UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

MÉTRICAS PARA PROGRAMAÇÃO ORIENTADA 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

EDUARDO JOSÉ CARDOSO

BLUMENAU, DEZEMBRO/1999

1999/2-11

MÉTRICAS PARA PROGRAMAÇÃO ORIENTADA A OBJETOS

EDUARDO JOSÉ CARDOSO

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

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. Everaldo Artur Grahl — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. Dalton Solano dos Reis

_____

Prof. Everaldo Artur Grahl

_____

Prof. Marcel Hugo

DEDICATÓRIA

No decorrer deste curso sempre estive acompanhado de pessoas muito especiais que me apoiaram e incentivaram-me a chegar até aqui. Este trabalho é dedicado a minha irmã ao meu pai Bento José Cardoso que me deu todo apoio e condições para estar na faculdade, e principalmente para minha mãe Ivanildes Klock Cardoso, que me apoiou e incentivou em todos os momentos de minha vida, mas que por escolha de Deus, partiu desta vida no decorrer do curso, restando a mim somente muitas saudades e boas lembranças da pessoa que mais amei na vida.

AGRADECIMENTOS

Agradeço a todos que de uma maneira ou de outra contribuíram para que chegasse até aqui, dando apoio e força para que continuasse esta trajetória até o fim, apesar das dificuldades que tive durante o curso.

Em especial agradeço a toda a minha família e amigos, que sempre estiveram ao meu lado no decorrer da minha vida.

A todos os professores que contribuíram para minha formação. Ao meu orientador Everaldo Artur Grahl pelo apoio e incentivo dado neste semestre para a realização deste trabalho.

Por último, agradeço a todas as pessoas que direta ou indiretamente cruzaram o meu caminho, tornando a minha vida digna de ser vivida, e dizer Valeu a Pena!!!

Obrigado a todos.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

RESUMO

Este trabalho descreve a construção de um protótipo para aplicação de métricas em sistemas orientados a objeto. Para tanto foi desenvolvido um software em ambiente Delphi para facilitar o cálculo de métricas como contagem de métodos, métodos por classe, classes sucessoras, classes ascendentes e classes descendentes específicas para a orientação a objeto.

ABSTRACT

This work describes the construction of a prototype for application of metrics in object oriented sytems. For so much a software was developed in ambient Delphi to facilitate the calculation of metric as count of methods, methods for class, successors class, ascending class and specific descending class for the object orientation.

1 INTRODUÇÃO

Com o processo de desenvolvimento de software adotado a partir de 1970 os projetos de software tornaram-se grandes e mais complexos, sendo necessário o controle do processo de forma emergente. O processo incluiu a definição do ciclo de vida do software pela sequência das fases, e mais ênfase no gerenciamento e controle de recursos deste projeto, visto em [MOL94].

A metodologia de objetos vem oferecer uma solução alternativa para o desenvolvimento de sistemas, e significa uma grande evolução desde a programação estruturada, apesar da metodologia de orientação a objetos também utilizar os mesmos princípios da programação estruturada (abstração, hierarquização, decomposição) e acrescentar novos princípios, como classe, objeto e herança.

Atualmente com a grande expansão no desenvolvimento de software orientado a objeto criou-se a necessidade de estimar os projetos orientados a objeto através das seguintes métricas visto em [AMB98]:

a) número de classes reutilizáveis;

b) porcentagem de classes;

c) contagem de métodos;

d) contagem de atributos de instância.

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.

1.1 OBJETIVO

O objetivo principal do trabalho é a especificação e implementação de um software que permita calcular métricas específicas para Programação Orientada a Objeto (POO). Este software irá analisar o código fonte de programas Orientado a Objeto (OO) em Delphi e fornecerá os resultados de cálculos de algumas métricas estudadas.

1.2 ORGANIZAÇÃO DO TEXTO

O primeiro capítulo trata da origem e objetivo do trabalho.

No segundo capítulo encontram-se as definições e conceitos de métrica.

No terceiro capítulo encontra-se a definição e metodologia de orientação a objetos.

No quarto capítulo são mostradas as definições de métricas OO.

No quinto capítulo encontra-se a definição das ferramentas, que foram utilizadas para a definição e implementação deste trabalho e a especificação do protótipo e sua implementação.

No sexto capítulo está a apresentação do protótipo.

No sétimo capítulo encontra-se a conclusão deste trabalho.

2 DEFINIÇÃO E CONCEITOS DE MÉTRICAS

Segundo [FER95], as métricas podem ser definidas como métodos de determinar, quantitativamente, a extensão em que o processo e o produto de software têm certos atributos. Isto inclui uma fórmula para determinar o valor da métrica como também sua forma de apresentação e as diretrizes de utilização e interpretação de resultados obtidos no contexto do ambiente de desenvolvimento de software.

Um dos aspectos que deve ser observado quando das iniciativas de utilização de métricas é quanto à sua utilidade no contexto de um projeto ou do ambiente como um todo, além, naturalmente, 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.

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 [MOL94], as quais podem ser enquadradas atualmente:

a) Medida da 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, no número de linhas de código necessário para a implementação;

c) Garantia da Qualidade do Software: estas técnicas foram melhoradas significativamente entre os anos de 1970 e 1980. Neste caso, se dá ênfase à identificação de informações faltantes, durante as várias 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 sequência das fases, e mais ênfase no gerenciamento e controle de recursos deste projeto.

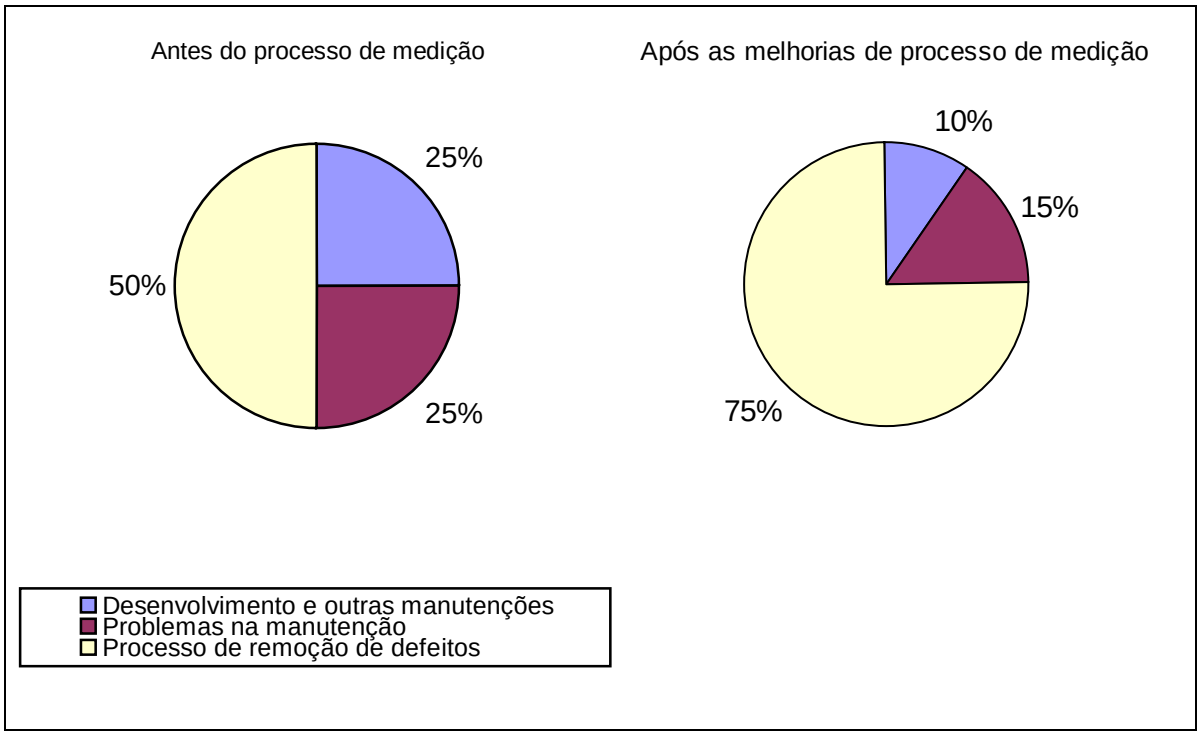
Os desenvolvedores de sistemas, a partir do aparecimento destas tendências, começaram então a usar as métricas no propósito de melhorar o processo de desenvolvimento de software.

2.2 QUALIDADE E PRODUTIVIDADE DE SOFTWARE ATRAVÉS DE MÉTODOS QUANTITATIVOS

Segundo [FER95], para que a qualidade possa ser implementada em uma empresa é necessário planejá-la antecipadamente. Isto consiste no estabelecimento de metas da qualidade, definições de estratégias, execução das tarefas como foram previstas no planejamento, como também coletar informações para a análise do andamento do processo.

Desse modo, segundo [FER95], o objetivo primário da realização de medições no tocante ao desenvolvimento de software é obter níveis cada vez melhores da qualidade, visando sempre à satisfação plena dos clientes ou usuários, a um custo economicamente compatível. Portanto, segundo [GOL95], as métricas apresentam resultados satisfatórios em relação à qualidade que podem ser expressados graficamente como através da Figura 1, visto que antes do processo de medição como mostra o gráfico 25% do tempo eram gastos com desenvolvimento e outras manutenções e 25% com problemas de manutenção. Após as melhorias do processo de medição reduziu para 10% o tempo de desenvolvimento e manutenção 15% o de problemas de manutenção, obtendo nestes casos uma redução média de 50% do tempo em relação ao processo sem melhorias de medição.

Figura 1 – O motivo das medições.



Fonte: [GOL95].

A qualidade é o alvo da realização de medições em um software. Para tanto é necessário compreender o significado deste termo, a fim de identificar o objeto da melhoria no desenvolvimento de um projeto. A qualidade é um tema pelo qual se pode comentar vários aspectos, porque tudo gira em torno da melhoria do software visando qualidade. A conseqüência da qualidade no que se faz é o aumento da produtividade, que seria o precursor de um custo cada vez menor no trabalho realizado.

Segundo [FER95], em uma organização todos devem ser educados (permanentemente) para a melhoria contínua de todos os processos de desenvolvimento de sistemas na empresa, visando à redução ou até mesmo à eliminação das causas de problemas que possam ocasionar defeitos e o não cumprimento de prazos e estimativas iniciais, em relação aos produtos e serviços. A empresa deve mostrar que tem capacidade para controlar seu processo, pois a partir disto é que o cliente aceita o produto com maior satisfação.

2.3 OBJETIVOS COM A UTILIZAÇÃO DE MÉTRICAS

Segundo [FER95], 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 estas atividades. Alguns 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 produto;
j) avaliar o impacto de atributos no processo de desenvolvimento com novas ferramentas;
k) determinar tendências relativas a certos atributos do processo.

Segundo [CAP95], as razões para se medir um software são as seguintes:

- indicar a qualidade do produto;
- determinar a produtividade da equipe;
- formar uma base de informações para as estimativas;
- ajudar a justificar a necessidade de novas ferramentas ou de treinamento.

Para a utilização dos modelos de estimativas, segundo [FER95], é necessário adaptá-los conforme a realidade da organização. Esta realidade constitui-se em uma série histórica com as observações acerca de prazos, esforço, custo, tamanho do software e outros atributos referentes a cada projeto.

