(2)

AIR WAR COLLEGE

RESEARCH REPORT

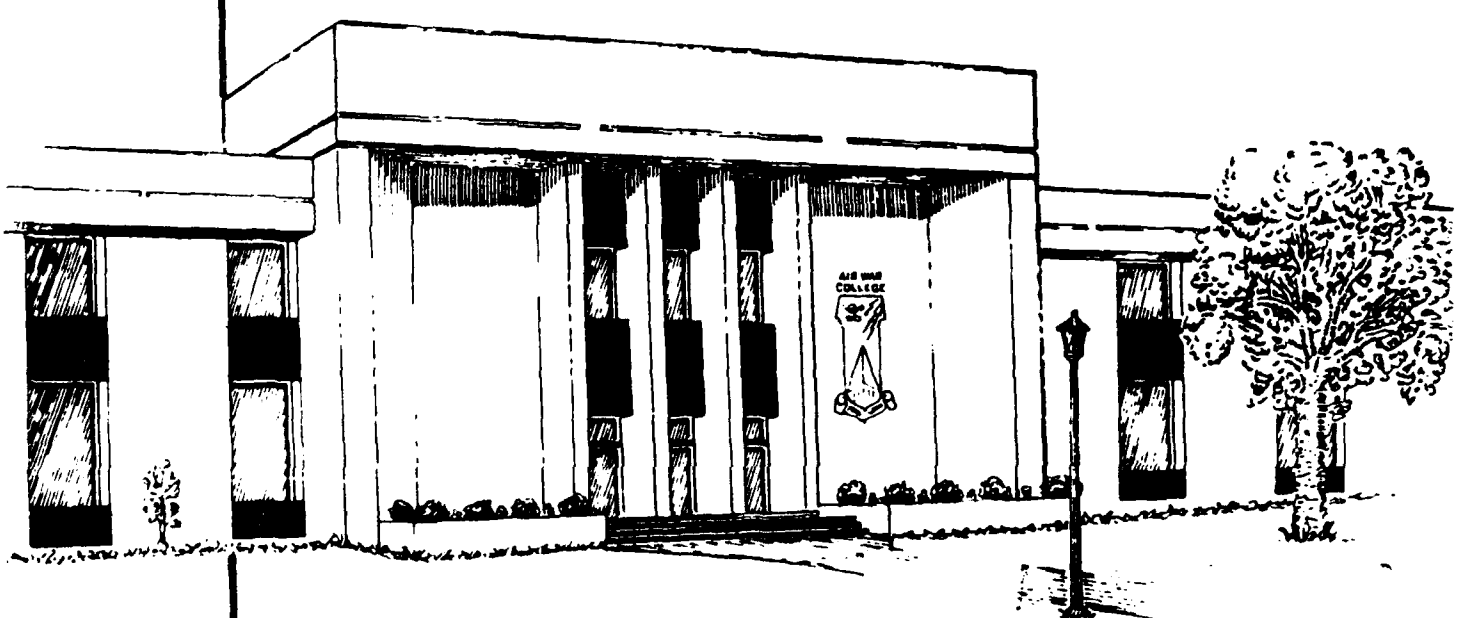
SOFTWARE METRICS

USEFUL TOOLS OR WASTED MEASUREMENTS?

LIEUTENANT COLONEL ROBERT A. AUSTIN

LIEUTENANT COLONEL JOHN M. CASE, JR

1990



AIR UNIVERSITY
UNITED STATES AIR FORCE
MAXWELL AIR FORCE BASE, ALABAMA

AIR WAR COLLEGE

AIR UNIVERSITY

Software Metrics

Useful Tools or Wasted Measurements?

by

Robert A. Austin
Lieutenant Colonel, USAF

and

John M. Case, Jr.
Lieutenant Colonel, USAF

A DEFENSE ANALYTICAL STUDY SUBMITTED TO THE FACULTY

IN

FULFILLMENT OF THE CURRICULUM

REQUIREMENT

Advisor: Mr. Robert O. Dahl

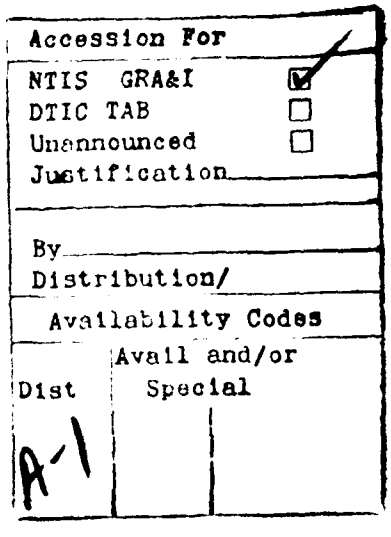
MAXWELL AIR FORCE BASE, ALABAMA

May 1990

DISCLAIMER

This study represents the views of the authors and does not necessarily reflect the official opinion of the Air War College or the Department of the Air Force. In accordance with Air Force Regulation 110-8, it is not copyrighted but is the property of the United States government.

Loan copies of this document may be obtained through the interlibrary loan desk of Air University Library, Maxwell Air Force Base, Alabama 36112-5564 (telephone [205] 293-7223 or AUTOVON 875-7223).



EXECUTIVE SUMMARY

TITLE: Software Metrics: Useful Tools or Wasted Measurements?

AUTHORS: Lieutenant Colonel Robert A. Austin

and

Lieutenant Colonel John M. Case, Jr.

A large number of Air Force programs require significant software development during implementation. Unfortunately, the software almost always takes longer than anticipated and requires extensive manpower to remove latent defects from the delivered product. Industry and academia are doing a great deal of work in the developing field of software production and quality metrics--tools designed to help estimate project size and assess software quality, thus alleviating this industry-wide problem. This study examines the current state of these tools to see if they provide useful measures which can improve the software component of Air Force programs or if the effort used to implement these metrics represents wasted manpower. The study concludes with an assessment that the tools provide valuable data which, when properly applied, can help software managers set achievable schedules for quality software products which can be completed on time and require significantly less software maintenance manpower.

BIOGRAPHICAL SKETCHES

Lieutenant Colonel Robert A. Austin (B.S. in Botany, University of Maryland; M.S. Management, Troy State University) has spent the majority of his career working with computers. He spent a year in the Education with Industry program working on the development of a personal computer. From there, he was assigned to Air Training Command where he managed a variety of software development projects and pioneered the introduction of personal computers throughout the Command. Subsequently, he served as Chief, Systems Division, Office of the Administrative Assistant to the Secretary of the Air Force, where he managed an organization which provided computer and data communications support to the Secretary of the Air Force and his staff.

Lieutenant Colonel John M. Case, Jr. (B.S. and M.S. in Mathematics, University of Tennessee; M.S. Computer Science, The Johns Hopkins University) has been actively involved with computers since his college undergraduate days. In the first half of his Air Force career, he designed numerous classified intelligence processing systems for Electronic Security Command and the National Security Agency. Subsequently he served as Deputy Director for Engineering in the Architecture Division of the Air Force System Integration Office at Space Command, where he oversaw the development of a major missile warning display system, and served as Deputy Program Manager for Data Networks with the NATO Communications and Information Systems Agency.

TABLE OF CONTENTS

Title Page i

Disclaimer ii

Executive Summary. iii

Biographical Sketches. iv

Table of Contents. v

Chapter One: Introduction 1

- The Growth of Computer Technology. 1
- Need for Computer Software 2
- Evolution of Software Metrics. 4
- Problem Statement. 9
- Study Limitations. 9
- Study Methodology. 10

Chapter Two: Software Metric Overview 11

- Why Do We Measure? 11
- What Do Software Metrics Measure?. 13

Chapter Three: Predictive Metrics 16

- Introduction 16
- Criteria 17
- Methods. 21
- Expert Judgment. 21
- Algorithmic Models 22
- Source Lines of Code Measure 23
- Function Point Analysis. 25
- Summary. 31

Chapter Four: Quality Metrics 32

- Introduction 32
- What is Software Quality?. 34
- Quality Factors. 35
- Criteria for Measuring Quality Factors 40
- Summary. 48

Chapter Five: Application of Metrics. 50

- Introduction 50
- Examples 50
- Costs. 53

Chapter Six: Summary and Conclusions. 55

Summary. 55
Conclusions. 57
Recommendations. 58

Bibliography 60

CHAPTER ONE

INTRODUCTION

The Growth of Computer Technology

Almost everyone in America is aware of the revolutionary advances in computer technology that have occurred in the last two decades. It is difficult to think of an area that has not been touched by this revolution--microwave ovens, automobiles, children's toys, cameras, stereo systems, etc., all use computers to control at least some of their functions. Many of us now have complete systems at home that rival the power of mainframes in our college days.

Military weapon systems reflect this same trend towards increased use of computers. Indeed, the military funded much of the research and development which fueled the revolution. This rapid technological growth has resulted in an exponential increase in the computer software component of Air Force weapon systems. As an example, the F-4 fighter used during the Vietnam War contained no digital computers--and therefore no software. The F-16A which replaced it in 1981 had seven computer systems with fifty digital processors and 135,000 lines of code. When the D model of the F-16 became operational in 1986 it had fifteen computer systems and 236,000 lines of code.[1]

---

[1] James W. Canan, "The Software Crisis," Air Force Magazine, 69 (May 1986):49.

Need for Computer Software

Computers do not perform their functions just by assembling the hardware into the appropriate boxes. All computer-based systems need to be told what to do and when to do it. This is the function of the software--the set of instructions which guide the computer in performing its tasks, thus giving the computer its personality. Without software a computer is just a collection of electronic parts that consume electricity and accomplish nothing.

The design and development of computer software is a complex task, and has grown even more so as the capabilities of computer hardware have dramatically improved. Early computers were programmed exclusively in assembly language--the fundamental language of the computer. As the complexity of programs grew, however, it soon became apparent that higher-level languages were needed, and these were soon developed. The rational for language development was to make it easier to program the computers--resulting in easier-to-develop, more reliable computer systems. Ada, the language developed by the Department of Defense for embedded systems, represents the state-of-the-art language for use in defense-oriented systems.

Despite over forty years of experience in the development of computer systems, the services and defense industry--indeed the computer industry in general--has a dismal

record of software production. Systems are rarely completed on time and require excessive manpower to maintain after they are delivered. A typical example of schedule overruns was reported by the Software Development Integrity Program (SDIP) office at the Air Force Aeronautical Systems Division at Wright-Patterson Air Force Base in Ohio. None of the projects tracked in this study were completed on time! Figure 1 illustrates their disturbing findings.[2]

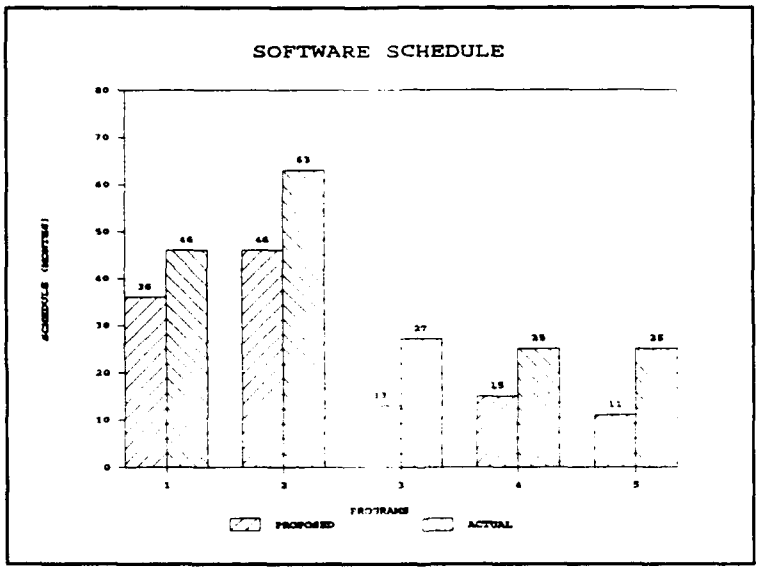


Figure 1

A more recent example is the Peace Shield project. In 1985 the Air Force awarded an order for a $3.7 billion command, control and communications system to Boeing Aerospace &

---

[2]Philip S. Babel, "Software Development Integrity Program (SDIP)," Paper for Air Force Aeronautical Systems Division, Wright-Patterson AFB, Ohio, 1988, pg. 2.

