Technical Report
CMU/SEI-92-TR-019
ESC-TR-92-019

# Software Measurement for DoD Systems: Recommendations for Initial Core Measures

Anita D. Carleton
Robert E. Park
Wolfhart B. Goethert
William A. Florac
Elizabeth K. Bailey
Shari Lawrence Pfleeger

**Technical Report**

CMU/SEI-92-TR-019
ESC-TR-92-019
September 1992

# Software Measurement for DoD Systems: Recommendations for Initial Core Measures

Anita D. Carleton
Robert E. Park
Wolfhart B. Goethert
William A. Florac
Software Process Measurement Project

Elizabeth K. Bailey
Institute for Defense Analyses

Shari Lawrence Pfleeger
The MITRE Corporation



**Software Engineering Institute**
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.



This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212.
Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page. The URL is http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: (703) 767-8222 or 1-800 225-3842.]

# Table of Contents

# List of Figures

# Acknowledgments

Since 1989, the SEI has been assisted in its software measurement initiative by a Measurement Steering Committee that consists of senior representatives from industry, government, and academia. The people on this committee have earned solid national and international reputations for contributions to measurement and software management. They have helped us guide the efforts of our working groups so that we could integrate their work with not only this report, but also our other software measurement activities. We thank the members of the committee for their many thoughtful contributions. The insight and advice these professionals have provided have been invaluable:

William Agresti
The MITRE Corporation

Henry Block
University of Pittsburgh

David Card
Computer Sciences Corporation

Andrew Chruscicki
US Air Force Rome Laboratory

Samuel Conte
Purdue University

Bill Curtis
Software Engineering Institute

Joseph Dean
Tecolote Research

Stewart Fenick
US Army Communications-Electronics Command

Charles Fuller
Air Force Materiel Command

Robert Grady
Hewlett-Packard

John Harding
Bull HN Information Systems, Inc.

Frank McGarry
NASA (Goddard Space Flight Center)

John McGarry
Naval Underwater Systems Center

Watts Humphrey
Software Engineering Institute

Richard Mitchell
Naval Air Development Center

John Musa
AT&T Bell Laboratories

Alfred Peschel
TRW

Marshall Potter
Department of the Navy

Samuel Redwine
Software Productivity Consortium

Kyle Rone
IBM Corporation

Norman Schneidewind
Naval Postgraduate School

Herman Schultz
The MITRE Corporation

Seward (Ed) Smith
IBM Corporation

Robert Sulgrove
NCR Corporation

Ray Wolverton
Hughes Aircraft

As we prepared this report, we were aided in our activities by the able and professional support staff of the SEI. Special thanks are owed to Linda Pesante and Mary Zoys, whose editorial assistance helped guide us to a final, publishable form; to Lori Race, who coordinated our meeting activities and provided outstanding secretarial services; and to Helen Joyce and her assistants, who so competently assured that meeting rooms, lodgings, and refreshments were there when we needed them.

And finally, this report could not have been assembled without the active participation and many contributions from the other members of the SEI Software Process Measurement Project and the SWAP measurement team who helped us shape these materials into forms that could be used to support the DoD Software Action Plan:

John Baumert
Computer Sciences Corporation

Mary Busby
IBM Corporation

Andrew Chruscicki
US Air Force Rome Laboratory

Judith Clapp
The MITRE Corporation

Donald McAndrews
Software Engineering Institute

James Rozum
Software Engineering Institute

Timothy Shimeall
Naval Postgraduate School

Patricia Van Verth
Canisius College

# Software Measurement for DoD Systems: Recommendations for Initial Core Measures

**Abstract**. This report presents our recommendations for a basic set of software measures that Department of Defense (DoD) organizations can use to help plan and manage the acquisition, development, and support of software systems. These recommendations are based on work that was initiated by the Software Metrics Definition Working Group and subsequently extended by the SEI to support the DoD Software Action Plan. The central theme is the use of checklists to create and record structured measurement descriptions and reporting specifications. These checklists provide a mechanism for obtaining consistent measures from project to project and for communicating unambiguous measurement results.

## 1. Introduction

In its 1991 Software Technology Strategy [DoD 91], the Department of Defense (DoD) set three objectives to be achieved by the software community by the year 2000:

- Reduce equivalent software life-cycle costs by a factor of two.
- Reduce software problem rates by a factor of ten.
- Achieve new levels of DoD mission capability and interoperability via software.

To achieve these objectives, the DoD needs a clear picture of software development capabilities and a quantitative basis from which to measure overall improvement. With quantitative information, national goals can be set to help keep the entire community competitive and focused on continuous improvement of products and processes. This is not possible today. Few organizations have a comprehensive, clearly defined software measurement program, and measurement is frequently done in different ways. Because there are no standard methods for measuring and reporting software products and processes, comparisons across domains or across the nation are impossible. A US company cannot know if its software quality is better or worse than the national average because no such national information is available. The meters, liters, and grams available as standards in other disciplines are missing, and there is seldom a clear understanding of how a measure on one software project can be compared or converted to a similar measure on another.

The Software Technology Strategy has now been made part of a larger DoD initiative called the Software Action Plan (SWAP). This plan establishes 17 initiatives related to developing and managing software systems. One of its initiatives is to define a core set of measures for use within DoD software projects. The Software Engineering Institute (SEI) was asked to lead this initiative because there was a natural convergence between DoD objectives and work that the SEI already had underway (Figure 1-1).

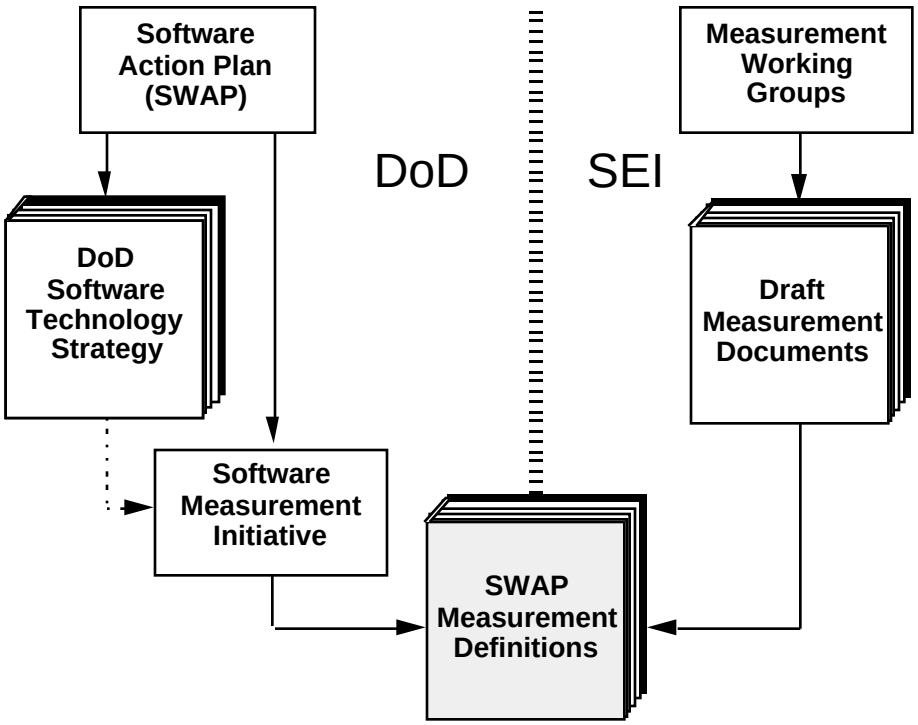


Figure 1-1 Convergence Between DoD and SEI Objectives

| Principal Components of the Software Measurement Strategy Discussed by the SWAP Working Group | |
|---|---|
| Short Title | Subject |
| SEI Core Set | Recommendations for Initial Core Measures |
| STEP | Army Software Test and Evaluation Panel—Software Metrics Initiatives |
| AFP 800-48 | Acquisition Management—Software Management Indicators |
| MIL-STD-881B | Work Breakdown Structures for Defense Materiel Items |
| I-CASE | Integrated Computer-Aided Software Engineering |
| STARS | Software Technology for Adaptable, Reliable Systems |
| CMM | Capability Maturity Model for Software |

Figure 1-2 Proposed SWAP Software Measurement Strategy—Principal Ingredients

The tasks assigned to the SEI were to prepare materials and guidelines for a set of basic measures that would help the DoD plan, monitor, and manage its internal and contracted software development projects. These materials and guidelines would provide a basis for collecting well-understood and consistent data throughout the DoD. They would also support other measurement activities the DoD is pursuing. Figure 1-2 on the facing page lists some of the principal components of the measurement strategy the SWAP Working Group has been discussing. The timelines associated with this strategy are presented in Appendix C.

The memorandum of understanding that initiated the SWAP measurement work called for the SEI to build upon existing draft reports for size, effort, schedule, and quality measurement that had been prepared by the Software Metrics Definition Working Group. These drafts were distributed for industry and government review in the fall of 1991. We have now extended that work, guided by the comments we have received; and our results are presented in three "framework" documents that are being published concurrently with this report [Park 92], [Goethert 92], [Florac 92]. These documents provide methods for clearly communicating measurement results. They include measurement definitions; checklists for constructing alternative definitions and data specifications; instructions for using the checklists to collect, record, and report measurement data; and examples of how the results can be used to improve the planning and management of software projects. It is from the framework documents that the recommendations in this report are drawn. The framework documents should be viewed as companion reports by anyone seeking to implement the recommendations presented herein. Figure 1-3 shows the interrelationships among these

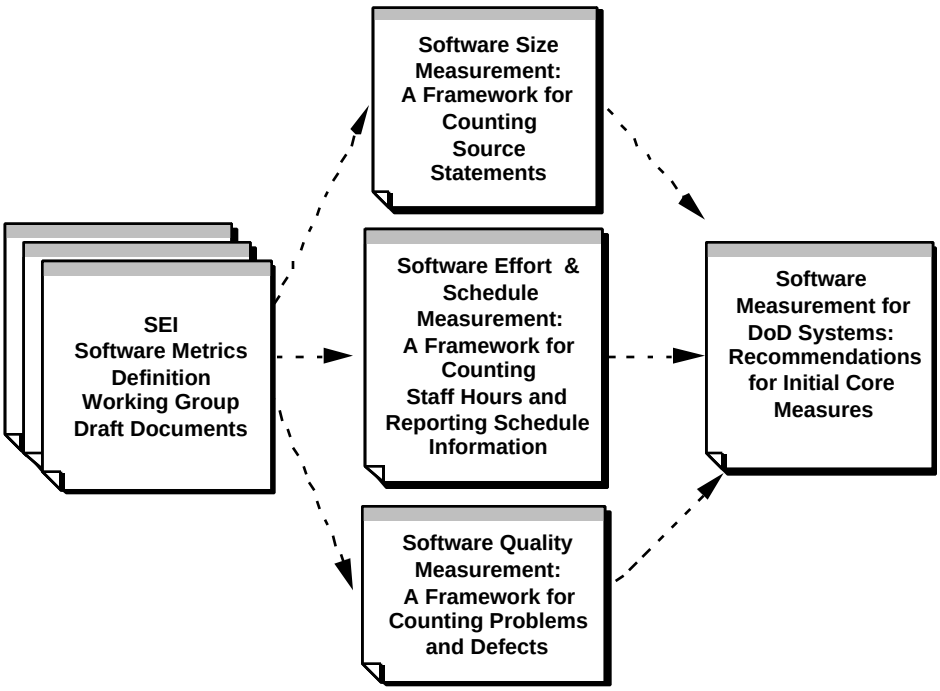


Figure 1-3 Relationships Between This Report and Its Supporting Documents

reports.

The starting point for our measurement definition work has been management's need for answers to several key questions that are present in any software project:

- How large is the job?
- Do we have sufficient staff to meet our commitments?
- Will we deliver on schedule?
- How good is our product?
- How are we doing with respect to our plans?

To address these questions, we have concentrated on developing methods for obtaining unambiguous measures for size, effort, schedule, and quality. Reliable assessments of these characteristics are crucial to managing project commitments. Measures of these characteristics also serve as foundations for achieving improved levels of process maturity, as defined in the SEI Capability Maturity Model for Software [Humphrey 89], [Paulk 91], [Weber 91].

The objective of our measurement work is to assist program managers, project managers, and government sponsors who want to improve their software products and processes. The purpose of the recommendations in this report and its supporting framework documents is to provide operational mechanisms for getting information for three important management functions:

- Project planning—estimating costs, schedules, and defect rates.
- Project management—tracking and controlling costs, schedules, and quality.
- Process improvement—providing baseline data, tracing root causes of problems and defects, identifying changes from baseline data, and measuring trends.

The measures we recommend in this report form a basis from which to build a comprehensive measurement and process improvement program. We support these measures with structured methods that can help organizations implement clear and consistent recording and reporting. The methods include provisions for capturing the additional details that individual organizations need for addressing issues important to local projects and processes.

## A Note on Implementation Policy

Our understanding is that the DoD plans to implement the recommendations in this report. Although we expect to be assisting the DoD in this endeavor, responsibility for implementation rests with the Department of Defense. Questions with respect to implementation policy and directives should be referred to the appropriate DoD agencies.

# 2. Integrating Measurement with Software Processes

Collecting and using even the most basic measures in ways that are meaningful will prove to be a challenge for many organizations. Although some projects already collect forms of the measures we recommend and a number of others as well, it is also true that many measurement efforts have failed because they attempted to collect too much too soon [Rifkin 91]. This chapter describes an implementation strategy that addresses both the challenge and planning of software measurement. The strategy stresses foundations that must be laid if measurement is to be successful.

## 2.1. Defining the Measurement Process

Measurement definitions like those in our framework documents address but one part of a measurement program. A broader process and process infrastructure is needed to establish successful software measurement within an organization. Figure 2-1 shows the sequence of tasks that should be performed [McAndrews 92]. Organizations often tend to overlook the first two steps and jump immediately to prototyping or collecting data. When this happens, measurement is apt to become viewed as just another cost rather than as an integral part of management and process improvement.

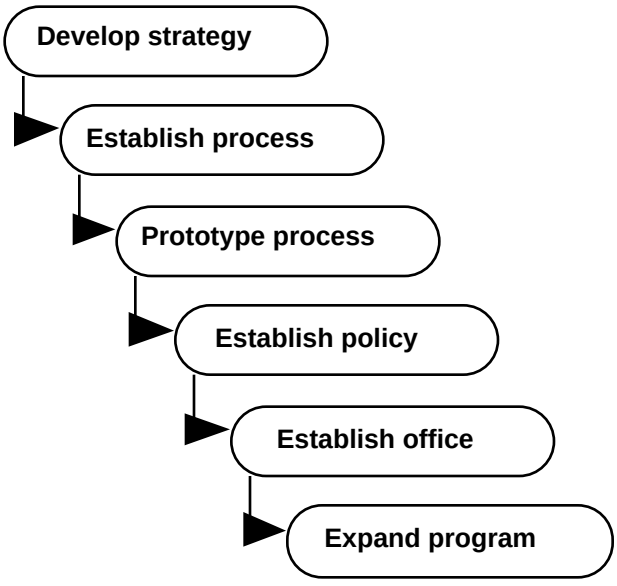


Figure 2-1 Steps for Establishing a Software Measurement Process Within an Organization

In the context of Figure 2-1, the **Establish process** step entails identifying and integrating a consistent, goal-related, measurement process into an organization's overall software

methodology. Figure 2-2 provides a top-level view of the stages that comprise successful measurement processes [McAndrews 92]. Organizations begin the process with specific management needs. During the first stage, they identify the measurements they must collect. They do this by identifying the current organizational situation, primary goals, and corresponding primitive measures that will be used to determine progress toward the goals. Basili's Goal/Question/Metric (GQM) paradigm [Basili 88] provides guidelines that can be used at this stage to help identify the primitive measures. Without careful analysis of management needs before measurement begins, organizations frequently collect data that is not meaningful for making decisions within their overall process.

