

**Software Measurement for  
Semiconductor Manufacturing  
Equipment**

**SEMATECH** and the **SEMATECH logo** are registered service marks of SEMATECH, Inc.

# Software Measurement for Semiconductor Manufacturing Equipment

Technology Transfer # 95012684A-TR

**SEMATECH**

*March 31, 1995*

**Abstract:** This document defines an introductory set of software metrics to be used by equipment and software suppliers to SEMATECH member companies. These metrics—developed by SEMATECH’s Software Process Improvement (SPI) Project—are designed to measure the size, effort, progress to schedule, and product quality for software projects. The document identifies four basic measures for evaluating software systems; outlines actions for implementing the measures; and illustrates how they can be used to provide early warnings, generate reliable projections, and suggest and evaluate process improvements. The metrics are designed to help supplier companies plan, manage, and improve the lifecycle process of software systems. Although specific actions will vary at individual sites, some general recommendations for use of the metrics are provided.

**Keywords:** Software Development, Software Development Life Cycle, Software Reliability, Software Reuse, Metrology

**Authors:** Herb Krasner and Fred Langner

**Approvals:** Fred Langner, Author  
Harvey Wohlwend, Project Manager  
Dan McGowan, Technical Information Transfer Team Leader



## Table of Contents

1 EXECUTIVE SUMMARY.....	1
1.1 Size.....	1
1.2 Effort.....	1
1.3 Schedule/Progress.....	1
1.4 Quality.....	1
1.5 Using the Metrics.....	1
2 INTRODUCTION.....	3
2.1 Purpose.....	3
2.2 Software Measurement and the SPI Objective.....	3
2.3 The View of Software Management.....	4
3 INTEGRATING MEASUREMENT WITH SOFTWARE PROCESSES.....	5
3.1 Defining the Measurement Process.....	6
3.2 Measurement and the Software Capability Maturity Model.....	7
4 SPECIFIC SOFTWARE MEASURES TO BE USED.....	9
4.1 Software Size.....	9
4.2 Effort.....	10
4.3 Schedule/Progress.....	11
4.4 Quality.....	13
5 IMPLEMENTING THE BASIC MEASURES.....	15
6 BASIC MEASUREMENT USES.....	16
6.1 Establishing Project Feasibility.....	16
6.2 Evaluating Plans.....	16
6.2.1 Effort.....	17
6.2.2 Size.....	17
6.2.3 Schedule.....	18
6.3 Tracking Progress.....	20
6.4 Improving the Process.....	21
6.4.1 Evaluating the Impact of Design and Code Inspections.....	21
6.4.2 Improving Maintenance.....	22
6.5 Calibrating Cost Models.....	22
7 RECOMMENDATIONS.....	23
8 REFERENCES.....	24

## List of Figures

Figure 1 Basili's Goal-Question-Metric (GQM) Paradigm.....	5
Figure 2 Steps to Establish a Measurement Program .....	6
Figure 3 Stages of a Measurement Process.....	7
Figure 4 The Case of Disappearing Reuse.....	10
Figure 5 Staffing Profile .....	17
Figure 6 Exposing Potential Cost Growth from Disappearing Code Reuse .....	17
Figure 7 Deviations from Original Plan Indicate Problems.....	18
Figure 8 Comparison of Compressed and Normal Schedules .....	18
Figure 9 Continually Slipping Milestones .....	19
Figure 10 Effects of Slipping Intermediate Milestones .....	19
Figure 11 Extrapolating Measurements to Forecast a Completion Date .....	20
Figure 12 Effects of Normal Schedules .....	21
Figure 13 Effects of Early Defect Detection.....	21
Figure 14 Declining Defect Density.....	22

## List of Tables

Table 1 Relationship of Software Measures to Process Maturity .....	8
Table 2 Measures for Initial Implementation.....	9

## Acknowledgements

This report is based on work of the Software Process Measurements Project at the Software Engineering Institute (SEI) at Carnegie Mellon University and specifically on that group's document, *Software Measurement for DoD Systems: Recommendations for Initial Core Measures* [5] (Technical Report CMU/SEI-92-TR-19).

SEMATECH wishes to thank the following SEI professionals for helping advance software measurement practices: Anita D. Carleton (project leader), Robert E. Park, Wolfhart B. Goethert, Elizabeth K. Bailey, Mary Busby, William A. Florac, John Baumert, Mark McWhinney, James Rozum, and Shari Lawrence Pfleeger. Their work was interpreted and tailored for use in this report by Herb Krasner of Krasner Consulting, with revisions by Dr. David E. Peercy of Sandia National Labs (SETEC), and Fred Langner and Harvey Wohlwend of SEMATECH's Software Process Improvement (SPI) project.

## **1 EXECUTIVE SUMMARY**

This report defines an introductory set of software metrics to be used by equipment and software suppliers to SEMATECH member companies. Four metrics—size, effort, schedule/progress, and quality (defect density)—are recommended for helping plan, manage, and improve the lifecycle process of software systems. These metrics are discussed below.

### **1.1 Size**

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 (i.e., modules). Size measurements can be used to track the status of code from each production process and to capture important trends.

### **1.2 Effort**

Reliable measures for effort are prerequisites to dependable measures of software cost. By tracking human resources assigned to individual tasks and activities, effort measures provide the principal means for managing and controlling costs and schedules. It is recommended that SEMI/SEMATECH organizations adopt staff-hours as the principal measure for effort.

### **1.3 Schedule/Progress**

Schedule and progress are primary project management concerns. Accordingly, it is important for managers to monitor adherence to intermediate milestone dates. Early schedule slips often foreshadow future problems. It is also important to have objective and timely measures of progress that accurately indicate status and that can be used to project completion dates for future milestones.

At minimum, the following information should be planned for and tracked:

- Number of modules completing unit test
- Number of lines of code completing unit test
- Number of modules integrated
- Number of lines of code integrated

### **1.4 Quality**

Determining what a customer or user views as true software quality can be elusive. 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 for software processes and products. These counts allow qualitative description of trends in detection and repair activities. They also allow the tracking of 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 usability.

### **1.5 Using the Metrics**

Within a project, the metrics can be used to

- Understand the present situation
- Standardize the content of future measurements

- Define, collect, and monitor the additional information needed for project planning and tracking

Across a company, metrics can be utilized to

- Understand the historical data
- Obtain consistent data from project to project
- Obtain consistent data over time, while adjusting to the needs and practices of increasing process maturity
- Determine potential process improvement areas that affect several projects

This report presents the rationale, definition, and typical use of each metric. It also explores how measurement-development and process-development fit together and support each other in a given product architecture.

## 2 INTRODUCTION

The SEMATECH Software Process Improvement (SPI) Project is chartered to develop a broad-scale software measurements program for the SEMI/SEMATECH community. The introductory set of metrics presented here is consistent with principles outlined in *SPI Guidelines for Improving Software 3.0* [21], (Technology Transfer #94092541A-ENG). This set supports the growth of an organization's software capability from immaturity to one characterized by repeatable, disciplined projects and disciplined, cross-project software engineering processes. It also provides the basis for data-driven software project management, quality engineering, and continuous process improvement.

