TMainMenu;
    MArquivo: TMenuItem;
    MCadastro: TMenuItem;
    SMCurso: TMenuItem;
    SMDisciplina: TMenuItem;
    SMSair: TMenuItem;
    MMatricula: TMenuItem;
    MMAluno: TMenuItem;
    MMSobre: TMenuItem;
    procedure SMSairClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure SMCursoClick(Sender: TObject);
    procedure SMDisciplinaClick(Sender: TObject);
    procedure MMatriculaClick(Sender: TObject);
    procedure CancelarMatrcula1Click(Sender: TObject);
    procedure MMAlunoClick(Sender: TObject);
    procedure MMSobreClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    //Declaraçao de todas as classes utilizadas no sistema
    Curso : TVetorCurso; //Vetor da classe de curso
    Disciplina : TVetorDisciplina; //Vetor da classe de disciplinas
    Matricula  : TVetorMatricula; //vetor da classe matrícula
    Aluno : TVetorAluno; //vetor da classe aluno
  end;

var
  frmPrincipal: TfrmPrincipal;

implementation

uses UCurso, UDisciplina, UMatricula, UCancelarMatricula, UAluno, USobre;

{$R *.DFM}

procedure TfrmPrincipal.SMSairClick(Sender: TObject);
begin
  //**********************************************
  // Libera todos os objetos criados para o sistema
  //**********************************************
        Curso.LiberaCursoVetor;
        Disciplina.LiberaDisciplinaVetor;
        Matricula.LiberaMatriculaVetor;
        Aluno.LiberaVetor;
        Application.Terminate;
```

```
        end;

        procedure TfrmPrincipal.FormCreate(Sender: TObject);
        begin
          //*****************************************************
          // Cria os objetos que serão utilizados em todos o sistema
          //*****************************************************
                Curso := TVetorCurso.CriaCursoVetor;
                Disciplina := TVetorDisciplina.CriaDisciplinaVetor;
                Matricula := TVetorMatricula.CriaMatriculaVetor;
                Aluno := TVetoraluno.CriaAluno;
        end;

        procedure TfrmPrincipal.SMCursoClick(Sender: TObject);
        begin
                frmCurso.ShowModal;
        end;

        procedure TfrmPrincipal.SMDisciplinaClick(Sender: TObject);
        begin
                frmDisciplina.showModal;
        end;

        procedure TfrmPrincipal.MMatriculaClick(Sender: TObject);
        begin
                frmMatricula.ShowModal
        end;

        procedure TfrmPrincipal.CancelarMatrcula1Click(Sender: TObject);
        begin
                frmCancelarMatricula.showModal;
        end;

        procedure TfrmPrincipal.MMAlunoClick(Sender: TObject);
        begin
                frmAluno.showmodal;
        end;

        procedure TfrmPrincipal.MMSobreClick(Sender: TObject);
        begin
                frmsobre.ShowModal
        end;

end.
```

Quadro 9 – CÓDIGO FONTE UNIT C_CURSO

```
        unit C_Curso;

        interface
        Uses classes;
        Type
                TCurso = class
                Protected
                        cdCurso : String;
                        nmCurso : String;
                        tpCurso : String;
                Public
                        Constructor CriaCurso (PcdCurso, PnmCurso, PtpCurso: String);
                        Destructor  LiberaCurso;
                        Function ConsultarCurso : String;
                        Function VerificaCurso(PCurso : String): boolean;
        end;

        implementation

        Constructor TCurso.CriaCurso ;
        //*************************************************************
        //*          Construção da Classe Curso                       *
        //*************************************************************
```

```
	begin
		inherited Create;
		cdCurso:= PcdCurso;
		nmCurso:= PnmCurso;
		tpCurso:= PtpCurso;
	end;

	Destructor TCurso.LiberaCurso;
	//***********************************************************
	//*           Liberação da Classe Curso                     *
	//***********************************************************
	begin
		inherited Destroy;
	end;

	//***********************************************************
	//*           Busca todos os dados de todos os curso        *
	//***********************************************************
	//* Retorno:                                                *
	//*   Branco: não encontrou o registro                      *
	//*   Com Valor: retorna todos os dados do registro         *
	//***************************** ****************************
	Function TCurso.ConsultarCurso : String;
	begin
		ConsultarCurso := ('       CURSOS       ' + chr(13)+
		                   'Código..: ' + cdCurso + chr(13)+
		                   'Nome ...: ' + nmCurso + chr(13)+
		                   'Tipo ...: ' + tpCurso);

	end;

	Function TCurso.VerificaCurso(PCurso : String): boolean;
	//***********************************************************
	//*  Busca  o dado do objeto conforme o parametro informado  *
	//***********************************************************
	//* Parametros                                              *
	//* PCurso:  código do curso a ser pesquisado               *
	//* Retorno:                                                *
	//*   False: não encontrou o registro                       *
	//*   True : retorna todos os dados do registro             *
	//***************************** ***************************
	begin
		VerificaCurso := False;
		if cdCurso = PCurso then
			VerificaCurso := True;
	end;
end.
```

Quadro 10 – CÓDIGO FONTE UNIT C_DISCIPLINA

