SOFTWARE MEASUREMENT GUIDEBOOK

Revision 1

JUNE 1995

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

SOFTWARE MEASUREMENT GUIDEBOOK

Revision 1

JUNE 1995

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

Foreword

The **Software Engineering Laboratory** (SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and created to investigate the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1976 and has three primary organizational members:

NASA/GSFC, Software Engineering Branch

University of Maryland, Department of Computer Science

Computer Sciences Corporation, Software Engineering Operation

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effects of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

This *Software Measurement Guidebook* has also been released as NASA-GB-001-94, a product of the Software Engineering Program established by the Office of Safety and Mission Assurance (Code Q) at NASA Headquarters.

The following are primary contributors to this document:

Mitchell J. Bassman, Computer Sciences Corporation

Frank McGarry, Goddard Space Flight Center

Rose Pajerski, Goddard Space Flight Center

Single copies of this document can be obtained by writing to

Software Engineering Branch
Code 552
Goddard Space Flight Center
Greenbelt, Maryland 20771

Abstract

This *Software Measurement Guidebook* presents information on the purpose and importance of measurement. It discusses the specific procedures and activities of a measurement program and the roles of the people involved. The guidebook also clarifies the role that measurement can and must play in the goal of continual, sustained improvement for all software production and maintenance efforts.

Contents

Figures

Tables

Chapter 1. Introduction

1.1 Background

This *Software Measurement Guidebook* is based on the extensive experience of several organizations that have each developed and applied significant measurement[1] programs over a period of at least 10 years. One of these organizations, the Software Engineering Laboratory (SEL) at the National Aeronautics and Space Administration (NASA) Goddard Space Flight Center (GSFC), has been studying and applying various techniques for measuring software since 1976. During that period, the SEL has collected measurement data from more than 100 flight dynamics projects ranging in size from 10,000 to over 1,000,000 source lines of code (SLOC). These measurement activities have generated over 200,000 data collection forms, are reflected in an online database, and have resulted in more than 200 reports and papers. More significantly, they have been used to generate software engineering models and relationships that have been the basis for the software engineering policies, standards, and procedures used in the development of flight dynamics software.

Many other organizations in both Government and industry have documented their significant measurement experiences. (See, for example, References 1 through 7.) The lessons derived from those experiences reflect not only successes but also failures. By applying those lessons, an organization can minimize, or at least reduce, the time, effort, and frustration of introducing a software measurement program.

The *Software Measurement Guidebook* is aimed at helping organizations to begin or improve a measurement program. It *does not* provide guidance for the extensive application of specific measures (such as how to estimate software cost or analyze software complexity) other than by providing examples to clarify points. It *does* contain advice for establishing and using an effective software measurement program and for understanding some of the key lessons that other organizations have learned. Some of that advice will appear counterintuitive, but it is all based on actual experience.

Although all of the information presented in this guidebook is derived from specific experiences of mature measurement programs, the reader must keep in mind that the characteristics of every organization are unique. Some degree of measurement is critical for all software development and maintenance organizations, and most of the key rules captured in this report will be generally applicable. Nevertheless, each organization must strive to understand its own environment so that the measurement program can be tailored to suit its characteristics and needs.

Historically, many software organizations have established development and maintenance processes and standards in an ad hoc manner, on the basis of guidance from outside the organization, or from senior personnel called upon to establish company standards. Often, this approach has led to incompatibilities, unconvinced development groups, and, occasionally, complete confusion. Too often, organizations attempt to generate policies or standards and to

[1] Some organizations use the terms *metrics* and *measurement* interchangeably.

adopt particular technologies without first understanding the existing processes and environment. This lack of understanding can make a bad situation worse. Before establishing policies and defining standards, an organization must clearly understand the environment and the existing processes. A commitment to understand and improve local software processes requires the establishment of a software measurement program, which is the precursor to continual process improvement.

The following rule is the single most important one regarding software measurement:

> ***Understand that software measurement is a means to an end, not an end in itself.***

A measurement program without a clear purpose will result in frustration, waste, annoyance, and confusion. To be successful, a measurement program must be viewed as one tool in the quest for the improved engineering of software.

1.2 Purpose

The purpose of this *Software Measurement Guidebook* is threefold. First, it presents information on the purpose and importance of measurement—information that has grown out of successful measurement applications.

Second, the guidebook presents the specific procedures and activities of a measurement program and the roles of the people involved. This guidebook discusses the basic set of measures that constitutes the core of most successful measurement programs. It also provides some guidance for tailoring measurement activities as a program matures and an organization captures its own experiences.

Finally, the guidebook clarifies the role that measurement can and must play in the goal of continual, sustained improvement for all software production and maintenance efforts throughout NASA. As NASA matures in its understanding and application of software, it is attempting to apply the most appropriate software technologies and methodologies available. Like any other software organization, NASA must build a firm foundation for software standards, policies, and procedures. A carefully established measurement program can provide the rationale for management decision making, leading to achievement of the goal of sustained improvement.

1.3 Organization

This "Introduction" is followed by six additional chapters and three appendices.

Chapter 2, "The Role of Measurement in Software Engineering," lays the groundwork for establishing a measurement program. The chapter explains why any software group should have a well-defined measurement program and provides examples of supporting data that can be valuable in justifying the costs involved in implementing such a program.

Chapter 3, "Establishing a Measurement Program," describes the essential steps for starting a measurement program. The chapter includes organization, key measurement data, classes and

sources of data, general cost information, and, most important, goal setting and application of the measurement program.

Chapter 4, "Core Measures," introduces the recommended core set of measures that can benefit any software organization.

Chapter 5, "Operation of a Measurement Program," discusses major organizational issues, data collection and storage, quality assurance (QA) of the data, feedback of data, and cost of operations.

Chapter 6, "Analysis, Application, and Feedback," presents information on the analysis of measurement data and the application and feedback of information derived from a measurement program.

Chapter 7, "Experience-Based Guidelines," offers some precautions for software organizations that plan to include software measurement among their development and maintenance processes.

Appendices A, B, and C provide sample data collection forms, a sample process study plan, and a list of rules, respectively.

Chapter 2. The Role of Measurement in Software Engineering

Chapter Highlights

THREE KEY REASONS FOR SOFTWARE MEASUREMENT

1. **Understanding Software**
   - Baseline models and relationships
   - Key process characteristics
   - Four measurement examples

2. **Managing Software Projects**
   - Planning and estimating
   - Tracking actuals versus estimates
   - Validating models

3. **Guiding Process Improvement**
   - Understanding
   - Assessing
   - Packaging

This chapter clarifies the role that a software measurement program can play in support of software development and maintenance activities and provides sound motivation for any organization to initiate or expand its analysis of data and application of results. The chapter explains the three key reasons for an organization to measure its software engineering processes and product, providing actual examples from software organizations with mature measurement programs.

A software organization may want to establish a software measurement program for many reasons. Those range from having good management information for guiding software development to carrying out research toward the development of some innovative advanced technique. However, more than 17 years of experience with software measurement activities within NASA have shown that the three key reasons for software measurement are to

1. Understand and model software engineering processes and products
2. Aid in the management of software projects
3. Guide improvements in software engineering processes

Any one of these reasons should be enough to motivate an organization to implement a measurement program. The underlying purpose of any such program, however, must be to achieve specific results from the *use* and *application* of the measures; collecting data is *not* the objective. Most failed measurement programs suffer from inadequate or unclear use of data, not from an inadequate or unclear data collection process. The rule in Chapter 1 implies that the measurement program must be defined in a way that satisfies specific objectives. Without such objectives, no benefit will be derived from the measurement effort.

2.1 Measurement To Increase Understanding

The most important reason for establishing a measurement program is to evolve toward an understanding of software and the software engineering processes in order to derive models of those processes and examine relationships among the process parameters. Knowing what an organization does and how it operates is a fundamental requirement for any attempt to plan, manage, or improve. Measurement provides the only mechanism available for quantifying a set of characteristics about a specific environment or for software in general.

Increased understanding leads to better management of software projects and improvements in the software engineering process. A software organization's objective may be to understand the status of the software engineering process or the implications of introducing a change. General questions to be addressed might include the following:

- How much are we spending on software development?
- Where do we allocate and use resources throughout the life cycle?
- How much effort do we expend specifically on testing software?
- What types of errors and changes are typical on our projects?

Figure 2-1 illustrates some more specific questions that may be of immediate concern to a software manager.

