UNIVERSIDADE FEDERAL DA PARAÍBA

CENTRO DE CIÊNCIAS E TECNOLOGIA

CURSO DE MESTRADO EM INFORMÁTICA

ESTUDO DE VALIDAÇÃO DE MÉTRICAS APLICADAS ÀS LINGUAGENS C E PASCAL

SAULO DE ARAUJO PEREIRA

CAMPINA GRANDE - PB

JUNHO - 1991

SAULO DE ARAUJO PEREIRA

ESTUDO DE VALIDAÇÃO DE MÉTRICAS APLICADAS ÀS LINGUAGENS C E PASCAL

Dissertação apresentada ao curso de Mestrado em Informática da Universidade Federal da Paraíba em cumprimento às exigências para obtenção do grau de mestre

ÁREA DE CONCENTRAÇÃO: CIÊNCIAS DA COMPUTAÇÃO

JOSÉ ANTÃO BELTRÃO MOURA
Orientador

MARIA DE FÁTIMA CAMÊLO
Co-orientadora

CAMPINA GRANDE - PB

JUNHO - 1991

P436e Pereira, Saulo de Araujo
Estudo de validacao de metricas aplicadas as linguagens C e Pascal / Saulo de Araujo Pereira. - Campina Grande, 1991.
140 f. : il.

Dissertacao (Mestrado em Informatica) - Universidade Federal da Paraiba, Centro de Ciencias e Tecnologia.

1. Linguagem de Programacao de Computador 2. Pascal (Linguagem de Programacao de Computador) 3. Dissertacao I. Moura, Jose Antao Beltrao, Dr. II. Camelo, Maria de Fatima, M.Sc. III. Universidade Federal da Paraiba - Campina Grande (PB) IV. Título

CDU 004.43(043)

ESTUDO DE VALIDAÇÃO DE METRICAS APLICADAS

ÀS LINGUAGENS C E PASCAL

SAULO DE ARAUJO PEREIRA

DISSERTAÇÃO APROVADA EM: 04/06/91

BANCA EXAMINADORA:

JOSE ANTÃO BELTRÃO MOURA, PhD

Presidente

MARIA DE FATIMA CAMELO, M.Sc

SILVIO ROMERO DE LEMOS MEIRA, PhD

MARIO TOYOTARO HATTORI, M.Sc

CAMPINA GRANDE

JUNHO - 1991

AGRADECIMENTOS

O autor gostaria de agradecer:

Aos orientadores Antão e Fátima, não só pela grande sabedoria e experiência, mas principalmente pela disposição em compartilhar seus conhecimentos;

À Infocon e Tecnal, por terem colocado seus programas fonte à disposição;

A Pedro Nicoletti, Rostand, Reginaldo, Dênio e Nivaldo, por terem colaborado na coleta de dados;

A UFPB/COPIN, PAQTC-PB e CAPES, pelo uso dos equipamentos e suporte financeiro;

A Shirley, pelo imenso apoio.

RESUMO

O uso fundamentado de métricas (padrões de medição) é essencial para a execução de projetos de software de forma mensurável, cujo desenvolvimento possa ser previsto, monitorado e aprimorado. Entre as métricas para a fase de codificação apresentadas na literatura, se destacam o **número ciclomático** e as métricas da **Ciência de Software**, pelo seu fundamento teórico e suporte empírico, e as métricas de **linhas de código** e **número de rotinas**, pela sua popularidade e simplicidade.

Este trabalho verifica a validade destas métricas aplicadas às linguagens **Pascal** e **C**. Para isto, uma ferramenta para a obtenção automática das métricas é desenvolvida. As análises, feitas em uma amostra de 1.670 rotinas, utilizam técnicas de regressão e baseiam-se em cinco parâmetros estatísticos.

Algumas das métricas estudadas mostram-se válidas na estimativa de tamanho de programa (métricas estimadoras do comprimento $\hat{N}$ e $\hat{N}_j$) e na estimativa de tempo de desenvolvimento de programa (métricas de comprimento e volume da Ciência de Software, linhas de código e número ciclomático). Métricas para estimativa de tamanho e tempo de desenvolvimento de rotinas e métricas para outras aplicações, como avaliação de nível de linguagem e conteúdo de informação de rotina, mostram-se imprecisas ou inconsistentes.

ABSTRACT

The utilization of metrics (measurement standards) is essential for the execution of software projects in a measurable way, whose development can be predicted, monitored and improved. Among the most important metrics are the cyclomatic number and the Software Science metrics – remarkable for their theoretical basis and empirical support – and the number of lines of code and routines – remarkable for their simplicity and popularity.

This dissertation verifies the validity of those metrics when applied to Pascal and C programs. A tool is developed for the automatic collection of the data used by the metrics. The analyses consider 1,670 routines and use regression and other statistical methods.

Some of the studied metrics appear to be valid for the estimation of program size ($\hat{N}$ and $\hat{N}_j$) and for the estimation of program development time (the Software Science size and volume metrics, lines of code and cyclomatic number). Metrics for estimation of routine size and development time, and for evaluation of language level and information content of routine appear imprecise or inconsistent.

ÍNDICE

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS

As seguintes abreviaturas são utilizadas ao longo deste trabalho.

Parâmetros Estatísticos:

- r: coeficiente de correlação estatística;
- a: primeiro coeficiente da equação de regressão. Se a equação é de uma reta, "a" é a interseção da reta com o eixo y;
- b: segundo coeficiente da equação de regressão. Numa reta, é o coeficiente angular;
- c: terceiro coeficiente da equação de regressão;
- DRM: desvio relativo médio;
- DQM: desvio quadrático médio;
- Prev(0,25): percentagem de prognósticos com erro menor que 25%.

Métricas da Ciência de Software:

- N, $\hat{N}$ e $\hat{N}_j$: comprimentos real, estimado de Halstead e estimado de Jensen;
- η_1 e η_2: número de operadores e operandos distintos;
- N_1 e N_2: número de ocorrências totais de operadores e operandos;
- η: vocabulário do programa;
- V: volume do programa;

- I: conteúdo de informação;
- $\hat{L}$ e $\hat{D}$: nível e dificuldade do programa estimados;
- NL: nível de linguagem;
- E: esforço de programação;
- N_{prog} e E_{prog}: N e E calculados para todo o programa, ignorando-se os limites das rotinas.

Outras Métricas:

- NC: número ciclomático;
- NR: número de rotinas;
- LDC ou LDCt: linhas de código total;
- LDCe: linhas de código executável.

1 - INTRODUÇÃO

1.1 - Introdução

Na instalação de novos sistemas computadorizados, a relação de investimentos em software e hardware cresceu de 11:9 em 1980 para cerca de 16:4 em 1990 [FORT89]. A razão para este índice tão alto tem sido o desenvolvimento muito rápido da tecnologia de hardware. Nenhuma outra tecnologia conseguiu em apenas 30 anos apresentar tamanha evolução na relação custo-desempenho — da ordem de 10^6 [BROO87; BAER84].

O hardware mais potente e barato, por sua vez, permite uma maior popularização e disseminação de computadores na sociedade e o uso desses em aplicações cada vez mais abrangentes e complexas. Isto faz a demanda por software crescer cerca de 10 vezes por década [MIZU83], enquanto que, nesse mesmo período, estima-se que a produtividade na produção de software cresce a uma taxa de apenas 50% [MOAD90]. Em conseqüência, a indústria de software mostra-se incapaz de suprir tal demanda, apesar de investimentos vultosos anuais. (A estimativa para 1990 é de aproximadamente 250 bilhões de dólares em todo o mundo [BOEH87].)

Tendo como objetivo "produzir o melhor software possível ao menor custo possível" [CARD87, p. 845], a disciplina de **engenharia de software** [RAMA84; MILL80], surgida no início da década de 70, tornou-se alvo de muito interesse, tanto acadêmico como comercial. A sua evolução, contudo, é lenta e árdua, sem as grandes revoluções que tanto caracterizam a história do

hardware. E ainda hoje, os maiores problemas desta disciplina são os mesmos de 20 anos atrás: baixa produtividade, atrasos de cronograma, gastos acima dos previstos, falta de controle sobre o desenvolvimento e baixa qualidade. Tanto é que uma pesquisa do governo americano, citada em [BUCK84], descobriu que em nove projetos de software encomendados pelo governo, totalizando valores de 6,8 milhões de dólares: 45% dos software foram entregues mas nunca usados, 29% foram pagos mas não entregues, 19% foram retrabalhados para poderem ser usados, 3% foram usados após pequenas mudanças e apenas 2% foram usados da maneira como foram entregues.

Diante desta situação problemática no desenvolvimento de software, diversas técnicas e metodologias têm sido elaboradas e aplicadas com o intuito de melhorar a qualidade da produção do software e, também, do software em si, o que é essencial neste competitivo mercado. Exemplos mais relevantes dessas técnicas são os modelos de especificação, de representação de dados e de projetos, a programação estruturada, as técnicas orientadas a objetos, a especificação formal, as ferramentas CASE (engenharia de software auxiliada por computador), etc [BROO87; RAMA84].

No entanto, uma área fundamental da engenharia de software, relativamente pouco abordada, é a que compreende os **aspectos quantitativos** da produção. Apesar de sua importância no controle e gerência de desenvolvimento, os fatores que relacionam custo e tempo à quantidade de software desenvolvido são pouco levantados e analisados [DEMA89]. Por esta razão, as me-

todologias de previsão dos fatores quantitativos do desenvolvimento de software ainda estão pouco evoluídas. Assim, é freqüente a previsão de prazos e recursos baseada exclusivamente na experiência de gerentes de produção e em dados empíricos.

O desenvolvimento de software, como qualquer outra atividade, não pode ser gerenciado se não pode ser medido. Desta forma, avanços na previsão e controle acontecerão quando se puder medir e estimar propriedades do software com precisão. A identificação, estudo e aperfeiçoamento destas propriedades mensuráveis, ou métricas, como serão chamadas daqui em diante, são essenciais para a execução de projetos de forma mensurável, cujo desenvolvimento possa ser previsto, acompanhado e monitorado [DEMA89; GRAD87]. As métricas, em suma, "trabalham como mecanismos de controle que, além de avaliarem um processo, também servem de guia para o aperfeiçoamento deste processo" [MOHA81, p. 117]. Este trabalho trata de métricas aplicadas à engenharia de software.

1.2 - Adoção de Métricas em Engenharia de Software

A medição de características do software pode ser útil em praticamente todo seu processo de desenvolvimento. Normalmente, costuma-se dividir o processo de desenvolvimento de software em diversas fases bem definidas embora, na prática, estas fases possam se sobrepor. Um modelo comumente aceito é mostrado na figura 1.1 e divide o ciclo de desenvolvimento nas seguintes fases:

- Requisitos e especificação (especificação completa das

funções requeridas e características de desempenho);

- Projeto (descrição de como o software deve ser implementado para cumprir sua especificação; consiste normalmente na decomposição do software em módulos e submódulos);
- Codificação (codificação e testes de cada módulo);
- Testes (procura e correção de erros após a integração dos módulos);
- Manutenção (melhoria e correção de erros do software após a liberação para uso).

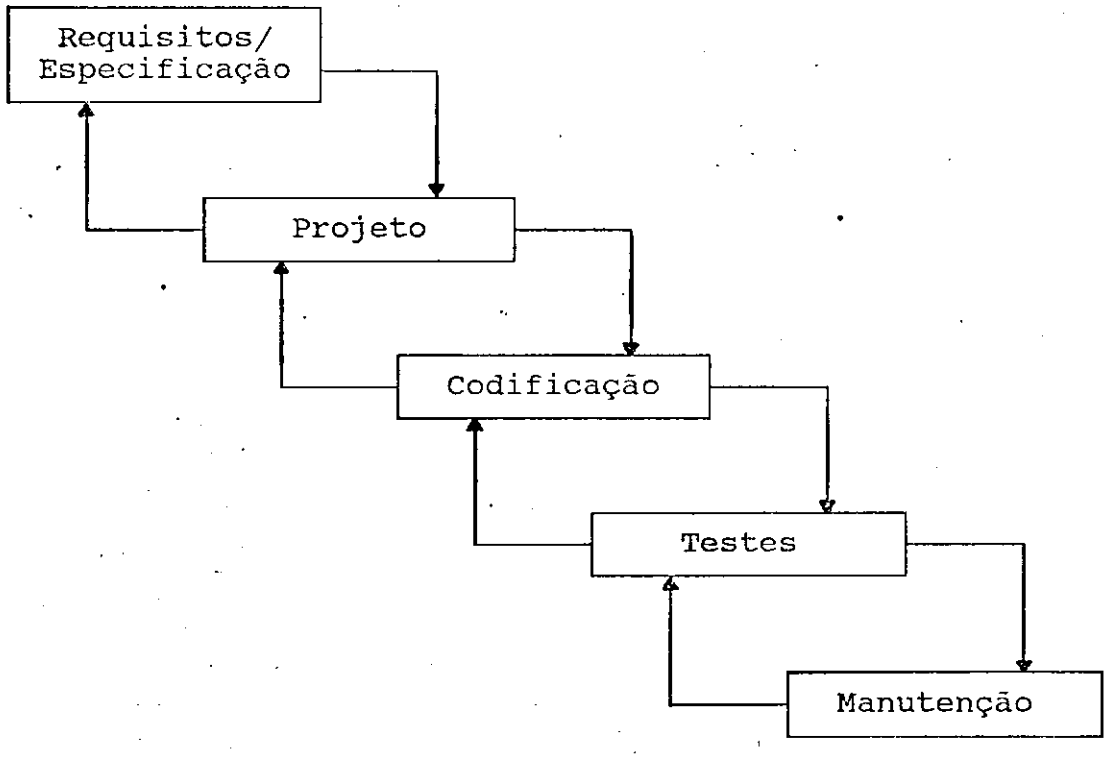


Figura 1.1 - Ciclo de desenvolvimento de software

Embora as métricas possam ser aplicadas às diversas fases

do ciclo de desenvolvimento de software, a grande maioria das métricas encontradas na literatura se concentra na fase de codificação. As métricas para outras fases mostraram-se até agora bem mais imprecisas, principalmente por dependerem muito da metodologia de desenvolvimento adotada.

As métricas da fase de codificação normalmente se baseiam na contagem de uma ou mais propriedades textuais do programa fonte para avaliar os seguintes aspectos [CONT86]:

- o tamanho do software (como o número de linhas de código ou alguma outra contagem de tokens);
- a complexidade da estrutura lógica (como fluxo de controle, nível de aninhamento ou recursão);
- a complexidade das estruturas de dados (como o número de variáveis usadas).

Ainda que numerosas métricas tenham sido propostas por pesquisadores desde meados da década de 70, para poucas foram feitos experimentos demonstrando os benefícios de sua utilização. Desta forma, mais do que identificar novas métricas é essencial, hoje, executar testes de validação das existentes, pois só assim elas poderão ser efetivamente adotadas na caracterização e avaliação do software e na previsão de seus atributos [CURT83]. Este estudo busca acrescentar subsídios para a validação de métricas aplicadas à fase de codificação de software.

1.3 - Ciência de Software e Número Ciclomático

Entre as diversas métricas para a fase de codificação apresentadas na literatura, se destacam as métricas da Ciência de Software de Maurice Halstead [HALS77] e o número ciclomático de Thomas McCabe [MCCA76], pela atenção e críticas recebidas. É importante notar que estas métricas não se restringem à fase de codificação, apenas. Para fins comparativos, foram também experimentadas as métricas de **linhas de código** e **número de rotinas**, pois, apesar de terem grandes limitações (ver seção 2.3.1), são as mais populares e fáceis de obter.

A Ciência de Software é uma teoria que, apesar de simples na derivação de suas equações — todas baseadas na contagem de operadores e operandos do código fonte — tem uma gama de aplicações bastante ampla e abrangente e, a princípio, não faz restrições quanto a diferentes linguagens de programação e ambientes de produção. A publicação de artigos científicos descrevendo a utilização destas métricas nas linguagens Fortran [BASI83b], APL [KONS85], PL/I [ELSH76a; ELSH76b], PL/S [SMIT80] e Cobol [SHEN80] mostra evidências que a teoria de Halstead é relativamente precisa e útil, principalmente na determinação da complexidade de código e nas estimativas de comprimento de programa, de tempo de programação e do número de erros no programa.

O número ciclomático foi criado com o intuito de quantificar a complexidade do fluxo de controle e baseia-se no número de caminhos de execução possíveis em um programa. Este número é dado pela quantidade de decisões no código mais um. Por refle-

tir de quantas maneiras diferentes um programa pode ser executado, esta métrica é uma base mais precisa para previsões de dificuldade de testes e manutenção causada pela complexidade do código.

A precisão (ou grau de aproximação em relação aos dados reais) destas métricas na medição de esforço de desenvolvimento e complexidade do código ainda é uma questão em aberto. Tendo em vista que ambientes e linguagens de programação variam, estudos individuais dentro de organizações produzem resultados muito variáveis, dificultando análises de validação das métricas. Assim, estudos em diversos ambientes serão necessários antes de se responder a essa questão com algum grau de confiança. Tais estudos se tornam mais árduos devido à escassez de dados seguros sobre o processo de desenvolvimento de software. Aparentemente, poucos coletam formalmente tais estatísticas, a não ser quando conscientes da importância de medir o processo de desenvolvimento.

Em função da revisão bibliográfica realizada, pode-se afirmar que muito pouco foi publicado sobre a validação das métricas de Halstead e McCabe para programas em Pascal e quase nada quanto à linguagem C — linguagens preferidas hoje em ambientes acadêmicos e para desenvolvimento de software básico.

Esta dissertação estuda a adequação das métricas da Ciência de Software de Halstead e do número ciclomático de McCabe para modelar o comportamento de programadores em Pascal e em C.

1.4 - Objetivos e Contribuição do Trabalho

O principal objetivo deste trabalho é analisar a validade das métricas da Ciência de Software e do número ciclomático na implementação de programas nas linguagens C e Pascal, tanto do ponto de vista de consistência interna como no auxílio de estimativa de tempo de programação. Com isto, pretende-se examinar a aplicação destas métricas a ambientes pouco estudados na literatura especializada.

O estudo de validação é feito em duas etapas. A etapa de **validação interna** verifica a consistência das diversas equações e suposições por trás das métricas de interesse. A etapa de **validação externa** confronta o tempo de desenvolvimento estimado pelos modelos de métricas com o tempo real observado. Em ambas as etapas, são utilizadas técnicas estatísticas que permitem avaliar a precisão das equações e métricas e estabelecer intervalos de confiança para as métricas, quando utilizadas como estimadoras.

O objeto da análise consistiu em 1.670 rotinas em C e Pascal, totalizando mais de 50.000 linhas de código fonte, provenientes de aplicativos comerciais, processador de texto, software básico e programas de estudantes. Os programas em C foram desenvolvidos no ambiente UNIX (o qual oferece um conjunto de ferramentas poderosas para maior produtividade no desenvolvimento de software — e como tal, pode influenciar os resultados). Os programas em Pascal foram produzidos no ambiente DOS. Os resultados são, portanto, significativos do ponto de vista acadêmico e de interesse prático, visto que os programas

analisados estão disponíveis comercialmente (exceto, obviamente, aqueles provenientes de tarefas escolares).

Com relação à abrangência dos resultados a serem apresentados, vale uma ressalva. Devido à escassez de dados coletados durante o desenvolvimento dos programas analisados, principalmente os escritos em C, a análise da utilidade das métricas como estimadoras de parâmetros como distribuição de erros, eficiência de remoção de erros antes da liberação de produto e distribuição de falhas residuais (a serem corrigidas na fase de manutenção) não pôde ser realizada. É intuitivo, contudo, que esses parâmetros se correlacionem fortemente com o nível de qualidade e produtividade de desenvolvimento. Por isto, devem ser alvo de atenção em trabalhos futuros.

É nossa opinião que o presente trabalho contribui para a engenharia de software devido aos seguintes pontos. Primeiro, esta dissertação expande os horizontes de aplicação de um conjunto de métricas simples, mas robustas (como será visto), a ambientes de programação populares (C no Unix e Pascal no DOS). Segundo, o estudo de validação oferece uma base sólida, a partir da qual, modelos de avaliação de produção e produtividade no desenvolvimento de software podem ser construídos devidamente ajustados para refletir características operacionais de produtos de software de qualquer porte. O ajuste destes modelos exigirá certamente a coleta de dados durante alguns anos. Alguns dos possíveis procedimentos na definição dos modelos são apresentados aqui. Terceiro, o mero fato de serem oferecidos valores para alguns parâmetros da fase de codificação e ter-se

demonstrada sua validade para software feito no Brasil tem despertado o interesse de profissionais em adotarem modelos gerenciais baseados nas métricas estudadas. Esse interesse levará certamente a uma eventual melhoria da qualidade de nossos produtos de software e a um nível gerencial mais balizado. Quarto, o estudo realizado, se bem que incipiente, motivará esforços para uma maior abrangência de pesquisa na área e ao desenvolvimento de mais ferramentas para coleta automática de dados — o que muito contribuirá para estudos futuros de validação.

1.5 - Organização da Dissertação

O restante desta dissertação está organizado como segue. O capítulo dois conceitua métrica de software e introduz classificações nas quais se enquadram as métricas da Ciência de Software e o número ciclomático.

No terceiro capítulo são descritas as abordagens e equações das métricas da Ciência de Software e do número ciclomático, enquanto que no capítulo quatro são apresentados os principais trabalhos publicados a respeito da aplicação dessas métricas em diversas linguagens.

O capítulo cinco descreve, de forma sucinta, como foram realizados os experimentos deste estudo e quais foram os critérios adotados para a validação estatística das métricas. É apresentada uma ferramenta, desenvolvida especialmente para este trabalho, cuja finalidade é coletar os dados e calcular as métricas para códigos em Pascal e C. As metodologias de cálculo

seguidas pela ferramenta na obtenção das métricas são também descritas detalhadamente.

Os capítulos seis e sete apresentam os resultados das análises efetuadas com códigos em Pascal e em C, respectivamente, comparando-os com conclusões a que se chegaram em outros estudos. Nesses capítulos também são feitos testes de validação interna e externa das métricas para amostras constituídas de rotinas e programas, a fim de avaliar a sua adequação. A leitura desses capítulos é dispensável ao leitor interessado apenas nos principais resultados, que são resumidos no capítulo posterior.

O capítulo oito apresenta um sumário do estudo, com conclusões a respeito das métricas abordadas. Neste capítulo, também são feitas recomendações para trabalhos futuros, especialmente quanto a metodologias de coletas de dados para cálculo das métricas.

O apêndice A, por fim, introduz conceitos básicos e algumas técnicas pertinentes à Estatística utilizados neste trabalho.

2 - MÉTRICAS DE SOFTWARE

Este capítulo define o que é uma métrica de software e apresenta classificações das métricas quanto à área de aplicação, fase de desenvolvimento e abordagem básica, com o intuito de enquadrar as métricas aqui estudadas. Para as métricas citadas, mas não discutidas, são indicadas referências bibliográficas.

2.1 - Definição

De acordo com [GRAD87, p. 4], "uma métrica de software define uma maneira padronizada de medir algum atributo do processo de desenvolvimento de software". Por exemplo, tamanho, custo, defeitos, comunicações, dificuldade e ambiente são todos considerados atributos. Embora o termo **métrica** seja largamente usado e por isso adotado neste trabalho, alguns pesquisadores o rejeitam preferindo usar o termo **medida** (measure, em inglês). De acordo com [RAMA88], uma métrica é um conceito matemático bem definido, o que não é, segundo os autores, o caso das medidas de software.

2.2 - Utilização de Métricas na Avaliação de Produtividade

Com a crescente demanda de software, torna-se crucial o desenvolvimento de novas tecnologias que proporcionem uma maior produtividade e qualidade. As métricas são essenciais na validação e aprovação destas tecnologias por auxiliarem na avalia-

ção mais objetiva dos benefícios trazidos pela sua adoção. Antes de estudar o que as métricas de produtividade e qualidade medem, é conveniente definir produtividade.

A produtividade normalmente se refere à relação entre as medidas de saídas e entradas de um processo. No desenvolvimento de software, um exemplo bem comum é o número de linhas de código por pessoa por dia. A entrada é a quantidade de pessoas e dias, e a saída o número de linhas de código. A figura 2.1, adaptada de [GRAD87], mostra os componentes principais da entrada (o CUSTO) e da saída (o VALOR) do processo de desenvolvimento.

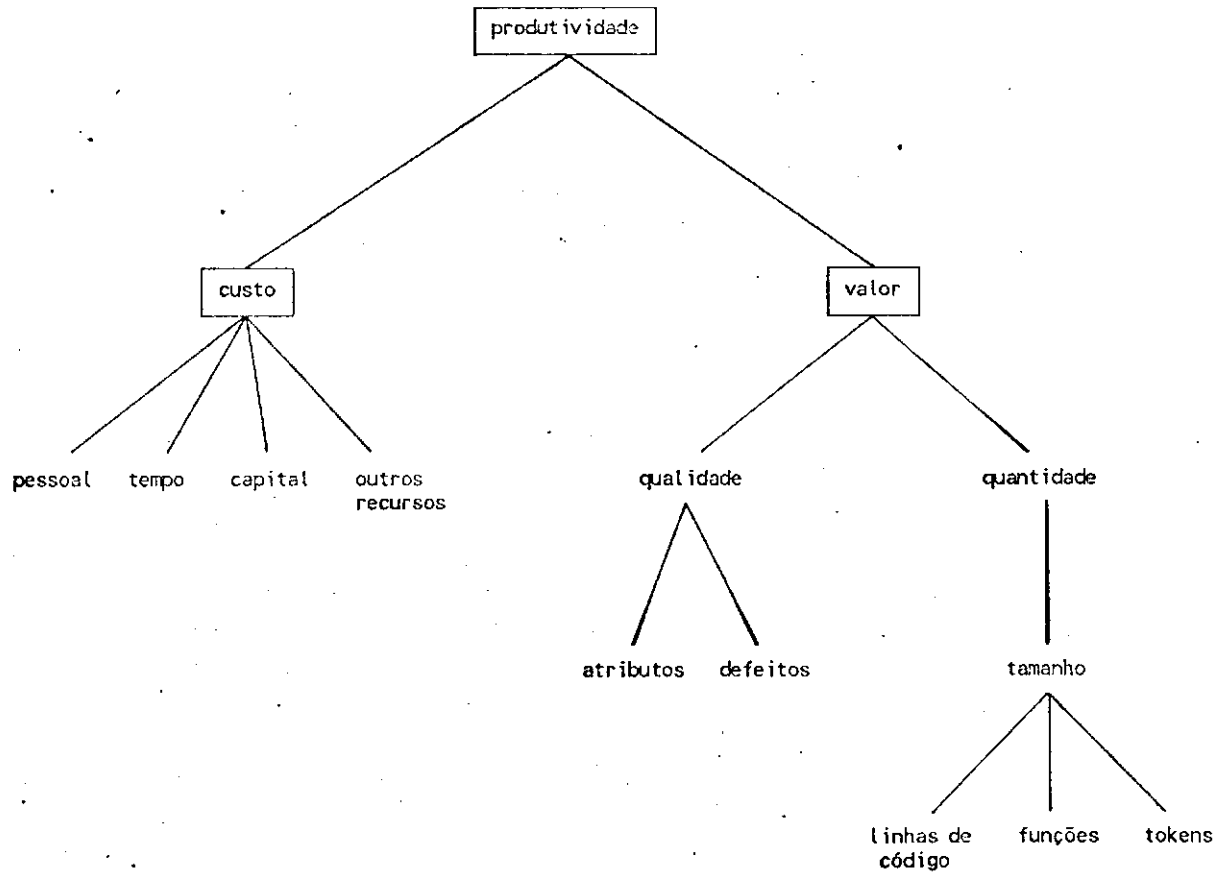


Figura 2.1 - Fatores que afetam a produtividade

Por influenciar diretamente a produtividade, os fatores de valor — a qualidade e quantidade de software — são tópicos da maior importância na engenharia de software. Mas, ao contrário dos fatores de custo, que normalmente não apresentam dificuldades para serem quantificados, os fatores de valor são de caráter mais subjetivo e de difícil mensuração — afinal, software é um produto essencialmente resultante de um esforço lógico. As práticas comuns para medição destes ítens são o assunto das próximas seções.

