AnaSoft: Um Analisador de Software Baseado em Métricas para Medir Complexidade

Carlos Mora Rodriguez

Este exemplar corresponde a redação final da tese devidamente corrigida e defendida pelo Sr. Carlos Mora Rodriguez e aprovada pela Comissão Julgadora.

Campinas, 20 de Março de 1990

Prof. Orion de Oliveira Silva

Orientador

Dissertação apresentada ao instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para obtenção do Título de Mestre em Ciência da Computação.

M79a

12602/BC

UNICAMP
BIBLIOTECA CENTRAL

AnaSoft: Um Analisador de Software Baseado em Métricas para Medir Complexidade[1]

Carlos Mora Rodriguez[2]

Orientador: Prof. Dr. Orion de Oliveira Silva[3]

Co-orientador: Prof. Dr. Arthur João Catto[4]

Departamento de Ciência da Computação

IMECC - UNICAMP

21 de Fevereiro de 1990

[1]Dissertação apresentada no Instituto de Matemática, Estatística e Ciência da Computação da Unicamp, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

[2]Bacharel em Ciência na área de Ciência da Computação, pelo City College of the City University of New York (1985).

[3]Professor e Chefe do Departamento de Engenharia de Software do ITA-Instituto Tecnológico da Aeronáutica.

[4]Professor Colaborador MS-4 do Departamento de Ciência da Computação - IMECC - UNICAMP; Diretor Geral do CTI-Centro Tecnológico para Informática.

Este trabalho dedico aos meus pais Antonia e Emerito.

AGRADECIMENTOS

Meus mais sinceros agradecimentos para:

- O professor Orion e sua família, pela sua confiança, amizade, dedicação e grande contribuição neste trabalho.
- Toda minha família tanto em Nova Iorque como na Repúbica Dominicana, pela força e estímulo constante que sempre recebi.
- A minha família chilena no Brasil, Dr. Faundes, Dra Ellen, Daniel, EllenSu e Viviana pela confiança, força, carinho e apoio que sempre me presentearam.
- O professor Arthur J. Catto, pela paciência, boa vontade, dedicação e atenção.
- A duas pessoas, Antonio Figueiredo e Adalgisa Barison, que tiveram uma participação importantíssima neste trabalho, com a sua ajuda, apoio, força, paciência e extrema boa vontade.
- A professora Claudia Medeiros, pelo apoio, contribuição, amizade e boa vontade.
- Meus amigos no Brasil, pelo carinho, amizade, apoio e principalmente pelo acolhimento tao generoso que foi fundamental para a minha adaptação neste belo país; entre eles Alfonso, Antonio Maranhão, Henrique, Thomas, Claudinha Japonesa e todos os outros.
- todos aqueles que também contribuiram e n ão foi possível mencionar pelo limite de espaço. Sem todas esas pessoas este trabalho não teria sido possível.

Conteúdo

RESUMO

Esta dissertação apresenta um estudo sobre as diferentes métricas de *software* para medir complexidade e propõe um analisador de código baseado em três destas métricas. Os aspectos considerados mais importantes em relação à complexidade de um programa são: a quantidade de dado manipulado, o fluxo de informação entre os módulos ou procedimentos e, finalmente, o fluxo de controle. As métricas escolhidas medem estes três fatores e fazem um diagnóstico da complexidade dos procedimentos do programa. Portanto, o objetivo do analisador proposto consiste em facilitar a manutenção de um software através de uma análise da complexidade dos procedimentos que os compõem. Finalmente, a ferramenta é testada em vários programas e são apresentadas as conclusões finais, que incluem extensões para pesquisas futuras.

SUMMARY

This disertation presents a study of the different software metrics available to measure complexity. In addition, it proposes a code analyzer, AnaSoft, based on three of the most important ones. The software aspects considered most importatnt in relation to software complexity are: the quantity of information processed, the flow of information among the components and the flow of control. The selected metrics measure this three fators and, at the same time, perform a diagnostic of the procedures' complexity. With this in mind, the main objective of the tool proposed is to aid the software maintener to perform a more efficient job. The analizer presented here was tested sucesfully in several software and conclusion were drawn. Finally, further extension for future research are suggested.

Capítulo 1

Introdução

A Engenharia de Software tem por objetivo a aplicação dos conhecimentos científicos ao processo de desenvolvimento de Software para produzir sistemas de baixo custo e de alta qualidade. Para alcançar estes objetivos é necessário identificar os fatores que influenciam um programa de tal forma que este fique mais caro e não necessariamente de melhor qualidade.

Quando surgiram os primeiros computadores, os altos custos do Hardware afetavam negativamente a qualidade dos programas. Isto acontecia porque a preocupação maior consistia em desenvolver Software que coubesse na memória da máquina sem se importar com a qualidade. O aperfeiçoamento da tecnologia do Hardware contribuiu para que a memória ficasse mais barata e o Software passasse a receber mais atenção por parte dos programadores. Atualmente, a qualidade dos Software tornou-se prioridade. Pode-se dizer, portanto, que houve uma mudança da preocupação inicial com a parte de Hardware, para uma preocupação maior em produzir programas de mais qualidade.

Como conseqüência desta mudança de objetivos, surgiu um conjunto de metodologias e ferramentas [BERGL82,DAVIS88] que visavam o melhoramento e otimização do processo de construção de programas. Entre estas técnicas pode-se citar a Programação Estruturada proposta por Dijkstra [MILLS86] , Tipos Abstratos de Dados para facilitar a implementação de níveis de complexidade [LISKO74], Máquinas Virtuais, Programação Orientada por Objetos [BOOCH86][HALBE86], ambientes automáticos para elaboração de Software [BARST85] [BALZE85] [DRUMM87]. Em resumo, pode-se dizer que o objetivo principal do processo de desenvolvimento tem sido controlar o produto final, o programa, através da utilização de algoritmos, ou metodologias, próprias para especificação e projeto ou através

da automação do processo (Ciclo de Vida). Além disso pode-se dizer que a preocupação inicial com a relação programa-máquina transformou-se em outro tipo de relação, programa-programador. Esta crescente preocupação com esta última relação programa-programador justifica-se também a partir de conclusões de numerosos relatórios na área, que apontam um aumento exagerado e progressivo no custo de Desenvolvimento e Manutenção de programas entre eles [BOEHM81].

1.1 Problemas na Engenharia de Software

A bibliografia consultada aponta que grande parte do orçamento para Software das empresas que utilizam computadores é dedicado para a área de manutenção. Uma análise cuidadosa desta situação nos revela que são vários os fatores responsáveis pelo aumento no custo de Software, entre os quais pode-se identificar:

- Uma grande carência de pessoal especializado, bem como o alto salário requerido. E, se for considerado que a cada dia aparecem sistemas novos e mais especializados e que as Universidades e centros de formação e capacitação não formam pessoal suficiente para atender a demanda do mercado, a tendência geral é de que o problema se agrave.

- O pequeno rendimento na produção de programas. Sabe-se que, em geral, a transferência de tecnologia é um processo que requer bastante tempo; e a área de Engenharia de Software não é uma exceção. O processo de construção de programas tem resistido fortemente à introdução de novas metodologias e à automatização, de tal forma que hoje em dia a maior parte do processo é feito por especialistas, utilizando-se ferramentas manuais ou semi-automáticas. É importante ressaltar que atualmente existe um grande esforço por parte de pesquisadores para desenvolver novas ferramentas e ambientes automáticos que facilitem a criação de programas, e permitam aumentar a produtividade.

- Uma grande demanda por Software mais confiável, fácil de usar e mais amigável "user-friendly" com o usuário. A necessidade de programas contendo estas características surge do fato de que, a cada dia, aumenta de uma maneira vertiginosa o número de pessoas sem conhecimento em computação e que estão utilizando o computador. Esta

necessidade fica mais evidente ainda quando se percebe que os computadores estão sendo usados cada vez mais em áreas de extremo risco, como aeroportos (por exemplo, para controlar os vôos), lançamento de foguetes, tanto civis como militares, nos sistemas de defesa dos países desenvolvidos, nos hospitais, para monitoramento de pacientes, etc. Uma falha em qualquer um desses sistemas acarretaria conseqüências graves e de custos imensuráveis.

- A defasagem existente entre as tecnologias de Hardware e Software. Ou seja, uma vez que um sistema de Hardware (ou um novo computador) é desenvolvido, os programas de aplicação para este sistema geralmente demoram muito a chegar ao mercado. O resultado disto é que os novos sistemas não se tornam operacionais por um período de tempo relativamente grande, e os custos são maiores do que foram planejados inicialmente.

- Incapacidade de cumprimento de prazos de entrega de Software. Para garantir que um sistema a ser desenvolvido seja concluído na data planejada são necessárias duas etapas:

  1. É necessário achar uma maneira de estimar com alto grau de certeza quanto tempo um determinado Software vai demorar para ser concluído.
  2. Precisa-se de uma ferramenta/metodologia para controlar o processo de desenvolvimento de uma maneira eficiente.

  A natureza seqüencial do processo de desenvolvimento faz com que um pequeno atraso em uma das fases propague-se automaticamente para as fases seguintes. A recuperação do atraso torna-se muito difícil, e como conseqüência disto o preço do sistema será maior.

- O alto custo da manutenção de Software, problema que está causando muita preocupação na comunidade da área de computação. Esta fase do Ciclo de Vida nem sempre recebeu a merecida atenção. Aliás, por muito tempo o significado deste termo foi o de correção de erros de um sistema de Software concluído e entregue. Porém, em um sentido mais amplo, manutenção significa qualquer mudança feita no Software para torná-lo mais eficiente; estas modificações podem incluir correção de defeitos, melhorias no tempo de execução, inclusão de novas rotinas para ampliar o produto e, inclusive, melhoria da documentação. Pode-se dizer que, em geral, os Software não são projetados pensando-se na

facilidade da manutenção e que a execução desta tarefa é feita por pessoas com pouca experiência ou preparação e isto dificulta a tarefa.

Vários estudos recentes têm sugerido que um dos fatores que mais influenciam o custo de um sistema é a sua manutenção. Esta fase do Ciclo de Vida, que no início não existia ou não era considerada muito importante, tem-se tornado uma das mais importantes e dispendiosas. [BOEHM81] afirma que ela sozinha equivale a 40% - 70% dos custos de desenvolvimento. O aumento nos custos de manutenção tem levado a um aumento geral nos custos de produção de Software e, consequentemente, a um aumento considerável no preço do produto final.

Se mede um Software para poder indicar a sua qualidade; para avaliar a produtividade das pessoas que o fizeram; para apontar os beneficios ou prejuizos que uma metodologia acarreta; para criar uma base de dados que sirva para fazer estimativas e estabelecer valores padrões; e, finalmente, para justificar a aquisição de novas ferramentas ou treinamento adicional dos programadores.

1.2 Possível Solução

Os cientistas da área de Engenharia de Software têm começado a aceitar a idéia de que é necessário analisar todo o processo de desenvolvimento de programas, desde a fase de Especificação até a Manutenção (Ciclo de Vida), como uma forma mais eficaz de melhorar a qualidade e controlar os custos dos programas. Esta tese postula que a utilização de **Métricas para Software**, método manual ou automático para quantificar características de Software e facilitar a comprensão acerca da maneira como estas influenciam um sistema ou o próprio processo de desenvolvimento, é essencial para o gerenciamento eficiente do processo de construção de programas. As métricas têm como função determinar quantitativamente as características essenciais dos programas (como por exemplo complexidade, confiabilidade e esforço), de tal forma que se possa fazer sua classificação, comparação e análise matemática. Uma vez que as métricas estejam identificadas, precisa-se medir os programas de uma forma algorítmica e objetiva para que os valores obtidos sejam consistentes nos diferentes programas e, ao mesmo tempo, independentes do medidor. O passo seguinte para controlar o processo é construir modelos dos fatores mais pertinentes, como por exemplo esforço e dificuldade, baseados nas métricas disponíveis. Uma vez que um gerente de projeto conheça esses fatores e o seu comportamento, as suas decisões

podem influenciar os fatores de tal maneira que as metas do processo de desenvolvimento sejam realizadas dentro dos prazos estabelecidos.

Em outras palavras, o uso adequado de métricas para programas tem o potencial de nos permitir reconhecer e desenvolver programas altamente eficientes de uma forma objetiva, e estimar com precisão os custos adicionais ou ganhos [CONTE86].

O maior avanço no estudo das métricas foi dado por M. H. Halstead da Universidade de Purdue. Ele publicou o resultado de sua pesquisa em 1977 sob o título de "Software Science" [HALST77]. Partindo do princípio de que os algoritmos têm características semelhantes às leis da Física, Halstead definiu um conjunto de métricas que visavam medir, entre outras coisas, o esforço necessário para construir programas. Através da contagem do número de operadores, operandos e a freqüência com que estes aparecem em um programa, ele afirma ser possível quantificar a magnitude deste esforço. O valor obtido deve ser considerado como o número de comparações mentais necessárias para construir o programa. Além disso, Halstead definiu várias outras métricas para estimar o número de erros, qualidade e o tempo de elaboração de um programa.

Uma outra métrica que tem ganho ampla aceitação na área de complexidade é a de T. McCabe [McCAB76]. Esta métrica, chamada de Complexidade Ciclomática, está baseada na teoria clássica dos grafos. Basicamente, ela conta o número de estruturas de controle de um procedimento e através desta contagem calcula todos os caminhos possíveis nele. McCabe relaciona a Complexidade Ciclomática à dificuldade em testar, comprender e modificar um programa. Esta medida foi também usada para medir a dificuldade mental de implementar um algoritmo. À medida em que se aumenta o número de estruturas de controle de um programa, aumenta-se também a sua complexidade [CURTI79].

O trabalho de [RAMA88] é também considerado muito importánte pois sugere a síntesis das métricas de Halstead "Software Science" e a de McCabe, Complexidade Ciclomática, para medir as caraterísticas dos Software. Esta métrica é chamada por Ramamurthy de **Medidas Ponderadas**, "Weighted Measures". Tanto as métricas de Halstead como a de McCabe só medem um conjunto determinado de caraterísticas e estes conjuntos são disjuntos; portanto Ramamurthy apresenta uma familia de métricas baseadas na síntese das métricas "Software Science" e Complexidade Ciclomática, que seriam capazes de medir todas as caraterísticas destas duas métricas juntas.

Além deste trabalho existe um outro um pouco parecido [DAVIJ88] que também sugere a utilização de combinação de métricas para medir progra-

mas. Davis afirma que é muito difícil definir uma métrica única que possa medir todas as caraterísticas de um programa e que portanto deve se escolher cuidadosamente aquelas que efetivamente medem as caraterísticas que se deseja. Ele analisa, avalia e compara as métricas do Esforço (E) de Halstead, Complexidade Ciclomática (V(G)) de McCabe e Linhas de Código para determinar a eficiência de cada uma destas.

Henry e Kafura [HENRY81] apresentaram uma outra métrica para medir a complexidade baseada no fluxo de dados ou interconeções entre os módulos. Segundo Henry, a complexidade de um programa pode ser medida através de uma análise do seu tamanho e de sua relação com o ambiente de Software.

A maioria destas métricas tem por objetivo principal quantificar e analisar as características de Software para determinar a influência que estas exercem no seu desenvolvimento, entendimento e manutenção.

1.3 Objetivos

Tendo em vista os problemas descritos anteriormente que afetam a Engenharia de Software, particularmente as áreas de Testes e Manutenção, esta tese propõe-se estudar, analisar e apresentar uma solução para simplificar os problemas da Manutenção e Testes de Software, solucionando estes problemas através da aplicação de métricas.

A solução apresentada consiste em uma ferramenta automática, baseada em várias das métricas mais eficientes (Esforço, Dificuldade, Complexidade Ciclomática e Fluxo de Dados) para medir a complexidade de um Software. Esta solução surgiu como conseqüência de numerosos estudos, entre eles [JAY85][KAFUR87][EVANS87][LI87], que apontam uma relação direta entre a complexidade de um Software e a facilidade de Teste e Manutenção. Acredita-se que esta relação é diretamente proporcional, ou seja, quanto maior a complexidade de um Software maior a dificuldade de testá-lo e mantê-lo.

Avaliar um Software é uma tarefa complicada, por vários motivos. Por exemplo, frequentemente a característica que se quer quantificar só existe no nível conceitual, isto implica falta de parâmetros de comparação, o que dificulta mais a tarefa. Outro problema é a grande variedade de técnicas para avaliação, e a falta de um consenso geral entre os pesquisadores sobre o que se quer medir. Ainda outro problema ocorre na hora de decidir qual é o fator, ou fatores, que determinam ou influenciam mais a característica de

interesse. A maioria dos pesquisadores da área de métricas de Software concorda que existem vários aspectos que influenciam a complexidade incluindo o **Tamanho, Linguagem de Programação, Estrutura Lógica, Quantidade de Dados e Fluxo de Informação (ou Comunicação com o Ambiente).**

1.4 Contribuição

A ferramenta implementada tem como função principal auxiliar o chefe de uma equipe de desenvolvimento a manter um controle mais eficiente da complexidade do Software. Ela tem a vantagem de poder ser aplicada no sistema desenvolvido antes que seja entregue ao usuário. Além disto, também pode ser aplicada em sistemas que já existem e, desta maneira, permitir manutenção preventiva. Ela alivia o chefe da equipe de desenvolvimeto da necessidade de fazer uma revisão manual do Software desenvolvido para verificar se os critérios de complexidade estabelecidos na fase inicial do processo foram mantidos pelos programadores. A sua aplicação permite uma melhor alocação dos recursos financeiros e humanos na manutenção de sistemas de Software através de uma análise da complexidade. Finalmente, com a aplicação desta ferramenta, o projetista poderá ter medidas mais precisas da Qualidade do Software desenvolvido e até aperfeiçoar as métricas, adaptando-as a seu ambiente de trabalho.

1.5 Organização da Tese

Esta tese está organizada da seguinte maneira: O Capítulo 2 define e analisa as características de Qualidade de um Software. Cada característica é definida de acordo com outros critérios de nível mais baixo. O Capítulo 3 define o Ciclo de Vida de um Software, as suas fases, e também apresenta as metodologias que implementam dito Ciclo. É dada maior ênfase ao método clássico de desenvolvimento, que é contrastado com outras metodologias mais recentes. O Capítulo 4 apresenta e analisa as métricas classificando-as de acordo com o aspecto do Software que elas quantificam. O Capítulo 5 descreve a ferramenta proposta, AnaSoft, no nível funcional, enquanto o Capítulo 6 descreve a sua implementação. O Capítulo 7 apresenta vários resultados obtidos com a aplicação da ferramenta e, finalmente o Capítulo 8 contem as conclusões e sugere possíveis extensões para este trabalho.

Capítulo 2

Ciclo de Vida

A elaboração de um programa implica todo um processo que começa na mente do Cliente/Usuário quando ele sente necessidade do produto, e se completa quando este produto é entregue testado e funcionando. Este processo formal, ou informal, pelo qual todo Software passa chama-se de **Ciclo de Vida**. Pode-se definir este Ciclo como um conjunto de fases que começa com a especificação dos requisitos e continua até a sua manutenção. Geralmente, as etapas que compõem o Ciclo de Vida de um Software são definidas separadamente, mas na realidade elas se interpõem. A Figura 2.1, retirada de [GOLBE86], mostra um diagrama do Ciclo de Vida em forma de "V". A extremidade esquerda desta Figura representa a parte inicial do processo, quando se formalizam os requisitos, e a parte direita a fase de aceitação do Software pelo usuário. As setas horizontais mostram em que momento as fases da esquerda são validadas.

2.1 Fases do Ciclo de Vida

O Ciclo de Vida, no geral, está composto de várias fases principais que são: **Requisitos/Especificação, Projeto, Detalhamento, Programação/ Codificação, Integração, Instalação/Aprovação** e **Manutenção.** Estas fases nem sempre acontecem de uma maneira seqüêncial e aparecem como formando parte do Ciclo de Vida. A seguir, uma descrição de cada uma delas.

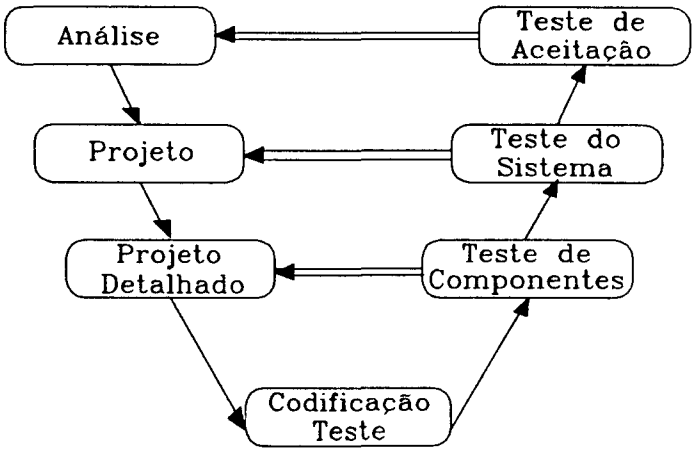

Figura 2.1: Modelo do Ciclo de Vida em forma de "V"

2.1.1 Requisitos/Especificação

O objetivo desta fase é a elaboração completa, consistente e clara dos requisitos do projeto, os quais servirão como base dos acordos entre a equipe de desenvolvimento e o usuário. Também deve ficar absolutamente definido o que o Software deve fazer. Ou seja, nesta fase se definem as especificações da parte funcional, assim como as características de desempenho do Software. Deve-se também avaliar nesta fase tudo o que concerne a recursos e orçamentos preliminares. É preciso que os analistas comuniquem-se de uma forma clara e precisa com o usuário para evitar erros na especificação.

A importância desta fase baseia-se em dois fatos. Primeiro, esta é a fase mais fácil de ser ignorada pois o impulso inicial é partir logo para a Codificação. Segundo, os erros e deficiências cometidos nesta fase são extremamente caros e difíceis de corrigir. Estudos realizados pela IBM, GTE e TRW, citados por [BOEHM76], concluem que vale a pena investir no esforço de achar erros na fase de especificação dos requisitos. Por exemplo, [BOEHM76] afirma que um homem/hora gasto na fase de especificação, equivale a 100(cem) homens/hora necessários para corrigir o mesmo erro na fase de operação de um programa.