### 2.1 Purpose

This document provides a technical summary of the SPI Program initiative for establishing essential software measurement practices within each SEMI/SEMATECH company. It also introduces the recommended initial SPI metrics set.

This document can be used by:

- SEMI/SEMATECH companies wanting to understand and measurably improve their capability to develop software effectively
- SEMATECH SPI project members who are assisting specific SEMI/SEMATECH companies in developing software measurement programs
- SEMATECH member companies wishing to initiate fact-based dialog with equipment and software suppliers on software improvement

This document assumes the reader has some knowledge and experience in developing and/or maintaining software and understands the problems faced by users and developers of software-controlled systems.

### 2.2 Software Measurement and the SPI Objective

Within a given company, organizational software improvement typically involves the achievement of the following goals:

- Delivering ever-improving software quality to customers/users while reducing cycle time, cost, and defects
- Increasing the overall maturity, productivity, and effectiveness of the software engineering process (including both development and support)

The primary mechanism for accomplishing these goals is a structured and institutionalized program of continuous SPI, reinforced with metrics. Developing such a program requires a clear understanding of software engineering capabilities and a baseline for measuring improvements. Quantitative information enables goals to be established and progress to be monitored.

In fall 1992, the SPI Project conducted a survey on the state of software engineering among equipment and software suppliers to the semiconductor industry (See *Software Process Improvement (SPI) Issues Survey Findings Report*, Technology Transfer #93011489A-TR) [16].

The survey formed the following conclusions:

- Most suppliers do not know basic facts (such as cost and defects) about their software-development efforts.
- There is little understanding of the components of software-development costs and profits and apparent uncertainty about what should be measured.
- Little agreement exists on how to measure the software process or product quality. Except for “ease of use,” there is little agreement on what characterizes good software quality.
- Software-project performance is not closely controlled in most companies, and all companies have difficulty accommodating change.
- Most companies’ software processes are immature, with few (if any) management controls, no metrics other than milestone progress, and very little management awareness of their inner workings.
- The arrival of the ship date is the most common criterion for determining if software is ready for delivery.

The survey found that few SEMI/SEMATECH companies have comprehensive, clearly defined software measurement programs. Those that do, use different measurement and reporting systems, making effective comparisons across the community impossible.

To address these deficiencies, the SPI project has prepared this report recommending a set of basic metrics to help the SEMI/SEMATECH community plan, monitor, and manage its internal and contracted software engineering projects. These metrics will provide a basis for collecting well understood and consistent data throughout the community.

### **2.3 The View of Software Management**

The reason for establishing a measurement program is management’s need for answers to key questions about software-intensive projects:

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

To answer these questions, methods are needed to accurately measure software for size, effort, schedule/progress, and quality. Reliable assessments of these characteristics are crucial to managing project commitments and improving process maturity.

Further, the measurement program provides mechanisms for getting data for the following important management functions:

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

The measurement program outlined in this document is designed for all stages of the software lifecycle, not just for developing new software.

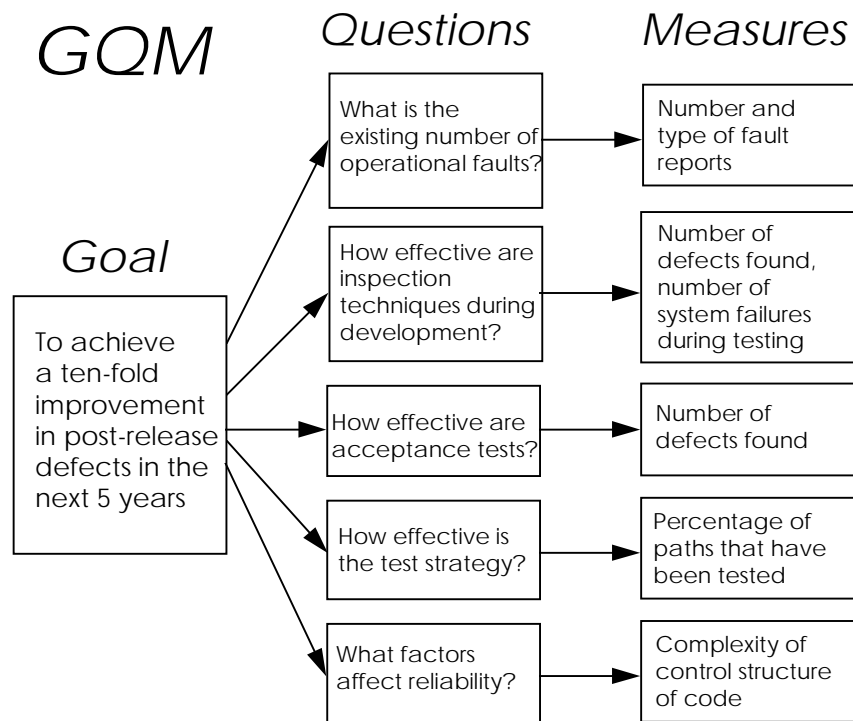
### 3 INTEGRATING MEASUREMENT WITH SOFTWARE PROCESSES

Collecting and using even the most basic measurements in meaningful ways will prove challenging to many organizations. This section describes a strategy for planning a software measurement program.

The first step is to answer these basic questions:

- What are the measurement program's goals?
- What data is needed to address these goals?
- Why should data be collected at all?
- How will data be collected, validated, and stored?
- How will data be analyzed?

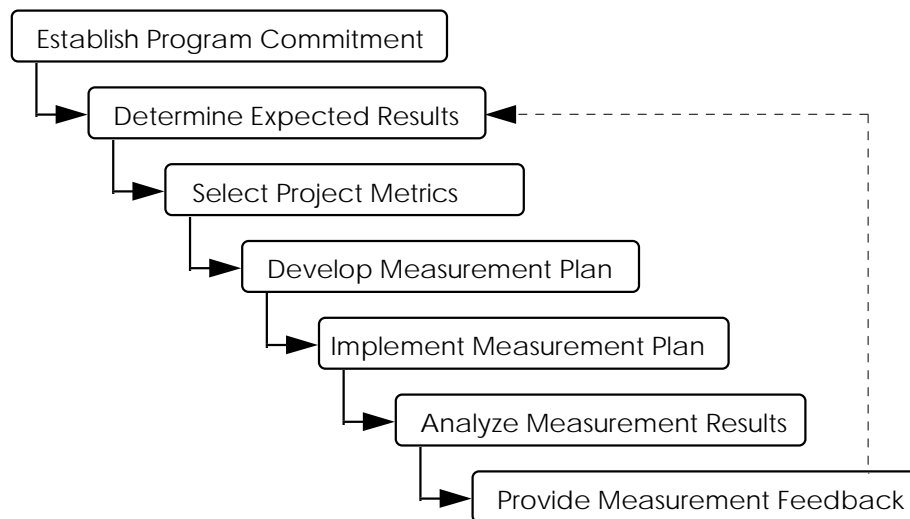
Basili's Goal-Question-Metric (GQM) paradigm [1] can be used to answer some of these questions. Figure 1 shows an example related to software reliability.