A utilidade das métricas deve ser traçada desde o início da implantação de métricas para avaliação de software. Elas devem ser simples de entender e bem objetivas no seu emprego, facilitando os processos de tomada de decisão. Seu objetivo também pressupõe a minimização do julgamento pessoal na coleta das informações, sendo que o valor da

informação obtido como resultado das medições, deve exceder o custo de coletar, armazenar e calcular as métricas. Outro objetivo importante é que as métricas selecionadas devem propiciar informações que possibilitem avaliar acertos ou não de decisões tomadas no passado, como também de possíveis decisões no futuro, evitando assim a surpresa das conseqüências destas decisões.

Segundo [BAS89], 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.

Segundo [FUC95], a questão mais primária no âmbito das métricas é o estabelecimento das metas da medição. Portanto esta meta engloba os seguintes requisitos:

- um objeto de interesse o qual seria referente ao produto ou o processo;
- um propósito como o entendimento, caracterização ou melhoria;
- uma perspectiva que identifica quem é o interessado pelos resultados, por exemplo, o gerente, os desenvolvedores de software ou cliente;
- uma descrição do ambiente para fornecer um contexto próprio para qualquer resultado.

As metas costumam ser vagas e ambíguas, geralmente expressas em um nível impreciso de abstração. Por exemplo, as palavras entender, avaliar, qualidade, vantagens e desvantagens têm significados diferentes para pessoas diferentes ou variam de acordo com o ambiente. A necessidade de entender melhor a utilização dos recursos para melhorar o processo de estimativa de custos explica o que se quer fazer, mas deixa dúvidas a respeito dos tipos de dados que devem ser coletados. A necessidade de avaliar o uso de uma tecnologia, como as inspeções de projeto, requer a perspectiva do que se espera da metodologia, da mesma forma como acontece com um projeto de pesquisa. As metas devem ser bem

desenvolvidas, mas também refinadas quantitativamente para dar precisão e esclarecer seu significado com relação a um determinado ambiente.

Segundo [YOU95], se as pessoas envolvidas no desenvolvimento de um sistema estiverem unidas em adquirir novos conhecimentos para melhoria do processo do sistema, o conceito de métricas de software será percebido pelo pessoal técnico como um fenômeno positivo; porém, deve ficar claro desde o começo que o objetivo do esforço de métrica não é punir as pessoas que cometem erros, mas sim melhorar o processo. Desse modo, quando um gerente de projetos utiliza indevidamente uma métrica de software para reprimir alguém que apresentou um desempenho inferior, todas as pessoas da equipe serão atingidas, e como conseqüência, poderão não colaborar mais com o objetivo dos trabalhos.

2.4 IMPORTÂNCIA DA MEDIÇÃO DO SOFTWARE

Segundo [FER95] a realidade aponta para a necessidade de medições, visto que:

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

Desse modo, 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 tinha sido concluído.

Portanto a medição do software tem a importância de fornecer aos responsáveis pelo seu desenvolvimento, as informações que permitem ao gerente planejar o seu projeto de forma adequada controlando todo o trabalho com maior exatidão, tornando seu conceito em relação ao sistema mais seguro e confiável do ponto de vista do cliente.

Segundo [YOU95], para um engenheiro de software individual pode haver pouca motivação para se entregar a uma metrificação de software. Se esta pessoa considera-se um bom profissional, por que ele deveria se preocupar com quantas linhas de código ele escreveu em um certo dia? No entanto, está no âmago do argumento de se captar métricas de software: o desejo de melhorar. Se todas as pessoas estivessem basicamente satisfeitas com aquilo que estivessem fazendo, não haveria a necessidade de medir e nenhuma necessidade de acompanhar o curso das coisas.

Antigamente as pessoas não estavam satisfeitas com o software que tinham em seu poder, no entanto, elas não se importavam com este aspecto. Apenas os desenvolvedores é que estavam satisfeitos com o seu desempenho no trabalho. Portanto eles não se preocupavam com medições. Hoje a situação se modificou muito. As empresas de software necessitam ser as melhores sempre, caso contrário elas não permanecem no negócio.

Portanto, ainda segundo [YOU95], Capers Jones argumenta que a produtividade é uma função do tamanho do projeto e a distribuição de esforço em atividades comuns relacionadas a software varia consideravelmente como uma função do tamanho do projeto. Esta afirmação é verdadeira se uma organização possuir um programa de metrificação que proporcione tais informações.

Para compreender melhor a utilização deste programa de metrificação em relação a distribuição de esforço x tamanho de um projeto, tem-se ainda os seguintes motivos:

- as métricas são necessárias para validar e ajustar modelos “genéricos” de produtividade de software, confiabilidade e assim por diante;
- investimentos em ferramentas de produtividade, técnicas, metodologias e outros não podem ser justificados sem boas métricas.

Segundo [ARI93], as técnicas de avaliação de custos concentram-se fundamentalmente em dois aspectos nos produtos para software: um é o tamanho que é o fator primário da estimativa de custo de um software, sendo este, um trabalho que requer conhecimento específico das funções do sistema em termos de extensão, complexidade e número de interações. Outros são o trabalho requerido e o tempo para a execução dos trabalhos. Alguns exemplos seriam a quantidade de código escrito e testado, o número de páginas de especificações funcionais, ou o número de diagramas na fase de modelagem do sistema. Na aplicação, não há relação entre as várias técnicas que tem sido estabelecidas e a maioria das técnicas são aplicadas individualmente. O objetivo é identificar como diferentes métodos podem ser dispostos em ordem metodológica para melhor estimar o custo do software.

Dependendo da situação e do entendimento da organização em relação ao assunto métricas, elas podem tornar-se desnecessárias em várias fases do desenvolvimento do software, pois o desenvolvedor muitas vezes estima certas métricas fundamentais incorretamente, neutralizando todas as outras avaliações, tendo aquela em significado maior ao sistema. Compreendendo como vários métodos de estimativa de custos de software se identificam, consegue-se avaliar com mais eficiência as determinantes de um software.

Segundo [FUC95], algumas diretrizes básicas para se implantar um sistema de métricas em uma instituição:

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

ficar desmotivadas sem compreender a utilidade do trabalho empregado para a coleta de dados;

h) publicar casos de sucesso na utilização de métricas e encorajar o intercâmbio de idéias. É importante comunicar a noção de que as métricas de software não são passivas, mas, ao contrário, uma ferramenta para a ação. Dessa forma as métricas podem ser usadas para ajudar a identificar as coisas que estão erradas num projeto antes que um problema sério se desenvolva; em geral, métricas podem ser usadas para que se possa ver como várias partes do processo de desenvolvimento de software podem ser melhoradas.

2.5 PROBLEMAS NA IMPLANTAÇÃO DE MÉTRICAS

Segundo [MOL94], as empresas de software podem não aceitar a utilização dos conceitos de métricas. Algumas razões são expostas abaixo:

a) as métricas podem restringir a criatividade no processo: as métricas podem ser vistas como um tempo perdido para avaliar processos. Esta idéia deve ser superada através de exemplo de métricas e experiência. As métricas são necessárias para ajudar a modificar o desenvolvimento de software, fazendo com que o processo seja continuamente melhorado;

b) as métricas foram criadas para dar mais trabalho: manter os registros de dados métricos causam um trabalho adicional. Por este motivo deve-se planejar, economizando assim tempo e dinheiro sempre levando em consideração todo o ciclo de vida de desenvolvimento de um software;

c) os benefícios não estão claros: quando são introduzidas métricas em um projeto, pode ocorrer pouca concordância entre os membros da equipe. Isto ocorre porque não há explicação necessária para a introdução deste método. Os benefícios dos métodos quantitativos devem ser claramente explicados para toda a organização;

2.6 COMO SÃO AS MÉTRICAS ATUAIS

Hoje as métricas se caracterizam em indicadores globais e trazem àquele que as utiliza uma melhor visibilidade e controle do complexo processo de desenvolvimento de software. Os melhores exemplos da utilização de métricas são representados pelas empresas que já tiram proveito desta técnica. Isto traz ao software uma melhor competitividade no mercado além de contribuir na melhoria da qualidade e produtividade do software. Verifica-se assim que segundo [MOL94], as empresas americanas que utilizam métodos quantitativos para melhoria do processo de desenvolvimento de software, reduzem seus custos aproximadamente 25% ao ano e o tempo de desenvolvimento próximo a 10% ao ano.

Estes números estimulam os gerentes de sistemas a se aproximarem dos métodos quantitativos controlando o projeto de software através desta estratégia, e assim, conseguindo maiores lucros e aproveitamento do tempo de trabalho.

Segundo [YOU95], jamais deve-se estar completamente satisfeito com as métricas utilizadas. Algumas se mostrarão irrelevantes, e sempre haverá a necessidade de novas métricas. Dessa forma, o investimento em pessoas, ferramentas e outros recursos para a métrica de software deve ser considerado como um custo contínuo de se fazer negócios, exatamente como os negócios globais consideram os contadores que fazem a demonstração de mutações do patrimônio líquido, balancetes e o balanço patrimonial, como um custo contínuo de se fazer negócios.

3 ORIENTAÇÃO A OBJETOS

Conforme [WIN93], “a modelagem de informações, popularizado por Peter Chen nos anos 80, é o precursor mais próximo da análise e do projeto orientados ao objeto. O primeiro material publicado sobre análise orientada ao objeto foi de Sally Shlaer e Stephen Mellor (1988)”. A técnica, começa pela definição de objetos, classes e atributos. Diferente da decomposição funcional, a técnica pela definição de objetos, classes e atributos. Diferente da decomposição funcional, a técnica orientada ao objeto resulta em pouca codificação dirigida aos dados, os quais permanecem estáveis mesmo quando as exigências são alteradas.

Nos anos 90, os aplicativos deverão satisfazer requisitos mais sofisticados, utilizar estruturas e arquiteturas de dados mais complexas, e estar aptos a atender uma grande e crescente base de usuários. Todos estes fatores sugerem a necessidade de um grande salto na capacidade dos desenvolvedores de software para construir, expandir e manter sistemas complexos em larga escala. Além disso, para atender uma grande base de usuários, é preciso que estes software sejam mais flexíveis de usar.

A orientação a objeto apresenta um novo paradigma para a construção de software: Objetos e classes são os blocos de construção, enquanto métodos, mensagens e hereditariedade compões o mecanismo primário.

A orientação a objeto tornou-se a nova e determinante visão dos softwares dos anos 90, com a qual programadores e usuários finais são beneficiados por sua implementação. Apesar da mudança de desenho e programação envolvendo objetos parecer extrema, a orientação ao objeto é a maneira mais natural de desenho e modelagem de softwares. Após a completa implementação da nova metodologia, os usuários finais poderão expandir a sua habilidade em modificar e talvez até criar novos aplicativos. Os programadores estarão aptos a desenhar aplicativos mais complexos em porções modulares e intercambiáveis.

A análise e os projetos orientados a objeto, são continuidade da evolução da técnicas de análise estruturada. Esses métodos refletirão uma mudança na estratégia de análise, para incluir mecanismos fundamentais na elaboração de sistemas orientados a objeto. Num sistema orientado a objeto, a ênfase não é sobre a transformação de entradas em saídas, mas sobre o conteúdo das entidades, os objetos. O critério de agrupamento de funções não é um processo,

em vez disto, os métodos são agrupados se operarem na mesma abstração de dados. Métodos próximos em uma seqüência podem residir em objetos diferentes.