2.2.1 - Qualidade

A abordagem mais comum no tratamento de qualidade de software consiste em dividir a qualidade em componentes ou **atributos** e estudar maneiras de medir a presença (ou ausência) destes atributos no software. Boehm, por exemplo, decompõe a qualidade nos seguintes atributos: compreensibilidade, confiabilidade, estruturação, eficiência e facilidade de teste, manutenção e uso [BOEH78]. Entretanto, devido à dificuldade, ou talvez até impossibilidade, de se quantificar alguns destes atributos, não há atualmente métricas de qualidade bem estabelecidas [CONT86].

Mas, de acordo com alguns pesquisadores [LEW88; IVAN87], os atributos de qualidade dependem fortemente de um subatributo bem mais conhecido, relativamente independente e mais facilmente quantificável que é a **complexidade**. Este tipo de complexidade, também chamado de complexidade psicológica, está relacionado com as características do software que o fazem difícil de

entender, testar e manter [WAGU87].

Por esta razão, as métricas de complexidade são úteis na avaliação da dificuldade de desenvolvimento, de testes e principalmente de manutenção. Considerando que a manutenção representa cerca de 70 a 75% [PARI90; BOEH78] do custo total no ciclo de vida de grandes sistemas de software, deduz-se que todos os esforços para diminuir o trabalho de manutenção são bem-vindos, e as métricas de complexidade têm papel essencial neste sentido. Uma boa métrica de complexidade pode, inclusive, ser usada para identificar módulos de software de difícil manutenção e maior probabilidade de apresentar erros.

Dentre as métricas de complexidade encontradas na bibliografia, destacam-se principalmente o número ciclomático e uma das métricas de Halstead (o esforço, definida posteriormente), ambas estudadas neste trabalho.

2.2.2 - Quantidade

Para a medição de quantidade, comumente são quantificados os produtos resultantes do desenvolvimento nas diversas fases. O quadro 2.1 mostra alguns exemplos.

FASES	EXEMPLOS DE MÉTRICAS
Requisitos/ Especificação	Número de funções, interfaces, dados e estados de transição que constam na documentação da especificação (métrica Bang) [DEMA89].
Projeto	Número de módulos, conexões, dados, tokens de controle, etc., que constam no projeto estruturado [DEMA89]. Interação entre os módulos e complexidade interna [LEW88]; Grau de estabilidade do projeto [YAU85]; Acoplamento e coesão dos módulos [ARTH85].
Codificação	Número de linhas de código, instruções e rotinas; métricas de Halstead; número ciclomático
Testes	Número de defeitos encontrados; porcentagem de caminhos possíveis testados [GRAD90].
Manutenção	Número (e tipo) de modificações corretivas, preventivas e adaptativas; confiabilidade [CONT86].

Quadro 2.1 - Exemplos de Métricas em Diversas Fases

As métricas tratadas neste trabalho se restringem apenas à fase de codificação.

2.3 - Principais Métricas para a Fase de Codificação

As métricas para a fase de codificação normalmente se propõem a medir a quantidade ou complexidade do código. É comum a classificação dessas métricas em três categorias [CONT86]: **métricas de volume** (ou de tamanho), **métricas de estrutura lógica** (ou de organização de controle) e **métricas de estrutura de dados**. Algumas métricas podem ser incluídas em mais de uma cate-

goria.

Costuma-se, também, chamar as métricas de volume de métricas extensivas e as de estrutura lógica ou de dados de métricas intensivas.

2.3.1 - Métricas de Volume

As métricas de volume são as mais comuns e medem o tamanho do software, como por exemplo, os números de linhas de código, de instruções, de declarações e de rotinas. Até mesmo o número ciclomático pode ser enquadrado aqui, já que é o número de decisões de um programa mais um. Também nesta categoria podem ser incluídas todas as métricas de Halstead, por serem derivadas da contagem de operadores e operandos.

A métrica **número de linhas de código** é largamente usada na estimativa de custo e na avaliação de produtividade e complexidade. Uma utilização típica é a medição de produtividade; por exemplo, em [GRAD87] cita-se que a produtividade média americana é de 100 a 500 linhas de código por mês por pessoa. Mas apesar de ser a métrica mais levantada e estudada, ela apresenta grandes limitações (que as outras métricas em geral apresentam, mas em menor grau):

- variação quanto ao método de contagem (inclusão ou não de linhas de declaração, linhas de comentário, linhas nulas, linhas de continuação, etc.);
- variação quanto à linguagem (um programa em código de máquina, por exemplo, apresenta muito mais linhas que um

programa equivalente em Pascal);

- variação quanto ao estilo de programação (um mesmo programa pode ser escrito de forma mais ou menos condensada);
- insensibilidade à qualidade e complexidade.

Uma métrica também de uso muito comum e de obtenção relativamente simples é o **número de rotinas**. Uma rotina pode ser definida como sendo uma "coleção de instruções, juntamente com as declarações dos parâmetros formais e das variáveis locais manipuladas por estas instruções, que pode ser invocada como uma unidade operacional" [BASI79, p. 36]. (Embora existam definições baseadas nos novos paradigmas de programação, esta definição é bem aplicada às lingagens aqui tratadas.) O número de rotinas é uma métrica útil na previsão do tamanho de software, pois pode ser conhecida com boa precisão no final da fase de projeto lógico. No entanto, apresenta uma limitação substancial: a variação de tamanho e complexidade de uma rotina para outra é muito grande, especialmente entre rotinas de diferentes tipos de software e metodologias de programação.

Outros tipos de métricas de volume [REDI86; BERR85; HARR86] tentam avaliar a legibilidade do texto do programa através da análise do estilo. Alguns dos elementos considerados são: endentação de instruções, quantidade de comentários, variáveis e rotinas, comprimento médio dos nomes das variáveis, tamanho médio das rotinas, etc. Esta abordagem não leva em conta a estrutura lógica do código e sim a sua representação física.

Outra métrica de volume muito usada com bons resultadas na avaliação de produtividade é a métrica de **pontos de função**, de-

senvolvida por Allan Albrecht [ALBR79]. A abordagem básica se concentra no número de entradas, saídas, consultas, arquivos-mestre e interfaces. Estes números são ponderados de acordo com o grau de complexidade para gerar o número de pontos de função. A produtividade é dada pela relação entre este número e o total de horas de trabalho. A vantagem desta abordagem é a total independência quanto à linguagem usada. Mas um ponto fraco é a inabilidade de refletir qualquer atributo de qualidade do software.

Por último, em [POOR88] há uma maneira de se avaliar a qualidade de um programa através da verificação do cumprimento de 22 regras de qualidade que têm pesos variando de 1 a 4. Como a maioria das regras são subjetivas, a automatização das métricas é muito difícil. Para contornar essa dificuldade, foram selecionadas três métricas automatizáveis principais (uma das métricas de volume de Halstead; a relação entre o número de operandos alterados e o número total de operandos e o número de instruções de desvio).

2.3.2 - Métricas de Estrutura Lógica

As métricas de estrutura lógica medem a compreensibilidade das estruturas de controle do código. Desta forma, o **número ciclomático**, quando visto como o número de caminhos de controle, é também uma métrica desta categoria.

As métricas **Número de Interseção Máxima** [CHEN78] e **Knots** [WOOD79] tentam medir a complexidade estrutural através de gráficos de fluxo de controle correspondentes ao programa. O Núme-

ro de Interseção Máxima é o número de interseções geradas quando uma linha contínua entra, por uma única vez, em todas as regiões do gráfico de fluxo de controle correspondente ao programa do qual se deseja medir a complexidade. A métrica de Knots é calculada desenhando-se ao lado do código linhas indicando as mudanças no fluxo de controle (provocadas por goto, while, etc) e contando-se o número de interseções destas linhas. Esta métrica é inaplicável a programas estruturados (sem instruções de desvio goto), como os considerados neste estudo.

Acreditando que o nível de aninhamento de uma instrução dificulta a compreensão de sua função, alguns pesquisadores desenvolveram métricas (como por exemplo, Bandwidth e Band) que medem a complexidade como sendo proporcional ao nível médio de aninhamento das instruções [DUNS78; LIND89; DAVC82; JENS85].

Uma abordagem inteiramente diferente das citadas baseia-se no fato de que programas são entendidos através da assimilação de grupos de instruções que têm uma função comum [DAVI88; WEYU88; OVIE80]. Estes grupos são chamados comumente de blocos. Apesar de não haver um consenso sobre o que define precisamente um bloco, uma opinião bastante aceita define-o como sendo uma seqüência de instruções consecutivas com a propriedade de que não há nenhum desvio de controle para qualquer instrução dentro do bloco que não seja a primeira. A complexidade do código é então dada pela dependência de controle de um bloco em relação aos outros, que ocorre quando existe um desvio de controle de um bloco para outro. Uma desvantagem dessas métricas é a dificuldade de obtenção.

De acordo com [DAVI88], a complexidade da estrutura lógica parece estar mais relacionada a atividades pertinentes à construção de programas, comuns na fase de projeto e codificação.

2.3.3 - Métricas de Estrutura de Dados

As métricas de organização de dados são medidas do uso e visibilidade dos dados e também das interações entre os dados dentro de um programa. Dentre as métricas de Halstead, duas básicas podem ser incluídas aqui: o número de operandos distintos (variáveis e constantes) e o número total de ocorrências de operandos.

A métrica de Data Binding [BASI75; STEV74] é um exemplo desta categoria por medir a interação entre as rotinas. Outro exemplo é a métrica Span [ELSH76a] que tenta medir a proximidade entre referências e definições de cada item de dado.

Métricas para medir características do fluxo de dados de um programa e que se baseiam no conceito de blocos (visto na seção anterior) foram desenvolvidas em [DAVI88; OVIE80]. A complexidade é dada pela dependência de dados de um bloco em relação a outros. Uma dependência entre dois blocos existe se uma variável pode ser modificada em um bloco e ser referenciada (usada) em outro.

Davis e LeBlanc [DAVI88] consideram que a complexidade da estrutura de dados aparenta ser mais importante nas atividades relativas à compreensão de um programa existente, comuns nas fases de testes e manutenção.

Neste capítulo, as principais métricas encontradas na li-

teratura foram apresentadas e classificadas quanto à fase do desenvolvimento de software e quanto à abordagem básica utilizada. A utilização de métricas na medição de produtividade foi também discutida. No próximo capítulo são descritas as abordagens e equações das métricas da Ciência de Software e do número ciclomático.

3 - MÉTRICAS DA CIÊNCIA DE SOFTWARE E NÚMERO CICLOMÁTICO

3.1 - Métricas da Ciência de Software

3.1.1 - Histórico

A Ciência de Software, inicialmente conhecida como Física de Software, foi desenvolvida a partir de 1972 por Maurice Halstead e está relacionada com as propriedades mensuráveis do texto de programas fonte.

Lembrando que os antigos programas em linguagem de máquina consistiam apenas de uma série de instruções de uma palavra, cada uma contendo um código de operação e o endereço de um operando, Halstead concluiu que um programa consistia apenas de operadores e operandos e nada mais. Ele, então, estendeu essa abordagem para linguagens mais modernas e elaborou uma série de relações entre as propriedades mensuráveis, usando derivações matemáticas e observações experimentais.

A partir da publicação de seu livro "Elements of Software Science" [HALS77], a Ciência de Software passou a ser alvo de numerosos trabalhos experimentais. No geral, os resultados encontrados por esses trabalhos foram relativamente bons, mas aquém dos apresentados por Halstead.

Contudo, a falta de progressos da teoria, com o falecimento de Halstead em janeiro de 1979, e as críticas nas derivações das fórmulas [MORA78; FENI79; SHEN83; COUL83] fizeram a Ciência de Software cair em desuso no início dos anos 80. Recentemente, o número de estudos publicados sobre a Ciência de Software mos-

tra que há um interesse renovado na teoria. O motivo para isto talvez seja a falta de grandes progressos na área de métricas. Afinal, até mesmo alguns dos críticos da teoria afirmam que apesar de incoerências no seu fundamento teórico, algumas métricas de Halstead parecem ainda estar entre as melhores e podem ser muito úteis [SHEN83].

As aplicações da teoria são bastante variadas, sendo as principais:

- servir como medida de quantidade e complexidade de software existente;
- prever o comprimento de programas;
- avaliar o efeito da escolha de uma linguagem na produtividade;
- prever o número de erros no programa;
- estimar o tempo que um programador típico deve levar para implementar um dado algoritmo.

3.1.2 - Apresentação

Todas as relações da teoria de Halstead são derivadas a partir da contagem dos operandos e operadores de um programa. Os operandos e operadores são as partículas-átomos que compõem um programa. Os operandos contêm valores que são modificáveis ou usados como referência para mudanças, ou seja, são as variáveis e constantes. Os operadores são as instruções, delimitadores, símbolos aritméticos e de pontuação, etc., que atuam sobre os operandos. São também operadores as instruções que afetam o fluxo de controle do programa, como DO WHILE, IF THEN, etc. Os operandos e operadores das linhas de declaração são normalmente

ignorados por não serem inerentes ao algoritmo.

3.1.3 - As Métricas Básicas

As quatro métricas básicas das quais se derivam todas as relações da teoria são:

η_1	número de operadores distintos usados no programa
η_2	número de operandos distintos usados no programa
N_1	número total de ocorrências de operadores no programa
N_2	número total de ocorrências de operandos no programa

Para ilustrar a obtenção dos valores das métricas de Halstead será utilizada uma implementação em Pascal do algoritmo de Euclides para encontrar o máximo divisor comum de dois números [HALS77]:

```
Begin
    If A = 0 Then
        GCD := B
    Else
        If B = 0 Then
            GCD := A
        Else
            Begin
                Repeat
                    R := A mod B;
                    A := B;
                    B := R
                Until R = 0;
                GCD := A
            End
End.
```

A obtenção das métricas básicas pode assim ser feita:

Operador	Número de Ocorrências
:=	6
;	3
=	3
Begin End	2
If Then	2
Else	2
Repeat Until	1
mod	1
.	1
$\eta_1 = 9$	$N_1 = 21$

Operando	Número de Ocorrências
B	5
A	5
0	3
GCD	3
R	3
$\eta_2 = 5$	$N_2 = 19$

3.1.4 - Vocabulário (η) e Comprimento (N)

O **vocabulário** (η) de um dado programa é definido como a soma dos números de operadores e operandos distintos usados naquele programa (fórmula 3.1). Esse número é uma medida do número de diferentes elementos com que um programador deve lidar para implementar o programa.

Vocabulário: $\eta = \eta_1 + \eta_2$ (3.1)

No exemplo: $\eta = 9 + 5 = 14$

O **comprimento** (N) de um dado programa é definido como sendo a soma do número total de ocorrências de operadores e operandos. Intuitivamente, o comprimento é uma medida do tamanho do programa e mede o número de vezes que um programador lida com os elementos de programação. O comprimento é dado pela fórmula:

Comprimento: $N = N_1 + N_2$ (3.2)

No exemplo: $N = 21 + 19 = 40$

Halstead sugeriu uma relação na qual o comprimento pode ser estimado a partir do vocabulário. Essa fórmula, conhecida como equação do comprimento, é a seguinte:

Comprimento estimado: $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$ (3.3)

No exemplo: $\hat{N} = 9 \log_2 9 + 5 \log_2 5 = 40,1$

Esta fórmula é de grande importância para a fase de projeto na previsão do comprimento final de programas. Pois se os valores de η_1 e η_2 são determinados, o comprimento final do programa pode ser estimado com boa precisão. Devido a isso, essa equação é a base da teoria de Halstead.

3.1.5 - Volume - V

O volume é também uma medida do tamanho do programa, medido em número de bits. É definido como:

Volume: $V = N \log_2 \eta$ (3.4)

No exemplo: $V = 40 \log_2 14 \approx 152$ bits

A derivação dessa fórmula pode ser compreendida como segue. Para cada uma das N ocorrências de elementos de um programa $\log_2 \eta$ bits são necessários para a identificação do elemento, seja ele operador ou operando, supondo um armazenamento binário. Assim, V mede o número de bits requeridos para descrever um programa.

Quando se traduz um programa numa linguagem de alto nível

para uma de nível mais baixo, seu volume provavelmente cresce, porque são necessários mais operadores e operandos. Para o exemplo em Pascal, foram precisos 152 bits, enquanto que para um programa equivalente em código de máquina foram precisos 389 bits [HALS77].

3.1.6 - Nível e Dificuldade de um Programa

Ao se traduzir a implementação de um algoritmo em uma linguagem de alto nível para uma de nível mais baixo, usam-se operadores mais simples e em maior número. Ou seja, à medida que o **nível do programa (L)** — como denomina Halstead -- diminui, o volume cresce. O nível de programa indica, portanto, o quão sucinta está a implementação de um algoritmo. Por definição:

$$\text{Nível:} \qquad L = \frac{(2+\eta_2^*)\ \log_2\ (2+\eta_2^*)}{N\ \log_2\ \eta} \qquad (3.5)$$

onde η_2^* é o número de parâmetros lógicos de entrada e saída do algoritmo. O valor de L de acordo com a fórmula varia de 0 a 1, já que N e η são maiores ou iguais a $(2+\eta_2^*)$. No exemplo:

$\eta_2^* = 3$ (A, B e GCD)

$L = (2+3)\ \log_2\ (2+3)\ /\ 152 = 0{,}076$

Uma dificuldade da fórmula (3.5) para o cálculo do nível é que ela é freqüentemente inaplicável por causa da dificuldade de se determinar exatamente quais são os parâmetros lógicos de entrada e saída de um programa (por exemplo, um compilador). E como o próprio Halstead observou, o conceito de η_2^* pode ser

estendido para incluir constantes e, talvez, outras variáveis implícitas dentro do algoritmo [HALS77]. Para contornar este problema, Halstead sugeriu uma fórmula alternativa, comumente utilizada, para estimar o nível:

Nível Estimado: $\hat{L} = (2/\eta_1)\ (\eta_2/N_2)$ (3.6)

No exemplo: $\hat{L} = (2/9)\ /\ (5/19) = 0,058$ (erro de -24%)

Uma outra métrica de Halstead é chamada de **dificuldade** e é igual ao inverso do nível de programa:

Dificuldade Estimada: $\hat{D} = (\eta_1/2)\ (N_2/\eta_2)$ (3.7)

Um argumento intuitivo para essa fórmula é que a dificuldade do programa aumenta se operadores adicionais são introduzidos ($\eta_1/2$ cresce). Já a relação $N_2/\eta 2$ representa o número médio de vezes que os operandos são usados. Quanto mais freqüentemente uma variável foi modificada em um programa (N_2/η_2 aumenta) mais difícil será conhecer o seu valor atual [CHRI81].

3.1.7 - Conteúdo de Informação - I

Ao verificar que o produto $\hat{L}$xV de um algoritmo implementado em várias linguagens permanecia praticamente constante, Halstead sugeriu uma nova métrica, e denominou esta suposta constante de **Conteúdo de Informação** (fórmula 3.8).

Conteúdo de Informação: $I = \hat{L} \times V$ (3.8)

No exemplo: $I = 0,058 \times 152 \approx 8,8$

Várias versões do algoritmo de Euclides foram publicados

em diversas linguagens, e nelas o valor de I variou de 10 a 12,9 (média de 11,4 e desvio padrão de 1,0) [HALS77].

3.1.8 - Nível de Linguagem

A existência de uma técnica que avaliasse o efeito da adoção de uma linguagem qualquer na produtividade seria de imenso valor, dada a proliferação de linguagens — algumas tidas como mais produtivas do que outras. Uma métrica desta natureza foi proposta por Halstead e denominada de **nível de linguagem - NL.** Esta métrica é baseada em uma hipótese que sugere que, para diferentes rotinas numa mesma linguagem, à medida que o volume V cresce, o nível do programa L diminui de tal forma que o produto $L^2 \times V$ permanece aproximadamente constante. Este produto, de acordo com Halstead, caracteriza uma linguagem de programação.

Nível de linguagem: $NL = L^2 \times V$ (3.9)

No exemplo: $NL = 0{,}076^2 \times 152 \approx 0{,}9$

A métrica de nível de linguagem, contudo, ganha mais significado quando aplicada a uma grande população de rotinas, o que será feito posteriormente neste trabalho.

3.1.9 - Esforço

O esforço requerido para implementar um programa aumenta com o tamanho do programa. Além disso, mais esforço é necessário para implementar um programa em um nível mais baixo, ou seja, de maior dificuldade, quando comparado com outro programa

equivalente de nível mais alto. Assim, Halstead elaborou uma métrica chamada de esforço, para avaliar o esforço de desenvolvimento, definida como:

Esforço: $E = V / L = V \times D$ (3.10)

No exemplo: $E = 152 / 0,076 = 2.000$

Este número representa o número de discriminações (decisões) mentais elementares - d. m. e., que um programador fluente e concentrado faria para implementar o algoritmo.

3.1.10 - Tempo de Programação

Pela hipótese de Halstead, o tempo de programação deve ser diretamente proporcional ao esforço E (fórmula 3.10), citado na seção anterior.

$$\hat{T} = E / S \quad (3.11)$$

onde S é uma constante que representa a velocidade de raciocínio de um programador, isto é, o número de discriminações mentais que ele é capaz de fazer por segundo.

Para determinar o valor da constante S, Halstead obteve uma amostra de 12 programas, em linguagem de máquina, e mediu o tempo requerido para escrever cada um deles. Ele calculou o esforço para esses programas e concluiu que S = 18 é um valor razoável.

De acordo com o psicólogo Stroud [STRO66], a mente humana é capaz de realizar 5 a 20 discriminações mentais por segundo. Como o valor de S de Halstead é próximo do limite superior des-

sa faixa, ele denominou esta constante de **número de Stroud.**

No exemplo em Pascal, o tempo esperado para implementação do algoritmo é:

$$\hat{T} = 2000 / 18 = 111 \text{ s}$$

ou um pouco menos de dois minutos.

Em alguns experimentos, $\hat{T}$ foi usado para estimar o tempo necessário para a compreensão de um programa. Dados experimentais dão algum suporte a ambas interpretações [FITZ78a].

3.1.11 - Impurezas

Para que um programa obedeça às relações da Ciência de Software é necessário, de acordo com Halstead, que os programas sejam bem escritos, possivelmente para publicação. Ele identificou seis tipos de impurezas cuja presença nos programas prejudicariam a precisão das relações de sua teoria:

I. **Operações Complementares:** uma operação cancela o efeito de uma anterior;

II. **Operandos Ambíguos:** um mesmo operando serve para identificar diferentes objetos em diferentes locais do programa;

III. **Operandos Sinônimos:** dois operandos têm o mesmo significado no programa;

IV. **Subexpressões Comuns:** uma expressão é repetida mais de uma vez no programa (deveria ser atribuída a uma variável ou agrupada em um procedimento);

V. **Atribuições Desnecessárias:** uma expressão é atribuída a uma variável ou agrupada em um procedimento e utilizada apenas uma vez;

VI. **Expressões Não Fatoradas:** uma expressão que pode ser fatorada não está nessa forma.

O efeito da retirada de impurezas foi analisado por Gordon em [GORD79b], onde ele conclui que a retirada resulta no decréscimo do esforço e aumento da compreensibilidade do programa.

3.2 - Número Ciclomático

Em dezembro de 1976, Thomas J. McCabe [MCCA76] desenvolveu uma métrica baseada no fluxo de controle do programa que se constitui em uma técnica matemática para a modularização de software e para a identificação de módulos potencialmente difíceis de testar e fazer manutenção. A métrica de complexidade de McCabe é baseada na teoria dos grafos [BERG73] e depende apenas da estrutura de decisões do programa, e não do seu tamanho físico. Por esta razão, ela é aplicável não só a programas fonte, mas também a projetos (designs) de software, desde que estejam suficientemente detalhados.

A abordagem de McCabe consiste em: "Dado um programa de computador, associe-o a um grafo orientado com nodos únicos de entrada e saída. Cada nodo no grafo corresponde a um bloco de código no programa onde o fluxo é seqüencial, e os arcos correspondem a ramificações existentes no programa" [HALL84, p. 19]. O grafo resultante é conhecido como grafo de fluxo de controle do programa.

A complexidade do programa, dado o grafo de fluxo de controle correspondente, é o número de caminhos básicos no grafo que, quando combinados, geram qualquer caminho possível entre a entrada e a saída. Este número é chamado de **número ciclomático**

(NC) e é calculado pela fórmula:

$$NC = a - n + 2p \qquad (3.12)$$

onde:

a: número de arcos,

n: número de nodos,

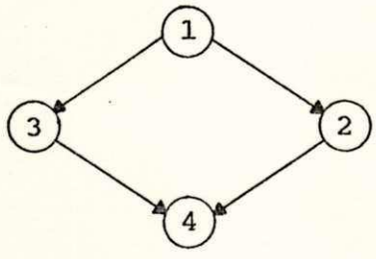
p: número de componentes conectados (rotinas).

Por exemplo, suponha que um segmento de código deva ser escrito para encontrar o menor valor de duas variáveis. Três versões diferentes deste programa são listadas a seguir na linguagem C, seguidas do cálculo do número ciclomático. (Os números entre parênteses ao lado do código se referem aos nodos correspondentes no grafo.)

Versão 1:

```
if ( x <= y )                (1)
     menor = x;              (2)
else
     menor = y;              (3)
...                          (4)
```

Grafo de fluxo de controle correspondente:

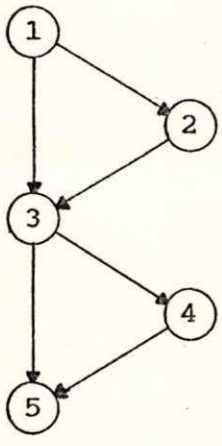


NC = 4 - 4 + 2 = 2

Versão 2:

```
if ( x <= y )                    (1)
     menor = x;                  (2)
if ( y < x )                     (3)
     menor = y;                  (4)
...                              (5)
```

Grafo de fluxo de controle correspondente:



NC = 6 - 5 + 2 = 3

Versão 3:

```
menor = min ( x, y );            (1)
```

Grafo do fluxo de controle correspondente:

NC = 0 - 1 + 2 = 1

Portanto, de acordo com a abordagem de McCabe, a segunda versão é a de maior complexidade, por apresentar maior número ciclomático.

Para um programa consistindo de várias rotinas, o cálculo

do número ciclomático pode ser feito de duas maneiras, gerando os mesmos resultados. Na primeira, utiliza-se a fórmula (3.12) para todo o programa, substituindo p pelo número de rotinas. Na segunda, calcula-se o número ciclomático para cada rotina (com p = 1, como foi feito nos exemplos) e somam-se os resultados.

Uma forma mais simples de determinar o número ciclomático de um programa é contando o número de decisões (causadas por instruções IF, WHILE, REPEAT, etc.) e operadores booleanos existentes e somar um ao resultado. A razão da inclusão dos operadores booleanos na contagem é que cada um gera uma nova decisão no grafo de estrutura.

McCabe acredita que um limite superior razoável para o número ciclomático de um módulo é 10. A intenção deste limite é fazer com que os módulos sejam mais manejáveis, e assim diminuir o esforço nas fases de teste e manutenção. A única situação em que este limite parece inadequado ocorre em rotinas especiais com grandes instruções do tipo CASE.