**Figure 1 Basili's Goal-Question-Metric (GQM) Paradigm**

### 3.1 Defining the Measurement Process

Figure 2 shows the typical flow of activities a company should perform to establish a software measurement program. These steps are closely aligned with those in *SPI Guidelines for Improving Software: Release 3.0*, [21] Technology Transfer #94092541A-ENG.

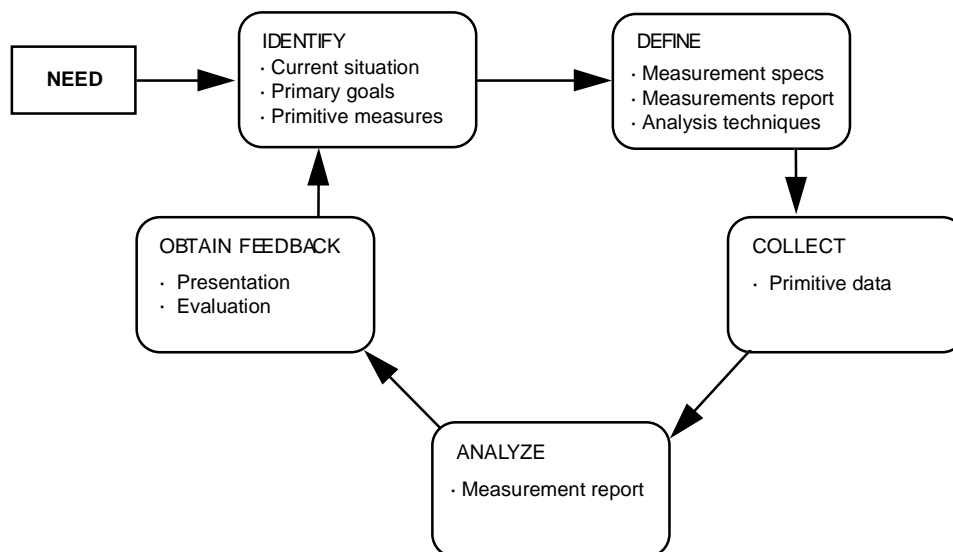


**Figure 2 Steps to Establish a Measurement Program**

These steps are explained as follows:

1. Establish Program Commitment: Define why the program is needed, obtain management approval, identify ownership.
2. Determine Expected Results: Use SPI Guidelines to set improvement context.
3. Select Project Metrics: Apply GQM method to derive project measures.
4. Develop Measurement Plan: For a specific project, define metrics to be collected, data collection, analysis and presentation methods, and relationship to overall improvement program.
5. Implement Measurement Plan: Collect and analyze data, provide project feedback, and modify project/program as necessary.
6. Analyze Measurement Results: Store project measurement results, analyze results against historical project results.
7. Provide Measurement Feedback: Report results of analysis as project lessons learned, update measurement program, and repeat the process, beginning with step 2.

Figure 3 summarizes the stages of a successfully implemented measurement process. The organization begins by defining its specific needs and identifying the measurements it must collect. In doing so, the organization describes its current situation and establishes primary goals and corresponding primitive measures to determine progress toward the goals. Such careful analysis of needs is necessary so that meaningful data can be collected to aid later decision-making.



**Figure 3 Stages of a Measurement Process**

Once identified, the primitive measures must be clearly defined. The “Identify” stage identifies measures such as staff-hours or SLOC, while the “Define” stage establishes how to collect these measures, analyze data to establish baselines, and determine progress toward goals. Without this stage, data tends to be inconsistent, unreliable and difficult to compare with that of other projects.

Completion of these first two stages yields a documented baseline measurement plan. The organization can follow this plan to execute the next two stages for collecting and analyzing data. The “Obtain Feedback” stage includes presenting and distributing measurement results and incorporating reviewers' comments into reports. This stage also evaluates the effectiveness of efforts in prior stages to address project goals. The situation and goals are then re-evaluated and the cycle is repeated, much as in the Plan/Do/Check/Act cycle identified in W.E. Deming’s *Out of the Crisis* [7].

### 3.2 Measurement and the Software Capability Maturity Model

Measures help enable real process maturity and are essential to process management. Without metrics, organizations cannot know if they have succeeded in establishing repeatability.

Software engineering literature describes dozens of measures that can be applied to a wide variety of projects, processes, and product attributes [6]. Choosing measures, collecting data, analyzing results, and taking action require time and resources and make sense only when directed toward specific improvement goals. This section describes how software measurement and process maturity interact.

Some software development processes are more mature than others, and evidence of this has been documented [15]. A distinguishing feature of a mature process is the ability of developers and

managers to see and understand activity in the overall software development effort. At the lowest levels of maturity, the process is not well understood. As maturity increases, the process becomes better understood, better defined, and more visible.

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 increase visibility. The Software Capability Maturity Model [19] serves as a guide for determining what to measure first and how to plan an increasingly comprehensive measurement program.

The following classes of measures are suggested for different levels of the Software Capability Maturity Model:

- Level 1 measures provide baselines for comparison as an organization seeks to start improving.
- Level 2 measures focus on project planning and tracking.
- Level 3 measures become increasingly directed toward measuring and comparing the intermediate and final products produced across multiple projects.
- Level 4 measures capture characteristics of the development process to allow control of the individual activities of the process.
- Level 5 processes are mature enough and managed carefully enough to allow measurement to provide feedback for dynamically changing processes across multiple projects.

**Table 1 Relationship of Software Measures to Process Maturity**

<b>Maturity Level</b>	<b>Focus of Measurements</b>
1	Establish baselines for planning and estimating
2	Track and control project
3	Define and quantify products and processes within and across projects
4	Define, quantify, and control subprocesses and elements
5	Dynamically optimize and improve across projects

The initial set of metrics described in this report primarily addresses Level 2 and 3 issues. The measurement program should begin with an examination of the software process currently in use and with a determination of what is visible (i.e., identify the current situation). If one process is more mature than others, tailored measures can enhance the visibility of that process and help meet overall project goals, while basic measures are raising the other processes to a higher level of capability. For example, in a well-defined configuration management environment, it may be appropriate and desirable to track reused elements from their origins to their uses in final products.

Evidence suggests that successful measurement programs start small and grow according to the evolving needs of the organization [10]. 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 Software Capability Maturity Model can be used 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.

## 4 SPECIFIC SOFTWARE MEASURES TO BE USED

This section presents recommendations for implementing a set of four basic measures to evaluate semiconductor manufacturing software systems. Methods for defining and reporting results are provided for each measure. These methods are supported with reasons for using the measures, and recommendations are included for making the measures effective.

Table 2 lists the basic measures and relates them to the units and characteristics they address. Every SEMI/SEMATECH company should use them for acquiring, developing, and maintaining software systems.

**Table 2 Measures for Initial Implementation**

Type of Measure	Unit of measure	Characteristic addressed
Software Size	Counts of physical source lines of code (SLOC)	Size, progress, reuse
Effort	Counts of staff hours expended	Effort, cost, resource utilization
Schedule/Progress	Calendar dates (events/milestones)	Schedule, progress
Quality	Counts of software problems & defects	Quality, acceptability for delivery, improvement trends

Although other metrics also describe software products and processes, the measures listed above are practical, produce meaningful information, and can be defined to promote consistent use.