```
	unit C_Disciplina;

	interface
	Uses Classes;

	type
		TDisciplina = class
		private
			cdDisciplina : string;
			dsDisciplina : string;
			cdCurso      : string;
		public
			Constructor CriaDisciplina (PcdDisciplina, PdsDisciplina, PcdCurso:
String);
			Destructor  LiberaDisciplina;
			Function ConsultarDisciplina : String;
			Function VerificaDisciplina (PcdDisciplina : string) : Boolean;
	end;

	implementation
```

```
       Constructor TDisciplina.CriaDisciplina (PcdDisciplina, PdsDisciplina, PcdCurso:
String);
       //*          Construção da Classe Disciplina                  *
       begin
               inherited Create;
               cdDisciplina:= PcdDisciplina;
               dsDisciplina:= PdsDisciplina;
               cdCurso:= PcdCurso;
       end;

       Destructor TDisciplina.LiberaDisciplina;
       //*           Liberação da Classe Disciplina                  *
       begin
               inherited Destroy;
       end;

       Function TDisciplina.ConsultarDisciplina : String;
       //*         Busca todos os dados e todas as disciplinas       *
       begin
               ConsultarDisciplina := ('          DISCIPLINAS              ' + chr(13) +
                                       'Código ..: ' + cdDisciplina + chr(13)+
                                       'Nome ....: ' + dsDisciplina + chr(13)+
                                       'Curso ...: ' + cdCurso);
       end;

       Function TDisciplina.VerificaDisciplina (PcdDisciplina : string) : Boolean;
       //*  Busca  o dado do objeto conforme o parametro informado   *
       begin
               VerificaDisciplina := False;
               if cdDisciplina = PcdDisciplina then
                       VerificaDisciplina := True;
       end;
end.
```

Quadro 11 – CÓDIGO FONTE UNIT C_ALUNO

```
       unit C_Aluno;

       interface
       uses classes;

       Type

               TAluno = class
               Protected
                       cdAluno   : String;
                       nmAluno   : String;
                       cdCurso   : String;
                       dtNascto  : TDateTime;
               Public
                       Constructor CriaAluno (PcdAluno, PnmAluno, PcdCurso: String; PdtNascto:
TDateTime);
                       Destructor  Libera; virtual;
                       Function ConsultarAluno : String; virtual;
                       Function VerificaAluno(PcdAluno: string): boolean; virtual;
       end;

       implementation
       //*************************************************************
       //*          Construção da Classe Aluno                       *
       //*************************************************************
       Constructor TAluno.CriaAluno (PcdAluno, PnmAluno, PcdCurso: String; PdtNascto:
TDateTime);
       begin
               inherited Create;
               cdAluno  := PcdAluno;
               nmAluno  := PnmAluno;
               cdCurso  := PcdCurso;
               dtNascto := PdtNascto;
```

```
        end;

        //*********************************************************
        //*           Liberação da Classe Aluno                   *
        //*********************************************************
        Destructor TAluno.Libera;
        begin
        end;

        //*********************************************************
        //*              Mostra todos os Alunos                   *
        //*********************************************************
        Function TAluno.ConsultarAluno : String;
        begin
        end;

        //*********************************************************
        //*              Verifica todos os Alunos                 *
        //*********************************************************
        Function TAluno.VerificaAluno(PcdAluno: string): boolean;
        begin

        end;
end.
```

Quadro 12 – CÓDIGO FONTE UNIT C_GRADUAÇÃO

```
        unit C_Graduacao;

        interface
        uses C_Aluno, SysUtils, classes;
        Type
                TGraduacao = class(TAluno)
                Protected
                        dtVestibular : TDateTime;
                Public
                        Constructor CriaGraduacao (PcdAluno, PnmAluno, PcdCurso: String;
PdtNascto, PdtVestibular: TDateTime);
                        Destructor  Libera;override;
                        Function ConsultarAluno : String; override;
                        Function VerificaAluno(PcdAluno: string): boolean; override;
        end;

        implementation

        Constructor TGraduacao.CriaGraduacao (PcdAluno, PnmAluno, PcdCurso: String; PdtNascto,
PdtVestibular: TDateTime);
        //*          Construção da Classe Aluno Graduação             *
        begin
                inherited  CriaAluno (PcdAluno, PnmAluno, PcdCurso,PdtNascto);
                dtVestibular := PdtVestibular;
        end;

        Destructor  TGraduacao.Libera;
        //*           Liberação da Classe Aluno Graduação             *
        begin
                inherited destroy;
        end;

        Function TGraduacao.ConsultarAluno : String;
        //          Retorna os dados do aluno graduação conforme      *
        //          indicado no vetor de Aluno do form principal      *
        Var
                AuxBuscar : String;

        begin
                AuxBuscar  :=  ('     GRADUAÇÃO     ' + CHR(13)+
                               'Código..........: ' + cdAluno + chr(13)+
                               'Nome ...........: ' + nmAluno + chr(13)+
                               'Curso...........: ' + cdCurso + chr(13)+
```

```
                              'Data Nascimento..: ' + DateToStr(dtNascto) + chr(13)+
                              'Data Vestibular .: ' + DateToStr(dtVestibular)) ;

              ConsultarAluno := AuxBuscar;
       end;

       Function TGraduacao.VerificaAluno(PcdAluno: string): boolean;
       begin
               VerificaAluno := false;
               If cdAluno = PcdAluno then
                       VerificaAluno := true;
       end;

end.
```

Quadro 13 – CÓDIGO FONTE UNIT C_POSGRADUAÇÃO

```
       unit C_PosGraduacao;

       Interface
       uses C_Aluno, SysUtils, classes;
       Type
               TPosGraduacao = class(TAluno)
               Protected
                       dtAnoFormacao   : String;
                       dsCursoFormacao : string;