Além de um custo muito alto no desenvolvimento de Software, existem outros problemas gerados por uma definição imprecisa dos requisitos. Por exemplo, a técnica de "Top-Down", que parte das especificações e executa refinamentos sucessivos, seria impossível de implementar sem uma boa especificação dos requisitos. Um outro problema aparece na hora de aplicar o teste de "caixa preta", que consiste em comparar os resultados do produto com a especificação. Se por algum motivo as epecificações estiverem erradas, o teste não tem validade. Finalmente, as especificações funcionais são utilizadas pelos chefes das equipes para controlar e verificar se o desenvolvimento do Software progride no caminho certo; sem especificações corretamente definidas, a chefia de um projeto não conseguiria controlar o processo de desenvolvimento de uma maneira efetiva.

Atualmente há uma tendência forte de desenvolver programas corretos por construção. Nestes programas, o refinamento obedece regras específicas, visando diminuir a probabilidade de erros.

Na prática, os requisitos de um projeto são definidos usando-se uma linguagem natural, sem formalização específica. Existe, porém, uma tendência cada vez maior de se utilizar linguagens para especificação de requisitos, como por exemplo PSL "Problem Statement Language", assim como anali-

sadores de definição de problemas automáticos, ou PSA "Problem Statement Analyser" [BOEHM76].

2.1.2 Projeto

Esta fase do processo não deve ser iniciada enquanto a anterior não tiver sido completada. Em outras palavras, não é possível fazer um bom projeto quando não se tem uma especificação completa, consistente, válida e sem ambigüidade. Na elaboração do projeto, especificam-se a configuração geral do sistema, a linguagem de implementação, os subsistemas maiores e suas interfaces, configurações de controle, estruturas de dados e um plano de teste. Infelizmente, esta etapa do desenvolvimento é feita ainda manualmente. Algumas das implicações quando se faz um projeto são as seguintes: o projetista pode fazer tanto um bom projeto como um projeto ruim; por incrível que pareça, as possibilidade dele fazer um projeto ruim são grandes. Em um projeto, o número de fatores que devem ser considerados é muito grande; por exemplo, o sistema operacional que vai ser usado, a linguagem, interfaces, custos de desenvolvimento, e outros. Contudo, os projetistas dispõem de um grande número de ferramentas para ajudá-los a diminuir as alternativas que precisam levar em consideração. Por exemplo, há hoje uma forte tendência à padronização de equipamentos, o que diminui o problema na hora de escolher a máquina em que o Software vai ser implementado. Pode-se afirmar, porém, que ainda hoje não se faz suficiente esforço na validação e análise de risco antes de aprovar o desenvolvimento de um projeto de Software.

Entre as técnicas mais usadas na área de projeto está a já mencionada técnica de desenvolvimento "Top-down" ou HIPO. Ela parte de uma definição do problema em seu nível mais alto de detalhamento usando uma estrutura de controle hierárquica, que controla uma entrada "Input", um processo e uma saída "Output". Depois continua com um refinamento iterativo aplicado a cada componente, até que o sistema todo esteja especificado no nível de detalhamento desejado. Os refinamentos em cada nível são chamados de Níveis de Abstração ou Máquinas Virtuais [BOEHM76].

Uma outra técnica muito utilizada chama-se modularização. Ela consiste no agrupamento de funções de acordo com o tipo de relacionamento, entre elas: relação coincidível, lógica, clássica, procedural, comunicativa, funcional, e outras. A regra geral para o agrupamento de funções sugere que cada função seja agrupada junto àquelas em que a relação é mais forte.

Outras técnicas também usadas são as de representação do projeto com

diagramas tanto de fluxo de dados, DFD de Jackson, como de controle. Várias das técnicas atuais estão sendo modificadas para incorporar a visão de objeto, ou seja, os programas ou sistemas são visto como manipuladores de objetos (como por exemplo, um avião seria um objeto composto de assas, turbinas, rodas e outras peças) no lugar de arquivos.

2.1.3 Detalhamento do Projeto

Esta fase é uma extensão da anterior e se preocupa com a especificação mais detalhada dos módulos, como tamanho, comunicações necessárias entre eles, algoritmos que vão ser usados, interfaces para as estruturas de dados, estruturas de controle interno, etc.. Nesta fase devem-se ressaltar as limitações importantes do sistema relacionadas com o tempo de execução e armazenamento e, finalmente, deve-se incluir um plano de teste individual para cada módulo, que são os testes unitários. O resultado desta fase consiste no programa em pseudo-código ou numa linguagem natural estruturada.

2.1.4 Programação/Codificação

Esta fase consiste na codificação do projeto elaborado na fase anterior. Ou seja, nela implementam-se os algoritmos (módulos) na linguagem pré-selecionada e iniciam-se os testes dos módulos, bem como dos subsistemas. Pode-se dizer que o método de programação mais usado é o da Programação Estruturada. Porém, as tendências das pesquisas que estão sendo feitas na área concentran-se na geração automática de código, como por exemplo, os ambientes CASE. Além da programação estruturada, a programação orientada por objeto também está sendo usada em quase todas as áreas de programação, independentemente da aplicação. Por exemplo, há uma forte tendência na área de Banco de Dados para desenvolver novos sistemas que incorporem a visão por objeto.

2.1.5 Instalação/Aprovação

Esta fase é comumente chamada de fase de teste. Nos anos 70 a fase de teste não era iniciada até que o Software estivesse terminado. Esta fase recebia pouca atenção e o esforço para testar era mínimo. Atualmente, as exigências de confiabilidade dos sistemas, bem como o aumento nos custos de desenvolvimento de Software têm incentivado o engenheiro de Software a aprimorar o processo de teste. Um dos principais fatores apontados como

responsáveis pelo aumento excessivo dos custos de desenvolvimento de Software, têm sido as mudanças necessárias para corrigir os erros encontrados em programas depois de entregues.

Nesta fase o produto passa por testes finais de aprovação no ambiente para o qual ele foi projetado. Também a documentação e os manuais que acompanham o sistema são entregues; o pessoal da empresa é treinado; problemas são anotados e correções efetuadas até o cliente ficar satisfeito e, finalmente, aceitar o produto.

As técnicas mais conhecidas para teste são definidas como Teste Caixa Preta, baseado na especificação funcional, e Teste Caixa Branca, baseados na estrutura lógica do programa; este último foi introduzido por [McCAB76] . Um outro tipo de teste também aplicado é o Teste de Caixa Cinza definido por [KANDU79] e que representa uma combinação dos dois anteriores. Além dos testes mencionados anteriormente, há uma tendência, sem muito sucesso, de incorporar a validação matemática dos programas. Esta corrente tem tido pouca aceitação porque ela é tediosa, muito demorada e requer uma forte base em lógica e matemática.

2.1.6 Manutenção

Esta é uma das fases mais importantes do Ciclo de Vida de um Software e no passado recebeu pouca atenção. A manutenção consiste em todas as mudanças que são feitas em um programa depois que ele for entregue ao cliente/usuário final. Esta etapa pode ser classificada em dois tipos: **Atualização** do Software, o que implica modificações na especificação funcional, e **correção** de erros, sem afetar a especificação funcional.

Independentemente de se um programa vai ser atualizado ou corrigido, existem duas atividades que fazem parte de toda Manutenção. A primeira refere-se ao entendimento do programa, e isto implica a necessidade de boa documentação, bom mecanismo de correspondência entre os requisitos e o código, assim como um código bem estruturado. A segunda tem a ver com a modificação do programa. Para se modificar um Software é preciso que tanto o próprio Software como o Hardware e as estruturas de dados sejam flexíveis e fáceis de se expandir. Também precisa-se de mecanismos que evitem ou minimizem os efeitos colaterais das mudanças, sem esquecer que também a documentação tem que ser fácil de se modificar.

Algumas das ferramentas que ajudam a simplificar o processo de Manutenção são a Programação Estruturada, Formatação Automática de Código, Modularização e Abstração de Dados. Porém, uma outra tendência na área

surgiu da conclusão dos pesquisadores e cientistas de que a Manutenção é afetada principalmente pela complexidade do programa [HARRI82]. A idéia básica consiste em definir um mecanismo que facilite a medição da complexidade de cada módulo ou procedimento de um programa e, desta forma, fazer uma distribuição da manutenção de acordo com esta complexidade. Este assunto será tratado com mais profundidade no próximo Capítulo.

As etapas descritas acima, e que fazem parte do Ciclo de Vida de um Software, são geralmente seqüenciais e estão fortemente relacionadas de tal forma que uma mudança em uma delas afeta significativamente as atividades das outras. Recomenda-se que cada fase seja submetida a uma cuidadosa verificação e validação para garantir que o sistema a ser implementado satisfaça as especificações e requisitos.

2.2 Metodologias para Desenvolvimento de Software

Existe uma grande variedade de metodologias que têm como finalidade modelar o Ciclo de Vida de um Software. Estas surgiram da necessidade de decompor o processo de elaboração de programas, e desta forma diminuir a complexidade dos problemas ou tarefas a serem automatizadas; da urgência em diminuir os altos custos do desenvolvimento de programas; da necessidade de obter um Software mais eficiente; e da insatisfação com o desempenho e a funcionalidade dos programas produzidos. Contudo, os problemas acima citados têm motivado os Engenheiros de Software a desenvolver novas metodologias que tentam encarar a Crise de Software de uma maneira mais prática e eficiente. O modelo mais importante durante muito tempo foi o **Modelo Clássico**, também chamado de **Cachoeira**. Bergland, em um estudo comparativo de metodologias, identifica outras três metodologias também importantes, que são: **Decomposição Funcional, Fluxo dos Dados** e **Estrutura dos Dados** [BERGL82]. Segue-se uma breve descrição de cada uma delas.

2.2.1 Modelo Cachoeira

Este é o modelo mais conhecido e usado pela comunidade científica dentre todos os que existem atualmente (Figura 2.2, retirada de [BOEHM81]). Também chamado de modelo clássico, ele foi introduzido por Roy pela primeira vez em 1970, e mais tarde refinado por Boehm em 1976. Pode-se afirmar que todas ou quase todas as metodologias utilizadas no processo

de desenvolvimento de Software hoje em dia são uma variação do método clássico; as diferenças aparecem apenas na nomenclatura. Apesar do modelo clássico apresentar algumas dificuldades, ele é o método mais usado. Além de ser o modelo que mais tempo de uso tem, de ser muito conhecido, estendido e aperfeiçoado, ele oferece um esquema do qual todos os outros modelos podem ser derivados.

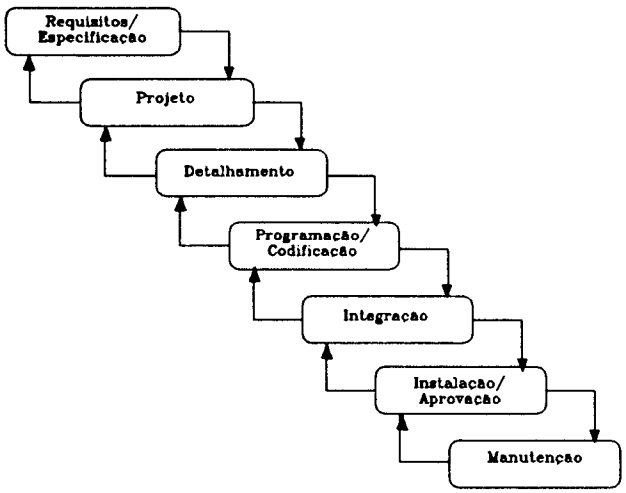

Figura 2.2: Modelo Clássico ou Cachoeira

2.2.2 Decomposição Funcional

Esta metodologia, analisada am [BERGL82], consiste na divisão do problema em componentes menores, os quais podem ser implementados em funções. O princípio básico por trás dela é aquele de "dividir para conquistar". A técnica aplicada no projeto é a de "Top-down". A estratégia consiste em, primeiro, definir claramente as funções desejadas; segundo, dividir, conectar e verificar cada função, de tal forma que ela corresponda a duas ou mais sub-funções, cada uma resolvendo uma parte do problema. As vantagens do uso desta metodologia são: produz estruturas de programas muito eficientes; é eficiente para aplicações matemáticas e tem uma ótima

flexibilidade para suportar futuras alterações. As desvantagens mais evidentes são: se não é usada corretamente produz coesão lógica e, ocasionalmete, partições exageradas ou "telescoping"; em alguns casos, como ela não determina em que a especificação deve se basear (tempo, fluxo de dados, grupos lógicos, acesso a recursos compartilhados, e outros), isto pode levar a um nível de coesão bom, regular ou ruim; é imprevisível, no sentido de que dois projetistas têm poucas chances de resolver um problema da mesma maneira aplicando esta técnica.

2.2.3 Metodologia Baseada no Fluxo de Dados

Esta metodologia pode ser definida como uma decomposição funcional baseada no fluxo dos dados. Cada processo é definido como uma caixa preta que transforma uma entrada em uma saída; as caixas são ligadas formando um processo computacional [BERGL82]. A Figura 2.3 mostra un diagrama com estes tres elementos.

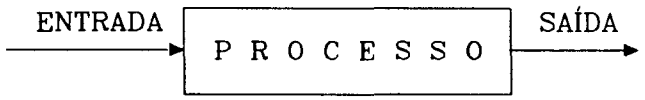

Figura 2.3: Componentes do Modelo de Fluxo de Dados

A estratégia propõe que primeiro se defina o modelo como um diagrama de fluxo de dados; segundo, que se identifiquem os elementos de entrada, saída e transformações; terceiro, que sejam expandidos os processos; quarto, que se identifiquem as entradas, saídas e transformações dos processos expandidos; finalmente, que se refine e otimize o resultado.

As vantagens apontam para uma estrutura de programa sequencial; bom desempenho quando se aplica para desenvolver grandes sistemas, geralmente sistemas comerciais; e facilita o desenvolvimento através da separação do processo de definição do processo de datalhamento da estrutura do pro-

grama. Por outro lado, as desvantagens sugerem que a estrutura final pode resultar em uma rede e não em uma hierarquia; o processo de elaboração do diagrama de fluxo de dados pode ser excessivamente complicado se o sistema for muito grande; finalmente, este método é recente comparado com os outros.

2.2.4 Metodologia que usa Estrutura de Dados

Esta metodologia está baseada na idéia de que um programa deve interpretar o meio ambiente através de suas Estruturas de Dados. Portanto, quando se usa um modelo correto para a estrutura de dados, o programa vai representar o ambiente de uma maneira correta e, desta maneira, o princípio de correspondência fica plenamente satisfeito [JACKS70].

A estratégia da metodologia sugere que primeiro se elabore um diagrama de rede que represente o ambiente do problema; segundo, que se defina e verifique a estrutura do programa; depois, que se definam e distribuam as operações elementares; finalmente, que se equiparem as duas estruturas, a dos dados e a do programa.

Quando o problema que se quer solucionar é muito complexo, Jackson sugere que se elabore uma rede hierárquica onde cada rede represente um programa mais simples. Ou seja, da mesma maneira como na decomposição funcional, usa-se o artifício de subdividir o problema em subproblemas mais simples, para facilitar a sua solução.

Este método tem várias vantagens importantes. Por exemplo, ele permite desenvolver um projeto que modela o mundo real. Segundo, a metodologia permite recursividade na sua aplicação e isto leva a produzir projetos com bom níveis de abstração. Terceiro, ela pode ser ensinada facilmente e, finalmente, ela garante consistência de estruturas quando programadores diferentes a usam para resolver o mesmo problema. Esta metodologia é a que mais se aproxima da que seria considerada à ideal.

As desvantagens mais apontadas são as seguintes: primeiro, ela pode produzir choque de estruturas, ou seja, quando não existe acoplamento entre a estrutura de dados e do programa; segundo, ela não é fácil de ser aplicada em programas grandes; e por último, ela não tem sido usada o tempo suficiente para se fazer uma avaliação mais profunda.

2.2.5 Outros Modelos

Além do modelo clássico de desenvolvimento de Software, existem outros modelos que também são considerados importantes. [DAVIS88] identifica

várias das novas metodologias mais utilizadas, entre as quais encontram-se Protótipos Descartáveis, Desenvolvimento Gradativo, Protótipos Evolutivos, Código Reusável e Síntese Automática de Software. Além disso, ele define um novo paradigma que serve como instrumento de comparação do desempenho de cada metodologia com o desempenho do método clássico, baseando-se nos seguintes aspectos: até que ponto um sistema em operação está atrasado em satisfazer os requisitos do usuário, **"Shortfall"**; quanto tempo uma metodologia demora para satisfazer um requisito, desde o momento em que este aparece até que se torna funcional, **"Lateness"**; com que rapidez um Software pode adaptar-se aos novos requisitos, **Adaptabilidade**; por quanto tempo um sistema pode ser atualizado e manter-se viável, ou seja, quanto tempo ele sobrevive desde o momento em que foi criado até que seja declarado obsoleto, **Longevidade**; finalmente, com que rapidez uma metodologia consegue implementar os novos requisitos **Inadequação**. Davis concluiu que todas as metodologias conseguem melhorar o tempo gasto em implementar uma solução, a capacidade de satisfazer os requistos e a eficiência do Software desenvolvido acima do método clássico. A seguir vamos descrever cada uma das metodologias estudada por [DAVIJ88].

- A metodologia **Protótipos Descartáveis** preocupa-se principalmente em garantir que o Software satisfaça as necessidades do usuário. Portanto, logo no início do processo de desenvolvimento elabora-se uma versão parcial do sistema, a qual é entregue ao usuário para ser usada e testada e, mesmo, para que ele dê seu parecer sobre os pontos positivos e negativos do protótipo. As opiniões do usuário são utilizadas para modificar as especificações originais e, desta maneira, garantir que estas reflitam de uma maneira exata as necessidades do cliente.

- A idéia principal no **Desenvolvimento Gradativo** consiste em construir o sistema gradativamente, ou seja pedaço por pedaço. Em outras palavras, o objetivo é acrescentar funcionalidade e desempenho na medida em que o sistema vai sendo desenvolvido. Sendo assim, o sistema ficará operacional mais rapidamente e, ao mesmo, tempo o desenvolvimento poderá acompanhar qualquer mudança das necessidades do usuário, de uma maneira mais eficiente.

- A metodologia **Protótipo Evolutivo** sugere primeiro que se implementem os requisitos que estão mais claros e foram melhor entendidos, com a intenção de que uma vez implementado o protótipo, aqueles requisitos que não ficaram claros possam ser entendidos. Este é o método

típico de fazer testes: resolver as perguntas mais fáceis primeiro e depois as mais difíceis.

- **O Modelo de Código Reusável** sugere que se utilizem códigos que já foram testados e usados em outros projetos anteriores e, desta maneira, reduzem-se o tempo e os custos de desenvolvimento.

- **A Síntese Automática** sugere a transformação dos requisitos diretamente em código. Primeiramente, os requisitos seriam especificados em uma linguagem formal de alto nível "VHLL", e depois traduzidos para código de máquina. Este modelo reduz grandemente o tempo e custo de desenvolvimento. Entretanto, ele ainda esta em fase experimental.

Pode-se resumir o Ciclo de Vida em quatro partes básicas chamadas de fases de desenvolvimento: Projeto Lógico, Codificação, Teste e Manutenção. Em geral, estas fases incluem outras que contribuem com o que mais tarde se chamará de **Esforço de Desenvolvimento**; este esforço vai variar dependendo do equipamento, tamanho e tipo da aplicação. Segundo [BOEHM81] o esforço aproximado que cada uma destas fases exige é o seguinte:

FASE DO CICLO DE VIDA	% DO ESFORÇO
Projeto	15
Detalhamento	25
Codificação	40
Teste	20

Foi observado que nenhuma das metodologias apresentadas anteriormente está perto de ser uma metodologia completa. Isto é, todas exibem algum tipo de dificuldade, como por exemplo difícil de aprender, não funcionar bem com sistemas de grandes porte, ou não corresponder com a visão do mundo real. Portanto, cabe a seguinte pergunta: Será que é possível desenvolver-se uma metodologia completa? Se a resposta for positiva, quais são as características que esta metodologia deve possuir? Quando se analisam os problemas inerentes a cada metodologia, conclui-se que não há metodologia completa capaz de satisfazer todos os requisitos necessários para o desenvolvimento de Software. Tais requisitos incluem:

- facilidade para definir e elaborar um modelo do problema.
- consistência quando for utilizada por pessoas diferentes. Ou seja, quando dois ou mais programadores implementarem uma mesma solução os projetos finais deveriam ser iguais.
- objetividade quanto a estrutura final. Ou seja, a estrutura deve refletir o mundo real.
- flexibilidade para incorporar mudanças à medida que o programa vai sendo desenvolvido.
- automatizável, ou seja, o homen só interfere na parte criativa do processo, como por exemplo, elaboração da concepção do projeto, elaboração dos algoritmos, etc.
- facilidade de aprendizado.

Capítulo 3

Qualidades de um Software

Já foi dito no Capítulo anterior que um dos objetivos das metodologias para desenvolvimento de Software é produzir programas de boa qualidade. Sistemas bem projetados e de fácil manutenção garantem uma boa qualidade e custos mais reduzidos. As qualidades de um Software variam de um produto para outro. Não existe um padrão de qualidade perfeito com o qual se possa comparar qualquer produto de Software. Na prática, a aplicação do sistema determina a sua qualidade. Por exemplo, os Software que controlam as naves espaciais ou os lançamentos de foguetes nucleares necessariamente têm que ser da mais alta qualidade. Um pacote para calcular salários ou acionar o alarme de uma "agenda pessoal" não precisa de critérios de qualidade igualmente rígidos.

Os aspectos mais importantes de um programa, comumente chamados de Qualidades de um Programa, são:

- **Acoplamento**
- **Confiabilidade**
- **Correção**
- **Eficiência**
- **Facilidade de Manutenção**
- **Flexibilidade**
- **Integridade**
- **Portabilidade**

- **Reusabilidade**
- **Testabilidade**
- **Usabilidade**

Cada uma destas características pode ser avaliada durante ou depois do processo de desenvolvimento do programa. Contudo, não existe um consenso geral sobre como se pode determinar a presença ou ausência delas. No geral, aplica-se a intuição e o bom senso.