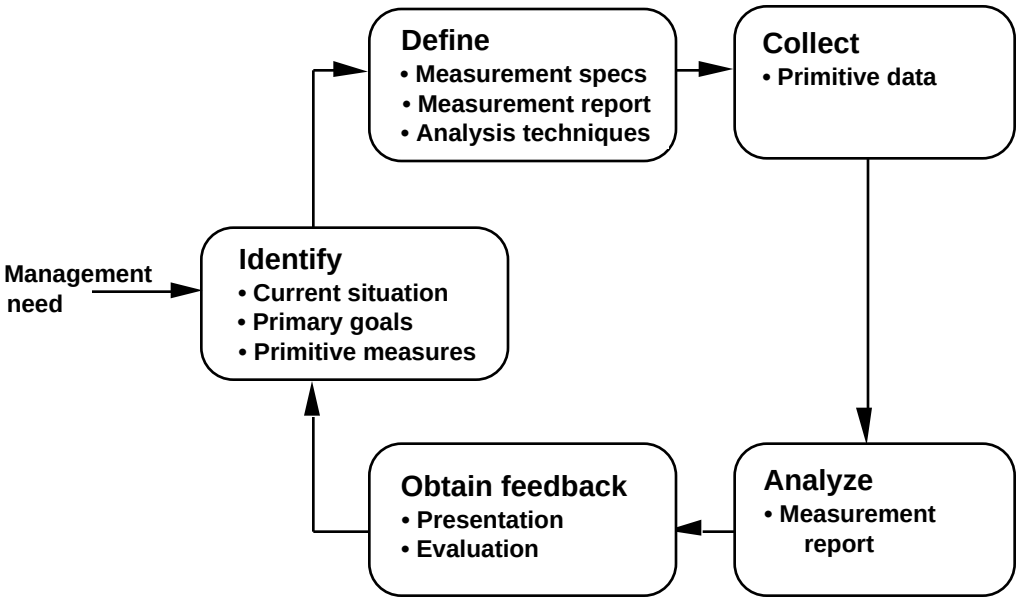


Figure 2-2 Stages of a Measurement Process

Once primitive measures are identified, they must be defined. While the **Identify** stage identifies measures such as staff-hours or source lines of code, the **Define** stage states how these measures are to be collected and used. During this stage, organizations create specifications for each of their primitive measures. They also define reporting formats and procedures for analyzing the data that will be collected. The procedures show how the data will be analyzed to establish baselines and determine progress toward management goals. Without this stage, data tends to be inconsistently reported and untrustworthy, and managers seldom know if they can compare their data with that of other projects.

Completion of the first two stages results in a documented baseline measurement plan. Organizations can now follow this plan to execute the next stages: **Collect** data and **Analyze** the data as defined. The **Obtain feedback** stage includes presenting and distributing measurement results and incorporating reviewers' comments into the reports. In addition, the **Obtain feedback** stage evaluates the effectiveness of the efforts at each stage in

addressing the original project goals. The situation and goals are then re-evaluated and the cycle is repeated, much as in the Plan/Do/Check/Act cycle upon which the quality methods of Shewhart and Deming are built [Deming 86].

## 2.2. Measurement and the Capability Maturity Model

Measurement is one of the enablers of process maturity. Recent publications such as [Grady 87] and [ami 92] explain why a measurement program is an important part of any successful development or maintenance activity. Measures are essential for establishing repeatable processes—without them, organizations will never know whether they have succeeded in establishing repeatability. Software engineering literature describes dozens of measures that can be applied to a wide variety of project, process, and product attributes [Conte 86], [Pfleeger 91]. Measurement is also essential in the larger context of process assessment and improvement. Choosing measures, collecting data, analyzing the results, and taking action require time and resources. These activities make sense only when they are directed toward specific improvement goals. This section describes how software measurement and process maturity go hand in hand.

Some software development processes are more mature than others, and evidence of this has been documented [Kitson 92]. A key discriminator among process maturity levels is the ability of developers and managers to see and understand what is happening in the overall development process. At the lowest levels of maturity, the process is not well understood at all. As maturity increases, the process becomes better understood and better defined.

Measurement and the ability to see and understand are closely related—a developer can measure only what is visible in a process, and measurement helps to increase visibility. The Capability Maturity Model (CMM) can serve as a guide for determining what to measure first and how to plan an increasingly comprehensive measurement program [Humphrey 89], [Paulk 91], [Weber 91]. Baumert's recent report, *Software Measures and the Capability Maturity Model*, describes the use of software measures in this context [Baumert 92].

Figure 2-3 [adapted from Pfleeger 89 and Pfleeger 90] outlines the classes of measures suggested by the different levels of the Capability Maturity Model. Selection of specific measures depends on the concerns at each level and on the information that is attainable. Measures at level 1 provide baselines for comparison as an organization seeks repeatability. Measures at level 2 focus on project planning and tracking, while measures at level 3 become increasingly directed toward measuring the intermediate and final products produced during development. The measures at level 4 capture characteristics of the development process itself to allow control of the individual activities of the process. At level 5, processes are mature enough and managed carefully enough to allow measurement to provide feedback for dynamically changing processes across multiple projects.

Every measurement program should begin with an examination of the software process model currently in use and a determination of what is visible (i.e., identify the current situation). Measures should never be selected simply because the overall maturity is at a

| Maturity Level | Characteristics | Focus of Measurements |
|---|---|---|
| 1. Initial | Ad hoc, chaotic | Establishing baselines for planning and estimating |
| 2. Repeatable | Processes depend on individuals | Project tracking and control |
| 3. Defined | Processes are defined and institutionalized | Definition and quantification of intermediate products and processes |
| 4. Managed | Processes are measured | Definition, quantification, and control of subprocesses and elements |
| 5. Optimizing | Improvements are fed back to processes | Dynamic optimization and improvement across projects |

Figure 2-3 Relationship of Software Measures to Process Maturity

particular level. If one part of a process is more mature than others, tailored measures can enhance the visibility of that part and help meet overall project goals while basic measures are bringing the rest of the process up to a higher level of capability. For example, in a repeatable process with a well-defined configuration management activity, it may be appropriate and desirable to track reused elements from their origins to their uses in final products, even though the measures associated with this level of detail are not generally characteristic of level 2 organizations.

Evidence suggests that successful measurement programs start small and grow according to the goals and needs of the organization [Rifkin 91]. A measurement program should begin by addressing the critical problems or goals of each project, viewed in terms of what is meaningful or realistic at that organization's process maturity level. The process maturity framework then acts as a guide for expanding and building a measurement program that not only takes advantage of visibility and maturity, but also enhances process improvement activities.

# 3. Recommendations for Specific Measures

This chapter presents our recommendations for a basic set of software measures for use with DoD software systems. These recommendations are based on checklists, forms, and operational practices that are presented and discussed more completely in three framework reports:

- *Software Size Measurement: A Framework for Counting Source Statements* [Park 92]
- *Software Effort & Schedule Measurement: A Framework for Counting Staff-Hours and Reporting Schedule Information* [Goethert 92]
- *Software Quality Measurement: A Framework for Counting Problems and Defects* [Florac 92]

The framework reports should be used as references when implementing the recommendations in this report.

In the discussion that follow, we first introduce the basic measures and explain the criteria we used in developing the definitions and practices we recommend. Then, for each measure, we illustrate portions of the checklists we have constructed for defining and reporting measurement results. We support these illustrations with reasons for using the measures, and we provide advice and recommendations for making the measures effective.

## 3.1. The Basic Measures

We recommend that four basic measures be among the management tools used within DoD organizations for acquiring, developing, and maintaining software systems. These measures address important product and process characteristics that are central to planning, tracking, and process improvement. Figure 3-1 lists the measures and relates them to the characteristics they address.

| Unit of measure | Characteristics addressed |
|---|---|
| Counts of physical source lines of code | Size, progress, reuse |
| Counts of staff-hours expended | Effort, cost, resource allocations |
| Calendar dates | Schedule |
| Counts of software problems and defects | Quality, readiness for delivery, improvement trends |

Figure 3-1 Measures Recommended for Initial DoD Implementation

The measures in Figure 3-1 are not the only ones that can be used to describe software products and processes. But they are practical measures that do produce useful information. And, importantly, they are measures that we can define in ways that promote consistent use.

The exact definitions we recommend are presented in the three framework reports cited above. These definitions follow structured rules that state explicitly what is included in each measure and what is excluded. They are accompanied by checklists that individual organizations can use to specify and obtain the supporting data that they need for addressing the management issues that are important to them. The checklists can be used also to describe the measurement data that is reported now, so that receivers of the information will not be misled by unstated assumptions or local variations in measurement practices.

In preparing the framework reports, we were guided by two criteria:

- *Communication*: If someone uses one of our checklists or definitions to record and report measurement results, will others know precisely what is included, what is excluded, and how the measurement unit is defined?
- *Repeatability*: Would others be able to repeat the measurements and get the same results?

These properties are essential if misunderstandings and misinterpretations are to be avoided. They are essential also if consistency is to be achieved across projects or from organization to organization.

As we show in the framework reports, each basic measure provides for collecting data on multiple attributes. Rarely, if ever, will experienced managers be satisfied with just a single number. For example, problems and defects will usually be classified according to attributes such as status, type, severity, and priority; effort will be classified by labor class and type of work performed; schedules will be defined in terms of dates and completion criteria; and size measures will be aggregated according to programming language, statement type, development status, origin, and production method. Moreover, to be of value, both estimates and measured values must be collected at regular intervals (e.g., weekly or monthly). Thus, what may at first glance appear to be just a few measures is, in reality, much more. Implementing the collection and use of these measures uniformly across the DoD using clearly specified, consistent definitions, will be a major accomplishment. Neither the difficulty nor the value of this task should be underestimated.

The four sections that follow present our recommendations for addressing each of the basic measurement categories. They illustrate the checklist-based methods we recommend, give reasons for using each measure, and present specific recommendations for implementation and use.

## 3.2. Size

We recommend that DoD organizations adopt physical source lines of code (SLOC) as one of their first measures of software size. The coverage we recommend is defined in the framework report on software size [Park 92, Figure 5-1]. Figure 3-2 shows a portion the definition so that you can see how we have used checklists to make measurement rules explicit.

**Definition Checklist for Source Statement Counts**

Definition name: ***Physical Source Lines of Code (basic definition)*** Date: ***8/7/92*** Originator: ***SEI***

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Measurement unit:** | | **Physical source lines** | ✔ | | | | |
| | | **Logical source statements** | | | | | |
| **Statement type** | **Definition** | ✔ | **Data array** | | | **Includes** | **Excludes** |
| | *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | | | | |
| 1 Executable | **Order of precedence ->** | | | | 1 | ✔ | |
| 2 Nonexecutable | | | | | | | |
| 3 Declarations | | | | | 2 | ✔ | |
| 4 Compiler directives | | | | | 3 | ✔ | |
| 5 Comments | | | | | | | |
| 6 On their own lines | | | | | 4 | | ✔ |
| 7 On lines with source code | | | | | 5 | | ✔ |
| 8 Banners and nonblank spacers | | | | | 6 | | ✔ |
| 9 Blank (empty) comments | | | | | 7 | | ✔ |
| 10 Blank lines | | | | | 8 | | ✔ |
| 11 | | | | | | | |
| 12 | | | | | | | |
| **How produced** | **Definition** | ✔ | **Data array** | | | **Includes** | **Excludes** |
| 1 Programmed | | | | | | ✔ | |
| 2 Generated with source code generators | | | | | | ✔ | |
| 3 Converted with automated translators | | | | | | ✔ | |
| 4 Copied or reused without change | | | | | | ✔ | |
| 5 Modified | | | | | | ✔ | |
| 6 Removed | | | | | | | ✔ |
| 7 | | | | | | | |
| 8 | | | | | | | |

Figure 3-2 A Part of the Recommended Definition for Physical Source Lines of Code

As Figure 3-2 shows, the measure we recommend for physical source lines is a version of one often called noncomment, nonblank source statements. However, it is considerably more explicit. Not only does it spell out the rules to be used when comments are on the same lines as other source statements, it also addresses all origins, stages of development, and forms of code production and distinguishes between delivered and nondelivered statements, code that is integral to the product and external to the product, operative and inoperative (dead) code, master source code and various kinds of copies, and different source languages.

The full definition in the framework report on software size produces a single measure of size for each source language used. We recommend the DoD and its supporting organizations use this measure to describe the overall size of the products they build and support.

Because the picture we get with a single measure is seldom sufficient to competently plan and manage software activities, we also recommend that individual organizations use the checklist to specify the supporting measurements they make for tracking and analyzing the activities most important to them. Figure 3-3 is an example of how two sections of the checklist can be used to designate individual data elements for project tracking. Here the **Data array** boxes for the **How produced** and **Development status** attributes are checked to show that these sections of the checklist are requests for individual data elements, not modifications to the basic definition. The other sections of the checklist then define the rules that apply when measuring these elements.

| **How produced** | **Definition** ☐ | **Data array** ✔ | **Includes** | **Excludes** |
|---|---|---|---|---|
| 1 Programmed | | | ✔ | |
| 2 Generated with source code generators | | | ✔ | |
| 3 Converted with automated translators | | | ✔ | |
| 4 Copied or reused without change | | | ✔ | |
| 5 Modified | | | ✔ | |
| 6 Removed | | | ✔ | |
| 7 | | | | |
| 8 | | | | |

| **Development status** | **Definition** ☐ | **Data array** ✔ | **Includes** | **Excludes** |
|---|---|---|---|---|
| *Each statement has one and only one status, usually that of its parent unit.* | | | | |
| 1 Estimated or planned | | | | ✔ |
| 2 Designed | | | | ✔ |
| 3 Coded | | | ✔ | |
| 4 Unit tests completed | | | ✔ | |
| 5 Integrated into components | | | ✔ | |
| 6 Test readiness review completed | | | ✔ | |
| 7 Software (CSCI) tests completed | | | ✔ | |
| 8 System tests completed | | | ✔ | |
| 9 | | | | |
| 10 | | | | |
| 11 | | | | |

Figure 3-3 Specifying Data for Project Tracking (A Partial Example)

Measurement specifications like the one in Figure 3-3 produce arrays of data elements that can be used to track the status of the code produced by each production process. This information can be used to prepare graphs like Figure 3-4.