               Public
                       Constructor CriaPosGraduacao (PcdAluno, PnmAluno, PcdCurso,
PdtAnoFormacao, PdsCursoFormacao: String; PdtNascto: TDateTime);
                       Destructor  Libera; override;
                       Function ConsultarAluno : String; override;
                       Function VerificaAluno(PcdAluno: string): boolean; override;
       end;

       implementation

       Constructor TPosGraduacao.CriaPosGraduacao(PcdAluno, PnmAluno, PcdCurso,
PdtAnoFormacao, PdsCursoFormacao: String; PdtNascto: TDateTime);
       //*       Construção da Classe Aluno Pós_Graduação            *
       begin
               inherited  CriaAluno (PcdAluno, PnmAluno, PcdCurso,PdtNascto);
               dtAnoFormacao   := PdtAnoFormacao;
               dsCursoFormacao := PdsCursoFormacao;
       end;
       Destructor  TPosGraduacao.Libera;
       //*           Liberação da Classe Aluno Pós_Graduação         *
       begin
               inherited destroy;
       end;

       Function TPosGraduacao.ConsultarAluno : String;
       //*                     Mostra os Alunos                      *
       //          Retorna os dados do aluno graduação conforme      *
       //          indicado no vetor de Aluno do form principal      *
       Var
               AuxBuscar : String;

       begin
               AuxBuscar  :=  ('        PÓS-GRADUAÇÃO        ' + chr(13) +
                              'Código...........: ' + cdAluno + chr(13)+
                              'Nome .............: ' + nmAluno + chr(13)+
                              'Curso.............: ' + cdCurso + chr(13)+
                              'Data Nascimento....:' + DateToStr(dtNascto) + chr(13)+
                              'Curso de Formação .:' + (dsCursoFormacao) + chr(13)+
                              'Ano de Formação ...:' + (dtAnoFormacao)) ;

              ConsultarAluno := AuxBuscar;

              end;
```

```
       Function TPosGraduacao.VerificaAluno(PcdAluno: string): boolean;
       begin
               VerificaAluno := false;
               If cdAluno = PcdAluno then
                       VerificaAluno := true;

       end;

end.
```

Quadro 14 – CÓDIGO FONTE UNIT C_MATRICULA

```
       unit C_Matricula;

       interface
       Uses classes, SysUtils, C_DiscMatri;

       Const
               TamVetor = 20;
       Type
                TMatricula = class
                       cdMatricula : String;
                       cdAluno     : String;
                       dsSemestre  : String;
                       dtMatricula : TDateTime;
                       IntDisc     : Integer;
                       DiscMatri   : Array[1..TamVetor] of TDiscMatri;

               Public
                       Constructor CriaMatricula (PcdMatricula, PcdAluno,PdsSemestre :String ;
PdtMatricula : TDateTime);
                       Destructor  LiberaMatricula;
                       Function CadastrarDiscMatri (PcdDisciplina : String): Boolean;
                       Function ConsultarMatricula : String;
                       Function VerificarMatricula (PcdMatricula : String) : Boolean;
       end;

       implementation

       Constructor TMatricula.CriaMatricula (PcdMatricula, PcdAluno,PdsSemestre :String ;
PdtMatricula : TDateTime);
       //*          Construção da Classe Matricula                    *
       begin
               inherited Create;
               cdMatricula:= PcdMatricula;
               cdAluno:= PcdAluno;
               dsSemestre:= PdsSemestre;
               dtMatricula:= PdtMatricula;
               IntDisc := 0;
       end;

       Destructor TMatricula.LiberaMatricula;
       //*           Liberação da Classe Matricula                    *
       Var
               IntAux    : Byte;
       begin
               for IntAux := 1 to IntDisc do
                       DiscMatri[intAux].LiberaDiscMatri;

               inherited Destroy;
       end;

       Function TMatricula.CadastrarDiscMatri (PcdDisciplina : String): Boolean;
       //*           Busca todos os dados de todas as matriculas      *
       begin
               If IntDisc <= TamVetor then
               begin
                       inc(IntDisc);
                       DiscMatri[IntDisc]:= TDiscMatri.CriaDiscMatri(PcdDisciplina);
```

```
                CadastrarDiscMatri := True;
        end else
                CadastrarDiscMatri := false;

    end;

    Function TMatricula.ConsultarMatricula : String;
    Var
    //*               Consultar Matricula                        *

        strAux : string;
        intAux : integer;
    begin
        StrAux := '         MATRÍCULA        ' + chr(13) +
                  'Código ....:'+ cdMatricula  + chr(13) +
                  'Nome Aluno :'+ cdAluno      + chr(13) +
                  'Semestre ..:'+ dsSemestre   + chr(13) +
                  'Data ......:'+ dateToStr(dtMatricula)  + chr(13) +
                  '         DISCIPLINAS      ' + CHR(13) ;

        for IntAux:= 1 to IntDisc do
                StrAux:= StrAux + DiscMatri[intAux].ConsultaDiscMatri;

        ConsultarMatricula := strAux;
    end;

    Function TMatricula.VerificarMatricula(PcdMatricula : String) : Boolean;
    begin

        VerificarMatricula := False;
        if cdMatricula = PcdMatricula then
                VerificarMatricula := True;

    end;
end.
```

Quadro 15 – CÓDIGO FONTE UNIT C_DISMATRI

```
    unit C_DiscMatri;

    interface
    Uses classes;