[CONTE86] sugere que o ideal seria definir componentes primitivos, ou de mais baixo nível, para cada qualidade, já que seriam mais objetivos e fáceis de avaliar; estabelecer mecanismos para medir ou detectar a presença dos atributos primitivos e, depois, produzir um só resultado que representaria o conjunto de características presentes em um programa. Uma maneira possível de se fazer isto seria a seguinte:

Suponha-se que C_i (i = 1,2,3,...,n) é uma característica de alto nível e P_i o número de características primitivas associadas a cada C_i . Suponha-se também que G_{ij} representa a medida da j-ésima característica de baixo nível associada a C_i. A quantidade G_{ij} pode variar de 0 até 5, sendo 5 o valor máximo possível que uma característica primitiva poderia ter (o valor 5 não é absoluto). Então, poder-se-á definir o valor V_i para a i−ésima característica de alto nível da seguinte maneira:

$$V_i = \sum_{j=1}^{P_i} G_{ij}$$

e o valor total como:

$$VT = \sum_{i=1}^{P_n} \sum_{j=1}^{i} G_{ij}.$$

É possível fazer esta fórmula ficar mais flexível, atribuindo-se um peso p_i tal que:

$$\sum_{i=1}^{n} p_i = n;$$

isto produziria o seguinte valor ponderado:

$$VTp = \sum_{i=1}^{n} p_i \sum_{j=1}^{P_i} G_{ij}.$$

A utilização desta medida (peso) permitiria ao usuário especificar a característica de qualidade que considera mais importante para o seu sistema. Finalmente poder-se-ia calcular o resultado anterior normalizado da seguinte forma:

$$VTpN = \frac{VTp}{VT_{max}}$$

Onde VT_{max} é o valor máximo possível entre todas as características primitivas, onde mais de uma característica pode ter o mesmo valor. Isto é calculado da seguinte maneira:

$$VT_{max} = 5 \sum_{i=1}^{n} P_i.$$

O valor normalizado tem a seguinte propriedade:

$$0 \leq VTpN \leq 1.$$

Quanto mais perto de 1 o valor estiver, maior a qualidade geral do programa. Uma forma de estabelecer um domínio de valores adequados para VTpN seria calcular estes em programas que possuam características conhecidas e aceitas de modo geral, e adotar estes valores como medidas padrão para serem usadas na avaliação de outros Software; os valores padrão poderiam ser ajustados de acordo com o ambiente de cada firma.

3.1 Porque é Importante se Avaliar as Qualidades de um Programa

Imagine-se uma situação em que um usuário quer comprar um pacote estatístico e a firma garante que ele será entregue na data combinada, que o preço do produto é razoável e que o Software executa as funções estatísticas de uma forma eficiente e correta. Estas qualidades não implicam necessariamente que o usuário vai ficar satisfeito com o Software. Há vários motivos que poderiam fazer o cliente desistir de comprar o pacote. Por exemplo, se o Software é difícil de entender ou modificar, será preciso investir fortemente na manutenção do produto e a firma não esteja disposta a fazer este investimento; se o programa é difícil de utilizar, será necessário tempo e dinheiro extra em treinamento; se o pacote apresenta uma dependência muito grande da máquina na qual ele foi desenvolvido, a sua utilização ficará muito restrita. Fica evidente, assim, que a avaliação das qualidades de um programa

é uma necessidade. Isto se torna mais importante à medida em que surgem a cada dia novos sistemas garantindo "satisfazer todas as necessidades do usuário".

3.2 Como se Pode Medir as Qualidades de um Software

O trabalho mais completo apresentado para medir as "Qualidades" de um programa foi feito por Larry Boehm em 1978 [BOEHM78]. Nesse trabalho, ele define uma estrutura clara, simples e bem definida para avaliar as qualidades de um Software e apresenta alguns resultados da aplicação do paradigma em vários produtos comerciais.

Boehm utiliza um conjunto consistente de definições de características de baixo nível que podem ser relacionadas diretamente com um Software. Depois, ele identifica um conjunto de caraterísticas de alto nível, e a cada uma delas associa várias características de baixo nível. Por exemplo, a correção é definida como influenciada pelas características de completeza, consistência e concordância.

As conclusões de seu trabalho são as seguintes: primeiro, quando se presta atenção explícita às características de qualidades dos Software, é possível diminuir os custos do Ciclo de Vida; e segundo, o estado da arte impõe limitações específicas na habilidade de se avaliar quantitativa e automaticamente as qualidade de um Software. Além disto, ele define e classifica um grande número de métricas para medir qualidade, em relação aos benefícios potenciais que elas representam, de quantificação e automação; finalmente, Boehm identifica várias atividades particulares do Ciclo de Vida dos Software que influenciam significativamente a sua qualidade.

3.3 Atributos de Qualidade de um Software

Seguindo o método de Boehm, define-se um conjunto de atributos primitivos ou de baixo nível que podem ser verificados diretamente a partir do Software. Os seguintes atributos servirão de base para definir as características de qualidade de nível mais alto:

- **Coerência da Comunicação** - garante o uso padronizado de protocolos e rotinas de interfaces.

- **Coerência dos Dados** - garante que a representação de dados usada seje padronizada.
- **Clareza** - garante que o programa contém comentários internos que explicam a sua função.
- **Complexidade** - afeta negativamente a clareza do produto.
- **Comunicabilidade** - garante o monitoramento do programa e a identificação dos erros.
- **Concisão** - garante que a implementação do projeto foi realizada usando a menor quantidade possível de códigos.
- **Consistência** - garante que foram utilizadas técnicas compatíveis nas fases de projeto, implementação e documentação.
- **Concordância** - garante a relação entre os requisitos especificados e a implementação.
- **Eficiência na Execução** - garante que o programa tem um tempo de execução mínimo.
- **Exatidão** - garante que os cálculos e os resultados da saída são precisos.
- **Expansão** - garante que tanto os dados como o programa possam ser expandidos.
- **Facilidade de Uso** - garante facilidade ou dificuldade de uso do programa.
- **Facilidade de Verificação** - permite a avaliação do Software, dos dados e dos resultados.
- **Generalização** - garante que as funções desenvolvidas para o Software possam ser reutilizáveis.
- **Independência do Hardware** - garante que o programa pode ser executado em máquinas que possuam configurações de Hardware diferentes.
- **Independência do Sistema de Software** - determina a independência do programa em relação ao ambiente de Software em que ele funciona.

- **Instrumentação** - garante que o uso do Software pode ser medido e permite a quantificação dos erros.
- **Integridade** - garante que foram implementadas todas as funções necessárias para satisfazer os requisitos.
- **Modularização** - garante que o Software tem uma estrutura de controle composta de módulos altamente independentes.
- **Segurança** - garante o controle e proteção do programa e dos dados.
- **Simplicidade** - garante que a implementação foi feita buscando a maneira mais fácil possível de ser entendida.
- **Tolerância de Erros** - garante que o Software funcione em circunstâncias adversas.
- **Transição** - garante que a transferência do Software do ambiente em que ele foi desenvolvido para o ambiente do usuário seja viável.

3.4 Qualidades de um Software

O Software é de qualidade se ele tem as seguintes propriedades:

- **Acoplamento** - O acoplamento refere-se ao esforço que um programa ou sistema exige para ser interligado a outros sistemas. Ele é função da Coerência da Comunicação, Coerência dos Dados, Generalização e Modularização, e deve ser incorporado ao programa nas fases de especificação e de projeto. Esta característica torna-se mais importante a cada dia pois há uma forte tendência para a utilização de redes de computadores, que permitem a comunicação e o compartilhamento de informações; por exemplo, estão se tornando cada vez mais comuns os sistemas de banco de dados, que se comunicam com pacotes gráficos, editores de texto e planilhas eletrônicas.
- **Confiabilidade** - Esta qualidade indica até que ponto pode-se esperar que um Software funcione sem falhar. Seis critérios essenciais determinam esta qualidade: Precisão, Complexidade, Consistência, Tolerância de Erros, Modularidade e Simplicidade. Esta característica é especificada logo no início do processo de desenvolvimento, e depois é concretizada na fase de codificação.

- **Correção** - Indica até que ponto o Software satisfaz os requisitos. Ela depende de três críterios principais: Completeza, Consistência e Concordância. As metodologias para desenvolvimento sugerem que a verificação da correção do Software comece a partir das fases de especificação e projeto. Na prática, a correção do programa é determinada nas fases de teste e manutenção.

  O método principal para se tentar obter a correção de um programa consiste no monitoramente dos erros e omissões, através do uso de ferramentas auxiliares. O resultado deste monitoramento é usado também para calibrar a metodologia utilizada para desenvolver um determinado Software. Na fase de codificação, são utilizados compiladores e analisadores para a otimização do código. Contudo, a correção é uma das mais importantes e difíceis tarefas e, em alguns casos, impossível de se garantir. Isto se deve a que, muitas vezes, alguns erros só aparecem quando o sistema está em operação. Vale a pena lembrar que o projeto espacial americano Apollo, que levou o primeiro homem à lua, quase fracassou devido a uma falha de Software. O sistema da nave estava tentando calcular a aterrissagem e a viagem de volta ao mesmo tempo e sobrecarregou-se, obrigando um dos astronautas a tomar o controle da nave e fazer uma aterrissagem manual.

- **Eficiência** - A eficiência refere-se à eficácia com que um Software utililiza os recursos de máquina para executar a sua tarefa.

  Os grandes avanços na tecnologia para fabricação de Hardware tornaram possível a construção de máquinas rápidas, com memória expandível e dispositivos eficientes, diminuindo um pouco a importância deste atributo. Isto fez com que, em muitas ocasiões, outras qualidades, como a facilidade de manutenção, por exemplo, sejam consideradas mais importantes que a eficiência.

  A eficiência geral de um programa ou sistema de Software pode ser avaliada com base em três critérios: Concisão, Facilidade de Uso e Eficiência da Execução. Atualmente utilizam-se compiladores que determinam os pontos de ineficiência de um programa, otimizam código e contêm uma ampla biblioteca de funções escritas visando eficiência máxima na execução. Outras ferramentas, tais como analisadores dinâmicos e estáticos de programas, verificam o fluxo de controle assim como a definição dos dados.

- **Facilidade de Manutenção** - Esta característica preocupa-se com os efeitos que os erros produzem num programa e com o esforço para localizá-los e corrigí-los. Ela depende de seis outros atributos de programas que são: Concisão, Consistência, Instrumentação, Modularização, Documentação e Simplicidade. Há muitas pesquisas em andamento para identificar e entender as caraterísticas que dificultam a manutenção de um sistema, visando diminuir as suas chances de serem incorporadas a projetos futuros.

  [DAVIJ88] afirma que a complexidade é uma das caraterísticas que mais prejudicam a Manutenção de Software. Ele propõe a utilização de métricas para medir a complexidade e poder controlá-la.

  A facilidade de Manutenção está intimamente ligada á confiabilidade, e os atributos que afetam a uma também afetam a outra.

- **Flexibilidade** - Esta qualidade refere-se a facilidade com que um sistema permite ser modificado. Em outras palavras, trata-se do esforço necessário para incorporar funcionalidade adicional a um Software. Ela está intimamente ligada à fase de manutenção do Ciclo de Vida, pois de 80% a 90% do trabalho feito nesta fase consiste na atualização de rotinas [CONTE86] [JAY85]. Como a fase de manutenção é a mais cara do processo de desenvolvimento de Software o seu custo pode ser reduzido desenvolvendo Software mais flexiveis pois quanto mais flexível um sistema mais fácil será mantê-lo.

  A flexibilidade depende de outros atributos de mais baixo nível tais como: Complexidade, Concisão, Consistência, Expansão, Generalização, Modularização, Documentação Própria e Simplicidade.

- **Integridade** - Indica até que ponto os dados e os programas estão protegidos contra operações críticas ou usuários não autorizados. Ela depende das qualidades de baixo nível: Inspeção, Instrumentação e Segurança.

  A medida em que aumenta a complexidade de um sistema, este se torna cada vez mais difícil de ser controlado. Isto implica que sistemas de grande porte precisam incorporar mecanismos para se proteger e controlar o acesso aos dados e ao próprio sistema.

- **Portabilidade** - Esta qualidade refere-se ao esforço necessário para transferir um programa de uma máquina para outra. Ela depende do

Ambiente de Software, Modularização e Documentação. Esta característica ganhou importância devido a grande diversidade de máquinas com configurações diferentes, e à necessidade de se desenvolver produtos que funcionem em todas elas. Os dois fatores mais influentes na portabilidade de um programa são a dependência do código do Hardware e do ambiente de Software. Outros fatores incluem as diferenças nos dipositivos de Entrada e Saída e as extensões especiais de uma determinada linguagem.

- **Reusabilidade** - A presença desta qualidade num Software facilita o seu uso em outras aplicações diferentes daquela para a qual ele foi desenvolvido. Ela depende de outros dois atributos: Documentação e Independência do Ambiente de Software. O desenvolvimento de um código reutilizável permite agilizar o processo de desenvolvimento e ao mesmo tempo simplificar a manutenção. Um módulo reusável implica em um módulo fácil de entender, manter e modificar. Em outras palavras, o seu projeto lógico tende a ser genérico, sem mencionar que a correção de erros e a atualização de códigos reusáveis é enormemente facilitada.

  [MEYER87] afirma que a forma mais eficiente de se construir programas reusáveis é mediante o uso da técnica da Programação Orientada por Objeto; segundo ele, as propriedades de abstração de dados e herança fazem desta técnica a mais apropriada.

- **Testabilidade** - Refere-se ao esforço necessário para avaliar a estrutura e a correção de um sistema. Ela depende de seis características: Inspeção, Complexidade, Instrumentação, Modularização, Documentação e Simplicidade. É importante que a flexibilidade para se testar programas seja garantida nas fases de Projeto e Codificação, pois esta qualidade é difícil de ser incorporada no final do processo de desenvolvimento.

- **Usabilidade** - Refere-se ao esforço que um programa ou sistema de Software requer para ser usado. Ela depende de dois critérios: Facilidade de Uso e Transição. Para muitos engenheiros de Software, esta característica tem se tornado uma das mais importantes pois, se um programa é confiável, fácil de manter, flexível, eficiente, e mesmo assim o usuário comum não consegue usá-lo, então o Software não é util.

O objetivo do Engenheiro de Software é desenvolver sistemas que possuam estas características de qualidade. A relação entre os atributos de

baixo nível e as características de alto nível pode ser resumida da seguinte maneira:

- **Acoplamento** (Coerência de Comunicação, Coerência dos Dados, Generalização, Modularização).
- **Confiabilidade** (Complexidade, Consistência, Modularização, Precisão, Simplicidade)
- **Correção** (Completeza, Concordância, Consistência,)
- **Eficiência** (Concisão, Eficiência da Execução, Facilidade de Uso)
- **Facilidade da Manutênção** (Complexidade, Concisão, Consistência, Documentação Própria, Instrumentação, Modularização, Simplicidade)
- **Flexibilidade** (Complexidade, Concisão, Consistência, Documentação Própria, Expansão, Generalização, Modularização, Simplicidade)
- **Integridade** (Inspeção, Instrumentação, Segurança)
- **Portabilidade** (Documentação Própria, Generalização, Independência do Ambiente de Software, Independência do Hardware, Modularização)
- **Reusabilidade** (Documentação Própria, Generalização, Independência do Ambiente de Software, Independência do Hardware, Modularização)
- **Testabilidade** (Complexidade, Documentação Própria, Inspeção, Instrumentação, Modularização, Simplicidade).
- **Usabilidade** (Facilidade de Uso e Transição).

æ

Capítulo 4

Métricas para Software

Conforme definido no Capítulo 1, o termo métrica refere-se a um instrumento, método ou técnica através do qual pode-se avaliar o grau de presença de uma determinada característica em um Software. Há uma grande quantidade de métricas disponíveis que medem desde o esforço para entender um programa até o seu número de erros. Este trabalho preocupa-se com aquelas métricas que se relacionam com a qualidade de um Software. Isto se deve a que numerosos estudos, [BOEHM76, CONTE86, HALST76, HARRI82, JAY81, LI87] entre outros, apontam uma forte relação entre a complexidade de um programa e suas características, como por exemplo flexibilidade, facilidade de manutenção e confiabilidade. O objetivo desta tese consiste em desenvolver uma ferramenta automática baseada nestes tipos de métricas. A seção que se segue define formalmente o que é Complexidade de um Software.

4.1 Complexidade de um Programa

A complexidade de um programa pode ser avaliada de várias maneiras. Existem dois tipos de complexidade: lógica e computacional. [CURTI79] define a complexidade lógica de um Software como as características que o tornam difícil de ser entendido e modificado. A complexidade computacional foi definida por [RABIN77] como aquela que se refere a eficácia com que um algoritmo utiliza-se dos recursos da máquina em que ele foi implementado. Uma outra definição que engloba as duas anteriores foi elaborada por Curtis e apresentada por [CONTE86]. Segundo esta definição, a complexidade é uma característica da interface do Software; esta influencia os recursos que

um sistema utiliza ou reserva para a sua interação com o Software. Esta definição implica que a complexidade é uma função do próprio sistema e de sua interação com outros sistemas, sejam eles máquinas, usuários ou outros programas. Neste trabalho, o termo complexidade refere-se de agora em diante àquela característica que afeta o desempenho de um programador a desenvolver, comprender ou modificar um programa.

A complexidade que acaba de ser definida pode ser encontrada em níveis diferentes, como por exemplo em nível do problema, do projeto, do programa ou do sistema. Para cada um destes níveis têm sido apresentada métricas, como por exemplo [HARRIS82,KAFUR87,DAVIJ88], que buscam medir de uma forma precisa a dificuldade que o programador ou analista encontra para realizar tarefas como projetar, programar, testar ou dar manutenção a um Software. De uma forma geral, pode se-dizer que se uma tarefa é mais difícil que outra, então ela exige mais habilidade e esforço para ser executada.

A importância da complexidade dos Software é mostrada na figura 4.1 retirada de [LI87].

A seção seguinte apresenta as métricas mais importantes e usadas para medir complexidade de código.

4.2 Métricas para Medir Complexidade

Um dos problemas que mais afetam o desenvolvimento de Software é o alto custo da manutenção. Boehm afirma que esta fase do Ciclo de Vida exige entre 40% e 70% do orçamento das firmas que desenvolvem Software [BOEHM81]. Estudos mais recentes entre eles [HARRI82, JAY85, LEW88] têm demonstrado que a utilização de métricas para medir complexidades representa uma alternativa para ajudar a resolver o problema. Harris afirma que um programa difícil de modificar ou manter requer mais tempo e esforço e, portanto, implica um custo alto [HARRI82]. A abordagem por ele proposta consiste em estudar os atributos que tornam um programa complexo; implementar técnicas para evitar ou controlar esses atributos; desenvolver Software que sejam fáceis de manter, para desta maneira, diminuir os custos da manutenção. Segundo Henry, a complexidade de um Software está relacionada com o tamanho, a estrutura lógica e a sua interface [HENRY81]. As métricas que se seguem estão classificadas segundo estes parâmetros.

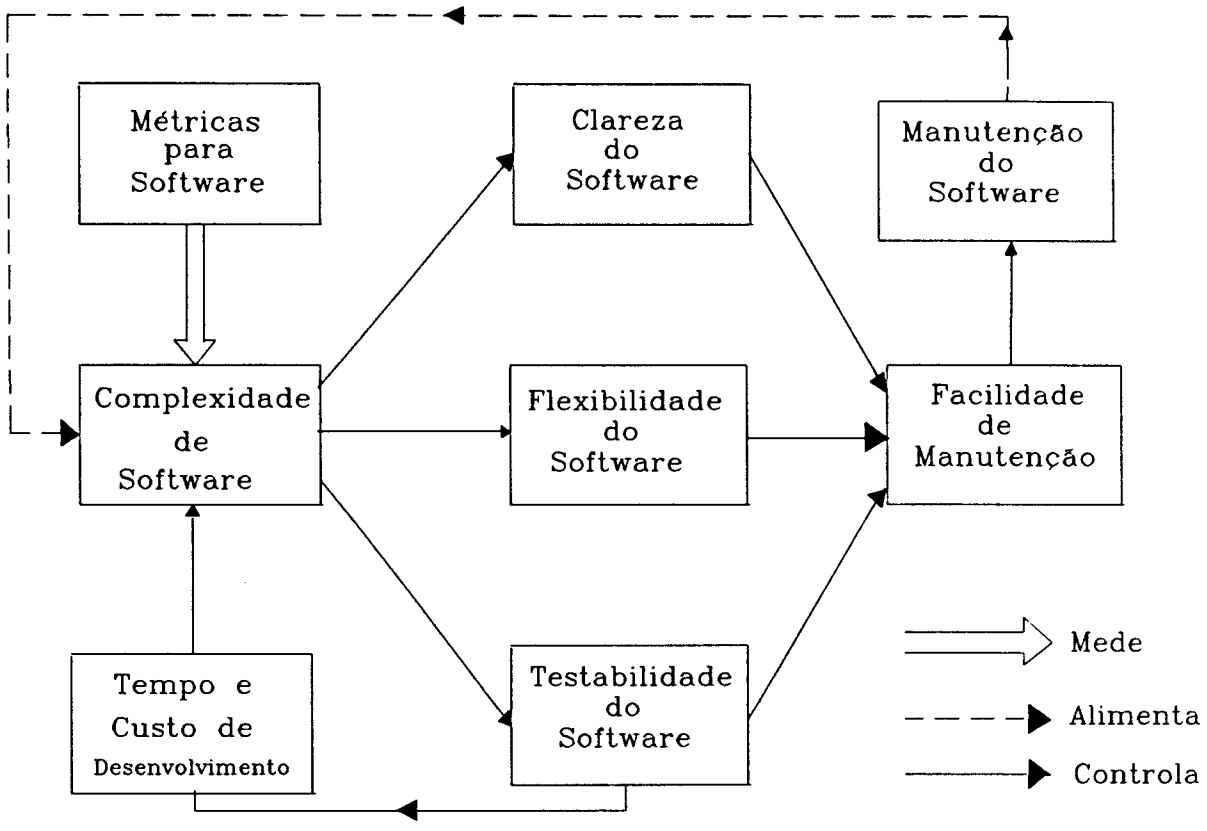

Figura 4.1: Importância da Complexidade de Software

4.2.1 Métricas do Tamanho

O tamanho é uma característica de todo programa, independentemente da linguagem em que ele for implementado. As métricas de tamanho, como seu nome indica, são aquelas que quantificam a magnitude de um programa. Atualmente existem várias maneiras de medir o tamanho de um programa, porém as três mais populares são: Número de Linhas de Código (LOC), Número de Símbolos e Número de Funções ou Módulos. A importância dessas métricas deve-se, em parte, ao fato de que, primeiro estas são fáceis de calcular depois de terminado o programa; segundo, o tamanho de um programa é um dos fatores mais importantes para estimar os custos de seu desenvolvimento, por exemplo, todos os 19 modelos para estimativa de custos de um Software analisados em [MOHAN81] utilizam o tamanho como parâmetro. E finalmente, a produtividade dos programadores muitas vezes é avaliada pelo tamanho. Como um grande número de pesquisadores concorda que o esforço para construir um programa depende em grande parte do número de linhas escritas, a métrica do tamanho torna-se um fator importante quando se quer fazer um estudo deste aspecto. É evidente que em um mesmo programa algumas linhas são mais difíceis de construir do que outras; portanto, o número de linhas não é a métrica mais adequada, apesar de ser a mais usada, pois esta atribui o mesmo peso para todas as linhas. Hoje em dia existem duas tendências: a primeira consiste em contar o número de símbolos básicos e ignorar a linha, e a segunda consiste em agrupar as linhas em funções.