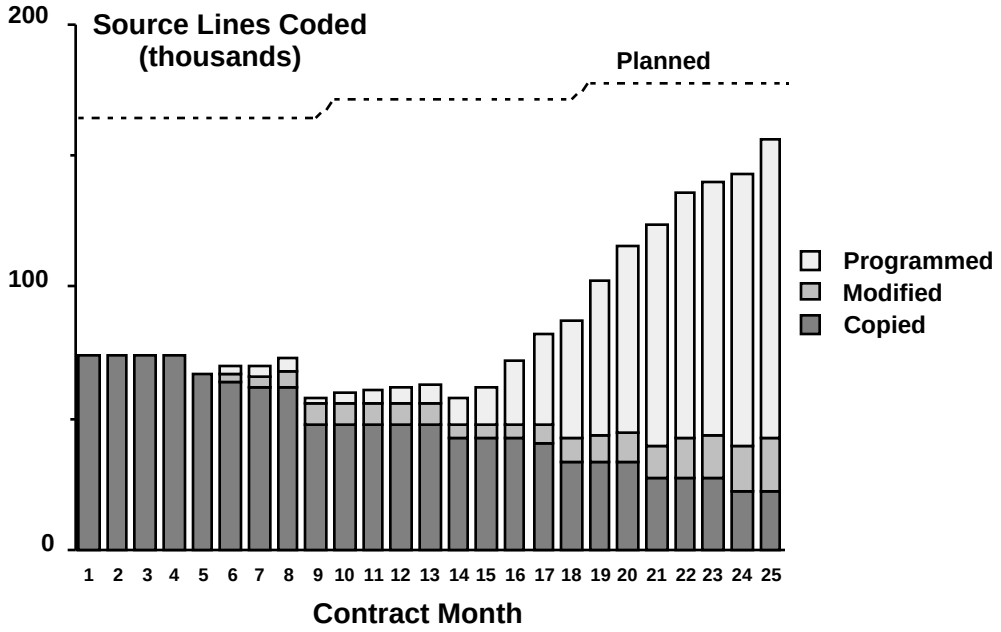


Figure 3-4 The Case of Disappearing Reuse

In this instance, it is apparent that a single measure of size would give a misleading picture of progress and cost. Similar graphs that plot the amount of code by development status can also be useful in relating progress to schedules.

### 3.2.1. Reasons for using physical source line measures

Some of the more popular and effective measures of software size are physical source lines of code (SLOC), logical source statement (instructions), function points (or feature points), and counts of logical functions or computer software units (CSUs). The following paragraphs list several of the pros and cons associated with these measures.

**Physical source lines of code (SLOC)**

- These are among the easiest measurements to make. Just count end-of-line markers.
- Counting methods are not strongly dependent on the programming language used. You need to specify only how you will recognize the statement types you will *not* count (e.g., comments, blank lines). It is relatively easy to build automated counters for physical source line measures.
- Measurement results can be more subject to variations in programming style than with other measures.
- Most of the historical data that has been used to construct the cost models used for project estimating is based on physical measures of source code size.

**Logical source statements (instructions)**

- Users must specify exact and complete rules for identifying the beginnings and endings for all possible statement types.
- Every language is different.
  - Users must specify complete rules for each.
  - Different rules for different languages can cause lack of comparability across languages.
- Users must specify the rules to be used for recognizing and counting embedded statements for each source language.
- Reference manuals for several important languages reserve the term “statement” to mean executable statement. This can introduce confusion with respect to elements like declarations, comments, compiler directives, and blank lines—any of which we may want to count.
- Methods for designating comments vary widely among languages. Without a consistent concept for what constitutes a “logical comment,” most organizations have to resort to physical line counts to achieve any form of comparability if the extent of commenting is to be measured.
- Users must state clear and consistent rules for distinguishing between expressions and statements. This can present problems in expression-based languages such as C and C++.

**Function Points and Feature Points**

- Function point and feature point counts do not depend on the source languages used.
- Estimates for function points and feature points can be obtained early in the development cycle.
- Function point and feature point counts are oriented toward the customer’s view (what the software does) rather than the producer’s view (how he does it). This puts the focus on value received, rather than on the particular design that is employed.
- Because function point and feature point counts are system-level (distributed) measures, we cannot use them to determine status. For example, they cannot help us say that a project is 80% of the way through coding and 35% of the way through component integration. This makes them unsuitable for project tracking.
- Function point and feature point counts are not equally applicable to all kinds of software. Although effective in business environments, they have not enjoyed widespread success in embedded systems or heavily computational applications.
- Automated function point or feature point counters do not yet exist.

**Counts of units or functions**

- These are clearly useful measures. We suggest you use them to supplement other basic size measures. But as of now there is no work that we know to help in constructing and communicating formal definitions for these measures.

Although we recommend starting with physical source lines as one of the first measures of software size, we do not suggest that anyone abandon any measure that is now being used to good effect. Nor do we suggest that counts of physical source lines are the best measure for software size. It is unlikely that they are. But until we get firm, well-defined evidence that other measures are better, we opt for simplicity—and in particular, for the simplest set of explicit rules that can be applied across many languages.

### 3.2.2. Specific recommendations for counting physical source lines

- Use the size checklist and supplemental rules forms to record the rules you currently use in your size reports [Park 92, Figures 3-2, 7-1, 7-2, 7-3].
- Adopt physical source lines of code (SLOC) as one of your first measures of software size. This measure should supplement rather than replace size measures you currently use.
- When counting physical source lines of code, use the definition for basic SLOC in Figure 5-1 of the Software Size Measurement framework report [Park 92].
- When adopting (or modifying) the recommended definition for physical source lines, complete a copy of the supplemental rules form for each language you plan to measure [Park 92, Figure 7-1]).
- If you presently count logical source statements, consider adopting the definition in Figure 5-2 of the Software Size Measurement framework report [Park 92]. Complete this definition by filling out the supplemental rules form for each source language you use [Park 92, Figure 7-2].
- Use Data Spec A or a subset thereof to collect data for project tracking [Park 92, Figure 5-4].
- Use Data Spec B or a similar specification at the completion of projects to collect postmortem data for future planning and estimating [Park 92, Figure 5-9].
- Use the size checklist and supplemental rules forms to record and report the rules you use when preparing estimates of software size.

## 3.3. Effort

We recommend that DoD organizations adopt staff-hours as their principal measure for effort. The staff-hour unit we recommend is the one used by the IEEE in its draft *Standard for Software Productivity Metrics*: "A staff-hour is an hour of time expended by a member of the staff" [IEEE P1045/D5.0].

The coverage we recommend for total staff-hour measures is defined in the framework report on effort and schedule measurement [Goethert 92, Figure 8-1]. Figure 3-5 shows two parts of the checklist to illustrate some of the coverage rules we recommend.

## Staff-Hour Definition Checklist

Definition Name: **Total System Staff-Hours For Development**

Date: **7/28/92**

Originator:

Page: 1 of 3

| Type of Labor | Totals include | Totals exclude | Report totals |
|---|---|---|---|
| Direct | ✔ | | |
| Indirect | | ✔ | |
| **Hour Information** | | | |
| Regular time | | | ✔ |
| Salaried | ✔ | | |
| Hourly | ✔ | | |
| | | | |
| Overtime | | | ✔ |
| Salaried | | | |
| Compensated (paid) | ✔ | | |
| Uncompensated (unpaid) | ✔ | | |
| | | | |
| Hourly | | | |
| Compensated (paid) | ✔ | | |
| Uncompensated (unpaid) | ✔ | | |
| | | | |
| **Employment Class** | | | |
| Reporting organization | | | |
| Full time | ✔ | | |
| Part time | ✔ | | |
| Contract | | | |
| Temporary employees | ✔ | | |
| Subcontractor working on task with reporting organization | ✔ | | |
| Subcontractor working on subcontracted task | ✔ | | |
| Consultants | ✔ | | |
| | | | |

| Product-Level Functions continued | Totals include | Totals exclude | Report totals |
|---|---|---|---|
| | | | |
| **System-Level Functions** | | | ✔ |
| (Software effort only) | | | |
| System requirements & design | | | |
| System requirements analysis | ✔ | | |
| System design | ✔ | | |
| Software requirements analysis | ✔ | | |
| Integration, test, & evaluation | | | |
| System integration & testing | ✔ | | |
| Testing & evaluation | ✔ | | |
| Production and deployment | | ✔ | |
| Management | ✔ | | |
| Software quality assurance | ✔ | | |
| Configuration management | ✔ | | |
| Data | ✔ | | |
| Training | | | |
| Training of development employees | ✔ | | |
| Customer training | | ✔ | |
| Support | ✔ | | |

Figure 3-5 Sections of the Recommended Definition for Staff-Hour Reports

From the full definition in the framework report on effort and schedule measurement, organizations can construct a number of data specifications to meet specific needs. For example, subtotals can be requested for any element or collection of elements in the checklist. Many organizations will want to use these subtotals to estimate and track the effort expended on major functional activities. Measuring effort for requirements analysis, design, coding, unit testing, integration, IV&V, and configuration management are cases in point. Similarly, individual totals for overtime or for system-level activities may be useful. Individual totals can also be collected for elements that are excluded from formal definitions of total effort.

The coverage requested by our recommended definition ranges more widely than some projects or organizations may encounter. This should not be a problem. If some elements in the checklist are not present in a particular project, the effort associated with them will be zero and the overall totals will not be affected.

Of more concern are cases where elements are present but not recorded. There is an important distinction that must be maintained between elements *excluded* from a report and elements of effort *not performed* on a project. It is generally safer and more explicit to include elements in your coverage and report zero values for them when they are not present than to exclude these values and make readers guess about their existence. This is the reason the definition we recommend for staff-hour coverage includes almost all elements in the definition checklist. It is also the reason why you will find individual reports for subtotals to be useful additions to overall measures of total effort.

### 3.3.1. Reasons for using staff-hour measures

Reliable measures for effort are prerequisites for reliable measures of software cost. They are also important in a more direct way. The principal means we have for managing and controlling costs and schedules is through planning and tracking the human resources we assign to individual tasks and activities.

Some candidate units for measuring and reporting effort data are labor-months, staff-weeks and staff-hours. The concept of a labor-month is well understood. Nevertheless, using labor-months to record and report effort data presents two obstacles:

- There is no standard for the number of hours in a labor-month. Practices vary widely among contractors and within the government, and reported values range from less than 150 hours per labor-month to more than 170. Moreover, it is possible for individual organizations to use different definitions for a labor-month on different projects, either because of government requirements or because the organization is working as a subcontractor to another contractor.
- Labor-months often do not provide the granularity we need for measuring and tracking individual activities and processes, particularly when our focus is on process improvement.

Measuring effort in terms of staff-weeks presents many of the same problems and some additional ones as well. For example, although the basic assumption is that a calendar week is five working days, the length of a standard working day can vary from organization to organization. Weekend work, overtime work, and holidays falling within a week must also be addressed and defined if staff-week measures are to be used.

By using staff-hours as the fundamental unit for recording and reporting effort data, we avoid these problems. Labor-month and staff-week measures can still be calculated from staff-hours, should these measures be needed for presentations or other summaries.

### 3.3.2. Specific recommendations for counting staff-hours

- Adopt staff-hours as your fundamental measure of effort.
- Use the Staff-Hour Definition Checklist and the Supplemental Information Form to record the definition of staff-hours you are currently using [Goethert 92, Appendix E and Figure 4-1].
- Use the Staff-Hour Definition Checklist to specify the rules for the effort elements you want included in and excluded from total staff-hour measures.
- When adopting a definition for the coverage of staff-hour measures, use the one recommended in the framework report on software effort and schedule measurement [Goethert 92, Figure 8-1].
- Report staff-hour totals at the computer software configuration item (CSCI) level, build level, and system level.
- Use the Staff-Hour Definition Checklist to report and communicate attributes and values included in the staff-hour measures.
- At the beginning of projects, use the staff-hour definition checklist to define the coverage of the effort measures you want reported.
- During projects, use the reporting forms in the framework report to augment (not replace) your contractually required status reports.
- At the end of projects, retain your staff-hour definition checklists, periodic staff-hour reports, and final staff-hour report.

## 3.4. Schedule

We recommend that DoD projects adopt structured methods for defining two important and related aspects of the schedules they report: the **dates** (both planned and actual) associated with project milestones, reviews, audits, and deliverables; and the **exit** or **completion criteria** associated with each date.

The checklist we recommend for defining dates for milestones, reviews, audits, and deliverables is presented in the framework report on effort and schedule measurement [Goethert 92, Figures 6-1, 6-2]. Figure 3-6 below shows portions of this checklist so you can see how the rules for defining dates are recorded. The figure illustrates a specification that requires dates for all milestones, reviews, and audits from the software specification review (SSR) through functional and physical configurations audits to be reported. Critical design reviews (CDRs) and subsequent milestones are to be reported for each build as well. In the example, three events are required for exit or completion criteria for SSR, PDR, and CDR;

**Schedule Checklist** Date: _____
**Part A: Date Information** Originator: _____
Page 1 of 3

Project will record planned dates: Yes ✔ No _____
If Yes, reporting frequency: Weekly _____ Monthly ✔ Other: _____
Project will record actual dates: Yes ✔ No _____
If Yes, reporting frequency: Weekly _____ Monthly ✔ Other: _____

Number of builds ____

| **Milestones, Reviews, and Audits** | Include | Exclude | Repeat each build | Relevant dates reported* |
|---|---|---|---|---|
| **CSCI-Level** | | | | |
| Software specification review | ✔ | | | **2,3,6** |
| Preliminary design review | ✔ | | | **2,3,6** |
| Critical design review | ✔ | | ✔ | **2,3,6** |
| Code complete | ✔ | | ✔ | **1** |
| Unit test complete | ✔ | | ✔ | **6** |
| CSC integration and test complete | ✔ | | ✔ | **5** |
| Test readiness review | ✔ | | ✔ | **3** |
| CSCI functional & physical configuration audits | ✔ | | ✔ | **3** |

*Key to indicate "relevant dates reported" for reviews and audits
1 - Internal review complete
2 - Formal review with customer complete
3 - Sign-off by customer
4 - All high-priority action items closed
5 - All action items closed
6 - Product of activity/phase placed under configuration management
7 - Inspection of product signed off by QA
8 - QA sign-off
9 - Management sign-off
10 - _____
11 - _____

Figure 3-6 Sections of the Schedule Checklist for Milestones, Reviews, and Audits

but only a single criterion is used for each of the remaining milestones, reviews, and audits. Reporting the dates associated with the individual criteria helps insure accuracy in the overall report. It also provides insight into process timelines that can be useful for planning future projects and for process improvement.