Electronics Corporation of Seattle. Boeing and several of its partners on this program--Computer Sciences Corporation, Westinghouse, ITT and General Electric--are among the most prestigious computer systems houses in the United States. The original software estimate of 223,000 lines of code has more than tripled to 792,000. The delivery date has slipped by more than four years (from 39 months to 92 months).[3] Clearly, some of the largest and most experienced firms in the industry are unable to predict software costs and schedule with any degree of accuracy--a disturbing fact that has caused the Air Force many problems with a $3.7 billion contract.

Evolution of Software Metrics

The field of software metrics has evolved in an effort to solve these software production and quality problems. Software metrics are tools which have been developed to measure various parameters of software, such as lines of code, complexity, quality, and programmer productivity. The following definitions are typical of those we encountered in the literature:

> A metric is a measurable indication of some quantitative aspect of a system.[4]

---

[3]David Hughes,"Boeing Told to Solve Peace Shield Problems," Aviation Week & Space Technology, December 18, 1989, p. 114.

[4]Bo. Lennselius, Claes Wohlin, and Ctirad Vrana, "Software Metrics: fault content estimation and software process control," Microprocessors and Microsystems, 11 (September 1987):365.

> A software metric is a rule for assigning a number or identifier to software in order to characterize the software.[5]
>
> ...software metrics are used to characterize the essential features of software quantitatively, so that classification, comparison, and mathematical analysis can be applied.[6]

Industry and academia have done extensive work in the past two decades aimed at producing software metrics which can do a better job of predicting project size--allowing more accurate scheduling--and controlling software quality--enabling software engineers to produce more reliable systems which will require less maintenance. Have these efforts produced useful tools, or are they simply intellectual exercises developed to justify the researchers' jobs? To answer that, we need to understand several trends which compel managers to seek tools which can help control the software problem. These include:

(1) Software costs are big and growing. The Department of Defense expects to spend $30 billion on software in 1990. This figure has grown rapidly during the last ten years. The figure in 1979 was only $3.3 billion.[7] The civilian sector shares the same trends - software costs

---

[5] H. E. Dunsmore, "Software Metrics: An overview of Evolving Methodology," Information Processing and Management, 20 (1984):183.

[6] S. D. Conte, H.E. Dunsmore, and V.Y. Shen, Software Engineering Metrics and Models, Menlo Park, California, Benjamin/Cummings, 1986, pg. 3.

[7] David J. Marcus, "Project Bold Stroke," Signal Magazine, April 1986, p. 100.

are growing an average of 12 percent per year and will equal 13 percent of our Gross National Product by the end of 1990.[8] These costs are large enough to merit serious efforts to understand and control them.

(2) The software portion of systems is growing. The cost of systems is typically divided between hardware and software costs. Over the last decade the cost of software has increased dramatically as a percentage of the overall project cost. Figure 2 illustrates this trend.[9] Clearly, software has become the major cost factor for new systems, a fact which is particularly true for weapon systems.

(3) There is a growing shortage of computer professionals. The national shortfall of civilian and military software professionals--engineers, managers and programmers--is expected to approach 1,000,000 in 1990.[10] This shortfall is not unique to the military, but affects the civilian sector as well. This forces the military to compete with private companies for scare resources. Many companies are offering top dollar for

---

[8]Barry Boehm and Philip N. Papaccio, "Understanding and Controlling Software Costs," IEEE Transactions on Software Engineering, 14 (October 1988):1462.

[9]Barry W. Boehm, Software Engineering Economics, Englewood Cliffs, N.J., Prentice-Hall, 1981, p. 18.

[10]Canan, pg. 46.

experienced people and are luring away the best military people.

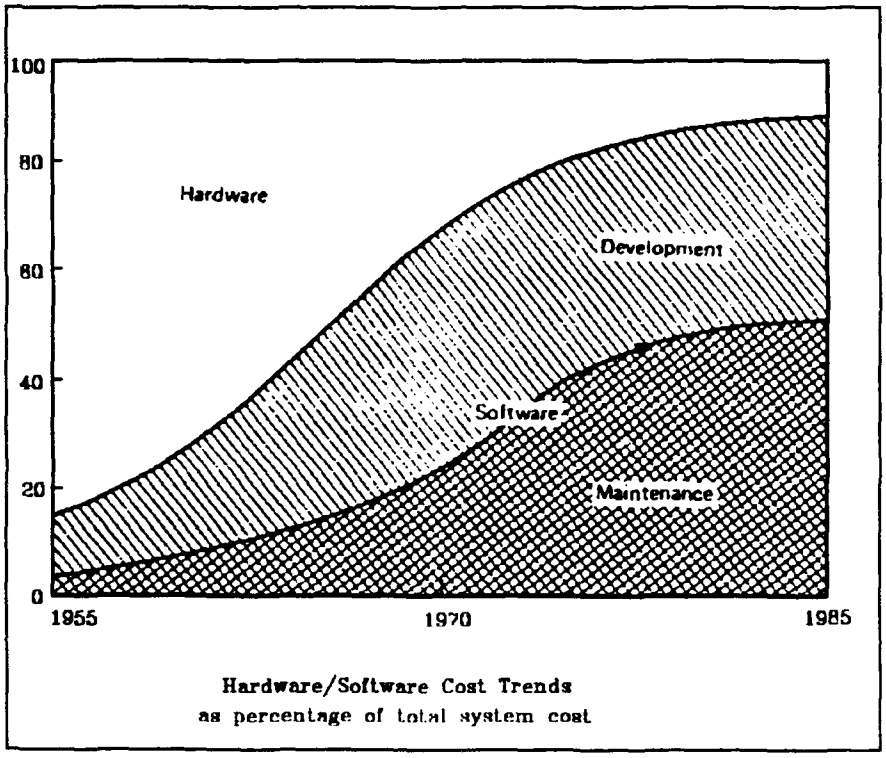


Figure 2

(4) Many useful software products are not getting developed. Due to the rapid growth of software and the shortage of qualified software engineers, many development organizations are simply backlogged. For example, the U.S. Air Force Data Systems Design Center has a four year backlog of important data processing software projects which cannot be developed because of

limited manpower and funding.[11] A similar backlog exists at other software development organizations.

(5) Software quality is a growing problem. The quality of software suffers when the demand exceeds the industry's ability to produce it. Poor quality software wastes valuable resources and, in many military applications, hampers mission accomplishment, or may even endanger lives (it was, in fact, a sof are error that kept the British frigate SHEFFELD from detecting the Argentine missile which destroyed the ship and cost the lives of 20 British sailors during the Falkland Islands conflict[12]).

Clearly there is ample reason to believe the efforts to develop software metrics are aimed at real industry problems and are not job-preserving research efforts. The magnitude of the problem--and the dramatic personnel shortage--ensures it cannot be solved by merely increasing the number of programmers. Such an increase is necessary, but it must be accompanied by gains in productivity. To accomplish this the industry must have useful tools to measure the software product--and this is the key function metrics serve.

---

[11] Boehm and Papaccio, pg. 1462.

[12] P. E. Borkovitz, "Eliminating Bugs from Weapon System Computer Programs," Military Technology, No. 5/88, May 1988, pg. 71.

Problem Statement

The central focus of this study is to evaluate whether software metrics have evolved to a point where their application outside the realm of academia produces real gains, and, if so, whether they can be used to improve the software components of Air Force acquisition programs.

Study Limitations

Software metrics have been designed to measure virtually every aspect of software systems. We have limited this study to a representative sampling of the available metrics, and have divided those we have examined into two categories: (1) predictive metrics, which are designed to provide resource requirement estimates to aid in project scheduling; and (2) quality metrics, which provide guidelines to help improve the quality of the delivered product. These categories are not mutually exclusive, nor do they represent the only possible categories for the available metrics, but they provide a reasonable framework from which to view the benefits which can be gained through currently available software metrics.

Study Methodology

The methodology we employed was to survey the appropriate literature, interview personnel who have recent experience in the development of large computer systems, and combine this data with the relevant experiences of the authors. Chapter Two discusses some of the attributes which software metrics have been designed to measure and the basic rationale for measuring those specific attributes. Chapter Three discusses some of the predictive metrics we examined, and Chapter Four does the same for the quality metrics. Although these discussions necessarily include some details, they are primarily targeted at the intended functions of the metrics rather than the detailed specifics--these are adequately covered in the references. Chapter Five provides a few examples of programs where metrics have been applied and evaluates how well the metrics have worked at achieving their objectives. Chapter Six presents a summary of our results and the conclusions of the study.

CHAPTER TWO

SOFTWARE METRIC OVERVIEW

Why Do We Measure?

The introductory remarks in Chapter One described software metrics as a field which evolved in response to software production and quality problems which were endemic to the software industry. Is there any historical evidence that developing measuring tools will help with these problems? Before answering this question, we will briefly consider one of the classics among computer science textbooks.

One of the classic works in computer science--studied by almost any computer science student since the mid-1970's--is a series of books by Donald Knuth collectively entitled The Art of Computer Programming. Note the word "Art" in the title. Professor Knuth is a highly respected computer scientist. One of the key purposes of his reference books was to inject more scientific discipline into the programming field[13]. He recognized, however, that programming was largely an art--and that programmers behaved accordingly, "crafting" their programs and considering the results their own personal possessions. The need to evolve software development into a strict engineering

---

[13] Knuth, Donald E. The Art of Computer Programming, Volume 1. (Reading, Massachusetts: Addison-Wesley Publishing Company, 1973).

discipline was recognized, but was--and remains today--far from reality.

Knuth's desire to advance software development into the realm of science was shared by the developers of the field of software metrics. Capers Jones, Chairman of Software Productivity Research, Inc. and a noted pioneer in software metrics, has stated:

> Although it is not always appreciated, the great advances in chemistry, physics, and other scientific disciplines in the 19th and 20th centuries were preceded by advances in the measurement of physical attributes and the development of accurate measuring instruments in the 17th and 18th centuries. Indeed, it can almost be said that scientific progress of any kind is totally dependent on the ability to measure quantities precisely.[14]

Tom DeMarco, another noted computer scientist, has stated it more succinctly: "You can't control what you can't measure."[15] This is perhaps the fundamental rationale for the development of software measuring instruments--with the hope that, by formalizing software development, the "art" will become a science, and in the process eliminate the problems addressed in Chapter One. As history proves, however, this can only happen if appropriate measuring tools--or metrics--are developed and applied to the software process.