    Type
        TDiscMatri = class
        Private
                cdDisciplina: string;
        Public
                Constructor CriaDiscMatri(PcdDisciplina: String);
                Destructor  LiberaDiscMatri;
                Function ConsultaDiscMatri : string;

    end;

    implementation

    Constructor TDiscMatri.CriaDiscMatri(PcdDisciplina: String);
    //*          Construção da Classe DiscMatri                   *
    begin
        inherited Create;
        cdDisciplina := PcdDisciplina;
    end;
    Destructor TDiscMatri.LiberaDiscMatri;
    //*           Liberação da Classe DiscMatri                   *
    begin
        inherited Destroy;
    end;
    Function TDiscMatri.ConsultaDiscMatri : string;
    begin
        ConsultaDiscMatri := 'Disciplina : ' + cdDisciplina + chr(13)
```

```
        end;
end.
```

Quadro 16 – CÓDIGO FONTE UNIT UALUNO

```
unit UAluno;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, UPrincipal, C_Graduacao, C_PosGraduacao, Mask;

type
  TfrmAluno = class(TForm)
    GroupBox1: TGroupBox;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    edCodigo: TEdit;
    edCurso: TEdit;
    edNome: TEdit;
    GroupBox2: TGroupBox;
    rbGraduacao: TRadioButton;
    rbPosGraduacao: TRadioButton;
    btCadastrar: TButton;
    Button2: TButton;
    gbGraduacao: TGroupBox;
    Label5: TLabel;
    gbPosGraduacao: TGroupBox;
    Label6: TLabel;
    edAno: TEdit;
    Label7: TLabel;
    edFormacao: TEdit;
    btConsultar: TButton;
    edNascto: TMaskEdit;
    edVestibular: TMaskEdit;
    procedure rbGraduacaoClick(Sender: TObject);
    procedure rbPosGraduacaoClick(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure btCadastrarClick(Sender: TObject);
    procedure btConsultarClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmAluno: TfrmAluno;

implementation

{$R *.DFM}

procedure TfrmAluno.rbGraduacaoClick(Sender: TObject);
begin
        gbGraduacao.Visible := True;
        gbPosGraduacao.Visible := False;
end;

procedure TfrmAluno.rbPosGraduacaoClick(Sender: TObject);
begin
        gbGraduacao.Visible := False;
        gbPosGraduacao.Visible := True;
end;
procedure TfrmAluno.Button2Click(Sender: TObject);
begin
        edCodigo.text:= '';
```

```
              edNome.text:= '';
              edCurso.text:= '';
              edNascto.Text:= '';
              edVestibular.Text:='';
              edAno.Text:='';
              edFormacao.Text:='';
              frmAluno.Close;
       end;

       procedure TfrmAluno.btCadastrarClick(Sender: TObject);
       begin
              if not frmPrincipal.Aluno.VerificaAlunoVetor(edCodigo.text) then
              begin
                     if frmPrincipal.Curso.VerificaCursoVetor(edCurso.Text) then
                     begin
                            if rbGraduacao.Checked then
                                   frmPrincipal.Aluno.CriaAlunoGraduacao
(edCodigo.text,edNome.text,
                                                                              edCurso.text,
StrToDate(edNascto.Text),

StrToDate(edVestibular.Text))
                            else
                                   frmPrincipal.Aluno.CriaAlunoPosGraduacao
(edCodigo.text,edNome.text, edCurso.text,

edAno.Text,edFormacao.Text,StrToDate(edNascto.Text));
                            MessageDlg('Aluno incluido com sucesso. ',
mtInformation,[mbOk], 0);
                     end else
                            MessageDlg('Curso informado não cadastrado. ',
mtInformation,[mbOk], 0);
              end else
                     MessageDlg('Aluno informado já cadastrado. ', mtInformation,[mbOk], 0);

       end;

       procedure TfrmAluno.btConsultarClick(Sender: TObject);
       Var
              ListaAluno : TStringList;
              IntCont    : integer;
       begin
              ListaAluno := TStringList.Create;
              ListaAluno.Clear;
              ListaAluno := frmprincipal.Aluno.ConsultarAlunoVetor;
              For IntCont := 1 to ListaAluno.Count do
                     MessageDlg(ListaAluno.Strings[IntCont-1], mtInformation,[mbOk], 0);

       end;
end.
```

Quadro 17 – CÓDIGO FONTE UNIT UDISCIPLINA

```
       unit UDisciplina;

       interface

       uses
         Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
         StdCtrls, UPrincipal;

       type
         TfrmDisciplina = class(TForm)
           GroupBox2: TGroupBox;
           Label1: TLabel;
           Label2: TLabel;
           Label3: TLabel;
           edNome: TEdit;
           edCodigo: TEdit;
           edCurso: TEdit;
```

```
          GroupBox1: TGroupBox;
          btSair: TButton;
          btConsulta: TButton;
          btCadastra: TButton;
          procedure btSairClick(Sender: TObject);
          procedure btCadastraClick(Sender: TObject);
          procedure btConsultaClick(Sender: TObject);
          procedure FormCreate(Sender: TObject);
        private
          { Private declarations }
        public
          { Public declarations }
        end;

      var
        frmDisciplina: TfrmDisciplina;

      implementation

      {$R *.DFM}

      procedure TfrmDisciplina.btSairClick(Sender: TObject);
      begin
              frmDisciplina.Close;
      end;
      procedure TfrmDisciplina.btCadastraClick(Sender: TObject);
      begin
              if frmPrincipal.Curso.VerificaCursoVetor(edCurso.Text) then
              begin
                      if not frmPrincipal.Disciplina.VerificaDisciplinaVetor (edCodigo.Text)
then
                      begin
                              If
frmPrincipal.Disciplina.CadastrarDisciplinaVetor(edCodigo.Text,ednome.Text, edcurso.Text )
then
                                      MessageDlg('Disciplina incluida com sucesso.',
mtInformation,[mbOk], 0)
                              else
                                      MessageDlg('Quantidade de disciplinas esgotada.',
mtInformation,[mbOk], 0);