Figure 2-1. Motivation for Understanding the Software Engineering Process

To be able to address such issues, an organization must have established a baseline understanding of its current software product and process characteristics, including attributes such as software size, cost, and defects corrected. Once an organization has analyzed that basic information, it can build a software model and examine relationships. For example, the expected level of effort can be computed as a function of estimated software size. Perhaps even more important, understanding processes makes it possible to predict cause and effect relationships, such as the effect on productivity of introducing a particular change into a process.

This guidebook emphasizes the importance of developing models of a local organization's specific software engineering processes. However, a general understanding of the engineering of software can also prove beneficial. It provides a foundation for appreciating which types of models and relationships apply in a specific software development or maintenance environment.

For example, a manager should know that, in any environment, the amount of effort required to complete a project is related to the size of the software product and that changing the size of the staff will have an effect on the ability to meet scheduled milestones. The precise effect within the local environment depends on a complex combination of factors involving staff productivity, experience, and maturity. The parameter values that tailor the model to the unique characteristics of the local environment must be derived, over time, under the careful administration of the measurement program.

Potential objections to establishing a measurement program and developing an understanding of the current processes are numerous:

- My organization is changing too fast.
- Each project is unique.

- Technology is changing too fast.
- Project results merely reflect the characteristics of the people on the projects.
- I don't care about future projects; I care only about current results.

Each of these objections may have some merit; nevertheless, it is essential to establish the baseline before introducing change. Managers who have never collected data to confirm or challenge basic assumptions about their environments may have inaccurate perceptions about the software processes in use within their organizations.

Experience derived from many NASA programs shows that an organization establishing a baseline understanding of its software engineering processes and products should concentrate on collecting measurement data to reflect certain key software characteristics. Table 2-1 suggests sample characteristics and refers to four examples that illustrate the points using actual NASA experience.

Table 2-1. Sample Software Characteristics

Understanding	Key Characteristics	NASA Experience
What are the cost (resource) characteristics of software in my organization?	• Distribution of effort among development activities—amount spent on design, code, test, or other activities • Typical cost per line of code • Cost of maintenance • Hours spent on documentation • Computer resources required • Amount of rework expected	Example 1
What are the error (reliability) characteristics of software in my organization?	• Number and classes of errors found during development or maintenance • How and when software defects are found • Number and classes of errors found in specifications • Pass/fail rates for integration and system testing	Example 2
How does my organization's rate of source code production (or change) compare to previous experience?	• Typical rate of growth of source code during development • Typical rate of change of source code during development or maintenance	Example 3
How does the amount of software to be developed relate to the duration of the project and the effort required? What is the relationship between estimated software size and other key parameters?	• Total number of lines of code produced • Schedule as a function of software size • Cost as a function of size • Total number of pages of documentation produced • Average staff size	Example 4

Example 1:
Effort Distribution Characteristics

Knowing the distribution of effort over a set of software development activities can contribute significantly to an understanding of software engineering processes. One NASA organization analyzed data from over 25 projects, representing over 200 staff-years of effort on actual mission software, to build the model shown in Figure 2-2. The model of effort distribution over a set of software development activities, which may occur across various phases of the software life cycle, is invaluable for management planning on new projects. The organization uses data from ongoing projects to update the model, which continues to evolve, providing more accurate information for future project managers in that environment.

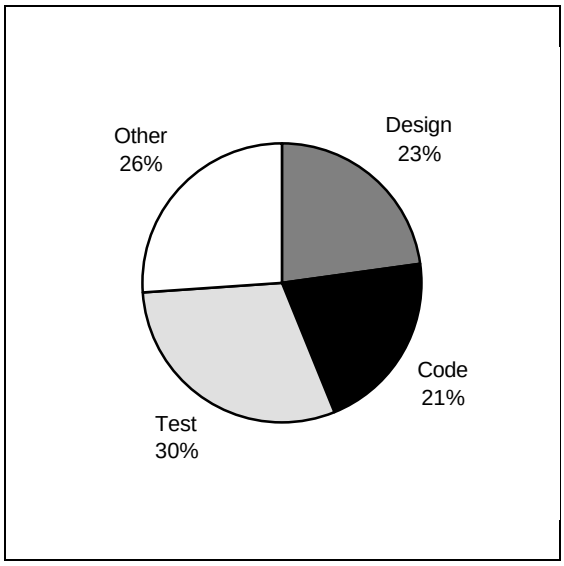


Figure 2-2. Effort Distribution by Activity

Many software organizations mistakenly assume that a generic model of distribution across life-cycle activities will apply for any organization and in any application domain. It is possible to derive a model, or a hierarchy of models, with more general applicability. For example, useful models can be derived by analyzing data from all software projects throughout NASA or for all flight simulator software projects throughout NASA. However, local organizations can apply such models with varying degrees of confidence and accuracy. Experience has shown that a model derived from, and updated with, data collected within the specific software environment is a more accurate tool—a more suitable means to a desired end.

Before local effort distribution was understood, managers had to rely on general commercial models.[2] There was also no understanding of how much time software developers spent on activities other than designing, coding, and testing software. In the model shown, for example, the "other" category includes activities such as training, meetings, and travel.

Experience has shown that such models are relatively consistent across projects within a specific environment. This model may not be directly applicable to other software development environments, however, because of variables such as personnel, application domain, tools, methods, and languages. Each software organization should produce its own effort distribution profile.

[2] Commercial models of effort distribution have historically recommended allocating 40 percent of project resources to analysis and design, 20 percent to coding, and 40 percent to testing.

An organization must also decide which activities and portions of the software or system life cycle will be included in the model or models. Even managers within the local organization can use the model shown in Figure 2-2 only for development projects, because no software maintenance data are included in the model. Any maintenance organization, however, can develop a similar model. Further, the sample domain is limited to software engineering concerns. An organization that develops or maintains *complete systems* must establish and maintain models that include activities across the entire system life cycle.

Example 2:
Error Distribution Characteristics

Another important part of understanding the software engineering process is being aware of the common classes of errors. Software project personnel must understand not only where errors originate and where they are corrected, but also the relative rates of error occurrence in different classes. A measurement program provides the means to determine error profiles. Software project personnel can use profiles of error characteristics to improve development processes on future projects or on later stages of an ongoing project.

Figure 2-3 represents a simple model of error characteristics for one NASA environment. A large sample of NASA projects collected data representing more than 10,000 errors over a 5-year period. The definitions of the error classes are meaningful to the organization that collected and analyzed the data but may not be suitable in other environments. Each organization must characterize the classes of errors that are important in its own environment.

The distribution percentages shown in the model are specific to the organization that provided the data. Moreover, in this environment, the general profile of errors does not change significantly across different projects. Although the error rate has steadily declined over a period of years, the profile shown has remained relatively stable.

An environment-specific model of error distribution can provide decision support for the planning and management of new projects. A manager who notices that one class of error is becoming more common can redirect effort to concentrate on that class during inspections and reviews. An error class distribution profile serves as a measurement tool to help both management and technical personnel isolate errors earlier in the software life cycle, reduce life-cycle costs, and increase software reliability.

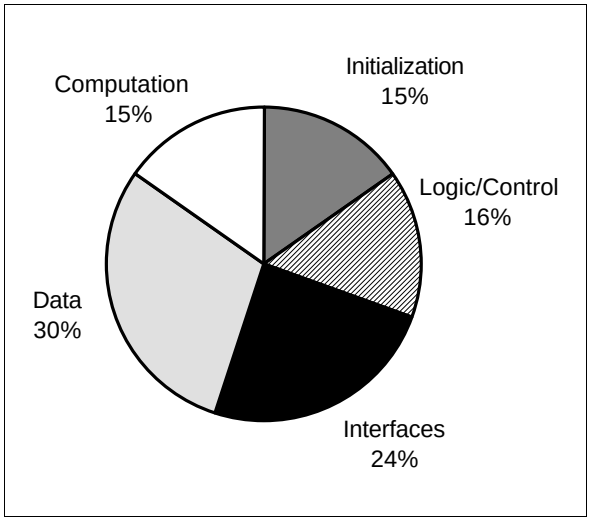


Figure 2-3. Error Class Distribution

Example 3:
Software Growth and Change Characteristics

Insight into the rates of growth and change of source code also helps to build a better understanding of software engineering processes. Code growth reflects the rate at which source code is added to a controlled library; code change reflects modifications to the controlled, or baselined, library. An understanding of the model for such rates can provide a basis for determining if a new project is progressing as expected or if it is producing or changing source code at a rate that differs from the organization's historical profile.

Figure 2-4 depicts the typical rate of growth of source code in a NASA environment. The data were derived from over 20 software projects that followed a waterfall life cycle. This information is used only to model typical projects in one particular environment, not to determine the quality of a given process.