---

[14] T.C. Jones, "Measuring Programming Quality and Productivity," IBM Systems Journal, Vol. 17, 1978, pg 39.

[15] Tom DeMarco, Controlling Software Projects, (New York: Yourdon Press, 1982), pg 3.

What Do Software Metrics Measure?

Software metrics have been designed to measure virtually every aspect of the software development and maintenance process[16]. Our survey of current literature in this area produced a wide variety of metric classifications, including such areas as code complexity, software entropy, number of defects, function points, communications complexity, logical stability, data complexity, interconnectivity, program readability, control complexity, lines of code, documentation adequacy, requirements execution, branch points, decision points, and numerous other classifications. We could not possibly address every metric--or even every metric category--in this paper. Indeed, a single reference listed over 300 metrics![17] We have therefore divided the metrics into the two general categories mentioned in Chapter One--predictive metrics and quality metrics--and focused our research with these categories in mind.

Predictive metrics are those metrics whose primary purpose is to provide data which will quantify the scope of a software effort. These metrics are sometimes referred to as quantitative metrics, or simply cost estimation. Predictive

---

[16] V. Côté, P. Bourque, S. Oligny, and N. Rivard. "Software Metrics: An Overview of Recent Results," The Journal of Systems and Software, Vol. 8, pp. 121-131.

[17] Côté, pp 123-126.

metrics are an attempt to quantify the resources needed to complete a software project. There is no good way to manage a software product without the capability to accurately predict the costs. Without a good metric, it would be impossible to develop a schedule or evaluate factors that could effect productivity. Thus the bottom line for predictive metrics is that they are tools to assist the manager in the realistic scheduling of software development resources.

Quality metrics are metrics designed to improve the quality of delivered software products. They are designed to be applied throughout the software life cycle, from initial design through the software maintenance phase. This category of metrics is aimed at providing software managers with the tools needed to maintain a high quality product throughout the development cycle. As we will see in Chapter Four, there are many different types of metrics in this category, but there is no general agreement as to which are most important. Indeed, a definition of software quality is not even universally agreed. There is, however, a great deal of literature in this area, with continuing research by both the academic community and the software industry.

Predictive and quality metrics are not exclusive categories. Indeed, several metrics fit both categories, depending upon how they are applied. Metrics generally are

designed to measure specific parameters which can then be used to assess factors which effect the desired measurement category. For example, to measure reliability--a factor which impacts software quality--we might employ metrics that measure code complexity, number of defects, and control complexity. The code and control complexity measures, however, can also be employed as predictive metrics. The next two chapters focus on the factors which impact the selected categories and some of the metrics which are applicable in evaluating these factors.

CHAPTER THREE

PREDICTIVE METRICS

Introduction

Predictive metrics are important to any software development effort constrained by resources. If an organization has an infinite amount of resources, then predictive metrics are not of much value. However, virtually every project has some constraint - typically money and time. During the infancy of software development, it was neither sufficiently large nor complex enough to warrant much management attention. In recent years however, as software development has matured, it has grown in size and complexity and now demands serious attention from management at all levels. Managers must be able to control software development costs and accurately forecast the time and resources required for each project. As stated in Chapter One, the computer industry has not been very good at estimating the size of software. A study done at the Air Force Aeronautical Systems Division illustrates the problem.[18] Figure 3 shows the software growth in five weapon system programs between the estimate at project start and the final size at acceptance testing. The average growth for these five projects is over 100%. It is difficult to manage a program that doubles in size

[18]Philip S. Babel, "Software development integrity program (SDIP)," Paper for Air Force Aeronautical Systems Division, Wright-Patterson AFB, Ohio, p. 1.

during project development. In this era of declining Defense budgets, cost overruns in software development can cause entire projects to be canceled or scaled back.

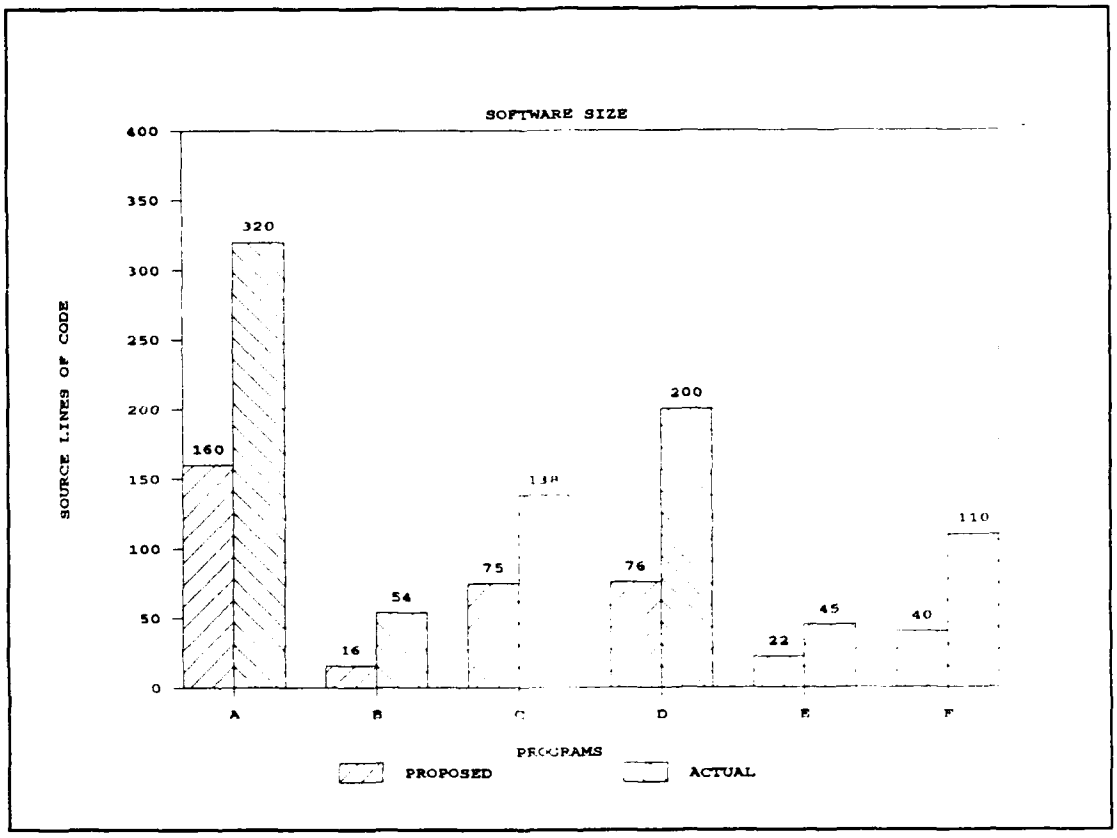


Figure 3

Criteria

There are numerous methods for predicting or estimating software development costs. Most of the methods have their roots in the academic or industrial community or both. Due to the largely mathematical nature of software, the academic

community has long been interested in software metrics. Their interest tends toward very complete and very precise methods. On the other hand, the work from the industrial side tends to stress completeness and precision only to the extent needed to make good management decisions. Both approaches have merit and there is a good deal of overlapping between the two. Before examining some of these methods, it is useful to establish some criteria for evaluation. These criteria, derived from a primarily industrial or managerial standpoint, are:

(1) Easy to use. A good software metric must be easy to use. If a metric is complex and difficult to use, then people will not use it. Long and complex methods increase the chance for error, are difficult to verify, and require considerable resources. As stated in Chapter One, there is a shortage of computer programmers and analysts. Because of this shortage, metrics must be usable by fairly low level analysts.

(2) Consistent with different languages. A good metric should be not be language dependent. In other words, it would work as well for software written in Cobol or Ada. This capability is important because of today's rapidly changing software environment. Twenty years ago, virtually every business applicatio was written in Cobol. The metrics available worked well on that code.

Then, with the introduction of third and fourth generation languages, the situation changed. These new languages gave the developer the to ability to write code that was smaller and more efficient. The old Cobol metrics, however, did not work very well on the new languages. Today, it is not unusual to find applications written in more than one language. With the choice of several languages, often the decision of which language to use may not be made until part way through the development process. Likewise, in responding to software requirement, different contractors could elect to use different languages. It would be difficult to compare these proposals without a consistent metric.

(3) Not size constrained. - The ideal metric should work on large and complex programs as well as small and simple ones. If different metrics are needed for different size programs, then the analyst runs the risk of using the wrong metric and ending up with inaccurate data. Using different metrics also forces the analyst to make an additional determination - size. Will the software program be small, medium, or large? If the metric is used to predict project size, then one ends up with circular reasoning. The analyst must know how big

the software project is before he can select the appropriate software metric to estimate the size.

(4) Usable throughout the development lifecycle. In order to be a truly useful metric, it must be useable during all phases of the development process. Predictive metrics implies the ability to predict what software will cost. The metric must be useable before the development process gets underway. Cost estimates often provide the basis for canceling or going ahead with a project. If the requirements change during the development phase, the manager must be able to use the same software metric to estimate the impact of the changes.

(5) Accurate. A software metric must be accurate to be useful. The Chief Scientist for TRW Defense Systems Group, Barry Boehm, suggests a standard for accuracy. He states that a model is doing well if it can estimate software development costs within 20% of the actual costs, 68 to 70% of the time.[19] While this standard is not as accurate as we might like, it is good enough for decision making and trend analysis.

---

[19] Boehm, Software Engineering Economics, pg. 32.

(6) Meaningful to management. The metric should produce information that is helpful to manager. The most detailed and accurate software metric is useless unless it provides the information the manager needs to make decisions.

Methods

In his book on Software Engineering Economics, Barry Boehm suggests several broad methods of software cost estimation.[20] These methods include two categories: expert judgment and algorithmic models.

Expert Judgment

Expert judgment involves using the judgment of one or more experts, who use their experience to arrive at a cost estimate. These experts often base their estimates on similar projects or past experience. Sometimes the cost estimate is driven by the available resources or by the cost estimate believed to win the job. Expert judgment can be a good estimating tool, but the expert may be biased, optimistic, pessimistic, or unfamiliar with key aspects of the project. These weaknesses can be overcome by using more than one expert. However, the more experts involved, the longer it takes to

[20] Boehm, pg. 329.

complete the estimate. It is difficult to obtain a quick estimate based on group consensus.

How well does expert judgment fit our criteria? First, it is easy to use - for an expert. But, as previously stated, there is a shortage of computer experts. That shortage limits the usefulness of this model. Secondly, if would be difficult to make expert judgment consistent with different languages. Unless the expert was equally familiar with all languages (unlikely), then there would be a natural bias towards a particular language. Additionally, the model tends to work better for small projects. Larger projects require several experts and more time to reach a consensus.

Generally, expert judgment does not satisfy our criteria completely. It is a good model to use if there are experts available, the projects are in one or two languages, and the projects are not very large nor complex.

Algorithmic Models

Algorithmic models provide one or more mathematical algorithms which produce a cost estimate as a function of a number of variables considered to be the major cost drivers.[21] Compared to the expert judgment model, algorithmic models have

[21] Boehm, pg. 330.

a number of strengths. They are objective and not subject to the biases of an expert. They use mathematical relationships, which make them repeatable. Given the same data, the model will always provide the same estimate. There are many good algorithmic models available. The majority are based, to one degree or another, on the measurement of source lines of code (SLOC). Some examples of this type of model are: Software Life Cycle Model (SLIM), RCA PRICE-S Model, RCA PRICE-SL Model, and Constructive Cost Model (COCOMO). All provide reasonable cost estimates and each has its own strengths and weaknesses. Of the ones listed, COCOMO is probably the most popular and complete model in use today.