Linhas de Código

O número de linhas de código é a métrica mais conhecida atualmente. Contudo, apesar de sua simplicidade, não há uma regra geral que estabeleça como ela deva ser obtida. Por exemplo, comentários e as linhas em branco não devem ser incluídas na contagem total do número de linhas, já que não afetam a execução do programa e não são tão difíceis de construir como as linhas executáveis. [CONTE86] define uma linha de código da seguinte maneira: "Uma linha de código é qualquer linha do programa que não é um comentário ou uma linha em branco, sem importar o número de comandos ou fragmento de comandos na linha". Isto inclui todas as linhas que contêm cabeçalhos, declarações e comandos executáveis e não executáveis.

Esta métrica apresenta vários problemas. Entre outros, ela não é de aplicação geral, ou seja, programadores utilizando linguagens diferentes, por exemplo Pascal e Assembler, não podem ser comparados em termos de pro-

dutividade. O programador que usa Assembler, por exemplo, receberia um grau muito mais alto pois essa linguagem, por ser ela de baixo nível, associa uma linha de código para cada instrução, e isto faz com que os programas em Assembler possuam uma grande quantidade de linhas de código. Um outro problema é que esta métrica não pode ser usada para avaliar as outras fases do processo de desenvovimento de Software que não involvem codificação. Além disso, a métrica incentiva os programadores a produzir mais código, o que causa um aumento da complexidade e diminuição da qualidade. Portanto, hoje em dia recomenda-se que a métrica de linha de código seja usada apenas como um instrumento que provê informação adicional sobre o processo de desenvolvimento.

Número de Funções

Pode-se definir uma função como um conjunto de comandos que realiza uma tarefa específica em um programa. A idéia de usar o número de funções como uma métrica de tamanho surge do fato de que um grande número de programadores desenvolve seus programas a partir de funções. [COULT83] afirma que esta forma de grupar conceitos é o que em psicologia chama-se de pedaços, "Chunk".

O tamanho de uma função é geralmente pequeno e isto se deve, em grande parte, à limitação que o indivíduo tem para processar informações. Uma pessoa não consegue manipular uma grande quantidade de informações eficientemente. Por isto, é comum que a maioria das metodologias recomende que os programas sejam divididos em partes menores chamadas de módulos ou funções.

Uma das utilidades desta métrica é que ela pode ser usada como parâmetro para medir a produtividade de um programador. Cada função pode ser vista como um aspecto do programa que o usuário deseja. A contagem das funções é viável porque elas são um resultado concreto do processo de desenvolvimento.

Há duas maneiras de contar o número de funções. A primeira consiste em contar as funções definidas no programa ou código. Funções na linguagem Pascal podem ser identificadas facilmente porque todas têm a mesma estrutura de chamada. A segunda forma de contar as funções baseia-se nos aspectos externos dos programas tais como entrada do usuário, transações, saídas, arquivos utilizados e interfaces [JAY85]. Esta segunda maneira leva em consideração a complexidade que estes componentes acrescentam a um Software.

Índice de Pontos de Função

Esta métrica, apresentada em [ALBRE83], tem por objetivo avaliar a complexidade de um sistema de Software através de cinco fatores: **Entradas, Saídas, Transações, Arquivos Mestres** e as **Interfaces Externas**. Estes aspectos de um sistema são ponderados, ajustados de acordo com o ambiente de desenvolvimento e os requisitos do usuário e depois somados para obter uma medida chamada de Ponto de Função, "function point". A seguir encontra-se um exemplo de pesos usados na ponderação de cada fator:

Elemento 1 = Número de entradas do usuário

Simples * 3

Médio * 4

Complexo * 5

Elemento 2 = Número de saídas do usuário

Simples * 4

Médio * 5

Complexo * 7

Elemento 3 = Número de transações do usuário

Simples * 3

Médio * 4

Complexo * 6

Elemento 4 = Número de arquivos

Simples * 7

Médio * 10

Complexo * 15

Elemento 5 = Número de interfaces externas

Simples * 5

Médio * 7

Complexo * 10

Os valores correspondem aos pesos, determinados através da experiência com casos reais, que os usuários forneceram para cada um dos fatores de desenvolvimento. A soma destes pontos é ajustada para determinar a complexidade final. O grau de complexidade pode aumentar ou diminuir o valor de pontos da função em 35%.

Uma vez determinados os fatores que afetam a complexidade, como por exemplo reusabilidade do código, conversão e instalação do sistema ou digitação dos dados "On line", entre outros, o grau de influência de cada fator é avaliado de 0(zero) a 5(cinco) onde 0(zero) denota nenhuma influência e 5(cinco), denota que o fator é de influência essencial. A soma destes fatores representa o grau de influência que é usado para ajustar a complexidade e obter os pontos de função da seguinte maneira:

Grau Total de Influência (N)

Ajuste da Complexidade = (0.65 + 0.01 * N)

Ponto de Função = (Ponto de Função sem ajustar) * (Complexidade)

Pontos de função são utilizados para medir a produtividade, sendo comparados com o custo do Software. Caso sejam consideradas como esforço as horas gastas no desenvolvimento do sistema, então a produtividade pode ser calculada da seguinte maneira:

Produtividade = Pontos de Função / Esforço.

A métrica pontos de função provê um mecanismo que permite a avaliação e comparação da produtividade de projetos desenvolvidos com diferentes linguagens de programação, ou que utilizam tecnologias diferentes. Além disso, as medidas podem ser obtidas no final da fase de especificação funcional, e isto permite estimar o custo de desenvolvimento. Esta métrica pode também ser facilmente usada pelo usuário sem treinamento profissional na área, para avaliar o sistema. Um exemplo de sua implementação é a ferramenta automática PAFUNCio [MEDEI88]. Em [SYMON88] é feita uma análise da métrica ponto de função, na qual são tratadas algumas de suas dificuldades e sugeridas possíveis melhorias.

Número de Símbolos

Algumas linhas de código de um programa são mais difíceis de construir e, portanto, mais complexas do que outras. Como já foi ressaltado, a métrica linha de código não leva isto em consideração. A métrica número de símbolos, apresentada por [HALST76], faz parte de uma família de métricas chamada de "Software Science" baseada na contagem de todos os símbolos de um programa, classificados em operadores e operandos. Entende-se por símbolo uma unidade sintática básica distinguível por um computador [CONTE86]. "Software Science" interpreta um programa de computador como uma coleção de símbolos que podem ser separados em operadores e operandos. [SHEN83] definiu a teoria de "Software Science" da seguinte maneira: "A maioria dos programas são produzidos por programadores trabalhando concentradamente através de um processo de manipulação de operadores e operandos únicos".

Para fazer sua formulação, Halstead baseou-se no fato de que todo programa pode ser reduzido a uma sequência de instruções em linguagem de máquina, composta de um operador e um ou vários endereços de operadores. No geral, qualquer símbolo ou palavra-chave que especifica uma ação é considerado como um operador, e qualquer símbolo usado para representar

dados é considerado um operando. Símbolos de pontuação, como ponto e vírgula ";", vírgula "," e ponto"." também estão incluídos na categoria de operadores. São operadores os símbolos aritméticos "+", "-", "*" e "/", nomes de comandos tais como REPEAT, WHILE, READ, FOR, etc., símbolos especiais como ":=", "()" e "[ ]", e inclusive nomes de funções como EOF.

Halstead chegou a interessante conclusão de que, conhecendo o número de operadores, operandos e a freqüência com que estes aparecem em um programa, é possível determinar um conjunto de características tais como tamanho, volume, nível de dificuldade, nível da linguagem, etc.. Esta análise é relativamente simples e está baseada em uma contagem que pode ser feita automaticamente [HALST77].

A teoria "Software Science" apresenta algumas dificuldades. Primeiro, não existe consenso geral entre os pesquisadores sobre as maneiras mais significativas de classificar e contar os símbolos. Em outras palavras, cada pesquisador define a sua estratégia para construir o algoritmo de análise. A segunda crítica, por sua vez, refere-se ao fato de que essa teoria depende da Linguagem de Programação utilizada, porque cada linguagem tem o seu próprio conjunto de símbolos. Além disso, na contagem dos operadores e operandos não repetidos (únicos), frequentemente acontecem ambigüidades. Por exemplo, em Pascal o operador () é usado tanto na definição de conjunto (e.g., carros= (gol,monza,fiat)) como na chamada de funções (e.g. sin(x)). A crítica mais importante, refere-se à falta de base matemática, pois Halstead baseou- se em sua intuição. Na prática, porém, a teoria "Software Science" tem demonstrado grande sucesso.

As métricas mais relevantes associadas à contagem do número de símbolos são divididas em simples e complexas.

Métricas Simples

São aquelas obtidas diretamente do código e são as seguintes:

1. η_1: número de operadores únicos
2. η_2: número de operandos
3. N_1: número total de ocorrência dos operadores
4. N_2: número total de ocorrência dos operandos

O tamanho do vocabulário representado pelo número de elementos únicos que compõem um programa é definido como:

$$\eta = \eta_1 + \eta_2$$

e o comprimento do programa em relação ao número total de elementos é dado por

$$N = N_1 + N_2$$

Métrica Complexas

Estas métricas são obtidas a partir das métricas simples e são fáceis de calcular.

1. Tamanho de um Programa

Aplicando a intuição, as métricas básicas e alguns conceitos da termodinâmica, Halstead determinou que o tamanho de um programa pode ser estimado com a seguinte relação:

$$\hat{N} = \eta_1 * log_2(\eta_1) + \eta_2 * log_2(\eta_2)$$

O símbolo (^) que aparece acima de algum dos carateres indica que a métrica é estimada. O comprimento de um programa é função do número de operadores e operandos não repetidos que aparecem nele. Apesar de Halstead não ter apresentado derivações matemáticas rigorosas para suas relações, resultados práticos verificam ser esta uma métrica válida e útil para diversos ambientes [FITS78] [SHEN83].

Experimentos adicionais confirmaram que existe uma correlação grande entre $\hat{N}$, η_1 e η_2, que se torna mais forte quando o programa está livre de impurezas [BAKER80] [CURTI79] [WOODF79].

2. Volume

O volume de um programa, em um dado momento quando se utiliza um código binário para o vocabulário, é definido como

$$V = N * log_2(\eta)$$

onde N representa o comprimento do programa e $log_2\ \eta$ o número de bits necessários para armazená-lo. Este volume também pode ser interpretado

como o número de comparações necessárias para desenvolver um programa de comprimento N, supondo que se utilize um método de busca binária para selecionar um elemento do vocabulário de tamanho η. O volume vai depender da linguagem de implementação, e aumenta quando o programa é traduzido de uma linguagem de alto nível para uma outra de nível mais baixo.

3. Volume Potencial

[HALST76] afirmava que a implementação mais reduzida de um algoritmo é feita através de uma chamada a um procedimento que já existe. Tal implementação só requer uma invocação do nome e dos parâmetros para Entrada e Saída. Portanto, o volume potencial V^* de um algoritmo implementado como uma chamada de procedimento pode ser expresso da seguinte maneira:

$$V^* = (2 + \eta_2^*) * log_2(2 + \eta_2^*)$$

onde o 2 é obtido através da soma dos seguintes operadores: nome do procedimento, incluindo os parênteses que englobam os operandos, e o símbolo de atribuição; η_2^* representa o número de parâmetros de Entrada e Saída do procedimento. Por exemplo, suponha-se que é chamado o procedimento ORDENA da seguinte maneira: ORDENA(x,n) onde **x** representa o vetor a ser ordenado e **n** o número de elementos. Como o vetor contem ambos os elementos a serem ordenados e depois estes mesmos elementos já ordenados, ele deve ser contados duas vezes. Ou seja, n^* teria, neste caso, o valor 3.

4. Nível de um Programa

Halstead definiu o nível de um programa como:

$$L = \frac{V^*}{V}$$

em outras palavras, o nível de um programa equivale à proporção entre o volume potencial (V^*) e o volume (V). Halstead afirmava que o nível aumenta quando o número de operandos (η_2) aumenta, e diminui quando o número de operadores (η_1) e a sua frequência (N_1) diminuem. Isto o levou a formular a seguinte expressão para estimar o nível do programa:

$$\hat{L} = (\frac{2}{N_1}) * (\frac{\eta_2}{N_2})$$

5. Nível de Dificuldade

A medida em que o volume (V) de uma aplicação aumenta, o nível do programa (L) diminui e a dificuldade (D) aumenta, pode-se definir o nível de dificuldade (D) como:

$$D = \frac{V^*}{V} = \frac{1}{L}$$

É fácil provar que o uso redundante de operandos, ou a ausência de estruturas de controle de alto nível, contribuem para aumentar esta métrica do volume e, portanto, a sua dificuldade.

Como o volume potencial não é muito fácil de calcular, Halstead apresentou uma fórmula para estimar o nível de dificuldade:

$$\hat{L} = \frac{1}{\hat{D}} = (\frac{2}{\eta_1}) * (\frac{\eta_2}{N_2})$$

A justificativa desta relação é intuitiva. Quando o número de operadores (η_1) de um programa aumenta, ou quando se utilizam operandos de uma forma redundante, então o nível do programa (L) também aumenta. E quando (L) aumenta então (D) diminui.

6. Nível da Linguagem

Para [HALST76], era possível atribuir níveis às linguagens, de uma forma linear e contínua, usando a contagem do uso de operadores e operandos. Com a definição desta métrica, seria possível selecionar a linguagem mais apropriada para futuras aplicações, testar a capacidade potencial de uma determinada linguagem, e inclusive, testar o esforço relativo para produzir programas equivalentes usando linguagens diferentes.

A hipótese de Halstead foi a seguinte: se a linguagem é mantida constante e o algoritmo é modificado, à medida em que o volume potencial (V^*) aumenta, o nível do programa (L) diminui proporcionalmente, de tal forma que o produto ($L * V^*$) permanece constante. Ele chamou este produto de nível da linguagem:

$$\lambda = L * V^*$$

esta relação representa a limitação inerente a uma dada linguagem.
Como

$$V = L * V^*$$

então

$$\lambda = L * V^2$$

[SHEN83] afirma que a relação apresentada acima poderia ser útil para comparar linguagens de programação, caso o mesmo conjunto de problemas seja implementado em linguagens diferentes pelo mesmo programador.

7. Esforço

Segundo Halstead, a dificuldade de programar aumenta quando o volume do programa aumenta e diminui quando o nível do programa aumenta. A partir desta observação, ele apresentou a função esforço (E):

$$E = \frac{V}{L}$$

e o esforço para compreender um programa

$$E_c = \frac{\eta_1 * N * N_2 * log_2(\eta)}{2 * \eta_2}$$

onde E_c significa uma medida do esforço mental necessário para criar um determinado programa. (E) representa o número de comparações mentais que um programador, trabalhando concentradamente, deveria fazer para implementar um determinado algoritmo. Em termos práticos, isto quer dizer,por exemplo, que um projeto seria implementado mais rápido, e com menos erros, se fosse feito em Pascal do que em uma linguagem de baixo nível.

8. Tempo

Em um estudo feito pelo psicólogo Strout e citado em [HALST77], é sugerido que a mente humana é capaz de fazer um número limitado de raciocínios por segundo, "Elementary Mental Discriminations". Segundo Strout, esse número, β, chamado de o número de Strout, fica entre 5 e 20. Baseado nesta e, outras conclusões, Halstead propôs a seguinte relação:

$$T = \frac{E}{\beta}$$

onde β é uma constante que representa a velocidade do programador; isto é, o número de discriminações por segundo de que este seria capaz. Estudos posteriores, realizados por ele próprio, apontaram $\beta = 18$ como um valor razoável. Mais tarde, T foi também relacionado com o tempo que um programador experiente demoraria em ler e entender um programa. [COULT83] fez uma crítica interessante acerca desta parte da teoria. Ele argumenta que o número de Strout foi originalmente aplicado à memória media, ou um dos três níveis de memória segundo os modelos que tentam explicar o seu funcionamento, e que, portanto, estes resultados não podem ser generalizados e aplicados à função de programar e entender um algoritmo.

Para estimar o tempo que um programador experiente demoraria em comprender um programa utilizando o esforço (E_c), Halstead propôs

$$\hat{T}_c = \frac{E_c}{\beta} = \frac{\eta_1 * N * N_2 * log_2(\eta)}{2 * \eta_2 * \beta}$$

4.2.2 Métricas da Estrutura Lógica

A parte de um algoritmo relacionada com o teste e decisões, depois do teste, chama-se estrutura lógica de um programa. Dentro do grupo de métricas que medem a estrutura lógica de um programa pode-se citar aquelas que contam o número de decisões, como por exemplo, a Complexidade Ciclomática de McCabe; aquelas que medem o nível de aninhamento dos comandos, e aquelas que medem o uso de transferência de controle. A Complexidade Ciclomática é considerada a melhor, mais sofisticada e mais conhecida [CONTE86, CURTI79, JAY85]. A seguir são discutidas métricas da estrutura lógica.

Contagem de Decisões do Programa

O fluxo de controle de um programa é seqüencial a menos que exista uma interrupção ou desvio. Existem três tipos de desvios: **desvio para trás, desvio para frente** e **desvio horizontal**. Estes desvios, no geral, acontecem depois de ser executado um comando condicional, com exceção de "GOTO". Cada vez que este último comando aparece no programa o desvio é executado incondicionalmente.

Um desvio para frente acontece depois que um comando condicional é executado, criando-se duas ou mais alternativas de ação. A Figura 4.2 representa graficamente este tipo de desvio.

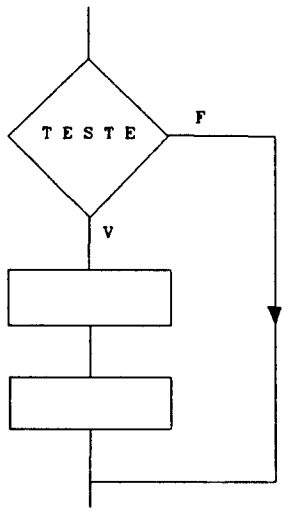

Figura 4.2: Desvio para Frente

O segundo tipo de desvio, para trás, é frequentemente usado para criar repetições de um bloco de código, laço ou "Loop". Este desvio pode ser incondicional ou condicional, quando se executa um teste que permite a repetição ou conclusão do laço. As Figuras 4.3 mostram exemplos deste tipo de desvio.

O desvio horizontal refere-se a uma chamada de procedimento ou subrotina e, normalmente não é considerado um desvio, pois depois da sua execução o controle deve ser retornado para a instrução que segue o desvio. A figura 4.4 seguinte ilustra este tipo.

A métrica contagem de decisões consiste em contar todas as instruções de

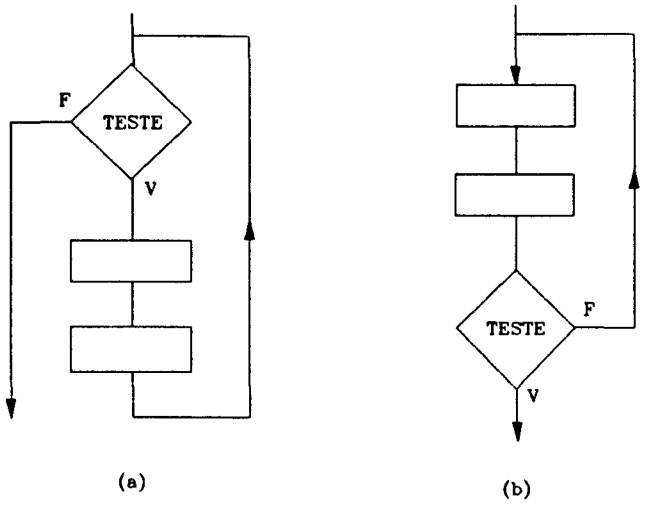

Figura 4.3: Desvios para Trás

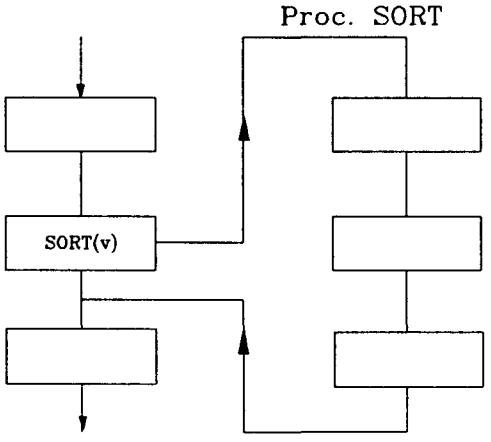

Figura 4.4: Desvio para Horizontal

desvio como IF, WHILE, CASE, REPEAT, FOR para a linguagem Pascal e todas as outras instruções que servem para controlar os laços. A idéia principal é que quanto maior o número deste tipo de instruções em um programa, maior a sua complexidade.

A maioria das linguagens de programação permite o uso de instruções condicionais compostas do tipo

IF condição1 AND condição2 THEN açao.