                              edCodigo.Text := '';
                              edNome.Text := '';
                              edCurso.Text := '';
                      end else
                              MessageDlg('Código informado já encontra-se cadastrado.',
mtInformation,[mbOk], 0);

             end else
                      MessageDlg('O curso informado não está cadastrado.' ,
mtInformation,[mbOk], 0);
      end;

      procedure TfrmDisciplina.btConsultaClick(Sender: TObject);
      var
              ListaDisciplina : TStringList;
              IntCont         : integer;
      begin
              ListaDisciplina := TStringList.Create;
              ListaDisciplina.Clear;
              ListaDisciplina := frmPrincipal.Disciplina.ConsultarDisciplinaVetor;
              For IntCont := 1 to ListaDisciplina.Count do
                      MessageDlg(ListaDisciplina.Strings[IntCont-1], mtInformation,[mbOk],
0);
      end;
      procedure TfrmDisciplina.FormCreate(Sender: TObject);
      begin
              edCodigo.Text := '';
              edNome.Text := '';
              edCurso.Text := '';
      end;
end.
```

Quadro 18 – CÓDIGO FONTE UNIT UCURSO

```
unit UCurso;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, UPrincipal;

type
  TfrmCurso = class(TForm)
    GroupBox1: TGroupBox;
    GroupBox2: TGroupBox;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    edNome: TEdit;
    edCodigo: TEdit;
    edTipo: TEdit;
    btSair: TButton;
    btConsulta: TButton;
    btCadastra: TButton;
    procedure btSairClick(Sender: TObject);
    procedure btCadastraClick(Sender: TObject);
    procedure btConsultaClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmCurso: TfrmCurso;

implementation

{$R *.DFM}

procedure TfrmCurso.btSairClick(Sender: TObject);
begin
        frmCurso.Close;
end;

procedure TfrmCurso.btCadastraClick(Sender: TObject);
begin
        if not frmprincipal.Curso.VerificaCursoVetor(edCodigo.Text) then
        begin
                if frmprincipal.Curso.CadastrarCursoVetor
(edCodigo.Text,edNome.Text,edTipo.Text) then
                         MessageDlg('Curso incluido com sucesso.',
mtInformation,[mbOk], 0)
                else
                         MessageDlg('Quantidade de cursos esgotada.',
mtInformation,[mbOk], 0);

                edCodigo.Text := '';
                edNome.Text := '';
                edTipo.Text := '';
        end else
                MessageDlg('O código informado já encontra-se cadastrado.',
mtInformation,[mbOk], 0);

end;

procedure TfrmCurso.btConsultaClick(Sender: TObject);
Var
        ListaCurso : TStringList;
        IntCont    : integer;
begin
        ListaCurso := TStringList.Create;
```

```
                ListaCurso.Clear;
                ListaCurso := frmprincipal.Curso.ConsultarCursoVetor;
                For IntCont := 1 to ListaCurso.Count do
                        MessageDlg(ListaCurso.Strings[IntCont-1], mtInformation,[mbOk], 0);

        end;

        procedure TfrmCurso.FormCreate(Sender: TObject);
        begin
                edCodigo.Text := '';
                edNome.Text := '';
                edTipo.Text := '';
        end;

end.
```

Quadro 19 – CÓDIGO FONTE UNIT C_VETORDISCIPLINA

```
        unit C_VetorDisciplina;

        interface
        Uses Classes, C_Disciplina;

        const
                TamVetor = 50; //Tamanho do vetor de disciplinas
        type

                TVetorDisciplina = class
                private
                        Disciplina : array [1..TamVetor] of TDisciplina;
                        IntCont    : byte;
                public
                        Constructor CriaDisciplinaVetor;
                        Destructor  LiberaDisciplinaVetor;
                        Function CadastrarDisciplinaVetor (PcdDisciplina, PdsDisciplina,
PcdCurso: String): Boolean;
                        Function ConsultarDisciplinaVetor : TStringlist;
                        Function VerificaDisciplinaVetor (PcdDisciplina : string) : Boolean;
        end;

        implementation

        Constructor TVetorDisciplina.CriaDisciplinaVetor;
        //*          Construção da Classe Disciplina                  *
        begin
                inherited Create;
                IntCont:= 0;
        end;

        Destructor TVetorDisciplina.LiberaDisciplinaVetor;
        //*           Liberação da Classe Disciplina                  *
        Var
                IntAux    : Byte;
        begin
                for IntAux := 1 to IntCont do
                        Disciplina[IntAux].LiberaDisciplina;

                inherited Destroy;
        end;

        Function TVetorDisciplina.CadastrarDisciplinaVetor (PcdDisciplina, PdsDisciplina,
PcdCurso: String): Boolean;
        //*             Função  para Cadastrar Disciplina             *
        //*********************************************************
        //* Parametros:                                               *
        //*   PcdDisciplina : código da disciplina                    *
        //*   PdsDisciplina : descrição da disciplina                 *
        //*   PcdCurso : código do curso que a disciplina pertence    *
        //* Retorno:                                                  *
        //*   True:  registro cadastrado                              *
```