Although each model approaches cost estimation a little differently, their reliance on the SLOC is a weakness which bears discussion.

Source Lines of Code Measure

The single most widespread method for measuring programmer productivity is source lines of code. A programmer writes lines of code to generate programs. The larger the program, the more lines of code. As a programmer becomes more skilled, he produces more lines of code per day. In addition, lines of code are easy to count. A very simple program can total the SLOC in most computer programs.

However, the advent of high-level languages has turned the SLOC measure into a paradox. High-level languages enable the programmer to write software using fewer lines of code. The effect is directly proportional to the level of language. The highest-level languages have the lowest production rates.

Lines of Code as a Productivity Indicator

	Assembler	PL/1	APL
Source lines	100,000	25,000	10,000
Person-Months for:			
Requirements	10	10	10
Design	30	30	30
Coding	115	25	10
Documentation	20	20	20
Integration/testing	25	15	10
Total person-months:	200	100	80
Total cost:	$1,000,000	$500,000	$400,000
Lines of source code per person-month:	500	250	125
Cost per source line:	$10	$20	$40

Figure 4

Figure 4 illustrates the impact of this paradox. The same application was developed using a low level language (Assembler) and two higher level languages (PL/1 and APL). As you can see, the Assembler effort was the most productive in terms of SLOC per person month and APL the least productive. However, if you look at the total cost of the project, the APL effort took less than half the time and half the cost of the

Assembler effort. In addition, the higher-level languages tend to generate code that is easier to maintain and better documented. Clearly, the SLOC measure of productivity is not meaningful when comparing more that one computer language.

In general, the algorithmic models fit our criteria nicely. They are accurate on both large and small projects, most are fairly easy to use, and all provide meaningful management information. The only weakness is their dependence on a measurement of SLOC. As discussed, this dependency makes high-level languages appear less productive - clearly a paradox. This reliance on SLOC also makes these models not very useable in estimating a project in the early stages. One needs to estimate the SLOC before using these models. That estimate is subject to all the biases and preconceptions of the expert, and can produce factually misleading cost data. Both of these weaknesses limit the usefulness of this category of algorithmic models. There is another algorithmic type model that does not rely on SLOC, but on an abstract concept called Function Points.

Function Point Analysis

In 1979, Allen Albrecht of IBM, introduced the Function Point Analysis (FPA) productivity measure. His original objective was:

> ...to define a measure for applications development and maintenance functions that

> avoided the problems inherent in productivity measures in use at that time. In effect, the measure was intended to help managers analyze applications development and maintenance work and to highlight productivity improvement opportunities.[22]

Basically, FPA is an attempt to measure the degree of software functionality provided to the end user. It tries to quantify the system from the external user's point of view. The Function point metric is derived by a weighted formula that consists of five items:

(1) Number of inputs, multiplied by 4

(2) Number of outputs, multiplied by 5

(3) Number of inquires, multiplied by 4

(4) Number of logical data files, multiplied by 10

(5) Number of interfaces, multiplied by 7

These attributes are counted and totalled. That figure is then adjusted either up or down given the presence or absence of 14 applications characteristics. This adjustment is within a range of +/- 25 percent and reflects the estimator's assessment of the complexity of the program. The result is the Adjusted Function Point total. This total becomes the unit of measure for that application. Figure 5 illustrates the steps involved in Function Point Analysis.

---

[22]Allen J. Albrecht, "Function Points helps managers assess applications, maintenance values," Computerworld, 19 (August 26, 1985, Special Report): 20.

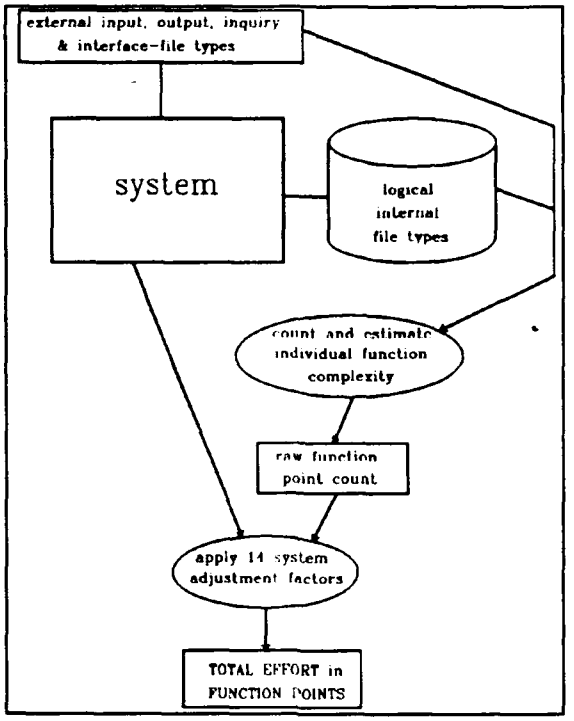


Figure 5

The result will be a quantifiable measure of what a programming team must produce. For example, a particular program may consist of 2,000 function points. There are two ways to relate the function point total to productivity. The first is to measure and track your own development efforts in terms of function points. That will give you a basis for measuring productivity. Capers Jones, chairman of Software Productivity Research Inc., Cambridge, MA, advocates this approach. His study of firms using Function Point Analysis shows that a company is doing well to produce more than 15

function points per staff-month on major projects.[23] However, to effectively use Function Point Analysis you must have one or two years worth of data to establish a baseline. This requires careful tracking and a strong commitment from management.

However, Function Point Analysis can be used without such a historical record. Research has shown a strong degree of equivalency between function points and "SLOC".[24] This relationship allows the user to use Function Point Analysis to estimate SLOC and then use a model like COCOMO to estimate the work effort. This technique can be very helpful in estimating costs early in the development cycle. This two step approach could be used until a base of knowledge is developed for Function Point Analysis.

Another interesting dimension of Function Point Analysis is its ability to measure the power of computer languages. The relationship of Function Points to SLOC varies from language to language. Studies conducted by the Software Productivity Research Inc. indicate that it takes about 105 Cobol source lines of code to produce one Function Point. Figure 6 lists several languages and their function point levels. As you can see from the figure, it takes 71 SLOC of Ada to produce a

---

[23]Capers Jones, "Building a better metric," Computerworld Extra, 22 (June 20, 1988):39.

[24]Allen J. Albrecht and John E. Gaffney, Jr., "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation," IEEE Transactions on Software Engineering, SE-9 (November 1983):639.

function point and only 29 in a fourth generation language like C++. This provides the manager a clear way to measure the increase in productivity associated with higher-level languages.

Function Point Levels for Selected Languages

Language	Source Statements per Function Point
Assembler	320
C	128
COBOL	105
FORTRAN	105
Jovial	105
Pascal	91
PL/1	80
Ada	71
LISP	64
BASIC	64
PROLOG	64
LISP	64
FORTH	64
APL	32
C++	29
SQL	11
Spreadsheet	6

Figure 6

One study found that Cobol applications using traditional methods and unsophisticated tools seldom achieved productivity rates greater than five function points per staff-month--but when more sophisticated tools and languages were used that rate climbed to over 15 function points per staff-month.[25]

[25] Jones, "Building a better metric," pg. 39.

Function Point Analysis offers several important strengths. First, function points can be collected before project start and anytime during the development life cycle. Secondly, they are independent of software development methodology and language used. Third, function points are easy to use. An inexperienced systems analyst can be quickly taught to provide accurate cost estimation using Function Point Analysis. When the concept of Function Point Analysis was first introduced, it was criticized because it lacked a way to quantify the structural complexity of software.[26] This lack of a quantifier could make it less accurate in estimating software requiring complex branching and loops. Subsequent testing has not confirmed this fault and has shown Function Point Analysis accurate with a wide range of projects.

Over the last eleven years, Function Point Analysis has gained rapid acceptance throughout the industry not only as a productivity measurement program, but as an estimating tool. Over 500 major corporations, including firms like IBM, Xerox and Bank of America have adopted the function point method. Some of these companies joined together to establish The International Function Point Users Group (IFPUG) to promote the use of Function Point Analysis and to exchange information. Both IBM and Software Productivity Research offer one day courses on Function Point Analysis. At the Seventh National Conference on

---

[26] Capers Jones, Programming Productivity, (McGraw-Hill Inc., 1986), p. 76.

Measuring DP Quality and Productivity, held on March 15-17, 1989, in Orlando, Florida, Function Points Analysis emerged as the clear winner as a tool for quantifying productivity.[27]

Summary

None of the methods discussed totally satisfied the criteria for a "perfect" predictive metric. Most worked well and produced good results under certain conditions. Function Point Analysis must be highlighted because it offers two unique and important characteristics: the ability to collect function points at any time during the system life cycle, and an independence from software languages and methodology which permit its use for virtually any software project. It also offers the unique ability to evaluate the effect of high-level languages and tools on productivity. In the words of Capers Jones:

> The function-point method is not perfect, but it is the most effective metric yet developed for information systems.[28]

---

[27] Mark Duncan, "What gets measured gets done," System Development, 9 (June 1989):3.

[28] Capers Jones, "Function Point Metrics: Key to Improved Productivity," InformationWEEK, February 23, 1987, p. 27.

CHAPTER FOUR

QUALITY METRICS

Introduction

The predictive metrics discussed in Chapter 3 are useful in determining the resources needed to implement a software project, but do not specifically address the problem of software quality. While it is indeed important to predict the scope of a software product to enable proper allocation of resources, it is equally important to control the quality of the resulting software product. A project completed on time and within budget is of little consequence if the product cannot be effectively used.

The majority of the cost of major software projects has historically been borne in the maintenance phase--typically at least 40 percent and sometimes as much as 70 percent for major systems.[29] Much of this can be attributed to quality problems in the software product--either in the delivered code or in the underlying requirements specifications or design documentation. A major side effect of this statistic is that much software is so unreliable when completed that it is simply not used. Figure 7 shows the results of a study which clearly indicates

---

[29]Grady Booch, Software Engineering with Ada, (Menlo Park, California: Benjamin/Cummings, 1983), pg. 7.

the magnitude of this problem--note that less than two percent of the software delivered on nine government contracts could be used effectively![30]

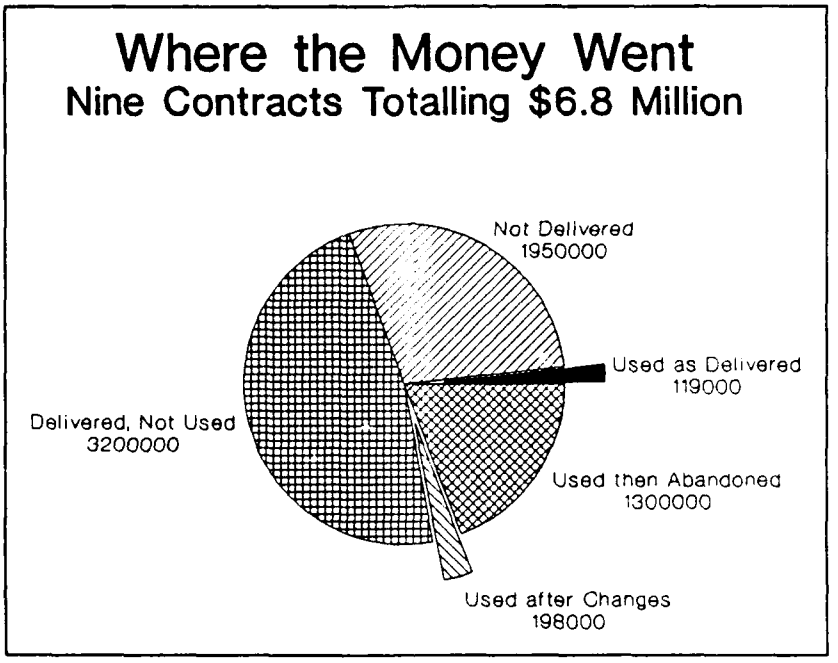


Figure 7

Consequently, there has been substantial interest in both the academic and industrial communities in developing measures which can be applied during the software life cycle to improve the quality of software products--thus reducing the maintenance cost and eliminating the waste exhibited by Figure 7. This chapter examines metrics primarily targeted at improving software quality and the attributes that they measure.