Esta instrução é equivalente a

IF condição1 THEN IF condição2 THEN ação

ou

IF condição2 THEN IF condição1 THEN ação

O mesmo acontece com o operador lógico OR. Pode se observar que a utilização de cada operador lógico acrescenta um ao número de decisões. [CONTE86] recomenda que, em lugar de contar as palavras chaves de decisão como IF, FOR, WHILE, etc., sejam contados os operadores lógicos AND e OR. A métrica Contagem de Decisões mede a complexidade de um procedimento de uma maneira mais precisa do que a métrica linha de código, pois esta última atribui o mesmo peso para todas as instruções.

Uma outra idéia foi proposta por [JAY85] para medir a complexidade usando a contagem de operadores de decisões. Ele sugere que, em lugar da contagem de decisões, seja utilizada a proporção número de decisões dividido pela métrica linhas de código executáveis (LCE) para obter a densidade de decisão, ou seja,

Densidade de Decisão = Contagem de Decisões / LCE.

Esta métrica revela com que densidade os comandos de decisão aparecem nas linhas de código do programa. Por exemplo, uma densidade de 10% significa que em cada 100 linhas de código executáveis é possível encontrar somente 10 instrucões de decisão. Um código com 5% de densidade seria mais fácil de manter do que outro com 50%.

As métricas contagem e densidade de decisão ajudam a esclarecer a com-

plexidade de um programa; elas representam uma aproximação da mais importante métrica de fluxo de controle, a Complexidade Ciclomática.

Complexidade Ciclomática

A complexidade ciclomática, de um grafo orientado, foi elaborada por [McCAB76] a partir de dois princípios da teoria dos grafos cujos conceitos básicos necessários para sua comprensão estão definidos no Anexo A. O primeiro define o número ciclomático **V(G)** de um grafo **"G"** com **"n"** vértices, **"a"** arestas e **"p"** componentes conexos como

$$V(G) = a - v + 2p$$

O segundo princípio diz que dado um grafo **G fortemente conexo**, o seu número ciclomático equivale ao número máximo de **circuitos linearmente independentes**. McCabe então associa a cada programa um grafo orientado com um único vértice de entrada e um outro de saída. Cada nó do grafo corresponde a um segmento de código do programa, onde o fluxo de controle é sequencial; e os arcos corresponden aos desvios do fluxo no programa. Este tipo de grafo é geralmente conhecido como **Grafo de Controle do Programa**. A Figura 4.5 seguinte mostra um exemplo deste tipo de grafo.

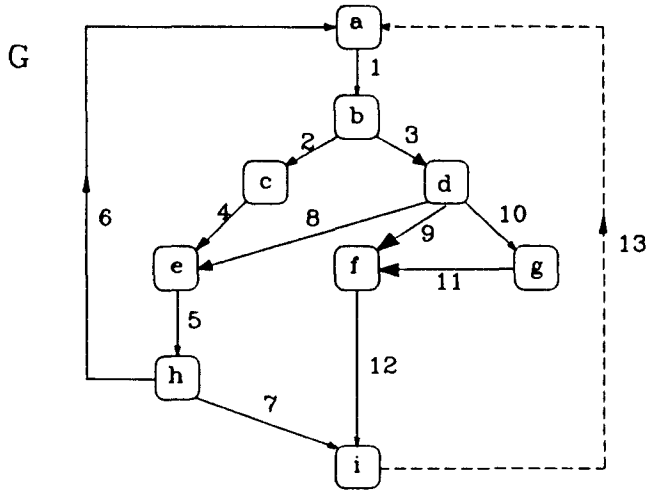

Figura 4.5: Grafo de controle de um Programa

Cada nó pode ser alcançado a partir do nó inicial **"a"** e, partindo de qualquer nó pode-se alcançar o nó final **"i"**.

O segundo principio pode ser aplicado ao grafo "G" da seguinte maneira:

Suponha-se que o vértice **"i"** é conectado ao vertice **"a"** (Figura 4.5). O grafo de controle **"G"** fica, portanto, fortemente conexo. Isto permite que o número maximo de circuitos linearmente independentes possa ser calculado como

$$V(G) = 13 - 9 + 2 = 6.$$

Os circuitos são os seguintes:

1. abceha
2. abcehia
3. abdeha
4. abdehia
5. abdfia
6. abdgfia

A estratégia de McCabe consiste, em medir ou quantificar a complexidade de um programa através do cálculo do número de caminhos linearmente independentes V(G) e desta maneira, controlar o tamanho dos programas. Com isto procura-se, facilitar o teste e manutenção dos módulos. Isto pode ser feito definindo limites máximo de complexidade e verificando se os módulos apresentam os limites estabelecidos. É evidente que quanto menor a complexidade de um procedimento ou módulo, mais fácil será testá-lo, entendê-lo e mantê-lo. [McCAB76] também apresenta uma técnica chamada de Caixa Branca, baseada na complexidade ciclomática, para verificar a correção dos programas. Essencialmente, este teste consiste em determinar todos os caminhos básicos de um programa e verificar que todos eles sejam testados no mínimo uma vez.

Algumas das características mais importantes desta métrica são:

1. V(G) > 1.
2. V(G) é o número máximo de caminhos linearmente independentes em G.
3. V(G) não é afetada pela inserção ou eliminação de comandos funcionais (não condicionais).
4. G contém um único caminho se, e somente se V(G) = 1.
5. Caso seja inserida mais uma aresta em G, V(G) incrementa em 1 (um).
6. V(G) depende somente da estrutura de decisão (controle) de G.

Esta métrica pode ser interpretada como o número de arestas que necessitam ser retiradas do grafo G para deixá-lo sem circuitos ou laços. McCabe chamou a atenção também para o fato de que a complexidade ciclomática e o número ciclomático equivalem ao número de regiões de um grafo.

A complexidade ciclomática pode ser reduzida da seguinte maneira, para facilitar a sua aplicação:

Seja **"p"** o número de predicados, então

$$a = v - 1 + p$$

ou

$$a - v = p - 1$$

adicionando-se "2" a cada lado tem-se:

$$a - v + 2 = p - 1 + 2$$

o que na verdade é

$$V(G) = a - v + 2 = p - 1 + 2$$

ou

$$V(G) = p + 1$$

no grafo da figura 4.5, V(G) = 5 + 1 = 6; ou seja, a métrica de McCabe pode ser reduzida a um simples algoritmo computacional. Também é fácil provar que a complexidade ciclomática de um programa composto de vários módulos é igual a soma das complexidades dos seus módulos. Por exemplo, em um programa "**q**" com "**m**" módulos:

$$V(G_q) = \sum_{i=1}^{m} V(G_i) = \sum_{i=1}^{m} V(A_i) - \sum_{i=1}^{m} V(V_i) + 2m$$

onde A_i e V_i representam o número de arestas e nós do i-ésimo módulo. E $V(G_i)$ representa a complexidade ciclomática do programa "**q**". Como

$$V(G_i) = P_i + 1,$$

então

$$V(G_i) = \sum_{i=1}^{m} (P_i) + m.$$

[BAKER80] comparou três métricas baseadas no fluxo de controle e concluiu que a complexidade ciclomática de McCabe possui uma base analítica firme, baseada na teoria dos grafos, e quantifica adequadamente a complexidade do fluxo de controle.

Além das métricas da estrutura do programa descritas nesta seção, existem outras que não foram incluídas neste trabalho para não estendê-lo demasiadamente. Entre elas pode-se citar o Número Mínimo de Caminhos de um Programa, a Acessibilidade de Cada Instrução ,"Reachability" [SCHNE79], o Nível de Aninhamento de Cada Instrução de um Programa, "Depth of Nesting" apresentada por Zolnowiski em 1981 e descrita em [CONTE86], métricas para o sistema multitarefa do sistema ADA[SHATZ88] e, finalmente, a Freqüência de Uso de Desvios [WOODW79].

4.2.3 Métricas Baseadas na Estrutura de Dados

Uma métrica baseada na estrutura de dados consiste na contagem da quantidade de dados que entram, são processados e saem de um programa [CONTE86]. Apesar de existir um grande número de métricas relacionadas com os dados de um programa, todas elas podem ser classificadas em três grupos: métricas que medem simplesmente a quantidade de dados; aquelas que medem o grau de compartilhamento dos dados entre os distintos módulos; e, finalmente, aquelas que medem o uso da informação entre os módulos.

Quantidade de Dados num Programa

Esta métrica é a mais simples entre aquelas relacionadas com a estrutura de dados. Como seu nome indica, a quantidade de dados consiste na contagem de todas as variáveis num Software. Uma variável é uma seqüência de caracteres alfanuméricos definidos por um programador, e que apresenta algum valor durante a compilação ou execução do programa [CONTE86]. A maioria dos compiladores produz uma listagem contendo todas as variáveis definidas em um programa. Portanto, esta métrica é muito simples de obter, mesmo manualmente. Os operadores η_2 e N_2 de Halstead também medem a quantidade de dados. A métrica η_2 não mede a quantidade de dados pois só inclui os operandos únicos; N_2 é mais adequada pois representa o número de vezes que as variáveis aparecem num programa.

O Uso de Dados nos Módulos

[DUNSM79] apresentou duas métricas, o número de variáveis ativas e a extensão de uma variável, para medir o uso de dados nos módulos de um programa. A justificativa para estas duas métricas deduz-se do fato que quanto mais ítens o programador tem que monitorar durante o processo de desenvolvimento, mais difícil será construir o Software. Para Dunsmore, uma variável está ativa desde a primeira até a última referéncia dela num procedimento [DUNSM79]. Ele afirma que a complexidade de um procedimento pode ser quantificada com a seguinte proporção

$$Complexidade = \frac{Uso\ total\ de\ Variáveis\ Ativas}{Linhas\ de\ Código\ Executáveis}$$

Por exemplo, o programa a seguir tem a Tabela 4.1 de variáveis ativas

```
1. program bubblesort(input,output);
2. type matrixint = array [1..100] of integer;
3. var i, j, Ultimo, Comprimento: integer;
4. continua: boolean;
5. a, b:matrixint;
6.
7. procedure troca(var x: matrixint; k:integer);
8. var t:integer;
9. begin
10. t := x[k];
11. x[k] := x[k+1];
12. x[k+1] := t
13. end;
14.
15. begin
16. read(Comprimento);
17. for j := 1 to Comprimento do
18. read(a[j],b[j]);
19.
20. Ultimo := Comprimento;
21. Continua := true;
22. While Continua do begin
23. Continua := false;
```

```
24. Ultimo := Ultimo - 1;
25. i := 1;
26. While i ¡ Ultimo do begin
27. if a[i] ¿ a[i+1] then begin
28. Continua := true:
29. troca(a,i);
30. troca(b,i)
31. end;
32. i := i + 1
33. end;
34. end;
35. for j := 1 to Comprimento do begin
36. writeln(a[j],b[j])
37. end
38. .
```

æ

LINHA	VARIÁVEIS ATIVAS	TOTAL
9		0
10	t , x , k	3
11	t , x , k	3
12	t , x , k	3
13		0
15		0
16	Comprimento	1
17	Comprimento , j	2
18	Comprimento , j , a , b	4
19	Comprimento , j , a , b	4
20	Comprimento , j , a , b , Ultimo	5
21	Comprimento , j , a , b , Ultimo , Continua	6
22	Comprimento , j , a , b , Ultimo , Continua	6
23	Comprimento , j , a , b , Ultimo , Continua	6
24	Comprimento , j , a , b , Ultimo , Continua , i	6
25	Comprimento , j , a , b , Ultimo , Continua , i	7
26	Comprimento , j , a , b , Continua	7
27	Comprimento , j , a , b , Continua	6
28	Comprimento , j , a , b , i	6
29	Comprimento , j , a , b , i	5
30	Comprimento , j , a , b , i	5
31	Comprimento , j , a , b , i	5
32	Comprimento , j , a , b , i	5
33	Comprimento , j , a , b	4
34	Comprimento , j , a , b	4
35	Comprimento , j , a , b	4
36	j , a , b	3
37		0

Tabela 4.1: Variáveis Ativas do Programa da Figura 4.5.

A complexidade do programa 4.1 portanto corresponde a (110/28) = 3.9.

A **extensão** refere-se ao número de linhas de código entre duas referências sucessivas da mesma variável [ELSHO76]. Se a variável aparece "n" vezes em um programa, ela tem "n" - 1 extensões. Uma variável com uma extensão muito grande pode requerer um esforço igualmente muito grande por parte do programador, pois ele teria que se lembrar da sua existência desde o início até o fim do processo de desenvolvimento.

As métricas apresentadas no grupo estrutura de dados foram definidas para serem aplicadas em nível de módulos. Contudo, elas podem ser estendidas para aplicação em nível de programa da seguinte maneira: suponha-se que um programa tenha m módulos e MVV representa a média de variáveis ativas

$$MVV_{programa} = \frac{\sum_{i=1}^{m} MVV_i}{m}$$

onde MVV_i representa a média das variaveis ativas para o i-ésimo módulo.

A extensão média (EXTM) é a soma de todas as extensões das variáveis de um módulo dividida pelo número de variáveis de um programa com "n" módulos. Isto é:

$$EXTM_{programa} = \frac{\sum_{i=1}^{n} EXTM_i}{N}$$

onde $EXTM_i$ representa a extensão média do i-ésimo módulo.

4.2.4 Métrica do Fluxo da Informação

O objetivo desta métrica é medir a estrutura de sistemas de grande porte, porém a sua aplicação também é válida para avaliar a complexidade de procedimentos ou módulos e da acoplagem de um sistema. Henry identifica quatro tipos de fluxos de dados que geralmente acontecem em um programa [HENRY81]. Esses tipos são definidos a seguir.

Dados quatro módulos A, B, C e D e uma estrutura de dados ED, existe um fluxo global de informação de A para B através da estrutura de dados ED se:

1. A deposita informação em ED e
2. B retira esta informação (1) de ED

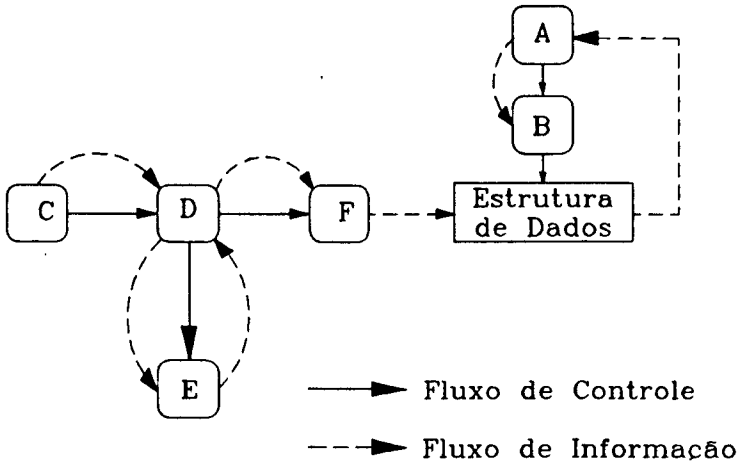

Figura 4.6: Exemplo de um Fluxo de Informação

Existe um fluxo local de informação de A para B através da estrutura de dados ED se qualquer uma das três condições seguintes é válida:

1. Se A chama B,
2. Se B chama A e A retorna um valor para B, e o valor é utilizado por B ou
3. Se C chama ambos, A e B, passando um valor de A para B.

Existe um fluxo direto de informação de A para B se a condição (1) da definição de fluxo local é verdadeira.

Existe um fluxo indireto de informação de A para B se a condição (2) ou (3) da definição de fluxo local é verdadeira.

Um fluxo de informação é geralmente representado da seguinte maneira
destino ← fonte1, fonte2, ..., fonteN

Henry usa o fluxo F.ED ← F.1.I, F.2.I, por exemplo, para indicar que a

estrutura de dados ED é atualizada, baseada no valor de seus dois parâmetros de entrada [HENRY81]. A partir destas informações, constróe-se uma árvore binária que mostra a relação entre os módulos.

Os autores garantem que as relações:

1. FAN-IN * FAN-OUT
2. $(\text{FAN-IN} * \text{FAN-OUT})^2$
3. $\text{COMPRIMENTO} * (\text{FAN-IN} * \text{FAN-OUT})^2$

onde o comprimento pode ser em linha de código (LOC), "FAN-IN" e "FAN-OUT" equivalem ao número de fluxos e estruturas de dados que entram, saem e são acessados respectivamente pelo procedimento, quantificam a complexidade de um procedimento, módulo e sistema. Segundo [HENRY81], a complexidade de um procedimento depende da complexidade de seu código, o que justifica a inclusão do comprimento, e a complexidade da conexão do procedimento com o seu meio ambiente. Para Henry, estas funções, além de quantificar a complexidade de um programa, ajudam a localizar erros potenciais logo na fase de projeto, mostram as interconexões entre os módulos com bastante clareza e facilitam o desenvolvimento de outras métricas com a finalidade de medir a complexidade, acoplamento, nível de interação e os pontos críticos de possíveis falhas de um sistema. Esta métrica foi aplicada ao sistema UNIX versão 6 com resultados muito positivos.

4.3 IMPUREZAS QUE AFETAM AS MÉTRICAS

Como foi indicado na teoria "Software Science" de Halstead o comprimento de um programa está relacionado com o tamanho do vocabulário que se precisa para implementá-lo; além disso, o produto de seu volume pelo seu nível (volume potencial) permanece constante quando o programa é traduzido para uma outra linguagem [HALST77]. Halstead validou sua teoria através da análise de algoritmos, publicados em periódicos científicos, como por exemplo IEEE e outros. Desta forma, as expressões de "Software Science" podem ser consideradas apenas algoritmos que já passaram por um extenso processo de depuração para aumentar a sua eficiência, ou seja, algoritmos que podem ser considerados como "puros".

VERSÕES	η_1	η_2	N_1	N_2
1 – $SQDSUM(P\ Q\ R)$	2	3	2	3
2 – $(P + Q) * (P + Q) \rightarrow R$	4	3	6	5
3 – $P + Q \rightarrow R,\ R * R \rightarrow R$	3	3	4	5
4 – $P + Q \rightarrow T1,\ P + Q \rightarrow T2,\ T1 * T2 \rightarrow R$	4	5	8	9
5 – $P + Q \rightarrow T,\ T * T + T - T \rightarrow R$	5	4	7	8
6 – $SUM(P\ QT),\ SQRT(TR)$	4	4	5	5
7 – $P * (P + Q) + Q * (P + Q) \rightarrow R$	4	3	8	7
8 – $P + Q \rightarrow R,\ R * R \rightarrow R$	4	3	5	6
9 – $P + Q \rightarrow T1,\ T1^2 \rightarrow R$	4	5	5	6
10 – $P + Q \rightarrow T,\ T * T + T - T + T - T \rightarrow R$	5	4	9	10
11 – $(P + Q)^2 \rightarrow R$	4	4	4	4
12 – $P * P + P * Q + P * Q + Q + Q * Q \rightarrow R$	3	3	8	9
13 – $SUM(P\ QR),\ SQR(RR)$	4	3	5	5
14 – $P \rightarrow T1,\ Q \rightarrow T2,\ T1 + T2 \rightarrow T3,\ T3 * T3 \rightarrow R$	4	6	9	10
15 – $LDAP\ ADDQ\ STOT\ MPYT\ STOR$	4	4	5	5
16 – $P^2 + Q^2 + 2 * P * Q \rightarrow R$	4	4	7	8
17 – $P + Q \rightarrow T,\ T * T \rightarrow R$	4	4	5	6
18 – $P * Q + 2 * P * Q + Q * Q \rightarrow R$	3	4	7	8

Tabela 4.2: Dezoito versões Impuras de um Algoritmo Trivial.

Diferentes implementações de um mesmo algoritmo resultam em valores distintos para as métricas de tamanho. A partir desta observação Halstead decidiu elaborar critérios de pureza de implementações, e identificou seis tipos de impurezas [HALST77]. Primeiro, ele definiu um algoritmo trivial de uma linha só, como uma chamada de procedimento. Depois, o algoritmo foi implementado de todas as maneiras possíveis com a intenção de obter todos os tipos de estilos "pobres", assim como as versões que eram diretas e talvez "puras". Terceiro, para todas as versões obtidas foram calculados os valores de η_1, η_2, N_1 e N_2 e também V , L e V^*. Feitos estes cálculos, todas as versões foram estudadas com a finalidade de identificar "impurezas". Toda vez que uma impureza era encontrada, era removida da versão original e de todas as outras versões em que aparecia. O algoritmo usado no experimento tinha por objetivo somar dois operandos P e Q, elevar o resultado ao quadrado e colocar o resultado no operando R. As Tabelas 4.2 e 4.3 foram extraídas de [HALST76], Tabela 7.1.

VERSÕES	V	L	L * V
1	12	1,00	11,6
2	31	0,30	9,3
3	23	0,40	9,3
4	54	0,28	15,0
5	48	0,20	9,5
6	30	0,40	12,0
7	42	0,21	9,1
8	31	0,25	7,7
9	35	0,42	14,4
10	60	0,16	9,6
11	24	0,50	12,0
12	44	0,22	9,8
13	28	0,30	8,4
14	63	0,30	8,4
15	30	0,40	12,0
16	45	0,25	11,2
17	33	0,33	11,0
18	42	0,33	14,0

Tabela 4.3: Dezoito versões impuras de um Algoritmo Trivial.

Impureza I - Operações Complementares

Uma das impurezas mais evidentes, e que podem ser identificadas de uma maneira intuitiva, consiste na aplicação sucessiva de dois operadores que se complementam, ao mesmo operando. Esta impureza pode ser identificada e eliminada pelos compiladores que fazem otimização de código. Observando-se a tabela 4.2 pode-se perceber que a versão (5) contém esta impureza. A versão original é

$$P + Q \rightarrow T,\ T * T + T - T \rightarrow R$$

O produto V * L é dado por:

$$V * L = (N_1 + N_2) * log_2(n_1 + n_2) * \frac{\eta_2 * 2}{\eta_1 * N_2}$$

$$= (7 + 8) * log_2(5 + 4) * \frac{2 * 4}{5 * 8} = 9.51.$$

Quando se simplifica a versão (5), removendo-se as operações complementares, obtem-se

$$P + Q \rightarrow T,\ T * T \rightarrow R$$

$$V * L = (5 + 6) * log_2(4 + 4) * \frac{2 * 4}{4 * 6} = 11.0.$$

Este último resultado está bem mais perto do que o da versão original. A versão (10) também possui este tipo de impureza.