Segundo [WIN93], os métodos estruturados não são adequados para projetos de software orientados a objeto, como, de fato, não há ampla aceitação do método de projeto orientado a objeto. Talvez uma razão para isto é que o próprio paradigma da programação orientada a objeto contém importantes técnicas de projeto. A ênfase na reutilização, modularidade e encapsulação, tanto quanto a nítida ligação entre projeto e código, é parte essencial de sistemas orientados ao objeto. Pelo fato de uma grande parte dos sistemas orientados a objeto estar fincada no desenho de informação, os processos de projetos e implementação encaixam-se muito bem. A programação orientada a objeto leva o projeto de um programa a salientar-se por meio do código. Um bom projeto inicial pode ser considerado como uma versão anterior do código ou o código como uma versão anterior do projeto.

Além disso, a técnica de projetos orientados a objeto originam sistemas flexíveis para mudanças. Ela se concentra nos elementos mais estáveis dos sistema, os objetos. Esta estabilidade e flexibilidade estão presentes no próprio projeto. Quando uma mudança é necessária, a característica de hereditariedade permite a reutilização e extensão dos modelos existentes.

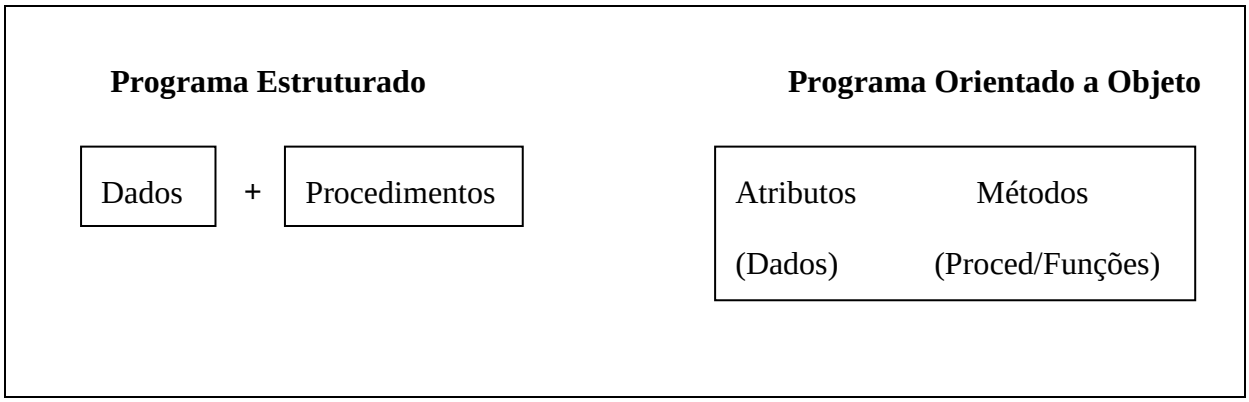
Essas técnicas orientadas a objeto são baseadas nesses mecanismos chamados de objetos, mensagens e métodos, classes e hereditariedade e para completar os mecanismos básicos, existem alguns conceitos chaves, como: encapsulamento, abstração, polimorfismo, persistência e reusabilidade. Todos os sistemas que são definidos como orientado a objeto contém esses mecanismos essenciais, ainda que não tenham sido implementados exatamente da mesma maneira.

3.1 OBJETOS

Segundo [CAL97], um programa tradicional consiste em procedimentos e dados. Um programa orientado a objeto consiste somente em objetos que contêm procedimentos e dados conforme ilustrado na Figura 2. Em outras palavras, objetos são módulos que contêm dados e instruções para manipular sobre estes dados. Dentro dos objetos, residem os dados das linguagens tradicionais, como números, matrizes, strings e registros, bem como funções,

instruções ou sub-rotinas que os operam. Portanto, os objetos são entidades que têm atributos específicos (dados) e maneiras de comportamento (procedimentos).

Figura 2 – Diferença de Procedimentos em Programa Estruturado e (POO).



Fonte: [CAL97].

Segundo [PAC93], os objetos são uma abstração de algo no domínio do problema, refletem a capacidade do sistema de guardar informações sobre o elemento abstraído, interagir com estes, ou ambos. Isto é confirmado por [MAR95], defini como, “um objeto é qualquer coisa, real ou abstrata, a respeito da qual armazenamos dados e os métodos que os manipulam”. No objeto estão contidos os dados (atributos) e os procedimentos (serviços) exclusivos ao próprio objeto, onde um objeto refere-se a uma ocorrência única.

A premissa básica de sistemas orientados a objeto é a combinação das estruturas de dados e procedimentos que manipulam os dados nestas estruturas, formando o chamado objeto. Acima de tudo, a combinação de dados e procedimentos permite que o princípio da modularidade seja usado na elaboração das aplicações. Este tipo de modularidade permite ao programador adicionar, remover e substituir partes do sistema, com pouca ou nenhuma conseqüência sobre as outras partes do programa ou aplicação. Em linguagens que suportam apenas programação procedural, as estruturas de dados são separadas dos procedimentos que manipulam os dados. Assim, qualquer procedimento ou declaração pode alterar qualquer dado no programa.

Programas tradicionais são compostos por suas partes básicas como mostra a figura 2, que são os procedimentos (operações a serem executadas) e os dados (informações

manipuladas pelos procedimentos). Desta forma as ações do programa são controladas por procedimentos que, quando executados, alimentam os dados.

Procedimentos e dados são criados globalmente, permitindo acesso ilimitado às suas operações. Dados e procedimentos são independentes entre si, porém eles dependem muito um do outro, já que os procedimentos devem fazer considerações sobre a forma como os dados serão manipulados.

A programação orientada a objeto é um método alternativo, que tem como característica principal a combinação (encapsulamento) de dados e procedimentos em uma única entidade (objeto). Com este controle da modularidade e da estrutura das aplicações, o programador tem total liberdade para adicionar, remover ou repor partes de um programa, gerando pouca ou nenhuma influência nas outras partes do mesmo.

Muitos programadores tentam usar um conjunto de regras que impeçam um procedimento de modificar dados globalmente; nada suporta este conceito tão naturalmente quanto os sistemas orientados a objeto. Os procedimentos que manipulam os dados são tipicamente subrotinas executados seqüencialmente, tal como na programação procedural. Mas a visão global de um programa orientado a objeto é baseado em objetos, diferentemente dos procedimentos individuais que apenas manipulam dados.

Segundo [PAC93], um sinônimo para o termo objeto, também encontrado na literatura, é instância de uma classe, ou simplesmente instância. Por exemplo, o objeto ‘Gol’ é uma instância da classe ‘Automóvel’.

3.2 CLASSE

Segundo [WIN93], classe é uma descrição de um conjunto de objetos quase idênticos. Uma classe consiste em métodos e dados que resumem as características comuns a um conjunto de objetos. A habilidade em abstrair descrições de métodos e dados comuns a um conjunto de objetos e armazená-los em uma classe é a centralização da capacidade da orientação a objeto. Definir classes significa posicionar codificação reutilizável em um depósito comum em vez de expressá-lo várias vezes. Em outras palavras, as classes contêm os moldes para a criação de objetos. Também, a definição de uma classe auxilia a esclarecer a definição de um objeto: um objeto é uma instância de uma classe.

As classes são usadas em sistemas de programação orientados a objeto, para permitir que objetos similares reutilizem estruturas de dados e comportamentos comuns. Onde [MAR95] define, “uma classe é uma implementação de um tipo de objeto. Ela especifica a estrutura de dados e os métodos operacionais que se aplicam a cada um de seus objetos”. Esta característica é uma poderosa arma de um sistema de programação orientada a objeto, ou seja, se existirem vários grupos de dados com estruturas similares que precisam ser manipulados de uma forma semelhante, a estrutura de dados e os procedimentos associados podem ser definidos como uma estrutura chamada classe. Então, sempre que for necessário usar um objeto parecido com outro já existente, o objeto apropriado pode ser criado baseado nesta classe. Fazendo uma analogia simples, a classe de um objeto pode ser comparada à estrutura de um registro (definição de arquivo) em um banco de dados. Uma classe descreve a estrutura ou o formato de um objeto, assim como uma definição de arquivo descreve o formato de cada registro em um dado arquivo de banco de dados.

Uma classe representa uma estrutura que contém dados, propriedades e operações (procedimentos e funções). Através do princípio de herança, novas classes que utilizam os dados, propriedades e operações de outras classes podem ser definidas. Ou seja, segundo [MAR95], “as classes de nível mais baixo herdam o comportamento de classes de nível mais elevado”.

Em outras palavras, um objeto necessita de todas as características de uma determinada classe, e também dados e procedimentos adicionais, ele pode ser definido baseado na classe apropriada e depois estendido com adição das estruturas de dados e procedimentos necessários. Alterar um objeto desta forma é na verdade o mesmo que criar uma nova classe. E, de fato, é desta forma que isto pode ser feito: novas classes podem ser definidas baseadas em classes existentes. Neste caso, a nova classe herda todas as características da classe usada como base, além de receber novas capacidades.

A hereditariedade é o mecanismo de compartilhamento automático de métodos e dados entre classes, subclasses e objetos, esse mecanismo poderoso não encontrado em sistemas procedurais (tradicionais). A hereditariedade permite aos programadores criar novas classes programando somente a diferença entre elas, ou seja, a possibilidade de se criar uma entidade baseada nas características das existentes. O comportamento da hereditariedade é confirmado por [MAR95], “uma classe herda as propriedades de classe-mãe; uma subsubclasse herda as

propriedades das subclasses e assim por diante. Uma subclasse pode herdar a estrutura de dados e os métodos, ou alguns dos métodos, de sua superclasse".

Definindo classes gerais básicas e depois criando novas classes com finalidades mais específicas, é possível criar uma hierarquia de classes, tal como uma árvore genealógica. A classe "raiz" tem uma estrutura mínima, e as classes nas pontas das ramificações possuem as estruturas mais complexas, servindo a propósitos específicos. O conceito de herança também faz com que, no momento em que uma classe ancestral é alterada, todas as classes descendentes também sejam automaticamente alteradas. Isto facilita a alteração de uma característica geral em um conjunto de classes, já que a alteração pode ser feita em uma única classe ancestral.

3.3 MENSAGENS

Segundo [CAL97], diferente dos dados passivos em sistemas tradicionais, os objetos possuem habilidade para agir. A ação ocorre quando um objeto recebe uma mensagem, isto é, uma solicitação para que se comporte de uma determinada maneira. Quando os programas orientados a objeto são executados, os objetos estão recebendo, interpretando e respondendo mensagens de objetos.

Afirma [PAC93], que em um programa orientado a objeto, nenhuma outra parte do programa pode acessar diretamente os dados no objeto. Toda a comunicação entre os objetos ocorre única e exclusivamente através de mensagens explícitas. Também, é através delas que os objetos integram em um programa, ou seja, elas simplesmente informam ao objeto o que este deve fazer.

Também, pode-se dizer que as mensagens, tratam-se de pedidos enviados a um objeto, a fim de que este modifique seu estado ou retorne algum valor, através de um método.