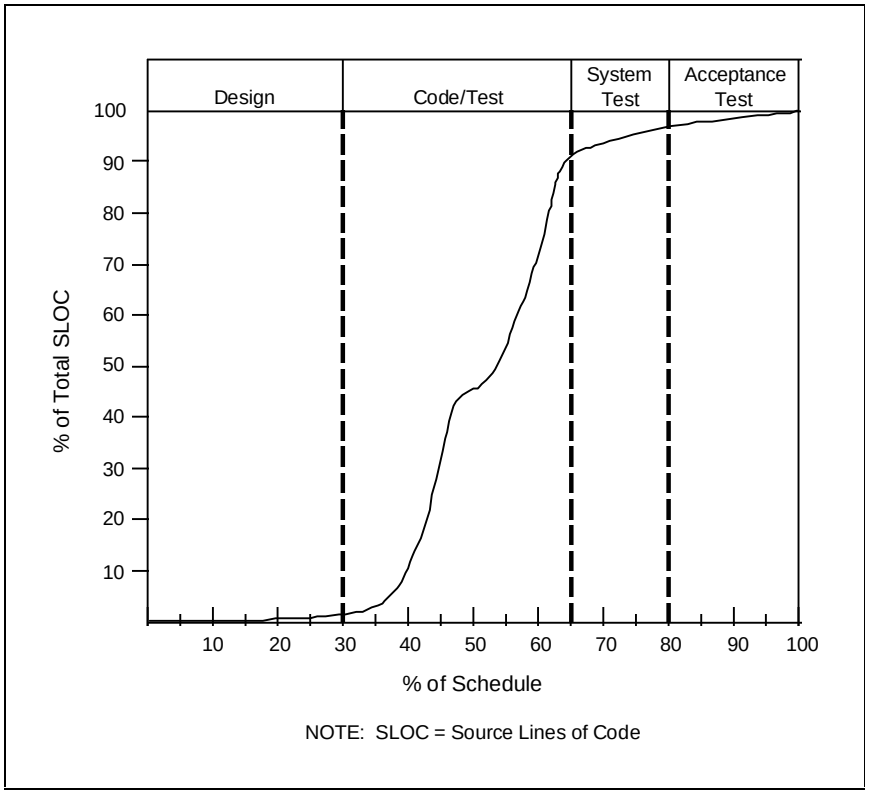


Figure 2-4. Growth Rate of Source Code

Figure 2-5 shows the accumulated changes to source code during the development phases in the same environment. Both of the profiles shown here were derived from measurement data that were inexpensive to collect and analyze, and the resulting models are quite stable.

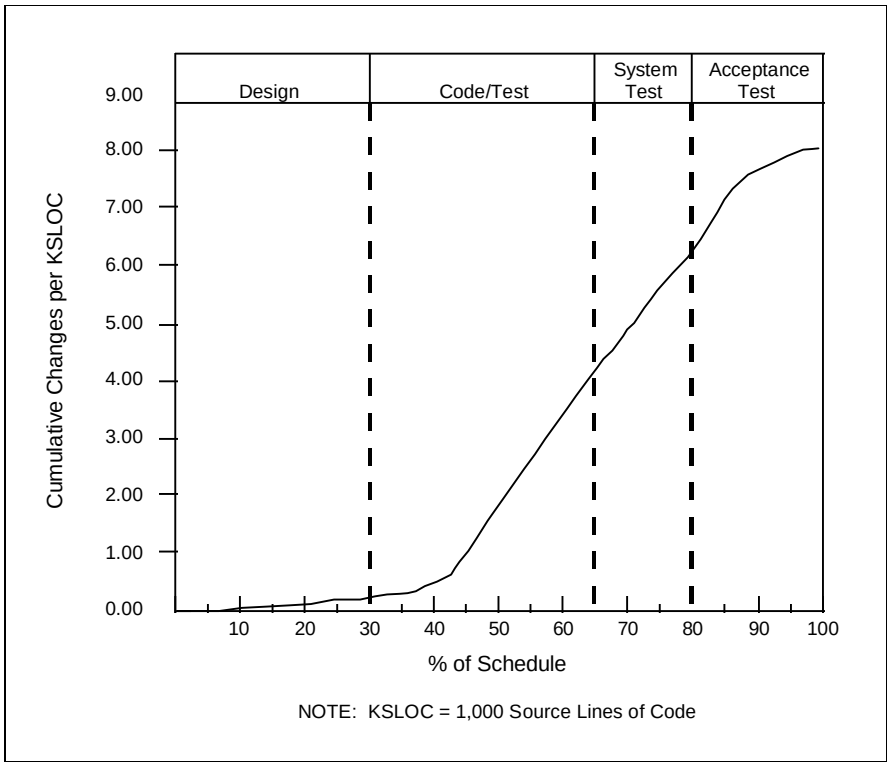


Figure 2-5. Change Rate of Source Code

Example 4:
Software Process Relationships

The functional relationships between product and process parameters provide additional understanding of an organization's software engineering processes. This understanding can be applied to the planning and management of subsequent projects in the same environment.

Figure 2-6 presents examples of a few key relationships that were found useful in several NASA environments. A SEL report (Reference 8) discusses those and other such relationships and how they can be applied. The relationship constants are periodically revised to reflect evolving organizational models. After the historical database has been created, the additional effort required to develop such relationships has proved to be small and worthwhile, leading to increased understanding of the software engineering process.

2.2 Measurement for Managing Software

The second key reason for establishing an effective measurement program is to provide improved management information. Having an understanding of the software environment based on models of the process and on relationships among the process and product parameters allows for better prediction of process results and more awareness of deviations from expected results. Thus, understanding the software engineering process leads to better management decision making. The understanding comes from analyzing local data; without analysis, any data collection activity is a

Effort (in staff-months)	=	$1.48 * (KSLOC)^{0.98}$
Duration (in months)	=	$4.6 * (KSLOC)^{0.26}$
Pages of Documentation	=	$34.7 * (KSLOC)^{0.93}$
Annual Maintenance Cost	=	0.12 * (Development Cost)
Average Staff Size	=	$0.24 * (Effort)^{0.73}$

Figure 2-6. Sample Process Relationships

waste of effort. The next step is to use the understanding that comes from the engineering models to plan and manage software project activities.

Focus on applying results rather than collecting data.

A measurement program that focuses on the collection process, or that does not have a clear plan for applying the acquired understanding, will fail.

Specifically, the knowledge gained about the software engineering process will be used to

- *Estimate* project elements such as cost, schedules, and staffing profiles
- *Track* project results against planning estimates
- *Validate* the organizational models as the basis for improving future estimates

Engineering models and relationships provide a foundation for the software engineering estimates that form an important part of the project management plan. Without accurate models based on similar classes of software development and maintenance activities, project management success is uncertain.

The next three sections address the use of models and relationships in more detail.

2.2.1 Planning and Estimating

One of the most critical responsibilities of a software project manager is developing a software project management plan, and one of the most important elements of that plan is a set of project estimates for cost, schedule, staffing requirements, resource requirements, and risks. Measurement results from similar completed projects are used to derive software engineering models (providing an understanding of the environment), which, in turn, are used to develop the estimates. The quality of the information in the historical database directly affects the quality of the software engineering models and, subsequently, the quality of the planning estimates for new projects.

A manager who can produce a product size estimate based on software functionality requirements can then derive such estimates as cost and schedule using organizational models and relationships. The standard size estimates within the SEL are currently based on developed lines of code (DLOC). (For a detailed discussion of DLOC—software size with a weighting factor applied to reused code—see Reference 9 and Sections 4.5.2 and 6.1.2 of this document.) Given a product size estimate and the distribution percentages shown in Table 2-2 (Reference 10), a manager can derive project cost (measured as staff effort) and schedule estimates using the relationships

$$\text{Effort (in hours)} = \text{DLOC} / \text{Productivity}$$

where

$$\text{Productivity} = 3.2 \text{ DLOC per Hour}$$

for FORTRAN, and

$$\text{Duration (in months)} = 4.9 \, (\text{Effort [in staff-months]})^{0.3}$$

for attitude ground support systems (AGSSs).

For example, assuming an estimated product size of 99,000 DLOC for an AGSS to be developed in FORTRAN, a total effort of approximately 200 staff-months and a total duration of approximately 24 calendar months can be estimated.[3] The table also provides derived project estimates for the cost and duration of each major life-cycle phase. In this model, the design phase comprises requirements analysis, preliminary design, and detailed design, and the test phase encompasses both system and acceptance test. Initial planning estimates may have to be adjusted for changes in requirements or schedule. It is also important to note that the specific parameters in the relationships shown here are highly dependent on environmental factors, such as the local definition of a line of code. Although anyone can use this model as a starting point, each organization must analyze its data to derive its own distribution model.