A utilização do número ciclomático como uma métrica de complexidade foi verificada por vários pesquisadores. No capítulo seguinte, serão apresentados os principais estudos publicados com esta métrica, como também com as métricas da Ciência de Software.

4 - EXPERIMENTOS PUBLICADOS

Neste capítulo, será feito um resumo dos principais trabalhos publicados acerca da aplicação da teoria de Halstead e do número ciclomático em diversas linguagens. Os resultados apresentados neste capítulo servirão para fins de comparação com os obtidos neste trabalho.

4.1 - Métricas da Ciência de Software

4.1.1 - A Equação do Comprimento

A fórmula de estimativa de comprimento, $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$, ou equação do comprimento, como é conhecida, é a base de toda a teoria de Halstead. Foi a partir da observação do fenômeno de que o tamanho do programa depende do número de diferentes operadores e operandos que Halstead derivou todo o seu modelo. Por essa razão e pela sua importância na estimativa de tamanho de programa, a equação do comprimento é o ponto de partida de qualquer estudo de validação da teoria de Halstead.

A bibliografia é farta no que se refere à experimentação da equação de comprimento em diversas linguagens, tais como: Algol [HALS77], APL [ZWEB79b; KONS85], Assembly [SMIT80], Cobol [SHEN80], Fortran [BASI83b; LI87; LIND89], PL/I [ELSH76b] e PL/S [SMIT80]. A equação do comprimento mostrou-se bastante robusta nestes estudos, tendo o coeficiente de correlação entre os comprimentos estimado e real variado de 0,90 a 0,98.

Experimentos conduzidos para a linguagem Pascal obtiveram

resultados semelhantes, com a correlação variando de 0,87 a 0,98 em [JOHN81; JENS85; DAVI87; LIND89]. É comum em estudos nesta linguagem a experimentação da seguinte fórmula empírica, sugerida por [JENS85]:

$$\hat{N}_j = \log_2 \eta_1! + \log_2 \eta_2! \qquad (4.1)$$

A tabela 4.1 mostra os coeficentes de correlação calculados entre N e $\hat{N}$ e entre N e $\hat{N}_j$, acompanhados de uma medida de precisão, o Desvio Relativo Médio, também conhecida como discrepância normalizada.

Nº de rotinas	N x $\hat{N}$ corr.	DRM	N x $\hat{N}_j$ corr.	DRM	Fonte
9	0,985	n/d	n/d	n/d	[JOHN81]
32	0,945	0,630	0,944	0,267	[JENS85]
29	0,858	0,594	0,860	0,259	[JENS85]
48	0,867	0,571	0,816	0,268	[JENS85]
31	0,912	0,612	0,910	0,273	[JENS85]
62	0,948	0,478	0,952	0,216	[JENS85]
29	0,978	0,448	0,977	0,400	[DAVI87]
3442	0,90	0,51	0,91	0,22	[LIND89]

Obs: DRM - Desvio Relativo Médio
n/d - não disponível

Tabela 4.1 - Correlação Entre N e $\hat{N}$, e N e $\hat{N}_j$ para Programas em Pascal

Na tabela 4.1, os resultados da referência [JENS85] correspondem ao estudo de cinco categorias de rotinas, de acordo com suas características operacionais. E em [DAVI87], o objeto de análise consistiu de programas de alunos.

Analisando-se as correlações, chega-se à conclusão de que o estimador $\hat{N}_j$ apresentou desempenho semelhante a $\hat{N}$ na correla-

ção com N. Mas ao se comparar o DRM das duas medidas, a métrica $\hat{N}_j$ mostrou-se mais adequada.

Apesar desses bons coeficientes de correlação encontrados, alguns pesquisadores apontaram que a precisão de $\hat{N}$ varia com o tamanho do programa [CHRI81; JOHN81; BASI83b; DAVI87; LI87; FELI89]. De acordo com essas pesquisas, $\hat{N}$ superestima N (ou seja, $\hat{N}$ é maior que N) para programas pequenos (normalmente $N < 200$) e subestima N para programas grandes ($N > 200$). Davies e Tan [DAVI87] sugerem que isso ocorre porque, em programas estruturados, os únicos novos operadores, além dos definidos na linguagem, são os procedimentos do usuário. Por isso, o componente η_1 cresce muito mais devagar do que o próprio N, e assim, o comprimento estimado é influenciado apenas pelo número de operandos distintos - η_2. Desta forma, enquanto que para programas pequenos $\hat{N}$ é relativamente preciso, para programas maiores $\hat{N}$ tende a ser menor que N devido à pequena contribuição de η_1.

4.1.2 - Dificuldade - D

Revendo a fórmula de estimativa de dificuldade,

$$\hat{D} = \eta_1 \, N_2 \, / \, (2 \, \eta_2),$$

verifica-se que ela só depende de η_1, η_2 e N_2. Assim, acreditando que a métrica η_1 tende a permanecer praticamente constante com relação ao comprimento, [CHRI81] propôs que o uso médio de operandos (N_2/η_2) é o maior influenciador da dificuldade do programa. Esta hipótese é apoiada por Li e Cheung [LI87], baseados na correlação entre $\hat{D}$ e N_2/η_2 (igual a 0,88) maior que en-

tre $\hat{D}$ e η_1 (igual a 0,81). De acordo com estas correlações, a dificuldade é mais influenciada pelo uso médio de operandos.

Basili [BASI83b], entretanto, encontrou resultados contrários que indicam que a dificuldade está mais ligada a η_1 (correlação de 0,865) do que a N_2/η_2 (correlação de 0,729).

4.1.3 - Conteúdo de Informação - I

A constância do conteúdo de informação - I (fórmula 3.8), se comprovada, estabeleceria uma métrica de complexidade extremamente importante. Halstead apresenta em [HALS77] um algoritmo implementado em seis diferentes linguagens (Algol 58, Fortran, Cobol, Basic, Snobol, APL e PL/I) para o qual o conteúdo de informação não varia mais de 10% em torno da média aritmética e o coeficiente de dispersão (relação entre desvio padrão e a média) é de apenas 5,2%.

Entretanto, calculando-se I para as 4 versões de 12 programas apresentados no mesmo livro para outros fins, foram obtidos coeficientes de dispersão variando de 5,4% a 90,7%. Em apenas 2 programas, as 4 versões ficaram dentro de 10% da média.

Em [SHEN81], numa análise de 237 programas em Cobol para 4 problemas, o melhor resultado foi um coeficiente de dispersão de 13%. Não foi incomum existirem casos individuais onde o conteúdo de informação difere em 100% da média. Os dados desse estudo não concordam com a constância do conteúdo de informação I, talvez por se tratarem de programas de estudantes, provavelmente mais variáveis que programas escritos profissionalmente.

4.1.4 - Nível de Linguagem - NL

A métrica NL (fórmula 3.9), proposta para quantificar o nível de uma linguagem, tem despertado o interesse de muitos pesquisadores. Mesmo sendo uma métrica polêmica, o seu valor médio foi calculado para várias linguagens (tabela 4.2).

Linguagem	Nº de Rotinas	Nível de Linguagem	Desvio Padrão	Fonte
APL	28	2,423	1,438	[KONS85]
COBOL*	16	2,07	0,90	[SHEN80]
PL/S	643	2,05	1,14	[SMIT80]
BASIC	32	2,04	1,57	[CHRI81]
APL	991	1,986	1,907	[DEKE86]
COBOL**	23	1,82	0,73	[SHEN80]
PL/I	120	1,53	0,92	[ELSH76a]
COBOL	24	1,40	0,69	[ZWEB79b]
ALGOL 58	14	1,21	0,74	[HALS77]
FORTRAN	14	1,14	0,81	[HALS77]
PILOT	14	0,92	0,43	[HALS77]
ASSEMBLY IBM 370	993	0,91	0,79	[SMIT80]
ASSEMBLY CDC 6500	n/d	0,88	0,42	[HALS77]

Obs: n/d não disponível
* para programadores experientes
** para programadores iniciantes

Tabela 4.2 - Nível de Linguagem (NL) para Várias Linguagens

Apesar de os valores médios da tabela poderem ser considerados razoáveis, ao se analisar os desvios padrões fica evidente que esta métrica, como sendo uma suposta constante para cada linguagem, deixa muito a desejar. A grande variabilidade em uma mesma linguagem sugere, de acordo com [CHRI81], que a métrica nível de linguagem de Halstead não mede o nível da linguagem e sim como a linguagem é usada no programa.

Em [SHEN83] essa métrica aplicada às linguagens BAL (linguagem de máquina) e PL/S apresentou uma dependência inversa com o tamanho (N) do programa.

4.1.5 - Esforço - E

A métrica de esforço, juntamente com a equação do tempo de programação (fórmulas 3.13 e 3.14), serão de utilidade inestimável na gerência de projetos, se forem comprovadas. A tabela 4.3 exibe os coeficientes de correlação, encontrados na bibliografia, entre essa métrica e o tempo de programação real observado (que normalmente inclui o tempo de projeto ou design, codificação e testes preliminares). Para efeito comparativo, as correlações entre o tempo e o número de linhas de código são também incluídas.

Linguagem	Nº de Rotinas	r (ExT)	r (LDCxT)	Fonte
APL	12	0,93	n/d	[HALS77]
PL/I	12	0,94	n/d	[HALS77]
Fortran	12	0,87	n/d	[HALS77]
Fortran	11	0,93	0,89	[HALS77]
Fortran	731	0,36	0,48	[BASI83b]
Fortran	143	0,71	0,71	[CONT86]
Fortran	77	0,70	0,80	[CONT86]
Fortran	32	0,74	0,72	[DAVI88]

Obs: n/d não disponível
E esforço
T tempo de programação
LDC linhas de código

Tabela 4.3 - Correlações entre Tempo de Programação e as métricas Esforço e Linhas de Código

É claro que nem todo o esforço despendido durante a construção do programa se reflete no código final. Quando um programa é construído, revisões podem ser feitas, partes do código podem ser descartadas e diferentes estruturas de dados podem ser consideradas. A métrica de esforço não inclui este esforço desperdiçado. Por esta razão, Gordon acredita que a métrica de esforço estima melhor a dificuldade de compreensão de um programa, servindo assim como um medidor de clareza [GORD79a].

Com relação à confiabilidade de programas, são vários os estudos que correlacionam as métricas de Halstead com o número de erros encontrados, sendo os principais: [FITZ78a; OTTE79; LIPO82; BASI83b; GAFF84; SHEN85; DAVI88; SCHN88b; LIND89]. Nesses estudos foram encontrados coeficientes de correlação linear entre as métricas de Halstead (comprimento N, volume V e esforço E) e o número de erros variando de 0,76 a 0,96.

4.1.6 - Correlações Entre as Principais Métricas

Na tentativa de se descobrir a semelhança entre as principais métricas da Ciência de Software e outras, como o número de linhas de código e o número ciclomático, é comum calcular-se o coeficiente de correlação linear entre elas. A tabela 4.4 resume algumas das correlações publicadas mais relevantes.

Linguagem Nº Rotinas/Fonte	N, V	N, E	N, NC	V, E	V, NC	E, NC	N,LDC	NC,LDC	V,LDC	E,LDC
Assembly:										
992 [SMIT80]							0,995		0,992	
Fortran:										
1794 [BASI83b]			0,874		0,873	0,535	0,873		0,875	0,500
255 [LI87]	0,997	0,944	0,915	0,947	0,918	0,897	0,960	0,891	0,949	0,913
1123 [LIND89]			0,85				0,90	0,84		
Pascal:										
32 [JENS85]	0,999	0,954	0,798	0,951	0,797	0,709				
29 [JENS85]	0,998	0,971	0,908	0,971	0,892	0,923				
48 [JENS85]	0,994	0,910	0,743	0,894	0,715	0,660				
31 [JENS85]	0,997	0,960	0,672	0,950	0,650	0,716				
62 [JENS85]	0,999	0,844	0,868	0,851	0,731	0,731				
29 [DAVI87]	0,997	0,972		0,984						
19 [KOKO88]	0,993	0,893	0,922	0,837	0,903	0,887	0,987	0,911	0,975	0,901
3442 [LIND89]			0,84				0,92	0,85		
981 [HENR90]	0,989	0,749	0,776	0,711	0,781	0,492	0,893	0,629	0,885	0,521
PL/S:										
643 [SMIT80]							0,952		0,929	

Obs: NC - Número Ciclomático LDC - Linhas de Código

Tabela 4.4 - Correlações Entre as Principais Métricas

Analisando as correlações da tabela 4.4, verifica-se que as métricas N, V e LDC, por apresentarem correlação alta entre si, medem aspectos semelhantes do código. Qualquer uma dessas métricas, portanto, pode ser usada para medir o tamanho de código (métricas extensivas).

A métrica NC normalmente não se correlaciona bem com nenhuma outra métrica, indicando que ela realmente pode ser uma métrica **intensiva**, ou seja, que não depende da quantidade de código, como foi proposta.

4.1.7 - Críticas à Ciência de Software

Desde a sua publicação em 1977, a teoria em que se baseia a Ciência de Software é alvo de muita controvérsia. Os principais pontos fracos da Ciência de Software levantados em [MORA78; FENI79; COUL83 e SHEN83] foram:

- As metodologias de contagem de operandos e operadores dão margem a diferentes interpretações, principalmente em linguagens de semântica complexa, como APL;

- As derivações das fórmulas de $\hat{N}$, E e T são muito questionáveis por se basearem em alguns passos considerados errôneos;

- A suposição de que o homem é capaz de fazer um número constante de discriminações por segundo é bastante questionada pelos psicólogos contemporâneos;

- Os experimentos de validação foram mal formulados: pequeno número de casos, tamanho pequeno dos programas, experimentação com apenas um programador e a utilização de trabalhos de estudantes;

- O número de operadores distintos η_1 pareceu ser aproximadamente constante em programas nas linguagens estruturadas como Pascal e PL/S, em alguns trabalhos;

- A métrica $\hat{N}$ tende a superestimar N para programas pequenos e a subestimar N para programas grandes;

- O número de parâmetros lógicos de entrada e saída (η_2^*), do qual depende o nível do programa (L), é muito difícil de ser determinado automaticamente;

- O conteúdo de informação I mostrou-se muito mais variável para programas equivalentes do que sugere Halstead;

- A métrica de esforço tende a superestimar o esforço real para grandes programas;

- A estimativa de esforço é difícil de ser feita antes de o programa ser terminado. E um modelo de estimativa, proposto por Halstead, baseia-se em η_2^*, que nem sempre pode ser estimado precisamente e no nível da linguagem, que está sujeito a uma variabilidade muito grande.

Espera-se que o presente trabalho contribua para a elucidação desses pontos críticos da Ciência de Software.

4.2 - Número Ciclomático - NC

Nesta seção, citam-se alguns dos trabalhos mais relevantes acerca da métrica de McCabe. Esses trabalhos verificaram o uso da métrica como determinadora do limite de complexidade de módulos e como previsora do tempo de programação.

A utilização do número ciclomático como determinador do limite máximo de complexidade de uma rotina foi testada em [WALS79], no qual verificou-se que rotinas com complexidade maior que 10 eram muito mais propensas a erros do que aquelas com complexidade menor. Já [RAMB85] acredita que um limite de complexidade de 14 é mais razoável para a identificação de rotinas mais propensas a erros. Em [SCHN79], um estudo em ALGOL totalizando 31 rotinas, descobriu-se que do total de 64 erros, 40 deles estavam em rotinas cujo número ciclomático era maior que 5. A partir destes estudos, pode-se concluir que as rotinas de maior complexidade são mais propensas a erros, provavelmente

por serem mais difíceis de testar.

Com relação ao esforço de desenvolvimento (tempo de programação para as fases de projeto, codificação e testes), os poucos trabalhos encontrados produziram resultados contraditórios. Em [CONT86], um experimento envolveu dois conjuntos de programas em Fortran com 143 e 77 rotinas, desenvolvidos em um concurso na Universidade de Purdue, EUA, para estudantes fluentes na linguagem. O número ciclomático foi relacionado com o tempo total de programação e obteve coeficientes de correlação linear de 0,51 e 0,78. Estes coeficientes são considerados regulares ao serem comparados com as outras métricas estudadas (linhas de código, esforço e comprimento N).

Basili et alii analisaram em [BASI83b] um software básico para satélites da NASA desenvolvido em Fortran, em um total de sete projetos. O coeficiente de correlação linear encontrado entre o número ciclomático e o tempo de programação foi de 0,48, para 215 rotinas retiradas de todos os projetos. Mas, quando as rotinas são separadas por projeto, este coeficiente foi de até 0,79.

Já em [KOKO88], foi feita uma análise com 19 programas da Faculdade de Ciências Técnicas na Iugoslávia, desenvolvidos em Pascal com 1026 a 4062 linhas de código. A correlação linear encontrada entre o número ciclomático e o tempo de programação foi de 0,938. O tempo de programação incluiu as fases de projeto, codificação, teste e correção de erros, e foi obtido entrevistando-se os programadores.

O número ciclomático é em sua fundamentação uma métrica intensiva de código, ou seja, não mede a quantidade de código e

sim a sua complexidade (de fluxo de controle). Por este motivo, a correlação entre o número ciclomático e o número de linhas de código, tida como uma métrica extensiva, deve ser relativamente baixa. A variação dos coeficientes encontrados, mostrados anteriormente na tabela 4.4, é grande demais para que se chegue a alguma conclusão. Espera-se que outros estudos investiguem a causa desta variação.

No próximo capítulo, descreve-se como os experimentos do presente trabalho foram conduzidos. Os critérios de validação, as metodologias de contagem e a ferramenta de cálculo de métricas são os principais tópicos discutidos.

5 - DESCRIÇÃO DOS EXPERIMENTOS

Neste capítulo são descritos alguns aspectos dos experimentos conduzidos neste trabalho. Inicialmente faz-se um breve resumo das experiências efetuadas em Pascal e C. Em seguida, os critérios adotados na validação das métricas são discutidos. Por fim, são descritas a ferramenta (denominada de QUANTUM) desenvolvida para automatizar a cálculo das métricas e as metodologias de contagem usadas.

5.1 - Descrição dos Experimentos Efetuados

Faz-se nesta seção uma breve descrição dos experimentos efetuados neste trabalho, cujos resultados são apresentados e discutidos nos capítulos seguintes.

As seguintes métricas foram levadas em consideração nos experimentos em Pascal e em C:

- Métricas da Ciência de Software: η_1, η_2, η, N_1, N_2, N, $\hat{N}$, $\hat{N}_j$, V, $\hat{L}$, NL, E, $\hat{T}$ — calculadas tanto para rotinas como para programas;
- Número ciclomático;
- Número de linhas de código executável;
- Tempo de projeto e codificação de cada rotina (apenas nos experimentos em Pascal).

O cálculo destas métricas foi feita automaticamente pela ferramenta Quantum, descrita na seção 5.3, com exceção do tempo de projeto e codificação levantado para as rotinas em Pascal.

Esse item leva em consideração o tempo para desenvolvimento da rotina, a partir da especificação detalhada da sua função. Não foram incluídas neste tempo as fases de requisitos e especificação, projeto global do sistema (inclusive modularização) e testes. Quando a informação sobre o tempo de implementação não era disponível, o que foi freqüente, entrevistaram-se os programadores para se estimar este dado. Nesta estimativa teve-se o cuidado de tentar considerar o tempo realmente gasto na implementação (incluindo código descartado e outros contratempos), ao invés de se estimar o tempo que o desenvolvimento deveria ter levado. Toda medida de tempo neste trabalho é dada em minutos.

Os experimentos de validação de métricas efetuados são divididos em duas etapas distintas: consistência interna e consistência externa. Em ambas as etapas, são utilizados técnicas e parâmetros estatísticos para fundamentação das análises e conclusões.

A etapa de **consistência interna** verifica o comportamento de diversas métricas e suposições nas quais se baseiam suas teorias. Alguns dos tópicos estudados nesta fase são:

- Estudo das métricas estimadoras de comprimento;
- Análise da variabilidade da métrica I (conteúdo de informação), para programas equivalentes;
- Análise e cálculo da média do nível de linguagem;
- Relacionamento entre as principais métricas estudadas.

Na etapa de **consistência externa**, realizada para a linguagem Pascal, verifica-se a validade das diversas métricas como

estimadoras do tempo de implementação. Esta verificação é feita considerando amostras constituídas de rotinas e amostras constituídas por programas. Aplicam-se, nesta fase, regressões lineares e não-lineares para as métricas, e através dos resultados destas regressões faz-se uma análise comparativa da adequação das métricas. As métricas são julgadas de acordo com os critérios discutidos na próxima seção.

5.2 - Validação Estatística das Métricas

Para que uma métrica possa ser utilizada efetivamente, é necessário que se faça uma série de testes estatísticos para a sua validação. A importância da elaboração de testes estatísticos cuidadosos foi levantada por vários pesquisadores [MORA78; FENI79; SHEN83], que criticam a validação de métricas baseada apenas no coeficiente de correlação linear. Um exemplo de má elaboração de testes estatísticos foi a validação inicial feita por Halstead (e outros pesquisadores) da sua teoria, alvo de muitas críticas [SHEN83; BOEH84]. Não só as validações foram feitas baseando-se apenas em médias e correlações estatísticas, como o número de casos em alguns experimentos foi muito pequeno (às vezes abaixo de 10).

Neste trabalho, foram feitas análises estatísticas criteriosas, considerando, além do coeficiente de correlação linear, outros parâmetros estatísticos importantes. Além disso, sempre que possível, foram feitas também regressões não lineares (logarítimica, polinomial, exponencial e exponencial de base e) ao se tentar inferir a relação entre duas variáveis.

Os principais parâmetros estatísticos selecionados para este trabalho foram: desvio padrão (da média, dos coeficientes de uma relação e de estimativa), coeficiente de dispersão, coeficientes de correlação linear e não-linear, desvio relativo médio, desvio quadrático médio e índice de acertos de previsões sob um determinado nível de erro.

No apêndice D encontram-se definições para todos os parâmetros citados e alguns dos conceitos básicos de Estatística. São também apresentadas as fórmulas de cálculo para regressão linear, coeficiente de correlação e outros parâmetros.

A avaliação da adequação de um modelo baseado em métricas é freqüentemente uma questão subjetiva. Como os parâmetros estatísticos são independentes entre si, um modelo que tem boa adequação de acordo com um parâmetro pode ser considerado inaceitável de acordo com outro. Para servir de guia, foram adotados neste estudo limites de aceitação para o desvio relativo médio, o desvio quadrático médio e o índice de acerto de previsões, sugeridos por [CONT86] e exibidos no quadro 5.1. De acordo com [CONT86], um modelo baseado em métricas precisa satisfazer pelo menos um desses limites para ser considerado aceitável. Esses limites, longe de serem definitivos, podem mudar à medida que os modelos baseados em métricas de software evoluam.

Parâmetro Estatístico	Símbolo	Valor Desejável
Desvio Relativo Médio	DRM	$\leq 0,25$
Desvio Quadrático Médio	DQM	$\leq 0,25$
Prognósticos com Erro $\leq 25\%$	Prev(0,25)	$\geq 75\ \%$

Quadro 5.1 - Valores Desejáveis para os Parâmetros Estatísticos

Os principais parâmetros estatísticos selecionados para este trabalho foram: desvio padrão (da média, dos coeficientes de uma relação e de estimativa), coeficiente de dispersão, coeficientes de correlação linear e não-linear, desvio relativo médio, desvio quadrático médio e índice de acertos de previsões sob um determinado nível de erro.

No apêndice D encontram-se definições para todos os parâmetros citados e alguns dos conceitos básicos de Estatística. São também apresentadas as fórmulas de cálculo para regressão linear, coeficiente de correlação e outros parâmetros.

A avaliação da adequação de um modelo baseado em métricas é freqüentemente uma questão subjetiva. Como os parâmetros estatísticos são independentes entre si, um modelo que tem boa adequação de acordo com um parâmetro pode ser considerado inaceitável de acordo com outro. Para servir de guia, foram adotados neste estudo limites de aceitação para o desvio relativo médio, o desvio quadrático médio e o índice de acerto de previsões, sugeridos por [CONT86] e exibidos no quadro 5.1. De acordo com [CONT86], um modelo baseado em métricas precisa satisfazer pelo menos um desses limites para ser considerado aceitável. Esses limites, longe de serem definitivos, podem mudar à medida que os modelos baseados em métricas de software evoluam.

Parâmetro Estatístico	Símbolo	Valor Desejável
Desvio Relativo Médio	DRM	$\leq 0,25$
Desvio Quadrático Médio	DQM	$\leq 0,25$
Prognósticos com Erro $\leq 25\%$	Prev(0,25)	$\geq 75\ \%$

Quadro 5.1 - Valores Desejáveis para os Parâmetros Estatísticos

5.3 - QUANTUM - A Ferramenta Para Cálculo das Métricas

Foi especialmente desenvolvida para este trabalho, uma ferramenta, denominada de Quantum, para o cálculo de métricas de programas em Pascal e C.

Esta ferramenta foi desenvolvida em linguagem C, no ambiente DOS, e tem duas versões: uma para o cálculo de métricas de códigos em Pascal e outra para códigos em C. Para facilitar o reconhecimento das sintaxes das duas linguagens foram também utilizados os utilitários LEX e YACC. Em cada versão, a parte escrita em C tem aproximadamente 1500 linhas de código, enquanto as partes escritas para os compiladores LEX e YACC têm cerca de 250 linhas cada.

5.3.1 - Execução

A execução da ferramenta gera um arquivo texto contendo o levantamento das seguintes métricas (para todo o programa):

- Métricas da Ciência de Software;
- Tabela de operandos e operadores, acompanhados das freqüências de uso de cada um deles no código;
- Número ciclomático;
- Número de linhas de código: linhas nulas, linhas de comentário, linhas de declaração, linhas de diretiva e linhas executáveis.

A sintaxe para a execução da ferramenta é a seguinte:

quantum <opções> arq1 arq2 [arq3]

onde arq1 é um arquivo com o código a ser analisado, arq2 é um arquivo que conterá todo o levantamento das métricas e arq3, opcional, é um arquivo texto que conterá os nomes das rotinas e suas métricas correspondentes no formato delimitado. Neste formato, cada linha contém as informações de uma rotina, com o nome desta entre apóstrofos, seguido dos valores das métricas correspondentes, separados por vírgulas. Este formato, por ser compatível com muitos programas (Chart, Wordstar, Dbase, etc.), se constitui em uma maneira prática de se utilizar os dados coletados pela ferramenta em aplicações mais complexas, como: armazenamento em bancos de dados, impressão formatada, regressões estatísticas e outros cálculos matemáticos.

Para se adequar às diversas utilidades, a ferramenta oferece algumas opções de funcionamento. São elas:

'-o' : faz a ferramenta gerar as tabelas de freqüência de operandos e operadores;

'-r' : faz a ferramenta calcular as métricas também para cada uma das rotinas do programa;

'-l xxx' : exclui do cálculo das métricas da Ciência de Software as rotinas com nível de linguagem maior que xxx;

'-d xxx' : define a diretiva xxx para fins de compilação condicional (exclusivo da versão para a linguagem C);

'-p xxx' : calcula, em um só passo, as métricas de programas contidos em vários arquivos. Para isto, deve ser especificado no lugar de xxx o arquivo que contém os nomes dos arquivos com os programas fonte (um nome de arquivo por linha). Esta opção é útil para programas espalhados em diversos arquivos (opção exclusiva da versão para C).