Impureza II - Operandos Ambíguos

Esta impureza acontece quando o mesmo nome é atribuído a duas entidades diferentes em lugares diferentes de um programa. Esta prática é comum quando se quer evitar o uso excessivo de memória, porém o resultado pode ser negativo em relação à clareza do programa.

A versão (8) da Tabela 2 é um exemplo típico deste caso. Nela pode-se observar que o operando R é usado primeiro para representar a soma, e depois para guardar o quadrado da soma. A versão original

$$P + Q \rightarrow R, R * R \rightarrow R$$

produz

$$V * L = (5 + 6) * log_2(4 + 3) * \frac{2 * 3}{4 * 6} = 7.7.$$

Depois de purificada, a versão (8) fica

$$P + Q \rightarrow T,\ T * T \rightarrow R$$

$$V * L = (5 + 6) * log_2(4 + 4) * \frac{2 * 4}{4 * 6} = 11.0.$$

onde, novamente, o valor purificado (11.0) está mais perto do valor teórico (11.6) do que a versão original. As versões (3) e (13) da Tabela 4.1 também apresentam este tipo de impureza.

Impureza III - Operandos Sinônimos

O complemento da ambigüidade é o uso de dois ou mais nomes diferentes para o mesmo objeto. Esta ambigüidade aparece sempre que são utilizados vários nomes, de tal forma que seu valor é sempre idêntico.

A versão (4), é uma ilustração deste tipo de impureza:

$$P + Q \rightarrow T_1, P + Q \rightarrow T_2,\ T_1 * T_2 \rightarrow R$$

Isto produz:

$$V * L = (8 + 9) * log_2(4 + 5) * \frac{2 * 5}{4 * 8} = 15.0.$$

a nova versão é:

$$P + Q \rightarrow T_1,\ T_1 * T_1 \rightarrow R$$

resulta em:

$$V * L = (5 + 6) * log_2(4 + 4) * \frac{2 * 4}{4 * 6} = 11.0.$$

Impureza IV - Subexpressões Comuns

Cada vez que uma combinação específica de termos deve ser usada mais de uma vez, é comum atribuir-lhe um nome novo e utilizá-lo nas referências seguintes. Quando isto não é respeitado, diz-se que o programa contém subexpressões comuns. Este tipo de impureza também pode ser detectado e

corrigido por compiladores que fazem otimização de código. Na Tabela 4.2, a versão (2) contém este tipo de impureza:

$$(P+Q)*(P+Q) \rightarrow R$$

O produto L * V resulta em

$$V*L = (5+6)*log_2(4+3)*\frac{2*3}{4*5} = 9.3.$$

e a versão purificada

$$P+Q \rightarrow T,\ T*T \rightarrow R$$

produz

$$V*L = (5+6)*log_2(4+4)*\frac{2*4}{4*6} = 11.0.$$

Além da versão (2), as versões (7) e (12) apresentam a impureza do tipo IV.

Impureza V - Atribuições Desnecessárias

Este tipo de impureza é considerada como o oposto da anterior. Ou seja, uma combinação de termos recebe um nome diferente, mas que só é usado uma vez no programa. Em outras palavras, o nome não tem nenhuma utilidade pois a subexpressão não aparece em nenhum outro lugar do programa. Na Tabela 4.2, só a versão (9) contém este tipo de impureza.

A versão original

$$(P+Q)^2 \rightarrow T_1, T_1 \rightarrow R$$

produz

$$V*L = (5+6)*log_2(4+5)*\frac{2*5}{4*6} = 14.5.$$

A versão purificada

$$(P+Q) \rightarrow R^2$$

fornece

$$V*L = (4+4)*log_2(4+4)*\frac{2*4}{4*4} = 12.0.$$

Impureza VI - Expressões Não Fatoradas

Intuitivamente, uma expressão fatorada é mais fácil de entender do que uma não fatorada. Este tipo de impureza é diferente das outras pois não existe um método mecânico para removê-la. Por outro lado, a sua ocorrência é rara. Na Tabela 2, a versão (18) é a única que apresenta o tipo

$$P * P + 2 * P * Q + Q * Q \rightarrow R$$

a qual produz

$$V * L = (7 + 8) * log_2(3 + 4) * \frac{2 * 4}{3 * 8} = 14.0.$$

A versão purificada

$$(P + Q)^2 \rightarrow R$$

nos dá:

$$V * L = (4 + 4) * log_2(4 + 4) * \frac{2 * 4}{4 * 4} = 12.0.$$

Pode-se concluir que a remoção das impurezas de um programa melhora a concordância entre V * L e o volume potencial. A Tabela 3 contém o resultado da purificação da Tabela 2 completa. Pode-se notar também que as versões (1), (6), (11), (15), (16) e (17), apesar de serem diferentes, não apresentam nenhuma impureza daquelas identificadas por Halstead e que seus valores para V * L estão aproximadamente entre 11.0 e 12.0.

Capítulo 5

AnaSoft: O Analisador

Os Capítulos anteriores ressaltaram o fato de que, atualmente, a manutenção de programa é uma das fases mais caras e difíceis do processo de desenvolvimento de Software (ciclo de vida). Um dos objetivos principais da Engenharia de Software tem sido prover os programadores dos conhecimentos científicos necessários para desenvolver produtos eficientes e de baixo custo. Também foi explicado que uma das características que mais afetam a manutenção de código é a complexidade [HARRI82]. O capítulo anterior mostra, além do mais, que existe uma grande variedade de técnicas, muitas delas com uma base científica sólida, que quantificam a complexidade de uma forma bastante precisa.

Este trabalho propõe o desenvolvimento de uma ferramenta, AnaSoft, que identifica, em programas codificados em Turbo Pascal, módulos ou procedimentos que contém uma complexidade muito alta e que, portanto tornam-se difíceis de manter ou modificar. Esta ferramenta usa como base três das métricas mais eficientes e melhor conhecidas para medir complexidade: **"Software Science", Complexidade Ciclomática e Fluxo da Informação.** Outro fator que influenciou a sua escolha, além de sua popularidade, é o fato de que um grande número de pesquisadores está de acordo com o princípio que estabelece que a complexidade de um sistema está fortemente relacionada com as suas estruturas internas de dados, a sua estrutura lógica e a sua interface com o meio ambiente. Acredita-se que estas três métricas escolhidas refletem este princípio de uma maneira realista.

Imagine-se em um ambiente de desenvolvimento de programas onde o programador chefe estabelece certos critérios de complexidade para garantir a qualidade dos Software a serem desenvolvidos pelos programadores. Seria importante para o chefe verificar, de uma forma independente dos programa-

dores que desenvolveram o código, se os critérios de qualidade predefinidos foram respeitados. Quanto maior o porte do Software, mais dispendiosa e demorada é esta tarefa, ameaçando o prazo de entrega do sistema. Com uma ferramenta automática do tipo que este trabalho propõe, a tarefa do programador chefe seria enormemente facilitada. O objetivo da ferramenta que esta dissertação propõe é auxiliar a equipe de desenvolvimento de programas a identificar os módulos ou procedimentos que apresentam um nível de complexidade excessivamente alto, para que estes possam ser ajustados de acordo com parâmetros pré-estabelecidos, e desta maneira prevenir futuros problemas na manutenção.

5.1 Qualidades a Serem Avaliadas

No Capítulo 2 deste trabalho foram definidas as qualidades de um Software. Dentre as características que foram definidas, existem no mínimo três que são de suma importância para este trabalho. A importância se deve ao fato de serem fortemente afetadas pela complexidade do programa, que é o que a ferramenta propõe medir. As caraterísticas são: Confiabilidade, Flexibilidade e Testabilidade.

A Flexibilidade é importante porque entre 80% e 90% do trabalho de manutenção consiste em modificar programas. Estas modificações, junto com o desenvolvimento de programas novos representa aproximadamente 75% do custo das empresas que desenvolvem Software [JAY85]. Um programa com uma complexidade muito alta torna a sua manutenção muito difícil, pois quanto mais complexo, menos flexível ele é [LI87,LEW88]. A complexidade daqueles procedimentos identificados pela ferramenta deve ser revisada e se necessário, diminuída. Esta ferramenta permite ao usuário estabelecer e mudar os parâmetros de complexidade para ajustá-los à realidade da empresa que desenvolve Software. Tanto os programas novos a serem desenvolvidos como os que já existem devem incorporar a qualidade de Flexibilidade, pois esta contribui para melhorar e facilitar a manutenção.

Assim como a Flexibilidade, a Confiabilidade de um programa também é afetada pela complexidade. Um programa que apresenta um grau excessivo de complexidade implica um módulo não muito claro e portanto não confiável. Isto também se aplica à Testabilidade: quanto mais complexo o procedimento ou módulos mais difícil será testá-lo. Os procedimentos identificados pela ferramenta devem ser revisados com o objetivo de constatar se é necessário ou não modificá-los para diminuir sua complexidade, e, com isto,

diminuir suas chances de apresentar problemas de manutenção no futuro.

5.2 Estratégia

Um dos problemas que as métricas para medir complexidade apresentam, e que foi discutido no Capítulo 4, resulta do fato de que não existe consenso geral entre os pesquisadores sobre a melhor maneira de se definir e contar os símbolos necessários para a quantificação da complexidade. Porém, antes de se desenvolver qualquer ferramenta, é imperativo estabelecer os parâmetros, critérios ou estratégias a serem usadas no processo. O grupo de pesquisa da área de métricas para Software da Universidade de Purdue definiu e difundiu uma estratégia geral para ser utilizada na avaliação de programas desenvolvidos em Pascal [CONTE86]. Esta tese considerou a utilização desta estratégia, com ligeiras modificações, para contribuir com a padronização e facilitar o trabalho de implementação do analisador. A estratégia é a seguinte:

- **Para Software Science**

  1. Todas as partes do programa, mesmo cabeçalhos e a parte de declarações, são consideradas no cálculo das métricas. Os únicos segmentos que não são incluídos são os comentários.
  2. São considerados operandos todas as variáveis, constantes, incluindo as constantes predefinidas FALSE, TRUE e MAXINT, tipos definidos pelos usuários, "strings", nome de arquivos, e a palavra reservada NIL. Todos os operandos são considerados como tendo escopo global. Em outras palavras, as variáveis locais, definidas com o mesmo nome em procedimentos diferentes, são contadas como ocorrências múltiplas do mesmo operando.
  3. Os seguintes símbolos são contados como um só (não é feita nenhuma diferença entre o uso do operador "*" tanto em operações com conjuntos como em operações aritméticas):

*	/	DIV	MOD	<	<=
;	<>	>=	>	:=	^
'	..	NOT	AND	OR	IN

PACKED	TO	FUNCTION	INTEGER	REAL	TEXT
CHAR	LABEL	PROGRAM	EXTERN	FORWARD	PROCEDURE

4. Os seguintes pares de símbolos são sempre contados como um só:

BEGIN-END	CASE-END	WHILE-DO	REPEAT-UNTIL
IF-THEN	IF-THEN-ELSE	FOR-DO	WITH-DO
SET OF	FILE OF	RECORD-END	ARRAY-OF

5. Os seguintes símbolos (ou pares de símbolos) são contados como um operador só, de acordo com a condição especificada ao lado de cada um:
   - "VAR" é contado como operador quando usado em listas de parâmetros de rotinas e não é considerado quando aparece como rótulo na declaração de variáveis.
   - "=" é contado como um operador relacional em expressões ou como um operador de definição em seções não executáveis do programa.
   - "+" é contado como um operador unário ou binário, dependendo da função que ele está desempenhando. O operador binário "+" é interpretado da mesma forma, tanto para conjuntos como para expressões aritméticas.
   - "-" é contado como um operador unário ou binário, dependendo da função que ele está desempenhando. O operador binário '-' não é diferenciado tanto no uso em conjuntos como em expressões aritméticas.
   - "." é contado, dependendo da sua função, como um símbolo para seleção de componentes de um "record" ou como um indicador de fim de programas.
   - ":" é contado como um operador de definição na seção "var" e em listas de parâmetros e como um operador separador em comandos "CASE" e "GOTO".
   - "()" é contado como um operador separador de argumentos ou como um operador em comandos, dependendo de sua função.

- "[ ]" é contado como um operador de índices ou como um operador de conjunto, dependendo de sua função.

6. Todas as chamadas de procedimentos e funções são contadas como operadores, incluindo o nome da função ou procedimento que é usado na declaração.
7. O comando "GOTO label" é contado como o operador "GOTO" e o operando "label".
8. A declaração de rótulos não é contada. Todos os rótulos a partir da palavra-chave "label" são ignorados.
9. Os seguintes dispositivos sintáticos não são contados: CONST/ TYPE/ VAR (como rótulo de seção).

- **Para a Complexidade Ciclomática V(G)**

1. A complexidade é acrescida de 1 (um) toda vez que qualquer um dos seguintes comandos aparecer:

WHILE	FOR	REPEAT	IF	AND
OR	PROCEDURE	FUNCTION	PROGRAM	

   AND e OR formam parte de laços/desvios controlados por variáveis booleanas. Os comandos condicionais (IF, WHILE, FOR, etc.) não podem ser contados diretamente como funções booleanas, pois as variáveis não seriam contadas corretamente. Por exemplo, na expressão "IF a < b THEN c := a"; bastaria considerar o operador "IF". Porém, na expressão "IF a < b AND b < c THEN a := b"; seria necessário considerar tanto o operador "IF" quanto o operador "AND".
2. Os rótulos do comando "CASE" devem ser contados da seguinte maneira:
   - aumentar um para cada dois-pontos (:) encontrados na parte executável do programa, mas não incluídas na lista de parâmetros do comando "WRITE(LN)".
   - aumentar no total o número de vírgulas quando há mais de um rótulo. Exemplo: 1,3,5,8:... (incrementa em 3).

- diminuir quando for encontrado o comando "LABEL KEYWORD" ou diminuir o número de vírgulas se for uma lista de rótulos do tipo "LABEL label1,label2,...labeln.". Isto é necessário para eliminar os rótulos do comando "GOTO".

OBSERVAÇÃO: A parte variável do comando ou operador "RECORD" do Turbo Pascal versão 4.0 não foi implementada.

5.3 Estrutura do Analisador

O objetivo inicial da implementação era desenvolver o programa de forma que a cada métrica escolhida correspondia a um módulo da ferramenta. Chegou-se, no entanto, à conclusão de que esta maneira era mais complicada, pois precisaria ser feita em dois passos. No primeiro teria-se que ler o arquivo de entrada em Pascal, criar-se-ia algum tipo de estrutura para guardá-lo e depois aplicar-se-ia as métricas. Depois de se fazer uma análise mais cuidadosa, chegou- se à conclusão de que seria mais conveniente realizar a análise do programa na medida em que ele fosse lido. Desta maneira, acredita-se que a ferramenta ficou mais eficiente, pois a memória adicional para se guardar o programa não foi necessária; gastou-se menos tempo de execução do analisador, pois grande parte da análise se faz ao mesmo tempo que o programa de entrada está sendo lido. As únicas métricas que precisam aguardar até o programa ser lido completamente para serem calculadas são as métricas do Volume, Dificuldade e Esforço.

Uma das motivações para se utilizar a linguagem Pascal neste trabalho de tese é que ela oferece uma excelente carta sintática, além do que é uma linguagem muito utilizada. A carta sintática foi uma ajuda inestimável no processo de definição e estruturação dos módulos e procedimentos. AnaSoft foi implementado de uma forma parecida com a de um compilador em Pascal, ou seja, cada procedimento corresponde a um elemento da carta sintática. Por exemplo, a estrutura **bloco** da carta sintática corresponde ao procedimento **Bloco** em AnaSoft. Isto facilita enormemente a sua flexibilidade, entendimento e portanto futuras modificações que sejam feitas no analisador.

O sistema AnaSoft foi dividido em três módulos principais. O primeiro contém aqueles procedimentos que acessam diretamente a estrutura de dados principal, TPROC. O segundo inclui aqueles procedimentos que calculam as métricas, "Software Science", Complexidade Ciclomática e Fluxo de Informação. O programa principal possui os procedimentos que correspondem

à carta sintática da linguagem turbo Pascal versão 4.0.

5.4 Métricas Implementadas

As métricas da dificuldade e esforço da família "Software Science" de Halstead, a Complexidade Ciclomática de McCabe e o Fluxo de Informação foram as métricas escolhidas para serem implementadas em AnaSoft.

No Capítulo 4, a métrica Fluxo de Informação foi apresentada de acordo com a definição em [HENRY81]. Depois de analisar esta métrica com atenção e considerar as mudanças e tendências mais recentes na área de Engenharia de Software, esta tese chegou à conclusão de que esta métrica pode ser simplificada de tal forma que a sua implementação possa ser facilitada sem prejudicar a sua efetividade.

Na sua definição de "FAN-IN" e "FAN-OUT", [HENRY81] inclui os acessos às estruturas de dados globais como fatores que contribuem para a complexidade de um Sistema de Software. Além disso, ela distingue as diferentes maneiras de que um procedimento pode ser chamado (com parâmetros e sem parâmetros) para diferenciar os acessos às estruturas de dados globais. A maneira como estes fluxos de informações são definidos, porém dificultam desnecessariamente a implementação da métrica.

Este trabalho propõe uma maneira de simplificar a definição e portanto a implementação da métrica proposta por [HENRY81]. Pode-se afirmar que cada vez que um procedimento chama um outro acontece um fluxo de informação. Este fluxo de informação acontece de uma forma explícita ou implícita. Um exemplo de um fluxo implícito seria uma chamada de um procedimento que simplesmente apresenta um diagrama na tela. Um fluxo explícito, por outro lado, seria representado por uma chamada de procedimento com passagem de parâmetros como, por exemplo, uma chamada para um procedimento que ordena um vetor (ORDENA(X)). Neste último caso, o fluxo de informação está representado pelo vetor "X". Em outras palavras, o fluxo de informação acontece e isso é o importante.

Os acessos às estruturas de dados globais não precisam ser aplicados a todos os procedimentos de um sistema de Software pelo simples motivo de que a tendência atual na área de desenvolvimento de Software (Níveis de Abstração e Programação Orientada por Objeto) é de separar todas as funções e procedimentos que acessam as estruturas de dados globais num modulo só, para garantir a consistência e segurança dos dados e ao mesmo tempo facilitar a Reutilização e Manutenção dos Sistemas. Quando se aplica

a métrica proposta por [HENRY81], todos os procedimentos são analisados visando determinar o tipo de fluxo que eles apresentam, e se eles fazem acesso às estruturas de dados globais ou não. Isto prejudica o desempenho da métrica, pois sabe-se que a maioria dos procedimentos não fazem este tipo de acesso e portanto esta Análise não é necessária. A Análise destes procedimentos pode ser feita em separado e de forma que não afete negativamente a eficiência e desempenho da ferramenta.

A definição original de "FAN-IN" pode ser redefinida da seguinte maneira: "FAN-IN" representa o número de vezes que um procedimento é chamado por outros procedimentos ou por ele mesmo (no caso de recursividade). E "FAN-OUT" representa o número de chamada de procedimentos que um procedimento faz. O resto da métrica fica igual, ou seja:

$$Complexidade = Tamanho * (FAN - IN * FAN - OUT)^2.$$

vspace6mm

Onde o tamanho usado é o mesmo apresentado por [HALST76] e apresentado no Capítulo 4 deste trabalho ($N = N_1 + N_2$). Um procedimento que apresenta uma complexidade do Fluxo de Informação muito alta pode estar desempenhando mais de uma função (é considerado conveniente desenvolver procedimentos que implementem uma tarefa só, pois isto ajuda a facilitar o entendimento e portanto a sua manutenção).

As métricas selecionadas têm demonstrado grande eficácia e aceitação na quantificação da complexidade. Por exemplo, o estudo feito por [HARRI81] mostrou os seguintes resultados, ao comparar a efetividade de várias métricas para medir complexidade:

MÉTRICA	EVIDÊNCIA EMPÍRICA	SENSITIVA AO CONTEXTO	APLICAÇÃO GERAL	ABRANGÊNCIA
Linha de Código	2	1	3	1
Soft. Science	3	1	3	2
Extensão de Uso	1	1	3	2
Compl. Ciclomática	3	1	3	1

$1 = ruim$ $2 = regular$ $3 = bom$

A função usada para calcular a Dificuldade foi a mesma descrita no

Capítulo 4 desta tese, ou seja:

$$Dificuldade = \frac{\eta_1 * N_2}{2 * \eta_2}.$$

E para o esforço foi implementada a seguinte:

Esforço = Dificuldade * Volume.

Estas duas fórmulas calculam, de maneiras diferentes, o esforço que um programador tem de fazer para entender, programar e dar manutenção a um programa. Pode-se dizer que ambas as funções têm uma relação direta com a produtividade do programador. A produtividade pode ser calculada a partir das proporções: Programador-dias/dificuldade ou Programador-dias /Esforço. [CURTI79], [BAKER80], [FITZS78], [JAY85] e [SHEN83] concordam que estas métricas têm uma forte correlação com a produtividade de um programador.

Para calcular estas duas métricas, o analisador separa primeiro os operadores e operandos de cada procedimento lido e os coloca na lista correspondente. É importante lembrar que cada procedimento e função gera duas listas. Por exemplo, imagine que o analisador está lendo os comandos compostos do procedimento ORDENA e que ele acabou de ler o símbolo "a" da seguinte expressão: a[i] := b[i+1]. Pode-se perceber que o símbolo "a" corresponde a um operando (uma variável), e que, portanto, deve ser colocada na lista de operandos do procedimento que está sendo lido. O sistema AnaSoft coloca a variável "a" na lista de símbolos do procedimento ORDENA, coloca "n" no tipo, indicando que o símbolo é um operando e inicializa o total com 1 (um), se a variável não existe; se o símbolo já existe, acrescenta 1 (um) ao total correspondente, para atualizar a freqüência ou o número de vezes que ela já apareceu neste procedimento.

Por outro lado, se for "+" o símbolo lido, AnaSoft o coloca na lista, com a única diferença de que seu tipo será "r" para indicar que o símbolo é um operador. Se for uma palavra reservada que foi lida, como por exemplo BEGIN, o analisador a coloca na lista de operadores reservados, da mesma maneira como fez com os outros operandos e operadores. A diferença está em que as palavra-chave não precisam do "TIPO", pois todas elas são operadores. Depois de completar a leitura, AnaSoft calcula o Volume (Capítulo 4 desta tese), a Dificuldade, e por último verifica quais dos procedimentos

ou funções excedem o limite de complexidade e imprime um relatório. O Capítulo 6 mostra o formato do relatório. A complexidade ciclomática e o fluxo de informação são calculados de forma semelhante.