```
//*   Falso: registro não cadastrado                           *
begin
        if IntCont < TamVetor then
        begin
                inc(IntCont);
                Disciplina[IntCont] := TDisciplina.CriaDisciplina (PcdDisciplina,
PdsDisciplina, PcdCurso);
                CadastrarDisciplinaVetor:= True;
        end else begin
                CadastrarDisciplinaVetor:= false;
        end;
end;

Function TVetorDisciplina.ConsultarDisciplinaVetor : TStringlist;
//*         Busca todos os dados e todas as disciplinas        *
//**************************************************************
//* Retorno:                                                   *
//*   Branco: não encontrou o registro                         *
//*   Com Valor: retorna todos os dados do registro            *
Var
        IntAux    : Byte;
        AuxBuscar : TStringList;
begin

        AuxBuscar := TStringList.Create;
        AuxBuscar.Clear;
        for IntAux := 1 to IntCont do
                AuxBuscar.Add (Disciplina[IntAux].ConsultarDisciplina );

        ConsultarDisciplinaVetor := AuxBuscar;
end;

Function TVetorDisciplina.VerificaDisciplinaVetor (PcdDisciplina : string) : Boolean;
//*  Busca  o dado do objeto conforme o parametro informado    *
//**************************************************************
//* Parametros                                                 *
//* PCurso:  código do curso a ser pesquisado                  *
//* Retorno:                                                   *
//*   False: não encontrou o registro                          *
//*   True : retorna todos os dados do registro                *
var
        IntAux : integer;
        IsAchou : Boolean;
begin
        IntAux := 1;
        IsAchou := false;
        while (IntAux <= IntCont) and (not IsAchou) do
        begin
                if Disciplina[IntAux].VerificaDisciplina(PcdDisciplina) then
                        IsAchou := True;

                inc(IntAux);
        end;
        VerificaDisciplinaVetor := IsAchou;
end;

end.
```

Quadro 20 – CÓDIGO FONTE UNIT C_VETORALUNO

```
nit C_VetorAluno;

interface
uses classes, C_Aluno, C_PosGraduacao, C_Graduacao;

const
        TamVetor = 30;
Type

        TVetorAluno = class
```

```
                Protected
                        Aluno : array[1..TamVetor] of TAluno;
                        intAluno : integer;
                Public
                        Constructor CriaAluno;
                        Destructor  LiberaVetor;
                        Function ConsultarAlunoVetor : TStringList;
                        Function CriaAlunoPosGraduacao (PcdAluno, PnmAluno, PcdCurso,
PdtAnoFormacao, PdsCursoFormacao: String; PdtNascto: TDateTime): boolean;
                        Function CriaAlunoGraduacao (PcdAluno, PnmAluno, PcdCurso: String;
PdtNascto, PdtVestibular: TDateTime): Boolean;
                        Function VerificaAlunoVetor (PcdAluno: string): boolean;
        end;

        implementation

        Constructor TVetorAluno.CriaAluno ;
        //*          Construção da Classe Aluno                        *
        begin
                inherited Create;
                intAluno := 0;
        end;

        Destructor TVetorAluno.LiberaVetor;
        //*           Liberação da Classe Aluno                        *
        var
                intAux : integer;
        begin
                for IntAux := 1 to IntAluno do
                        Aluno[intaux].Libera;
                inherited Destroy;
        end;

        Function TVetorAluno.CriaAlunoPosGraduacao (PcdAluno, PnmAluno, PcdCurso,
PdtAnoFormacao, PdsCursoFormacao: String; PdtNascto: TDateTime): boolean;
        //*           Cria a classe pos-graduação                      *
        begin
                if intAluno <= TamVetor then
                begin
                        inc(intAluno);
                        Aluno[intAluno]:= TPosGraduacao.CriaPosGraduacao(PcdAluno, PnmAluno,
PcdCurso, PdtAnoFormacao, PdsCursoFormacao,PdtNascto);
                        CriaAlunoPosGraduacao := True;
                end else
                        CriaAlunoPosGraduacao := False;

        end;

        Function TVetorAluno.CriaAlunoGraduacao (PcdAluno, PnmAluno, PcdCurso: String;
PdtNascto, PdtVestibular: TDateTime): Boolean;
        //*           Cria a classe     Graduação                      *
        begin
                if intAluno <= TamVetor then
                begin
                        inc(intAluno);
                        Aluno[intAluno]:= TGraduacao.CriaGraduacao (PcdAluno, PnmAluno,
PcdCurso,PdtNascto, PdtVestibular);
                        CriaAlunoGraduacao := True;
                end else
                        CriaAlunoGraduacao := False;

        end;

        Function TVetorAluno.ConsultarAlunoVetor : TStringList;
        //*              Mostra todos os Alunos                        *
        Var
                IntAux    : Byte;
                AuxBuscar : TStringList;
        begin

                AuxBuscar := TStringList.Create;
                AuxBuscar.Clear;
```

```
                for IntAux := 1 to IntAluno do
                        AuxBuscar.Add (Aluno[IntAux].ConsultarAluno);

                ConsultarAlunoVetor := AuxBuscar;
        end;

        Function TVetorAluno.VerificaAlunoVetor (PcdAluno: string): boolean;
        var
                IntAux : integer;
                IsAchou : Boolean;
        begin
                IntAux := 1;
                IsAchou := false;
                while (IntAux <= IntAluno) and (not IsAchou) do
                begin
                        if Aluno[IntAux].VerificaAluno(PcdAluno) then
                                IsAchou := True;