The following definitions use methods established in SEI literature reports [8], [9], [18] to state what each basic measure includes and excludes. Each measure provides for collecting data on multiple project attributes, since experienced software managers are rarely satisfied with a single number. For example, problems and defects usually are classified according to such attributes as status, type, severity, and priority. Effort is classified by labor class and type of work performed. Schedules are defined by dates and completion criteria, while size measures are aggregated according to programming language, statement type, development status, origin, and production method. To be of value, both estimates and measured values must be collected at regular intervals (weekly or monthly).

Thus, what may first appear to be just a few measures is actually much more. It will be a major accomplishment to implement the uniform collection and use of these measures across the entire SEMI/SEMATECH community. Neither the difficulty nor the value of this task should be underestimated.

### 4.1 Software Size

Some of the more popular and effective measures of software size are physical SLOC, logical source statement (instructions), function points (or feature points), and counts of logical functions or computer software units (i.e., modules).

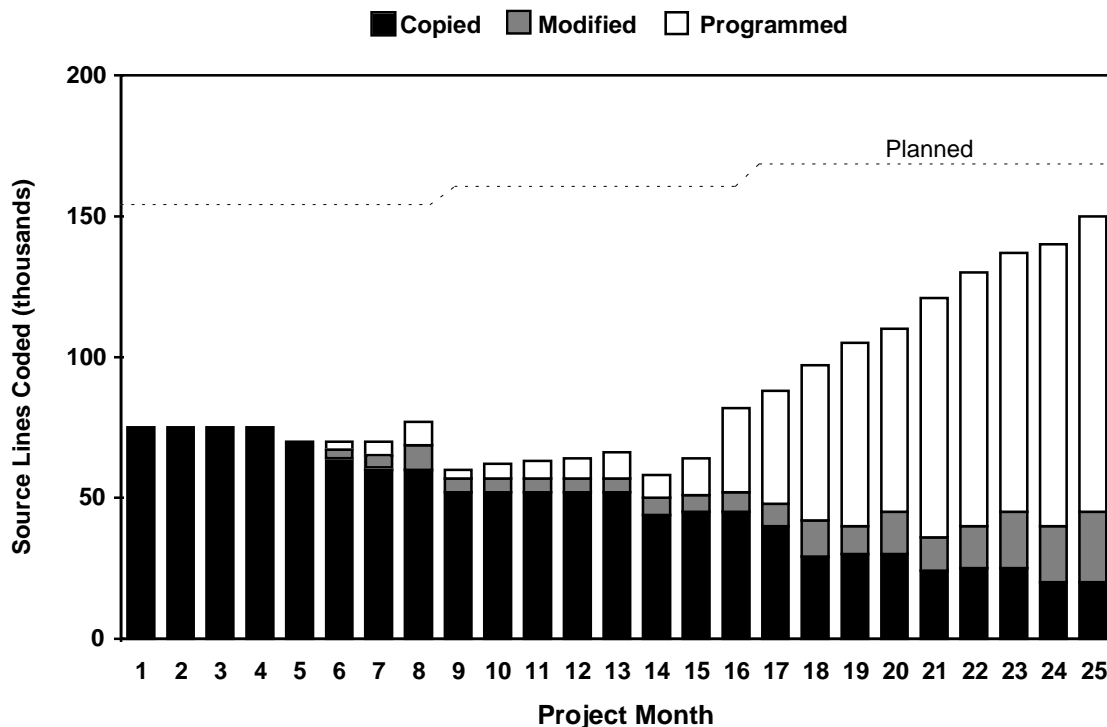
Organizations should adopt physical SLOC (noncomment, nonblank source statements) as a measure of software size for the following reasons:

- SLOC is easy to measure; measurements are made by counting end-of-statement markers.
- Counting methods strongly depend on the programming language used. One need only to specify how to recognize statement types not counted (e.g., comments, blank lines). Automated counters for physical source line measures are available.

- Most historical data for constructing the cost models used for project estimating is based on measures of source code size.
- Empirical evidence to date suggests that counting physical source lines is as effective as any other existing size measure.

An explicit guide for measuring SLOC is provided in *Software Size Measurement: A Framework for Counting Source Statements* [18]. This document spells out rules to be used when comments are on the same lines as other source statements. In addition, it addresses all origins, stages of development, and forms of code production. The definition also 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.

Size measurements can be used to track the status of code from each production process and to capture important trends, as in Figure 4.



**Figure 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 also can be useful in relating progress to schedules.

## 4.2 Effort

Reliable measures for effort are prerequisites to dependable measures of software cost. By tracking the human resources assigned to individual tasks and activities, effort measures also provide the principal means for managing and controlling costs and schedules.

SEMI/SEMATECH organizations should adopt staff-hours as the principal measure for effort. The staff-hour unit in the Institute of Electrical and Electronic Engineers' (IEEE's) draft Standard for Software Productivity Metrics [12] is recommended. *Software Effort & Schedule Measurement: A Framework for Counting Staff-Hours and Reporting Schedule Information* [9] provides a useful staff-hour definition checklist that enables managers to clearly state what is included and excluded.

Although other units for measuring and reporting effort data include labor-months and staff-weeks, the use of staff-hours is preferable for the following reasons:

- No standard exists for the number of hours in a labor-month. Practices vary widely among companies and reported values range from less than 150 to over 170 hours per labor month. Because of contractual requirements, individual organizations may define a labor-month differently for different projects.
- Labor-months often do not provide the granularity needed for measuring and tracking individual activities and processes, particularly when the focus is on process improvement.
- Measuring effort by staff-weeks presents many of the same problems as labor-months, as well as additional ones. For example, although the basic assumption is that a calendar week is five working days, the length of a standard working day varies among organizations. Weekend work, overtime, and holidays must also be addressed and defined if staff-week measures are used.
- Labor-month and staff-week measures still can be calculated from staff-hours, should these measures be needed for presentations or other summaries.

Information about staff hours is used to track resource utilization across tasks and products. It also provides Pareto chart data on cost concentrations for process improvement targets. Section 6 describes more fully the use of this measurement.

Properly collecting effort data requires that a data collection scheme with meaningful categories be established to track estimates and actuals. Thus, software process definition and effort collection are related.

### **4.3 Schedule/Progress**

Schedule and progress are primary project management concerns. When determining the ultimate value of a software product, timely delivery is often as important as functionality or quality. 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. Overly ambitious, unrealistic schedules often result.

Because schedule is a key concern, it is important for managers to monitor adherence to intermediate milestone dates. Early schedule slips often foreshadow future problems. It is also important to have objective and timely measures of progress that accurately indicate status and that can be used to estimate 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. Model developers and estimators must understand what activities the duration includes and excludes. If a project is described as having taken 3 1/2 years, it is reasonable to ask exactly what was included in that period. Did it include system requirements analysis and design, or just software activities? Did the period 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, so they can identify bottlenecks in the process.

Tracking dates for milestones, reviews, audits, and deliverables provides a macro-level view of project schedule. As mentioned, slips in early milestones may indicate future problems. In addition, tracking the progress of activities that culminate in reviews and deliverables can be revealing. Tracking the rate at which underlying units of work are completed provides objective knowledge of where the project is at any given point, and where it will be in the future.