The schedule checklist in the framework report lists several completion criteria for milestones, reviews, audits, and deliverables. Others can easily be added. Examples that could be appropriate for specific activities include the following:

- Internal review held.
- Formal review with customer held.
- All high-priority action items closed.
- All action items closed.
- Document entered under configuration management.
- Deliverer to customer.
- Customer comments received.
- Changes incorporated.
- Customer sign-off obtained.

The part of the checklist that addresses deliverables is similar to the part that describes milestones, reviews, and audits. Figure 3-7 shows the portion of the part that defines the deliverables associated with CSCI-level products.



**Part A: Date Information (cont.)**

| Deliverable Products | Include | Exclude | Repeat each build | Relevant dates reported* |
|---|---|---|---|---|
| **CSCI-Level** | | | | |
| Preliminary software requirements spec(s) | ✔ | | | **3** |
| Software requirements specification(s) | ✔ | | | **1,3,5,6** |
| Software preliminary design document(s) | ✔ | | | **1,3,5,6** |
| Software (detailed) design document(s) | ✔ | | ✔ | **1,3,5,6** |
| Software test description(s) (cases) | ✔ | | ✔ | **1,3,5,6** |
| Software test description(s) (procedures) | ✔ | | ✔ | **1,3,5,6** |
| Software test report(s) | ✔ | | ✔ | **3,7** |
| Source code | ✔ | | ✔ | **1,2,3,6,7** |
| Software development files | | ✔ | | |
| Version description document(s) | | ✔ | | |

*Key to indicate "relevant dates reported" for deliverable products
1 - Product under configuration control
2 - Internal delivery
3 - Delivery to customer
4 - Customer comments received
5 - Changes incorporated
6 - Sign-off by customer
7 - IV&V sign-off
8 -

Figure 3-7 Sections of the Schedule Checklist for CSCI-Level Products

Note that there are no sections in the checklist for specifying project activities or "phases." The reason for this is two-fold:

1. The variations associated with beginning and ending activities make it difficult to define start and end dates in a precise, unambiguous way. Most activities (e.g., requirements analysis, design, code) continue to some extent throughout the project. Where one project may consider requirements analysis complete at completion of a software specification review, another may consider it to be ongoing throughout development.
2. A second source of ambiguity stems from the fact that some activities start, stop, and start again, making it very difficult to pin down any meaningful dates.

In contrast to phases, project milestones, reviews, audits, and deliverables have clear-cut completion criteria that can be associated with specific dates. The schedule checklist we recommend deals exclusively with events of this sort.

**Schedule Reporting Form**
**Date Information**
CSCI-Level Information

Date: _____
Originator: _____
Project: Example from Figure 3-6
Period ending: _____
CSCI: _____
Build: _____

| **Milestones, Reviews, and Audits*** | **Planned** | **Changed** | **Actual** |
|---|---|---|---|
| Software specification review | | | |
| 2 - Formal review with customer complete | | | |
| 3 - Sign-off by customer | | | |
| 6 - Products under configuration management | | | |
| Preliminary design review | | | |
| 2 - Formal review with customer complete | | | |
| 3 - Sign-off by customer | | | |
| 6 - Products under configuration management | | | |
| Critical design review | | | |
| 2 - Formal review with customer complete | | | |
| 3 - Sign-off by customer | | | |
| 6 - Products under configuration management | | | |
| Code complete | | | |
| 1 - Internal review complete | | | |
| Unit test complete | | | |
| 6 - Products under configuration management | | | |
| CSC integration and test complete | | | |
| 5 - All action items closed | | | |
| Test readiness review | | | |
| 3 - Sign-off by customer | | | |
| CSCI functional & physical config. audits | | | |
| 3 - Sign-off by customer | | | |

*Only those completion criteria specified on the checklist appear below each deliverable.
Enter a check mark in the "Changed" column if Planned date has changed since last reporting.

Figure 3-8 Example of a Report Form for System-Level Milestone Dates

We recommend that milestones, reviews, audits, and deliverables be reported on forms like the one in Figure 3-8 on the previous page. This figure shows how a report has been tailored to reflect the dates and completion criteria checked in Figure 3-6. Note the column labeled **Changed**. A check in this column indicates that the value shown is different from the previous report (either changed or newly added). This is intended to make it easy for the receiver of the report to update only those values that have changed. We recommend that a separate report be filled out for each CSCI. A similar form is presented in the framework report for reporting system-level dates. Reporting forms, in general, are intended to be tailored.

A checklist for specifying progress measures is also illustrated in the framework report for effort and schedule measurement. Appendix D of that report gives three examples that show how this checklist can be used to define progress measures used in Air Force Pamphlet 800-48 (*Acquisition Management Software Management Indicators*) [USAF 92], the Army Software Test and Evaluation Panel (STEP) measurement set [Betz 92], and the MITRE metrics (*Software Management Metrics* ) [Schultz 88].

### 3.4.1. Reasons for using calendar dates

Schedule is a primary concern of project management. Timely delivery is often as important as either functionality or quality in determining the ultimate value of a software product. Moreover, project management can become especially complicated when delivery dates are determined by external constraints rather than by the inherent size and complexity of the software product. Extremely ambitious schedules often result.

Because schedule is a key concern, it is important for managers to monitor adherence to intermediate milestone dates. Early schedule slips are often a precursor to future problems. It is also important to have objective and timely measures of progress that provide an accurate indication of status and that can be used for projecting completion dates for future milestones.

Cost estimators and cost model developers are also very interested in schedules. Project duration is a key parameter when developing or calibrating cost models. Both model developers and estimators must understand what activities the duration includes and excludes. If we are told that a project took three and half years, a reasonable response is to ask exactly what was included in that time period. Does the period include system requirements analysis and design or just software activities? Does it include hardware-software integration and testing or just software integration?

People involved in process improvement also use schedule information. They need to understand the basic time dependencies of the project and so they can identify bottlenecks in the process.

Tracking dates for milestones, reviews, audits, and deliverables provides a macro-level view of project schedule. Not only can slips in early milestones be precursors of future problems, but also insight can be gained by tracking the progress of activities which culminate in

reviews and deliverables. By tracking the rate at which the underlying units of work are completed, we have an objective basis for knowing where the project is at any given point in time and for projecting where it will be in the future.

### 3.4.2. Specific recommendations for using calendar dates

We recommend the following practices be used by acquisition and development managers:

Dates of milestones, reviews, audits, and deliverables

- Require and report both planned and actual dates for milestones, reviews, audits, and deliverables.
- Use the checklist to specify the exact dates to be reported. A good first set includes the date of baselining for products developed as part of a given activity, the date of formal review, the date of delivery for interim products, and the date of formal sign-off.
- Some dates apply to the entire build or system. In other cases, there will be dates for each CSCI. Track schedule information at least to the CSCI level. For critical CSCIs, you may want to track dates for individual computers software components (CSCs) and computer software units (CSUs).
- Require that planned and actual dates be updated at regular intervals. Keep all plans. Much can be learned by looking at the volatility of plans over time and the extent to which they are based on supporting data (like the progress measures).

Progress measures

- Use the checklist for progress measures to specify the measures to be tracked.
- Require or produce a plan that shows the rate at which work will be accomplished. There should be a plan for each CSCI. Require that the planned and measured values be reported at regular intervals.
- Progress measures are meaningless without objective completion criteria. Make sure that these criteria can be audited. It is your way of being assured that progress is real.
- At a minimum, require that the following be planned for and tracked:

  the number of CSUs completing unit test.

  the number of lines of code completing unit test.

  the number of CSUs integrated.

  the number of lines of code integrated.

DOD-STD-2167A leaves a huge gap between the critical design review which precedes coding and the test readiness review which precedes CSCI testing. If there are problems in meeting integration and test schedules, the earlier you know about it the better. These simple measures have been found to be extremely useful. Schultz presents an example in which counts of the number of CSUs completing unit test were plotted weekly [Schultz 88]. A simple linear extrapolation of the plot provided a remarkably accurate projection of when unit

testing would be complete for all CSUs . We present a similar example for source lines of code in Appendix B (Figure B-11).

We recommend the following practices for cost estimators and database administrators:

- Whenever possible, use the checklist before data is reported to specify the dates you would like to see (e.g., “For all reviews, report the date the review began and the date that the last document to be included in that review was signed off”).
- Require a filled-out definition checklist from anyone submitting schedule data, so that you can understand and document what the dates represent.
- For project start and end dates, make sure that it is clear which activities are included and whether the dates are planned or actual.

## 3.5. Quality

We recommend that counts of software problems and defects be used to help plan and track development and support of software systems. We recommend also that they be used to help determine when products are ready for delivery to customers and to provide fundamental data for process and product improvement We recommend that these counts be clearly and completely defined. We recommend that the checklists and forms in the framework report on software quality measurement be one of the methods used for describing and reporting the quality of DoD software systems.

We do not recommend that standardized definitions for the details of problem and defect measurement be attempted above the organization level. The situation is different here than it is for size, effort, and schedule measures. Processes for detecting, recording, fixing, and preventing problems and defects are far more closely coupled with the specific software processes that individual developers and maintainers use. Each process may well use different stages for defect resolution and have different definitions for states such as open, recognized, evaluated, resolved, and closed. Moreover, these processes and the definitions that accompany them are the prerogatives of the individual organizations. However, we do believe that the checklists and forms we have constructed (or others much like them) can be used by most organizations in purely descriptive ways to report the definitions of the numerical measures they report.

The checklists and forms we recommend for counting problems and defects are presented in the framework report on software quality measurement [Florac 92]. Figures 3-9, 3-10, and 3-11 in the paragraphs that follow illustrate portions of these forms.

**Problem Count Definition Checklist.** The Problem Count Definition Checklist (Figure 3-9) provides a structured approach for dealing with the details that we must resolve to reduce misunderstandings when collecting and communicating measures of problems and defects. With such a checklist, we can address issues one at a time by designating the elements that people want included in measurement results. We can also designate the elements to be excluded and, by doing so, direct attention to actions that must be taken to avoid

contaminating measurement results with unwanted elements. We recommend that the Problem Count Definition Checklist be used within DoD projects to define and select the attributes and values that must be counted to implement the problem and defect measurements selected by the software organization.

| **Problem Count Definition Checklist-1** | | | | |
|---|---|---|---|---|
| **Software Product ID [Example V1 R1]** | | | | |
| **Definition Identifier: [Problem Count A]** | | **Definition Date [01/02/92]** | | |
| **Attributes/Values** | **Definition [ ]** | | **Specification [ X ]** | |
| **Problem Status** | **Include** | **Exclude** | **Value Count** | **Array Count** |
| Open | ✔ | | ✔ | |
| Recognized | | | | |
| Evaluated | | | | ✔ |
| Resolved | | | | ✔ |
| Closed | ✔ | | ✔ | |
| **Problem Type** | **Include** | **Exclude** | **Value Count** | **Array Count** |
| Software defect | | | | |
| Requirements defect | ✔ | | ✔ | |
| Design defect | ✔ | | ✔ | |
| Code defect | ✔ | | ✔ | |
| Operational document defect | ✔ | | ✔ | |
| Test case defect | | ✔ | | |
| Other work product defect | | ✔ | | |
| | | | | |
| Other problems | | | | |
| Hardware problem | | ✔ | | |
| Operating system problem | | ✔ | | |
| User mistake | | ✔ | | |
| Operations mistake | | ✔ | | |
| New requirement/enhancement | | ✔ | | |
| | | | | |
| Undetermined | | | | |
| Not repeatable/Cause unknown | | ✔ | | |
| Value not identified | | ✔ | | |
| **Uniqueness** | **Include** | **Exclude** | **Value Count** | **Array Count** |
| Original | ✔ | | | |
| Duplicate | | ✔ | ✔ | |
| Value not identifed | | ✔ | | |

Figure 3-9 A Portion of the Definition Checklist for Counting Problems and Defects

**Problem Status Definition Form.** Problem status is an important attribute by which problems are measured. The criteria for establishing problem status is determined by the existence of data that reflects the progress made toward resolving the problem. We can use the Problem Status Definition Form to define the meaning of various problem states by defining the criteria for reaching each state. We do this in terms of the problem's attributes. The role of the Problem Status Definition Form is to communicate, either in a descriptive or prescriptive sense, the meaning of the **Problem Status** attribute. Figure 3-10 shows a portion of the Problem Status Definition Form.

**Problem Status Definition Rules**

**Product ID: Example** **Status Definition ID: Customer probs**

**Finding Activity ID: Customer Support** **Definition Date: 06/30/92**

**Section I**

| When is a problem considered to be Open? A problem is considered to be Open when all the attributes checked below have a valid value: | | When is a problem considered to be Closed? A problem is considered to be Closed when all the attributes checked below have a valid value: | |
|---|---|---|---|
| ✔ | Software Product Name or ID | ✔ | Date Evaluation Completed |
| ✔ | Date/Time of Receipt | | Evaluation Completed By |
| | Date/Time of Problem Occurence | ✔ | Date Resolution Completed |
| ✔ | Originator ID | | Resolution Completed By |
| | Environment ID | ✔ | Projected Availability |
| | Problem Description (text) | | Released/Shipped |
| | Finding Activity | | Applied |
| | Finding Mode | ✔ | Approved By |
| | Criticality | | Accepted By |

**Section II**

What Substates are used for Open?

| # | Name | # | Name |
|---|---|---|---|
| 1 | Recognized | 6 | |
| 2 | Evaluated | 7 | |
| 3 | Resolved | 8 | |
| 4 | | 9 | |
| 5 | | 10 | |

Figure 3-10 A Portion of the Checklist for Defining Status Criteria

**Problem Count Request Form.** The Problem Count Request Form is a checklist used to describe (or specify) attribute values that are literals (dates, customer IDs, etc.). The purpose is to explain the terms used in a problem counting definition or specification. This checklist is used to also describe (or specify) problem or defect measurements in terms of dates, specific software artifacts, or specific hardware or software configurations. Figure 3-11 shows a portion of the Problem Count Request Form.

The framework report on software quality measurement [Florac 92] provides a structure for describing and defining measurable attributes for software problems and defects. It uses a checklists and supporting forms to organize the attributes so that methodical and straightforward descriptions of software problem and defect measurements can be made. The checklists and supporting forms can be used for defining or specifying a wide variety of problem and defect counts, including those found by static or non-operational processes (e.g., design reviews or code inspections) or by dynamic or operational processes (e.g., testing or customer operation). Use of these checklists offers a method for reducing ambiguities and misunderstandings in these measures by giving organizations clear definitions of the terms used when measuring and reporting problem sand defects.