Table 2-2. Distribution of Time Schedule and Effort Over Phases

Life-Cycle Phases	Distribution Model (Reference 10)		Sample Derived Estimates (for 99,000 DLOC)	
	Time Schedule (%)	Effort (%)	Completion Milestones (Months by Phase)	Staff-Months (Allocated by Phase)
Design	35	30	8.4	60
Code	30	40	7.2	80
Test	35	30	8.4	60

[3] The conversion between staff-months and staff-hours is organization-dependent. In this example, 1 staff-month = 157 staff-hours.

2.2.2 Tracking

An important responsibility of software project management is tracking the actual size, effort, budget, and schedule against the estimates in the approved plan. Successful, effective management requires visibility into the progress and general status of the ongoing project, so that timely and informed adjustments can be made to schedules, budgets, and processes. Periodic sampling of project measurement data provides that visibility.

The extent and effectiveness of the project tracking process depends on the availability and quality of a set of historical models and relationships. If the only available model is related to cost data, then management tracking will be limited to cost information. However, a more extensive set of derived models for staff size, software growth rate, software change rate, error rate, and other parameters will facilitate a broader tracking capability.

Figure 2-7 illustrates the process of tracking the actual software growth rate[4] against the planning estimates. In this illustration, the planned growth estimates are based on the model introduced in Figure 2-4. A deviation of the actual values from the expected curve indicates simply that something is different from the historical model. Such a deviation does not necessarily signal a problem; rather, it can provide the program manager with an opportunity to explain the difference. In particular, the deviation may have resulted from a planned improvement. For example, a project that is reusing a larger amount of code than the typical past project may show a sharp jump in growth rate when reused code is moved into the controlled library.

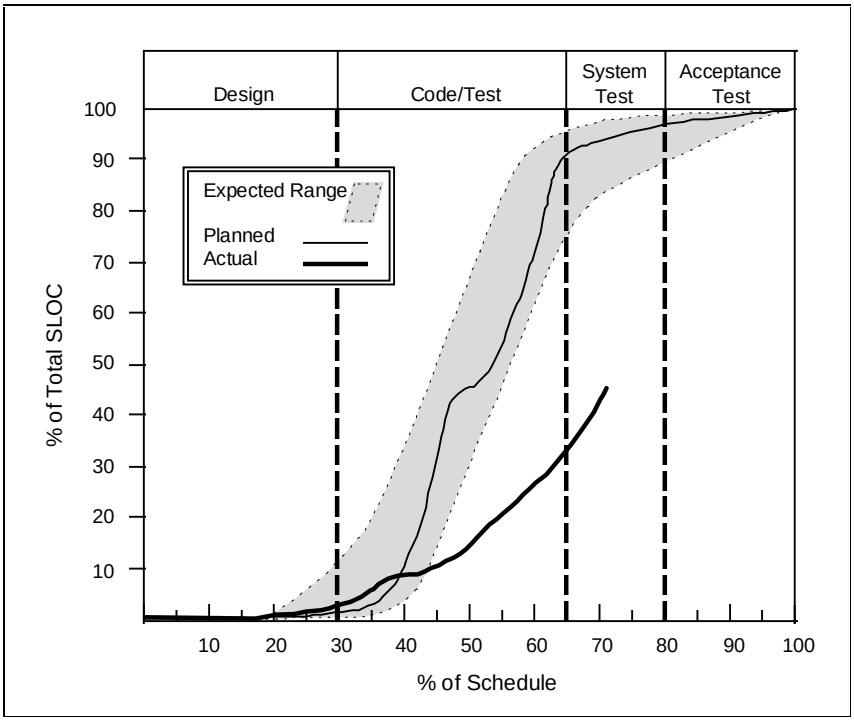


Figure 2-7. Tracking Growth Rate

[4] Software growth rate reflects the rate at which programmers complete the unit testing of source code. In Figure 2-7, the *actual* percentage of the total is computed with respect to the estimated size at completion.

2.2.3 Validating

Once a manager has the ability to track actual project measures against planning estimates, he or she can begin to use any observed differences to evaluate the status of the project and to support decisions to take corrective actions. Figure 2-7 also shows an allowable range of deviation around the planned or expected values on the growth curve. Observing the trend of the actual growth rate relative to the planned values can provide a management indicator of a healthy project (as determined by a growth pattern within the expected range) or a potential problem that requires further evaluation to determine the cause (as is the case in Figure 2-7). With the insight gained by observing the trend, a manager can adjust staffing or schedule to get the project back on track.

Although it is obvious that an actual value below the allowable range may indicate a cause for concern, it is perhaps less obvious that an actual value that falls above the allowable range should also generate a management investigation. In this example, a software growth rate above the allowable range may indicate that some other project activities are not being performed or, perhaps, that the wrong model was used for planning and estimation. Consistent and regular deviations may also indicate a need to adjust the organization's models.

Examples within this section have illustrated that a baseline understanding of the software engineering process derived from historical results provides the essential model, which leads to the planning estimate, which makes the tracking possible. The process of tracking actual versus planned growth values provides the insight for model validation, which facilitates adjustments by project management. The fundamental element of measurement support for project management is understanding the software engineering process.

2.3 Measurement for Guiding Improvement

The primary focus of any software engineering organization is to produce a high-quality product within schedule and budget. However, a constant goal, if the organization is to evolve and grow, must be continual improvement in the quality of its products and services. *Product* improvement is typically achieved by improving the processes used to develop the product. *Process* improvement, which requires introducing change, may be accomplished by modifying management or technical processes or by adopting new technologies. Adoption of a new technology may require changing an existing process. In any case, software measurement is a key part of any process improvement program; knowing the quality of the product developed using both the initial and the changed process is necessary to confirm that improvement has occurred.

There are several popular paradigms for software process improvement. For example, the Capability Maturity Model (CMM) for Software (Reference 11), produced by the Software Engineering Institute (SEI) at Carnegie Mellon University, is a widely accepted benchmark for software engineering excellence. It provides a framework for grouping key software practices into five levels of maturity. A maturity level is an evolutionary plateau on the path toward becoming a mature software organization. The five-level model, represented in Figure 2-8, provides a defined sequence of steps for gradual improvement and prioritizes the actions for improving software practices.

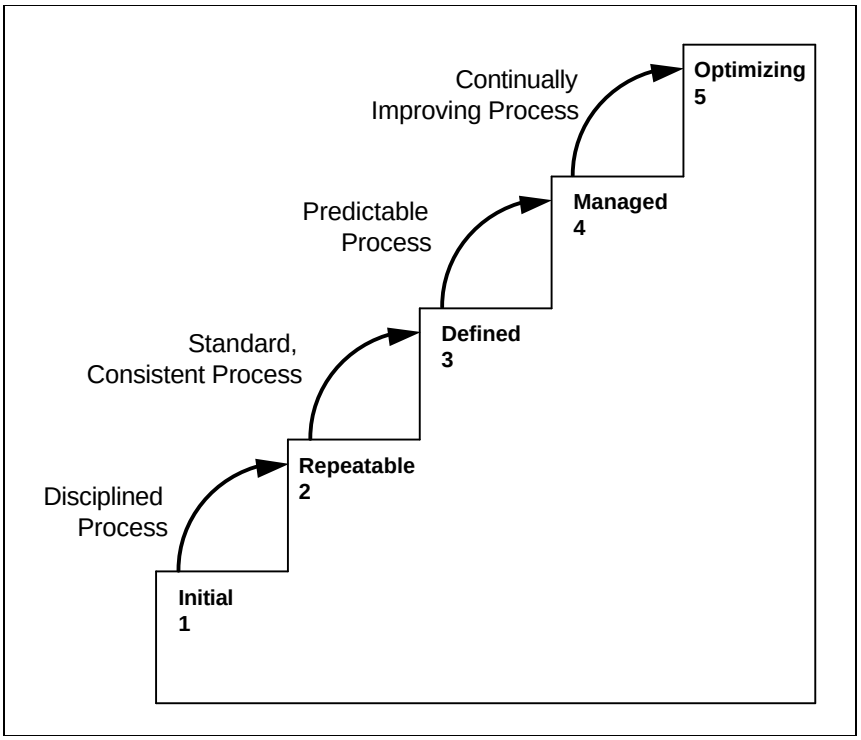


Figure 2-8. The Five Maturity Levels of the CMM

The SEI provides the following characterization of the five levels:

1. *Initial*—The software process is characterized as ad hoc and, occasionally, even chaotic. Few processes are defined, and success depends on the efforts of individuals.