Segundo [WIN93], assim como as "caixas-preta" de engenharia, a estrutura interna de um objeto é escondido dos usuários e programadores. As mensagens que o objeto recebe são os únicos condutos que conectam o objeto ao mundo exterior.

3.4 MÉTODOS E ENCAPSULAMENTO

Os procedimentos chamados métodos residem em objetos e determinam como o objeto atuará quando receber uma mensagem. Ou seja, mencionado por [MAR95], “os métodos especificam a maneira pela qual os dados de um objeto são manipulados.”. Os métodos trabalham em respostas às mensagens e manipulam os valores das variáveis de instância. De fato, os métodos detêm o único mecanismo para alteração dos valores das variáveis de instância. Os métodos também podem enviar mensagens a outros objetos, requisitando ação ou informação.

Partindo de [PAC93], do ponto de vista da programação, em um programa orientado a objeto, os métodos estão implementados em funções ou procedimentos colocados ao nível do objeto ao qual se relacionam. Métodos são as operações efetuadas pelos objetos e representam um comportamento específico, residente no objeto, que define como este deve agir quando exigido.

O encapsulamento, é o tempo formal que descreve a junção de métodos e dados dentro de um objeto de maneira que o acesso aos dados seja permitido somente por meio dos próprios métodos do objeto. É afirmado por [MAR95], “o ato de empacotar ao mesmo tempo dados e métodos é denominado encapsulamento.”. Nenhuma outra parte do programa orientado a objeto pode operar diretamente em um dado do objeto. A comunicação entre um conjunto de objetos ocorre exclusivamente por meio de mensagens explícitas, explicado por [WIN93].

Segundo [PAC93], pode-se entender intuitivamente o significado de encapsulamento quando se observa o mundo real. Nos objetos do mundo real, atributo e ações são inerentes ao próprio objeto. O encapsulamento, também é a propriedade de se implementar dados e procedimentos correlacionados em uma mesma entidade (objeto). Trata-se de uma das principais vantagens da programação orientada ao objeto sobre a programação estruturada, principalmente na reutilização de códigos.

3.5 ABSTRAÇÃO

Segundo [CAL97], a orientação a objeto estimula os programadores e usuários a pensarem sobre aplicações em termos abstratos. Começando com um conjunto de objetos, os programadores são conduzidos a criar comportamentos comuns e encaixá-los em superclasses abstratas. Cada nível de abstração faz com que o trabalho de programação fique mais fácil porque existe maior disponibilidade de codificação reutilizável.

Para o sucesso de um programa orientado a objeto está condicionado a uma boa abstração do problema. Também, é o processo de retirar do domínio do problema os detalhes relevantes e representá-los não mais em uma linguagem do domínio e sim na linguagem da solução, afirma [PAC93].

3.6 POLIMORFISMO

Na explicação de [WIN93], os objetos agem em resposta às mensagens que recebem. A mesma mensagem pode resultar em ações completamente diferentes quando recebidas por objetos diferentes. Este fenômeno é conhecido como polimorfismo. Com o polimorfismo, um usuário pode enviar uma mensagem genérica e deixar os detalhes de implementação para o objeto receptor. O polimorfismo é estimulado pelo mecanismo de hereditariedade.

Uma das estratégias adotadas na programação orientada a objeto é a de implementar os métodos o mais alto possível na hierarquia de classes. As variações necessárias nos métodos são feitas à medida em que se desça a árvore hierárquica. Outra vantagem do polimorfismo é a relativa facilidade de manutenção e extensão dos programas.

3.7 PERSISTÊNCIA

Segundo [CAL97], refere-se à permanência de um objeto, isto é, ao tempo pelo qual ele aloca espaço e permanece acessível na memória do computador. Na maioria das linguagens orientadas a objeto, as instâncias de classes são criadas enquanto o programa é executado. Algumas dessas instâncias são necessárias somente por um pequeno período de tempo. Quando um objeto não é mais necessário, ele é destruído e o espaço de memória alocado é recuperado. A recuperação automática do espaço de memória é chammado comumente de “coleta de lixo”.

Depois que um programa orientado a objeto foi executado, os objetos construídos não ficam armazenados, isto é, eles não são mais persistentes. Os objetos armazenados permanentemente são tratados como persistentes. O termo persistência está relacionado ao total de tempo que um objeto permanece na memória.

Na maioria dos programas orientados a objeto, as instância das classes são criadas durante a execução do programa e assim que o objeto não é mais necessário, ocorre uma destruição do mesmo, ou seja, o espaço de memória por ele ocupado é liberado.

Um objeto em um programa orientado a objeto surge de uma das seguintes formas:

- o objeto é criado estática ou dinamicamente pelo programa;
- o objeto é recuperado de um arquivo ou de uma base de dados relacional e levado à forma de objeto;
- o objeto é recuperado de uma base de dados orientada a objeto e lido diretamente para utilização.

3.8 REUSABILIDADE

Hoje um dos principais pontos no desenvolvimento e construção de software é a produção com qualidade, e isto também é anunciado como uma das principais chaves do paradigma orientado a objeto. E como isto é conseguido? Pela reusabilidade de software, proporcionado pela orientação a objeto, isto é afirmado categoricamente por [BEL96], “a orientação a objetos proporciona maior produtividade através de reutilização.”.

Isto também é confirmado por [SCO96], “reutilização, tem o potencial de materializar uma aumento de produtividade e uma redução de custos.”.

Os desenvolvedores de sistemas têm atualmente, uma obrigação cada vez maior, que é a de realizar e finalizar os projetos com menor prazo e custo e com maior qualidade, por isso, a reutilização é uma das características mais importantes do paradigma orientado a objeto. Com certeza a filosofia da reutilização não é uma idéia nova, pois já era utilizada pelo sistema de projetos estruturados, como por exemplo, na forma de biblioteca, como é mencionado por [FER95], “reutilização: extensão em que um programa pode ser usado em outras aplicações; relacionando ao empacotamento e escopo das funções que o programa desempenha.”

Não é somente a reusabilidade do paradigma da orientação a objeto que dará os resultados e atenderá as expectativas de produção e qualidade dos usuários e desenvolvedores, mas com certeza é um dos principais componentes para se atingir este fim.

Neste contexto de qualidade de software o aparecimento da orientação a objeto, vem se enquadrando, nas necessidades tanto dos usuários finais, como dos desenvolvedores. Pois o salto para esta tecnologia ocorre simultaneamente ao surgimento de uma gama considerável de componentes de software, incluindo linguagens, interfaces do usuário, banco de dados e sistemas operacionais. Embora a programação orientada a objeto não seja uma panacéia, ela já demonstrou que pode auxiliar no gerenciamento do crescimento de complexidade e do aumento dos custos no desenvolvimento de software.

Segundo [WIN93], a orientação a objeto está integrada aos componentes fundamentais de software nos anos 90, ela representa o que a programação estruturada representava nos anos 70, um importante paradigma para melhorar a criação, a manutenção e o uso do software. Mudará também a maneira pela qual os programadores trabalham e aumentará a velocidade com a qual criam novas gerações de aplicativos, sem falar na capacidade de programação do usuário final. A orientação a objeto aumentará a funcionalidade que pode ser embutida aos aplicativos e permitirá aos usuários finais o acesso a diferentes tipos de dados em plataformas heterogêneas.

A orientação a objeto é particularmente adequada a um desenvolvimento sem remendos, isto é, em que o mesmo formalismo e a correspondente notação são utilizados em todo o ciclo de vida, através de refinamentos incrementais. As barreiras tradicionais entre a análise e o projeto e particularmente entre o projeto e a codificação, caracterizadas por mudanças de formalismo com correspondentes regras (formais ou informais) de transição, são possíveis de ser reduzidas. A análise e o desenho tem assim um papel mais importante do que nunca. A codificação pode vir a se tornar numa atividade de refinamento do desenho.

A melhor qualidade interna dos sistemas orientados a objetos é devida às novas abstrações trazidas por este paradigma, tais como classes, métodos, herança, polimorfismo, encapsulamento ou troca de mensagens e por uma acrescida ênfase na reutilização. Contudo, a utilização dessas abstrações pode ser muito variada, dependendo principalmente da habilidade do analista/desenhador pelo que, para o mesmo universo de discurso, poderemos ver surgir

produtos com diferentes qualidades, bem como ganhos de produtividade diferenciados, visto em [ABR93].

Basicamente a orientação a objeto mostra grandes ganhos para os desenvolvedores e por conseqüência para os usuários. Mas para isto existe um preço a ser pago principalmente pelos desenvolvedores dos projetos de software, que é uma mudança no seu estilo de ser, ou seja, de como eram projetos os sistemas tradicionais. Isto é explicado por [TAU96], “a orientação a objeto oferece um novo paradigma para o desenvolvimento de aplicações, prometendo melhorias significativas de qualidade e produtividade. Mas, como todo novo paradigma, rompe com os conceitos atuais e, por isso, provoca grandes mudanças culturais”.

4 MÉTRICAS PARA ORIENTAÇÃO A OBJETO

Foram pesquisadas algumas métricas para utilização em orientação a objeto, este conjunto de métricas deve ser concebido com base nos critérios descritos anteriormente. Este inclui, entre outras, as seguintes métricas:

- fator de herança de métodos;
- fator de herança de atributos;
- fator acoplamento, de encapsulamento;
- fator de polimorfismo;
- fator de reutilização.

Segundo [ABR93], as métricas se destinam a quantificar a presença ou ausência de uma certa propriedade ou atributo, pode-se vê-las como propriedades. O seu valor variará então entre 0 (Zero – Total ausência) e (Um – Máxima presença possível). Este tipo de interpretação permite a aplicação da teoria estatística às métricas de software. Métricas estatisticamente independentes podem, por exemplo, ser combinadas de forma a que o resultado possa ser ainda considerado como uma probabilidade.

Esta teoria é confirmada por [ZUS96], onde diz que a ação para ser tomada é a que segue: u(∅)=0, a qual corresponde novamente para a probabilidade. A correspondência não deveria influenciar a suposição que o software teria de qualquer maneira a medida exata para a programação orientada a objetos. No caso, uma das maiores propriedades da função da probabilidade é limitação [0,1].

Continua [ZUS96], a dizer que, a observação que as medidas na combinação com duas operações descritas aqui (parcialmente) são semelhantes para probabilidade de medir.