**Problem Count Request Form**

**Product ID, Ver/Rel: [Example V1R1]  Problem Count Def ID: [Problem Count A]**

**Date of Request: [6-15-92]  Requester's Name or ID: [W.T. Door]**

**Date Count to be made: [7-1-92]**

**Time Interval for Count: From [1-1-92] To [6-30-92]**

| **Aggregate Time By:** | Day | Week | Month | Year |
|---|---|---|---|---|
| Date opened | | | | |
| Date closed | | | | |
| Date evaluated | | | | |
| Date resolved | | | | |
| Date/time of occurence | | | | |

| **Report Count By:** | Attribute Sort Order | Select Value, Sort Order | Special Instructions or Comments |
|---|---|---|---|
| **Originator** | | | |
| Site ID | | | |
| Customer ID | | | |
| User ID | | | |
| Contractor ID | | | |
| Specific ID(s) list | | | |
| **Environment** | | Sort Order | |
| Hardware config ID | | | |
| Software config ID | | | |
| System config ID | | | |
| Test proc ID | | | |
| Specific ID(s) list | | | |
| **Defects Found In:** | | | |
| Select a configuration component level: | Type of Artifact | | |
| | Requirement | Design | Code | User Document |
| Product (CSCI) | | | | |
| Component (CSC) | ✔ | ✔ | ✔ | ✔ |
| Module (CSU) | | | | |
| Specific (list) | | | | |

Figure 3-11 A Portion of the Checklist for Requesting Counts of Problems and Defects

### 3.5.1. Reasons for counting problems and defects

Determining what truly represents software quality in the view of a customer or user can be elusive. But whatever the criteria, it is clear that the number of problems and defects associated with a software product varies inversely with perceived quality. Counts of software problems and defects are among the few direct measures we have for software processes and products. These counts allow us to quantitatively describe trends in detection and repair activities. They also allow us to track our progress in identifying and fixing process and product imperfections. In addition, problem and defect measures are the basis for quantifying other software quality attributes such as reliability, correctness, completeness, efficiency, and usabilty [IEEE P1061/D21].

Defect correction (rework) is a significant cost in most development and maintenance environments. The number of problems and defects associated with a product are direct

contributors to this cost. Counting problems and defects can help us understand where and how they occur and provide insight into methods for detection, prevention, and prediction. Counting problems and defects can also provide direct help in tracking project progress, identifying process inefficiencies, and forecasting obstacles that will jeopardize schedule commitments.

### 3.5.2. Specific recommendations for counting problems and defects

Our principal recommendation is that organizations begin using the checklists in the framework report on software quality measurement to describe the meaning of their problem and defect tracking reports [Florac 92]. There are many opportunities for using the checklists to advantage. We outline several below:

**Ongoing projects.** For projects that are currently in development and are measuring problem and defects, we recommend using the Problem Count Definition Checklist and supporting forms to verify that the data collected conforms to requirements and needs. This may reveal two things about the measurements: (1) the measurements do not "measure up", that is, they are less than clear and precise in their meaning, or (2) the existing measurements fall short of what is needed to control the development or maintenance activity. If the measurements in use can be documented by using the Problem Count Definition checklist, we recommend you to use the checklist to describe the measurements to those who will use the measurement results. The combination of a completed checklist and its supporting forms then becomes a vehicle for communicating the meaning of measurement results to others, both within and outside the originating organization.

**New and expanding projects.** For projects that want to establish or expand a measurement system, an initial task is to define the measurements that will be used to determine and assess progress, process stability, and attainment of quality requirements or goals. We recommend using the Problem Count Definition Checklist and supporting forms as mechanisms for specifying the software problem and defect measurement part of the software process. Using the checklists to precisely define the measurements helps to crystallize several significant questions—what data is required, when is it required, who collects it, how is it collected, where and how is it kept, when it is reported, who has access, and how are the measurements to be used?

**Serving the needs of many.** Software problem and defect measurements have direct application to estimating, planning, and tracking various parts of the software development process. Users within organizations are likely to have different purposes for using and reporting this data. We recommend that the Problem Count Definition Checklist be used to negotiate and resolve the conflicting views that can arise from these different purposes.

**Repository starting point.** Finally, the attributes and attribute values in the checklists can serve as a starting point for developing a repository of problem and defect data that can be used as a basis for comparing past experience to new projects, showing the degree of improvement or deterioration, rationalizing or justifying equipment investment, and tracking product reliability and responsiveness to customers.

# 4. Implementing the Basic Measures

This chapter outlines some priorities and related actions we recommend for implementing the basic measures discussed in this report and the supporting framework documents.

## 4.1. Initial Steps

The checklists and supporting forms in the framework documents can be used in two distinct ways: (1) to *describe* measurement results that are being reported now and (2) to *prescribe* those that will be collected in the future. When implementing the methods in the framework reports, first priority should go to describing information that is currently being reported. With clear descriptions for measurement results, misunderstandings can be minimized and inappropriate decisions avoided. Descriptions of existing measures can be obtained in short order and at very little cost. Standardizing definitions across projects or across organizations, on the other hand, will require more in the way of guidelines, training, and user support. This will require time to put in place, and the DoD need not wait to benefit from the methods in the framework reports. If organizations cannot describe what they are doing now, it is unlikely that the measures they use will be interpreted correctly.

To help make our advice specific, we have organized the priorities that we see into three classes—priorities within a project, within an organization, and within the DoD. We anticipate that the methods in the framework reports will be used in slightly different ways at each organizational level.

Within a project:

1. **Understand the data you are getting now.** Have your acquisition, development, and maintenance organizations use the checklists and supplemental forms to describe the measurements they currently report. To do this, they need only read the framework documents, reproduce the forms found at the back of each, and fill out those that are useful for describing the information they report.
2. **Standardize the content of future measurement reports.** Use the checklists and supplemental forms to define the counting and coverage rules that you want applied when collecting and reporting future software measurements.
3. **Define and collect the additional information you need for project planning and tracking.** Use the checklists and related forms to describe the additional data elements you want recorded and reported to support your primary measures.

Within an organization:

1. **Understand the historical data you already have.** Use the checklists for size, effort, and schedule to describe the data from past projects that you currently use as references for estimating and planning. This will help you see what the data

contains and what it excludes, so that projections made from the data can be applied appropriately to new projects.

2. **Get consistent data from project to project.** Use the checklists and their supplemental forms to ensure that the same definitions get applied to all projects.
3. **Get consistent data over time, while adjusting to the needs and practices of increasing process maturities.** Use the same checklist-based definitions with all projects. Use additional checklists as needed to create specialized specifications for the individual data elements that address your changing needs.

Within the DoD:

1. **Understand the data you are getting now.** Have reporting organizations attach copies of the checklists and their supplemental forms to each measurement report, so that readers of the reports will know exactly what the numbers in the reports represent.
2. **Get consistent data across different organizations.** Use the checklists and supplemental forms to create and communicate standardized specifications for the principal data elements you want recorded and reported for each project.
3. **Get consistent data over time, while permitting individual organizations to adjust to increasing process maturities.** Use the checklists and supplemental forms to define the basic data you want recorded and reported for each project. Permit each organization to add data elements and reports to meet their individual needs, provided they use the checklists and supplemental forms to specify their methods and report their results.

## 4.2. Related Actions for DoD Consideration

In addition to the priorities just listed, other actions will be needed if a consistent and sustained measurement program is to be implemented successfully across the DoD. To make implementation effective, we recommend that the DoD consider the following actions:

- Develop instructional and training materials to support the introduction and use of the definitions and measurement methods that will be implemented.
- Offer an initial implementation period in which individual organizations are permitted to test the checklist-based forms in their own operations. This will help insure that the proposed practices and definitions work for those organizations and that people are comfortable when using the forms to specify and report measurement results.
- Designate and staff an organization to respond to user questions and provide advice and assistance to organizations that are implementing the practices described in the framework reports.
- Consider preparing and distributing software to assist in automatically collecting the recommended measures. This would avoid duplicative development of automated tools and help ensure standardized (and comparable) measurement results.

- Provide for a review and revision cycle for updating the framework documents. Feedback from organizations that use the methods in the framework documents is very likely to identify ways in which the checklists and definitions can be improved and made to better serve DoD needs.
- Plan for republishing and redistributing of the framework documents after they are updated.
- Consider extending the set of framework reports to cover other useful measures. Extensions that should be considered include the following:
    - Develop checklists and methods for formally defining counts of software units (or functions). These definitions would be helpful in estimating software size and in planning and tracking convergence to completion. They could also be used early in planning cycles, before size estimates for individual software units are available. Since many of the coverage issues are the same as for source statement counting, a substantial foundation for this work has already been laid.
    - Apply the checklists in the size measurement report [Park 92] to prepare rules for identifying the beginnings and endings of logical source statements for the principal programming languages used in DoD systems. These rules would help organizations implement a measure of source code size that is less dependent on programming style than physical source lines of code. These rules could be added to automated code counters so that counts for logical source statements can be obtained at the same time as counts for physical source lines.
    - Construct analogs of the staff-hour checklists and supplemental forms that can be used to define and specify dollar cost measures. Much of the groundwork has been laid already in the framework report on effort and schedule measurement.

## 4.3. From Definition to Action—A Concluding Note

The power of clear definitions is not that they require action, but that they set goals and facilitate communication and consistent interpretation. In the framework reports and the initial measures we recommend, we seek only to bring clarity to definitions. Implementation and enforcement, on the other hand, are action-oriented endeavors. These endeavors are the prerogative and responsibility of the Department of Defense and its acquisition, development, and support agencies. We hope that the recommendations in this report and in the supporting framework documents provide the foundations and operational methods that will make this implementation effective.

# References


[ami 92] *Metric Users' Handbook*. London, England: The ami Consortium, South Bank Polytechnic, 1992.

[ANSI/IEEE 729-1983] *IEEE Standard Glossary of Software Engineering Terminology* (ANSI/IEEE Std 729-1983). New York, N.Y.: Institute of Electrical and Electronic Engineers, Inc., 1983.

[Bailey 86] Bailey, Elizabeth K.; Frazier, Tom P.; & Bailey, John W. *A Descriptive Evaluation of Automated Software Cost-Estimation Models* (IDA Paper P-1979). Alexandria, Virginia: October 1986.

[Basili 88] Basili, V.; & Rombach, H. D. "The TAME Project: Towards Improvement Oriented Software Environment". *IEEE Transactions on Software Engineering*, *14* , 6 (June 1988): 758-773 .

[Baumert 92] Baumert, John H. *Software Measures and the Capability Maturity Model* (CMU/SEI-92-TR-25). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1992.

[Betz 92] Betz, Henry P.; & O'Neill, Patrick J. *Army Software Test and Evaluation Panel (STEP) Software Metrics Initiatives Report*. Aberdeen Proving Grounds, Md.: U. S. Army Materiel Systems Analysis Activity, 1992.

[Conte 86] Conte, S. D.; Dunsmore H. E.; & Shen V. Y. *Software Engineering Metrics and Models*. Menlo Park, California: Benjamin-Cummings, 1986.

[Deming 86] Deming, W. Edwards. *Out of the Crisis*. Cambridge, Mass.: Center for Advanced Engineering Study, Massachusetts Institute of Technology, 1986.

[DoD 91] *Department of Defense Software Technology Strategy* (draft). Washington D.C.: Department of Defense, December 1991.

[DOD-STD-2167A] *Military Standard, Defense System Software Development* (DOD-STD-2167A). Washington, D.C.: United States Department of Defense, February 1989.

[Florac 92] Florac, William A. et al. *Software Quality Measurement: A Framework for Counting Problems and Defects* (CMU/SEI-92-TR-22). Pittsburgh, Pa: Software Engineering Institute, Carnegie Mellon University, September 1992.

[Goethert 92] Goethert, Wolfhart B. et al. *Software Effort Measurement: A Framework for Counting Staff-Hours* (CMU/SEI-92-TR-21). Pittsburgh, Pa: Software Engineering Institute, Carnegie Mellon University, September 1992.

[Grady 87] Grady, Robert B.; & Caswell, Deborah L. *Software Metrics: Establishing a Company-Wide Program*. Englewood Cliffs, N.J.: Prentice-Hall, 1987.

[Humphrey 89] Humphrey, Watts S. *Managing the Software Process*. Reading, Mass.: Addison-Wesley, 1989.

[IEEE P1045/D5.0] *Standard for Software Productivity Metrics [draft]* (P1045/D5.0). Washington, D.C.: The Institute of Electrical and Electronics Engineers, Inc., 1992.

[IEEE P1061/D21] IEEE *Standard for a Software Quality Metrics Methodology* (IEEE Standard P-1061/D21). New York, N.Y.: Institute of Electrical and Electronic Engineers, Inc., 1990.

[Kitson 92] Kitson, David H.; & Masters, Steve. *An Analysis of SEI Software Process Assessment Results* (CMU/SEI-92-TR-24). Pittsburgh, Pa: Software Engineering Institute, Carnegie Mellon University, July 1992.

[McAndrews 92] McAndrews, Donald R. "Establishing a Software Measurement Process," *Proceeding of the 4th Annual Software Quality Workshop*. Alexandria Bay, New York: Rome Laboratory, August 2-6, 1992.

[MIL-STD-881B] *Work Breakdown Structures for Defense Material Items* (MIL-STD-881B, draft). Air Force System Command, 18 February 1992.

[Park 92] Park, Robert E. et al. *Software Size Measurement: A Framework for Counting Source Statements* (CMU/SEI-92-TR-20). Pittsburgh, Pa: Software Engineering Institute, Carnegie Mellon University, September 1992.

[Paulk 91] Paulk, Mark et al. *Capability Maturity Model* (CMU/SEI-91-TR-24). Pittsburgh, Pa: Software Engineering Institute, Carnegie Mellon University, August 1991.

[Pfleeger 89] Pfleeger, Shari Lawrence. *Recommendations for an Initial Metrics Set* (Contel Technology Center Technical Report CTC-TR-89-017). Chantilly, Virginia: 1989 (available from GTE).

[Pfleeger 90] Pfleeger, Shari Lawrence; & McGowan, Clement L. "Software Metrics in a Process Maturity Framework". *Journal of Systems and Software*, *12* (July 1990): 255-261.

[Pfleeger 91] Pfleeger, Shari Lawrence. *Software Engineering: The Production of Quality Software, 2nd edition*. New York, New York: Macmillan, 1991.

[Rifkin 91] Rifkin, Stan; & Cox, Charles. *Measurement in Practice* (CMU/SEI-91-TR-16). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1991.

[Schultz 88] Schultz, Herman P. *Software Management Metrics* (ESD-TR-88-001). Bedford, Mass.: The MITRE Corporation, 1988.

[USAF 92] *Software Management Indicators* (Air Force Pamphlet 800-48). Washington, D.C.: Department of the Air Force, 1992.

[Weber 91] Weber, Charles V.; Paulk, Mark C.; Wise, Cynthia J.; & Withey, James V. *Key Practices for the Capability Maturity Model* (CMU/SEI-91-TR-25). Pittsburgh, Pa: Software Engineering Institute, Carnegie Mellon University, August 1991.