---

[30]"Contracting for Computer Software Development--Serious Problems Require Management Attention to Avoid Wasting Additional Millions," GAO Report to the Congress of the United States, 9 November 1979, pg. 11.

What is Software Quality?

Software quality seems like an obvious concept, but when we attempt to precisely define it we encounter some difficulty. Definitions in the literature vary from author to author, and indeed some authors carefully avoid trying to define it. Tom DeMarco, a noted computer scientist, defines software quality as "...the absence of spoilage,"[31] where spoilage is defined as "...effort dedicated to diagnosis and removal of the faults that were introduced during the development process."[32] He presents the data shown in Figure 8, based on industry averages, to show how much effort could be saved by eliminating spoilage--which can be accomplished by engineering quality software.

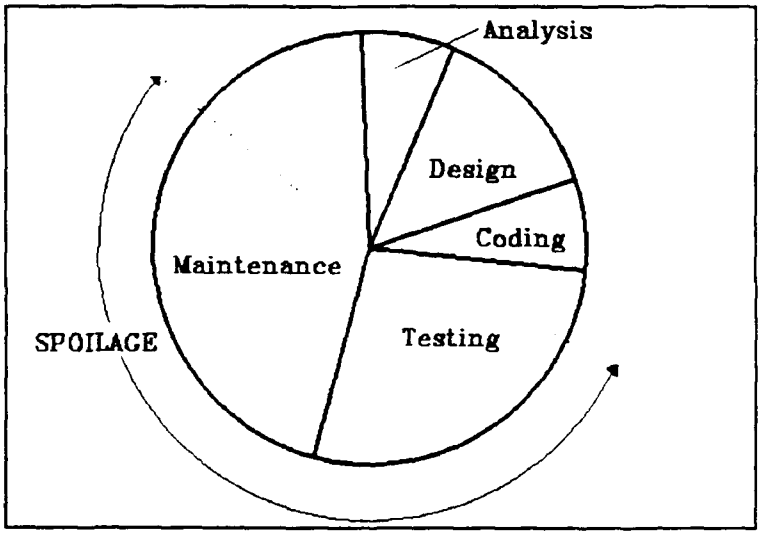


Figure 8

[31] DeMarco, pg. 200.

[32] DeMarco, pg. 199.

DeMarco's definition is not shared by all authors, but the general concept--that quality software must be correct, comprehensible, and reliable--is generally agreed. Many authors share the ideas initially proposed by Barry Boehm, who defines quality software as software which exhibits the following general characteristics, or quality factors:[33]

- Reliability
- Portability
- Efficiency
- Human Engineering
- Testability
- Understandability
- Modifiability

Almost every author agrees with this list of quality factors or some variation on them. For purposes of this paper, we will consider what is meant by each of these factors and briefly discuss the metrics which can be used to measure them.

Quality Factors

As described above, most authors cite a set of quality factors which are essentially supersets of those listed by Barry Boehm. The lists generally contain between ten and fifteen factors, and there are, understandably, many duplications in various authors' lists. Rather than list all of the variations, we have limited the following discussion to Boehm's original seven, but have used definitions from various authors to make

[33] B. W. Boehm, J.R. Brown, and M. Lipow. "Quantitative evaluation of software quality." Proceedings of the Second International Conference on Software Engineering. (1976): 592-605.

each factor as clear as possible. The references given are to the author whose definition most closely matches that we have used for the specified factor.

(1) Reliability. Reliability refers to the degree that the software operates without errors. This is different than hardware reliability, since software does not break like physical things do. Virtually all large software systems contain errors. They can, however, still be reliable, depending on the severity of the errors and the probability of occurrence. A misspelled word in an operator message, for example, is generally not a significant error and would have no affect on system performance. One of the best definitions of reliability we encountered was:

> Software reliability is the probability that the software will execute for a particular period of time without a failure, weighted by the cost to the user of each failure encountered.[34]

(2) Efficiency. Efficiency refers to the execution efficiency and storage efficiency of the delivered code. Simply stated, efficiency is "...the amount of computing resources and code required by a program to perform a function."[35]

---

[34]Glenford J. Myers, Software Reliability, (New York: John Wiley and Sons, 1976), pg. 7.

[35]Jim A. McCall, Paul K. Richards, and Gene F. Walters, Factors in Software Quality, Volume 3, RADC Report TR-77-369, November 1977, pg. 2-3.

(3) Human Engineering. Human engineering represents the design of the interfaces between the software and the user. Quality human engineering refers

> ...collectively to all the attributes that make this interface more palatable: ease of use, error protectedness, quality of documentation, uniform syntax, etc.[36]

These interfaces should be fully specified in the development specifications. Inputs and outputs should be self-explanatory, easy to learn and understand, unambiguous, and designed to avoid misinterpretation.

(4) Portability. Portability refers to the ease with which software can be moved to a different computer. This is important for software with a long anticipated life cycle, or which must be used in many different places which do not have the same computer hardware. Portability is greatly enhanced by the use of higher order languages. One of the fundamental reasons for the development of Ada was module reusability, which is a subset of portability (which generally refers to the entire software system rather than its components), and probably the most important aspect of portability as applied to Air Force weapon systems.[37]

---

[36] Jim A. McCall, Paul K. Richards, and Gene F. Walters, Factors in Software Quality, Volume 1, RADC Report TR-77-369, November 1977, pg. A-9.

[37] Grady Booch, "Reusable Software Components." Defense Electronics. Volume 19, No. 5 (May 1987): S58-S59.

(5) Testability. Testability refers to the ability of the design and code to support evaluation of its performance. In general, well-stated performance requirements will result in testable software. One point frequently noted throughout the literature is that testability does not mean we can test to the point of complete validation. All large software systems will contain some "bugs," and we cannot expect to completely verify correctness. The goal of testing is to "...assure that the probability of failure due to hibernating bugs is sufficiently low to be acceptable."[38] Testability refers to the ability to support tests which will allow this level of confidence, and clearly varies based on the particular system--with very high requirements for major Air Force weapons systems.

(6) Understandability. Understandability is a characteristic of the design and code that makes its purpose and functions easy to learn and follow. It is specified through programming standards which include such features as program commenting requirements, naming

[38]Boris Beizer, Software Testing Techniques, (New York: Van Nostrand Reinhold Company, 1983), pg. 14.

conventions, limited control structures, and the use of high order languages.[39]

(7) Modifiability. Modifiability is a characteristic of the design and code that makes it easy to change. It is a difficult characteristic to specify and evaluate because objective measures of modifiability are not available during the design and development stages. However, this is a very important factor--"The first requirement of a large system of software is that it be built so that it is easy to change."[40] Fortunately, structured programming techniques include features, such as modularity and cohesiveness, which enhance modifiability. Software modifiability is particularly crucial in software intensive Air Force weapons systems. Recent Air Force initiatives have placed strong emphasis on this requirement, recognizing its importance in quality software systems.[41]

---

[39]George Neil and Harvey I. Gold, Software Acquisition Management Guidebook: Software Quality Assurance, ESD Report TR-77-255, August 1977, pg. 88.

[40]Joseph M. Fox, Software and Its Development, (Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1982), pg. 75.

[41]Debra L. Haley, "Software Supportability--A Quality Initiative," Air Force Journal of Logistics, Vol XIII, No 1 (Winter 1989): 22-28.

Criteria for Measuring Quality Factors

The previous section described a set of factors which will be exhibited by quality software. While some authors propose metrics for direct measurement of these factors, most consider the factors too general for direct measurement and break them down into components which can then be measured. Figure 9 shows the resulting hierarchy.[42]

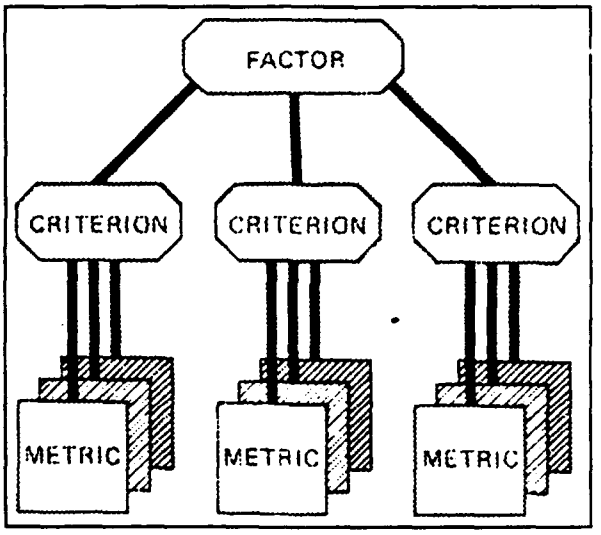


Figure 9

The individual criterion are not unique to specific quality factors. For example, one criterion frequently mentioned is structuredness, which impacts four of the seven listed factors. The specific criteria applicable to the quality factors, like

---

[42]Michael W. Evans and John J. Marciniak. Software Quality Assurance and Management, (New York: John Wiley and Sons, 1987), pg. 160.

the quality factors themselves, vary from author to author. In the following paragraphs we discuss a representative set of criteria for each of the quality factors, and briefly outline the applicable metrics.

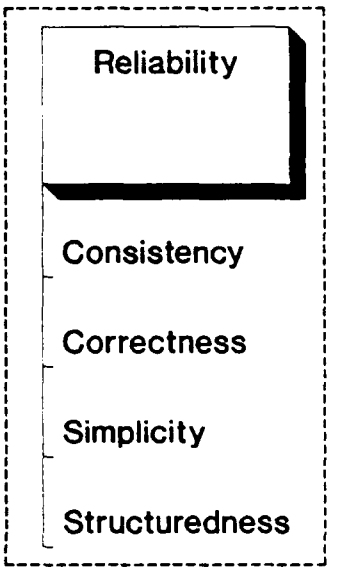


Figure 10

The criteria which apply to the reliability factor are shown in Figure 10. Correctness means the ability of the software to produce the specified outputs when given the specified inputs.[43] This can be objectively measured through software compliance audits, defect measures (errors per thousand lines of code and similar measures) and the testing process. Note the distinction here between the use of testing as a metric and the factor of testability, which has a somewhat different context (and will be discussed later). Consistency refers to uniform standards for notation, symbology, terminology, and comments--in effect a measure of the uniformity and cohesiveness of the software design. Software exhibiting consistent design and documentation characteristics is less likely to contain errors and is easier to maintain. Several consistency metrics have been automated,[44] and some of the modern software development aids, particularly CASE tools,

---

[43] Evans, pg. 163.