Projects should adopt structured methods [9] for defining two important and related aspects of the schedules they report:

- Dates (both planned and actual) associated with project milestones, reviews, audits, and deliverables
- Exit or completion criteria associated with each date

Reporting the dates associated with the completion criteria helps ensure accuracy in overall reporting. It also provides insight into process timelines that can be useful for planning future projects and for process improvement.

Examples of suggested completion criteria 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
- Product delivered to customer
- Customer comments received
- Changes incorporated
- Customer signoff obtained

Project activities or “phases” are not sufficient. The variations associated with beginning and ending activities make it difficult to define start and end dates precisely. Most activities (e.g., requirements analysis, design, code) continue to some extent throughout the project. Where one project may consider requirements analysis finished with a software specification review, another may consider it to be ongoing throughout development. A second source of ambiguity stems from the fact that some activities start, stop, and restart, making it very difficult to establish meaningful dates.

In contrast to phases, project milestones, reviews, audits, and deliverables should have clear-cut completion criteria linked to specific dates and should be reported separately by product, using the following practices:

- Dates of milestones, reviews, audits, and deliverables
  - Require and report both planned and actual dates.
  - Specify the 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 signoff.
  - Allow some dates to apply to an entire build or system. In other cases, dates should be specified for each software component. For critical components, it may be appropriate to track dates for individual software units or modules.

- 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
  - Specify the measures to be tracked.
  - Require or produce a plan that shows the rate at which work will be accomplished. A plan should be developed for each major software component. Require that the planned and measured values be reported at regular intervals.
  - Require objective completion criteria to make progress measures meaningful. Make sure that these criteria can be audited to be sure that progress is real.

At a minimum, require that the following be planned for and tracked:

- The number of modules completing unit test
- The number of lines of code completing unit test
- The number of modules integrated
- The number of lines of code integrated

#### **4.4 Quality**

Determining what a customer or user views as true software quality can be elusive. 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 for software processes and products. These counts allow qualitative description of trends in detection and repair activities. They also allow the tracking of 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 usability [13].

Defect correction (rework) is a significant cost in most software development and maintenance environments. The number of problems and defects associated with a product contribute directly to this cost. Counting problems and defects can aid understanding of where and how they occur and provide insight into methods for detection, prevention, and prediction. Counting problems and defects also can directly help track project progress, identify process inefficiencies, and forecast obstacles that will jeopardize schedule commitments.

Counts of software problems and defects should be used to help plan and track development and support of software systems. They should be used to help determine when products are ready for delivery to customers and to provide fundamental data for process and product improvement. These counts must be clearly and completely defined.

The SEI report on software quality measurement [8] provides a structure for describing and defining measurable attributes for software problems and defects. It uses a checklist and supporting forms to organize the attributes so that methodical and straightforward descriptions of software problem and defect measurements can be made. These can be used to define or specify a wide variety of problem and defect counts, including those found by static or operational processes (e.g., design reviews or code inspections) or by dynamic or operational processes (e.g., testing or customer operation).

Opportunities abound for using defect and problem reporting to advantage, including the following:

**Old projects.** For projects in the post-release stage, a basic metric is defect logging (software trouble/problem reports), classification, prioritization, resolution, and customer notification. This is a customer service function and is critical to increasing customer satisfaction.

**Ongoing projects.** For projects currently in development and already measuring problems and defects, the data collected should be verified to conform to requirements and needs. This may reveal that the measurements are less than clear and precise in their meaning or that they fall short of what is needed to control the development or maintenance activity.

**New projects.** For projects establishing or expanding 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.

**Serving the needs of many.** Software problems and defect measurements apply directly to estimating, planning, and tracking various parts of the software development process. Users are likely to have different purposes for using and reporting this data. The data 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, justifying equipment or tool investment, and tracing product reliability and responsiveness to customers.

## **5 IMPLEMENTING THE BASIC MEASURES**

This section outlines some priorities and related actions for implementing the basic measures discussed in Section 4.

When implementing the measurement program, first priority should go to formally describing information currently being reported. With clear descriptions for measurement results, misunderstandings can be minimized and inappropriate decisions avoided. Descriptions of existing measures can be obtained quickly at little cost. Standardizing measurement definitions across projects will require more in the way of guidelines, training, and user support.

Within a project, the metrics will be used to

1. Understand the present data you are collecting
2. Standardize the content of future measurement reports
3. Define and collect the additional information needed for project planning and tracking

Across a company, metrics will be used to

1. Understand historical data, and use for estimating and planning future projects
2. Obtain consistent data from project to project (ensure that the same definitions are applied to all projects)
3. Obtain consistent data over time, while adjusting to increasing process maturity
4. Determine potential process improvement areas that span several projects

## **6 BASIC MEASUREMENT USES**

This section illustrates ways the recommended measures can be used to provide early warnings of potential problems, generate reliable projections, or suggest and evaluate process improvements. The subsections discuss the following:

1. Use of the measures to establish the feasibility of size/schedule/cost parameters prior to the start of a project.
2. Examples of information that can be obtained from project plans. Some patterns that characterize high-risk projects are shown.
3. Use of measurements to track projects. Some identifiable symptoms that point to a project at risk are illustrated. Included are examples of trends that reflect likely problems. Also provided are examples of how the measures have been used to provide objective information on a project's current status and to generate projections regarding the future.
4. Use of the basic measures in process improvement. Examples are presented using measurement results to streamline maintenance and evaluate the impact of design and code inspection.
5. Use of the measures in calibrating cost models.

### **6.1 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 must begin with realistic estimates. Many projects are in trouble before they start. They may aim for a level of functionality within a schedule and budget that has never been achieved.

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 (for example, when the hardware will be ready or when the marketing department promised the new release). The budget also may be determined by outside constraints. Software cost models allow estimators to combine these basic project dimensions to determine whether the project is feasible. If not, one can reduce the functionality, expand the schedule, or increase the budget.

Several cost models allow estimators to enter a specific schedule as a constraint and to observe its effect on total effort. Some also allow users to enter effort as a constraint and observe the effect on schedule.

A key issue at the beginning of any software project is schedule. A highly compressed schedule leads to substantially increased effort and costs. If such measures are available from past projects, they can be used to calibrate several available cost models. These models accept historical data that describe projects in terms of size (in SLOC), total staff-hours or staff-months of effort, and total duration.

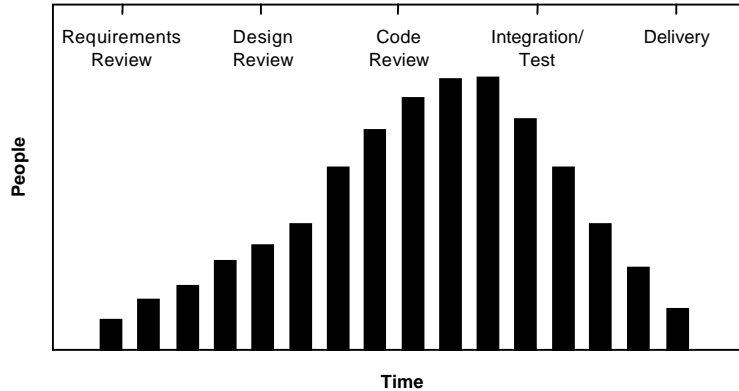
Managers like to avoid impossible projects and control risky projects. In both cases, examinations of tradeoffs using software cost models provide a basis for understanding the extent of cost and schedule risk.