# Appendix A: Acronyms and Terms

## A.1. Acronyms

| | |
|---|---|
| **BAFO** | best and final offer |
| **CMM** | Capability Maturity Model |
| **CMU** | Carnegie Mellon University |
| **CDR** | critical design review |
| **CSC** | computer software component |
| **CSCI** | computer software configuration item |
| **CSU** | computer software unit |
| **DACS** | Data & Analysis Center for Software |
| **DISA** | Defense Information Systems Agency |
| **DoD** | Department of Defense |
| **I-CASE** | Integrated Computer-Aided Software Engineering |
| **IV&V** | independent verification and validation |
| **IEEE** | The Institute of Electrical and Electronics Engineers, Inc. |
| **KSLOC** | thousands of source lines of code |
| **PDR** | preliminary design review |
| **QA** | quality assurance |
| **SEI** | Software Engineering Institute |
| **SLOC** | source lines of code |
| **SSR** | software specification review |
| **STARS** | Software Technology for Adaptable, Reliable Systems |
| **SWAP** | Software Action Plan |
| **WBS** | work breakdown structure |

## A.2. Terms Used

**Attribute –** A quality or characteristic of a person or thing. Attributes describe the nature of objects measured.

**Baseline –** A specification or product that has been formally reviewed and agreed upon, which thereafter serves as the basis for future development, and which can be changed only through formal change control procedures.

**Blank comments –** Source lines or source statements that are designated as comments but contain no other visible textual symbols.

**Blank lines –** Lines in a source listing or display that have no visible textual symbols.

**Comments –** Textual strings, lines, or statements that have no effect on compiler or program operations. Usually designated or delimited by special symbols. Omitting or changing comments has no effect on program logic or data structures.

**Compiler directives –** Instructions to compilers, preprocessors, or translators. Usually designated by special symbols or keywords.

**Computer software component (CSC) –** A distinct part of a computer software configuration item (CSCI). CSCs may be further decomposed into other CSCs and computer software units (CSUs) [DOD-STD-2167A].

**Computer software configuration item (CSCI) –** A configuration item for software [DOD-STD-2167A].

**Computer software unit (CSU) –** An element specified in the design of a computer software component (CSC) that is separately testable [DOD-STD-2167A].

**Declarations –** A non-executable program statement that affects the assembler's or compiler's interpretation of other statements in the program. Examples include type and bounds declarations, variable definitions, declarations of constants, static initializations, procedure headers and argument lists, function declarations, task declarations, package declarations, interface specifications, generic declarations, and generic instantiations.

**Defect –** (1) Any unintended characteristic that impairs the utility or worth of an item, (2) Any kind of shortcoming, imperfection or deficiency, (3) Any flaw or imperfection in a software work product or software process. Examples include such things as mistakes, omissions and imperfections in software artifacts, or faults contained in software sufficiently mature for test or operation. See also **fault.**

**Delivered statements –** Statements that are delivered to a customer as part of or along with a software product. There are two subclasses: (1) statements delivered in source form and (2) statements delivered in executable form but not as source.

**Duplicate problem –** A problem encountered in a software product for which a problem report has already been created.

**Embedded statement –** A statement used within or as an argument to another or inserted between begin/end markers.

**Error –** (1) differences between computed, observed, or measured values and the true, specified, or theoretically correct value or conditions, (2) an incorrect step process or data definition, (3) an incorrect result, (4) a human action that produces an incorrect result. Distinguished by using "error" for (1), "fault" for (2), "failure" for (3), and "mistake" for (4). (IEEE 610.12-1990). See also **failure**, **fault** and **mistake.**

**Executable statement –** A statement that produces runtime actions or controls program flow.

**Failure –** The inability of a system or component to perform its required functions within specified performance requirements (IEEE 610.12-1990).

**Fault –** (1) a defect in a hardware device or component, (2) an incorrect step in a process or data definition in a computer program (IEEE 610.12-1990).

**Format statement –** A statement that provides information (data) for formatting or editing inputs or outputs.

**Logical source statement –** A single software instruction, having a defined beginning and ending independent of any relationship to the physical lines on which it is recorded or printed. Logical source statements are used to measure software size in ways that are independent of the physical formats in which the instructions appear.

**Measure –** *n*. A standard or unit of measurement; the extent, dimensions, capacity, etc. of anything, especially as determined by a standard; an act or process of measuring; a result of measurement. *v*. To ascertain the quantity, mass, extent, or degree of something in terms of a standard unit or fixed amount, usually by means of an instrument or process; to compute the size of something from dimensional measurements; to estimate the extent, strength, worth, or character of something; to take measurements.

**Measurement –** The act or process of measuring something. Also a result, such as a figure expressing the extent or value that is obtained by measuring.

**Mistake (software) –** Human action that was taken during software development or maintenance and that produced an incorrect result. A software defect is a manifestation of a mistake. Examples are (1) typographical or syntactical mistakes (2) mistakes in the application of judgment, knowledge, or experience (3) mistakes made due to inadequacies of the development process.

**Module –** (1) A program unit that is discrete and identifiable with respect to other units (2) A logically separable part of a program (adapted from ANSI/IEEE 729-1983). (This is comparable to a CSU as defined in DOD-STD-2167A.)

**Nondelivered statements –** Statements developed in support of the final product, but not delivered to the customer.

**Origin –** An attribute that identifies the prior form, if any, upon which product software is based.

**Problem report –** A document or set of documents (electronic or hard copy) used to recognize, record, track, and close problems. (Sometimes referred to as trouble reports, discrepancy reports, anomaly reports, etc.).

**Problem (software) –** A human encounter with software that causes a difficulty, doubt, or uncertainty with the use of or examination of the software. Examples include: (1) a difficulty encountered with a software product or software work product resulting from an apparent failure, misuse, misunderstanding, or inadequacy (2) a perception that the software product or software work product is not behaving or responding according to specification (3) an observation that the software product or software work product is lacking function or capability needed to complete a task or work effort.

**Software life cycle –** The period of time that begins when a software product is conceived and ends when the software is no longer available for use. Typically includes following stages or phases: concept, requirements, design, implementation, test, installation and checkout, operation and maintenance, and retirement (IEEE 610.12-1990).

**Software product –** The complete set, or any of the individual items of the set, of computer programs, procedures, and associated documentation and data designated for delivery to a customer or end-user (IEEE 610.12-1990).

**Software work product –** Any artifact created as part of the software process, including computer programs, plans, procedures, and associated documentation and data, that may not be intended for delivery to a customer or end-user [Weber 91].

**Source statements –** Instructions or other textual symbols, either written by people or intended to be read by people, that direct the compilation or operation of a computer-based system.

**Staff-hour –** An hour of time expended by a member of the staff [P1045/D4.0].

**Statement type –** An attribute that classifies source statements and source lines of code by the principal functions that they perform.

# Appendix B: Illustrations of Use

The purpose of this chapter is to illustrate some ways the measures in this report can be used to provide early warnings of potential problems, generate reliable projections for the future, or suggest and evaluate process improvements. The examples are organized into five sections:

1. The first section discusses the use of the measures in establishing the feasibility of size-schedule-cost parameters prior to the start of a project.
2. The second section gives examples of information that can be obtained from project plans. Some patterns that characterize high-risk projects are shown.
3. The third section discusses the use of measurements to track projects. Some identifiable symptoms that point to a project at risk are illustrated. This section includes examples of trends that reflect likely problems. It also includes examples of how the measures have been used to provide objective information regarding a project's current status and to generate projections regarding the future.
4. The fourth section discusses the use of the basic measures in process improvement. Examples are presented of using measurement results to streamline maintenance and evaluate the impact of design and code inspections.
5. The fifth section discusses the use of the measures in calibrating cost and reliability models.

Most of the examples in these sections are real. Several have been provided through the generosity of John McGarry (Naval Undersea Warfare Center, Newport), Steve Keller (Dynamics Research Corporation), and Douglas Putnam (Quantitative Software Management, Inc.). These individuals have extensive experience in working with software measurements. Their willingness to share their observations and insights is greatly appreciated. The measurement results in the examples have been changed to prevent identification of the projects in question, but the relative values and trends are shown as they were reported. It is hoped that the examples may help to convey some of the richness contained in the measures recommended in this report.

It is important to add that the measures often need supplementary information to be useful. This supplementary data can characterize the system being developed, the development process, resources, and so forth. The parameters of cost-estimation models provide good examples of the kinds of data that are useful in comparing and calibrating measures across projects and organizations.

## B.1. Establishing Project Feasibility

If a project is to be successful (that is, if it is to deliver the required functionality on schedule, within budget, and with acceptable quality), it has to begin with realistic estimates. Many

projects are in trouble before they ever start. They may be aiming for a level of functionality within a time frame and budget that has never been achieved before.

Software cost models provide an objective basis for determining the feasibility of the planned functionality-effort-schedule combination. Functionality is represented by estimates of size combined with descriptions of complexity, hardware constraints, etc. On many projects the schedule is determined by outside pressures (when the hardware platform will be ready or when the marketing department promised the new release, for example.). The budget may also be determined by outside constraints. Software cost models allow estimators to put these basic project dimensions together to determine whether the project is at all feasible. If it is not, one has the option of down-sizing the functionality, increasing the schedule, or increasing the budget.

Several cost models allow estimators to enter a specific schedule as a constraint and to observe the effect on total effort. Some also allow users to enter effort as a constraint and observe the effect on schedule. Bailey describes and compares the capabilities of seven commercially available software cost models [Bailey 86], and information on capabilities added since that report was published can be obtained from the model vendors. Bailey also includes a survey of the program offices associated with approximately twenty major DoD software acquisitions. It is interesting that fewer than one-third reported that a formal cost model had ever been used in estimating resource requirements.

One of the key issues at the beginning of any software project is schedule. A highly compressed schedule leads to substantially increased effort and costs. A real-life example of this cost-schedule tradeoff is shown in Figure B-1. The data in the figure are from two projects within the same organization. They represent the same types of application (business data processing) and are of similar complexity. Project 1, however, was on an extremely tight schedule, while the time pressures for Project 2 were much more relaxed. At the peak of staffing, Project 1 brought on three times as many people as Project 2 and ended up requiring 50% more staff-hours of effort. Project 1 also experienced a much greater proportion of defects than Project 2.

| Characteristic | Project 1 | Project 2 |
|---|---|---|
| Size (KSLOC) | 73 | 80 |
| Application | Data processing | Data processing |
| Duration (months) | 11 | 24 |
| Peak effort (avg. staff per month) | 46 | 13 |
| Total effort (staff-hours) | 30400 | 20216 |
| Defects at delivery | 20 per KSLOC | 4 per KSLOC |

Note: Projects were of similar complexity

Figure B-1 Illustration of Effects of Schedule Acceleration

If measures like these are available from past projects, they can be used to calibrate several of the available cost models. These models accept historical data that describe projects in terms of size (in source lines of code), total staff-hours or staff-months of effort, and total duration. This use of the software measures is discussed later in Section B.5.

All managers would like to avoid impossible projects. Most would like to avoid (or at least control) risky projects as well. In both cases, examinations of tradeoffs like the one illustrated in Figure B-1 provide a basis for understanding the extent of cost and schedule risk.

## B.2. Evaluating Plans

Much can be learned by looking at project plans, often before any software has been developed. The following examples show how potential problems can sometimes be identified from staffing and schedule plans and from successive size estimates.

### Effort

Effort profiles can provide one useful early indication of project problems. Figure B-2 shows the planned manpower for a project with a very steep ramp-up during preliminary design. It is highly questionable that this number of people can be effectively utilized at this stage of the project. In fact, this project became more than two years late and had a 100% cost overrun. A more typical staff loading is shown in Figure B-3.

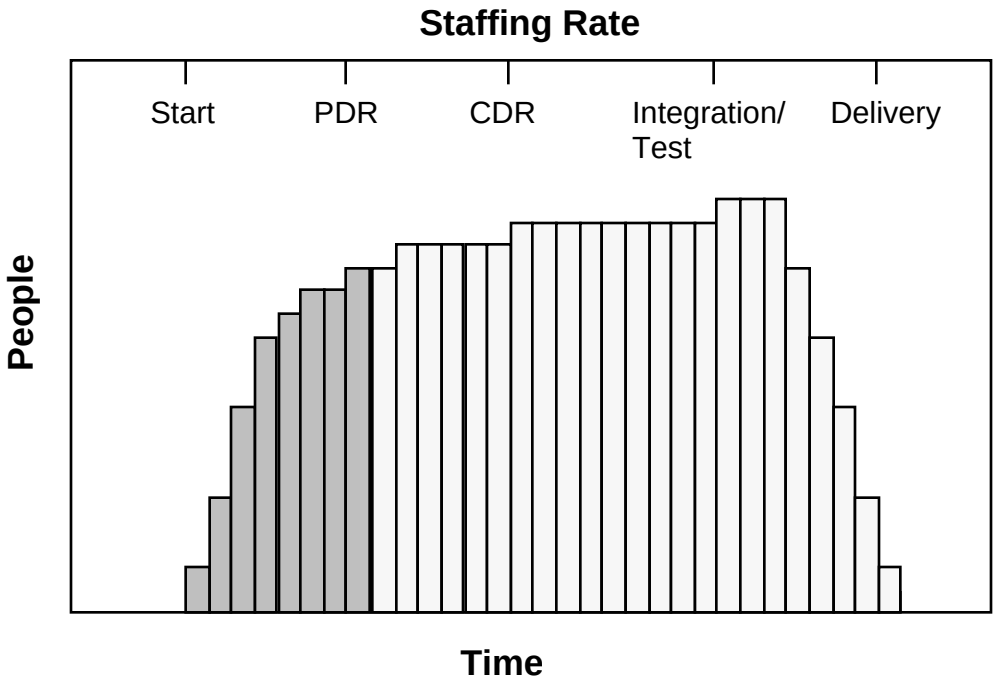

Figure B-2 Indications of Premature Staffing

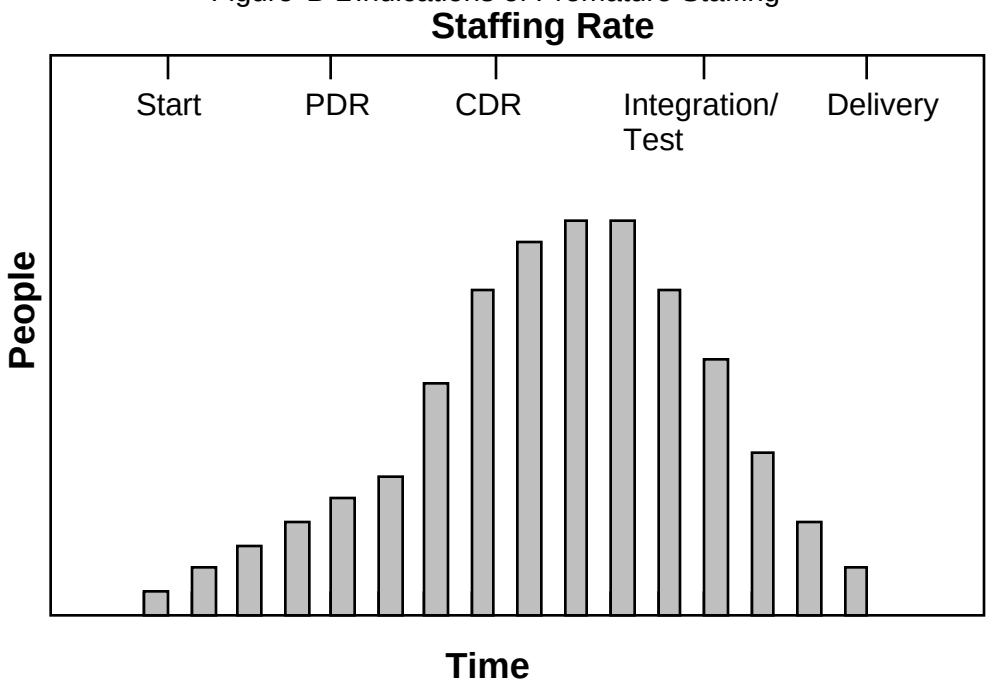


Figure B-3 A More Typical Staffing Profile

## Size

Size is often underestimated early on. A great deal of useful information can be obtained from periodically updating size estimates. As more is understood about the product, the estimates are likely to change. Size growth will have implications for cost and schedule that should be identified and dealt with as early as possible. Figure B-4 shows one such case. In