Segundo [ABR93], deve-se observar linhas para se seguir numa medição de um projeto, sendo que estes critérios também possam ser utilizados por vários projetistas em momentos diferentes, que deverão alcançar valores muito semelhantes. Para um melhor entendimento, deve-se observar alguns passos a seguir, como:

1) Definir a Forma de Obtenção das Métricas: devem ser recolhidas e analisadas ao longo do tempo em tantos projetos quanto possível, para estabelecer comparações e atingir conclusões. Contudo, estes projetos são certamente de dimensões

diferenciadas. Se as métricas que não as destinadas especificamente a quantificar a dimensão, forem dela dependentes, não é possível juntar dados conclusivos;

2) As Métricas Devem Ser Independentes: representam algum atributo do produto ou processo. Tem-se então um confronto com as unidades de medição. Unidades subjetivas ou artificiais levam inevitavelmente a desentendimentos;
3) As Métricas Devem Ser Coerentes: o custo de recuperação dos efeitos provocados por um erro, aumenta exponencialmente com o progresso do projeto alcançando desde que ele foi cometido. As métricas, em especial as de desenho, destinam-se a expor os defeitos provocados por esses erros, muitas vezes embutidos no desenho. Deve-se ser capaz de recolher métricas logo que um primeiro desenho esteja disponível, para identificar possíveis deficiências, antes que se invista muito esforço baseado nesse desenho;
4) As Métricas Devem Ser Obtidas Numa Fase Inicial do Ciclo de Vida: um sistema de software é geralmente desenvolvido por uma equipe de pessoas. Muitas vezes, dada a dimensão e complexidade do problema, a especificação é subdividida em subsistemas ou módulos quase independentes. Cada membro ou pequeno grupo de membros da equipe pode ser responsável por cada uma dessas subdivisões. Então, para além das métricas aplicáveis ao sistema na sua globalidade, necessita-se obter também para cada uma dessas partições, de forma a permitir identificar os mais "mal desenhados". Isto é confirmado por [DEM91], "a coleta e a análise de dados métricos são atividades passíveis de erro";
5) As Métricas Devem Ser Aplicáveis a Partições do Sistema em Consideração: o levantamento manual de métricas é uma tarefa longa e repetitiva, logo cansativa e pior que tudo, dispendiosa. Desde que o primeiro passo seja cumprido e que os desenhos também estejam formalmente definidos (numa linguagem qualquer de especificação). Todos que desenvolvem projetos, tem um objetivo a atingir com a utilização das métricas, por isso, mesmo sendo trabalhoso deve ser realizado com muito cuidado, para ficar bem claro objetivo a ser buscado. Ou seja, segundo [FER95], "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 obviamente do estágio de maturidade em que se encontra essas atividades";

6) As Métricas devem ser facilmente trabalhadas: estão disponíveis atualmente muitas linguagens de especificação e programação suportando as abstrações do paradigma orientado a objeto, segundo [COA92], “afirma-se que uma linguagem suporta um estilo de programação se ela fornecer facilidade que tornem conveniente (razoavelmente fácil, seguro e eficiente) a utilização deste estilo”. Ou seja, cada uma delas tem os seus mecanismos que permitem, um maior ou menor detalhamento, para realizar essas abstrações, também confirmado por [COA92], “todas as linguagens permitem a implantação orientada ao objeto. Contudo, algumas linguagens proporcionam uma sintaxe mais rica para a captura explícita da representação básica.”. Uma vez mais, a necessidade de uma base comum de entendimento para o processo de análise das métricas, tende a se levar a evitar o nível sintático;
7) As Métricas Devem Ser Independentes da Linguagem Utilizada: O conjunto de métricas utilizadas para se atingir os objetivos desejados, que são por exemplo, aumentar a produtividade do desenvolvimento, aperfeiçoar continuamente o processo e produto, melhorar a qualidade do planejamento, desenvolvimento e o resultado do processo, entre outros, não devem estar ligado a nenhuma espécie de linguagem para que os resultados sejam os melhores possíveis.

As definições das métricas são baseadas num conjunto de funções formalmente definidas e na teoria dos conjuntos. Isso garante os passos 1 e 6. Todas as métricas são expressas como quociente em que o numerador corresponde ao valor corrente da utilização de uma dada abstração do paradigma. O denominador é o valor máximo atingível para utilização dessa mesma abstração. Daqui resulta que todas as métricas são independentes de dimensão do sistema em consideração e também sem dimensão, pelo que os passos 2 e 3 são cumpridos. O passo 4 também é obedecido pois estas métricas podem ser utilizadas logo que um desenho preliminar esteja disponível. Podem ainda ser aplicadas a qualquer junção de classes, ou a qualquer combinação desses, pelo que se pode afirmar que o passo 5 é garantido. Não é feito nenhuma referência as estruturas específicas de uma determinada linguagem, pelo que também o passo 7 é atendido, segundo[ABR93].

4.1 ENCAPSULAÇÃO E OCULTAÇÃO DE INFORMAÇÃO

Muitos dos componentes de uma classe são desenhados para realizar certas funções que são só relativas à classe propriamente dita. Estes componentes devem ser ocultados dos programadores que usam essa classe, não por uma questão de proteção, mas sim para ajudá-los a lidar com a complexidade. Com efeito, os detalhes internos de implementação não garantem (ou não deveriam garantir) uma melhor perspectiva sobre a forma de utilizar os serviços dessa classe. Outra vantagem muito importante deste mecanismo de ocultação de informação é que a utilização da classe é independente da sua implementação, logo permitindo alterá-lo sem efeitos colaterais. Isto é confirmado por [MAR95] onde, "ocultação de informação: o objeto esconde seus dados de outros objetos e permite que os dados sejam acessados por intermédio de seus próprios métodos.".

Em conclusão, toda a informação sobre uma classe deve ser privada dessa classe, a não ser que especificado em contrário. A parte pública de uma classe, designada por interface, deve ser apenas a "ponta do iceberg". Os métodos públicos representam os serviços que uma classe fornecedora é capaz de disponibilizar a classes clientes. A parte ocultada é designada por implementação.

Segundo [ABR93], o número de métodos definidos na classe Ci é dados por:
Md(Ci) = Mv(Ci) + Mh(Ci).

Onde:

- Mv(Ci) – número de métodos visíveis (interface) da classe Ci.
- Mh(Ci) – número de métodos ocultados (implementação) da classe Ci.

Inversamente o número de atributos definido na classe Ci é dado por:
Ad(Ci) = Av(Ci) + Ah(Ci)

Onde:
Av(Ci) – número de atributos visíveis da classe Ci.
Ah(Ci) – número de atributos ocultados da classe Ci.

4.2 HERANÇA

A herança é o mecanismo para expandir semelhanças entre classes. Semanticamente permite representar situações de generalização e especialização. Do ponto de vista do desenho, permite simplificar a definição das classes que herdam. Quando uma classe herda de uma outra, isso significa que pode usar os seus métodos e atributos, a não ser que sejam redefinidos localmente. Uma classe Cd que herda direta ou indiretamente de uma classe Ca diz-se ser descendente (ou subclasse) da classe Ca a qual, inversamente, se diz ascendente (ou superclasse) da classe Cd. Sendo mais restritivo, uma classe Cc que herda diretamente de uma classe Cp diz-se sucessora da Cp a qual, inversamente, é chamada de progenitora da classe Cc. A herança pode ser simples ou múltipla, dependendo do número de progenitoras. Isto é confirmado por [MAR95], que diz, “herança simples é quando uma classe pode herdar a estrutura de dados e as operações de uma superclasse. Herança múltipla é quando uma classe pode herdar a estrutura de dados e as operações de mais de uma superclasse.”. A relação de herança será aqui representada por uma seta, como por exemplo em Cc $\rightarrow$ Cp ou Cd $\rightarrow$ Ca.

Para a definição do conjunto de métricas necessita-se de algumas métricas de classe. Estas serão introduzidas seguidamente, por meio de funções onde o argumento é a classe em consideração e o valor retornado corresponde à métrica em causa. Segundo [ABR93], seja Ci qualquer classe do sistema em consideração e onde TC é o número total de classes. Define-se:

- Total de Sucessoras, CC(Ci) – número de classes sucessoras de Ci (nota-se: se CC(Ci) = 0, então Ci é uma classe folha);
- Total de Descendentes, DC(Ci) – número de classes descendentes de Ci;
- Total de Progenitoras, PC(Ci) – número de classes progenitoras de Ci (nota-se: se PC(Ci) = 0, então Ci é uma classe de base; se PC(Ci)>1, tem-se herança múltipla);
- Total de Ascendentes, AC(Ci) – número de classes ascendentes de Ci;
- Métodos Definidos, Md(Ci) – número total de métodos definidos na classe Ci;
- Métodos Novos, Mn(Ci) – número total de métodos definidos na classe Ci, que não estão sobrepostos aos herdados;
- Métodos Herdados, Mi(Ci) – número total de métodos herdados na classe Ci que não foram redefinidos;
- Métodos Redefinidos, Mo(Ci) – número total de métodos herdados na classe Ci que foram redefinidos;

- Métodos Disponíveis, Ma(Ci) – número total de métodos que podem ser invocados em associação à classe Ci;
- Número Total de Métodos Herdados – Tmi(Ci);
- Número Total de Métodos Disponíveis – Tma(Ci).

Continuando, as relações aplicam-se e podem ser facilmente transpostas para atributos em vez de métodos:

- Md(Ci) = Mn(Ci) + Mo(Ci);
- Ma(Ci) = Md(Ci) + Mi(Ci).

Define-se então o fator de herança de métodos como:

- MIF = Tmi/Tma, note: MIF = 0, significa que há uma herança efetiva.

De igual forma se define o fator de herança de atributos:

- AIF = Tai/Taa, onde Tai e Taa tem definições semelhantes a Tmi e Tma.

4.3 ACOPLAMENTO E ENCAPSULAMENTO

Uma classe Cc é dita cliente de uma classe Cs e Cs uma fornecedora da classe Cc sempre que Cc contenha pelo menos uma referência a um componente (método ou atributo) da classe Cs. Esta relação cliente fornecedora será aqui representada por Cs $\Rightarrow$ Cs.

Algumas destas relações cliente-fornecedora podem ser vistas como comunicação entre as instâncias das classes (objetos). Estas comunicações devem ser explicitadas para efeitos de compreensão. Algumas abordagens utilizam a designação mensagem, evento ou estímulo, para se referir à chamada que uma instância de uma classe cliente faz a um serviço (método) da classe fornecedora.

As referências a classes fornecedoras não é efetuada simplesmente através de mensagens. Uma referência ao tipo da classe fornecedora pode ser efetuada em situações como:

- num atributo público ou privado;
- num argumento ou num atributo local de um método público ou privado;

Um maior número de relações cliente-fornecedora aumenta a complexidade, reduz o encapsulamento e o potencial de reutilização, além de limitar a facilidade de compreensão e de manutenção, segundo[ABR93].

4.4 POLIMORFISMO

Polimorfismo é uma palavra de origem grega que significa “muitas formas”. Quando aplicada a abordagens orientadas a objetos, pretende traduzir a possibilidade de enviar mensagens(solicitar serviços), sem saber qual vai ser a forma (classe) do objeto que a associará a um dos seus métodos de interface. Todas as potenciais classes receptoras pertencem à mesma hierarquia de classes. A associação pode ser estática (em tempo de compilação) ou dinâmica (em tempo de execução).

As mensagens podem destinar-se a instâncias de uma certa classe ou de seus descendentes, mas não ao contrário. Considere-se por exemplo que a classe bola é especializada pelas classes bola_tenis, bola_golfe, bola_futebol e bola_basquete. Se ao enviar a mensagem bola.new (chamada ao operador construtor do método), obtem-se uma bola_tenis ou uma bola_golfe, tudo está bem. Contudo, se a mensagem bola_futebol.new é enviada, a obtenção de uma instância qualquer da classe bola não é aceitável, pois esta pode construir, por exemplo, uma bola_basquete.