[44] Côté, pp. 125-126.

provide an automated enforcement of consistent development practices. Simplicity refers to the use of straightforward algorithms and software structures in the implementation. Historically, reliability has correlated well with code simplicity. Simplicity is the opposite of code complexity, and is often measured by mathematical complexity measures, the most popular of which are McCabe's cyclomatic complexity measure and Halstead's volumetric quality measures.[45] Structuredness refers to the degree to which the code exhibits a structured pattern of organization of its elements. Several automated metrics have been developed to measure the degree to which software conforms to commonly accepted structured programming practices.[46]

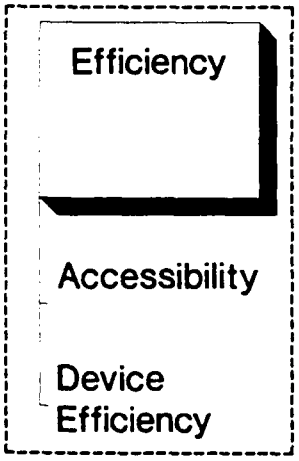


Figure 11

Figure 11 shows some of the criteria which can be used to measure efficiency. Accessibility refers to the ability to selectively--and intentionally--use specified parts of the software code or data elements.[47] This is not a performance characteristic, but is a design element which tends to be exhibited by efficient code. It also facilitates the measurement of efficiency, as it is difficult to assess the efficiency of a module if its

[45] Evans, pp. 158-160.

[46] Côté, pp. 125-126.

[47] Neil, pg. 86.

performance cannot be effectively isolated. Device efficiency refers to the optimized use of the actual computer hardware--the central processing unit, memory, and peripherals. This element can be precisely measured, at the system level, with hardware performance monitors, storage efficiency analysis techniques, and other well-established tools. Other metrics used to assess efficiency include similar software programs ("benchmarks"), analysis of data requirements against actual storage used, and several specialized metrics which measure such things as floating point efficiency, looping efficiency, and other specific code elements.

The elements which characterize the human engineering factor are shown in Figure 12. Accessibility is discussed above, but, in the context of human engineering, also refers to the ease with which specific modules can be accessed from the external user interface. Communicativeness is a feature of software which enhances the understandability of its functions--the inputs and outputs.[48] Software with easily understood inputs and outputs is easier for a user to understand and use. Usability refers to the ease-of-use characteristics of software. In some respects this is perhaps the most important

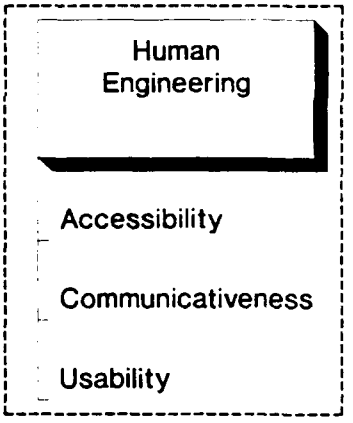


Figure 12

[48] Neil, pg. 86.

feature of a software-based system. Clearly a difficult-to-use system will not be well received, and may not be used. We should not forget the lessons shown in Figure 7. A general officer--or a fighter pilot who must use a system quickly and efficiently--must not be expected to learn and use complicated access procedures.

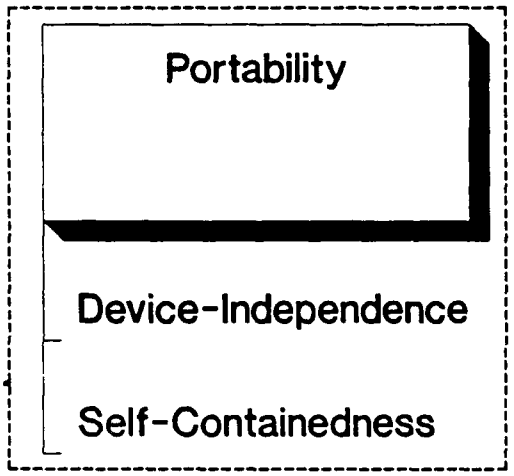


Figure 13

Figure 13 shows the criteria associated with the factor of portability. Self-containedness refers to the degree to which individual software modules are independent, i.e. the modules can perform all of their functions without using code in other modules. This enhances portability by allowing any required redesign to be isolated to individual modules. Device independence refers to the freedom of the code from dependence on specific hardware elements. Device independent code is not affected by changes to the computer hardware or peripheral equipment. To maximize device independence, code that directly relates to specific hardware devices should be minimized and isolated to specifically identified code segments. Portability

measures include the implementation language, modularity metrics, and machine independence.

Testability is a function of accessibility, self-containedness, self-descriptiveness, and structuredness, as shown in Figure 14. Accessibility, self-containedness, and structuredness are discussed above. It is clear that software with easy access to the functioning of self-contained modules which are well structured is much easier to test than software which does not have these properties. The fourth property, self-descriptiveness, refers to the degree to which the actual program code, together with its program comments, allows a software engineer to understand the structure, processing flow, and design.[49] This is a subjective judgment, but can be reasonably well assessed by an experienced software engineer who has had experience in the applications area appropriate for a specific software system.

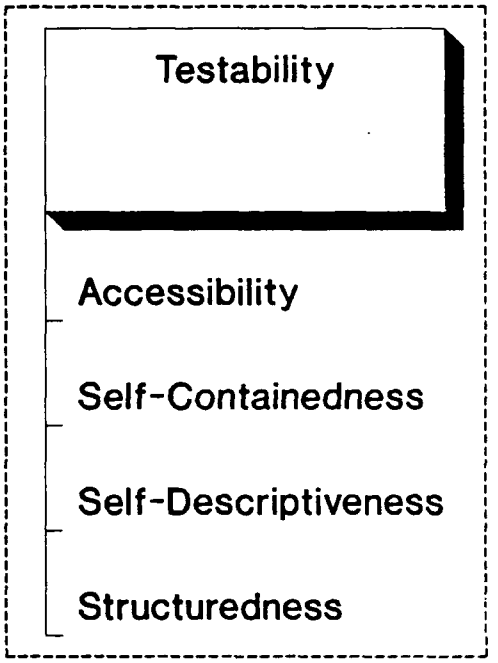


Figure 14

[49] Neil, pg. 88.

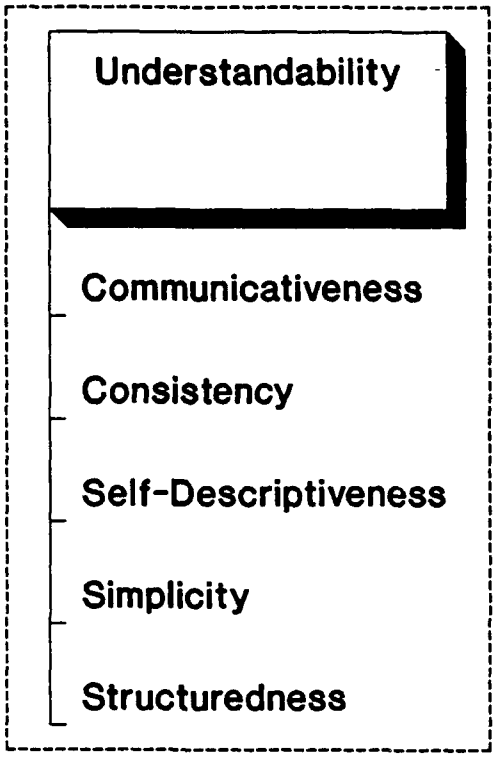


Figure 15

Figure 15 shows the criteria which relate to understandability. These factors, which have all been discussed above, combine to help make software understandable. Some authors include factors such as conciseness (the absence of redundant or excessive code) and readability (easily understood comments which allow quick isolation of functional code segments), but we feel these essentially duplicate the simplicity and self-descriptiveness measures. Software that is consistent in its design, communicates its functions well, is self-descriptive and well structured in its design, and that is not overly complex in its structure will be understandable. The metrics described above can be used for the individual factors, and a reasonably objective evaluation of understandability can then be made.

Figure 16 shows the criteria for the final factor--modifiability. Self-containedness, simplicity, and structuredness are important features of easily modified code, and have all been discussed already. In addition to evaluating

modifiability based on measures of the individual factors, the function point metrics discussed in Chapter Three are often used as a measure of modifiability--by simply measuring the number of maintenance function points per month which the maintenance programming team is able to produce.

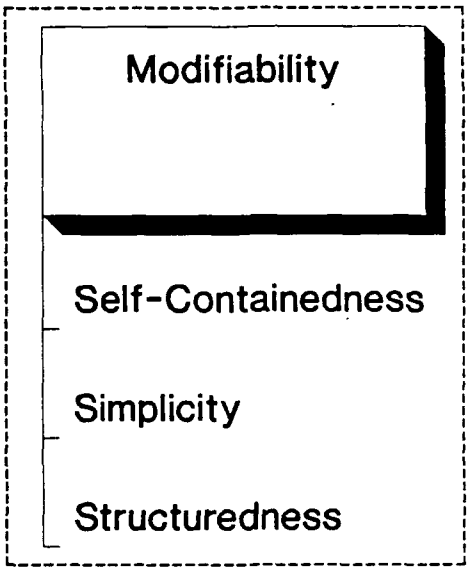


Figure 16

The criteria discussed above are not, as mentioned previously, all-inclusive. Different authors use varying subsets or supersets to apply to their own particular list of quality factors. The criteria we have listed do, however, represent a good cross-section of the literature and should make it obvious that certain criterion are more important than others--most notably simplicity, structuredness, accessibility, and self-containedness, each of which impact at least three of the quality factors. Not surprisingly, these are also areas where there are a reasonable number of automated metrics, which are detailed in the references for the above paragraphs.

Summary

Throughout our research one point became increasingly clear--quality metrics are almost universally subjective measures. While the situation is not as bad as it was in the late 1970's--when a study of quality metrics concluded "...the state-of-the-art for determining software quality is ... through subjective evaluation."[50]--it is still largely so.

This clearly conflicts with the frequently stated tenet that metrics must be "measurable, independent, accountable, and precise."[51] It does not, however, mean that quality measures cannot be useful. A metric can be measurable, independent, and accountable--and at least somewhat precise--despite its subjectivity. Further, some metrics which satisfy all four of the above requirements have proven very useful in assessing software quality--notably complexity measures, structuredness tests, and the use of function point analysis for measuring modifiability.

The specific metrics to use depend on which quality factors are considered most important, which will vary depending on the application for the target system. What is clear is that quality must be considered throughout the development life

---

[50] Neil, pg. 88.

[51] DeMarco, pg. 50.

cycle, and that quality metrics provide the measurement tools needed to ensure the software product is meeting quality goals. It is important to recognize that quality metrics do not produce quality--they measure it. Thus software managers must ensure the development environment supports quality engineering practices by providing adequate development tools, automated configuration management support, and strong management support for quality. In other words, the development environment must be managed like any other engineering discipline, with quality one of the key considerations. The quality metrics described in this chapter can then be used to provide feedback to management to let them know how well they have achieved the quality goals.