5.3.2 - Funcionamento

O funcionamento da ferramenta pode ser dividido esquematicamente em três módulos. O primeiro módulo, o LEX, lê os caracteres do código fonte, identifica os tokens e os passa para o segundo módulo, o YACC. Este módulo é responsável, principalmente, pela coleta dos dados. O terceiro módulo reúne todas as rotinas básicas do programa, utilizadas tanto pelo módulo LEX como o YACC. As principais funções de cada módulo são descritas resumidamente a seguir.

O módulo LEX, ao analisar o código fonte, tem como principais funções:

- identificar os tokens da linguagem;
- ignorar todos os comentários;
- identificar diretivas;
- distingüir literais, constantes inteiras e reais;
- detectar a ocorrência de typecasts ou moldes (exclusivo da versão para C);
- identificar o tipo de linha sendo processada;
- enviar os tokens devidamente identificados para o YACC.

O módulo YACC, por sua vez, é responsável por:

- interpretar diretivas de compilação condicional (exclusivo da versão para C);
- interpretar diretivas de inclusão de arquivos;
- identificar os trechos executáveis do código;
- fazer o devido tratamento de rotinas aninhadas (exclusivo para a versão em Pascal);

- fazer a contagem dos diferentes tipos de linha;
- interpretar a semântica de operadores ambíguos; por exemplo o '-' pode ser um operador binário ou unário, e o '*' um operador de multiplicação ou de indireção na linguagem C;
- interpretar se um identificador é uma variável ou chamada de rotina. A versão para Pascal gerencia uma lista com as rotinas da biblioteca e do usuário, devido à dificuldade de se diferenciar uma variável e uma chamada de função nesta linguagem;
- gerenciar as tabelas de operadores e operandos (rotinas, variáveis, constantes string, inteira e real), controlando a freqüência na rotina e no programa inteiro.

O terceiro módulo contém todas as rotinas básicas chamadas pelos módulos anteriores. Estas rotinas podem ser dispostas nas seguintes categorias:

- Rotinas de acesso a arquivos de entrada (programas fonte);
- Rotinas de acesso a arquivos de saída (resultados);
- Rotinas de gerenciamento das listas de operandos e operadores;
- Rotinas de gerenciamento da lista de tipos definidos pelo usuário (exclusivo para C);
- Rotinas de gerenciamento das diretivas definidas (exclusivo para C);
- Rotinas de gerenciamento da lista de arquivos e unidades já lidas (evitam a releitura de arquivos);
- Rotinas de gerenciamento da lista de arquivos incluídos a

serem lidos (exclusivo para Pascal);

- Rotinas de gerenciamento da lista de rotinas das bibliotecas disponíveis no Turbo Pascal - versão 4 e 5;
- Rotinas de cálculo das métricas da Ciência de Software.

As metodologias de cálculo seguidas por esta ferramenta são descritas nas próximas seções.

5.4 - Metodologia de Contagem das Métricas da Ciência de Software

O uso apropriado das métricas da Ciência de Software depende da adoção de uma estratégia bem definida de contagem de operandos e operadores, dos quais dependem todas as outras métricas.

Halstead nunca forneceu uma definição precisa para operandos e operadores, dando origem a ambigüidades. Por exemplo, a construção IF ... THEN ... ELSE ... é considerada como um único operador em [LASS79] e [BASI83b], e como dois em [ZWEB79a]. A metodologia aqui adotada, especificada na seção 5.4.1, é baseada no trabalho de Salt [SALT82] que apresentou regras de contagem para a linguagem Pascal. Estas regras foram também seguidas por outros pesquisadores [FELI89; JOHN81]. Na falta de trabalhos publicados acerca da aplicação da Ciência de Software à linguagem C, adaptou-se a metodologia para esta linguagem, como descreve a seção 5.4.2.

Uma questão relevante, pouco considerada nos artigos consultados, diz respeito a programas constituídos por várias ro-

tinas. A dúvida está em como proceder no cálculo das métricas.

São poucos os trabalhos encontrados na bibliografia que abordam esta questão. A teoria de Halstead a princípio foi elaborada para aplicação em uma rotina. E, por isso, ele sugere que o esforço para um programa composto de várias rotinas deve ser calculado somando-se os valores do esforço calculados para cada uma das rotinas [HALS77].

Para o cálculo de $\hat{N}$, Halstead, baseado no trabalho de Ingojo [INGO75], acredita que este estimador tanto pode ser calculado pela soma dos $\hat{N}$ de cada rotina, como para todo o programa. Nas suas palavras, "parece razoável assumir que a relação vocabulário-comprimento [isto é, a Equação do Comprimento] governa as partes individuais de um programa logicamente decomposto tão bem quanto o programa inteiro" [HALS77, p. 18]. Esta opinião é também compartilhada por [SCHN88a].

Através de um exemplo em Pascal, Salt calcula as métricas de Halstead para o programa inteiro, ignorando a divisão em rotinas. O mesmo procedimento foi usado em [JOHN81; DAVI87 e FELI89]. Isso pode ter sido a causa de maus resultados, pois no presente trabalho faz-se uma análise comparativa dos dois procedimentos, chegando-se à conclusão de que as métricas $\hat{N}$ e E de programas são mais precisas se calculadas para cada rotina do programa e depois somadas.

5.4.1 - Metodologia de Contagem de Operandos e Operadores Para a Linguagem Pascal

As seguintes regras para contagem de operandos e operado-

res do Pascal padrão elaboradas por Salt [SALT82] foram seguidas neste trabalho (com modificações nos ítens 2 e 9 e criação do item 10):

1. Apenas entidades da parte executável de um programa são consideradas. O cabeçalho do programa, partes de declaração e comentários são ignorados.

2. Variáveis, constantes (incluindo as constantes-padrão: FALSE, TRUE, MAXINT, etc.), literais, nomes de arquivo e a palavra reservada NIL são contadas como operandos.

3. As seguintes entidades são sempre contadas como um operador:

*	/	DIV	MOD	<	<=
=	<>	>=	>	:=	^
,	..	NOT	AND	OR	IN
ELSE	TO	DOWNTO			

4. Os seguintes pares de entidades são sempre contados como um operador:

BEGIN END	CASE END	WHILE DO
REPEAT UNTIL	IF THEN	FOR DO
WITH DO		

5. As seguintes entidades ou pares de entidades são operadores que podem ser contados de maneiras diferentes, conforme as observações descritas abaixo:

   + pode ser contado tanto como um + unário ou como um + binário, dependendo de sua função.

   - pode ser contado tanto como um - unário ou como um - binário, dependendo de sua função.

   . pode ser contado tanto como um símbolo de seletor de componente de um record ou como um terminador de programa, dependendo de sua função.

   : não é contado quando segue um rótulo (label).

   ; só é contado quando é requerido pela sintaxe; todos os ponto-e-vírgulas redundantes são ignorados.

   () pode ser contado tanto como um operador delimitador de argumentos nas chamadas de rotinas ou como um delimitador de expressões, dependendo da função.

[] pode ser contado tanto como um operador de subscrição ou como um operador de conjunto, dependendo da função.

6. Chamadas de procedimentos e funções são contadas como operadores.

7. Uma instrução GOTO (isto é, o GOTO e o rótulo que o acompanha) é contada como um operador.

8. Rótulos não são contados.

9. As cláusulas do CASE são contadas como um operando (a constante) e um operador (':'). (Salt preferiu contar as cláusulas inteiras como operadores, com a característica particular de que mesmo as cláusulas idênticas são contadas como operadores diferentes.)

10. Se a contagem não for feita para cada uma das rotinas isoladamente e sim para o programa como um todo, todos os operandos serão contados como se tivessem escopo global. Em outras palavras, as variáveis locais com o mesmo nome em diferentes rotinas serão contadas como múltiplas ocorrências do mesmo operando.

5.4.1.1 - Extensão Para o Turbo Pascal - Versões 4 e 5

Para análise de programas em implementações específicas do Pascal, se tornam necessárias algumas extensões nas regras citadas anteriormente. Para as implementações do Turbo Pascal, versões 4 e 5, da Borland International, as seguintes regras adicionais foram seguidas:

1. A cláusula USES e a diretiva $I fazem com que o analisador passe a ler o arquivo referenciado, se este ainda não tiver sido lido. Todas as outras diretivas são ignoradas.

2. As cláusulas INTERFACE, FORWARD, EXTERNAL e INLINE e declarações de rotinas fazem com que os nomes das rotinas declaradas sejam armazenados numa tabela de operadores, com o contador de ocorrência igual a zero, inicialmente.

3. O operador + representa três operadores diferentes, conforme o contexto: + unário, + binário e + de concatenação de strings.

4. As entidades seguintes são contadas como operadores:

   # @ $

5. A cláusula ELSE de um CASE é contada como qualquer outro ELSE.

5.4.2 - Metodologia de Contagem de Operandos e Operadores Para a Linguagem C

As seguintes regras para contagem de operandos e operadores da linguagem C foram seguidas neste trabalho:

1. Apenas entidades da parte executável de um programa são consideradas. O cabeçalho do programa, partes de declaração, definições de macroinstruções e comentários são ignorados.

2. Diretivas de inclusão de arquivos e de compilação condicional são interpretadas de acordo com a sua função; outras diretivas são ignoradas.

3. Variáveis e constantes inteiras, flutuantes, de caractere, de enumeração e de cadeia (incluindo os sufixos designadores de tipo), definidas pelo usuário ou padrão da linguagem, são contadas como operandos.

4. Se a contagem não for feita para cada uma das rotinas isoladamente e sim para o programa como um todo, todos os operandos serão contados como se tivessem escopo global. Em outras palavras, as variáveis locais com o mesmo nome em diferentes rotinas serão contadas como múltiplas ocorrências do mesmo operando.

5. As seguintes entidades são sempre contadas como um operador:

++	--	;	.	->	<
<=	>	>=	==	!=	&&
||	*=	/=	%=	+=	-=
<<=	>>=	&=	^=	|=	...
<<	>>	/	%	!	=
,	|	~	^	ASM	BREAK
CASE	CONTINUE	DEFAULT	ELSE	RETURN	SIZEOF

6. As seguintes entidades compostas de mais de um token são sempre contadas como sendo um só operador:

{}	[]	()	? :
DO WHILE ()	FOR ()	IF ()	SWITCH ()
WHILE ()			

7. As seguintes entidades ou pares de entidades são operadores que podem ser contados de maneiras diferentes, conforme as observações descritas abaixo:

   + pode ser contado tanto como um + unário ou como um + binário, dependendo de sua função.
   - pode ser contado tanto como um - unário ou como um - binário, dependendo de sua função.

   & pode ser contado tanto como um operador E bit-a-bit ou como um operador de endereço, dependendo de sua função.

   * pode ser contado tanto como um operador de multiplicação ou como um operador de indireção, dependendo de sua função.

   : só é contado quando faz parte de uma cláusula do SWITCH que não seja DEFAULT. É ignorado quando faz parte de outros rótulos ou da construção ? :.

8. Chamadas de funções e referências a macroinstruções parametrizadas são contadas como operadores. Os parênteses são ignorados por acompanharem sempre uma chamada de uma função.

9. Rótulos são ignorados.

10. Uma instrução GOTO (isto é, o GOTO e o rótulo que o acompanha) é contada como um único operador.

11. As cláusulas do SWITCH são contadas como um operando (a constante) e um operador (':').

12. Moldes (typecasts) são considerados como um único operador.

5.4.2.1 - Análise Crítica da Metodologia Adotada Para C

Sendo um trabalho pioneiro na linguagem C, alguns aspectos desta metodologia de contagem de operandos e operadores devem ser estudados com mais profundidade e aperfeiçoados.

O procedimento quanto às macroinstruções foi simplista. Todas as definições são ignoradas e as referências consideradas como operandos, exceto se a macroinstrução for parametrizada, quando então é considerada como um operador. Embora essa abor-

dagem pareça satisfatória para as rotinas que não definem macroinstruções, mas apenas fazem referência a elas, não considerar o conteúdo da macroinstrução pode levar a imprecisões maiores. Afinal a macroinstrução pode conter instruções que tomam tempo para a criação e compreensão de um programa.

O mesmo ocorre com as definições de variáveis. Ao contrário de outras linguagens, é comum em C o uso de declarações de variáveis bastante complexas. Apesar de se demandar esforço dos programadores na criação e compreensão de tais declarações, estas podem diminuir a complexidade do código. Na abordagem aqui adotada, preferiu-se ignorar todas as declarações de variáveis, como é feito normalmente na aplicação das métricas da Ciência de Software em outras linguagens. Entretanto, um estudo que avaliasse o efeito das declarações seria bastante oportuno.

5.5 - Metodologia de Contagem do Número Ciclomático

Contrastando com as métricas da Ciência de Software, há muito pouca ambigüidade e imprecisão na definição do número ciclomático de McCabe — tido como o número de decisões no código fonte mais um. Para se descobrir o número de decisões no programa é preciso, apenas, contar as instruções condicionais e os operadores booleanos.

Há, porém, uma pequena controvérsia com relação ao operador booleano NOT. Enquanto há pesquisadores que incluem o operador no cálculo [SHEN85 e ARTH85] outros não o consideram [CONT86]. McCabe no seu trabalho original [MCCA76] não menciona

quais os operadores booleanos que devem influenciar no cálculo. Neste trabalho, preferiu-se considerar o operador NOT, tendo em vista que, embora não modifique a estrutura de decisões do programa, ele dificulta a compreensão de um predicado condicional, aumentando assim a complexidade psicológica.

No quadro 5.2 encontram-se os tokens que contribuem no cálculo do número ciclomático nas linguagens C e Pascal, de acordo com a metodologia aqui adotada.

Linguagem	Instruções	Operadores Booleanos
PASCAL	IF, REPEAT, WHILE, FOR, CASE*	AND, OR, NOT
C	IF, FOR, WHILE, DO, ? :, SWITCH*	&&, ||, !
* Ao contrário das outras instruções, CASE e SWITCH acrescentam ao número ciclomático o número de casos que contêm, não se considerando as cláusulas ELSE (Pascal) e DEFAULT (C), que não influem no cálculo.		

Quadro 5.2 - Instruções que Influem no Cálculo do Número Ciclomático

5.6 - Metodologia de Contagem de Linhas de Código

Mesmo sendo a métrica de linhas de código a mais popular e uma das mais simples de se obter, não existe concordância sobre o que é uma linha de código. Enquanto alguns interpretam todo caractere de fim de linha como sendo uma linha de código, outros descartam as linhas nulas, linhas de comentário, linhas de declaração ou linhas de continuação. São bastante numerosas as diferentes possibilidades de contagem de linhas de código.

Neste trabalho, buscou-se uma metodologia que fosse coerente e que estivesse de acordo com as práticas mais comuns. Assim, uma linha de código é enquadrada em apenas uma das categorias a seguir (em ordem decrescente de prioridade):

1) linha executável
2) linha de declaração
3) linha de diretiva
4) linha de comentário
5) linha nula

Se uma linha contiver elementos de duas ou mais categorias, ela se enquadra na categoria de maior prioridade. Por exemplo, uma linha com diretiva e com comentário é considerada uma linha de diretiva. Já uma linha com diretiva e código executável é considerada uma linha executável.

As métricas de linhas de código mais comuns encontradas na bibliografia são: número total de linhas de código (LDCt), que considera todas as categorias exceto as de linhas nulas, e o número de linhas de código executável (LDCe), que considera apenas a primeira categoria.

Neste capítulo foram apresentados os critérios e metodologias usados na condução dos experimentos deste trabalho. Os resultados obtidos nos experimentos são discutidos nos próximos capítulos.

6 - EXPERIMENTOS REALIZADOS COM PASCAL

6.1 - Características da Amostra Analisada

Nos experimentos com métricas aplicadas à linguagem Pascal, foram analisados 16 programas cujos tamanhos variam de 43 a 3096 linhas de código executável. Além desses programas, foi também considerada uma biblioteca de rotinas básicas, utilizada pelos seis maiores programas analisados.

Os programas aplicativos são em sua maioria comerciais, tais como: controle de contas a pagar, contas a receber, estoque e aluguel. Outros programas são de engenharia de irrigação e utilitários.

Para proporcionar uma maior homogeneidade nas amostras de dados, dividem-se, nos experimentos, os programas em três categorias, de acordo com suas características funcionais:

- **Categoria I:** engloba todas as rotinas que formam uma biblioteca básica, com funções de controle de tela, impressora, teclado, disco e de interação com o usuário, entre outras.
- **Categoria II:** contém os programas que utilizam a biblioteca básica citada na categoria anterior.
- **Categoria III:** compreende os programas que não utilizam a biblioteca de rotinas especiais, ou seja, utilizam apenas as funções oferecidas pela própria linguagem.

Os programas e rotinas são distribuídos nestas categorias de acordo com a tabela 6.1.

Categoria	Nº de Programas	Nº de Rotinas	LDCe
I	—	70	3.360
II	6	225	10.879
III	10	61	4.181
Total	16	356	18.420

Tabela 6.1 - Distribuição dos Programas Analisados (Pascal)

6.2 - Consistência Interna

O propósito desta seção é verificar o comportamento das métricas estudadas e as suposições nas quais se baseiam. Isso é feito estudando diversos relacionamentos estatísticos existentes entre as métricas. Chama-se essa etapa de verificação da consistência interna das métricas, por não ser feito nenhum estudo de correlação com algum dado relacionado ao desenvolvimento de software em si, como tempo de programação ou número de erros.

6.2.1 - Estimadores de Comprimento

Nesta seção, verifica-se a precisão das duas métricas estimadoras do comprimento de código: $\hat{N}$ (fórmula 3.3) e $\hat{N}_j$ (fórmula 4.1). A verificação é feita em duas diferentes abordagens. A primeira consiste na aplicação destas métricas como estimadoras do comprimento de rotinas. A segunda abordagem verifica a precisão da métrica como estimadora do comprimento de progra-

mas.

A análise da precisão dos estimadores é feita através da aplicação de regressões lineares e não-lineares simples para as métricas $\hat{N}$ e $\hat{N}_j$, quando relacionadas aos comprimentos reais observados.

São também feitas análises da variação da precisão dos estimadores quanto ao tipo e comprimento das rotinas ou programas.

6.2.1.1 - Estimadores de Comprimento Aplicados a Rotinas

Nesta seção, tenta-se avaliar a precisão dos estimadores de comprimento $\hat{N}$ e $\hat{N}_j$, quando aplicados individualmente a uma rotina. Para isso, traça-se o gráfico formado pelos valores de N e $\hat{N}$ (ou $\hat{N}_j$) para as rotinas consideradas e faz-se uma regressão linear para cada métrica. A reta resultante da regressão é forçada a passar pelo ponto (0,0) conhecido. Inicialmente será feita uma análise para o estimador $\hat{N}$.

Estimador $\hat{N}$:

A tabela 6.2 mostra os valores dos parâmetros estatísticos resultantes da aplicação de regressão linear entre $\hat{N}$ e N. Apresentam-se os resultados das regressões efetuadas para todas as rotinas e para as rotinas de cada uma das três categorias. (Para descrição das abreviaturas usadas nas tabelas, ver Lista de Abreviaturas na página xiii.)

Categoria	r	b	DRM	Prev(0,25)	DQM
I	0,928	1,024	0,668	18,5%	0,588
II	0,904	0,890	0,317	53,8%	0,424
III	0,950	1,072	0,604	31,1%	0,419
Todas	0,915	0,946	0,427	36,8%	0,463

Tabela 6.2 - Adequação de $\hat{N}$ na Estimativa de N
(Total: 356 rotinas em Pascal)

Análise da tabela 6.2:

- **Coeficiente de correlação:** Verifica-se na tabela que as correlações entre $\hat{N}$ e N variaram de 0,904 a 0,950, ficando bastante próximos dos valores obtidos por outros estudos. Embora esstes coeficientes sejam considerados bons, os outros índices estatísticos ficaram bem aquém dos desejáveis para um estimador considerado "surpreendentemente preciso" em [FITZ78a].

- **Adequação por categoria:** Com relação à precisão de $\hat{N}$ em cada categoria, verifica-se que a pior adequação foi na biblioteca de rotinas básicas (categoria I). Isto, de certa forma, é esperado pelo próprio caráter excepcional dos algoritmos destas rotinas. As rotinas são projetadas para uso de programas diversos e desta forma não fazem "parte de um programa logicamente decomposto" [HALS77, p. 18], requisito para uma boa adequação da métrica. Já a categoria de rotinas que utiliza a biblioteca (II) foi a única na qual $\hat{N}$ apresentou parâmetros próximos dos aceitáveis. Um possível motivo para isso é a homogeneidade dessa categoria quanto ao tipo de rotinas. Além disso, o grande número de rotinas da amostra diminui o impac-

to de dados espúrios.

- Coeficiente Angular: Os valores do coeficiente angular das regressões lineares foram citados para possibilitar a verificação de sua variação nas diferentes categorias. Esta se mostrou razoavelmente pequena (de 0,890 a 1,072), e o valor médio do coeficiente (0,946) se aproximou muito do valor teórico unitário, com um erro de apenas 5%. Esta dedução é um ponto a favor da teoria de Halstead, pois possibilita o uso da equação do comprimento da maneira que foi formulada, sem a introdução de coeficientes.

Estimador $\hat{N}_j$:

Analisa-se agora a precisão do outro estimador de comprimento ($\hat{N}_j$) que apresentou os parâmetros estatísticos da tabela 6.3, quando relacionado linearmente com o comprimento real das rotinas.

Categoria	r	b	DRM	Prev(0,25)	DQM
I	0,928	1,370	0,530	24,3%	0,560
II	0,913	1,188	0,283	56,0%	0,405
III	0,956	1,407	0,478	27,9%	0,397
Todas	0,923	1,260	0,353	41,6%	0,441

Tabela 6.3 - Adequação de $\hat{N}_j$ na Estimativa de N
(Total: 356 rotinas em Pascal)

A tabela 6.3 revela que o estimador $\hat{N}j$ é melhor do que $\hat{N}$ em praticamente todos os aspectos. Isso apóia a suposição de

[JENS85] que este estimador é mais preciso do que $\hat{N}$. No entanto, a melhora quanto ao Desvio Relativo Médio é bem mais restrita do que nos trabalhos [JENS85] e [LIND89]. Além disso, é importamente observar que contra esse estimador pesa o fato de ter sido necessário o uso de um coeficiente angular de 1,260. A falta de publicação desse coeficiente em outros trabalhos impede a análise de sua variabilidade e implica na necessidade de aplicação de regressão linear no ambiente em que se deseja utilizar a métrica.

A figura 6.1 exibe o relacionamento existente entre $\hat{N}_j$ e N, para as 356 rotinas da amostra.

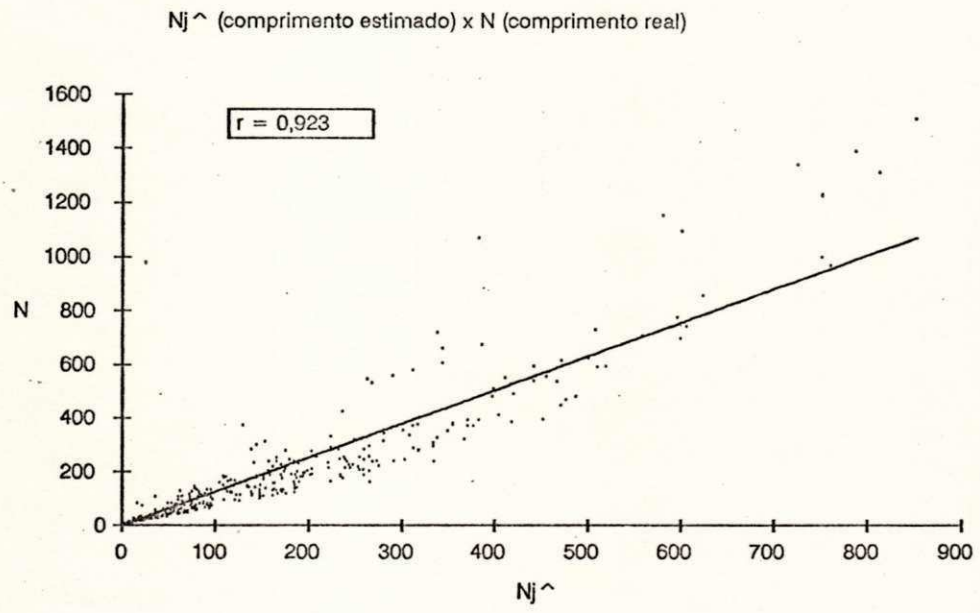


Figura 6.1 - Relacionamento entre $\hat{N}_j$ e N
(356 rotinas em Pascal)

Outras Regressões:

Com o intuito de se encontrar melhores relações entre o

vocabulário e o comprimento de rotinas, outras regressões lineares e não-lineares foram tentadas com as métricas $\hat{N}$, η_1, η_2 e η. As que obtiveram precisão significativa foram as regressões polinomiais do segundo grau para $\hat{N}$ e η, com a curva passando pelo ponto (0,0). As estatísticas destas regressões para todas as rotinas são mostradas na tabela 6.4, onde b e c se referem aos coeficientes da curva de regressão: $Y = cX^2 + bX + 0$.

Métrica	r	b	c	DRM	Prev (0,25)	DQM
$\hat{N}$	0,907	0,583	0,000642	**0,230**	57,6%	0,377
η	0,944	1,429	0,0402	**0,250**	53,6%	0,380

Tabela 6.4 - Estatísticas da Regressão Polinomial do Segundo Grau para $\hat{N}$ e η (356 Rotinas em Pascal)

Em geral, estes dois modelos se mostraram superiores aos analisados anteriormente, podendo inclusive serem reconhecidos como aceitáveis, considerando seus baixos desvios relativos médios (em negrito na tabela). A utilização desses modelos, no entanto, necessita de outros trabalhos confirmatórios.

6.2.1.2 - Precisão dos Estimadores com o Tamanho da Rotina

Na seção 4.1.1 se comentou que estudos em Pascal [JOHN81; DAVI87; FELI89] e em outras linguagens [CHRI81; BASI83b; LI87] mostraram uma tendência de $\hat{N}$ superestimar N para rotinas pequenas (normalmente com $N < 200$) e subestimá-lo para rotinas grandes. A justificativa por parte dos autores, foi de que η_1

permanecia relativamente constante com o aumento de N, refletindo assim em menores valores para $\hat{N}$.

Esta seção verifica essa hipótese analisando as relações $\hat{N}/N$, $\hat{N}_j/N$ e η_1/η para as rotinas agrupadas por ordem de comprimento. A tabela 6.5, com essas relações, mostra que realmente existe uma tendência de queda nas relações com o aumento do tamanho. E embora η_1 esteja longe de ser uma constante, verifica-se que sua participação em η dimuinui com o comprimento das rotinas.

Dessa tabela, também se deduz que $\hat{N}$ superestima N para todas as categorias, sendo mais preciso para rotinas com N variando de 242 a 1511. Com relação a $\hat{N}_j$, em termos de média aritmética de sua relação com N, verifica-se que este é bem preciso em rotinas cujo comprimento varia de 62 a 240.

Rotinas	Variação de N	Média de $\hat{N}$ / N	Média de $\hat{N}j$ / N	Média de η_1 / η
1 - 90	5 - 62	1,78	1,15	0,59
91 - 180	62 - 126	1,44	1,00	0,48
181 - 270	162 - 240	1,39	1,01	0,48
271 - 356	242 - 1511	1,09	0,82	0,41

Tabela 6.5 - Variação das Relações $\hat{N}/N$ e $\hat{N}_j/N$ com o Comprimento das Rotinas (Pascal)

6.2.1.3 - Estimadores de Comprimento aplicados a Programas

Nesta seção, tenta-se descobrir a precisão dos estimadores de comprimento da Ciência de Software quando aplicados aos 16 programas analisados. Foram testados quatro estimadores: os de

Halstead e Jensen, cada um calculado de duas maneiras distintas. Na primeira, os estimadores ($\hat{N}$ e $\hat{N}_j$) são calculados para cada uma das rotinas do programa e então são somados. Na segunda maneira, os estimadores ($\hat{N}_{prog}$ e $\hat{N}_{j\text{-}prog}$) são calculados tratando-se o programa como um todo e ignorando-se os limites das rotinas. Tenta-se, com essa abordagem, descobrir qual a maneira mais precisa de se proceder com as métricas da Ciência de Software em programas constituídos de várias rotinas.