                        inc(IntAux);
                end;
                VerificaAlunoVetor := IsAchou;
        end;

end.
```

Quadro 21 – CÓDIGO FONTE UNIT C_VETORCURSO

```
        unit C_VetorCurso;

        interface
        Uses classes, C_Curso;
        Const
                TamVetor = 10;  //Tamanho do vetor de cursos

        Type
                TVetorCurso = class
                Private
                        Curso   : array[1..TamVetor] of TCurso; //Vetor de cursos
                        IntCont : byte; //Indice do vetor de cursos
                Public
                        Constructor CriaCursoVetor;
                        Destructor  LiberaCursoVetor;
                        Function CadastrarCursoVetor (PcdCurso, PnmCurso, PtpCurso: String):
Boolean;
                        Function ConsultarCursoVetor : TStringList;
                        Function VerificaCursoVetor(PCurso : String): boolean;
        end;

        implementation

        Constructor TVetorCurso.CriaCursoVetor ;
        //*          Construção da Classe Curso                      *
        begin
                inherited Create;
                IntCont:= 0;
        end;

        Destructor TVetorCurso.LiberaCursoVetor;
        //*           Liberação da Classe Curso                      *
        var
                IntAux : integer;
        begin
                for IntAux := 1 to IntCont do
                        Curso[IntAux].LiberaCurso;

                inherited Destroy;
        end;

        Function TVetorCurso.CadastrarCursoVetor (PcdCurso, PnmCurso, PtpCurso: String):
Boolean;
```

```
//*              Função para Cadastrar o Curso                *
//* Parametros:                                               *
//*   PcdCurso : código do curso                              *
//*   PnmCurso : nome do curso                                *
//*   PtpCurso : tipo do curso                                *
//* Retorno:                                                  *
//*   True:  registro cadastrado                              *
//*   Falso: registro não cadastrado                          *
//*************************************************************
begin
        if IntCont < TamVetor then
        begin
                inc(IntCont);
                Curso[IntCont]:= TCurso.CriaCurso(PcdCurso, PnmCurso, PtpCurso);
                CadastrarCursoVetor:= True;
        end else begin
                CadastrarCursoVetor:= false;
        end;
end;

Function TVetorCurso.ConsultarCursoVetor : TStringList;
//*           Busca todos os dados de todos os curso          *
//*************************************************************
//* Retorno:                                                  *
//*   Branco: não encontrou o registro                        *
//*   Com Valor: retorna todos os dados do registro           *
Var
        IntAux    : Byte;
        AuxBuscar : TStringList;
begin

        AuxBuscar := TStringList.Create;
        AuxBuscar.Clear;
        for IntAux := 1 to IntCont do
                AuxBuscar.Add (Curso[IntAux].ConsultarCurso );

        ConsultarCursoVetor := AuxBuscar;
end;

Function TVetorCurso.VerificaCursoVetor(PCurso : String): boolean;
//*  Busca  o dado do objeto conforme o parametro informado   *
//*************************************************************
//* Parametros                                                *
//* PCurso:  código do curso a ser pesquisado                 *
//* Retorno:                                                  *
//*   False: não encontrou o registro                         *
//*   True : retorna todos os dados do registro               *
var
        IntAux : integer;
        IsAchou : Boolean;
begin
        IntAux := 1;
        IsAchou := false;
        while (IntAux <= IntCont) and (not IsAchou) do
        begin
                if Curso[IntAux].VerificaCurso(PCurso) then
                        IsAchou := True;

                inc(IntAux);
        end;
        VerificaCursoVetor := IsAchou;
end;

end.
```

Quadro 22 – CÓDIGO FONTE UNIT C_VETORMATRICULA

```
       unit C_VetorMatricula;

       interface
       Uses classes, SysUtils, C_Matricula;

       Const
               TamVetor = 10;  //Tamanho do vetor de matricula

       Type

               TVetorMatricula = class
               Private
                       Matricula : array[1..TamVetor] of TMatricula; //Vetor de matricula
                       IntCont   : byte; //Indice do vetor de matriculas
               Public
                       Constructor CriaMatriculaVetor;
                       Destructor  LiberaMatriculaVetor;
                       Function EfetuarMatriculaVetor (PcdMatricula, PcdAluno,PdsSemestre
:String ; PdtMatricula : TDateTime): Boolean;
                       Function CadastrarDiscMatriVetor (PcdDisciplina : String): Boolean;
                       Function ConsultarMatriculaVetor : TStringList;
                       Function VerificarMatriculaVetor(PcdMatricula : String): boolean;

       end;

       implementation

       Constructor TVetorMatricula.CriaMatriculaVetor ;
       //*          Construção da Classe Matricula                   *
       begin
               inherited Create;
               IntCont:= 0;
       end;

       Destructor TVetorMatricula.LiberaMatriculaVetor;
       //*           Liberação da Classe Matricula                   *
       Var
               IntAux    : Byte;
       begin
               for IntAux := 1 to IntCont do
                       Matricula[IntAux].LiberaMatricula;

               inherited Destroy;
       end;