2. *Repeatable*—Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

3. *Defined*—The software process for both management and engineering activities is documented, standardized, and integrated into an organization-wide software process. All projects use a documented and approved version of the organization's process for developing and maintaining software.

4. *Managed*—Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled using detailed measures.

5. *Optimizing*—Continuous process improvement is enabled by quantitative feedback from the process and from testing innovative ideas and technologies.

The CMM is an organization-independent model that emphasizes improving processes to reach a higher maturity level when compared to a common benchmark. Such a model presupposes that the application of more mature processes will result in a higher quality product. In contrast, the SEL has introduced a process improvement paradigm for NASA with specific emphasis on

producing a better product based on the individual goals of the organization. Figure 2-9 illustrates the SEL's Understand/Assess/Package paradigm.

In the SEI model, a baseline assessment of an organization's deficiencies, with respect to the key processes defined at each of the maturity levels, determines the priority with which the organization implements process improvements. In the SEL model, the specific experiences and goals of the organization drive changes. (See Reference 12 for a more detailed comparison of the two paradigms.)

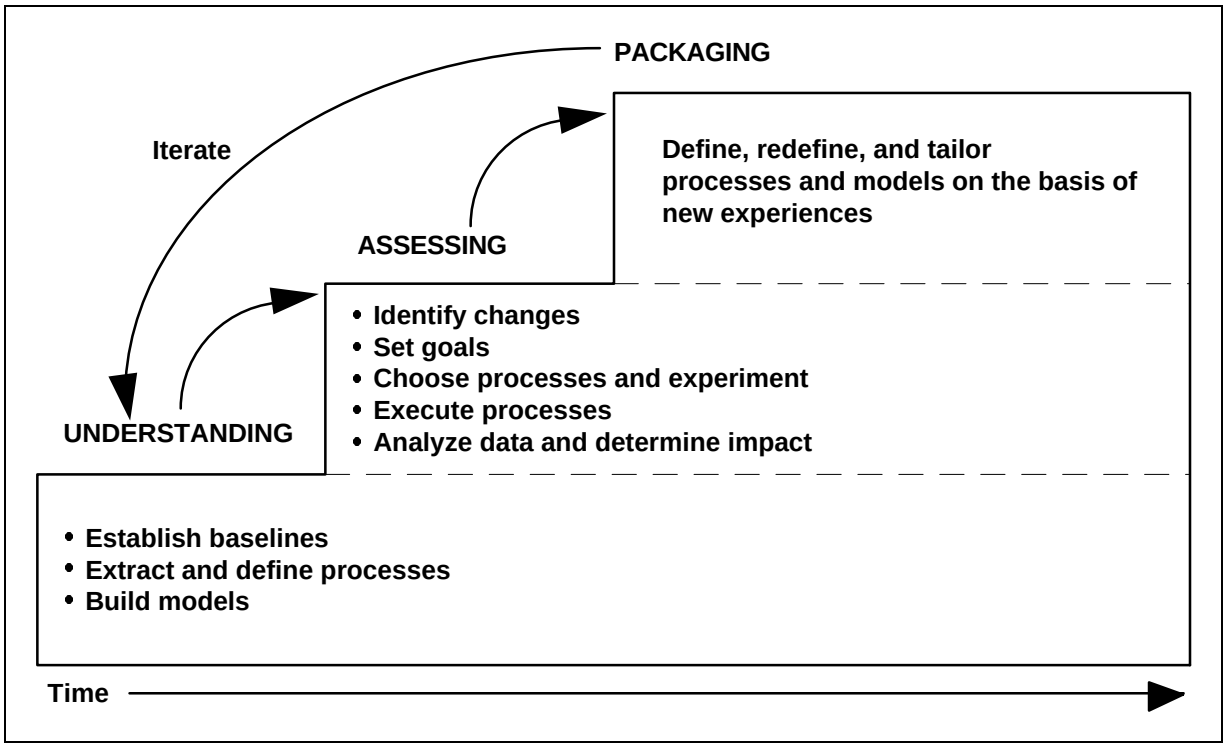


Figure 2-9. The Understand/Assess/Package Paradigm

2.3.1 Understanding

Section 2.1 introduced understanding as the primary reason for establishing a measurement program; that same understanding provides the foundation for NASA's process improvement paradigm. To provide the measurement basis for its software engineering process improvement program, an organization must begin with a baseline understanding of the current processes and products by analyzing project data to derive (1) models of the software engineering processes and (2) relationships among the process and product parameters in the organization's environment.

As the organization's personnel use the models and relationships to plan and manage additional projects, they should observe trends, identify improvement opportunities, and evaluate those opportunities for potential payback to the organization. As improvements are implemented, new project measurement results are used to update the organization's models and relationships. These updated models and relationships improve estimates for future projects.

Improvement plans must be made in the context of the organization's goals. Improvement can be defined only within the domain of the organization—there are no universal measures of improvement. An organization may base its process improvement goals on productivity, cost, reliability, error rate, cycle time, portability, reusability, customer satisfaction, or other relevant characteristics; however, each organization must determine what is most important in its local environment. Using measurement as the basis for improvement permits an organization to set specific quantitative goals. For example, rather than simply striving to reduce the error rate, an organization can establish a goal of lowering the error rate by 50 percent. Determining the effect of introducing change requires initial measurement of the baseline.

2.3.2 Assessing

Once an organization understands the current models and relationships reflecting its software process and product, it may want to assess the impact of introducing a process change. It should be noted that a *change* is not necessarily an *improvement*. Determining that a change is an improvement requires analysis of measures based on the organization's goals. For example, assume that an organization's goal is to decrease the error rate in delivered software while maintaining (or possibly improving) the level of productivity; further assume that the organization has decided to change the process by introducing the Cleanroom method (Reference 13). Cleanroom focuses on achieving higher reliability (i.e., lower error rates) through defect prevention. Because the organization's primary goal is to reduce the error rate, there is no concern that the Cleanroom method does not address reuse, portability, maintainability, or many other process and product characteristics.

During a recent study (Reference 14), the SEL assessed the impact of introducing the Cleanroom method. Table 2-3 shows the error rate and productivity measures for the baseline and the first Cleanroom project. The results of the experiment appear to provide preliminary evidence of the expected improvement in reliability following introduction of the Cleanroom method and may also indicate an improvement in productivity. Chapter 6 provides additional details of the SEL Cleanroom study.

Table 2-3. Impact of the Cleanroom Method on Reliability and Productivity

Data Source	Error Rate (Errors per KDLOC)	Productivity (DLOC per Day)
Baseline	5.3	26
Cleanroom	4.3	40

NOTE: KDLOC = 1,000 Developed Lines of Code

2.3.3 Packaging

NASA experience has shown that feedback and packaging of measured results must occur soon after completion of an impact assessment. Packaging typically includes written policies, procedures, standards, and guidebooks. High-quality training material and training courses are also essential parts of the packages.

For example, to incorporate the Cleanroom method as an integral part of its software development activities, an organization must first prepare the necessary documentation and provide training to all affected project personnel. Packaging is discussed in more detail in Chapter 5.

Chapter 3. Establishing a Measurement Program

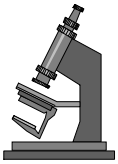
Chapter Highlights

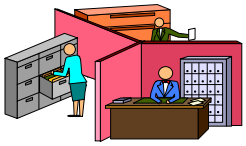
GOALS

- Understanding the organization's goals
- Understanding measurement's application
- Setting expectations
- Planning for early success

SCOPE

- Focusing locally
- Starting small



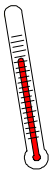


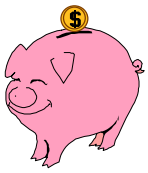
ROLES AND RESPONSIBILITIES

- Providing data
- Analyzing and packaging
- Collecting and storing

SELECTING MEASURES

- Ensuring that measures are applicable
- Minimizing the number of measures
- Avoiding over-reporting





MEASUREMENT COSTS

- Project costs—the source of data
- Technical support costs
- Analysis and packaging costs

After an organization understands the roles that measurement can play in software engineering activities, it is ready to establish a measurement program. The effective application of information derived from measurement entails building models, identifying the strengths and weaknesses of a particular process, and aiding the management decision process. A clear, well-defined approach for the application and analysis of measurement information will minimize the cost and disruption to the software organization. Building on the advice of the preceding chapter, this chapter addresses the following topics and provides recommendations for successfully establishing a new measurement program:

- Understanding the organization's goals
- Defining the scope of the measurement program
- Defining roles and responsibilities within the organization
- Selecting the appropriate measures
- Controlling the cost of measurement

3.1 Goals

First, the organization must determine what it wants to accomplish through measurement. This requirement leads to the next rule:

Understand the goals.