Os parâmetros estatísticos das regressões lineares para esses estimadores estão na tabela 6.6.

Estimador	r	b	DRM	Prev(0,25)	DQM
$\hat{N}$	0,992	0,826	**0,154**	68,8%	**0,120**
$\hat{N}j$	0,992	1,124	**0,149**	**75,0%**	**0,118**
$\hat{N}$prog.	0,941	2,181	0,719	43,8%	0,328
$\hat{N}j$-prog	0,954	2,695	0,626	43,8%	0,289

Tabela 6.6 - Adequação de $\hat{N}$, $\hat{N}_j$, $\hat{N}_{prog}$ e $\hat{N}_{j\text{-}prog}$
Quando Aplicados a 16 Programas em Pascal

Por apresentarem quase todos os parâmetros (em negrito) dentro dos limites desejáveis, citados em 5.2, as métricas $\hat{N}$ e $\hat{N}_j$ são válidas para a previsão do comprimento, para ambientes semelhantes ao analisado. O mesmo não pode ser dito para as métricas $\hat{N}_{prog}$ e $\hat{N}_{j\text{-}prog}$. Embora tenham correlações altas com N, elas apresentaram outras estatísticas abaixo dos valores desejados.

Na comparação entre $\hat{N}$ e $\hat{N}_j$, conclui-se que $\hat{N}_j$ apresenta-se ligeiramente superior ou no mínimo igual a $\hat{N}$ em todos os índi-

ces estatísticos. Até mesmo o seu coeficiente angular foi mais próximo do proposto (desvio de 12,4% contra 17,4%).

O erro padrão de estimativa (não mostrado na tabela) foi de aproximadamente 500 para as métricas $\hat{N}$ e $\hat{N}_j$. Com isto, o intervalo de confiança de 80% para estimativas com estas métricas foi:

Intervalo de Confiança: $\hat{N} \pm 640$

Este resultado pode ser considerado bom, tendo em vista a média de N dos programas ter sido igual a 3964.

A figura 6.2 mostra o relacionamento entre $\hat{N}_j$ e N para os 16 programas da amostra.

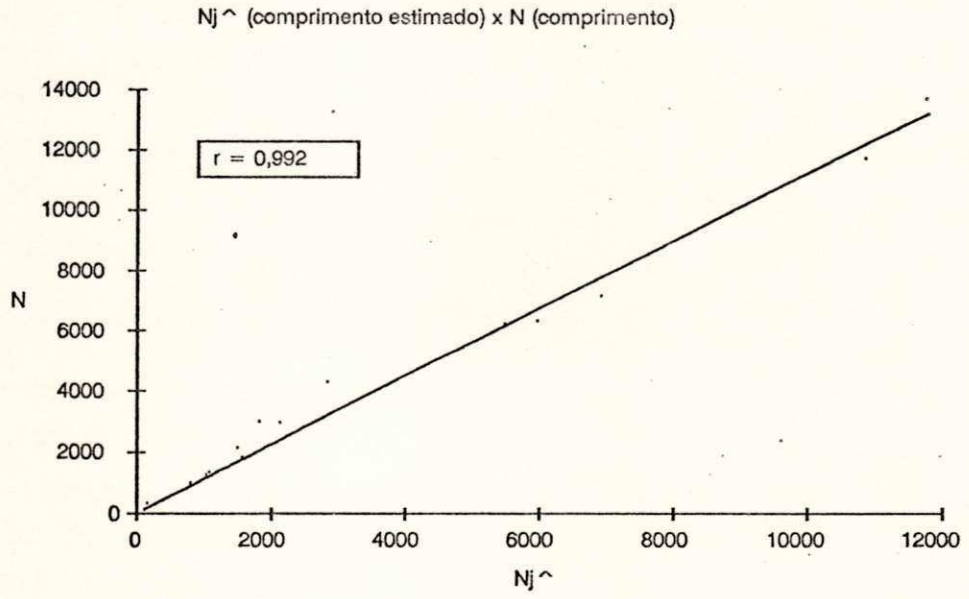


Figura 6.2 - Relacionamento entre $\hat{N}_j$ e N
(16 programas em Pascal)

6.2.2 - Dificuldade e Nível de Programa

Esta seção trata das métricas de dificuldade e nível de

programa por se tratarem de métricas semelhantes (já que uma pode ser escrita em função da outra).

Um experimento realizado em [LI87] e [BASI83b], citado no capítulo 4, para avaliar as maiores influências na métrica de dificuldade é aqui repetido. Calcularam-se, para todas as rotinas, as correlações estatísticas entre esta e seus dois componentes: η_1 e N_2/η_2. A correlação obtida por η_1 (0,894) foi um pouco maior do que a obtida por N_2/η_2 (0,800). Isto sugere que η_1 (número de operadores distintos) é um pouco mais influente no valor da métrica dificuldade.

A única importância da métrica de dificuldade na Ciência de Software é a sua participação direta no cálculo do esforço (E = VxD). Como a métrica de esforço mostrou-se imprecisa (como será visto adiante), outros experimentos com a métrica de dificuldade (e também nível de programa) são irrelevantes e não serão apresentados.

6.2.3 - Nível de Linguagem

De acordo com Halstead, rotinas de uma mesma linguagem apresentam valores da métrica Nível de Linguagem (NL) aproximadamente iguais. Esta seção verifica esta hipótese e calcula o valor médio desta métrica para a linguagem Pascal.

No experimento, segue-se a recomendação feita por [KONS85] para que rotinas com nível de linguagem muito elevado fossem retiradas do cálculo da média. A razão para isso é a suspeita de que essas rotinas contenham impurezas (ver seção 3.1.11). Ao todo foram excluídas da amostra sete rotinas, por terem nível

maior que o limite de 15 adotado. Essas rotinas, de fato, eram caracterizadas pela impureza do tipo V - atribuições desnecessárias. Suas funções eram de iniciar ou modificar o conteúdo de registros de dados.

A tabela 6.7 mostra as médias para todas as rotinas, divididas em categorias por ordem de comprimento. Pelo valor de seu nível médio, a linguagem Pascal se situa acima de Cobol e um pouco abaixo de APL na tabela 4.2. Ainda que o seu valor médio não surpreenda e esteja dentro do esperado, o nível de linguagem apresentou um desvio padrão de 2,56. Este valor tão elevado foi causado pela variação da métrica — desde 0,12 até 14,50. E, mesmo considerando as rotinas de apenas uma categoria (programas puros, por exemplo), esses valores são semelhantes. Além disso, verifica-se também uma queda do nível médio de linguagem com o aumento do tamanho da rotina.

O fato de a métrica nível de linguagem ter grande variabilidade e ser dependente do tamanho da rotina já tinha sido detectado em outros estudos (ver seção 4.1.4). Os resultados obtidos aqui reprovam esta métrica que, de acordo com Halstead, deveria ser praticamente constante para diferentes algoritmos numa mesma linguagem.

Rotinas	NL mínimo	NL médio	NL máximo	Desvio Padrão
1 - 90	0,55	2,95	14,50	2,82
91 - 180	0,12	2,40	7,59	1,77
181 - 270	0,27	2,17	14,12	2,75
271 - 349	0,18	1,65	14,09	2,62
todas	0,12	2,31	14,50	2,56

Tabela 6.7 - Variação do Nível da Linguagem Pascal

6.2.4 - Relacionamento Entre as Principais Métricas

Esta seção objetiva verificar o relacionamento existente entre as principais métricas para a medição de quantidade e complexidade de código: N (comprimento), V (volume), E (esforço), LDCe (linhas de código executável) e NC (número ciclomático). O estudo do relacionamento é feito calculando-se os coeficientes de correlação linear entre cada métrica e as outras. A tabela 6.8 exibe os resultados obtidos para todas as rotinas.

Métricas	N	V	E	LDCe	NC
N	-	0,998	0,906	0,948	0,836
V	0,998	-	0,918	0,950	0,835
E	0,906	0,918	-	0,920	0,849
LDCe	0,948	0,950	0,920	-	0,897
NC	0,836	0,835	0,849	0,897	-

Tabela 6.8 - Correlações Entre Métricas (356 Rotinas em Pascal)

Analisando a tabela, verifica-se que a métrica LDCe apresenta correlações relativamente altas com N e V (0,95). Isso está de acordo com a suposição de que essas métricas medem o volume do código. As métricas E e NC apresentam correlações menores com as métricas extensivas (de 0,84 a 0,92). Isso significa que essas métricas medem aspectos diferentes do código, possivelmente relacionados à complexidade. E como têm correlação baixa entre si, elas são fundamentalmente diferentes.

Quando aplicado a programas, ao invés de rotinas, as correlações entre essas métricas tendem a crescer bastante, variando de 0,945 a 0,999.

Na figura 6.3 visualiza-se, para as 356 rotinas, a seme-

lhança entre N e LDCe e, principalmente, entre N e V. A figura 6.4 mostra a variação do esforço com o comprimento da rotina. Nela, há uma nítida tendência do esforço crescer exponencialmente com N, o que está de acordo com a suposição de que a modularização de software diminui a sua complexidade.

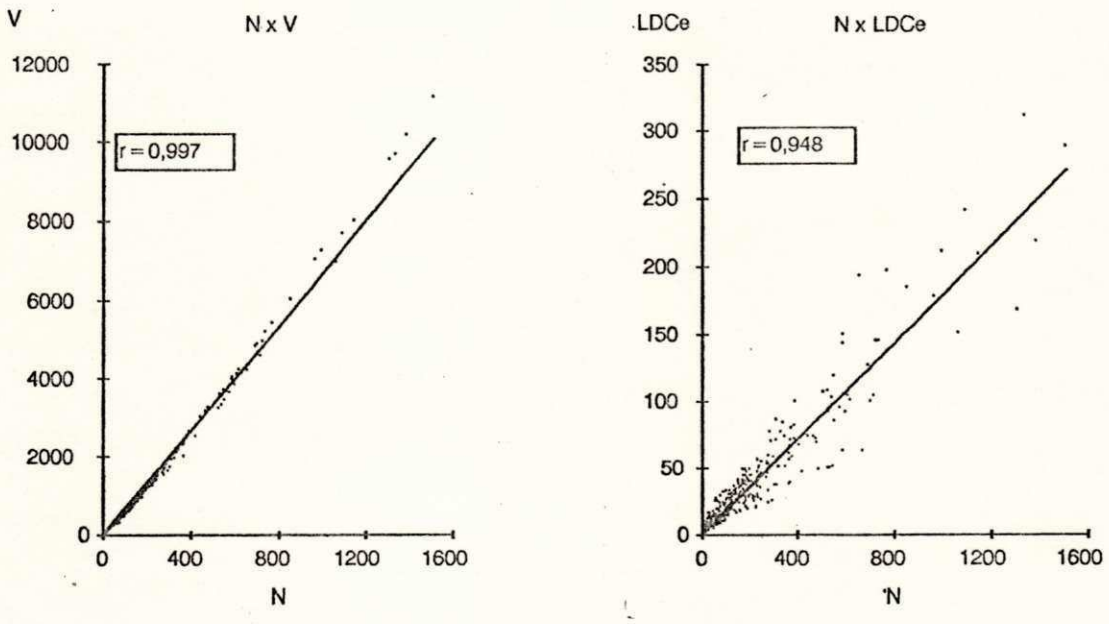


Figura 6.3 - Relacionamentos entre N e V, e N e LDCe

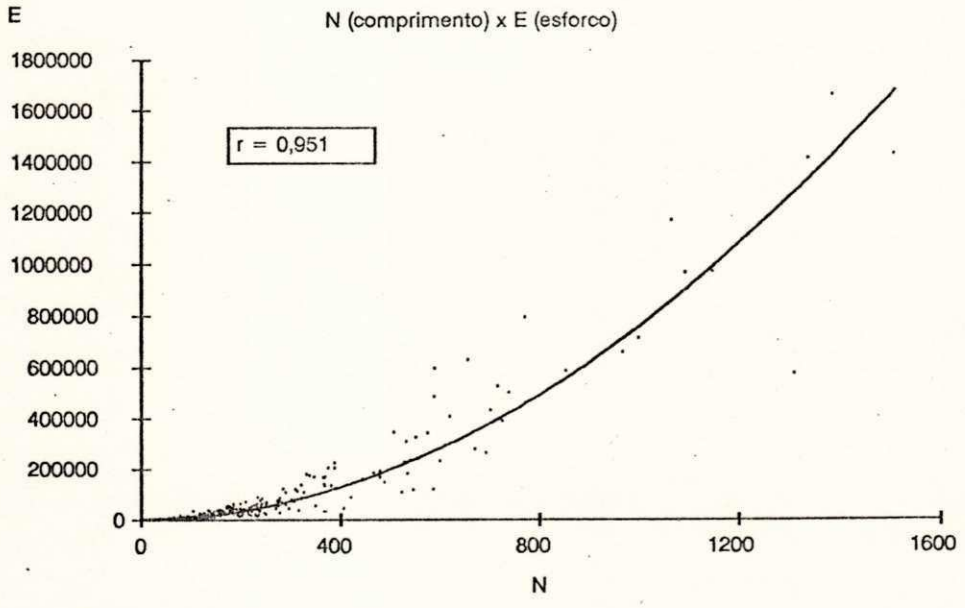


Figura 6.4 - Relacionamento entre comprimento e esforço

6.3 - Consistência Externa

Na etapa de consistência externa das métricas é feito um estudo de correlação entre as métricas de código e o tempo real de programação. Este estudo objetiva a seleção de métricas para pelo menos duas aplicações de grande utilidade na gerência de desenvolvimento:

- Estimativa de tempo de programação feita em dois passos. Inicialmente estima-se, a partir da especificação ou projeto do sistema, o valor da métrica que, por sua vez, servirá para a estimativa do tempo de programação.
- Utilização de métricas para avaliação da quantidade de software desenvolvido e da produtividade.

Foram estudadas as métricas mais comuns de complexidade e quantidade de código: N (comprimento), V (volume), E (esforço), NC (número ciclomático) e LDCe (linhas de código executável). Outras métricas da Ciência de Software (η_1, η_2, N_1, N_2 e $\hat{N}$) também foram tentadas, mas não obtiveram bons resultados.

O tempo de programação medido leva em consideração o tempo de desenvolvimento de uma rotina (em minutos), a partir da especificação de sua função. Não foram incluídas as fases de especificação, projeto global do sistema e testes.

Com relação aos métodos estatísticos foram empregadas regressões lineares e não-lineares, em duas amostras diferentes. A primeira amostra é formada por rotinas e a segunda por programas.

6.3.1 - Estimativa de Tempo de Rotinas

Nesta seção correlaciona-se o tempo de programação com a primeira amostra, consistindo de 356 rotinas em Pascal. Os melhores relacionamentos encontrados foram as regressões lineares para N, V, E e NC, e a regressão exponencial ($y = ax^b$) para E. Os parâmetros estatísticos destes modelos estão na tabela 6.9.

Modelo	r	a	b	DRM	Prev (0,25)	DQM
E	0,775	0	0,00026	0,755	7,6%	0,946
Epot	0,858	0,943	0,342	0,374	46,9%	1,124
N	0,817	0	0,186	0,484	34,6%	0,861
V	0,829	0	0,0280	0,472	32,0%	0,836
LDCe	0,863	0	1,041	0,419	43,3%	0,756
NC	0,853	0	3,196	0,578	28,1%	0,782

Obs: E_{pot} - modelo de regressão exponencial ($y = ax^b$) para E

Tabela 6.9 - Regressão do Tempo de Programação para Vários Modelos (356 Rotinas em Pascal)

Analisando a tabela, verifica-se que, no geral, a precisão das métricas como estimadoras de tempo de programação de rotinas foi baixa, bem aquém do desejável.

Entre as métricas da Ciência de Software, V e N se mostraram significativamente melhores do que a métrica E, originalmente proposta para estimativa de tempo. No entanto, ao se aplicar uma regressão exponencial, esta métrica teve um desempenho estatístico comparável às outras.

Entre as outras métricas testadas, destacam-se: LDCe, por ter se apresentado como a melhor em dois parâmetros (r e DQM),

embora por pequena margem, e NC pelo desempenho estatístico razoável para a métrica que, na fase de projeto, pode ser estimada com mais precisão do que as outras [HENR90].

Todas as rotinas cujo número ciclomático foram menor do que 10 levaram menos de 60 minutos para serem implementadas. Isso leva a crer que essa métrica parece ser uma forma simples de se evitar rotinas muito complexas e que levem mais de uma hora para serem implementadas.

Na tentativa de se obter melhores resultados, foram aplicadas regressões para as rotinas divididas por categorias. Os parâmetros estatísticos, no entanto, foram semelhantes aos da tabela 6.9. Até mesmo os coeficientes angulares não se alteraram muito, refletindo, portanto, a homogeneidade das rotinas quanto ao ponto de vista do tempo de desenvolvimento.

Comparando os resultados aqui obtidos com os publicados, conclui-se que os coeficientes de correlação não foram tão altos quanto os encontrados por Halstead (para a métrica E), nem tão baixos quanto os encontrados por Basili [BASI83b]. Na realidade, os parâmetros r, DRM e Prev(0,25) se assemelham bastante aos do experimento em [CONT86], embora tenham sido utilizados programas em Fortran neste estudo.

Em geral, a confiabilidade das métricas na estimativa de tempo de programação de uma rotina, pelos critérios estabelecidos, é baixa. Mas, de qualquer forma, até agora nenhuma outra técnica de estimativa publicada é comprovadamente melhor.

6.3.2 - Estimativa de Tempo de Programação

Nesta seção correlaciona-se o tempo de programação com a segunda amostra de dados, constituída por 16 programas em Pascal. As métricas dos programas (inclusive o tempo) são obtidas somando os valores das métricas de cada rotina pertencente ao programa. Além das métricas utilizadas na seção anterior, foram incluídas mais duas: o número de rotinas (NR) e a métrica de esforço calculada para o programa como um todo (E_{prog}).

Os melhores modelos encontrados foram as regressões lineares para N, V, E, LDCe, NC e NR, e a regressão exponencial para E. A tabela 6.10 mostra os parâmetros estatísticos calculados.

Modelo	r	a	b	DRM	Prev (0,25)	DQM
E	0,911	0	0,000448	0,371	50,0%	0,373
Eprog	0,850	0	0,0000235	0,645	25,0%	0,476
Epot	0,874	0,0096	0,784	0,287	56,2%	0,438
N	0,966	0	0,166	**0,200**	68,8%	**0,233**
V	0,967	0	0,0273	**0,192**	62,5%	**0,229**
LDCe	0,934	0	0,957	**0,197**	**75,0%**	**0,233**
NC	0,943	0	3,944	**0,196**	**81,0%**	0,299
NR	0,955	0	32,11	0,281	50,0%	0,267

Obs: E_{pot} - modelo de regressão exponencial ($y = ax^b$) para E
NR - número de rotinas

Tabela 6.10 - Regressão do Tempo de Programação para Vários Modelos (16 Programas em Pascal)

Passa-se, agora, a uma análise da precisão dos modelos na estimativa de tempo de programação:

- **Modelos válidos:** De acordo com os critérios adotados, os mo-

delos considerados válidos são aqueles baseados nas métricas N, V, LDCe e NC. Destas métricas destacam-se: V, por ter sido a melhor métrica em três parâmetros (r, DRM e DQM); LDCe, por ser a única com três parâmetros dentro dos limites desejáveis e NC por ter um bom desempenho estatístico e ser mais fácil de se estimar na fase de projeto do que N, V e LDCe.

- **Métricas da Ciência de Software:** A métrica E não apresentou bons parâmetros estatísticos, nem mesmo com uma regressão exponencial. Quando calculada para todo o programa, a métrica de esforço foi ainda pior do que quando calculada para cada rotina e depois somada.

- **Número de rotinas:** Esta métrica teve uma adequação razoável, considerando-se a vantagem de ser mais fácil de conhecer na fase de projeto. Deve-se ter em mente, contudo, que o número de rotinas é normalmente decidido na fase de modularização do software e por isso é muito influenciado pela metodologia adotada, tipo de software ou preferência dos projetistas. Assim, modelos de regressão baseado nesta métrica devem apresentar coeficientes bastante variáveis com o ambiente.

- **Intervalo de confiança:** O desvio padrão da estimativa foi de 221 minutos para NC e aproximadamente 170 minutos para N, V e LDCe. Para estas três últimas métricas o intervalo de confiança a 80% é, portanto, de ± 218 minutos. Um resultado bom, considerando a média da amostra de tempo de programação de 694 minutos.

- **Variabilidade dos coeficientes de regressão:** Para testar este

item, aplicaram-se regressões para programas das categorias II e III. Em seguida, foram calculadas as variações percentuais pela seguinte fórmula:

$$\text{Variação} = \frac{| C_{II} - C_{III} |}{C_t} \qquad (6.1)$$

onde C_{II}, C_{III} e C_t são os coeficientes da regressão para as categorias II, III e total, respectivamente. As variações dos modelos lineares para N, V, LDCe, NC e NR (tabela 6.11) mostram que o modelo baseado em NC é o que apresenta menos variação com o tipo de programa. Como se esperava, o modelo baseado no número de rotinas foi o que apresentou maior variação.

Modelos	N	V	LDCe	NC	NR
Variação (%)	37,9	29,3	24,7	3,2	70,8

Tabela 6.11 - Variação dos Coeficientes Angulares (Fórmula 6.1) em Programas de Categorias Diferentes

- **NC/LDCe:** Esperava-se que a densidade de decisões no programa, isto é, o número ciclomático por linha de código, tivesse influência preponderante no tempo de programação. Desta forma, um programa que apresentasse alta densidade de decisões provavelmente teria o tempo de programação observado maior do que o previsto por uma métrica de volume. Essa hipótese, no entanto, não foi comprovada, tendo em vista o coeficiente de correlação de apenas 0,052 entre NC/LDCe e a relação entre o tempo real e o tempo previsto por V.

Para visualizar melhor a precisão das métricas N, V, LDCe, NC e NR, a tabela 6.12 mostra os tempos reais observados e os estimados por cada modelo para os 16 programas. (Observando essa tabela, verifica-se que a 13ª linha — tempo real de 883 minutos — foi estimada com erro elevado por praticamente todos os modelos, merecendo, por isso, uma explicação. Trata-se de um programa de impressão de etiquetas que permite o uso de diversas fontes de letras diferentes. A combinação de numerosos tamanhos de letras e de etiquetas exigiu a realização de cálculos relativamente complexos para avaliação do espaço disponível para texto na etiqueta. Como esses cálculos se refletem sutilmente no código final, as métricas não conseguem estimar com precisão o tempo levado para efetuar esses cálculos.)

Tempo real	Tempo estimado por				
	N	V	LDCe	NC	NR
20	19,8	17,8	31,6	27,6	32,1
30	22,8	20,8	34,4	31,6	32,1
80	54,3	47,4	66,0	78,9	64,2
188	164,3	158,0	168,4	284,0	160,6
247	208,7	227,8	267,9	248,5	160,6
286	224,8	252,8	271,7	272,1	128,4
386	493,0	525,2	463,1	532,4	224,8
430	299,9	309,2	341,6	323,4	385,3
666	351,8	361,4	379,9	536,4	353,2
718	709,4	764,3	754,0	820,4	385,3
826	1034,4	1010,2	1033,4	989,9	1059,6
865	1050,7	1031,4	1134,8	1080,6	1059,6
883	488,7	503,3	560,7	812,5	481,6
1202	1188,2	1156,1	1208,5	1001,8	1316,5
1799	1947,8	1976,1	1954,8	2192,9	1701,8
2483	2279,4	2258,4	2173,9	1853,7	2472,5

Tabela 6.12 - Tempo de Programação Real Versus Estimado por Diversos Modelos (Pascal)

A figura 6.5 ilustra o relacionamento existente entre V e o tempo de programação.

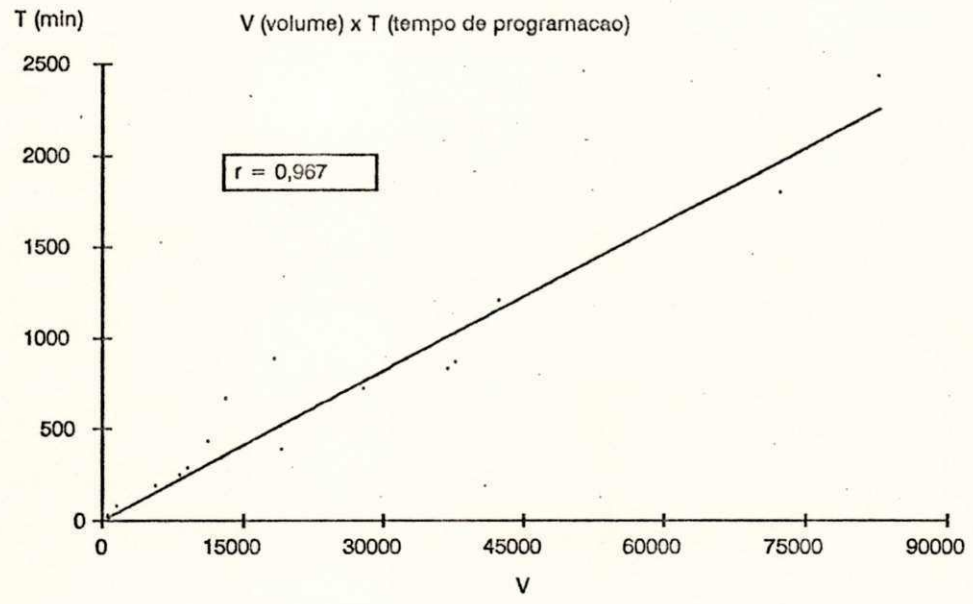


Figura 6.5 - Gráfico do tempo real versus o tempo estimado por V (16 programas em Pascal)

Um outro experimento foi feito correlacionando-se as métricas com o tempo de desenvolvimento total do programa. Este tempo inclui as fases de requisitos, especificações e projeto global, mas exclui a fase de testes de integração das rotinas. A relevância de tal experimento é menor, tendo em vista que as métricas só podem ser estimadas com alguma precisão a partir da fase de projeto, quando as fases de requisitos, especificações e projeto global já estão realizadas. De qualquer forma, os valores estatísticos foram inferiores aos apresentados e a única surpresa foi o fato de o coeficiente angular da melhor reta para o esforço ter sido de 0,000861, o que implica em um valor de S (constante de Stroud) de 19,4, bastante próximo do proposto por Halstead (18).

6.4 - Conclusões

As principais conclusões feitas a partir dos experimentos em Pascal realizados neste capítulo são comentadas a seguir.

- **Estimativa de Comprimento**

  Rotinas: As métricas estimadoras de comprimento ($\hat{N}$ e $\hat{N}_j$) obtiveram uma adequação inaceitável na amostra de rotinas, apesar de bons coeficientes de correlação (em torno de 0,92). A métrica $\hat{N}_j$ mostrou-se sensivelmente melhor que $\hat{N}$. Também foi detectada uma tendência decrescente das relações $\hat{N}/N$, $\hat{N}_j/N$ e η_1/η com o comprimento das rotinas.