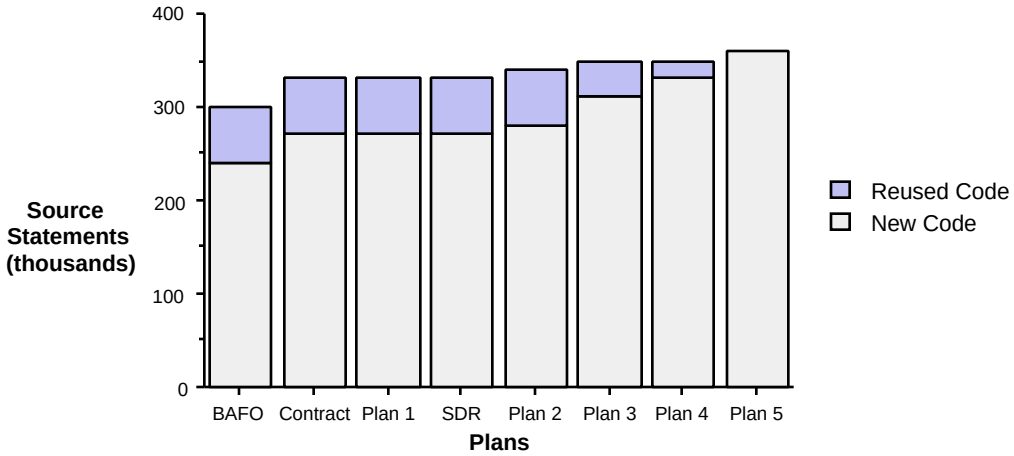


Figure B-4 Exposing Potential Cost Growth—The Disappearance of Reused Code

this example, counts of reused and new code have been extracted from each of a succession of development plans. This contract was bid and won on the basis that there would be substantial reuse of existing code. The first conclusion that we can draw from Figure B-4 is that code growth appears to be nearing 20% and that costs are likely to be similarly affected. The second conclusion is that the situation is actually much worse—all of the promised reuse has disappeared and the forecast for new code development is now up by 50%. If this information has not been reflected in current schedules and cost estimates, some serious questions should be asked.

The next three figures demonstrate the importance of keeping and comparing earlier sets of plans. Figures B-5, B-6, and B-7 show three views of the same project. Figure B-5 shows coding progress plotted against the third version of the development plan. If this is all the information we have, we might infer that the project was pretty much on schedule through month 10 but that it has now started to fall behind and may require some minor adjustments or replanning to bring it back onto track.

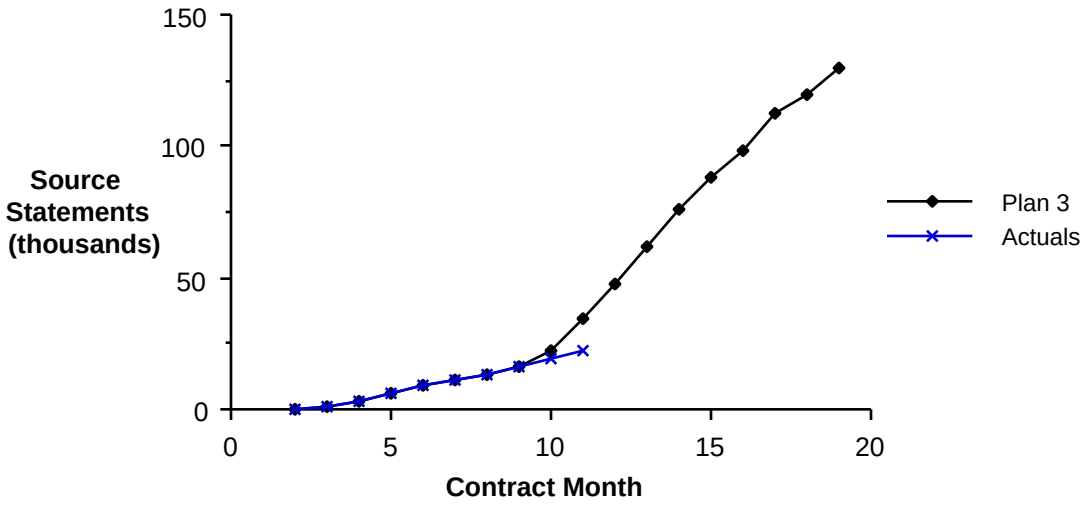


Figure B-5 Project Tracking—The Deviations May Seem Manageable

Figure B-6, however, suggests that the problems may be more serious than Figure B-5 suggests. This figure compares actual progress with the projections that were made in the original plan. Major shortfalls are now apparent. After seeing Figure B-6, we would certainly want to ask about the replanning that has occurred since the original plan and the impact that the departures from the original plan will have on projected costs and schedules.

Interestingly, we do not have to wait for actual measurements from the development organization to gain much of the insight we seek. In fact, we can obtain this insight even earlier. As Figure B-7 shows, if we simply plot the data from each of the developer's plans on the same graph, we see that early dates have simply been slipped and that no real schedule replanning has been done.

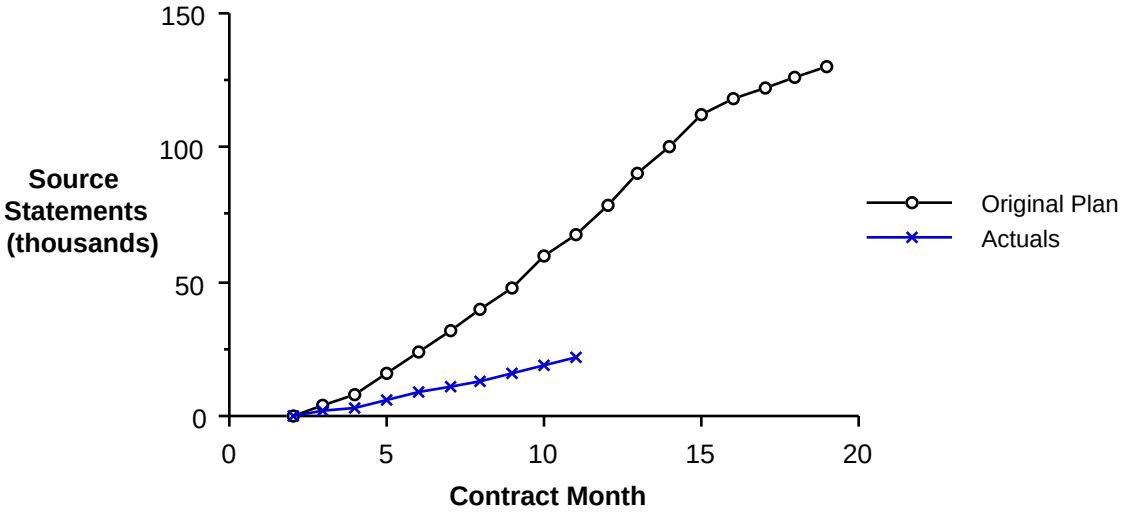


Figure B-6 Project Tracking—Deviations from Original Plan Indicate Serious Problems

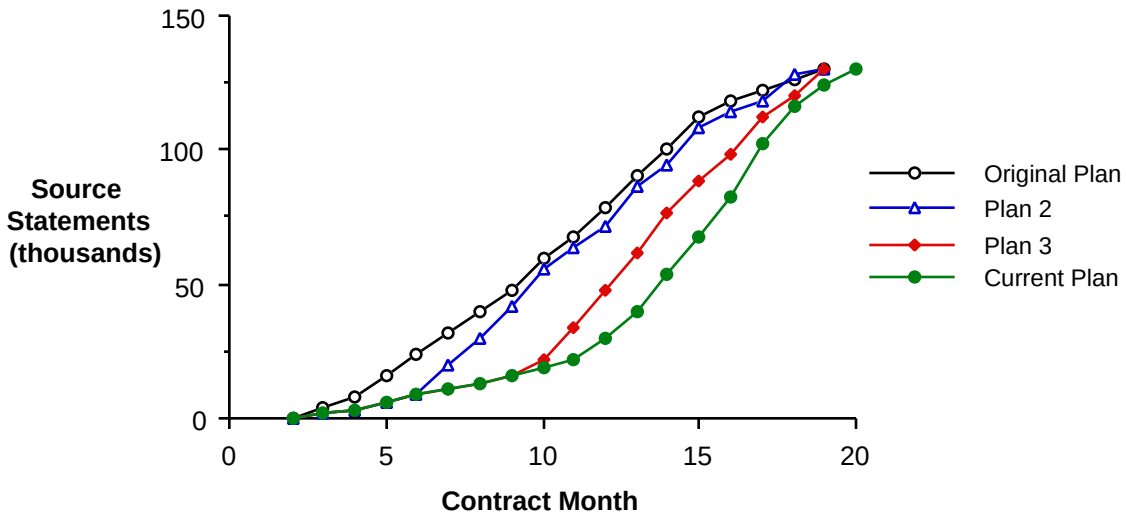


Figure B-7 Project Tracking—Comparisons of Developer's Plans Can Give Early Warnings of Problems

Figure B-7 suggests some even more probing questions that might be asked. Since the project has made only minor changes in the planned completion date despite falling significantly below the original profile over the last nine months, there is reason to examine the code production rate that the current plan implies. When we do this, we see that the current plan requires an average rate of 12,000 statements per month for months 12 through 20 to reach the planned completion date. This is highly suspect since the demonstrated

production capability has yet to reach an average rate of even 2,500 statements per month, something considerably less than the 7,600 statements per month that was required for the original plan. It would be interesting at this point to examine the current plan to see how it proposes to more than quadruple the rate of code production. If, in fact, the project sticks with its proposal to use accelerations of this magnitude to meet the original completion date, it may be wise to place increased emphasis on measuring and tracking the quality of the evolving product.

## Schedule

Attempts to compress schedules typically lead to increased risk. As discussed earlier, there are fairly severe limits as to how quickly any project can ramp up in terms of staffing. One common manifestation of a compressed schedule is the type of plan, shown for computer software configuration item number 1 (CSCI 1) in Figure B-8, in which fundamentally sequential activities are run in parallel. Contrast this with the plan for CSCI 2. One would feel much more comfortable with the latter schedule.

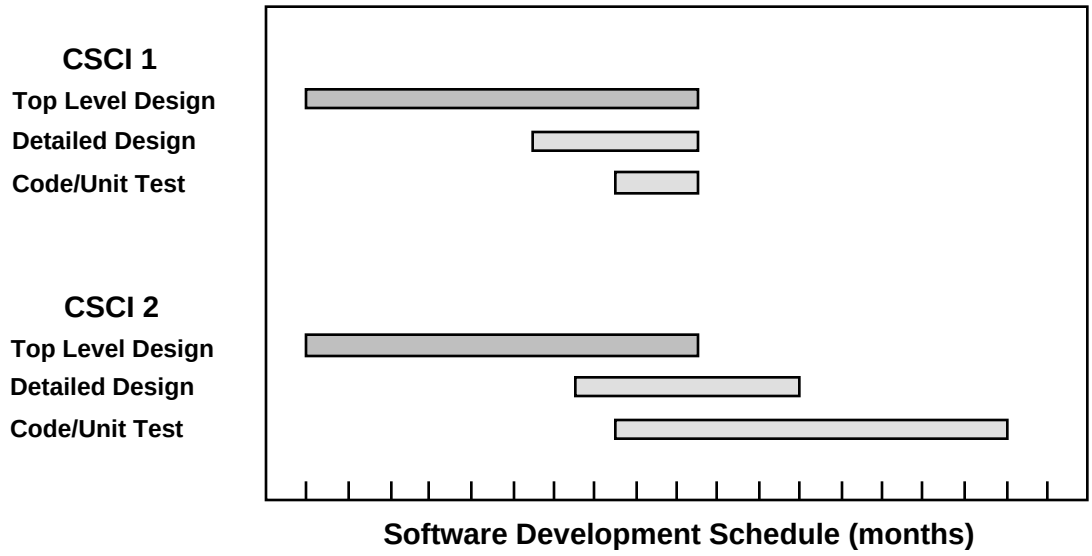


Figure B-8 Comparison of Compressed and Normal Schedules

Another obvious symptom of a project in trouble is a series of continually slipping milestones, with no objective basis for new projections. Figure B-9 shows an example. In each new plan, the scheduled delivery date slipped, with the result that delivery was a continually moving target. The plans were made every two to three months. It is interesting to note that the delivery slipped by almost that amount with each new plan.

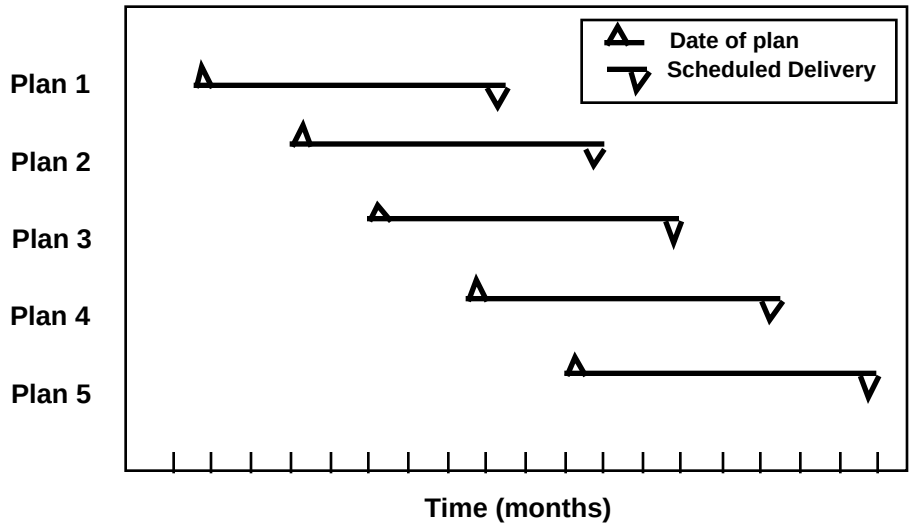


Figure B-9 Continually Slipping Milestones

A third pattern sometimes seen is a series of plans in which intermediate milestones keep slipping without corresponding adjustments in the delivery date. The result is that the amount of time allocated to integration and test gets pinched. When this happens, defect detection rates reach sharp peaks, and backlogs of open problem reports rise rapidly. The detection pattern shown in Figure B-10 is typical for projects in trouble—manpower and defects detected peak during integration and test. These peaks should not be interpreted as something that we put up with to get shorter schedules. Rather, they are indications that considerably more time than anticipated will be required to complete the project. The choices are either a realistic adjustment in the delivery date or the delivery of a product with a high number of residual defects.