### **6.2 Evaluating Plans**

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

### 6.2.1 Effort

Effort profiles, such as the example in Figure 5, can provide early indication of project problems. Project managers should beware of steep ramp-up curves and of throwing extra people at troubled projects. They are subject to “Brooks Law” [4]: *Adding manpower to a late project can make it later.*

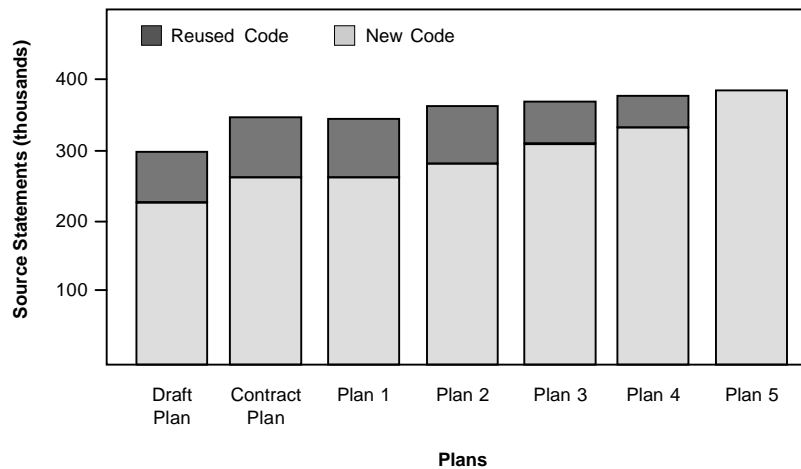


**Figure 5 Staffing Profile**

### 6.2.2 Size

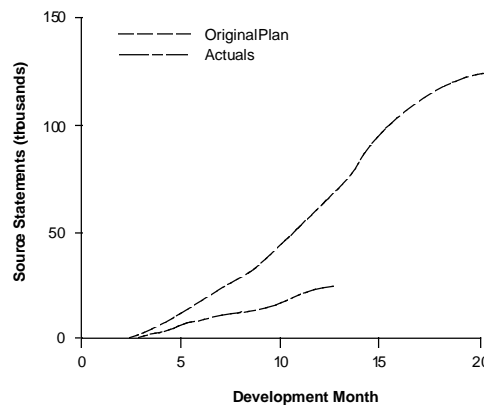
Size is often underestimated early in a project. A great deal of useful information can be obtained from using historical data from similar projects and periodically updating size estimates. As more is understood about the product, the estimates are likely to change. Size growth will impact cost and schedule in ways that should be identified and dealt with as early as possible.

Figure 6 shows one such case. In this example, counts of reused and new code have been extracted from each of a series of development plans. This project was planned so that there would be substantial reuse of existing code. Figure 6 indicates that code growth appears to be nearing 20% for each new forecast, with costs likely to rise similarly. It also indicates that the situation is actually much worse: by the final plan, all of the planned reuse has disappeared and the forecast for new code development is up by 50%. If this information is not reflected in current schedules and cost estimates, serious questions should be asked.



**Figure 6 Exposing Potential Cost Growth from Disappearing Code Reuse**

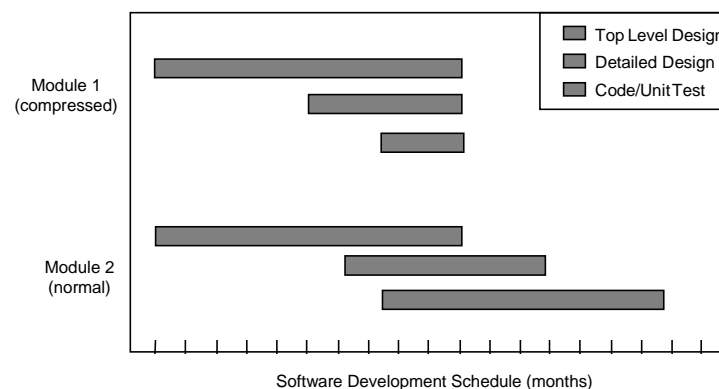
Figure 7 demonstrates the importance of keeping and comparing earlier sets of plans and inspires some probing questions. Since the project has made only minor changes in the planned completion date despite falling significantly below the original profile over the last nine months, there is reason to examine the code production rate that the current plan implies. Upon doing this, it is obvious 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 were required in the original plan. At this point, it is appropriate to question how the rate of code production could be more than quadrupled under the current plan. If the project continues to rely on accelerations of this magnitude to meet the original completion date, it may be wise to increase emphasis on measuring and tracking the quality of the evolving product.



**Figure 7** Deviations from Original Plan Indicate Problems

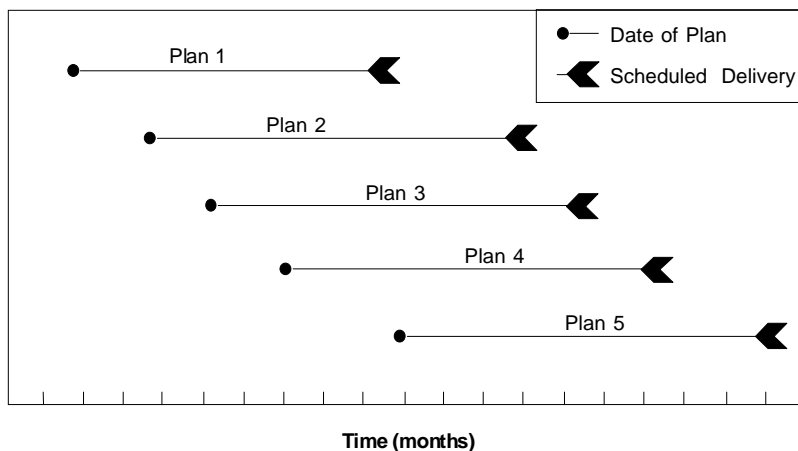
### 6.2.3 Schedule

Attempts to compress schedules typically lead to increased risk. As previously described, fairly severe limits exist as to how quickly any project can ramp up in terms of staffing. A common manifestation of a compressed schedule is the type of plan, illustrated for computer software Module 1 in Figure 8, in which fundamentally sequential activities run in parallel. Contrast this with the plan for Module 2. The latter schedule is much more desirable.