  Programas: Quando aplicados à amostra de programas, os estimadores de comprimento $\hat{N}$ e $\hat{N}_j$ mostraram-se válidos, devido à sua adequação muito boa (r, por exemplo, igual a 0,99). O estimador $\hat{N}_j$ foi apenas um pouco melhor que $\hat{N}$. E o melhor procedimento encontrado para estimativa de comprimento de programas foi a soma das estimativas de cada rotina do programa.

- **Nível de Linguagem:** Esta métrica apresentou uma média de 2,31 para a linguagem Pascal, mas teve uma variação muito grande (desvio padrão de 2,56) e um decréscimo com o comprimento da rotina.

- **Relacionamento Entre As Principais Métricas:** As métricas con-

sideradas extensivas (N, V e LDCe), como em outros estudos, obtiveram coeficientes de correlação altos entre si e baixos com relação às outras métricas E e NC.

- **Estimativa de Tempo de Programação**

Rotinas: Todas as métricas foram imprecisas na previsão de tempo de programação de rotinas. A métrica de esforço teve uma adequação muito ruim.

Programas: As métricas N, V, LDCe e NC foram consideradas válidas para a estimativa de tempo de programação, devido ao bom desempenho estatístico. O coeficiente de correlação entre estas e o tempo de programação variou de 0,93 a 0,97. V foi a melhor em três dos parâmetros, LDCe foi a única a ter o DRM, Prev(0,25) e o DQM dentro dos limites desejáveis e NC foi a que teve menor variação com o tipo de programa.

No próximo capítulo, serão discutidos os experimentos de aplicação de métricas à linguagem C.

7 - EXPERIMENTOS REALIZADOS COM C

Neste capítulo é feita a verificação da consistência interna das métricas na linguagem C. As principais análises efetuadas são: precisão dos estimadores de comprimento, tanto para rotinas como para programas, constância do conteúdo de informação, média e variação do nível de linguagem e relacionamento entre as principais métricas.

Nos experimentos em C não foi feita a consistência externa das métricas, devido à falta de informações disponíveis sobre o tempo de programação e sobre o número de erros encontrados para as rotinas analisadas.

7.1 - Características da Amostra Analisada

Nos experimentos com métricas aplicadas à linguagem C, foram analisados três conjuntos de programas, totalizando 33.554 linhas de código executável e caracterizados de acordo com a tabela 7.1.

Categoria	Nº de Programas	Nº de Rotinas	Linhas de Código
Software 1	—	385	9.288
Software 2	117	697	18.069
Alunos	89	232	6.197
Total	206	1.314	33.554

Tabela 7.1 - Distribuição dos Programas Analisados (C)

O primeiro conjunto de programas consiste em uma parcela bastante significativa de um software processador de textos. Este software está disponível comercialmente com o nome de InfoWord. Como as rotinas foram selecionadas aleatoriamente, não é apropriada a informação sobre o número de programas analisados.

O segundo conjunto envolve um software de gerenciamento de discos rígidos, com rotinas de otimização de velocidade, uso e fragmentação, entre outras. Foi desenvolvido por dois programadores, num total de 12 meses, e também está disponível comercialmente, sob o nome de Disk Manager. Foram considerados também como programas os módulos que, embora não fossem executáveis individualmente, incluiam rotinas e definições de dados e macros fortemente relacionados entre si.

O último conjunto engloba exercícios de alunos, desenvolvidos em um curso avançado de linguagem C. Ao todo foram analisadas até 17 versões diferentes para cada uma das seis especificações de programas propostas no curso. Este conjunto foi incluído neste experimento principalmente para verificação da constância do conteúdo de informação.

7.2 - Estimadores de Comprimento

Nesta seção, verifica-se a precisão das duas métricas estimadoras do comprimento de código: $\hat{N}$ (fórmula 3.3) e $\hat{N}_j$ (fórmula 4.1). A verificação é feita em duas diferentes abordagens. A primeira consiste na aplicação dessas métricas para uma amostra de rotinas. A segunda abordagem verifica a precisão dessas

métricas para uma amostra de programas.

A análise da precisão dos estimadores é feita com a aplicação de regressões simples lineares e não-lineares para as métricas $\hat{N}$ e $\hat{N}_j$, quando relacionadas aos comprimentos reais observados. As regressões lineares obtiveram os melhores resultados. São também feitas análises da variação da precisão dos estimadores quanto ao tipo e comprimento de rotinas ou programas.

7.2.1 - Estimadores de Comprimento Aplicados a Rotinas

Nesta seção, tenta-se avaliar a precisão dos estimadores de comprimento $\hat{N}$ e $\hat{N}_j$, quando aplicados a rotinas.

Estimador $\hat{N}$:

Os valores dos principais parâmetros estatísticos calculados numa regressão linear de $\hat{N}$ com rotinas de cada categoria são mostrados na tabela 7.2. No cálculo total, preferiu-se incluir apenas as rotinas desenvolvidas profissionalmente, para manter uma maior representatividade.

Categoria	r	b	DRM	Prev(0,25)	DQM
Software 1	0,925	0,707	0,329	45,6%	0,473
Software 2	0,931	0,803	0,367	39,4%	0,460
Alunos	0,892	0,883	0,434	36,6%	0,424
Total (1 e 2)	0,928	0,777	0,367	40,0%	0,476

Tabela 7.2 - Adequação de $\hat{N}$ na Estimativa de N (Rotinas em C)

Analisando a tabela 7.2, verifica-se que a métrica $\hat{N}$ manteve-se ligeiramente melhor para a categoria de software 1 e pior para as rotinas de alunos, talvez devido às impurezas freqüentes em programas dessa natureza. No geral, as estatísticas, com exceção do coeficiente de correlação, não são boas e estão fora dos limites de validação propostos. O coeficiente angular da reta de regressão tem um desvio de -22,3% em relação ao teórico, e o fato de ser menor que um (1) implica que $\hat{N}$ em média foi maior que N. Isso provavelmente seja explicado pela grande concisão da linguagem C, em relação a outras cujos coeficientes foram próximos de um.

Estimador $\hat{N}_j$:

Passa-se agora a uma análise sobre a precisão do outro estimador de comprimento - $\hat{N}_j$, que apresentou resultados mais precisos, como revela a tabela 7.3.

Categoria	r	b	DRM	Prev(0,25)	DQM
Software 1	0,927	0,963	0,243	60,9%	0,467
Software 2	0,933	1,080	0,278	53,4%	0,453
Alunos	0,892	1,224	0,362	40,9%	0,423
Total 1 e 2	0,930	1,049	0,274	55,2%	0,467

Tabela 7.3 - Adequação de $\hat{N}_j$ na Estimativa de N (Rotinas em C)

Este estimador apresentou resultados próximos dos considerados desejáveis, principalmente quanto ao parâmetro DRM. Em comparação com $\hat{N}$, $\hat{N}_j$ foi superior em todos os parâmetros e ca-

tegorias, o que reforça a suposição de que é o melhor estimador. Outro ponto a favor de $\hat{N}_j$ é o fato de que o coeficiente angular médio foi de 1,049, bem próximo do proposto por Jensen (erro de 4,9%). A figura 7.1 ilustra o relacionamento entre $\hat{N}_j$ e $\hat{N}$ para as rotinas dos software 1 e 2.

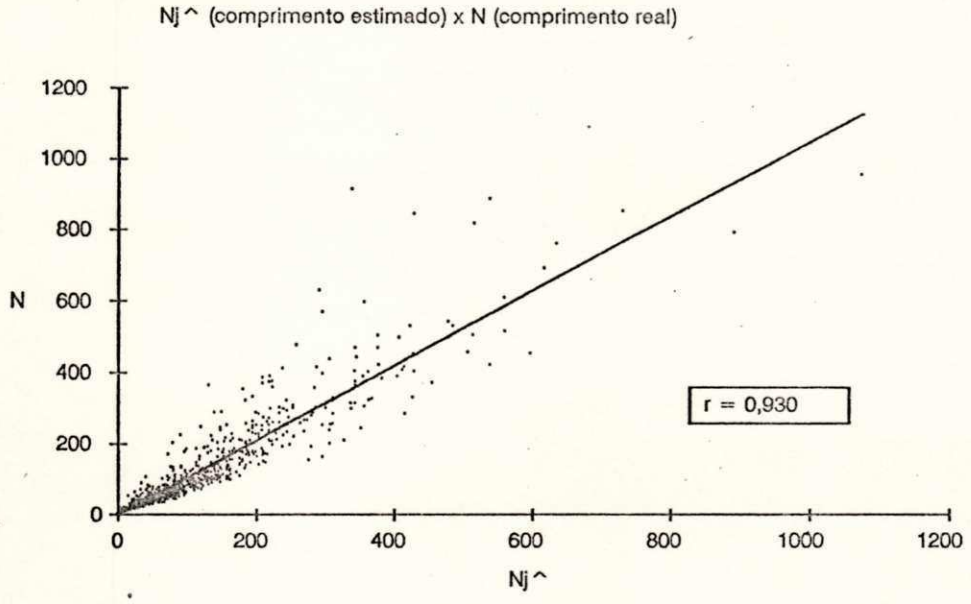


Figura 7.1 - Relacionamento entre $\hat{N}_j$ e N
(1082 rotinas em C)

7.2.2 - Estimadores de Comprimento Aplicados a Programas

Nesta seção, tenta-se descobrir a precisão dos estimadores de comprimento da Ciência de Software, quando aplicados em programas das categorias software 2 e alunos. Os comprimentos mínimo, médio e máximo dos programas nas duas categorias ocorreram de acordo com a tabela 7.4.

Categoria	Nº de Programas	N mínimo	N médio	N máximo
Software 2	117	5	634,8	4.700
Alunos	89	76	316,7	940
Total	206	5	497,4	4.700

Tabela 7.4 - Comprimentos dos Programas Analisados (C)

Ao todo, foram testados quatro estimadores: os de Halstead e Jensen, cada um calculado de duas maneiras distintas. Na primeira, os estimadores ($\hat{N}$ e $\hat{N}_j$) são calculados para cada uma das rotinas do programa e então são somados. Na segunda maneira, os estimadores ($\hat{N}_{prog}$ e $\hat{N}_{j\text{-}prog}$) são calculados tratando-se o programa como um todo e ignorando-se os limites das rotinas. A tabela 7.5 exibe os parâmetros estatísticos para cada métrica e categoria. Destacam-se, em negrito, os valores que estão dentro dos limites desejados.

Categoria/ Estimador	r	b	DRM	Prev(0,25)	DQM
Software 2:					
$\hat{N}$	0,988	0,758	**0,232**	65,8%	**0,201**
$\hat{N}j$	0,988	1,045	**0,189**	74,3%	**0,202**
$\hat{N}prog$	0,949	1,012	0,410	45,3%	0,420
$\hat{N}j$-prog	0,949	1,260	0,312	55,5%	0,419
Alunos:					
$\hat{N}$	0,904	0,793	**0,182**	70,1%	0,261
$\hat{N}j$	0,910	1,134	**0,176**	73,0%	**0,253**
$\hat{N}prog$	0,908	1,030	0,226	67,4%	0,256
$\hat{N}j$-prog	0,906	1,368	**0,204**	73,0%	0,257

Tabela 7.5 - Adequação de $\hat{N}$, $\hat{N}_j$, $\hat{N}_{prog}$ e $\hat{N}_{j\text{-}prog}$ (Programas em C)

De acordo com os parâmetros DRM e DQM, as métricas medidas por rotinas se mostraram válidas, por apresentarem valores normalmente abaixo do limite de 0,25 desejado. Os estimadores medidos para todo o programa se mostraram significativamente piores do que os correspondentes calculados para cada rotina e depois somados. Neste experimento, a métrica $\hat{N}_j$ também esteve melhor que $\hat{N}$.

O erro padrão da estimativa (não mostrado na tabela) foi de aproximadamente 112 para as métricas $\hat{N}$ e $\hat{N}_j$, quando todos os programas são considerados. Com isto, o intervalo de confiança de 80% para estimativas com estas métricas foi:

Intervalo de Confiança: $\hat{N} \pm 144$

Este resultado pode ser considerado bom, tendo em vista a média de N dos programas ter sido igual a 497. A figura 7.2 exibe o relacionamento existente entre a métrica $\hat{N}_j$ (medida por rotinas) e N para 117 programas da categoria do software 2.

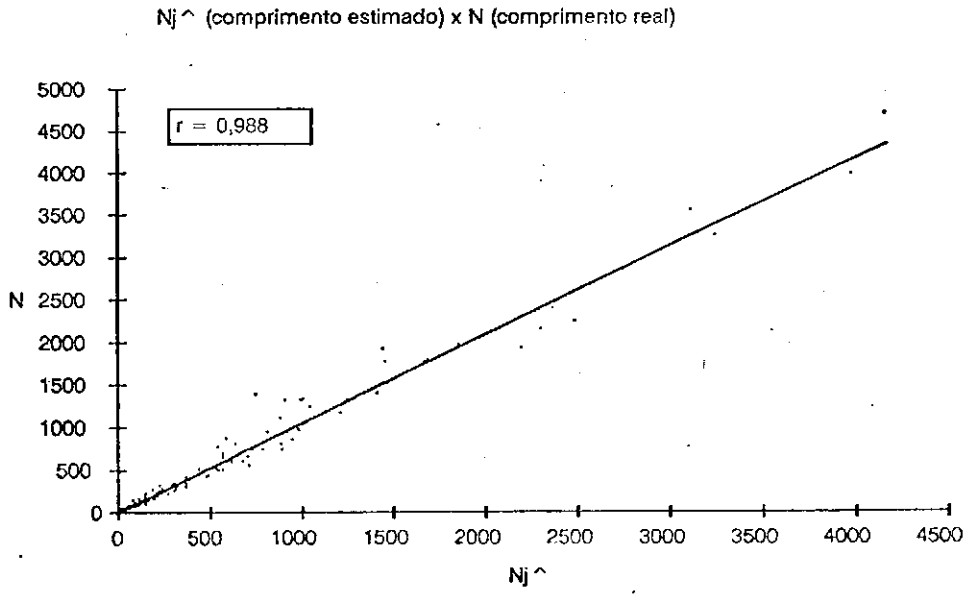


Figura 7.2 - Relacionamento entre $\hat{N}_j$ e N (117 programas em C)

7.2.3 - Variação da Precisão dos Estimadores com o Tamanho de Rotinas e Programas

Esta seção verifica a hipótese formulada em estudos com outras linguagens, na qual as relações $\hat{N}/N$, $\hat{N}_j/N$ e η_1/η tendem a decrescer com o aumento do tamanho de rotinas e programas.

Rotinas:

A tabela 7.6 mostra a queda destas relações na linguagem C, quando rotinas são analisadas.

Rotinas	Variação de N	Média de $\hat{N}$ / N	Média de $\hat{N}_j$ / N	Média de η_1 / η
1 - 220	1 - 17	1,70	1,03	0,70
221 - 440	18 - 34	1,94	1,25	0,63
441 - 660	34 - 54	1,81	1,21	0,63
661 - 880	54 - 95	1,61	1,11	0,62
881 - 1100	95 - 183	1,43	1,02	0,58
1101 - 1314	183 - 957	1,15	0,85	0,50

Tabela 7.6 - Variação das Relações $\hat{N}/N$, $\hat{N}_j/N$ e η_1/η (1.314 Rotinas em C)

Programas:

Quando se analisam essas relações para programas, nota-se, na tabela 7.7, uma estabilização da queda das relações quando N é maior ou igual a 177. Na figura 7.3, visualiza-se a variação da relação $\hat{N}/N$ para os programas ordenadas pelo comprimento.

Rotinas	Variação de N	Média de $\hat{N}$ / N	Média de $\hat{N}_j$ / N	Média de η_1/η
1 - 70	5 - 176	1,60	1,06	0,63
71 - 140	177 - 466	1,30	0,90	0,55
141 - 206	471 - 4700	1,29	1,92	0,42

Tabela 7.7 - Variação de $\hat{N}/N$, $\hat{N}_j/N$ e η_1/η
(206 Programas em C)

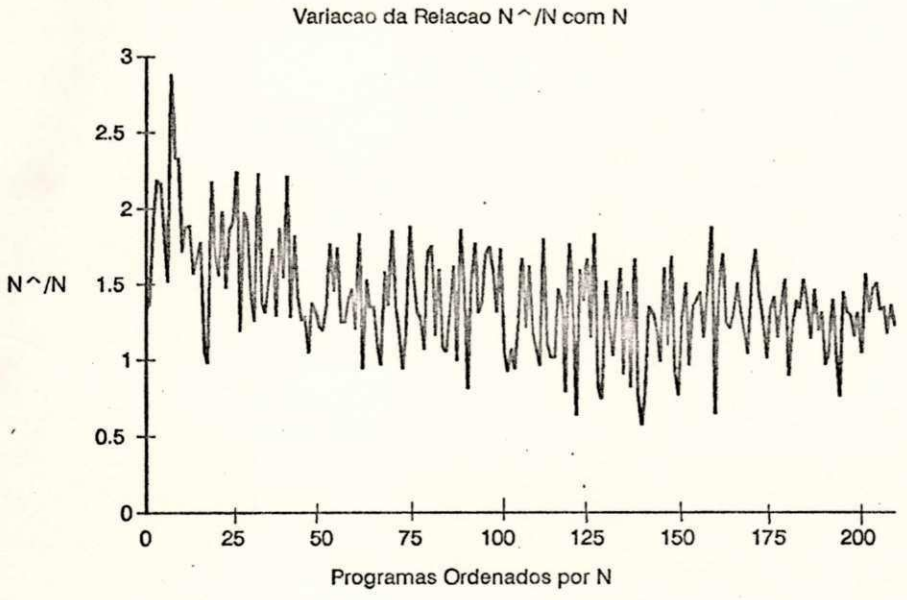


Figura 7.3 - Variação da relação $\hat{N}/N$ para 206 programas ordenados pelo comprimento (C)

7.3 - Dificuldade

Na seção 6.2.2, comentou-se que a importância dessa métrica é pequena, devido à grande imprecisão da métrica de esforço na estimativa de tempo de programação. De qualquer forma, repetiu-se aqui o experimento de avaliação das maiores influências na métrica de dificuldade.

Neste experimento, calculou-se, para todas as rotinas, as

correlações estatísticas entre a dificuldade e seus dois componentes: η_1 e N_2/η_2. A correlação obtida para η_1 (0,872) foi um pouco maior do que a obtida para N_2/η_2 (0,803). Isso sugere que η_1 é o fator mais influente no valor da dificuldade.

7.4 - Conteúdo de Informação

Essa métrica foi proposta por Halstead para medir a função de um algoritmo, devendo-se manter praticamente constante para versões equivalentes de um mesmo programa.

Para testar esta hipótese, foram analisados seis programas pequenos, cada um com até 17 versões elaboradas por estudantes. A tabela 7.8 mostra as variações do comprimento e do conteúdo de informação (I), apresentadas pelas versões de cada um dos programas. Também são mostrados os desvios padrão e os coeficientes de dispersão (relação entre desvio padrão e a média).

Programa	Variação de N	Variação de I	Desvio Padrão de I	Dispersão de I
1	76 - 310	22 - 52	11,4	33 %
2	91 - 278	24 - 56	8,9	30 %
3	168 - 380	19 - 98	21,0	33 %
4	219 - 503	38 - 129	26,5	20 %
5	284 - 527	21 - 106	22,4	34 %
6	458 - 940	103 - 198	32,0	22 %

Tabela 7.8 - Variação de I Para Seis Programas

Pelos dados da tabela, verifica-se que o conteúdo de informação mostrou-se totalmente inadequado para a linguagem C, já que esteve longe de ser considerado uma constante. Sua vari-

ação é praticamente da mesma ordem da variação de N. No entanto, deve-se lembrar que a amostra provém de programas de alunos e também que os programas analisados não são totalmente equivalentes entre si.

7.5 - Nível de Linguagem

Esta seção verifica a hipótese de que esta métrica é praticamente constante para diferentes rotinas numa mesma linguagem e calcula o valor médio desta métrica para a linguagem C.

Da mesma maneira que na linguagem Pascal, neste experimento retiraram-se as rotinas de nível de linguagem maior que 15, por serem suspeitas de apresentar impurezas (ver seção 3.1.11). Apenas duas rotinas (de atribuição inicial de variáveis) foram excluídas dos cálculos. A tabela 7.9 mostra as médias para todas as rotinas divididas em categorias pelo comprimento.

Rotinas	NL mínimo	NL médio	NL máximo	Desvio Padrão
1 - 220	0	3,06	11,23	1,71
221 - 440	0,33	2,49	14,16	2,38
441 - 660	0,23	1,62	9,38	1,38
661 - 880	0,08	1,20	7,20	1,12
880 - 1100	0,11	0,90	5,54	0,68
1100 - 1312	0,10	0,94	10,07	1,00
todas	0	1,70	14,16	1,69

Tabela 7.9 - Variação do Nível da Linguagem C
(1.312 Rotinas)

Pelo valor de seu nível médio (1,70), a linguagem C se situa um pouco acima de PL/I e Fortran na tabela 4.2, e abaixo do

valor de 2,36 encontrado para Pascal neste trabalho. Ainda que o seu valor médio não surpreenda e esteja dentro do esperado, o nível de linguagem apresentou um desvio padrão muito elevado, atingindo 1,69 (99,4% da média). Esta variação também foi muito elevada mesmo se for calculado o nível de linguagem para programas (através da média dos níveis de linguagem das rotinas que compõem o programa).

Verifica-se também, na tabela, uma nítida queda do nível de linguagem médio com o aumento do tamanho da rotina.

Essas duas características — grande variabilidade e dependência do comprimento da rotina — comprometem seriamente o uso da métrica nível de linguagem como uma constante característica da linguagem.

7.6 - Relacionamento Entre as Principais Métricas

Esta seção objetiva verificar o relacionamento existente entre as principais métricas para a medição de quantidade e complexidade de código: N (comprimento), V (volume), E (esforço), LDCe (linhas de código executável) e NC (número ciclomático). O estudo do relacionamento é feito calculando-se os coeficientes de correlação linear entre cada métrica e as outras. A tabela 7.10 exibe os resultados obtidos para todas as rotinas.

métricas	N	V	E	LDCe	NC
N	-	0,995	0,847	0,933	0,842
V	0,995	-	0,872	0,928	0,824
E	0,847	0,872	-	0,785	0,696
LDCe	0,933	0,928	0,785	-	0,879
NC	0,842	0,824	0,696	0,879	-

Tabela 7.10 - Correlações Entre Métricas (1314 Rotinas em C)

Observando a tabela, verifica-se que as métricas N, V e LDCe têm coeficientes de correlação relativamente altos entre si (0,928 a 0,995), o que indica que medem aspectos semelhantes de código. A correlação entre E e NC foi muito baixa (0,696); isto sugere que elas medem o código de maneira bastante diferente. Quando aplicados a programas, em vez de rotinas, as correlações da tabela 7.10 crescem significamente, variando de 0,896 a 0,997. Na figura 7.4 visualiza-se os relacionamentos entre N e V e entre N e LDCe, para 1.314 rotinas.

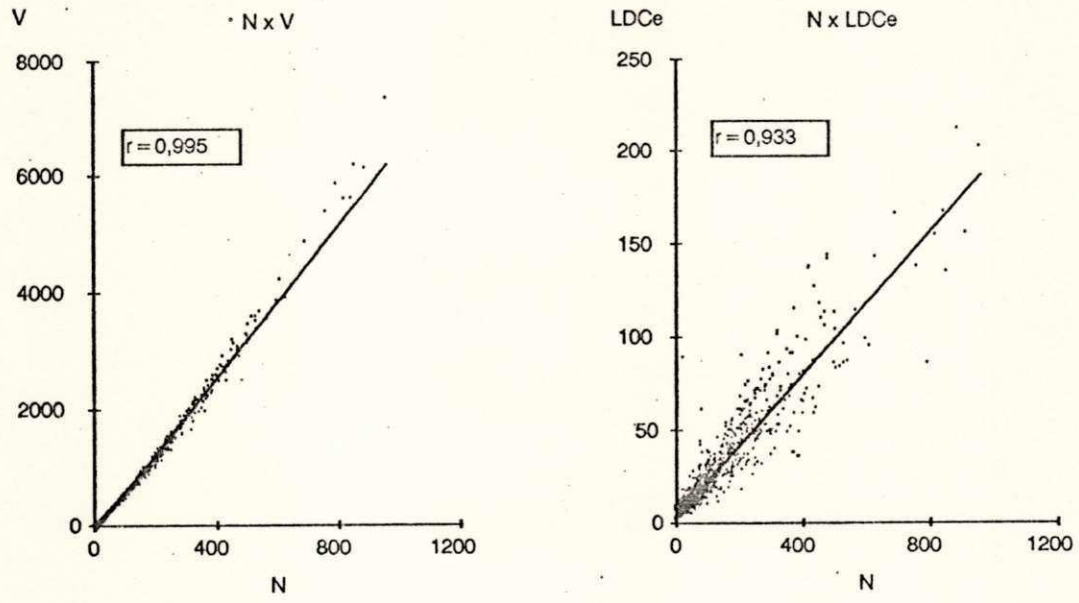


Figura 7.4 - Relacionamento entre N e V, e N e LDCe (1.314 rotinas em C)

7.7 - Conclusões

As principais conclusões tiradas a partir dos experimentos realizados com C, descritos neste capítulo são comentadas a seguir.

- **Estimativa de Comprimento**

  Rotinas: As métricas estimadoras de comprimento ($\hat{N}$ e $\hat{N}_j$) obtiveram desempenho estatístico inaceitável na amostra de rotinas, apesar do bom coeficiente de correlação (0,93 para as rotinas desenvolvidas por profissionais). A métrica $\hat{N}_j$ mostrou-se sensivelmente melhor que $\hat{N}$. Também foi detectado uma tendência decrescente das relações $\hat{N}/N$, $\hat{N}_j/N$ e η_1/η com o comprimento das rotinas.

  Programas: Quando aplicados à amostra de programas, os estimadores de comprimento $\hat{N}$ e $\hat{N}_j$ mostraram-se válidos, com uma adequação muito boa (r, por exemplo, igual a 0,99 para programas desenvolvidos por profissionais). O estimador $\hat{N}_j$ foi novamente melhor que $\hat{N}$. Não houve tendência de queda das relações $\hat{N}/N$, $\hat{N}_j/N$ e η_1/η para programas com comprimentos maiores que 177. E o melhor procedimento encontrado para estimativa de comprimento de programas foi a soma das estimativas de cada rotina do programa.

- **Conteúdo de Informação:** Esta métrica mostrou-se totalmente inconsistente para a amostra de programas analisados.

- **Nível de Linguagem:** Esta métrica apresentou uma média de 1,70

para a linguagem C, valor considerado razoável. Mas teve uma variação muito grande (desvio padrão de 1,69) e um decréscimo com o comprimento da rotina.

- **Relacionamento Entre As Principais Métricas:** As métricas consideradas extensivas N, V e LDCe, como em outros estudos, obtiveram coeficientes de correlação altos entre si e relativamente baixos com relação às outras métricas E e NC. Todos os coeficientes de correlação crescem quando programas, em vez de rotinas, são analisados.

8 - SUMÁRIO, CONCLUSÕES E RECOMENDAÇÕES

8.1 - Sumário

O uso fundamentado de métricas (padrões de medição) é essencial para a execução de projetos de forma mensurável cujo desenvolvimento possa ser previsto, monitorado e aprimorado.

As métricas de software definem uma maneira padronizada de medir algum atributo do processo de desenvolvimento, por exemplo, custo, tempo, tamanho e complexidade do software. Embora as métricas possam ser aplicadas às diversas fases do ciclo de desenvolvimento do software, a grande maioria das métricas encontradas na literatura especializada se concentra na fase de codificação.