The goals of an organization may be to increase productivity or quality, reduce costs, improve the ability to stay on schedule, or improve a manager's ability to make informed decisions. Typically, an organization that is implementing a measurement program has all of these goals. Although it is admirable to want to improve everything immediately, establishing priorities for achieving the goals incrementally is essential. After clarifying the organizational goals, the organization must recognize the need to establish a measurement program to achieve its goals.

Understand how to apply measurement.

If the goal is to improve productivity, for example, then the organization must know its current productivity rate and understand its product and process characteristics. Both prerequisites are supplied by measurement.

The results of a measurement program will be used in different ways at each level of the organization. Senior management will be interested primarily in how the program improves the capabilities and productivity of the organization and in the effect on the bottom line. Project managers will be concerned with the impact on planning and managing current project efforts. Software developers will be interested in how the program will make work easier compared with the impact of data collection requirements. Successful measurement programs begin by involving all participants in defining the goals.

Because personnel at different organizational levels will view a new measurement program from different perspectives, the success of the program demands that those responsible for introducing measurement follow the next rule:

Set expectations.

The implementation of a measurement program will inevitably introduce change; change will bring some resistance and some initial problems. To minimize resistance, both management and technical personnel must be prepared to expect and accept the change and to encourage others to be persistent and patient. Proper setting of expectations will enhance potential support and acceptance from all management and technical personnel affected by the changes.

Plan to achieve an early success.

The first project should be selected carefully with the objective of demonstrating evidence of early benefits. Measurement programs sometimes fail because well-intentioned measurement coordinators wait too long "for all the results to come in" before reporting progress to senior management. It is critical to report preliminary results as soon as possible after establishing the program. The startup investment is significant, so management must see an early return on that investment, or the program is likely to be canceled before measurement analysts can provide "all the results." Equally important, project personnel need to see evidence of the benefits of their efforts to reduce their inevitable resistance. The early payoff may be, for example, a better understanding of the typical classes of errors that are detected in the organization's software projects or an understanding of the relative amounts of time that personnel spend in coding as compared with testing.

Although early feedback is essential for success, it is prudent not to promise substantial improvement during the early phases of the program. Worthwhile analysis, synthesis, and packaging take time and effort. Development and maintenance teams must be conditioned to expect gradual, incremental improvements.

3.2 Scope

After the goals of the measurement program are established and understood, measurement personnel must define the scope of the program, making the following critical decisions:

- Which projects should be included in the organization's measurement program?
- Which phases of the software life cycle should be included?
- Which elements of the project staff should be included; for example, is it important to include the effort of secretarial support, publication support, and two or more levels of management?

Those responsible for making these decisions must consider both the previously defined goals and the need to gain acceptance from project personnel who will be affected by the new measurement program. The next two rules provide help in defining the scope.

Focus locally.

The scope of the measurement program should be limited to the local organization. Organizational goals should have been based on the need for specific self-improvements, not for making comparisons with others. When defining processes for data collection and analysis, it is important to use concepts and terms that are understood locally. Precious effort should not be expended developing universal or unnecessarily broad-based definitions of measurement concepts and standards. Similarly, it is important to focus on developing a high-quality local measurement data center. Combining detailed measurement data into larger information centers has never proved beneficial and has consumed significant amounts of effort. Consultation with management and software personnel can ensure proper focus and increase acceptance.

Start small.

When establishing a measurement program, it is always important to start with a small scope. Limiting the number of projects, restricting the portions of the software life cycle to those with already well-defined processes within the organization, and limiting staff involvement to essential personnel will all help to minimize resistance from, and impact on, managers and development or maintenance personnel. The scope of the program will evolve, but the time to increase the size of the program is after it has become successful.

3.3 Roles, Responsibilities, and Structure

After the organizational goals are well understood and the scope of the measurement program is defined, the next step is to define roles and responsibilities. In a successful measurement program, three distinct roles must be performed by components of the organization:

1. *The source of data*—providing measurement data from ongoing software development and maintenance activities
2. *Analysis and packaging*—examining measurement data and deriving process models and relationships
3. *Technical support*—collecting, storing, and retrieving project information

Figure 3-1 illustrates the components and the relationships among them. Each component must perform its distinct role while maintaining a close relationship with the other two components.

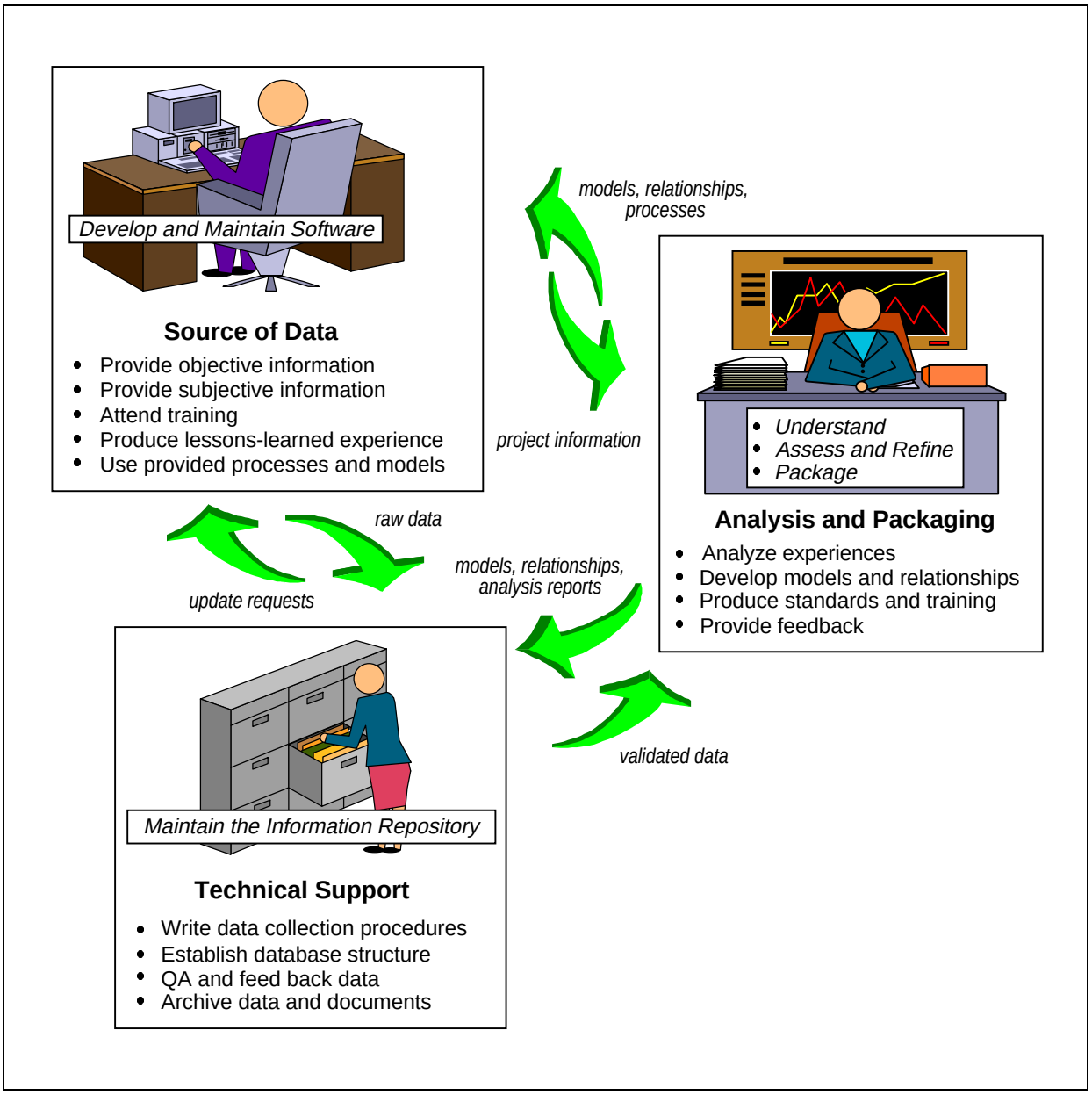


Figure 3-1. The Three Components of a Measurement Program

The next sections introduce the components' responsibilities in starting a measurement program and map the components into the organizational structure. (Chapter 5 briefly describes the operational responsibilities of the three components.)