CHAPTER FIVE

APPLICATIONS OF METRICS

Introduction

The predictive and quality metrics discussed in the preceding chapters appear, in theory, to be very useful tools to control software costs and improve productivity. How well have they performed in actual practice? In this chapter we will examine a few examples of systems which have used metrics to control the software factor and see how well they have done.

Examples

One example of a software metrics application comes from Alan Albrecht, the creator of the Function Point Analysis. His original research on productivity at IBM involved 22 projects completed over a five year period. By using the Function Point Analysis metric, he reported a productivity gain of about 3 to 1. He cited several factors as the chief causes of the improvements: the use of structured programming, the use of high-level languages, the use of on-line development and the use of a programming development library.[52] We note that the improvements were not attributed to the use of metrics; however, without a measuring tool--e.g. the function point metrics--it

[52] Capers Jones, Programming Productivity, McGraw-Hill Inc., 1986, p. 75.

would not have been possible to assess the productivity gains noted in the study.

Software engineer Barry Boehm reported similar results with projects he studied at TRW. He found that by using software metrics:

> most data processing installations can increase their software development and maintenance productivity by an additional 100% in three to four years and by and additional 400% in six to eight years.[53]

Here the productivity gain is directly attributed to the use of metrics, although the reference does note that this is in addition to other procedures a software development organization should adopt--such as structured programming techniques.

A more recent example comes from Steve Drummond the productivity coordinator for Hallmark Cards Inc. of Kansas City, Missouri. Several years ago, he began to measure software development efforts using function points. At first, the results were discouraging. He found substantial differences in the number of function points per man-day among different projects. However, as he continued to collect and analyze data, several trends emerged. The first trend was that lack of staff experience greatly contributed to a lower output. Secondly, projects that reused existing code were very productive. Third, large projects using traditional techniques were less productive

---

[53] Boehm, Software Engineering Economics, pg. 641.

than smaller ad hoc projects produced by the information center. He was able to use these and other trends to identify opportunities to improve productivity. The more data he collected, the better his understanding of the factors that influenced software development.[54] Again we see that metrics contributed to significant gains by allowing the manager to understand and measure what was occurring.

The Software Engineering Laboratory at NASA's Goddard Space Flight Center has conducted several experiments with software measurements in the flight dynamics area. Their conclusions have been overwhelmingly positive, including such statements as "...has resulted in a major improvement to the understanding and overall process of developing software in this environment,"[55] "...many valuable results can be produced via the measurement process,"[56] and, perhaps most telling, "Without software measurement, success or failure in developing software systems may be random."[57]

---

[54]Steve Drummond, "Measuring Applications Development Performance," Datamation, 31 (February 15,1985):108.

[55]Jon D. Valett and Frank E. McGarry, "A Summary of Software Measurement Experiences in the Software Engineering Laboratory," The Journal of Systems and Software, Vol. 9, Number 2 (February 1989): 146.

[56]Valett, pg. 146.

[57]Valett, pg. 147.

A precautionary note is in order, however. It must be clear what the objective of the metric program is. An early study of systems which emphasized hardware efficiency--which can be easily measured objectively and was therefore perhaps over-emphasized--demonstrated the results of improper application of metrics. The result was a threefold increase in software development cost because the developers were attempting to maximize the hardware utilization.[58] Errors of this magnitude are not likely today, since the hardware/software cost tradeoffs are much better understood--but the important lesson is not the specific example, but the consequences of the misapplication of a metric.

Costs

From these examples, we see that software metrics can be useful tools to improve productivity. But, it costs money to implement a software metric program, as this requires both the collection and the analysis of the data being measured. How much does this cost--and is it worth it?

Software expert Tom Demarco suggests that software metrics should cost from five to ten percent of the manpower

---

[58] Rein Turn, Hardware-Software Tradeoffs in Reliable Software Development. TRW Systems Engineering and Integration Division Paper TRW-SS-77-03, November 1977, pg. 2.

cost of the effort monitored.[59] At the beginning, the cost will be at the high end of that range because of initial training, setting up procedures, and working by trial and error. Once well established the cost should be closer to five percent.

The NASA Goddard Space Flight Center research mentioned earlier anticipated a cost of 8 to 10 percent, but, after the initial startup costs, has averaged less than 5 percent, a cost that the researcher's contend "...has been well worth it."[60]

As we have seen from the work of Albrecht and Boehm, the cost savings of a good metric program is somewhere between 50 to 300 percent. If, as De Marco suggests and the NASA experience confirms, the measurement program costs at most 10 percent, then we expect a savings of between 40 and 290 percent. Clearly, software metrics are a good investment.

---

[59] Demarco, pg. 38.

[60] Valett, pg. 146.

CHAPTER SIX

SUMMARY AND CONCLUSIONS

Summary

In his book, The Mythical Man-Month, Fred Brooks likens the development of large software projects to the efforts of prehistoric beasts trying to escape from the tar pits. He concludes his analogy by noting that "Everyone seems to have been surprised by the stickiness of the problem, and it is hard to discern the nature of it."[61]

Software metrics have evolved in an effort to help "discern the nature of it." This paper has briefly examined the types of metrics that are available, the qualities they measure, and what they can accomplish. As the application of computer technology continues to expand in Department of Defense weapon systems, it is imperative that managers understand the need to better control software systems and the role that metrics can play in the control process.

Chapter One focused on the root causes of the software problem--the exponential growth of software-based systems and

---

[61] Fred Brooks, Jr., The Mythical Man-Month, (Reading, Massachusetts: Addison-Wesley Publishing Company, 1978), pg. 4.

the lack of engineering controls on software development--and the need for measurement tools.

Chapter Two provided an overview of current and evolving software metrics--reiterating the need for measurement and providing a brief look at what metrics can measure. The wide variety of metric classifications was discussed, and the focus for this study narrowed to two classes: predictive metrics and quality metrics.

The next two chapters looked at predictive metrics (Chapter Three) and quality metrics (Chapter Four), with a focus on what they can do rather than how they do it. The implementation details of the metrics are beyond the scope of this study, but are amply discussed in the references. We examined what the metrics are designed to do, how well they accomplish these objectives, and the value these measurements have in real-world applications. Two key points that emerged were that lines-of-code is not a good basis for metrics and that, while many quality metrics are largely subjective, others have been developed that provide useful objective measures. Function point measurements stand out for their usefulness in both predictive and quality-measurement applications.

Chapter Five provided a few examples of systems where metrics have been applied. Here we saw that metrics have indeed

proven their usefulness when properly applied--and that when imprudently applied they have little utility.

In summary, we explored software metrics from a broad perspective to analyze their utility as productivity improvement tools--to answer the question stated in the title of this study: are they useful tools or wasted measurements?

Conclusions

It seems clear that software metrics are not wasted measurements but are, indeed, useful tools. Like any other tool, however, they must be handled carefully and applied properly or they can do more harm than good. Software managers must ensure that appropriate metrics are applied to their programs and must pay attention to the results. Software development is not going to get easier, but proper application of the available tools--including metrics--can let us do a better job. As Fred Brooks has said:

> The tar pit of software engineering will continue to be sticky for a long time to come... The management of this complex craft will demand our best use of new languages and systems, our best adaptation of proven engineering management methods, liberal doses of common sense, and a God-given humility to recognize our fallibility and limitations.[62]

---

[62] Brooks, pg. 177.

Recommendations

The Air Force should fund continuing research and development efforts aimed at further refinement and development of metrics which will keep pace with the evolving hardware and software. The most promising of the metrics tend to use criteria other than lines-of-code measures and are consequently less language-dependent than earlier metrics. While language-independence is a desirable trait for a metric, we have seen that this is not always possible. Emphasis must, therefore, be placed on metrics which work well with Ada, since virtually all new Department of Defense weapons systems will be using this language.

We also agree with Tom DeMarco, who suggests setting up a separate "Metrics Group" to do measurements. While he recognizes there is a cost associated with this, he notes that this independence can result in much more useful measurement data. In his words:

> If you let one of the litigants be judge, you won't have to waste much time explaining the facts of the case.... But the judgment might suffer.[63]

Our national defense is too important to let the judgment suffer!

---

[63] DeMarco, pg. 19.

As a footnote, it is interesting to note the close correlation of the quality factors identified in Chapter Four with the specified goals of the Ada language. The Ada "Steelman" document identified eight criteria for design of the language: generality, reliability, maintainability, efficiency, simplicity, implementability, machine independence, and complete definition.[64] Whether Ada has achieved these goals--which a previous Air War College study contends it has[65]--is beyond the scope of this paper, but it seems clear that Ada was designed with quality software in mind.

---

[64]Kenneth C. Shumate, Understanding Ada. New York: Harper and Row, 1984, pp. 14-15.

[65]Nicholas J. Babiak, Ada, The New DoD Weapon System Computer Language--Panacea or Calamity, Air War Defense Analytical Study, May 1989, pp. 62-67.

BIBLIOGRAPHY

Albrecht, A. J. and J.E. Gaffney Jr. "Software Function, Source Lines of Code and Development Effort Prediction," IEEE Transactions on Software Engineering, November 1983, pp. 639-648.

Albrecht, A. J. "Function Points Helps Managers Assess Applications, Maintenance Values," Computerworld, 19, special report (26 August 1985) pp. 20-21.

Babel, Philip S. "Software Development Integrity Program (SDIP)," Paper for Air Force Aeronautical Systems Division, Wright-Patterson AFB, Ohio, 1988.

Babiak, Nicholas J. Ada, The New DoD Weapon System Computer Language--Panacea or Calamity, Air War College Defense Analytical Study, May 1989.

Baker, Caleb and David Silverberg. "Boeing Scrambles To Correct Flaws in Peace Shield," Defense News, 4 (December 18, 1989):1,28.

Basili, V.R. and K. Freburger. "Programming Measurement and Estimation in the Software Engineering Laboratory," Journal of Systems and Software, 2 (1981):47-57.

Behrens, Charles. "Measuring Productivity of Computer Systems Development Activities with Function Points," IEEE Transactions on Software Engineering, SE-9, Number 6 (Nov 1983):648-652.

Beizer, Boris. Software Testing Techniques. New York: Van Nostrand Reinhold Company, 1983.

Boehm, B.W., J.R. Brown, and M. Lipow. "Quantitative evaluation of software quality." Proceedings of the Second International Conference on Software Engineering. (1976): 592-605.

Boehm, Barry W. and Philip N. Papaccio. "Understanding and Controlling Software Costs, IEEE Transactions on Software Engineering, 14 (Oct 1988):1462-78.

Boehm, Barry W. Software Engineering Economics, Englewood Cliffs, N.J. Prentice-Hall, 1981.

Booch, Grady. "Reusable Software Components." Defense Electronics. Volume 19, No. 5 (May 1987): S53-S59.

Booch, Grady. Software Engineering with Ada. Menlo Park, CA: Benjamin/Cummings, 1983.

Borden, Andrew G. "The Impact of Advanced Computer Systems on Avionics Reliability." Defense Electronics. Volume 19, No. 5 (May 1987): S7-S21.

Borkovitz, P. E. "Eliminating Bugs from Weapon System Computer Programs." Military Technology. Volume 5, No. 88 (May 1988): 71-72.

Boydston, R.E. "Programming Cost Estimate - Is it reasonable?," Proceedings 7th International Conference on Software Engineering, 1984, pp. 153-159.