Entre as diversas métricas para a fase de codificação, se destacam o **número ciclomático** [MCCA76] e as métricas da **Ciência de Software** [HALS77]. Numerosos estudos de validação destas métricas foram publicados para as linguagens mais antigas como Fortran, Cobol, APL e PL/I, com resultados relativamente bons. Mas poucos estudos de validação dessas métricas para a linguagem Pascal e nenhum para a linguagem C foram encontrados. Por essas razões, realizou-se neste trabalho um estudo de validação dessas métricas quando aplicadas às linguagens Pascal e C.

Para efeito de comparação, foram consideradas também duas métricas simples e populares: **número de linhas de código** e **número de rotinas**.

A adequação de modelos baseados em métricas normalmente é avaliada por estudos empíricos, onde se correlacionam estatis-

ticamente as métricas entre si e com outros dados do desenvolvimento, como tempo de programação, número de erros, número de mudanças, etc. Se uma boa correlação for encontrada entre, por exemplo, o tempo de programação e uma métrica, esta métrica provavelmente possa ser usada como estimadora de tempo ou como avaliadora da quantidade de código.

Na grande maioria dos estudos de avaliação das métricas, os testes de validação basearam-se exclusivamente no coeficiente de correlação linear. Este coeficiente, apesar de muito útil, não é suficiente, por si só, para a validação das métricas [MORA78; FENI79; CONT86]. Por este motivo, neste trabalho foram escolhidos, além do coeficiente de correlação linear, os seguintes parâmetros estatísticos: desvio relativo médio - DRM; desvio quadrático médio - DQM; índice de acerto de prognósticos com erro menor do que 25% - Prev(0,25) e erro padrão de estimativa.

Nos experimentos realizados neste trabalho, a amostra de programas foi constituída de 356 rotinas em Pascal (18.420 linhas de código executável) e 1314 rotinas em C (33.554 linhas de código executável). As rotinas fazem parte de software disponível comercialmente e de programas resultantes de tarefas acadêmicas. Para fazer o levantamento das métricas foi desenvolvida uma ferramenta, denominada de Quantum. Esta ferramenta, desenvolvida nas linguagens C, LEX e YACC, tem duas versões: uma para cálculo de métricas de códigos em Pascal e outra para códigos em C.

Além do número de linhas de código e rotinas, as princi-

pais métricas estudadas são listadas a seguir: (Outras métricas serão explicadas à medida que forem citadas.)

- **Comprimento (N)**: consiste na contagem de tokens (símbolos) do programa (fórmula 3.2);

- **Volume (V)**: o comprimento multiplicado por um fator que leva em conta a riqueza do vocabulário de operandos e operadores (fórmula 3.4);

- **Esforço (E)**: métrica da Ciência de Software proposta para estimar o esforço de programação (fórmula 3.10);

- **Número ciclomático (NC)**: mede a complexidade do fluxo de controle. É igual ao número de decisões no programa mais um (fórmula 3.12).

Uma vez levantadas as métricas, procedeu-se ao experimento de validação, realizado em duas etapas, verificando as consistências interna e externa.

8.1.1 - Consistência Interna das Métricas

Nesta etapa verificou-se o comportamento das métricas estudadas e as suposições nas quais se baseiam. Isto foi feito estudando diversos relacionamentos estatísticos existentes entre as métricas. Os resultados desta etapa, apresentados nos capítulos seis e sete (para as linguagens Pascal e C, respectivamente), foram bastante semelhantes, com exceção dos coeficientes das curvas de regressão. As principais conclusões chega-

das nesta etapa são comentadas a seguir.

Métricas para Estimativa de Comprimento - $\hat{N}$ e $\hat{N}_j$

Neste experimento foram analisados os estimadores de comprimento $\hat{N}$ (fórmula 3.3) e $\hat{N}_j$ (fórmula 4.1), para amostras constituídas de rotinas e programas.

- **Rotinas:** As duas métricas de estimativa de comprimento estudadas tiveram uma adequação inaceitável para as amostras de rotinas, apesar dos bons coeficientes de correlação (de 0,89 a 0,96). Além disso foi detectada uma dependência da precisão dos estimadores com o comprimento da rotina.
- **Programas:** Quando aplicados à amostra de programas os estimadores de comprimento mostraram-se válidos, devido ao seu desempenho estatístico muito bom (r, por exemplo, igual a 0,99, exceto para programas de alunos). O estimador $\hat{N}_j$ foi um pouco melhor que $\hat{N}$. O melhor procedimento encontrado para estimativa de comprimento de programas foi a soma das estimativas de cada rotina do programa. A dependência da precisão dos estimadores com o tamanho do programa foi pequena.

Conteúdo de Informação - I

De acordo com esta métrica da Ciência de Software (fórmula 3.8), diferentes versões de um programa com a mesma função deveriam apresentar o mesmo valor para I. Os experimentos rea-

lizados com programas de estudantes na linguagem C, no entanto, invalidaram esta métrica pela sua grande variação para programas equivalentes.

Nível de Linguagem - NL

Esta métrica da Ciência de Software (fórmula 3.9) é uma maneira de se definir quantitativamente o nível de uma linguagem. Experimentos publicados mostraram que para grandes amostras de rotinas em várias linguagens, os valores médios desta métrica estiveram intuitivamente coerentes.

No nosso estudo, a linguagem Pascal apresentou um nível elevado, ficando só abaixo de APL. A linguagem C apresentou um nível cerca de 25% menor. Embora os valores médios encontrados tenham sido razoáveis, a grande variabilidade desta métrica e o decréscimo com o comprimento da rotina comprometem o seu uso.

Relacionamento Entre As Principais Métricas

Para uma melhor compreensão, foram correlacionadas entre si diversas métricas utilizadas como avaliadoras de quantidade e complexidade de código: N, V, E, LDCe e NC.

Para rotinas, em ambas as linguagens, foi verificado que as métricas N, V e LDCe apresentaram correlações relativamente altas entre si. Desta forma, pode-se afirmar que medem aspectos semelhantes do código. Suas correlações com as métricas E e NC, como também a própria correlação entre as duas, foram bem menores. Isto indica que as métricas E e NC são fundamentalmente

diferentes das demais e entre si.

8.1.2 - Consistência Externa das Métricas

Nesta etapa foi feito um estudo de correlação entre as métricas de código e o tempo real de programação (incluindo projeto e implementação). Apenas códigos em Pascal foram considerados nesta etapa, já que os dados relativos ao tempo de programação dos códigos em C não eram disponíveis.

Foram empregadas regressões lineares e não-lineares, em duas amostras diferentes. A primeira amostra é formada por rotinas e a segunda por programas.

- **Rotinas:** Todas as métricas estudadas foram imprecisas na previsão de tempo de programação de rotinas. A métrica de esforço teve uma adequação muito ruim.
- **Programas:** As métricas N, V, LDCe e NC foram consideradas válidas para a estimativa de tempo de programação, devido ao bom desempenho estatístico. O coeficiente de correlação entre estas e o tempo de programação variou de 0,93 a 0,97. As métricas N, V e NC têm uma grande vantagem sobre LDCe, por serem independentes da linguagem e de apresentarem em menor grau os problemas desta última métrica, citados na seção 2.3.1.

8.2 - Conclusão e Considerações Finais

A partir dos resultados apresentados, pode-se afirmar que

algumas das métricas de código estudadas mostraram-se válidas na estimativa e avaliação de atributos de programas em C e Pascal. No entanto, quando se trata de rotinas, nenhuma das métricas foi considerada aceitável, dentro dos critérios estabelecidos. (Há uma explicação provável para isto: quando as estimativas são feitas para programas, os erros positivos de estimativa de algumas rotinas tendem a compensar os erros negativos de outras.) De qualquer forma, até agora, nenhuma outra técnica de estimativa publicada é comprovadamente melhor. Muitos estudos neste sentido serão necessários para uma utilização mais segura destas métricas. O quadro 8.1 mostra um resumo do estudo de validação das métricas para as duas linguagens.

Aplicação	Métricas	
	Válidas	Inválidas
Estimativa de Comprimento de Rotinas	nenhuma	$\hat{N}$, $\hat{N}_j$
Estimativa de Comprimento de Programas	$\hat{N}$, $\hat{N}_j$	$\hat{N}_{prog}$, $\hat{N}_j$-prog
Estimativa de Tempo de Programação de Rotinas	nenhuma	N, V, E, LDCe, NC
Estimativa de Tempo de Programação de Programas	N, V, LDCe, NC	E, NR
Outras	nenhuma	I, NL

Quadro 8.1 - Resumo do Estudo de Validação das Métricas (Pascal e C)

É interessante observar que apesar de as métricas serem

fundamentalmente diferentes, os resultados obtidos por algumas delas para amostras de programas foram semelhantes. A alta correlação entre essas métricas para amostras de programas indica que elas apresentam resultados semelhantes e que não são ortogonais, como se poderia esperar.

Ao serem comparados os resultados obtidos para Pascal com os obtidos para C, verifica-se que as diferenças não são significativas, com exceção dos coeficientes das curvas de regressões dos diversos modelos testados. Os resultados obtidos na fase de consistência interna de métricas em Pascal foram confirmados também nos experimentos em C. Isso dá uma maior segurança a esses resultados e sugere que as conclusões a que se chegou na fase de consistência externa de Pascal se mantenham verdadeiras também para a linguagem C.

A variação dos coeficientes das curvas de regressão entre as duas linguagens implica que um modelo adequado para um ambiente de programação só deve ser utilizado em ambientes semelhantes. Até mesmo mudanças menores no ambiente (como tipo de software, programadores, metodologias, etc.) podem provocar variações nesses coeficientes, exigindo ajustes no modelo. Nos experimentos realizados para uma mesma linguagem, os diversos coeficientes da curva de regressão variaram até 25%, ao se considerar diferentes categorias (com diferentes tipos e tamanhos do software e equipes de produção).

Dessa forma, os modelos baseados em métricas apresentam melhor precisão à proporção que são aplicados a ambientes de programação mais bem delineados, ou seja, com um grupo de produção, tipo de software, linguagem de programação e sistema

operacional definidos. A coleta e análise de métricas a longo prazo possibilitam conhecer coeficientes de modelos para avaliação de produtividade e qualidade para o ambiente. A precisão deste modelo poderá ser melhorada com o tempo.

8.3 - Recomendações Para Trabalhos Futuros

8.3.1 - Metodologia para Coleta de Dados do Desenvolvimento

A maior dificuldade encontrada neste trabalho foi a falta de dados relacionados ao próprio desenvolvimento do software como tempo de programação de cada fase, número e tempo de correção de erros do software, etc. A adoção de uma metodologia de coleta de informações deste tipo no ambiente de programação toma pouco tempo dos programadores e possibilita o recolhimento de dados muito valiosos. O estudo destes dados permite a elaboração de técnicas eficazes para a estimativa e controle da qualidade e produtividade do desenvolvimento de software. É apresentada aqui uma metodologia simples para coleta dos dados do desenvolvimento, baseada em formulários preenchíveis pela equipe de produção.

Na implantação de um procedimento de coleta de dados deve-se considerar os seguintes pontos [BASI84]:

- A equipe responsável pelo fornecimento dos dados deve estar motivada, bem informada e consciente da importância da coleta e da precisão dos dados.
- A coleta deve ser facilitada ao máximo, através do uso de

formulários simples e curtos e de ferramentas que automatizem as tarefas relacionadas à coleta.

- A equipe de produção deve ser esclarecida de que os dados coletados não serão utilizados para avaliação individual.

A referência [GRAD86] relata a experiência da Hewlett Packard na implantação de um plano de coleta de dados de métricas e apresenta alguns dos formulários adotados. Um desses formulários, aqui adaptado, é preenchido para cada software desenvolvido, ao ser lançado para uso. São coletados dados sobre as características gerais, tamanho e erros do software e o tempo e custo do seu desenvolvimento.

Características Gerais:

Este item recolhe as informações gerais sobre o projeto, tais como:

- tipo do software;
- metodologia de desenvolvimento;
- características da equipe de produção;
- linguagem utilizada;
- outras informações pertinentes à própria organização.

Tempo/Custo:

O formulário na figura 8.1 permite coletar o tempo decorrido e o gasto com pessoal (seja em unidade monetária ou em homens-mes), para cada fase do desenvolvimento. Para se dispor

dessas informações é suficiente que a equipe de produção preencha formulários semanais, estimando o tempo gasto na semana em cada fase. Deve-se fazer o levantamento de forma consistente, definindo se este tempo inclui atividades como reuniões, redação de documentação, interrupções, etc.

Para um levantamento mais completo pode-se coletar estas informações para cada módulo (ou até mesmo rotina) trabalhado.

Fases	Tempo	Custo
Especificação		
Projeto		
Codificação		
Testes		
TOTAL		

Figura 8.1 - Formulário para coleta de tempo/custo

Tamanho:

Este item coleta informações sobre o tamanho do software através do uso de métricas de código. Tendo em vista os resultados obtidos neste trabalho, recomenda-se o uso das métricas: comprimento (N), volume (V), LDCe (linhas de código executável), NR (número de rotinas) e NC (número ciclomático). Além disso, seria muito útil uma informação aproximada sobre a quantidade de código reutilizado presente no software (figura 8.2).

Métrica	Valor
Comprimento (N)	
Volume (V)	
Linhas de Código (LDCe)	
Número de Rotinas (NR)	
Número Ciclomático (NC)	
Código Reutilizado - %	

Figura 8.2 - Formulário para coleta de tamanho

Essas informações são levantadas automaticamente com o uso de ferramentas e podem ser coletadas periodicamente para acompanhamento da produção.

Com as informações de tamanho e de tempo/custo, é possível entre outras coisas:

- avaliar a produtividade;
- acompanhar o progresso do desenvolvimento;
- fazer estimativas (por similaridade de software).

Erros:

O formulário na figura 8.3 permite a coleta de dados sobre os erros no software descobertos e corrigidos antes de seu lançamento para uso. Inicialmente, conta-se apenas o número de erros introduzidos, encontrados e corrigidos em cada etapa do ciclo de vida, sem considerar a gravidade dos erros.

Fases	Número de Erros		
	Introduzidos	Encontrados	Corrigidos
Especificação			
Projeto			
Codificação			
Testes			
TOTAL			

Figura 8.3 - Formulário para coleta de informações sobre erros

A coleta mais detalhada de dados sobre os erros permite que se tenha um controle mais eficiente da qualidade do software e do processo de desenvolvimento. Este controle pode ser obtido coletando as seguintes informações para cada erro detectado (inclusive após o lançamento do software) [GRAD86]:

- gravidade do erro do ponto de vista do usuário;
- descrição do erro;
- modo como foi descoberto;
- sintomas do problema;
- prioridade de correção;
- tempo de correção;
- maneira como foi corrigido;
- localização do erro;
- fases em que foi introduzido, encontrado e corrigido.

Com estes dados torna-se possível, por exemplo:

- avaliar a qualidade do software;

- avaliar a qualidade do processo de desenvolvimento, em cada uma de suas fases;
- aprimorar as técnicas de depuração de programas;
- prever o número de erros de uma rotina;
- identificar rotinas problemáticas (útil para manutenção preventiva);
- avaliar os custos causados pela má qualidade.

8.3.2 - Variações na Metodologia de Contagem

As métricas da Ciência de Software e o número ciclomático ainda estão em evolução, pois não há acordo sobre a metodologia de contagem mais adequada para estas métricas. Um estudo que avaliasse a precisão dessas métricas quando diferentes metodologias de contagem são utilizadas é bastante oportuno. Estudos desta natureza são bastante raros na literatura especializada. A seguir, são apresentadas algumas das possíveis variações na metodologia de contagem das métricas da Ciência de Software e do número ciclomático.

Ciência de Software:

Na contagem dos operadores e operandos, podem ser feitas as seguintes modificações em relação à metodologia adotada neste trabalho:

- Considerar também as instruções de declaração (incluindo macros);
- Não considerar as instruções de escrita em tela;

- Ponderar cada operador de acordo com a sua complexidade (por exemplo, o nível de aninhamento [RAMA88]);

Número Ciclomático:

Com relação a contagem do número ciclomático, uma variação foi proposta por [MYER77]. Nela, o número ciclomático é computado como sendo um intervalo, com o extremo inferior correspondente ao número de decisões sem considerar os operadores booleanos e o extremo superior correspondente ao número de decisões, incluindo os operadores booleanos. O raciocínio do autor é que um operador booleano contribui para a complexidade em menor grau do que uma instrução de decisão propriamente dita. Por isso, é importante saber o quanto do número ciclomático de um programa é conseqüência do número de operadores booleanos.

APÊNDICE A - MÉTODOS DE ANÁLISE ESTATÍSTICA

O objetivo deste apêndice é descrever sucintamente os termos, conceitos e técnicas estatísticas utilizados neste trabalho. A abrangência deste apêndice é, naturamente, bastante limitada e se resume praticamente aos Métodos de Análise de Correlação e Regressão aqui aplicados. Para um maior aprofundamento nos conceitos e métodos estatísticos o leitor deve consultar conceituados livros de Estatística como [EZEK59; WONN72; UNDE54 e WALP79].

Neste apêndice será seguido um exemplo referente a um experimento hipotético, cujo cálculo dos parâmetros estatísticos poderá ser acompanhado pelo leitor. O experimento se constitui na elaboração de um programa por 10 programadores diferentes, quando se observaram os seguintes tempos de duração (em minutos) e métrica de esforço E de Halstead (em d.m.e. - discriminações mentais elementares).

Programador	Tempo (min)	Esforço
1	10	13500
2	14	12800
3	17	14000
4	23	18600
5	24	29400
6	26	30200
7	26	29600
8	30	33000
9	35	36800
10	45	37000

Neste apêndice a seguinte notação será utilizada:

X : corresponde aos diversos valores assumidos pela variável X.

$\bar{X}$: corresponde a média aritmética da variável X.

s_x : corresponde ao desvio padrão de X.

n : número de observações (dados) do experimento.

Y_{est}: estimativa de Y, a partir de um valor de X.

A utilização da Estatística neste trabalho se concentra basicamente nos estudos de correlação entre duas variáveis. No exemplo, as variáveis são o tempo levado para escrever um programa e a métrica de esforço de Halstead.

Na impossibilidade de se determinar algebricamente a relação entre duas variáveis, como as citadas, normalmente se recorre à aplicação de experimentos estatísticos cujos resultados podem indicar a que grau uma variável está relacionada a outra, se pode ser usada como estimadora, e a que nível de confiança se pode fazer uma estimativa.

Antes porém, de se estudar como avaliar a correlação entre duas ou mais variáveis, convém descrever alguns conceitos básicos pertinentes a séries estatísticas.

A.1 - Séries Estatísticas

Na análise de experimentos estatísticos, denomina-se de **série estatística** o conjunto de valores assumidos por uma determinada **variável** em um experimento.

No exemplo seguido neste apêndice são consideradas duas variáveis, o tempo e o esforço. Para cada uma das variáveis há uma série estatística correspondente. A série estatística para o tempo é (10, 14, 17, 23, 24, 26, 26, 30, 35, 45).

A.2 - Medidas de Tendência Central

As medidas de tendência central são valores típicos ou representativos de uma série estatística. As medidas mais comuns são: a média aritmética, a mediana e a moda. Cada uma delas apresenta vantagens e desvantagens, dependendo dos dados e dos fins desejados.

A **média aritmética**, ou simplesmente média, é definida como:

$$\overline{X} = \frac{\Sigma X}{n} \qquad (A.1)$$

onde X representa os valores assumidos pela variável e n o número de valores. No exemplo,

$$\overline{X} = (10+14+17+23+24+26+26+30+35+45) / 10 = 25 \text{ min}$$

A mediana é definida como sendo o valor que divide uma série estatística em duas metades, ou seja, 50 % dos valores da série são menores do que a mediana. Estando a série estatística ordenada em ordem crescente de grandeza, a mediana é o valor central se a série tiver número ímpar de valores, ou a média dos dois valores centrais, se a série tiver número par de valores. No exemplo há uma seqüência de 10 valores, cujos termos centrais são 24 e 26; a mediana é, portanto, 25 minutos.

A.3 - Medidas de Dispersão

Embora as medidas de tendência central sejam uma boa representação de uma série estatística, elas nada informam sobre

a distribuição dos valores. Para se ter uma melhor idéia do comportamento da distribuição de uma série estatística, é comum calcular-se também as medidas de dispersão. Estas medidas se referem ao grau de dispersão dos valores numéricos em torno do valor médio. A mais importante medida de dispersão é o desvio padrão.

O **desvio padrão** é também chamado de desvio da raiz média quadrática por ser a raiz quadrada da média dos desvios dos valores em relação a média aritmética da série. Assim,

$$s_x = (\Sigma(X - \bar{X})^2 / n)^{0,5} = (\Sigma X^2/n - \bar{X}^2)^{0,5} \qquad (A.2)$$

No exemplo, o desvio padrão da série de tempos é:

$$s = (942 / 10)^{0,5} = 9,70 \text{ min}$$

Enquanto que um desvio de 10 minutos pode ser considerado muito grande quando se trata de uma série de média de 25 minutos, este mesmo desvio seria muito pequeno se a média fosse 2500 minutos. Para evitar este problema, define-se uma medida de grandeza relativa do desvio padrão, o **coeficiente de variação ou dispersão** V, como sendo:

$$V = \frac{s}{\bar{X}} \qquad (A.3)$$

No exemplo, V = 9,7/25 = 0,388. Ou seja, o desvio padrão é de cerca de 39% da média.

Em uma série com distribuição de freqüencia dita normal (o que se verifica para muitas estatísticas, quando o tamanho da

amostra é maior que 30 [SPIE77]), 68,27% dos valores da série estão entre a faixa de $\overline{X} \pm s$, 80,0% entre $\overline{X} \pm 1{,}28s$, 95,45% entre $\overline{X} \pm 2s$ e 99,73% entre $\overline{X} \pm 3s$.

Mesmo para populações de distribuição não normal, a utilização deste e de outros conceitos da distribuição normal ainda é aplicável para alguns casos. De acordo com o **Teorema do Limite Central**, para um número de observações muito grande, a média aritmética das amostras (seleção aleatória de alguns valores) da população assume uma Distribuição Normal [WONN72].

A.4 - Relações Entre Duas Variáveis

Uma técnica muito utilizada na pesquisa de métricas é o teste com duas medidas para verificar se elas estão relacionadas de alguma maneira. Por exemplo, se se deseja descobrir se um programa com um valor maior de esforço leva mais tempo para ser elaborado, deve-se fazer uma coleta de dados sobre os programas desenvolvidos, cujo tempo de desenvolvimento e o esforço são conhecidos, e aplicar as técnicas estatísticas para verificação da hipótese. Esta seção se dedica a essas principais técnicas de verificação do relacionamento entre duas variáveis que obedecem aproximadamente a uma Distribuição Normal.

Um primeiro passo na análise do relacionamento é a elaboração de um gráfico, onde no eixo das abscissas normalmente se coloca a variável independente, ou causal, e no eixo das ordenadas a variável dependente, ou resultante.

No geral, se deseja verificar o efeito da variável do eixo x na variável do eixo y. Por esta razão, no exemplo coloca-se a

variável E (esforço) no eixo x e a variável Tempo no eixo y, como mostra a figura A.1.

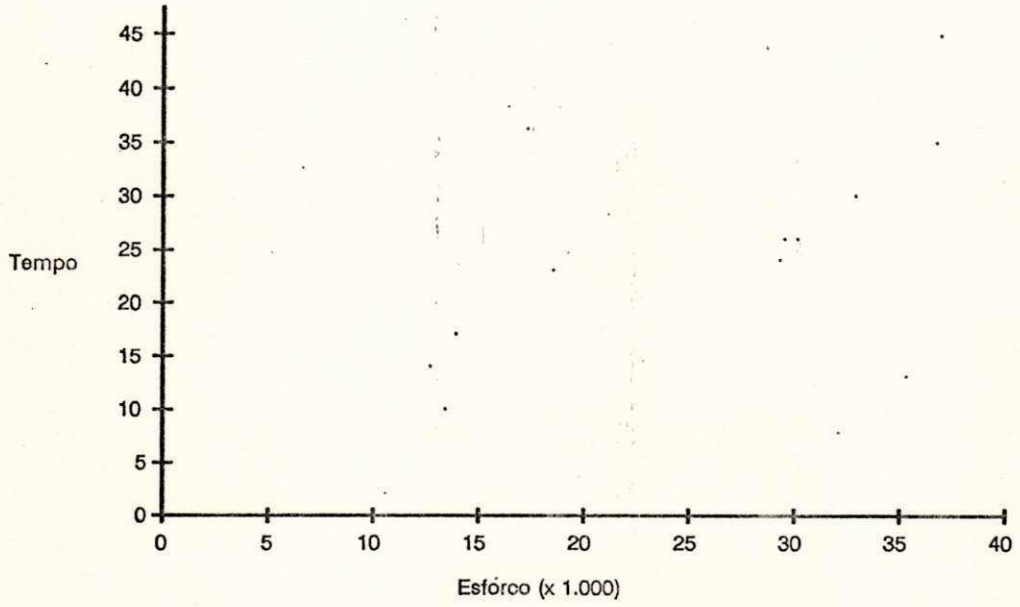


Figura A.1 - Esforço (d.m.e.) x Tempo (min) - Exemplo

A.4.1 - Regressão Linear

A regressão linear é o processo utilizado para encontrar a equação da reta que melhor se adapta para descrever o relacionamento entre duas variáveis. O método mais comum de regressão linear é o método dos mínimos quadrados. Neste método, a reta que mais se ajusta aos pontos dados é aquela que minimiza a soma dos quadrados das distâncias verticais entre a reta e o ponto real. Para se encontrar a equação da reta na forma $Y = a + bX$, aplicam-se as seguintes fórmulas:

$$b = \frac{\Sigma(XY) - n\overline{XY}}{\Sigma(X^2) - n\overline{X}^2} \qquad (A.4)$$

$$a = Y - bX \qquad (A.5)$$

O coeficiente a é chamado de interseção no eixo dos Y e o coeficiente b de coeficiente angular ou de regressão.

No exemplo, a melhor reta (figura A.2) é: Y = 0,8374362 + 0,0009479X.

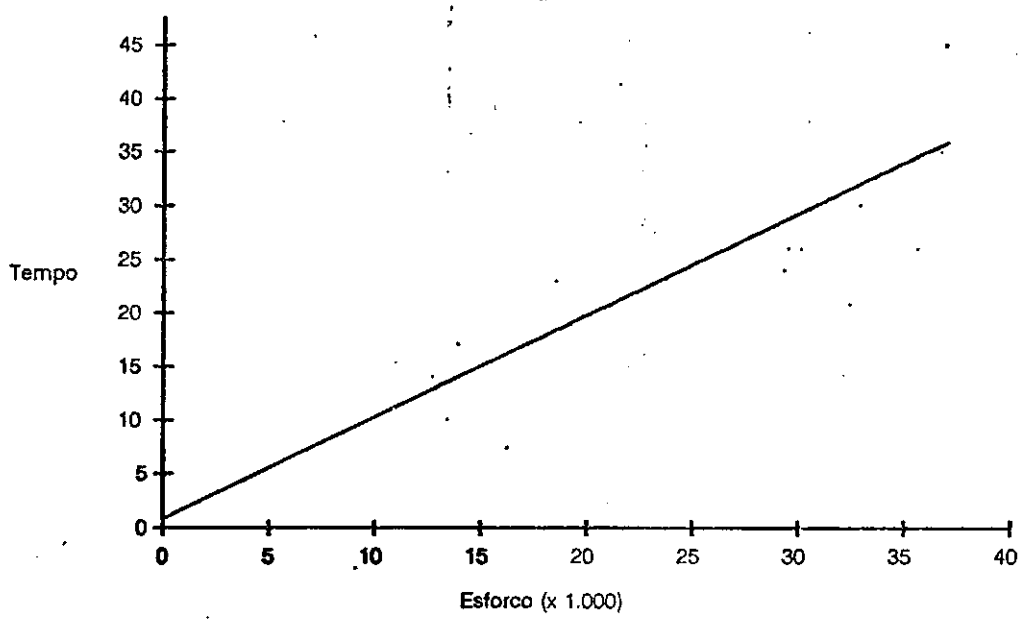


Figura A.2 - Equação da melhor reta - Exemplo: (Y = 0,8374362 + 0.0009479X)

Existem metodologias que permitem a regressão linear de tal forma que a curva cruze o eixo dos Y num ponto determinado. Isto é útil para encontrar equações de regressão mais razoáveis. No exemplo, pela própria definição de esforço, sabe-se que quando este se aproxima de zero, o programa passa a ter poucos tokens e portanto o tempo de programação aproxima-se também de zero. Dessa forma, a equação da reta seria mais coerente se cruzasse o eixo Y no valor zero. Este tipo de regressão é bastante freqüente neste trabalho.