A função utilizada para calcular a Complexidade Ciclomática foi a seguinte:

V(G) = Soma(Decisões, AND's, OR's e NOT's) + 1.

O motivo pelo qual esta fórmula foi utilizada já foi explicado no Capítulo 4, mas vale a pena lembrar que ela simplifica o processo de quantificação. O método original de McCabe é muito complicado para ser implementado num programa e portanto foge do escopo deste trabalho; e, mais importante, o resultado é o mesmo. Além disso, a função original de McCabe não se preocupa com a maneira como os vértices do grafo do programa estão ligados, mas sim com quantos. Por este motivo, ela tem sido considerada como insensitiva ao contexto. O fluxo de informação é calculada pelo analisador para cada procedimento na medida em que vão sendo lidos.

O cálculo da complexidade do fluxo de informacão é feito de acordo com a fórmula apresentada por [HENRY81] com a modificação fornecida no começo desta seção. A relação é a seguinte:

$$Complexidade = Tamanho * (FAN - IN * FAN - OUT)^2$$

FAN-IN corresponde ao número de chamadas de procedimentos que a função ou o procedimento faz (não foi feita distinção entre procedimentos e funções oferecidas pela linguagem e aqueles definidos no próprio Software), e FAN-OUT ao número de procedimentos que chamam um determinado procedimento ou função.

Os padrões de complexidade utilizados e aceitos pelos pesquisadores e adotados nesta tese aparecem na Tabela 5.1.

Note-se que a métrica Fluxo de Informação não foi incluída na Tabela 5.1. Isto se deve ao fato de que, por ter sido modificada, os valores obtidos da análise de vários sistemas são apresentados no Capítulo 7.

MÉTRICA	*COMPLEXIDADE*		
	BAIXA	MEDIA	ALTA
COMPLEXIDADE CICLOMÁTICA	10	50	>100
DIFICULDADE	<10	<25	>50
ESFORÇO	10000	100000	>100000

Tabela 5.1: Padrões de Complexidade de Acordo com [JAY85]

Capítulo 6

Implementação

6.1 Estruturas de Dados do Sistema AnaSoft

Na implementação do Sistema AnaSoft foram usadas duas estruturas principais de dados: TPROC e LGM. A primeira corresponde a uma lista ligada usada para guardar as informações de cada função e procedimento definidos no programa analisado; e a segunda corresponde a um vetor contendo palavras reservadas da linguagem Pascal. A Figura 6.1 mostra a representação gráfica da primeira estrutura de dados, a TPROC:

Cada elemento da lista de procedimentos tem as seguintes informações: onde,

- **NOME**: nome do procedimento;
- **COMPL_CICLOM**: métrica da estrutura lógica;
- **DIFICULDADE**: métrica de Software Science;
- **ESFORÇO**: métrica de Software Science;
- **FLOW IN**: número de chamadas que um procedimento faz;
- **FLOW OUT**: número de vezes que um procedimento é chamado;
- **TOPRES**: apontador para o topo da lista de operadores reservado;
- **TLSIMB**: apontador para o topo da lista de símbolos;
- **ANT**: apontador auxiliar;

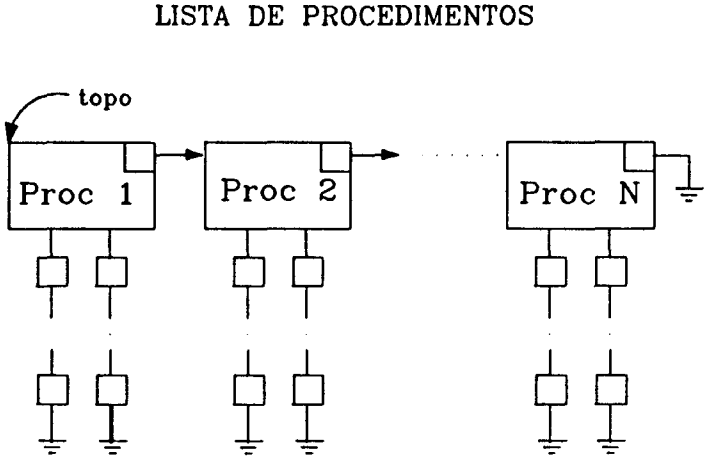

Figura 6.1: Estrutura de Dados Principal, TPROC, do Analisador

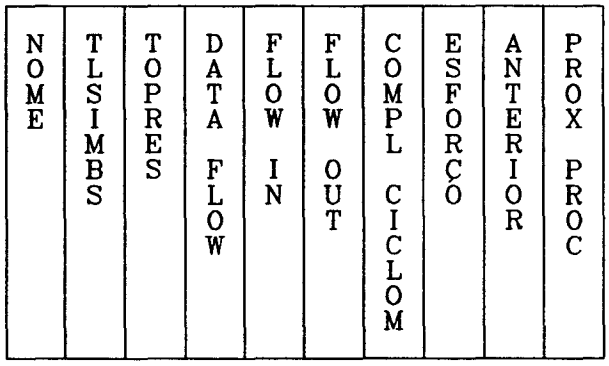

Figura 6.2: Atributos de um Elemento de Lista de Procedimentos

- **PROX_PROC**: apontador para o próximo procedimento;

Por exemplo, o procedimento que se segue, Procedimento 6.1, depois de lido, ficaria como na Figura 6.3.

```
1.  procedure proximo;
2.  begin
3.       if (eoln(Entr)) and (not(eof(Entr)) then
4.          begin
5.              readln(Entr);
6.              c := ' ';
7.          end
8.  end;
9.          if (eof(Entr)) then c := '\#'
10.         else
11.            begin
12.                read(Entr,c);
13.                if maxline = 81 then
14.                   begin
15.                       writeln(Diag);
16.                       maxline := 1;
17.                   end;
18.                if c <> ' ' then write(Diag,c);
```

```
19.                maxline := maxline + 1;

20. end;
```

Procedimento 6.3, lê caracteres e os copia em um arquivo.

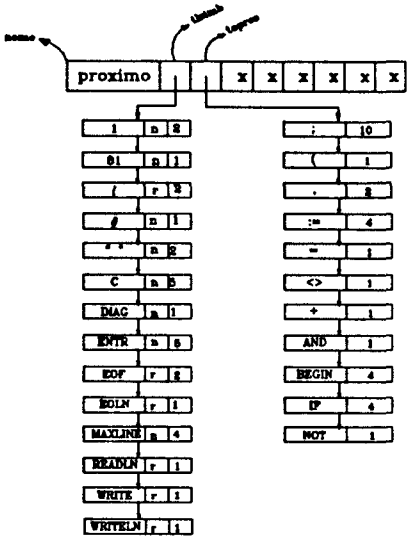

Figura 6.3: Procedimento Próximo depois de lido por AnaSoft

Lembre-se que o código "r" identifica o símbolo como um operador e "n" como operando; o número ao lado de cada símbolo representa a quantidade de vezes que ele aparece no procedimento. O símbolo "(", de acordo com a estratégia do Capítulo 6, pode desempenhar duas funções diferentes. Por exemplo, na linha 3 da Figura 6.3, a primeira vez ele aparece como um separador de expressão e a segunda, como um separador de parâmetros da função "EOLN()". Isso explica porque ele aparece nas duas listas do procedimento próximo da Figura 6.3.

O símbolo "81" aparece como um operando, "n" e com frequência 1 (um). Ou seja, ele só aparece uma vez no procedimento.

As Figuras seguintes mostram a estrutura da lista de símbolo e um elemento da lista em detalhe:

onde,

- **NOME**: próprio símbolo;

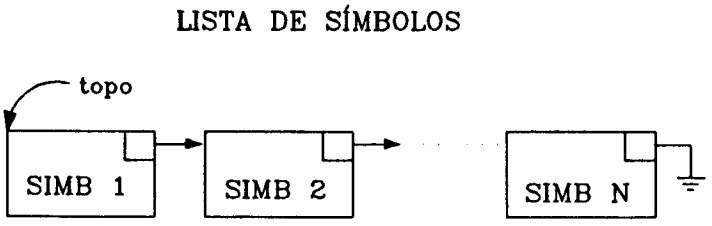

Figura 6.4: Estrutura da Lista de Símbolos de cada procedimento

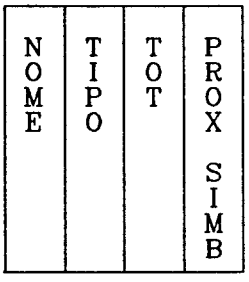

Figura 6.5: Atributos de um Elemento da Lista de Procedimentos

- **TIPO**: operando ou operador;
- **TOT**: freqüência dele no procedimento;
- **PROX_SIMB**: apontador para o próximo símbolo;

As métricas Dificuldade, Esforço e Infoflow seriam computadas no final, depois que o programa tenha sido lido completamente.

Para o exemplo do Procedimento 6.3 as métricas seriam:

$\eta_1 = 17$,
$\eta_2 = 7$,
$\eta = 24$,
$N_1 = 44$
$N_2 = 21$ e
$N = 65$.

$$Dificuldade = \frac{\eta_1}{2} * \frac{N_2}{\eta_2} = 8.5 * 3 = 25.5.$$

Esforço

$$= D * V = 26 * (65 log 24) = 5371.$$

$$Complexidade\ Ciclomática = a - v + 1 = 6.$$

$$FI = N * (Fluxo - In * Fluxo - Out)^2 = 65 * (7 * 5)^2 = 79625.$$

Para o Fluxo de Informação, suponha-se que o procedimento PROXIMO é chamado cinco vezes por outro procedimento.

A estrutura de operadores pré-definidos tem a mesma estrutura da lista de símbolos. A única diferença de que os operadores guardados aqui não tem o seu tipo especificado pois sabe-se que eles são operadores. Além destas estruturas, foi definido um conjunto de símbolos auxiliares, "pal_res", que ajudam a identificar os operadores e operandos na medida em que eles vão sendo lidos. Existe também a estrutura LGM, que auxilia a identificação das palavra-chaves. Esta última tem a forma de um vetor contendo os nomes das palavras reservadas da linguagem Turbo Pascal da seguinte forma:

PALAVRAS RESERVADAS = [LGMi]

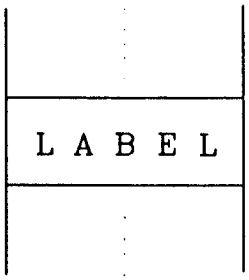

Figura 6.6: Figura

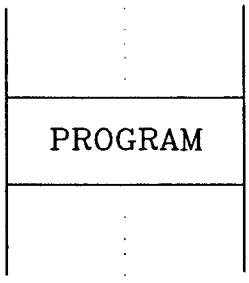

Figura 6.7:

Por exemplo, na Figura 6.6, o vetor LGH_i para "i = s_label" corresponde ao símbolo "LABEL".

Por exemplo, suponha-se que foi lido o símbolo "begin" da linha 2 do Procedimento 6.3. A função CÓDIGO, definida no analisador, procura o código do símbolo na tabela LGM. Ela devolveria o código "s_begin". Se o símbolo não estiver na tabela, como seria o caso da variável Maxline, por exemplo, a função devolve o código "s_identificador" indicando que o símbolo é uma variável. Desta maneira, separam-se os símbolos reservados dos não reservados.

6.2 Dificuldades Encontradas na Implementação

A dificuldade mais importante enfrentada na implementação da ferramenta aconteceu na seguinte situação. Suponha-se que se quer analisar um programa com o seguinte segmento de código:

```
procedure A(...);
var
...
procedure B(...);
var
...
procedure C(...);
begin C
...
end; C
begin B
...
end; B
begin A
...
end; A
```

Foi explicado que a ferramenta vai guardando as informações dos procedimentos a medida em que eles vão sendo lidos. Ou seja, quando a estrutura acima fosse analisada, o procedimento A seria o primeiro a ser classificado, e depois seguir-se-ia o procedimento B. Porém, no momento em que o procedimento B aparece para ser classificado as informações do procedimento A ainda não teriam acabado. Ainda faltaria o segmento "begin A ...end;

A ". O mesmo aconteceria com o procedimento C. Ou seja, em algum momento da análise de um programa é preciso suspender a classificação da informação de um procedimento, começar a classificação de algum outro e, depois, retomar a classificação do primeiro.

No caso da estrutura acima, a classificação começa com o procedimento A, que é suspensa para iniciar a classificação de B que, depois é suspensa para começar a classificação de C. Depois de completada a classificação de C, é retomada a classificação de B e finalmente a classificação de A.

Esta situação foi resolvida com a ajuda de um apontador auxiliar, ANT (Figura 6.2). O analisador atualiza cada apontador ANT (anterior) para que este sempre esteja apontando para o procedimento imediatamente acima, do nível de aninhamento. Por exemplo, suponha-se que na estrutura do exemplo acima o procedimento A tem nível 1, B nível 2 e C nível 3. No instante em que o procedimento C estiver sendo processado, o seu apontador estaria apontando para B. A Figura seguinte mostra o estado da estrutura.

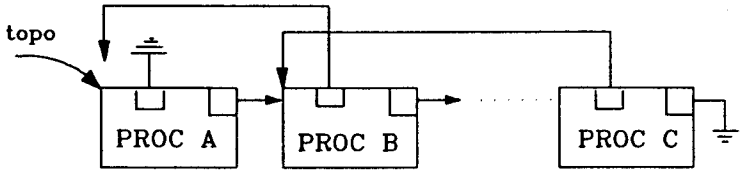

Figura 6.8: Estrutura de Dados TPROC

Uma vez concluída a classificação do procedimento C, é fácil saber qual era o procedimento que estava sendo processado antes. Como a Figura 6.8 mostra, é só seguir o apontador, ANT, de C para chegar até B. Aninhamento é uma prática muito comum na linguagem Pascal para definir procedimentos locais (somente acessíveis dentro do procedimento em que estão encaixados).

6.3 Entrada e Saída

A entrada do analisador é um programa escrito em qualquer versão de 1 (um) até 4 de Turbo Pascal. A primeira tela do Analisador é a seguinte:

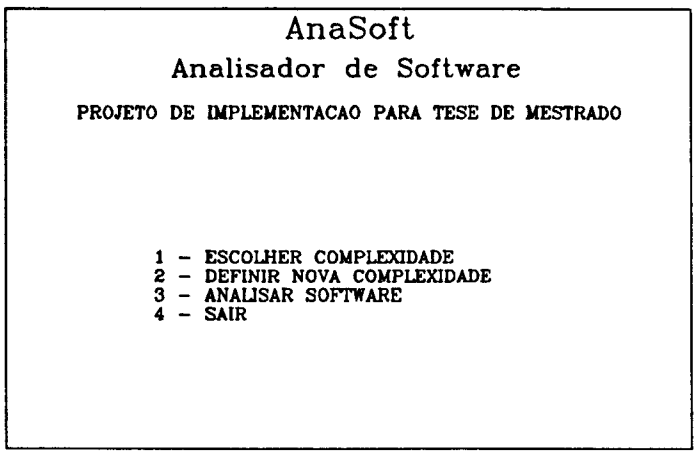

Figura 6.9: Tela 1

Esta tela, como está indicado, permite ao usuário escolher o nível de complexidade desejado (opção 1); definir seu próprio nível (opção 2). Também permite entrar com o nome do programa (opção 3) a que se quer analisar e, finalmente, terminar a seção (opção 4).

A segunda tela é mostrada na Figura 6.10

Esta tela aparece depois que o usuário selecionou a opção 1 da tela anterior. Ela permite ao usuário escolher o nível de complexidade desejado.

A terceira tela é mostrada na Figura 6.11

Esta tela permite ao usuário entrar com os valores de complexidade desejados (opção 2 da tela anterior).

A última tela, mostrada a seguir, permite ao usuário entrar com o nome do programa a ser analisado ou retornar à tela anterior.

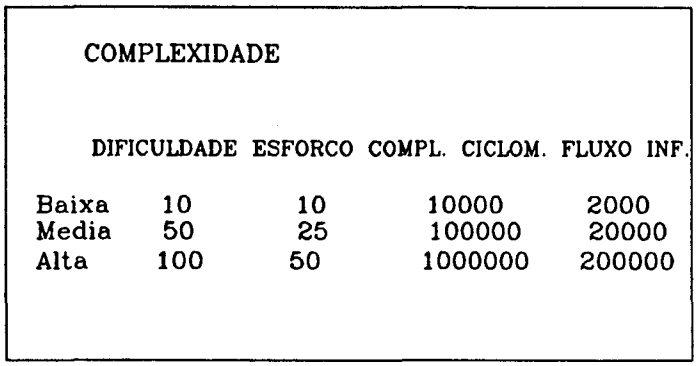

Figura 6.10: Tela 2

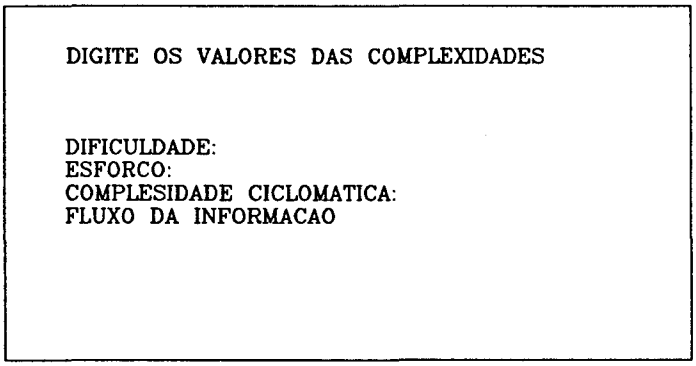

Figura 6.11: Tela 3

NOME DO PROGRAMA:

A saída do analisador é um relatório com o seguinte formato:

Figura 6.12: Tela 5

6.4 Informações Técnicas do Analisador

AnaSoft tem 2092 linhas de código distribuídas em três módulos. O tempo médio de processamento de um programa de 700 linhas é de aproximadamente 5 minutos. Este foi desenvolvido em um microcomputador compatível com IBM PC. A linguagem utilizada para a sua implementação foi Pascal, e o compilador Turbo Pascal versão 4.0.

Capítulo 7

Resultados

7.1 Introdução

A ferramenta proposta neste trabalho foi aplicada a 3 (três) diferentes Software incluindo a própria ferramenta. Isto foi feito com o objetivo de avaliar a sua eficácia em identificar procedimentos e funções com complexidade alta, que poderiam ter seu Teste e Manutenção prejudicados.

Como um dos objetivos da ferramenta proposta neste trabalho é auxiliar o programador na tarefa de Manutenção, é importante saber que existem duas maneiras de relacionar as métricas para Software e a Manutenção [KAFURA87].

A primeira, chamada de técnica **objetiva**, consiste em se fazer uma análise de correlação entre as medidas da complexidade de um componente de um Software e o tempo que demora ou se gasta em fazer a sua Manutenção. Esta técnica, também chamada de quantitativa, é muito utilizada para estudar as fases de desenvolvimento do Ciclo de Vida. A segunda técnica, chamada de **subjetiva**, consiste em tentar estabelecer uma relação entre o resultado quantitativo da aplicação das mètricas e o julgamento ou parecer dos especialistas que conhecem o Software profundamente. Esta foi a técnica utilizada neste trabalho. Este método não é desconhecido e é utilizado na atualidade em vários estudos, como por exemplo [SYMON88].

Um dos motivos pelo qual esta técnica foi utilizada foi para verificar se as métricas para Software são capazes de prover ao programador encarregado da Manutenção de informação consistente de acordo com a opinião dos especialistas. Esta informação poderá ser utilizada como guia para evitar e corrigir Manutenção deficiente. Um outro motivo para a utilização desta técnica foi o fato de que os Software analisados não possuiam informação

histórica sobre seu desenvolvimento (como por exemplo, número de erros encontrados, tempo de desenvolvimento e outros). Esta informação é necessária para se fazer Análise de Correlação. Recomenda-se, porém, que ambas as técnicas, objetiva e subjetiva, sejam utilizadas, sempre que seja possível, pois desta maneira os resultados terão mais credibilidade e a probabilidade de erro será menor.

O tamanho médio, dos Software analisados, em linhas de código compilável, segundo o compilador para a linguagem Turbo Pascal versão 4.0, é de 2,719 e o tempo médio de análise 4,40 minutos. Este tempo não inclui o tempo de impressão do relatório pois ele varia de acordo com a velocidade da impressora. Em outras palavras, o tempo medido foi o tempo de análise dos procedimentos e do cálculo das métricas.

7.2 Estratégia para Análise

Cada Software analisado foi avaliado com o nível de complexidade **baixo** (Capítulo 5). Os procedimentos e funções com complexidade maior que aquela estabelecida pelo tipo de complexidade baixa são identificados pela ferramenta para serem revisados. Depois de selecionados os procedimentos, foram escolhidos os quatros com dificuldade, esforço, complexidade ciclomática ou fluxo da informação mais altos, e brevemente comentados. Antes de se fazer a análise, foi pedido aos autores dos Software desenvolvidos que escolhessem os quatros (ou mais) procedimentos/funções mais complexos, de acordo com a dificuldade de Implementação, Teste e Modificação. A resposta foi comparada com o resultado do analisador. Naqueles casos em que a complexidade do procedimento foi muito altas, foi fornecida uma breve explicação da possível causa. Nas seções que se seguem são descritos os resultados.

Resultado 1.

O primeiro Software analisado foi a própria ferramenta, AnaSoft. Ela tem 2092 linhas de código compilável distribuidas em 46 procedimentos. O tempo de análise foi de 2,05 minutos.