Segundo [ABR93], se não houver uma mensagem enviada a uma (instância de) classe ou a uma das descendentes, será associada ao mesmo método. Inversamente, o máximo potencial possível de polimorfismo será obtido se todos os métodos forem redefinidos em todas as classes. Com efeito, se um método M numa classe Ci, é redefinido em todos as suas descendentes, então uma mensagem associada a M pode Ter DC (Ci) possíveis “endereços” para além da implementação de M na classe Ci. Estes correspondem precisamente ao mesmo número de distintas implementações de M nas descendentes de Ci (situação polimórficas).

4.5 REUTILIZAÇÃO

Segundo [CAL97], a reutilização, reforçada pelas abstrações do paradigma orientado a objeto, espera-se vir a produzir um grande impacto na produtividade e na qualidade do software produzido. Aparentemente permite poupar muito tempo de desenvolvimento, logo reduzindo o custo final do sistema ou permitindo aplicar essa poupança na construção de mais aplicativos, garantia de qualidade ou outras. Os componentes reutilizáveis são geralmente desenhados com maior cuidado que o normal. Além disso, a sua utilização repetida faz transparecer quaisquer defeitos no seu desenho ou implementação. É por isso que estes

componentes tendem a ser de maior qualidade, a qual é incorporada assim nos sistemas que deles fazem uso.

A reutilização nas abordagens orientadas a objetos podem tomar principalmente duas formas: a reutilização de componentes de uma biblioteca (de classe) e a reutilização por meio de herança. Segundo [ABR93], pode-se assim considerar três tipos de classes num dado sistema:

a) as classes de base construídas totalmente de novo;
b) as classes extraídas de uma biblioteca;
c) as classes que reutilizam total ou parcialmente as já existentes, por meio de herança.

O esforço de especialização, junto com o de construir as classes do tipo (a), corresponde precisamente a outra parte, isto é, a correspondente às classes do tipo (b) e a fração de todas as outras que pode ser imputada à herança. Conforme [ABR93], o cálculo desta fração, apenas se irá considerar os métodos, pois que estes são de muito mais “cara” construção e manutenção do que os atributos.

5 ESPECIFICAÇÃO DO PROTÓTIPO

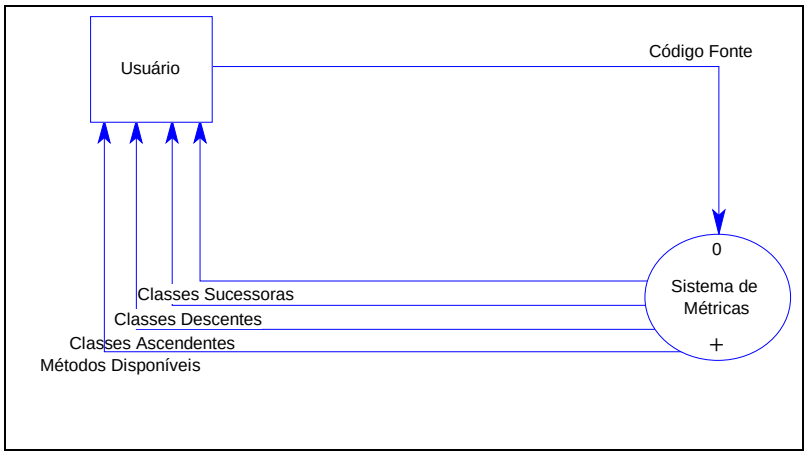
Neste capítulo está descrita a ferramenta de desenvolvimento utilizada para o desenvolvimento da protótipo e a especificação do mesmo, que inclui: a lista de eventos, as descrições de cada um dos eventos e o diagrama de contexto.

O ambiente Delphi4 foi utilizado para o desenvolvimento deste protótipo, pois seu uso adapta-se às características do protótipo, além de ser uma ferramenta amigável para o desenvolvimento.

5.1 DIAGRAMA DE CONTEXTO

O diagrama de contexto representa o software, as entidades externas e os fluxos de dados e é baseado na lista de eventos. Na figura 3 está representado o diagrama de contexto do protótipo.

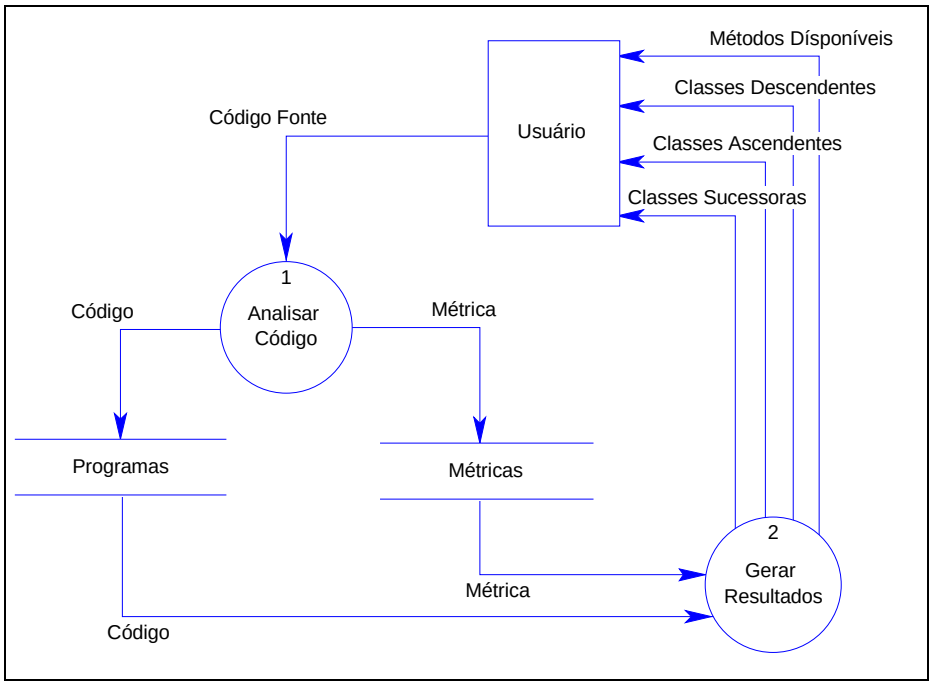
Figura 3 – Diagrama de contexto do sistema de métricas (OO)



5.2 DIAGRAMA DE FLUXO DE DADOS

Na figura 4 pode ser visto o diagrama de fluxo de dados do sistema.

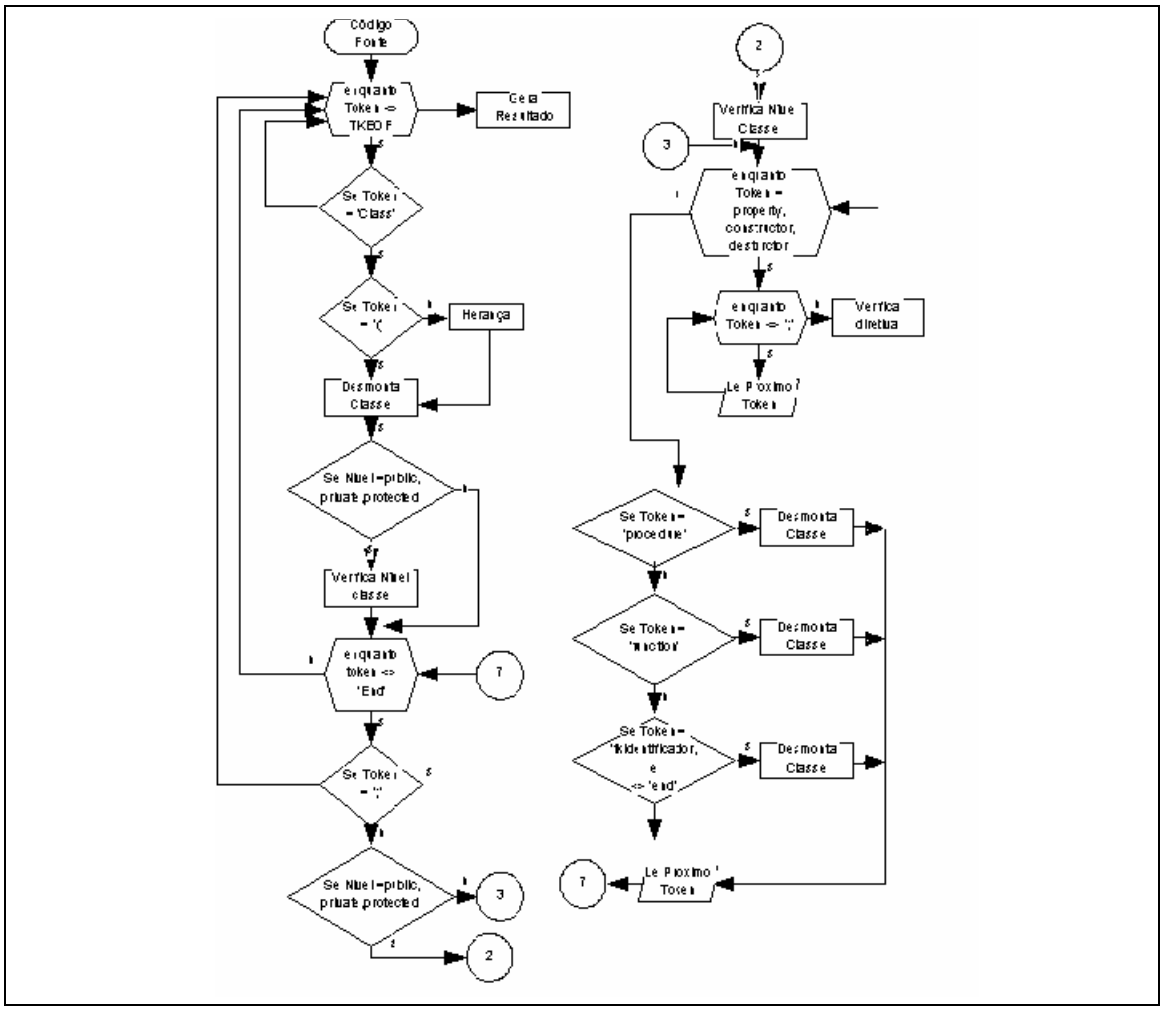
Figura 4 – Diagrama de fluxo de dados do sistema



5.3 FLUXOGRAMA DO PROTÓTIPO

A seguir será apresentado um fluxograma para melhor entendimento do protótipo desenvolvido. A notação utilizada é “uma representação gráfica de um fluxo lógico utilizando símbolos padronizados” [WEI94], cujos símbolos representam as operações realizadas pelo computador, proporcionando a visualização de como o programa se comportará de maneira mais legível. Os números que estão dentro dos círculos desenhados no quadro 1 abaixo, representam uma seqüência lógica de saída e entrada da mesma numeração em outra parte do fluxograma.

Quadro 1 .: Visualização algoritmo do protótipo através de fluxograma.



5.4 FERRAMENTA

Segundo [DAO95] e [SWA96] o ambiente de desenvolvimento do Delphi 4 está de acordo com o padrão Windows 95, e em sua tela principal (figura 5), pode-se controlar várias janelas como por exemplo a *Object Inspector*, *Form* e *Code Editor*. As características do ambiente de programação Delphi segundo [DAO95] e [SWA96] são:

a) linguagem descendente do Turbo Pascal;

b) possui programação orientada a objetos e em eventos;

c) linguagem compilada e não interpretada;

d) padrão SQL em banco de dados;

e) conectividade através de ODBC.