A.4.2 - Coeficiente de Correlação Linear

Uma vez encontrada a equação da reta que melhor se ajusta aos dados, como saber matematicamente, e não só visualmente, quão bem a reta se ajusta aos pontos? Para isso, se determina o **coeficiente de correlação linear**, ou simplesmente correlação, que é utilizado para medir o grau de relacionamento linear entre duas variáveis. Em outras palavras, a correlação indica a proporção da variação na variável dependente que pode ser explicada pela variável independente considerada [EZEK59].

Quando duas variáveis apresentam coeficiente de correlação linear alto, significa que quando há variação em uma variável são grandes as chances de haver uma variação de tamanho proporcional na outra variável. Quando as variações são no mesmo sentido, o coeficiente tem valor positivo. Quando são no sentido contrário (a medida que o valor de uma variável aumenta, o da outra diminue) o coeficiente assume valor negativo. Quanto mais fraco o relacionamento linear , mais o coeficiente aproxima-se de zero. Um relacionamento perfeito (por exemplo, relação entre a circunferência e o raio de um círculo), por outro lado, apresenta coeficiente igual a 1 ou -1. Os exemplos da figura A.3 ilustram a compreensão destes conceitos.

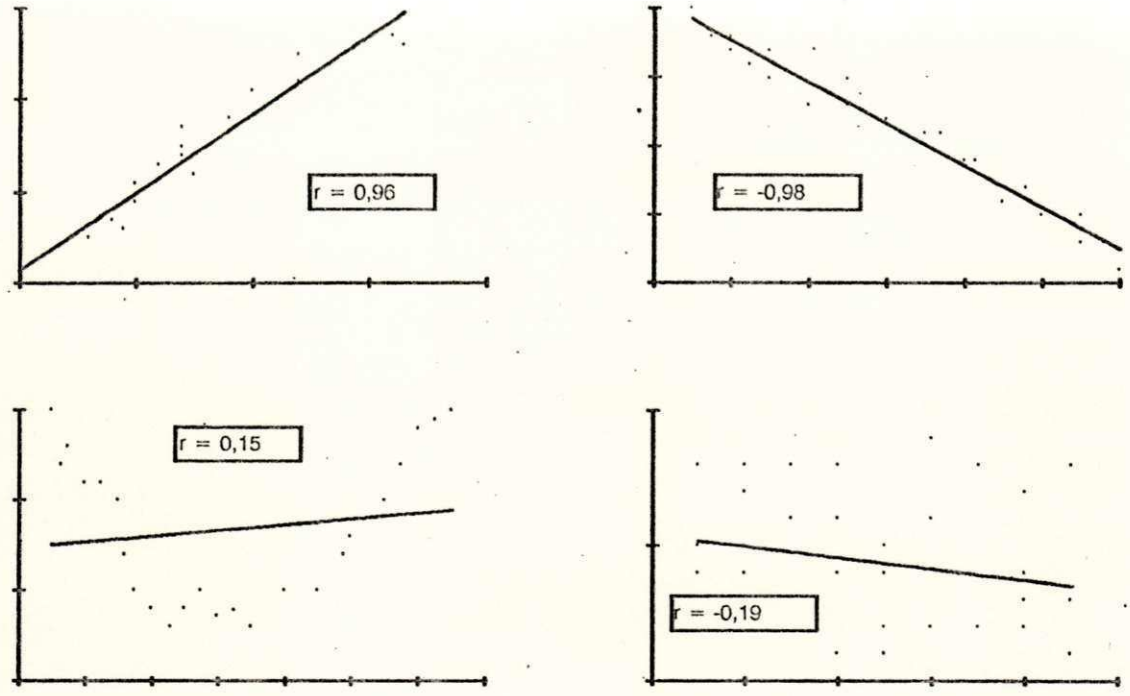


Figura A.3 - Exemplos de correlações lineares

O coeficiente de correlação pode ser calculado pela seguinte fórmula:

$$r = \frac{\Sigma\ (X - \overline{X})(Y - \overline{Y})}{(n-1)\ s_x\ s_y} \qquad (A.6)$$

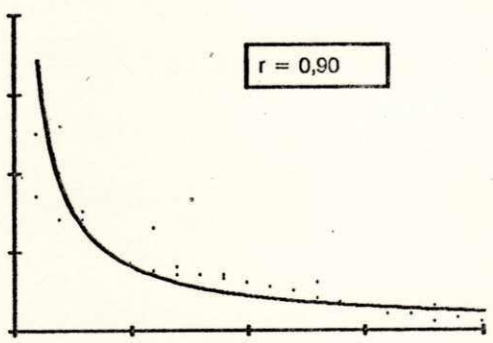
No exemplo, r = 0,90342.

A avaliação do valor de r é uma tarefa um tanto subjetiva e dependente da evolução da área pesquisada. Enquanto que um coeficiente de correlação como 0,95 pode ser pouco convincente na Engenharia Civil, por exemplo, uma correlação de 0,60 chega a ser bem animadora em estudos de Psicologia. Embora seja um limite difícil de estipular e sujeito a discussões, pode-se dizer que modelos de métricas de software que apresentam correlações acima de 0,90 são considerados precisos [FITZ78b].

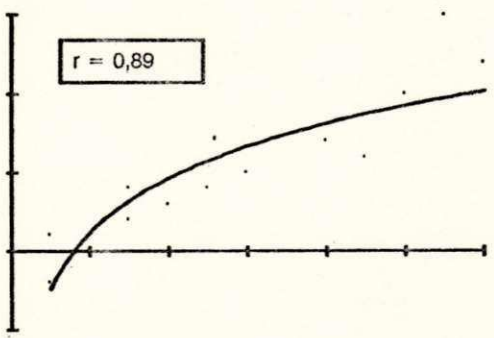
A.4.3 - Regressões Não Lineares

Quando os pontos de duas variáveis não têm boa correlação linear e portanto a equação da reta encontrada não tem muita utilidade, pode-se partir para descobrir outros tipos de relacionamentos. Por exemplo, de acordo com a nossa intuição, espera-se que o esforço em depurar um programa cresça rapidamente (talvez exponencialmente) com o tamanho desse.

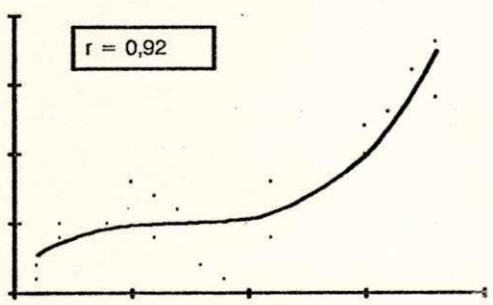
Existem métodos de regressão para diversos tipos de curvas. Os métodos de regressão praticados neste trabalho foram: logarítmica, polinomial, exponencial e exponencial de base e. A figura A.4 exemplica alguns tipos de regressão não-linear.



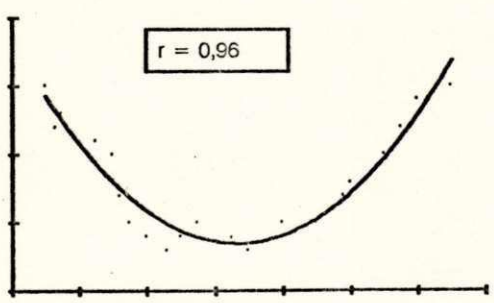
$Y = 34,216\ X^{-0,891}$



$-2,386 + 5,049 \ln X$



$Y = 1,54+1,39X-0,19X^2+0,01X^3$



$Y = 23,37-4,46X+0,33X^2$

Figura A.4 - Exemplos de regressões não-lineares

Para as regressões não-lineares, também se pode calcular os coeficientes de correlação, através da seguinte fórmula:

$$r = (1 - \Sigma(Y - Y_{est})^2 / \Sigma(Y - \bar{Y})^2)^{0,5} \qquad (A.7)$$

Em regressões não lineares, o coeficiente de correlação é sempre positivo — como se pode deduzir a partir da fórmula — variando de 0 a 1.

Ao se tentar aplicar as diversas regressões à série estatística do exemplo, as seguintes equações e coeficientes de correlação foram encontradas:

Regressão	Equação	Coef. r
Logarítmica	$-184 + 20,81 \ln E$	0,88
Polinomial grau 2	$20,48 - 0,00094 E + 0,0000000388 E^2$	0,93
Exponencial	$0,00196 E^{0,931}$	0,90
Expon. de base e	$8,00 e^{0,0000415 E}$	0,91
Linear	$0,837 + 0,00095 E$	0,90

De acordo com os resultados da análise acima, conclui-se que, para os dados observados, a melhor equação que explica o relacionamento entre as variáveis é a equação polinomial do segundo grau. Mas esta conclusão é precipitada devido a pequena quantidade de observações.

A.4.4 - Erro Padrão de Estimativa

Uma equação da curva que melhor se ajusta aos dados observados pode ser utiizada para estimativas. Assim, pode-se res-

ponder perguntas do tipo: supondo-se que um determinado programa terá esforço de 30.000, qual o tempo de desenvolvimento mais provável? Utilizando o modelo de regressão linear, basta substituir X = 30.000 na equação da reta para encontrar um Y de aproximadamente 29 minutos. Mas surge naturalmente uma pergunta: com que segurança pode-se afirmar que o tempo real de desenvolvimento será de 29 minutos?

Uma maneira simples de se responder esta pergunta é calculando o **erro padrão de estimativa**, dado por:

$$s_{y \cdot x} = (\Sigma(Y - Y_{est})^2 / n)^{0,5} \qquad (A.8)$$

No exemplo, o erro padrão de estimativa para a regressão linear é de aproximadamente 4,16.

O significado do erro padrão é semelhante ao do desvio padrão visto na seção A.3. E a partir dele, pode-se especificar o **intervalo de confiança** de uma estimativa. Assim, numa estimativa, espera-se que o valor em aproximadamente 68% das vezes estejam dentro do intervalo de confiança $Y_{est} \pm s_{y.x}$. No exemplo, deduz-se que em cada 100 programas de E = 30.000, espera-se que 68 tenham tempo de desenvolvimento entre 25,11 a 33,43 minutos (29,27 ± 4,16 minutos).

A.4.5 - Desvio Relativo Médio - DRM

As medidas de desvio são outra maneira de se avaliar o grau de ajustamento de uma curva de previsão. Uma delas, o **Desvio Relativo Médio - DRM**, é dado pela fórmula (A.9):

$$DRM = \Sigma \mid (Y - Y_{est})/Y \mid / n \qquad (A.9)$$

Como mede a média dos erros relativos, quanto menor o valor do DRM, mais precisa é a curva ajustada. No exemplo, o modelo de regressão linear apresentou um DRM de 0,153 ou 15,3%. De acordo com [CONT86], modelos de previsão na área de métricas de software são considerados satisfatórios se apresentarem DRM ≤ 0,25.

A.4.6 - Desvio Quadrático Médio - DQM

O **Desvio Quadrático Médio - DQM** é outra medida de avaliação do grau de ajustamento de uma curva baseada no valor médio do erro minimizado pelo modelo de regressão. Pode ser calculado por:

$$DQM = \left(n \, \Sigma(Y - Y_{est})^2 \right)^{0.5} / \Sigma Y \qquad (A.10)$$

No exemplo, o modelo de regressão linear apresenta um DQM de 0,164. De acordo com [CONT86] um modelo aceitável de previsão na área de métricas de software deve apresentar DQM ≤ 0,25. Séries com dados muito heterogêneos, ou seja, de grande variação, são mais prováveis de apresentar elevados DQM.

A.4.7 - Previsão a um Nível de Erro e - Prev (e)

Este critério mede o índice de acertos de estimativas de um modelo dentro de um certo erro (ou desvio) e. Seja k o número de dados cujas estimativas estejam dentro do erro e estabelecido, isto é, $|(Y-Y_{est})/Y| \leq e$ A previsão ao nível de erro

e, Prev(e), é definida como:

$$Prev\ (e) = k\ /\ n \qquad (A.11)$$

No exemplo, se o erro considerado for de 0,25, então para o modelo de regressão linear, a Prev (0,25) = 0,90. Isto significa que 90% das estimativas têm erro relativo menor que 25%. [CONT86] considera um modelo na área de métricas de software aceitável quando seu Prev(0,25) ≥ 0,75.

REFERÊNCIAS BIBLIOGRÁFICAS

[ALBR79] Albrecht, Allan J. Measuring application development productivity. Proceedings of the Joint SHARE/GUIDE Symposium, 1979, 83-92.

[ARTH85] Arthur, Lowell J. Measuring programmer productivity and software quality. EUA: John Wiley & Sons, 1985.

[BAER84] Baer, Jean-Loup. Computer architecture. Computer. 1984, 10(17), 77-79.

[BASI75] Basili, V. R. & Turner, A. J. Iterative enhancement: a practical technique for software development. IEEE Transactions on Software Engineering, 1975, 1, 390-396.

[BASI79] Basili, V. R. & Reiter Jr., R. W. An investigation of human factors in software development. Computer, 1979, 12, 21-38.

[BASI83a] Basili, V. R. & Hutchens, D. H. An empirical study of a syntactic complexity family. IEEE Transactions on Software Engineering, 1983, 9(6), 664-672.

[BASI83b] Basili, V. R., Selby Jr, R. W. & Phillips T. Metric analysis and data validation across Fortran projects. IEEE Transactions on Software Engineering, 1983, 9(6), 652-663.

[BASI84] Basili, V. R. & Weiss, D. M. A methodology for collecting valid software engineering data. IEEE Transactions on Software Engineering, 1984, 10(6), 728-738.

[BERG73] Berge, C. Graphs and hypergraphs. Holanda: North-Holland, 1973.

[BERG85] Berghel, H. L. & Sallach, D. L. Computer program plagiarism detection: the limits of the Halstead metric. J. Educational Computing Research, 1985, 1(3), 295-315.

[BERR85] Berry, R. E. & Meekings, B. A. E. A style analysis of C programs. Communications of the ACM, 1985, 28(1), 80-88.

[BOEH78] Boehm, B. W., Brown, J. R., Kaspar, H., Lipow, M., MacLeod, G. J. & Merritt, M. J. Characteristics of software quality. EUA: North-Holland, 1978.

[BOEH84] Boehm, Barry W. Software engineering economics. IEEE Transactions on Software Engineering, 1984, 10(1), 4-

21.

[BOEH87] Boehm, Barry W. Improving software productivity. Computer, 1987, 43-57.

[BROO87] Brooks, Fred P. No silver bullet - essence and accidents of software engineering. Computer, 1987, 20(4), 10-19.

[BUCK84] Buckley, Fletcher J. Software quality assurance. IEEE Transactions on Software Engineering, 1984, 10(1), 36-41.

[CARD87] Card, D. N., McGarry, F. E. & Page, G. T. Evaluating software engineering technologies. IEEE Transactions on Software Engineering, 1987, 13(7), 845-851.

[CHEN78] Chen, Edward T. Program complexity and programmer productivity. IEEE Transactions on Software Engineering, 1978, 4(3), 187-194.

[CHRI81] Christensen, K., Fitsos, G. P. & Smith, C. P. A perspective on Software Science. IBM Systems Journal, 1981, 20(4), 372-387.

[CONT86] Conte, S. D., Dunsmore, H. E. & Shen, V. Y. Software engineering metrics and models. EUA: Benjamim/ Cummings Publishing Company, 1986.

[COUL83] Coulter, Neal S. Software Science and cognitive psychology. IEEE Transactions on Software Engineering, 1983, 9(2), 166-171.

[CURT83] Curtis, Bill. Software metrics: guest editor's introduction. IEEE Transactions on Software Engineering, 1983, 9(6), 637-638.

[DAVC82] Davcev, D. Some new observations about Software Science indicators for estimating software quality. Proceedings of the First Symposium On Empirical Foundations of Information and Software Science, 1982.

[DAVI87] Davies, G. & Tan, A. A note on metrics of Pascal programs. ACM SIGPLAN Notices, 1987, 22 (8), 39-44.

[DAVI88] Davis, J. S. & LeBlanc, R. J. A study of the applicability of complexity measures. IEEE Transactions on Software Engineering, 1988, 14(9), 1366-1372.

[DEMA89] DeMarco, Tom. Controle de projetos de software, Brasil: Editora Campus Ltda., 1989.

[DUNS78] Dunsmore, H. E. The influence of programming factors

on program complexity. Dissertação de PhD, Dep. Computer Sciences. EUA: University Maryland, 1978.

[ELSH76a] Elshoff, J. L. An analysis of some commercial PL/I programs. IEEE Transactions on Software Engineering, 1976, 2, 113-120.

[ELSH76b] Elshoff, J. L. Measuring commercial PL/I programs using Halstead's criteria. ACM SIGPLAN Notices, 1976, 11(5), 38-46.

[EZEK59] Ezekiel, M. & Fox, K. A. Methods of correlation and regression analysis. EUA: John Wiley & Sons, 1959.

[FELI89] Felician, L. & Zalateu, G. Validating Halstead's theory for Pascal programs. IEEE Transactions on Software Engineering, 1989, 15(12), 1630-1632.

[FENI79] Fenichel, Robert. Heads I win, tails you lose. Correspondência à seção Surveyors' Forum. Computing Surveys, 1979, 11(3), 277-278.

[FITZ78a] Fitzsimmons, Ann & Love, T. A review and evaluation of Software Science. Computing Surveys, 1978, 10(1), 3-18.

[FITZ78b] Fitzsimmons, Ann & Love, T. Correspondência à seção Surveyors' Forum. Computing Surveys, 1978, 10(4), 504-505.

[FORT89] Revista Fortune, edição de novembro de 1989.

[GAFF84] Gaffney Jr., John E. Estimating the number of faults in code. IEEE Transactions on Software Engineering, 1984, 10(4), 459-464.

[GORD79a] Gordon, Ronald D. Measuring improvements in program clarity. IEEE Transactions on Software Engineering, 1979, 5(2), 73-90.

[GORD79b] Gordon, Ronald D. A qualitative justification for a measure of program clarity. IEEE Transactions on Software Engineering, 1979, 5(2), 177-182.

[GRAD87] Grady, R. B. & Caswell, D. L. Software metrics: establishing a company-wide program. EUA: Prentice-Hall, 1987

[GRAD90] Grady, Robert B. Work-product analysis: the philosopher's stone of software?. IEEE Software, 3, 26-34.

[HALL84] Hall, N. R. & Preiser, S. Combined network complexity measures. IBM Journal of Research and Development, 1984, 28(1), 15-27.

[HALS77] Halstead, Maurice H. Elements of Software Science. EUA: North-Holland, 1977.

[HARR86] Harrison, Warren & Cook, Curtis R. A note on the Berry-Meekings style metric., Communications of the ACM, 1986, 29(2),123-125

[HENR90] Henry, S. & Selig, C. Predicting source-code complexity at the design state, IEEE Software, 1990, 3, 36-44.

[INGO75] Ingojo, Jose C. Modularization in the Pilot compiler and its effect on the length. Computer Science Department Technical Report 169. EUA: Purdue University, 1975.

[IVAN87] Ivan, Ion. Programs complexity: comparative analysis, hierarchy, classification. ACM SIGPLAN Notices, 1987, 22(4), 94-102.

[JENS85] Jensen, H. A. & e Vairavan, K. An experimental study of software metrics for real-time software. IEEE Transactions on Software Engineering, 1985, 11(2), 231-234.

[JOHN81] Johnston, D. B. & Lister, A. M. A note on the Software Science length equation. Software Practice and Experience, 1981, 11, 875-879.

[KOKO88] Kokol, P., Ivanek, B. & Zumer, V. Software effort metrics: how to join them. ACM Sigsoft Software Engineering Notes, 1988, 13(2), 55-57.

[KONS85] Konstam, A. H. & Wood, D. E. Software Science applied to APL. IEEE Transactions on Software Engineering, 1985, 11(10), 994-1000.

[LASS79] Lassez, J. L. & Knijff, D. Evaluation of length and level for simple program schemes. Proceedings of IEEE COMPSAC79, EUA, 1979.

[LEW88] Lew, K. S., Dillon, T. S. & Forward, K. E. Software complexity and its impact on software reliability. IEEE Transactions on Software Engineering, 1988, 14(11), 1645-1655.

[LI87] Li, H. F. & Cheung, W. K. An empirical study of software metrics. IEEE Transactions on Software Engineering, 1987, 13(6), 697-708.

[LIND89] Lind, R. K. & Vairavan, K. An experimental investigation of software metrics and their relationship to software development effort. IEEE Transactions on Software Engineering, 1989, 15(5),

649-653.

[LIPO82] Lipow, M. Number of faults per line of code. IEEE Transactions on Software Engineering, 1982, 8(4), 437-440.

[MCCA76] McCabe, Tom J. A complexity measure. IEEE Transactions on Software Engineering, 1976, 2(4), 308-320.

[MILL80] Mills, H. D., O'Neill, D., Linger, R. C., Dyer, M. & Quinnan, R. E. The management of software engineering. IBM Systems Journal, 1980, 19(4), 415-477.

[MIZU83] Mizuno, Y. Software quality improvement. Computer, 1983, 16(3), 66-72.

[MOAD90] Moad, J. The software revolution. Datamation, 1990, 2, 22-30.

[MOHA81] Mohanty, Siba N. Software cost estimation: present and future. Software - Practice and Experience, 1981, 11, 103-121.

[MORA78] Moranda, P. B. Is Software Science Hard ?. Correspondência à seção Surveyors' Forum. Computing Surveys, 1979, 10(4), 503-504.

[MYER77] Myers, Glenford J. An extension to the cyclomatic measure of program complexity. SIGPLAN Notices, 1977, 10, 61-64.

[OTTE79] Ottenstein, Linda M. Quantitative estimates of debugging requirements. IEEE Transactions on Software Engineering, 1979, 5(5), 504-514.

[OVIE80] Oviedo, E. I. Control flow, data flow and program complexity. Proceedings of IEEE COMPSAC, 1980, 11, 146-152.

[PARI90] Parikh, Girish. Reengenharia de software - técnicas de manutenção de programas e sistemas. Rio: Livros Técnicos e Científicos Editora Ltda, 1990.

[POOR88] Poore, J. H. Derivation of local software quality metrics (software quality circles). Software - Practice and Experience, 1988, 18(11), 1017-1027.

[RAMA84] Ramamoorthy, C. V., Prakash, A., Tsai, W. & Usuda, Y. Software engineering: problems and perspectives. Computer, 1984, 17(10), 191-209.

[RAMA88] Ramamurthy, B. & Melton, A. A synthesis of Software Science measures and the Cyclomatic Number. IEEE

Transactions on Software Engineering, 1988, 14(8), 1116-1121.

[RAMB85] Rambo, R., Buckley, P. & Branyan, E. Establishment and validation of software metric factors. Proceedings of the International Society of Parametric Analysts Seventh Annual Conference. EUA: International Society of Parametric Analysts, 1985, 406-417.

[REDI86] Redish, K. A. & Smyth, W. F. Program style analysis: a natural by-product of program compilation. Communications of the ACM, 1986, 29(2), 126-133.

[SALT82] Salt, Norman F. Defining Software Science counting strategies. ACM SIGPLAN Notices, 1982, 17(3), 58-67.

[SCHN79] Schneidewind, N. F. An experiment in software error data collection and analysis. IEEE Transactions on Software Engineering, 1979, 5(3), 276-286.

[SCHN88a] Schneider, Victor. A reply to "a note on metrics of Pascal programs". ACM SIGPLAN Notices, 1988, 23(1), 38-39.

[SCHN88b] Schneider, Victor. Approximations for the Halstead Software Science software error rate and project effort estimators. ACM SIGPLAN Notices, 1988, 23(1), 40-47.

[SHEN80] Shen, V. Y. & Dunsmore, H. E. A Software Science analysis of COBOL programs. Technical Report CSD-TR-348. EUA: Purdue University, 1980.

[SHEN81] Shen, V. Y. & Dunsmore, H. E. A. Analyzing Cobol programs via Software Science. Technical Report CSD TR-348, EUA: Purdue University, 1981.

[SHEN83] Shen, V. Y., Conte, S. D. & Dunsmore, H. E. Software Science revisited: a critical analysis of the theory and its empirical support. IEEE Transactions on Software Engineering, 1983, 9(2), 155-165.

[SHEN85] Shen, V. Y., Yu, T., Thebaut, S. M. & Paulsen, L. R. Identifying error-prone software - an empirical study. IEEE Transactions on Software Engineering, 1985, 11(4), 317-324.

[SMIT80] Smith, C. P. A Software Science analysis of IBM programming products. Technical Report TR 03.081. EUA: IBM Santa Teresa Laboratory, 1980.

[SPIE77] Spiegel, Spiegel R. Estatística. Brasil: McGraw-Hill do Brasil, 1977.

[STEV74] Stevens, W. P., Myers, G. J. & Constantine, L. L. Structural design. IBM Systems Journal, 1974, 13(2).

[STRO66] Stroud, John M. The fine structure of psychological time. Annals of New York Academy of Sciences, 1966, 623-631.

[UNDE54] Underwood, X. Elementary Statistics, EUA: Appleton-Century-Crofts, 1954.

[WAGU87] Waguespack Jr., L. J. & Badlani, S. Software complexity assesment: an introduction and annotated bibliography. ACM SIGSOFT Software Engineering Notes, 1987, 12(4), 52-71.

[WALP79] Walpole, R. & Meyers, R. H. Probability and Statistics for engineers and scientists. EUA: MacMillan, 1979.

[WALS79] Walsh, T. J. A software-reliability study using a complexity measure. Proceedings of National Computer Conference. EUA: AFIPS Press, 1979, 761-768.

[WEYU88] Weyuker, Elaine J. Evaluating software complexity measures. IEEE Transactions on Software Engineering, 1988, 14(9), 1357-1365.

[WONN72] Wonnacott, T. H. & Wonnacott, R. J. Introductory Statistics. Canadá: John Wiley & Sons, 1972.

[WOOD79] Woodward, M. R., Hennel, M. A. & Hedley, D. A measure of control flow complexity in program text. IEEE Transactions on Software Engineering, 1979, 5(1), 45-50.

[YAU85] Yau, S. S. & Collofello, J. S. Design stability measures for software maintenance. IEEE Transactions on Software Engineering, 1985, 11(9), 849-856.

[ZWEB79a] Zweben, S. H. & Halstead, M. H. The frequency distribution of operators in PL/I programs. IEEE Transactions on Software Engineering, 1979, 5(2), 91-95.

[ZWEB79b] Zweben, S. H. & Kin-Chee, F. Exploring Software Science relations in COBOL and APL. Comsac 79 Proceedings, 1979, 702-707.