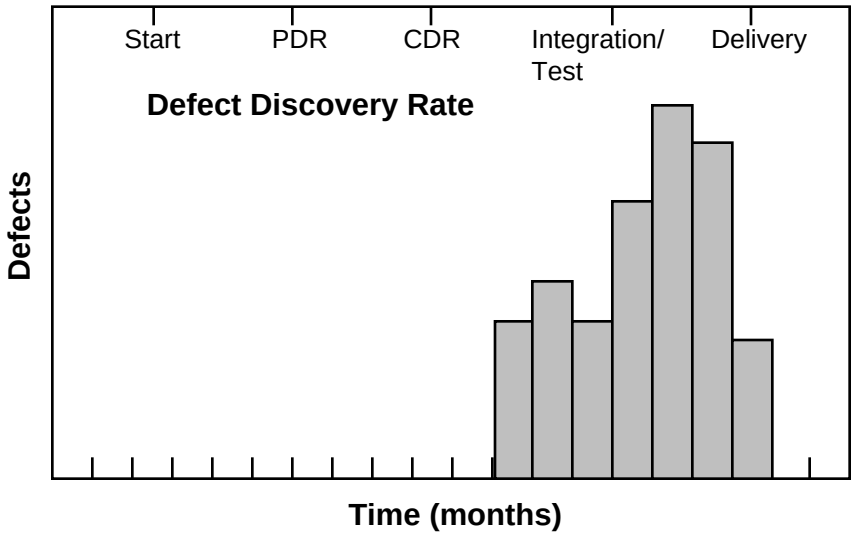


Figure B-10 Effects of Slipping Intermediate Milestones

In cases like this—which are more common than most people like to think—objective progress measures can play a key role in providing a basis for defensible schedule projections. The use of size estimates and projected defect rates and their comparisons with periodic measurement results also provide a basis for this type of analysis. This is illustrated in the next section.

## B.3. Tracking Progress

In managing any project, the following questions are fundamental:

- How much have we done?
- How much is there left to do?
- When will it be completed?

These questions can be answered for any activity whose outputs or products can be expressed in quantifiable units. For requirements analysis, this could be the number of requirements to be allocated to CSCIs; for preliminary design, it could be the number of external interfaces to be specified; for integration and test, it could be the number of test procedures to be executed. We simply need to estimate the total number of units to be completed and then, at regular time intervals (weekly or monthly), track the actual number completed. We can then generate a production curve for that activity. By extrapolating the curve, we get an objective basis for projecting the completion date for that activity. A simple linear extrapolation is often remarkably accurate [Schultz 88].

Figure B-11 shows an example of this type of analysis for code production. In this case, the total number of physical lines of code was estimated at 120K. The actual number which had completed coding was plotted over a five-month period. An extrapolation made at that point yielded a very accurate projection of when coding would be complete.

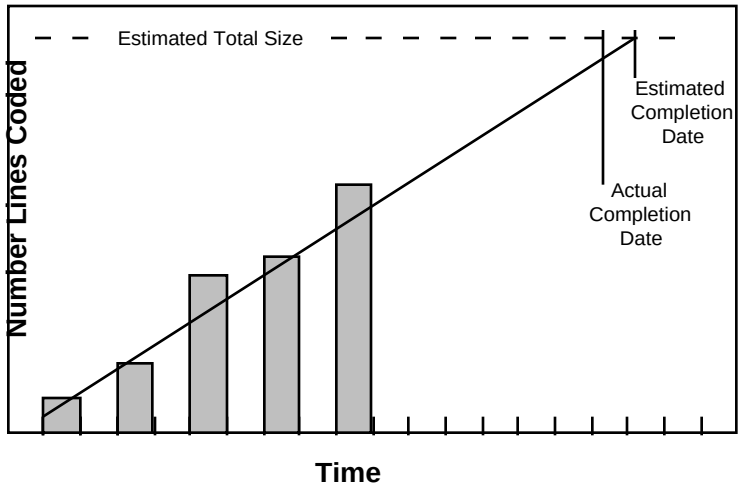


Figure B-11 Extrapolating Measurements to Forecast a Completion Date

This type of analysis is valid only if we have objective criteria for when units are counted as complete. The criteria in this case were that the code had completed unit test and had been entered under configuration control. At that point it was processed by an automated code counter.

We can perform the same analysis for individual CSCIs. If one or more CSCIs are lagging behind the others, we have the option of adding more people or in some other way working to increase the code production rate for the affected CSCIs.

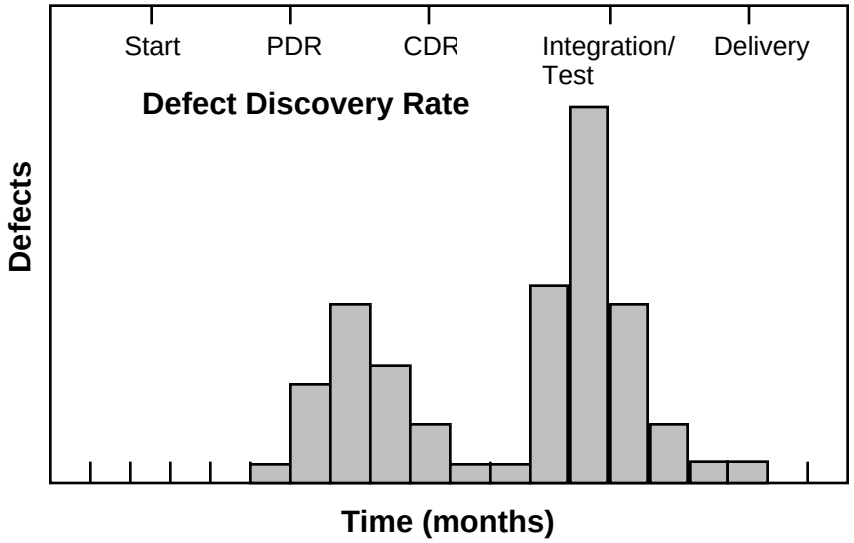


Figure B-12 Effects of Normal Schedules

Another set of measures that provides valuable information for projecting completion dates is the rate of defect discovery, especially during integration and test. Ideally, we would like to see a steady decline in defect discoveries as the schedule delivery date approaches. An example from one project that delivered on schedule is shown in Figure B-12. Contrast this with the pattern shown in Figure B-10, in which the number of defects detected peaked near the end of integration and test.

## B.4. Improving the Process

The measures we have recommended increase our insight into the development and maintenance process. This, in turn, helps us identify bottlenecks and problem areas so that appropriate actions can be taken. The measures also provide a basis for evaluating the impact of changes made to the software process. This section contains two such examples of this kind of use, one from development and the other from maintenance.

## Evaluating the impact of design and code inspections

Problem reports can be used to evaluate the impact of implementing design and code inspections. Figure B-13 shows the profile of problems reported over time for a project within an organization that had implemented inspections in an attempt to find as many defects as early in the process as possible. As can be seen from the figure, the number of defects detected peaks during design and coding activities and drops off quickly to a low level during integration and testing—just as we would hope to see. It is interesting to compare the pattern shown in Figure B-13 with that in Figures B-10 and B-12.

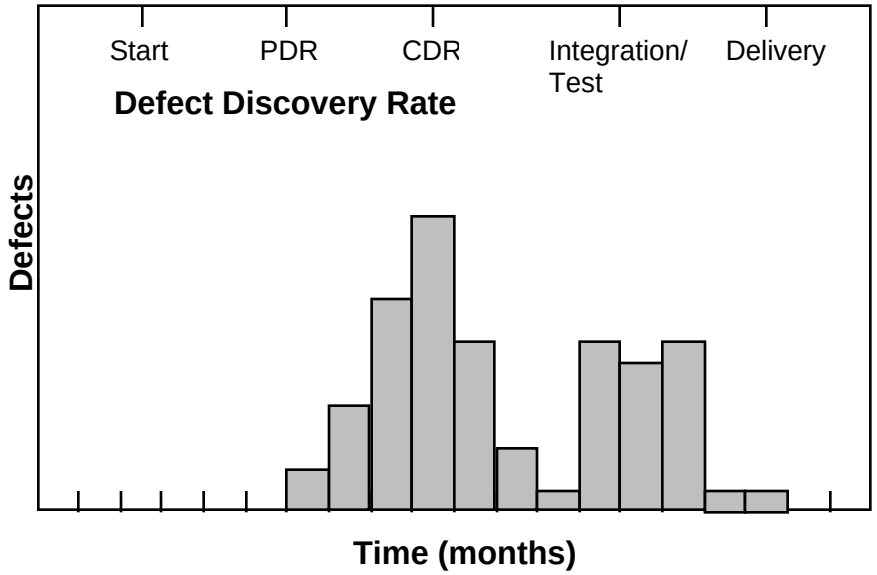


Figure B-13 Effects of Detecting Defects Early

## Improving maintenance

This example comes from the commercial sector. It is an instance where measures were used to bring much needed visibility to software maintenance. The organization that implemented these measures was maintaining fielded versions of their software worldwide. Whenever a problem came in from the field, the organization began tracking which system components were at fault and the amount of time spent isolating and fixing the problem. (This typically involved a work-around for the customer and a revision to the software for the next release.) With information about where the problems occurred and the effort to fix them, the organization was able to identify their most error-prone components and to know exactly how much they were spending to maintain them. This helped them make informed decisions about the costs and benefits of redesigning these components.

## B.5. Calibrating Cost and Reliability Models

The final use for basic software measures is in calibrating cost models. As noted earlier in Section B.1, several commercially available cost models allow estimators to use historical size-effort-duration data to calibrate their underlying estimating algorithms. Some models use this information to compute adjusted values for coefficients and exponents. Others use the data to derive tailored values for parameters that characterize resources, productivities, product difficulties, or organizational capabilities in more complex ways. In either case, once these values are determined from projects that organizations have actually completed, we have baselines that can be used as references to help make future estimates consistent with demonstrated past performance.

To be valid, calibration of cost models requires full knowledge of the definitions and coverages used when collecting and reporting effort, schedule, and size measurements. Calibration is greatly assisted when consistent definitions and measurement rules are used across projects and organizations. The checklists in the SEI framework reports should prove invaluable for both of these purposes.

# Appendix C: A Proposed DoD Software Measurement Strategy

This appendix shows where the SEI basic measurement set fits in the context of other DoD measurement activities. Figure C-1 is from the DoD SWAP Working Group. It shows a strategy that members of the working group have proposed for building on the measurement work described in the SEI framework reports and relates the activities to calendar years.

## Proposed DoD Software Measurement Strategy

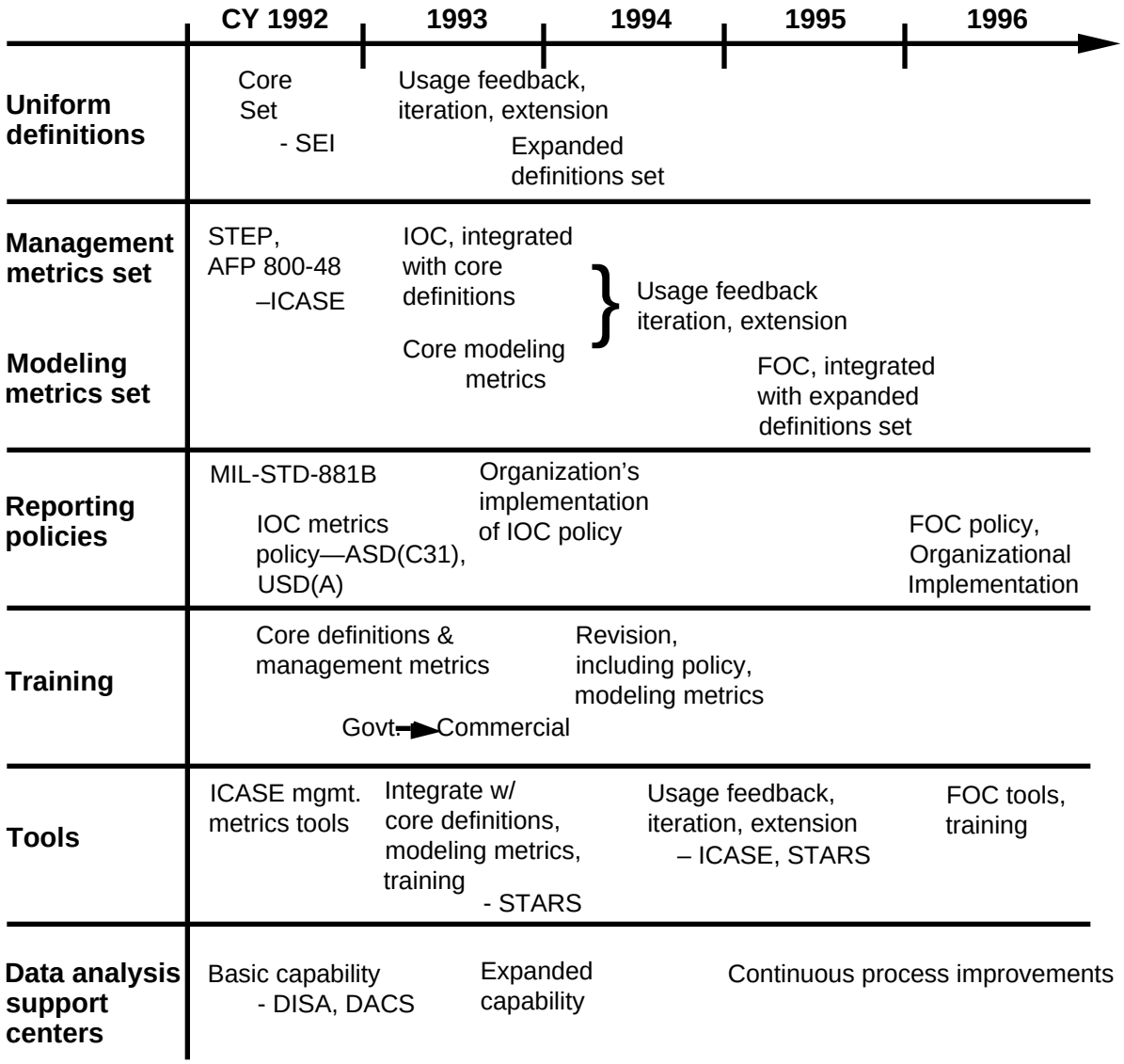


Figure C-1Context for Initial Core Measures