3.3.1 The Source of Data

The responsibility of the development and maintenance component is to provide project data. Providing data is the *only* responsibility imposed on the development and maintenance personnel; they are not responsible for analyzing the data. These personnel can reasonably expect to be provided with training that includes, at a minimum, the following information:

- Clear descriptions of all data to be provided
- Clear and precise definitions of all terms
- Who is responsible for providing which data
- When and to whom the data are to be provided

In exchange, the development and maintenance component of the measurement program receives tailored processes, refined process models, experience-based policies and standards, and tools.

3.3.2 Analysis and Packaging

The analysis and packaging component is responsible for developing and delivering the training that will provide the developers and maintainers with the specific information listed in the previous section. Analysis and packaging personnel must design and develop the data forms and receive the raw data from the repository. They are responsible for examining project data; producing tailored development and maintenance processes for the specific project domain; generating organization-specific policies and standards; and generalizing lessons, information, and process models. This measurement program component continually receives data from the developers and maintainers of software and, in return, continually provides organization-specific experience packages such as local standards, guidebooks, and models.

Organize the analysts separately from the developers.

The analysis and packaging personnel are necessarily separate from the development and maintenance personnel because their objectives are significantly different. Measurement analysts are concerned solely with improving the software process. Software developers' and maintainers' concerns include product generation, schedules, and costs. It is impractical to expect personnel who must deliver a high-quality product on schedule and within budget to be responsible for the activities necessary to sustain continual improvement; hence, those functions must be the responsibility of a separate component.

3.3.3 Technical Support

The technical support component maintains the information repository, which contains the organization's historical database. This component provides essential support services including implementing the database as specified by the analysis and packaging component. The support personnel collect data forms from the developers and maintainers on a prescribed schedule, perform data validation and verification operations to identify and report discrepancies, and add the project data to the historical database. They are also responsible for operating supplementary software tools (e.g., code analyzers) and for preparing reports of the analysis results. In addition, the support personnel archive data and perform all other database management system (DBMS) maintenance functions.

Example:
The Software Engineering Laboratory

Although their measurement roles and responsibilities are clearly distinct, the three components may beorganized in different ways within different organizations. A large organization may benefit by creating separate, structural components to perform the three distinct roles of the measurement program. A small organization with a small project may simply assign the roles to individual personnel. In some cases, a single individual may perform multiple roles as long as the amount of effort allocated to separate roles is clearly identified.

For example, the SEL is an organization of moderate size with approximately 300 software developers and maintainers. The organization develops and maintains mission support software for the Flight Dynamics Division at GSFC. Since 1976, the SEL has collected data from more than 100 software development projects. Typical projects range in size from 35,000 to 300,000 SLOC and require from 3 to 60 staff-years of effort. The process and product data have been analyzed to evaluate the impact of introducing methodologies, tools, and technologies within the local environment. In recent years, the SEL has expanded the scope of its activities to include the study of software maintenance (Reference 15). Process improvements have led to documented improvements in the organization's products.

Figure 3-2 illustrates the organizational structure of the SEL. In this example, the technical support personnel who maintain the repository are administratively affiliated with the analysis and packaging component but physically located with the source of data. This structure works well in the SEL for two reasons:

1. The technical support personnel receive funding from the same source as the analysis and packaging personnel. Developers and maintainers are funded by a different source.

2. The physical environment is structured with the forms processing, database host computing support, and library facilities collocated with the developers and maintainers, so the support personnel occupy that same space.

Many alternative structures would be just as functional and successful. The important feature is that the development and maintenance personnel are not responsible for analysis and packaging. In addition, SEL models and relationships are affected by the fact that the measurement program within this sample environment is limited to development and maintenance of operational mission support software.[5] Organizations that include other activities may derive significantly different models. Issues related to the cost considerations shown in the figure are addressed in Section 3.5. Reference 16 provides additional examples and details.

[5] Although the scope of the measurement program includes no data from prototype development or research activities, the software personnel do perform such activities as a part of their jobs.

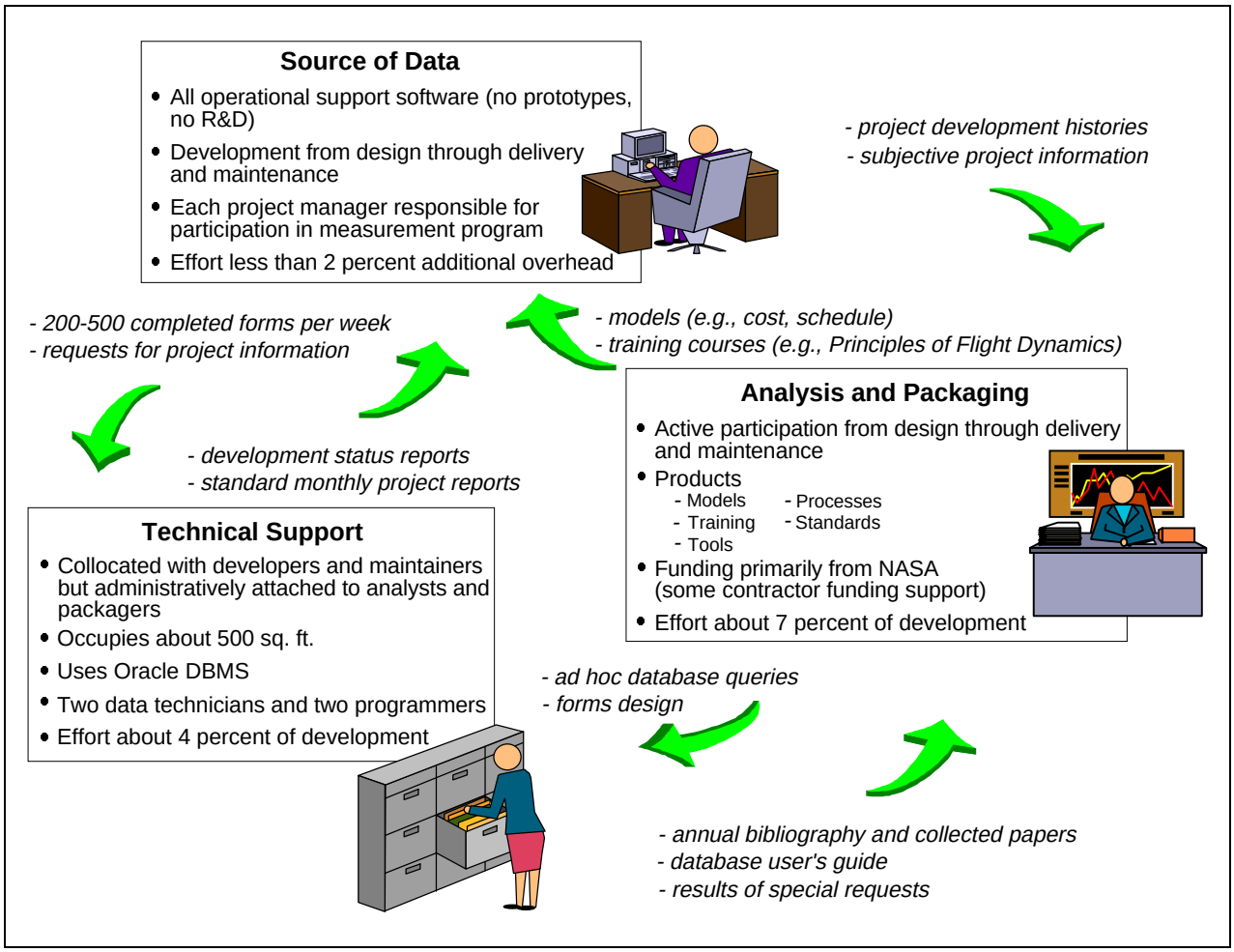


Figure 3-2. The SEL as a Sample Structure for Process Improvement

3.4 Selecting the Measures

Another important step in establishing a measurement program is selecting the measures to be used. Selected measures will fall into one or more categories, including objective measures (direct counts, obtained either manually or with the support of an automated tool), subjective measures (interpretive assessments about the status of the quality or completion of the product), and project characteristics (factual descriptions of the type, size, and duration of the project). Chapter 4 addresses measures in more detail. When selecting measures, the next rule is the most important:

> ***Make sure the measures apply to the goals.***

Measures should not be selected just because a published author has found them useful; they should directly relate to the defined goals of the organization. For example, if there is no goal to reduce processor time, it is a waste of time and effort to collect data on computer usage.

Keep the number of measures to a minimum.

Experiences from successful measurement programs within NASA suggest that a minimal set of measures is usually adequate for beginning a program and sufficient to fulfill all but the most ambitious goals. A basic set of measures—which typically consists of data for schedule, staffing, and software size—is introduced in the next chapter.