Este ambiente também disponibiliza alguns recursos como:

a) ambiente personalizado para o desenvolvimento;

b) programas compilados;

c) reutilização de componentes;

d) aplicativos;

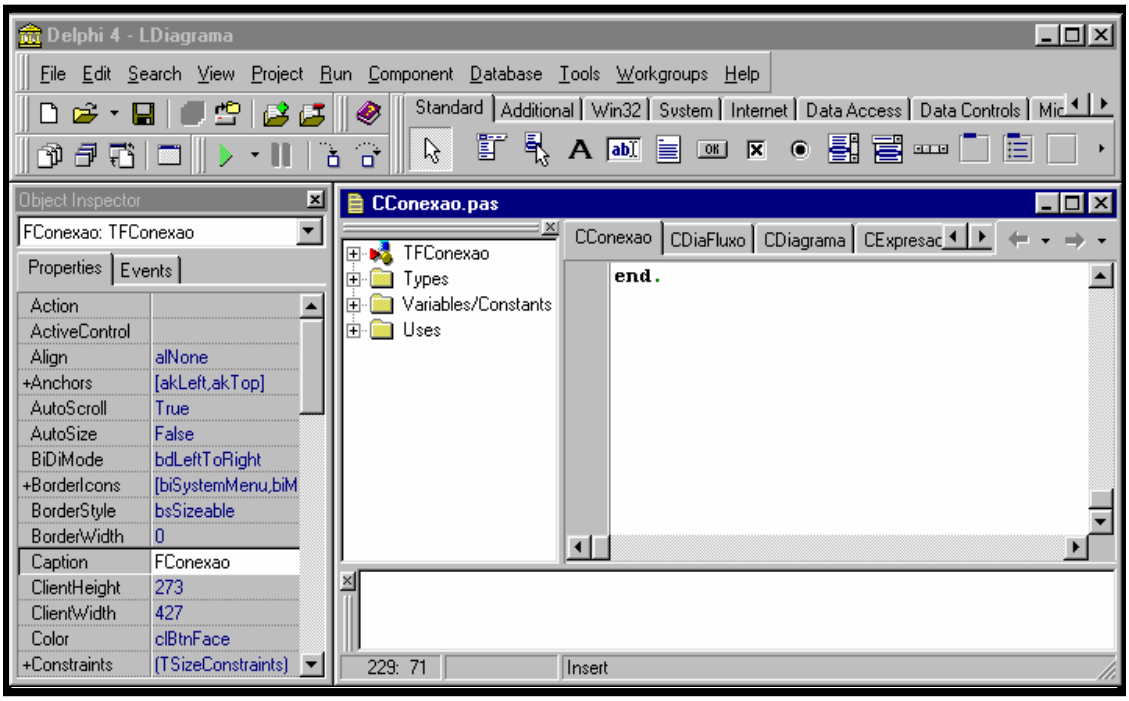
e) permite criar bibliotecas de funções;

f) assistente para criação de formulários;

g) suporte a OCX;

h) recursos para acesso a banco de dados.

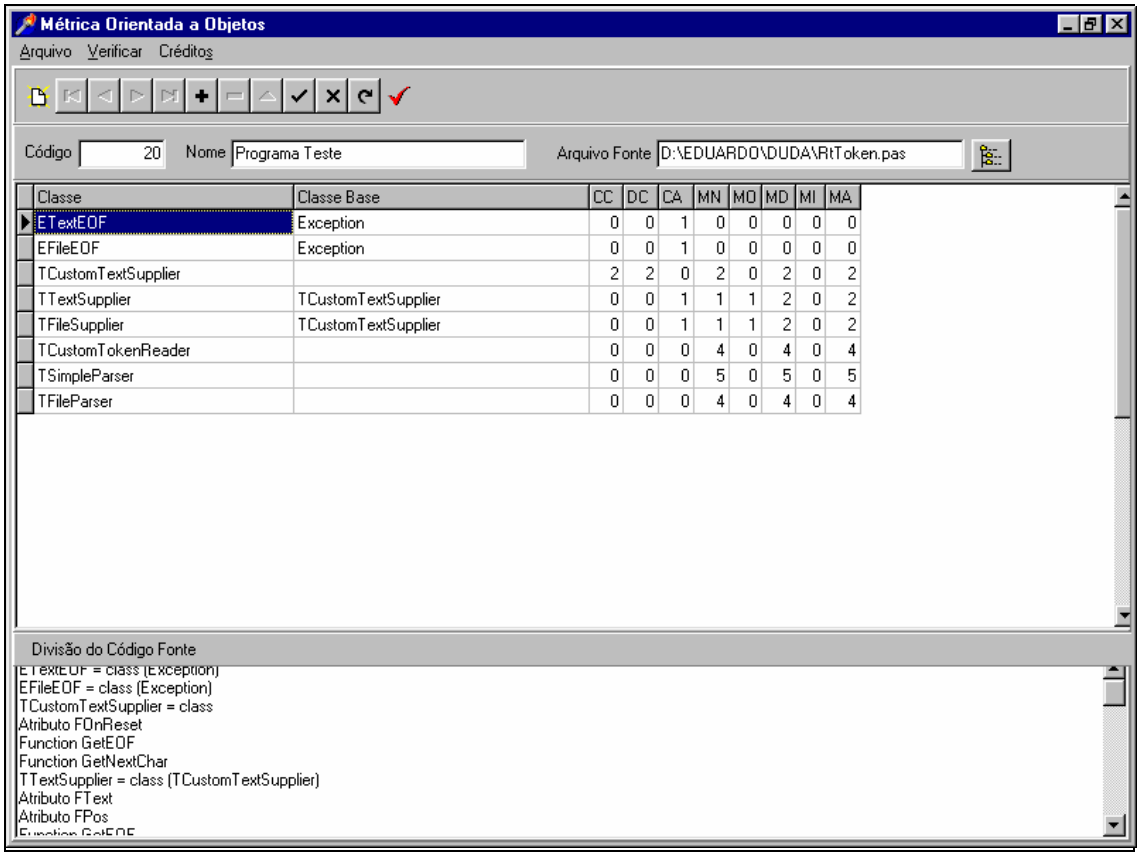
Figura 5 - Tela principal do Delphi 4.



5.5 DESCRIÇÃO DAS TELAS

Neste capítulo estão alguns detalhes do processo de implementação do protótipo. Veja a tela principal do sistema de métricas na figura 6.

Figura 6 – Tela principal do sistema de métricas



Este protótipo tem como principal objetivo o cálculo de métricas orientada a objetos. A maioria das funções realizadas pelo protótipo serão vistas posteriormente com as descrições dos componentes.

Para se avaliar um sistema no protótipo é preciso cadastrá-lo. Deve-se ir no menu Arquivo/Novo, ou clicar sobre o botão novo (+). O cursor irá se posicionar sobre o campo de edição “código do sistema” onde será efetuado o cadastramento. Após cadastrar o código do sistema, pressionando “enter” o cursor irá para o campo “Nome do Sistema” onde será cadastrado o nome adequado ao sistema, em seguida ao pressionar enter poderá ser cadastrado o caminho do “arquivo fonte” a ser avaliado. Ao cadastrar as informações do sistema a ser avaliado visto anteriormente, será habilitado o botão de verificação do sistema, que se pressionado resultará em métricas orientada a objetos que serão apresentadas na grade do protótipo, sendo mostrado nesta grade as classes do sistema avaliado com suas respectivas classes base (classe herdada pela classe avaliada), CC (classes sucessoras), DC (classes

descendentes), AC (classes ascendentes), métodos novos (métodos novos), MO (métodos redefinidos), MD (métodos definidos), MI (métodos herdados) e métodos disponíveis (MA).

Os valores apresentados na grade do sistema podem ser gravados através do botão gravar (V) na barra de ferramentas ou através do menu Arquivo/Gravar, tendo a opção de cancelar através do menu Arquivo/Cancelar. Temos um campo preenchido pelo sistema, onde são mostradas todas as classes que foram desmontadas com seus respectivos métodos e atributos do sistema avaliado. Para finalizar temos o botão ou o Arquivo/Sair, onde permite finalizar o protótipo.

Na parte superior da tela principal (figura 6), está uma barra de ferramentas (figura 7), que será agora explicada.

Figura 7 – Barra de Ferramentas

1) Fechar.: Fecha o protótipo em execução;
2) Primeiro.: Vai para o primeiro registro cadastrado no sistema de métricas;
3) Anterior.: Posiciona no registro anterior cadastrado;
4) Próximo.: Posiciona no próximo registro cadastrado;
5) Ultimo.: Posiciona no ultimo registro cadastrado;
6) Inserir.: Insere um novo registro ao sistema;
7) Excluir. : Exclui um registro já cadastrado;
8) Alterar.: Altera um registro já cadastrado;
9) Gravar.: Grava registro corrente;
10) Cancelar.: Cancela alteração ou inclusão de registro;
11) Verifica Código Fonte.: Varre o arquivo fonte especificado procurando as classes, classes base, atributos, métodos.

Na parte inferior a barra de ferramentas na figura 6, há uma grade com informações do código fonte avaliado contendo os seguintes dados:

a) classe;
b) classe base;

c) classes sucessoras (CC);

```
function TMainClass.GetSucessoras(aClassDef: TDefinitionClass): integer;
var
   xClassDef: TDefinitionClass;
   i : Integer;
begin
   Result := 0;
   for i := 0 to FListClass.Count -1 do
   begin
      xClassDef := TDefinitionClass(FListClass.Objects[i]);
      if CompareText(xClassDef.BaseClass, aClassDef.ClassName) = 0 then
         inc(Result);
   end;
```

d) classes ascendentes (AC);

```
function TMainClass.GetAscendentes(aClassDef: TDefinitionClass): integer;
var
   xClassDef: TDefinitionClass;
begin
   Result := 0;
   if aClassDef.BaseClass <> '' then
   begin
      inc(Result);
      xClassDef := GetClassDef(aClassDef.BaseClass);
      if xClassDef <> nil then
         Result := Result + xClassDef.ClassesAscendentes;
   end;
end;
```

e) classes descendentes (DC);

```
function TMainClass.GetDescendentes(aClassDef: TDefinitionClass): integer;
var
   xClassDef: TDefinitionClass;
   i: integer;
begin
   Result := 0;
   for i := 0 to FListClass.Count -1 do
   begin
      xClassDef := TDefinitionClass(FListClass.Objects[i]);
      if CompareText(xClassDef.BaseClass, aClassDef.ClassName) = 0 then
      begin
         inc(Result);
         Result := Result + xClassDef.ClassesDescendentes;
      end;
   end;
end;
```

f) métodos novos (MN);

```
function TMainClass.GetMetodosNews(aMetodos: TStrings): integer;
var
   ipub: Integer;
begin
   Result := 0;
   for ipub := 0 to aMetodos.Count -1 do
      if TTypeMetodos(aMetodos.Objects[ipub]) in [tmHerdavel, tmNone] then
         inc(Result)
end;

function TMainClass.GetNovos(aClassDef: TDefinitionClass): integer;
begin
   //Todos os metodos que estão na classe e não estão como virtual(Herdavél)
   Result := 0;
   Result := Result + GetMetodosNews(aClassDef.MetodosPublic) +
                      GetMetodosNews(aClassDef.MetodosProtected) +
                      GetMetodosNews(aClassDef.MetodosPrivate);
end;
```