**Figure 8** Comparison of Compressed and Normal Schedules

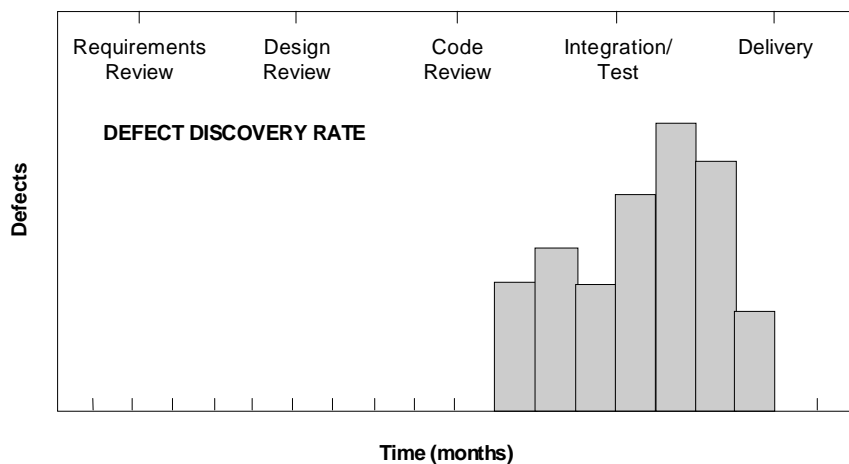
Another obvious symptom of a project in trouble is a series of continually slipping milestones, with no objective basis for new projections, as shown in Figure 9. In each new plan, the scheduled delivery date slips, resulting in a continually moving delivery date. In this case, new plans were made every two to three months, with each new delivery date slipping by about the same amount of time.



**Figure 9** Continually Slipping Milestones

A third pattern is seen in a series of plans in which intermediate milestones keep slipping without corresponding adjustments to the delivery date. The result is that the amount of time allocated to software integration and test gets compressed. When this happens, defect detection rates reach sharp peaks and backlogs of open problem reports rise rapidly.

The detection pattern shown in Figure 10 is typical for projects in trouble: manpower and detected defects peak during integration and test. These peaks should not be interpreted as phenomena to be tolerated when shortening schedules. Rather, they indicate that considerably more time than anticipated will be required to complete the software. The choices are realistically adjusting the delivery date or delivering a product with a high number of residual defects. The latter seems to be common practice in the SEMI/SEMATECH community [16].



**Figure 10** Effects of Slipping Intermediate Milestones

In cases where integration and test steps are compressed, 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.

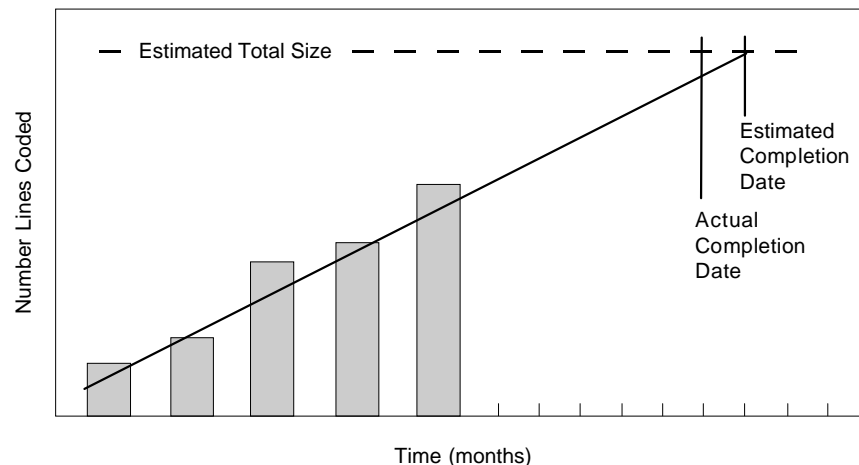
### 6.3 Tracking Progress

In managing a project, the following questions are fundamental:

- How much has been done?
- How much is left to do?
- When will it be completed?

These questions can be answered for any activity with outputs or products that can be expressed in quantifiable units. For requirements analysis, this could be the number of requirements to be allocated to each software component; 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. To do this, estimate the total number of units to be completed and then, at regular intervals (weekly or monthly), track the actual number completed. Then generate a production curve for that activity. Extrapolation of the curve will give an objective basis for projecting the completion date for that activity. A simple linear extrapolation is often remarkably accurate.

Figure 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 120,000. The actual number of lines of 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.



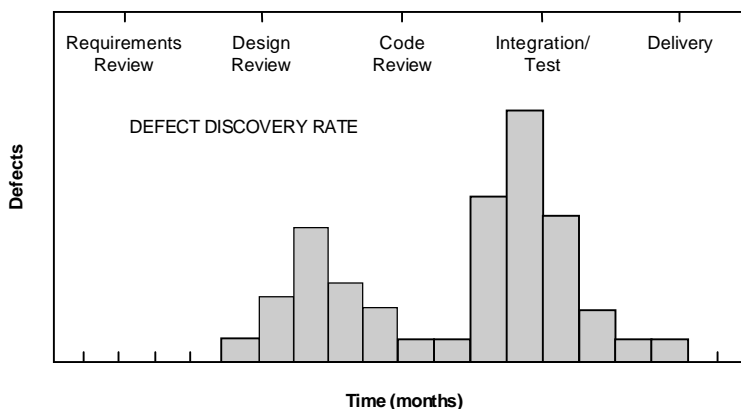
**Figure 11 Extrapolating Measurements to Forecast a Completion Date**

This type of analysis is valid only if objective criteria exists for counting units as complete. In this example, the criteria were that each line of code had completed unit test and had been entered under configuration control. At that point, it was processed by an automated code counter.

The same analysis can be performed for individual components. If one or more components are lagging, more people can be assigned to work on it, or the approach can be changed to increase the code production rate for the affected components.

Another measure that provides valuable information for projecting completion dates is the rate of defect discovery, especially during integration and test. Ideally, a steady decline in defect discoveries

will be seen as the scheduled delivery date approaches. Figure 12 shows an example of a project that delivered on schedule. Contrast this with the pattern shown in Figure 10, where the number of defects detected peaked near the end of integration and test.



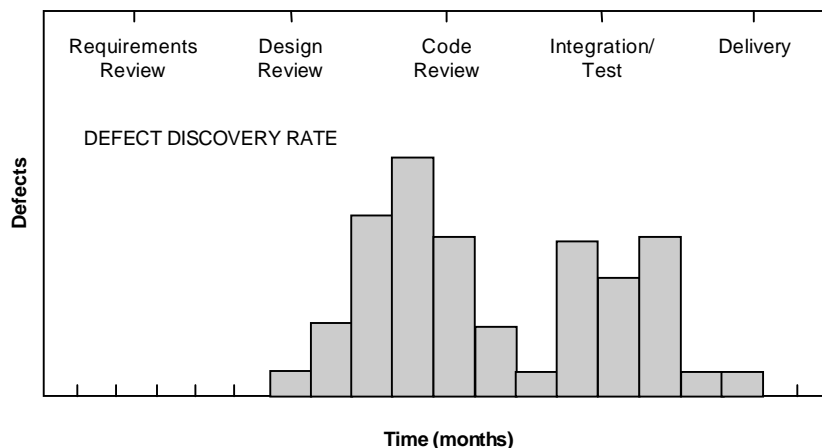
**Figure 12** Effects of Normal Schedules

## 6.4 Improving the Process

The recommended measures increase insight into the development and maintenance process. This, in turn, helps 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, one from development and the other from maintenance.