Entre os procedimentos identificados, INSERESIMB teve a dificuldade mais alta, 103. Este procedimento é, na verdade, um dos mais complexos pois encarrega-se de colocar cada símbolo não reservado, para os símbolos

reservados há um outro procedimento, em uma lista de símbolos que é mantida em ordem alfabética (Capítulo 5). Primeiro, INSERESIMB verifica se o símbolo está na lista e, dependendo disto, atualiza as informações dele. Este procedimento, inseresimb(), desempenha várias funções, como por exemplo busca do símbolo na lista, insere se ele não existe, atualiza informação se ele já existe, que podem ser separadas e implementadas em outros procedimentos, para desta maneira simplificá-lo. Segue-se o código deste procedimento.

```
PROCEDURE INSERESIMB(VAR toposimb:lista_simb;atomo:alfa;tipo:categoria);
VAR
  nsimb,aux1,aux2:lista_simb;

BEGIN { INSERESIMBOLO }

    if toposimb = nil then { se lista vazia }
       begin
           new(toposimb);
           toposimb^.nome := atomo;
           toposimb^.tipo := tipo;
           toposimb^.tot  := 1;
           toposimb^.prox_simb:= nil;
       end
    else    { se nao }
       begin
 { inicializa punteros auxiliares }
           aux1:= toposimb;
           aux2:= toposimb;
           while (aux2 <> nil) and (aux2^.nome < atomo) do
                 begin
                     aux1:= aux2;  { avanza ate achar lugar }
                     aux2:=aux2^.prox_simb; { correto na lista }
                 end;
           if aux2^.nome = atomo then
           { se simbolo ja existe, atualiza }
              begin                      { tot }
                  aux2^.tot:= aux2^.tot + 1;
              end
           else
```

```
        if aux2 = aux1 then
      { se inicio da lista }
         begin
            if aux1^.nome = atomo then
            { se e o simb procurado }
               begin
                  aux1^.tot:= aux1^.tot + 1;  { atualiza
}
               end
            else            { se nao }
               begin
                  new(nsimb);
                  nsimb^.nome:= atomo;
                  nsimb^.tipo := tipo;
                  nsimb^.tot  := 1;
                  nsimb^.prox_simb:= toposimb;
                  toposimb:=nsimb;
               end;
         end
      else { se nao }
         if aux2 = nil then      { se o fim da lista }
            begin                { insere no fim     }
               new(aux1^.prox_simb);
               aux2:=aux1^.prox_simb;
               aux2^.nome:=atomo;
               aux2^.tipo := tipo;
               aux2^.tot  := 1;
               aux2^.prox_simb:=nil;
            end
         else  { se nao, insere no meio da lista }
            begin
               new(nsimb);
               nsimb^.nome:=atomo;
               nsimb^.tipo := tipo;
               nsimb^.tot  := 1;
               nsimb^.prox_simb:=aux2;
               aux1^.prox_simb:=nsimb;
            end;
   end;
```

```
END;{ INSERESIMBOLO }
```

Além deste, a ferramenta identificou outros três procedimentos, INSEREPROC, INSEREOPRES e PEGASIMB que apresentam complexidade alta em relação à métrica da Dificuldade. Os dois primeiros foram implementados da mesma maneira que INSERESIMB e, portanto, não é surpresa que eles tenham sido identificados. Em relação a métrica do Esforço, os mesmos três primeiros procedimentos apresentam a complexidade mais alta.

Quando se observou a métrica da Complexidade Ciclomática, o procedimento PEGASIMB mostrou a mais alta (50). Isto se deve, em parte, a que este procedimento contem um comando "CASE" muito grande que aumenta muito o número de caminhos. Este procedimento é responsável pela separação dos carateres lidos em símbolos, por exemplo, a sequência E-N-D corresponde ao símbolo END; o procedimento PEGASIMB identifica se o símbolo é um identificador, número, comentário ou algum outro símbolo. A sua complexidade também pode ser reduzida através da separação de algumas de suas tarefas.

A última métrica observada nos procedimentos selecionados foi o Fluxo da Informação. Porém, ela não foi utilizada para selecionar procedimentos, a não ser de uma maneira experimental, para observar seu desempenho e propor valores a serem utilizados e refinados de acordo com o ambiente e o tipo de Software que se quer desenvolver. O procedimento que apresentou maior fluxo de informação foi PEGASIMB. Isto se justifica pela sua função, descrita acima; ele é chamado por todos os procedimentos que correspondem a carta sintática de Turbo Pascal versão 4.0, como por exemplo BLOCO, COMANDO_COMPOSTO, ATRIBUIÇÃO e outros. Ou seja, cada vez que um procedimento precisa ler o próximo símbolo PEGASIMB é chamado. Isto acontece constantemente durante a leitura do programa a ser analisado. Por este motivo a interação dele com o seu meio ambiente é muito intensa. Por estas características estarem presente seu teste e modificação não são simples. Os três procedimentos mencionados anteriormente também apresentaram níveis de Fluxo de Informação altos. Isto se deve a que cada símbolo tem que ser inserido em sua lista correspondente, já que a ferramenta utiliza três listas de símbolos: uma para símbolos comuns, não reservados; outra para símbolos reservados e outra para procedimentos e funções.

Resultado 2.

O segundo Software analisado foi desenvolvido para calcular a série Qui-quadrado. Ele avalia a função Qui-quadrado e os pontos percentuais de distribuição da série utilizando o polinômio de Bernoulli.

O Software tem 466 linhas de código compilável e o tempo de Análise foi de 0,85 minutos. Ele possui um total de 25 procedimentos.

AnaSoft selecionou a função B15 como tendo a complexidade mais alta (44) em relação à dificuldade. Segundo o autor esta função calcula o décimo quinto coeficiente numérico do polinômio de Bernoulli correspondente a B_j (para j = 15) na função acima. A função com o maior esforço, segundo a ferramenta, é BERNO. Esta é responsavel pelo cálculo do polinômio de Bernoulli. Ela também apresenta o maior fluxo de informação pois faz um grande número de chamadas a procedimentos pré-definidos.

A função B, usada também no cálculo de B_j, apresentou a complexidade ciclomática maior. Isto se deve também a que ela contém um comando "CASE" muito grande. Este Software apresenta uma estruturação muito boa, visando otimizar a eficiência, razão pela qual não foi sugerida nenhuma modificação.

Comparando os resultados emitidos por esta ferramenta e os comentários do autor do Software analisado, observou-se que os procedimentos mais complexos foram igualmente selecionados por ambos.

Resultado 3.

O terceiro Software analisado foi um compilador, chamado Fig, de uma linguagem para especificação de figuras . Ele tem 5600 linhas de código compiláveis, distribuidas em 105 procedimentos. O tempo de Análise foi de 10,9 minutos.

De acordo com a métrica da dificuldade, o procedimento FATOR apresentou o nível de complexidade maior entre todos aqueles analisados (106). A sua função consiste em processar os elementos básicos de uma expressão e colocá-los em uma árvore. Este mesmo procedimento também apresentou o nível de complexidade mais alta de acordo com a métrica do esforço (3.462.204). A sua revisão manual mostra que, na realidade, o procedimento FATOR é extremamente grande e complicado. Este procedimento contém um comando "CASE" excessivamente complexo. Para se ter uma idéia da complexidade deste procedimento e a dificuldade que ele apresenta para ser

mantido, reproduziu-se, a seguir, um fragmento de uma das opções de um dos comandos "CASE" utilizado por ele.

```
c_id:BEGIN
  e_constante := false ;
  q := NIL ;
  verifica(atomo,q,pos_fig_atu) ;
  IF q = NIL THEN
    BEGIN
     verifica(atomo,q,inic_tab_simb) ;
     if q = nil then
       if simbolo2 <> c_ponto then
          begin
           ap^.sel := s_identificador ;
           new(q) ;
           q^.categoria := var_local ;
           insere(atomo,q) ;
           new(q^.t1) ;
           q^.t1^.nome := indefinido ;
           q^.t1^.t2 := nil ;
           tp := q^.t1 ;
           ap^.id := q^.id
          end
       else
         begin
          atomoaux1 := atomo ;
          analex(fonte) ;
          analex(fonte) ;
          if simbolo2 <> c_ponto then
            begin
             if (atomo = 'X') or (atomo = 'Y') then
               begin
                ap^.sel := s_rot_res_pto ;
                new(q) ;
                q^.categoria := indefinida ;
                insere(atomoaux1,q) ;
                ap^.ap_categ := q ;
                ap^.resultado := nil ;
                new(ap^.coordenada) ;
```

```
if atomo = 'X' then
  begin
   new(q^.x) ;
   q^.x^.nome := indefinido ;
   q^.x^.t2 := nil ;
   q^.y := nil ;
   tp := q^.x ;
  end
else
 begin
  new(q^.y) ;
  q^.y^.nome := indefinido ;
  q^.y^.t2 := nil ;
  q^.x := nil ;
  tp := q^.y ;
 end ;
 ap^.coordenada^.c := atomo[1] ;
 ap^.coordenada^.prox := nil
end
else ...
```

Pode-se observar que o teste desta opção é bastante elaborado pois ele contém outros comandos condicionais, que aumentam enormemente o nível de complexidade. Este procedimento representa um ponto ideal por onde começar a simplificação do Software. Uma sugestão para reduzir a complexidade seria tornar este tipo de opção em um procedimento. Desta maneira a estrutura "c-id" ficaria da seguinte maneira:

case — of—-

...

c_id:processa_c_id;

...

onde processa_c-id seria definido previamente como um procedimento. É provável que este tipo de mudança afetaria a eficiência do Software, porém aumentaria a sua facilidade de teste, clareza e manutenção.

O procedimento IMPRIME_EXPRESSÃO apresentou a maior complexidade ciclomática (169). Este valor é também excesivamente alto e pode ser interpretado como o número de caminhos linearmente independentes que seriam necessários para testar o procedimento. Para qualquer programador

poder modificar este procedimento seria necessário um esforço muito grande pois para isto ele teria que primeiro entendê-lo.

Um dos fatores que aumentam a complexidade do procedimento IMPRIME_EXPRESSÃO acontece porque ele contém vários comandos "CASE" aninhados um dentro do outro que, por sua vez, mostram alternativas com vários comandos condicionais. Em vários casos, os comandos que são parte de uma alternativa poderiam ser separados em um único procedimento e, desta maneira, diminuir a complexidade.

Todos os procedimentos identificados pelo analisador AnaSoft como futuros pontos críticos, "trouble spots", foram citados pelo programador que desenvolveu o Software dentro do conjunto dos quatro ou mais procedimentos com complexidade mais alta. Isto pode ser considerado como prova inicial do funcionamento e eficácia da ferramenta proposta nesta tese.

Capítulo 8

Conclusão

8.1 Contribuição

Esta tese propõe um estudo das métricas de Software mais importantes para medir complexidade e o desenvolvimento de uma ferramenta baseada em três destas métricas, AnaSoft, que permita a Análise de programas escritos na linguagem Turbo Pascal, visando identificar procedimentos e funções que podem dificultar a sua manutenção. A ferramenta está baseada em várias métricas para medir complexidade de Software, escolhidas por serem consideradas as mais eficientes, melhores e de maior aceitação na área de métricas para Software. Além disso, este trabalho propõe uma simplificação da métrica proposta por [HENRY81] que tornam mais simples e eficiente o desempenho da ferramenta.

Os resultados obtidos com a aplicação da métrica em vários sistemas demonstram que a ferramenta é muito útil para auxiliar o processo de Desenvolvimento e Manutenção de Software. Como vantagens adicionais, pode-se citar que ela:

- contribui para a diminuição do tempo de desenvolvimento de Software, pois tem grande impacto nas fases de Teste e Manutenção. Os recursos humanos e econômicos para Teste e Manutenção podem ser alocados mais eficientemente, a partir do momento em que se tem uma idéia da dificuldade e do esforço necessário para realizar estas tarefas.

- contribui para facilitar a quantificação e definição dos parâmetros padrões utilizados para se comparar a complexidade dos módulos; ou seja, cada empresa que desenvolve Software pode aplicar a ferramenta

a seu próprio ambiente e ajustá-la de acordo com o nível de complexidade desejado, obtendo seus próprios valores-padrão.

- permite avaliar a complexidade de quatro maneiras diferentes: do ponto de vista do esforço do programador, da dificuldade que o programa apresenta, do fluxo de controle do programa e finalmente do fluxo de informação. Mais importante ainda, isto é feito antes do produto ser entregue ao usuário. É muito mais conveniente detectar pontos que, no futuro poderiam se tornar prejudiciais para o sistema, antes de entregar o Software ao usuário.

- permite identificar procedimentos e módulos altamente complexos que no futuro, poderiam agravar e deixar mais custosa a manutenção. Isto é feito através da quantificação e avaliação da complexidade de cada procedimento e função que faz parte do sistema.

- É simples, relativamente rápida e fácil de usar. Esta rapidez foi obtida pela maneira que a ferramenta foi implementada, e com a simplificação da métrica do Fluxo da Informação, proposta neste trabalho.

- Foi desenvolvida utilizando-se os conceitos de Modularização e Abstração de dados; estas técnicas tornam a ferramenta muito fácil de entendê-la, usá-la e modificá-la.

- Foi possível a modificação da métrica do fluxo da informação e desta maneira fazer uma avaliação mais realista de um programa de acordo com as novas tendências na área da Engenharia de Software.

- Foi desenvolvida para ser aplicada em programas implementados na linguagem Turbo Pascal Versão 4.0, mas pode ser aplicada, depois de simples modificações, em programas escritos nas versões anteriores e posteriores.

A limitação mais importante da ferramenta é que a sua aplicação só é válida para programas desenvolvidos para a linguagem Turbo Pascal 4.0 e versões anteriores. A sua organização e divisão em módulos lhe dá flexibilidade para ser modificada e aplicada em outras versões, inclusive em outras Linguagens de Programação.

8.2 Extensões

Os resultados desta tese sugerem as seguintes extensões para pesquisas futuras:

1. A modificação da ferramenta para ser aplicada em programas que sejam implementados utilizando-se módulos independentes através do recurso "unit" disponível na versão de Turbo Pascal 4.0.

2. A aplicação da ferramenta em ambientes de programação reais, para reavaliar a efetividade e habilidade da ferramenta em identificar módulos potencialmente problemáticos e confirmar os resultados iniciais obtidos neste trabalho.

3. A aplicação deste analisador em programas nas diversas áreas, como por exemplo comercial, científica e de aplicativos em geral, para determinar o comportamento da complexidade em cada uma destas áreas.

4. A aplicação da ferramenta em sistemas implementados, utilizando-se técnicas diferentes de desenvolvimento, para determinar a influência destas na manutenção.

5. A inclusão de outras métricas (Linha de Código no lugar de "N" no Fluxo de Informação, por exemplo), para estudar o seu comportamento e o seu impacto na linguagem Turbo Pascal ou outras linguagens.

6. A possibilidade de modificar a ferramenta para ser aplicada em outras linguagens, depois que a sua eficiência tenha sido amplamente demonstrada.

REFERÊNCIAS

[BAKER80] Baker, A. L.,e S. H. Zweben, ''A Comparison of Measure o Control Flow Complexity", IEEE Trans. on Soft. Eng., Vol SE-6, No 6, November 80, Pag. 506-512.

[BALZE85] Balzer, R., ''A 15 Year Perspective on Automatic Programming", IEEE Trans. on Soft. Eng., Vol SE-11, No 11, November 85, Pag. 1257-1268.

[BARST85] Barstow, D. R., ''Domain-specific Automatic Programming", IEEE Trans. on Soft. Eng., Vol SE-11, No. 11, November 85, pag. 1321-1336.

[BERGL82] Bergland, G. D., ''A Guide Tour of Program Design Methodologies", IEEE Computer, October 82, Pag.13-37.

[BOEHM73] Boehm, B. W., ''Software Impact: A Quantitative Assessment", Datamation, March 73, Pag. 48-59.

[BOEHM76] Boehm, B. W., ''Software Engineering", IEEE Trans. on Soft. Eng., Vol. CE-25, No. 12, December 76, Pag. 1226-1240.

[BOEHM78] Boehm, B. W. e outros, ''Carateristics of Software Quality", Elsevier North-Holland, 1978.

[BOEHM81] Boehm, B. W., ''Software Engineering Economics", Prentice-Hall, 1981.

[BOEHM88] Boehm, B. W. e Philip N. Papaccio, ''Understanding and Controlling Software Costs", IEEE Trans. on Soft. Eng., Vol. 14, No. 10, October 88, pag. 1462-1477.

[BOOCH86] Booch, G., ''Object Oriented Development", IEEE Trans. on Soft. Eng., Vol. SE-12, No. 2, February 86.

[CONTE86] Conte, S. D. e outros, ''Software Engineering Metrics and Models", California, Benjaming Cummings Pub. Co., 1986.

[COULT83] Coulter, N. S., ''Software Science Cognitive Psychology", IEEE Trans. on Soft. Eng., Vol. SE-9, No. 2., March 83, Pag. 166-171.

[CURTI79] Curtis, B. S. e outros, ''Measuring The Psychological Complexity of Software Maintenance Task with the Halstead and McCabe Metrics", IEE Trans. on Soft. Eng., Vol. SE-5, No. 2, March 79, Pag.96-104.

[DAVIS88] Davis, A. M. e outros, ''A Strategy for Comparing Alternative Software Development Life Cycle Models", IEEE Trans. on Soft. Eng., Vol. 14, No. 10, October 88, Pag. 1453-1461.

[DAVIJ88] Davis, J. S. e outros, ''A Study of The Applicability of Complexity Measures", IEEE Trans. on Soft. Eng., Vol. 14, No. 9 September 88, Pag. 1366-1375.

[DRUMM87] Drummond, R. e H. Liesemberg, A-HAND - Ambiente de Desenvolvimento de Software Baseado em Hierarquias de Abstracao em Niveis Diferenciados, DCC UNICAMP, April 1987.

[DUNSM79] Dunsmore, H. E.e outros, ''Data Referencing: An Empirical Investigation", IEEE Computer, Vol 12, No. 12, December 79, pag. 50-59.

[ELSHO76] Elshoff, J. L., ''An Analysis of Some Comercial PL/1 Programs", IEEE Trans. on Soft. Eng., Vol. SE-2, No. 2, June 76, Pag. 113-120.

[EVANS87] Evans, M. W. e J. Marciniak ''Software Quality Assurance", John Wiley \& Sons Inc., 1987.

[FITZS78] Fitzsimmons, E. e T. Love, ''A Review and Evaluation of Software Science", Computing Survey, Vol. 10, No. 1, March 78, Pag. 78, Pag. 2-18.

[GOLDG86] Goldberg, R., ''Software Engineering An Emerging Discipline", IBM System Journal, Vol 25, NOS 3/4, 1986, pag 334-353.

[HALBE86] Halbert, D. C. e P. D. O'Brien, ''Using Types and Inheritance in Object Oriented Programming", IEE Trans. on Soft. Eng., September 86, Pag. 71-79.

[HALST77] Halstead, M. H., ''Elements of Software Science", Elsevier North-Holland Inc., 1977.

[HARRI82] Harrison, W. e outros, ''Applying Software Complexity Metrics to Program Maintenance", IEEE Computer, Vol. 2, No. 4, September 82, pag. 65-79.

[HENRY81] Henry, S. e D. Kafura, ''Software Structure Metrics based on Information Flow", IEEE Trans. on Soft.Eng., Vol. SE-7, No. 5, September 81, Pag. 510-518.

[JACKS75] Jackson, M., A., ''Principles of Programming Design", Academic Press, N.Y., 1975.

[JAY85] Jay, L., A., ''Measuring Programmer Produtivity and Software Quality", John Wiley \& Sons Inc., N Y, 1985.

[KAFUR87] Kafura, D., ''The Use of Software Complexity Metrics in Software Maintenance", IEEE Trans. on Soft.Eng., Vol. SE-13, No. 3, March 87, Pag. 335-343.

[LEW88] Lew, K. S., ''Software Complexity and Its Impact on Software Reliability", IEEE Trans. on Soft.Eng., Vol. 14, No. 11, November 88, Pag. 1645-1655.

[LI87] Li, H. F., ''An Empirical Study of Software Metrics", IEEE Trans. on Soft.Eng., Vol. SE-13, No. 6, June 87, Pag. 697-708.

[LISKO74] Liskov, B. H. e J. Guttag, ''Abstraction and Specification in Program Development", Cambridge Mass., MIT 1986.

[LIPSC76] Lipschutz, S., ''Theoru and Problems of Discrete Mathematics", Mc Graw-Hill Inc., N. Y., 1976.

[McCAB76] McCabe, T. S., ''A Complexity measure", IEEE Trans. on Soft. Eng., Vol. SE-2, No. 4, December 1976, Pag. 510-518.

[MEDEI88] Medeiros, C. B. e outros, PAFUNCIO: Uma Ferramenta de Avaliacao de Desempenho de Aplicacoes, nao publicado.

[MILLS86] Mills, H. D., ''Structure Programming: Retrospect and Prospect", IEEE Software, November 86, Pag. 58-66.

[MOHAN79] Mohanty, S.N., ''Models and Measurement for Quality Assessment os Software", Computer Survey, Vol. 11, No. 3, September 79, pag. 251-275.

[RABIN77] Rabin, M. O., ''Complexity of Computations", Communication of the ACM, Vol. 20, No. 9, September 77, Pag. 625-633.

[RAMAM88] Ramamurthy, B. e Austin Melton ''A Synthesis of Software Science and The Ciclomatic Number", IEEE Trans. on Soft.Eng., Vol. 14, No. 8, Agost 88, Pag. 1116-1121.

[SCHNE79] Schneider, N. F. e outros, ''An Experiment in Software Errors Data Collection and Analysis", IEEE Trans. Soft. Eng., Vol. SE-5, No. 3, March 79, pag. 276-286.

[SHATZ88] Shatz, S. M., ''Towards Complexity Metrics for Ada Tasking", IEEE Trans. on Soft.Eng., Vol. 14, No. 8, Agost 88, Pag. 11222-1127.

[SHEN83] Shen, V. Y e outros, ''Software Science Revisited: A Critical Analysis of the Theory and its Empirical Support", IEEE Trans. on Soft. Eng., Vol. SE-9, No. 2, March 83, Pag. 155-165.

[SYMON88] Symons, C. R., ''Function Point Analysis: Difículties and Improvements", IEEE Trans. on Soft.Eng., Vol. 14, No. 1, January 88, Pag. 2-11.

[WOODF76] Woodfield, S. N., ''An Experiment on Unit Increase in Problem Complexity", IEEE Trans. on Soft. Eng., Vol. SE-7, No. 2, March 79, Pag. 76-79.

[WOODW79] Woodward, M. R. e outros, ''A Measure of Control Flow Complexity in Program text", IEEE Trans. Soft. Eng., Vol. SE-5, No. 1, January 1979, pag. 45-50.