Brooks, Frederick P., Jr. The Mythical Man-Month. Reading, Massachusetts: Addison-Wesley Publishing Company, 1975.

Brown, B.R., H. Herlich, M.D. Emerson, C.L. Williamson, M.V. Greco, W. Sherman. Productivity Measurement in Software Engineering, SSA/STECS/PRODUCTIVITY-83, U.S. Social Security Administration, Washington D.C., 1983.

Buckley, Fletcher J. "Standard Set of Useful Software Metrics is Urgently Needed," Computer, 22, (July 1989):88-90.

Bullen, Richard H. Jr. Engineering of Quality Software Systems: Software First Concepts. Mitre Corporation report RADC-TR-74-325, Volume III, Rome Air Development Center, January 1975.

Canan, James W. "The Software Crisis," Air Force Magazine, 69 (May 1986):46-52.

Card, David N. "Major obstacles hinder successful measurement," IEEE Software, 5 (Nov 88):82-84.

Cheng, L. L. Engineering of Quality Software Systems: Some Case Studies in Structured Programming. Mitre Corporation report RADC-TR-74-325, Volume VI, Rome Air Development Center, January 1975.

Clapp, Judith A., and Leonard J. LePadula. Engineering of Quality Software Systems. Mitre Corporation report RADC-TR-74-325, Volume I, Rome Air Development Center, January 1975.

Clark, Gregory A. Software Cost Estimation Models - - Which One to Use?, Air Command and Staff College student paper, Air University, Maxwell AFB, Alabama, 1986.

Conte, S.D., H.E. Dunsmore, V.Y. Shen, Software Engineering Metrics and Models, Menlo Park, California, Benjamin/Cummings, 1986.

Côté, V., P. Bourque, S. Oligny, and N. Rivard. "Software Metrics: An Overview of Recent Results," The Journal of Systems and Software. Volume 8 (1988): 121-131.

De Marco, Tom. Controlling Software Projects, New York, Yourdon Press, 1982.

Doggett, R. B., M. P. Kress, and K. I. Mehrer. Measuring and Reporting Software Status. Boeing Aerospace Company Document ASD-TR-78-49, Aeronautical Systems Division, August 1978.

Drummond, .Steve. "Measuring Applications Development Performance," Datamation, 31 (February 15, 1985): 102-108.

Duncan, Mark. "What Gets Measured Gets Done," Systems Development, 9 (June 1989):1-4.

Dunn, Robert and Richard Ullman. Quality Assurance for Computer Software. New York: McGraw-Hill Book Company, 1982.

Dunsmore, H.E. "Software Metrics: An overview of Evolving Methodology," Information Processing and Management, 20 (1984):183-192.

Evans, Michael W. The Software Factory: A Fourth Generation Software Engineering Environment. New York: John Wiley and Sons, 1989.

Evans, Michael W. and John J. Marciniak. Software Quality Assurance and Management. New York: John Wiley and Sons, 1987.

Feuche, Mike. "Attention is being generated by complexity metric tools," MIS Week, 9 (February 29, 1988): 27-29.

Firesmith, Donald G. "Should the DoD Mandate a Standard Software Development Process?" Defense Science and Electronics. Part 1: Volume 6, No. 4 (April 1987): 60-64; Part 2: Volume 6, No. 7 (July 1987): 56-59.

Fleischer, R. J. Engineering of Quality Software Systems: Effects of Management Philosophy on Software Production. Mitre Corporation report RADC-TR-74-325, Volume II, Rome Air Development Center, January 1975.

Fox, Joseph M. Software and Its Development. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1982.

Gaffney, J.R. Jr. "The Impact on Software Development Costs Using HOL's," IEEE Transactions on Software Engineering, SE-12 (1986):496-499.

Gear, C. William. Computer Organization and Programming. New York: McGraw-Hill Book Company, 1980.

Gerhardt, Mark S. "Don't Blame Ada." Defense Science and Electronics. Volume 6, No. 8 (August 1987): 53-68.

Glass, Robert L. "Software Metrics: Lightning Rods and Built Up Tension," Systems Development, 9 (March 1989):10.

Grey, Baron O. A. "Making SDI Software Reliable Through Fault-Tolerant Techniques." Defense Electronics. Volume 19, No. 8 (August 1987): 77-86.

Haley, Debra L. "Software Supportability--A Quality Initiative," Air Force Journal of Logistics. Vol XIII, No 1 (Winter 1989): 22-28.

Harrison, Warren. "Metrics Workshop Serves Practitioners, Academics," IEEE Software, 6 (May 1989):98-100.

Harvey, David. "Computer Science Corporation's Star*Lab." Defense Science and Electronics. Volume 6, No. 4 (April 1987) 69-71.

Hernandez, Capt Richard J. "Logistics Considerations of Applications Software Testing," Logistics Spectrum. Vol 22, Issue 4 (Winter 1988): 3-6.

Hughes, David. "Boeing Told To Solve Peace Shield Problems," Aviation Week & Space Technology, December 18, 1989, p. 114.

"Is Software Complexity Slowing the Computer Industry?" Byte. Volume 13, No. 12 (November 1988): 11.

Johnson, James R. "The Eight Myths of Measuring Software Development Performance," Mainframe, 4 (March 1989):29-32.

Jones, Capers. "A New Look at Languages," Computerworld, 22 (November 7,1988):97-103.

Jones, Capers. "Building a Better Metric," Computerworld, 22 Extra (June 20, 1988):38-39.

Jones, Capers. "Function - Point Metrics: Key to Improved Productivity," InformationWEEK, Issue 105, February 23, 1987, pp. 26-27.

Jones, Capers. "How not to Measure Programming Productivity," Computerworld, 20 (13 January 1986):65-66,70.

Jones, Capers. Programming Productivity, New York, McGraw-Hill Inc, 1986.

Jones, T.C. "Measuring Programming Quality and Productivity," IBM Systems Journal, 17 (1978):39-63.

Kitchenham, B.A. and N.P. Taylor, "Software Project Development Cost Estimates," Journal of Systems and Software, 5 (1985):267-278.

Kitchenham, Barbara. "An Evaluation of Software Structure Metrics," IEEE Computer Software & Applications Conference Proceedings, 1988, pp. 369-76.

Knaft, G.J. and J. Sacks, "Software development effort prediction based on Function Points," Proceedings of the Computer Software and Applications Conference, 1986, pp. 319-325.

Knuth, Donald E. The Art of Computer Programming, Volume 1. Reading, Massachusetts: Addison-Wesley Publishing Company, 1973.

Kress, M. P. Software Quality Assurance. Boeing Aerospace Company Document ASD-TR-78-47, Aeronautical Systems Division, January 1979.

Kulkarn, Anriudh, "A generic Technique for developing a Software Sizing and Estimate Model," IEEE Computer Software & Applications Conference Proceedings, 1988, pp. 155-61.

LaPadula, Leonard J. Engineering of Quality Software Systems: Software Reliability Modeling and Measurement Techniques. Mitre Corporation report RADC-TR-74-325, Volume VIII, Rome Air Development Center, January 1975.

Lennselius, Bo., Claes Wohlin, Ctirad Vrana, "Software Metrics: fault content estimation and software process control," Microprocessors and Microsystems, 11 (September 1987):365-376.

Linn, Randy K. and K. Vairavan. "An experimental investigation of software metrics and their relationship to software development effort," IEEE Tranactions on Software Engineering, 15 (May 1989): 649-654.

Ludlum, David A. "Measuring DP Efficiency, Quality," Computerworld, 20 (11 August 1986):71,74.

Marcus, David J. "Project Bold Stroke," Signal Magazine, April 1986, pp. 100-101.

McCabe, T.J. "A Complexity Measure," IEEE Transactions on Software Engineering, SE-2 (December 1976):308-320.

McCall, Jim A., Paul K. Richards, and Gene F. Walters. Factors in Software Quality. General Electric Company RADC-TR-77-369, Rome Air Development Center, November 1977.

"Measuring DP quality and Productivity," Systems Development, 8 (July 1988):4-8.

Morrocco, John. "Coming Up Short in Software." Air Force Magazine. Volume 70, No. 2 (Feb 1987): 64-69.

Musa, John D. "Faults, failures and a metrics revolution," IEEE Software, 6 (March 1989):85-87.

Myers, Glenford J. Software Reliability: Principles and Practices. New York: John Wiley and Sons, 1976.

Neil, George, and Harvey I. Gold. Software Acquisition Management Guidebook: Software Quality Assurance. System Development Corporation Document ESD-TR-77-255, Electronic Systems Division, August 1977.

Pollack, G.M. and S.Sheppard, A Design Methodology for the Utilization of Metrics Within Various Phases of Software Lifecycle Models, National Technical Information Service, Springfield, VA, 1987.

Randolph, General Bernard P. Speech to Air War College Class of 1990, 13 Dec 1989.

Robach, H. Dieter and Bradford T. Ulery. "Improving Software Maintenance Through Measurement," Proceedings of the IEEE, 77 (April 1989):581-96.

Roman, David. "A measure of Programming," Computer Decisions, 19 (January 26, 1987):32-33.

Schneidewind, Norman F. Software Maintenance: Improvement Through Better Development Standards and Documentation. Naval Postgraduate School Report NPS-54-82-002, February 1982.

Schultz, Herman, P. Software Management Metrics, Technical Report by Mitre, Bedford, MA, 1988.

Schumate, Kenneth C. Understanding Ada. New York: Harper and Row, 1984.

Shepperd, Martin. "An evaluation of Software Product Metrics," Information Software, 30 (April 1988):177-88.

Smith, Maj. Gen. Monroe T. "Project Bold Stroke: A Plan to Cap the Software Crisis." Government Executive. Volume 19, No. 1 (January 1987): 29-30.

Software Metrics, National Technical Information Service, Springfield, VA, November 1988.

Sullivan, J. E. Engineering of Quality Software Systems: Measuring the Complexity of Computer Software. Mitre Corporation report RADC-TR-74-325, Volume V, Rome Air Development Center, January 1975.

Symons, Charles R. "Function Point Analysis: Difficulties and Improvements," IEEE Transactions and Software Engineering, 14 (January 1988):2-11.

Taylor, David A. The Software Crisis and a Senior Leader's Awareness, Air Command and Staff College student paper, Air University, Maxwell AFB, Alabama, 1987.

Thayer, Thomas A., Myron Lipow and Eldred C. Nelson. Software Reliability: A Study of Large Project Reality. New York: North-Holland Publishing Company, 1978.

Turn, Rein. Hardware-Software Tradeoffs in Reliable Software Development. TRW Systems Engineering and Integration Division Paper TRW-SS-77-03, November 1977.

Valett, Jon D. and Frank E. McGarry. "A Summary of Software Measurement Experiences in the Software Engineering Laboratory," Journal of Systems and Software, 9 (February 1989):137-48.

Van Scotter. "Software Estimation Factors and Techniques," Air Force Journal of Logistics, 12:33-35.

Verner, June M. and Gramham Tate. " A Model for Software Sizing," Journal of Systems and Software, 7 (June 1987):173-177.

Weyuker, Elaine. "Evaluating Software Complexity Measures," Software Engineecing, 14 (September 1988):1357-65.

Woodfield, S.N., V.Y. Shen, H.E. Dunsmore. "A Study of Several Metrics for Programming Effort," Journal of Systems and Software, 2 (1981):97-103.