g) métodos redefinidos (MO);

```
//Todos os métodos do typo public e protected que sao herdados(override)
function TMainClass.GetReDefinidos(aClassDef: TDefinitionClass): integer;
var
   i: integer;
begin
   Result := 0;
   for i := 0 to aClassDef.MetodosPublic.Count -1 do
   begin
     if TTypeMetodos(aClassDef.MetodosPublic.Objects[i]) = tmHerdado then
       Inc(Result);
   end;

   for i := 0 to aClassDef.MetodosProtected.Count -1 do
   begin
     if TTypeMetodos(aClassDef.MetodosProtected.Objects[i]) = tmHerdado then
       Inc(Result);
   end;

end;
```

h) métodos definidos (MD);

```
function TMainClass.GetDefinidos(aClassDef: TDefinitionClass): integer;
begin
   Result := aClassDef.MetodosNovos + aClassDef.MetodosRedefinidos;
end;
```

i) métodos herdados (MI);

```
 function TMainClass.GetHerdados(aClassDef: TDefinitionClass): integer;
 var
    xAllMetHerdados: TStringList;
    xclassBase: TDefinitionClass;
    i: integer;
 begin
    Result := 0;
    if aClassDef.BaseClass <> '' then
    begin
      xclassBase := GetClassDef(aClassDef.BaseClass);
      if xClassBase <> nil then
      begin
        xAllMetHerdados := TStringList.Create;
        try
          xAllMetHerdados.Duplicates := dupIgnore;
          xAllMetHerdados.Sorted := true;
          GetAllMetHerdados(xAllMetHerdados, xClassBase);
          Result := xAllMetHerdados.Count;
          for i := 0 to aClassDef.MetodosPublic.Count -1 do
            if xAllMetHerdados.IndexOf(aClassDef.MetodosPublic.Strings[i]) <> -1 then
              Dec(Result);
          for i := 0 to aClassDef.MetodosProtected.Count -1 do
            if xAllMetHerdados.IndexOf(aClassDef.MetodosPublic.Strings[i]) <> -1 then
              Dec(Result);
        finally
          xAllMetHerdados.Free;
```

j) métodos disponíveis (MA)

```
function TMainClass.GetDisponiveis(aClassDef: TDefinitionClass): integer;
begin
    Result := aClassDef.MetodosDefinidos + aClassDef.MetodosHerdados;
end;
```

Na figura 6 acima tem-se a exposição de todas as classes com seus respectivos métodos e atributos da maneira em que foi descrita no código fonte.

5.6 AVALIAÇÃO DOS RESULTADOS

Avaliando os resultados do programa anexo visto acima na figura 6, pode-se dizer que a quantidade de classes chave do sistema é 50%. Segundo [AMB98], normal de 30% a 50% e quanto a herança de classe pode-se dizer que quanto mais profunda for a árvore, constituirá projetos de maior complexidade.

Quadro 2.: Código fonte avaliado.

```
 ETextEOF = class(Exception);
EFileEOF = class(Exception);

TCustomTextSupplier = class
private
  FOnReset: TNotifyEvent;
  function GetEOF: Boolean; virtual; abstract;
public
  function GetNextChar: Char; virtual; abstract;
  property EOF: Boolean read GetEOF;
end;

TTextSupplier = class(TCustomTextSupplier)
private
  FText: String;
  FPos: Integer;
  function GetEOF: Boolean; override;
  procedure SetText(const Value: String);
public
  constructor Create( const aText: String );
  function GetNextChar: Char; override;
  property Text: String read FText write SetText;
end;

TFileSupplier = class(TCustomTextSupplier)
private
  FBuffer: String;
  FPosBuffer: Integer;
  FPos: Integer;
  FFile: TFileStream;
  function GetEOF: Boolean; override;
  procedure LoadBuffer;
public
  constructor create( aFileName: String );
  destructor destroy;
  function GetNextChar: Char; override;
end;

TCustomTokenReader = class
private
  FNextChar: Char;
  procedure ResetReader( Sender: TObject );
  procedure SetTextSupplier(const Value: TCustomTextSupplier);
protected
  FTextSupplier: TCustomTextSupplier;
  FCurrentToken: String;
  FCurrentTokenKind: TTokenKind;
  FCurrentColumn: Integer;
  FCurrentLine: Integer;
  function ProcessChar(var aToken: String; var aTokenKind: TTokenKind;
    const aNewChar: Char): Boolean; virtual;
public
  constructor Create( aTextSupplier: TCustomTextSupplier );
  procedure NexToken;
end;
TSimpleParser = class
private
  FSupplier: TTextSupplier;
  FReader: TCustomTokenReader;
  function GetCurrentToken: String;
  function GetCurrentTokenKind: TTokenKind;
  function GetCurrentLine: Integer;
public
  constructor create( aText: String );
  destructor destroy; override;
  procedure SetNewText( aNewText: String );
  procedure NexToken;
end;
TFileParser = class
private
  FSupplier: TFileSupplier;
  FReader: TCustomTokenReader;
  function GetCurrentToken: String;
  function GetCurrentTokenKind: TTokenKind;
  function GetCurrentLine: Integer;
public
  procedure NexToken;
end;
```

No quadro 2 mostrado pode-se visualizar o código fonte avaliado pelo protótipo. Comparando os resultados apresentados na figura 6 com o código fonte em questão, visto que as classes(8) estão na cor azul, classe base(2) em vermelho, métodos(19) em verde e atributos em rosa temos a amostragem que os resultados gerados pelo protótipo são condizentes.

6 CONCLUSÃO

A utilização de métricas orientada a objetos é uma necessidade cada vez mais evidente. Alguns dos principais conceitos de orientação a objetos (herança, encapsulamento, ocultação, polimorfismo e mensagens) são fundamentais para facilitar a manutenção e influenciar na produtividade.

Com estes conceitos de orientação a objeto, tem-se vários segmentos para se aplicar métricas orientada a objeto, principalmente comparando os vários resultados obtidos com a ferramenta desenvolvida de diferentes sistemas, relacionando custo, tempo, qualidade.

O protótipo desenvolvido também pode servir para se fazer auditoria em sistemas orientado a objetos, visto que o mesmo tem a capacidade de analisar a relação entre as classes, métodos e atributos do sistema em desenvolvimento ou desenvolvido. O usuário tem condições de avaliar o sistema em desenvolvimento num intervalo de tempo e saber o quanto realmente foi desenvolvido, utilizado, definido, redefinido no sistema.

No processo de implementação do protótipo, foi usado o TFileStream para segmentar os blocos em tamanhos fixos de 250k e controlar a carga do próximo bloco a ser carregado para memória e a descarga do bloco atual, devido a falta de memória ao carregar o código fonte de uma só vez, sem segmentá-lo.

As métricas escolhidas poderão ser utilizadas em programas codificados em ambiente Delphi para se avaliar o software desenvolvido, tendo-se dado preferencia as métricas de classe cujas característica se relacionam com seus métodos e atributos. As classes com número alto de métodos são de aplicações específicas, ao passo que classes com menos métodos tem a tendência de serem mais reutilizáveis.

O objetivo de divulgar e aprofundar os conhecimentos adquiridos foi possível, em função das pesquisas realizadas e apresentadas neste trabalho. O protótipo apresenta sua função principal, que é analisar código fonte retornando algumas métricas mais importantes.

6.1 EXTENSÕES PARA NOVOS TRABALHOS

a) correlação entre as métricas orientada a objeto e atributos de qualidade padrão, como os que hoje estão sendo muito comentados, como as normas da ISO;

b) disponibilizar um número maior de métricas ao protótipo implementado e fazer busca de herança em outras units do código fonte avaliado;
c) incluir gráficos e relatórios.

REFERÊNCIAS BIBLIOGRÁFICAS

[ABR93] ABREU, Fernando Brito. **Metrics for object oriented software development**. Acta da 3rd International Conference on Software Quality, ASQC, Lake Tahoe, New York, 1993.

[AMB98] AMBLER, Scott W.. **Analise e projeto orientado a objeto**. Rio de Janeiro : Infobook, 1998.

[ARI93] ARIFOGLU, Ari. **A methodology for software cost estimation**. São Paulo : Acm, 1993.

[BAS89] BASILI, Victor R.. **Avaliação quantitativa da metodologia de software**. São Paulo : Mis, 1989.

[CAL97] CALLESCURA, Wilson. **Métricas para sistemas orientados a objetos**. Itajaí, 1997. Monografia (Pós-Graduação Lato Sensu em Informática) Pró-Reitoria de Pesquisa, Pós-Graduação e Extensão - ProPPEx, UNIVALI.

[CAP95] CAPUTO, Geraldo. **Um estudo sobre métricas de produtividade no desenvolvimento de sistemas**. São Paulo : Ibpi, 1995.

[COA92] COAD, Peter. **Análise baseada em objetos**. Rio de Janeiro : Campus, 1992.

[DAO95] DAMASCENO, Américo Jr.. **Aprendendo Delphi avançado.** São Paulo : Érica, 1995.

[DEM91] DEMARCO, Tom. **Controle de projetos de software.** Rio de Janeiro : Campos, 1991.

[FER95] FERNANDES, Aguinaldo Aragon. **Gerência de software através de métricas garantindo a qualidade do projeto, processo e produto.** São Paulo : Atlas, 1995.

[FUC95] FUCK, Mônica Andréa. **Estudo e aplicação de métricas de qualidade do processo de desenvolvimento de aplicações em banco de dados** . Blumenau, 1995. Monografia (Bacharelado em Ciências da Computação) Centro de Ciências Exatas e Naturais, FURB.

[GOL95] GODFARD, Scott. **Software metrics, analysis and reporting.** Massachusetts : Ibpi, 1995.

[MAR95] MARIANO, Sandra Regina H.. **Introdução a métrica como fundamento de um programa de qualidade em software.** Recife : Wqs, 1995.

[MOL94] MOLER, K. H.. **Software metrics a practitioner's guide to improved product development**. Londres : Chapman & Hall Computing, 1994.

[PAC93] PACHECO, Roberto C. S.. **Introdução a programação orientada a objetos em C++**. Florianópolis. F.E.E.S.C., 1993.

[SCO96] SCOLA, Antônio Carlos. **Orientação a objetos**. Rio de Janeiro : Axcel Books, 1996.

[SHI93] SHILLER, Larry. **Excelencia em software.** São Paulo : Makron Books, 1993.

[SWA96] SWAN, Tom. **Delphi: bíblia do programador**. São Paulo : Berkeley Brasil, 1996.

[YOU95] YOURDON, Chu Shao. **Delphi: bíblia do programador**. São Paulo : Berkeley Brasil, 1995.

[ZUS96] ZUSE, Horst. O**bject oriented software measures**. Rio de Janeiro : Hall Book do Brasil Ltda, 1996.

[WEI94] WEISERT, Conrad. **Dicionário de programação**. Rio de Janeiro : Campus, 1994.

[WIN93] WINBLAND, Edwards. **Software orientado a objeto**. São Paulo : Makron Books, 1993.