### 6.4.1 Evaluating the Impact of Design and Code Inspections

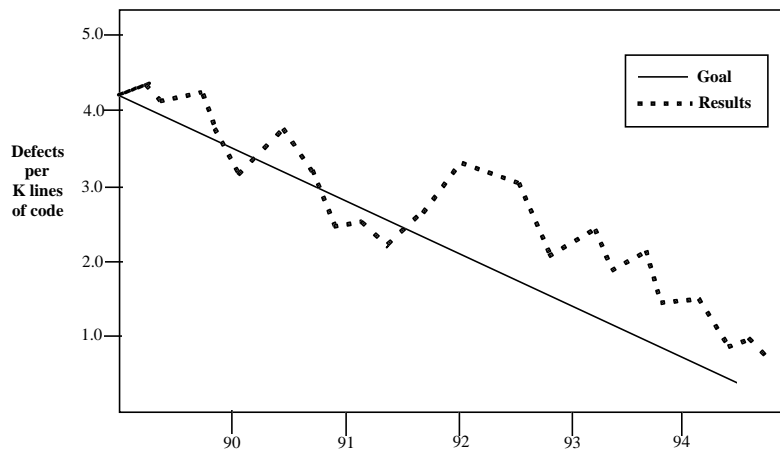
Problem reports can be used to evaluate the impact of implementing design and code inspections. Figure 13 shows the profile of problems reported over time for a project that had implemented inspections in an attempt to find as many defects as early in the process as possible. As the figure shows, the number of defects detected peaks during design and coding activities and drops off quickly (as desired) during integration and testing. Compare the pattern shown in Figure 13 with that in Figures 10 and 12.



**Figure 13** Effects of Early Defect Detection

## 6.4.2 Improving Maintenance

In this example, measures were used to bring much needed visibility to software maintenance. The organization implementing these measures was maintaining fielded versions of their software worldwide. When a problem came in from the field, the organization tracked which system components were at fault and the amount of time spent isolating and fixing the problem. (This typically involved a workaround 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.



**Figure 14 Declining Defect Density**

Figure 14 shows the defect reduction trend over a five-year period of a project in which customer software problem reports are logged and defects are identified and removed. Post-release (software maintenance) defect tracking is one of the keys to increased customer satisfaction.

## 6.5 Calibrating Cost Models

Basic software measures can also be used in calibrating cost models. As noted earlier, several commercially available cost models allow estimators to use historical size/effort/duration data to calibrate underlying estimating algorithms. Others use the data to derive tailored values for parameters that characterize resources, productivity, product difficulties, or organizational capabilities in more complex ways. In either case, once these values are determined from completed projects, the resulting baselines 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.

## 7 RECOMMENDATIONS

Specific actions on individual sites obviously will depend on existing software metrics initiatives and other software improvement initiatives. However, the following are recommended:

- Do simple "project archeology" as a valuable first step. Take the measurements defined in this paper and develop them as far as you can for some recently completed projects. Consider the questions:
  - How is my organization's software development process performing?
  - What collection mechanisms and data are available?
  - What key problem areas need work?
  - How could these core metrics and other metrics contribute?
- Consider the metrics program in the context of ongoing software improvement efforts in your organization.
- With this document as a base, establish specific goals and plans for a local metrics initiative.
- Using this document, existing metrics data, and your goals and plans, seek support and sponsorship for your local metrics program.

As far as possible, establish automated mechanisms for metrics data collection. Regularly collect these primary metrics and additional metrics specific to the local goals in your organization.

- Plan analysis resources for the metrics data within the overall software improvement work and take action to improve. Evolve the metrics program according to your needs.
- Collaborate with the SEMATECH SPI Project to consolidate semiconductor industry software metrics.
- Remember this formula: Measurement --> Analysis --> Action

## 8 REFERENCES

1. Basili, V. and Weiss, D.M., "A Methodology for Collecting Valid Software Engineering Data." *IEEE Transactions on Software Engineering*, vol. SE-10, No. 6, November 1984, pp. 728-738.
2. Basili, V. and Rombach, H., "The TAME Project: Towards Improvement Oriented Software Environment," *IEEE Transactions on Software Engineering*, vol. 14, No. 6, June 1988, pp. 758-773.
3. Boehm, B. W., *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, NJ, 1981.
4. Brooks, F. P., *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, Reading, MA, 1975.
5. Carleton, A.D., Park, R.E., Goethert, W.B., Florac, W.A., Bailey, E.K., Pfleeger, S.L., *Software Measurement for DoD Systems: Recommendations for Initial Core Measures* (CMU/SEI-92-TR-19). Software Engineering Institute, Pittsburgh, PA, 1992.
6. Conte, S., Dunsmore, H., and Shen, V., *Software Engineering Metrics and Models*. Benjamin Cummings, Menlo Park, CA, 1986.
7. Deming, W. E., *Out of the Crisis*. MIT Press, Cambridge, MA, 1986.
8. Florac, W.A., *Software Quality Measurement: A Framework for Counting Problems and Defects* (CMU/SEI-92-TR-22). Software Engineering Institute, Pittsburgh, PA, 1992.
9. Goethert et al., *Software Effort & Schedule Measurement: A Framework for Counting Staff-Hours and Reporting Schedule Information* (CMU/SEI-92-TR-21). Software Engineering Institute, Pittsburgh, PA, July 1992.
10. Grady, R. B. and Caswell, D.L., *Software Metrics: Establishing a Company-Wide Program*. Prentice-Hall, Englewood Cliffs, NJ, 1987.
11. Grady, R. B., *Practical Software Metrics for Project Management and Process Improvement*. Prentice-Hall, Englewood Cliffs, NJ, 1992.
12. *Standard for Software Productivity Metrics* (draft) (P1045/D5.0). Institute of Electrical and Electronic Engineers, Inc., Washington, DC, 1992.
13. *Standard for Software Quality Metrics Methodology* (P-1061/D21). Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1990.
14. *International Standard for Information Technology - Software Product Evaluation - Quality Characteristics and Guidelines for Their Use*. ISO/IEC 9126.1991(E), December 1991.
15. Kitson, D. and Masters, S., *An Analysis of SEI Software Process Assessment Results* (CMU/SEI-92-TR-24). Software Engineering Institute, Pittsburgh, PA, July 1992.
16. Krasner, Herb and Reed, Russ, *Software Process Improvement Issues Survey Findings Report*, Technology Transfer #93011489A-TR. SEMATECH, February 14, 1993.
17. Musa, J. D., Ianinno, A. and Okumoto, K., *Software Reliability: Measurement, Prediction, Application*. McGraw Hill, New York, NY, 1987.
18. *Software Size Measurement: A Framework for Counting Source Statements* (CMU/SEI-92-TR-20). Software Engineering Institute, Pittsburgh, PA, July 1992.
19. Paulk, M.C., Curtis, B., Chrissis, M.B., and Weber, C.V., *Capability Maturity Model for Software, Version 1.1* (CMU/SEI-93-TR-024). Software Engineering Institute, Pittsburgh, PA, 1993.

20. *Guideline for Definition and Measurement of Equipment Reliability, Availability, and Maintainability (RAM)*. Semiconductor Equipment and Materials International (SEMI), Mountain View, CA, 1992.
21. *SPI Guidelines for Improving Software: Release 3.0*, Technology Transfer #94092541A-ENG. SEMATECH, October 31, 1994.





**SEMATECH Technology Transfer  
2706 Montopolis Drive  
Austin, TX 78741**

**<http://www.sematech.org>**