       Function TVetorMatricula.EfetuarMatriculaVetor (PcdMatricula, PcdAluno,PdsSemestre :
String; PdtMatricula : TDateTime): Boolean;
       //*              Função para Efetuar a Matricula              *
       //************************************************************
       //* Parametros:                                               *
       //*   PcdMatricula: código da matricula                       *
       //*   PcdAluno : código do aluno                              *
       //*   PdsSemestre : descrição do semestre                     *
       //*   PflCancelada: flag para dizer se a mat. foi cancelada   *
       //*   PdtMatricula : data de efetivação da matricula          *
       //* Retorno:                                                  *
       //*   True:  registro efetuado                                *
       //*   Falso: registro não efetuado                            *
       begin
               if IntCont <= TamVetor then
               begin
                       inc(IntCont);
                       Matricula[IntCont]:= TMatricula.CriaMatricula (PcdMatricula,
PcdAluno,PdsSemestre, PdtMatricula );
                       EfetuarMatriculaVetor:= True;
               end else begin
                       EfetuarMatriculaVetor:= false;
               end;
       end;
```

```
       Function TVetorMatricula.CadastrarDiscMatriVetor (PcdDisciplina : String): Boolean;
       //*         Casdastra os dados das disciplinas referente a matrícula   *
       //* Retorno:  Branco: não encontrou o registro                         *
       //*   Com Valor: retorna todos os dados do registro                    *
       begin
               CadastrarDiscMatriVetor := Matricula[IntCont].CadastrarDiscMatri
(PcdDisciplina);

       end;

       Function TVetorMatricula.ConsultarMatriculaVetor : TStringList;
       //*         Consulta os dados de matrícula                             *
       Var
               IntAux       : Integer;
               AuxBuscar : TStringList;
       begin

               AuxBuscar := TStringList.Create;
               AuxBuscar.Clear;
               for IntAux := 1 to IntCont do
                       AuxBuscar.Add (Matricula[IntAux].ConsultarMatricula);

               ConsultarMatriculaVetor := AuxBuscar;
       end;


       //*******************************************************************
       //*         Verifica os dados de matrícula                           *
       //*******************************************************************
       Function TVetorMatricula.VerificarMatriculaVetor(PcdMatricula : String): boolean;
       var
               IntAux : integer;
               IsAchou : Boolean;
       begin
               IntAux := 1;
               IsAchou := false;
               while (IntAux <= IntCont) and (not IsAchou) do
               begin
                       if Matricula[IntAux].VerificarMatricula(PcdMatricula) then
                               IsAchou := True;

                       inc(IntAux);
               end;
               VerificarMatriculaVetor := IsAchou;
       end;
end.
```

REFERÊNCIAS BIBLIOGRÁFICAS

ANDRADE, Renata; DIAS, Márcio de Sousa; TRAVASSOS, Guilherme H. **Diretrizes para redução da complexidade do software orientado a objetos.** Rio de Janeiro, 1997. Disponível em:< http://www.ics.uci.edu/~mdias/public/cits97/cits97.html >. Acesso em: 10 set. 2001.

ARIGOFLU, Ali. A methodology for cost estimation. **ACM - Software engineering notes.** São Paulo, p. 96-105, vol 18, 1993.

ARTHUR, Lowell Jay. **Melhorando a qualidade do software**: um guia para o TQM. Rio de Janeiro: Infobook, 1994.

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

BORLAND INTERNATIONAL INC. **Object pascal language guide**. Scotts Valley: Borland, 1997.

CANTU, Marco **Dominando o Delphi 5,** a bíblia**.** São Paulo: Makron Books, 2000.

CARDOSO, Eduardo José. **Métricas para programação orientada a objetos**. 1999. 45 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

CHIDAMBER, Shyam R; KEREMER, Chris F. A Metrics suite for object oriented desing. **IEEE – Transactions on software enginieering.** São Paulo, p. 476-493, vol. 20, 1994.

CORDEIRO, Marco Aurélio. **Métricas de software**. Curitiba, 2000. Disponível em: <http://www.pr.gov.br/celepar/batebyte/edições/2000/bb101/métricass.htm>. Acesso em: 01 set. 2001.

DEMARCO, Tom**. Controle de projetos de software:** gerenciamento, avaliação, estimativa. Rio de Janeiro: Campus, 1989.

FERNANDES, Aguinaldo Aragon. **Gerência de software através de métricas**: garantindo a qualidade do projeto, processo e produto. São Paulo: Atlas, 1995.

FUCK, Mônica Andrea. **Estudo e aplicação das métricas da qualidade do processo de desenvolvimento de aplicações em banco de dados**. 1995. 104 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

JACOBSON, Ivar et al. **Object oriented software engineering**: a use case driven approach. Wokingham: Addison Wesley, 1992.

LORENZ, Mark; KIDD, Jeff. **Object-Oriented software metrics.** EnglewoodCliffsm NJ: Prentice-Hall, 1994.

MOLLER, K. H, PAULISH, D. J. **Software metrics:** a practitionerïs guide to improved product development. Los Alamitos: IEEE, 1993. 257p.

MARTIN, James. **Princípios de análise e projetos baseados em objetos**. Tradução Cristina Bazan. Rio de Janeiro: Campus, 1994.

POSSAMAI, Roque César. **Ferramenta de análise de estruturas básicas em linguagem Pascal para o fornecimento de métricas**. 2000. 71 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

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

ROCHA, Ana Regina Cavalcanti da; MALDONADO, José Carlos; WEBER, Kival Chaves. **Qualidade de software:** teoria e prática. São Paulo: Prentice Hall, 2001.

ROSENBERG, Linda. **Applying and interpreting object oriented metrics.** Utah, abr. 1998. Disponível em: <http://satc.gsfc.nasa.gov/support/STC_APR98/apply_oo apply_oo.html>. Acesso em: 20 ago. 2001.

SHEPPERD, Martin. **Foundations of software measurement.** New York: Prentice Hall, 1995.