This rule—to limit the number of measures and, by implication, the size of the measurement database—is a corollary of the rule to start small, which suggests limiting the scope of the measurement program itself. The rule should be taken literally: if a single measure is sufficient to address the organization's goal, then collecting data on two or three will provide no added benefits. For example, if the only goal is to improve quality, only defects should be measured; cost and schedule data should not be a concern.

Avoid over-reporting measurement data.

Any measurement program can be potentially disruptive to a software project; therefore, analysts must be cautious when providing feedback to development and maintenance personnel. Providing too much feedback can be just as serious a mistake as providing not enough. Reporting the results of analyzing all available measurement data is a waste of time, because much of the information will provide no additional insight. When presented with unnecessary and excessive charts, tables, and reports, software staff and managers may become annoyed and disenchanted with the value of the measurement program.

Collected data constitute only a small part of the overall improvement program and should always be treated as the means to a larger end. The tendency to assume that each set of data has some inherent value to the development and maintenance personnel and, therefore, should be analyzed, packaged, and fed back to them, must be avoided. Feedback must be driven by a need or directed toward supporting a defined goal. If no focus has been established for the analysis of code complexity, for example, then there will be no value in—and no appreciation for—the preparation of a complexity report. Such a report would be disruptive and confusing and could dilute the effectiveness of the measurement program.

The following common reports and graphs are often packaged and provided to the development and maintenance organization, not because they are needed, but simply because the data exist:

- Code complexity
- Design complexity
- Number of tests executed
- Plots of computer usage
- Charts of numbers of requirements changes
- Profiles of program execution

- Charts of the time spent in meetings

Each of those measures may have some value when used in support of an organizational goal. However, this type of information is too often reported because it is assumed to be inherently interesting, not because it relates to a particular need or goal.

3.5 Cost of Measurement

Cost is one of the most critical, yet misunderstood, attributes of a software measurement program. Many organizations assume that the cost of measurement is so excessive that they cannot justify establishing a measurement program. Others claim that measurement can be a nonintrusive, no-cost addition to an organization and will have no impact on the organization's overhead. The truth lies somewhere in between.

Budget for the cost of the measurement program.

Measurement is not free, but it can be tailored in size and cost to fit the goals and budgets of any software organization. A measurement program must be undertaken with the expectation that the return will be worth the investment. If the cost is not planned in the organization's budget, there will be frustrations, attempts at shortcuts, and a failed software measurement program. Planning must incorporate all of the hidden elements of the proposed effort—elements that are often more expensive during startup than after the measurement program becomes operational. The higher startup cost is an additional reason to *start small.*

Planners often incorrectly assume that the highest cost will be to the software development or maintenance organization. This part of the overhead expense, which includes completing forms, identifying project characteristics, and meeting with analysts, is actually the least expensive of the three major cost elements of the measurement program:

1. Cost to the software projects—the source of data
2. Cost of technical support
3. Cost of analyzing and packaging

The cost of the measurement program also depends on the following considerations of scope:

- Size of the organization
- Number of projects included in the measurement program
- Extent of the measurement program (parts of the life cycle, number of measures, etc.)

NASA experience shows that there is a minimum cost associated with establishing and operating any effective measurement program. The total cost will increase depending on the extent to which the organization wants, or can afford, to expand the program to address additional projects, more comprehensive studies, and broader measurement applications.

The cost information offered in this section is based on 17 years of experience from organizations ranging in size from approximately 100 to 500 persons. Additional information has been derived

from measurement programs in larger organizations of up to 5,000 persons. The number of projects active at any one time for this experience base has ranged from a low of 5 or 6 projects to a high of over 20 projects, ranging in size from 5 KSLOC to over one million SLOC. Because measurement costs depend on a large number of parameters, citing a single definitive value that represents the cost of any organization's measurement program is impossible. However, some general suggestions can be provided, and organizations can interpret these suggestions in the context of their own goals and environments.

Generally, the cost of measurement to the development or maintenance project will not exceed 2 percent of the total project development cost and is more likely to be less than 1 percent (which implies that the cost may be too small to be measured). The technical support element may reach a constant staff level of from one to five full-time personnel for data processing support. The analysis and packaging element will require several full-time analysts and may cost up to 15 percent of the total development budget. For example, the SEL spends an average of about 7 percent of each project's total development budget on analysis and packaging.

Figure 3-3 illustrates the costs of the elements of a software measurement program as percentages of the total organizational cost. Individual costs are discussed in more detail in the following sections.

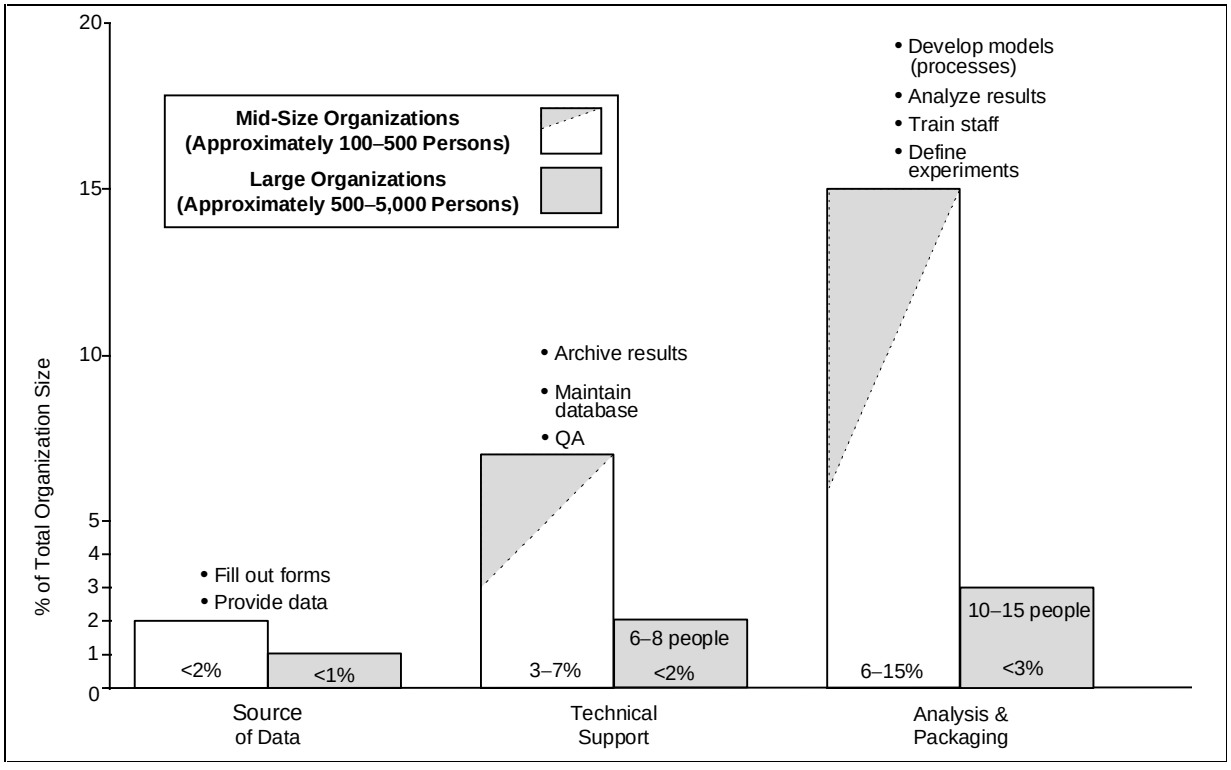


Figure 3-3. Cost of Software Measurement

3.5.1 Cost to the Software Projects

The cost of measurement should not add more than 2 percent to the software development or maintenance effort.

The smallest part of the measurement cost is the overhead to the development and maintenance organization. This overhead comprises the cost of completing forms, participating in interviews, attending training sessions describing measurement or technology experiments, and helping to characterize project development. Although startup costs may be as high as 5 percent of the development budget, the cost of operating an effective program will normally not exceed 1 or 2 percent, regardless of the number of active projects within the organization.

Legitimate costs are associated with introducing the providers of data to a new measurement program. However, part of the higher initial cost can often be attributed to the inefficiencies in an inexperienced organization's program. New programs typically ask developers or maintainers to complete unnecessary forms or require excruciating detail that is of little value or is not a part of the stated goal. A well-planned measurement program will never impose a significant cost impact on the development or maintenance organization.

3.5.2 Cost of Technical Support

The technical support component of the measurement program may cost from 3 to 7 percent of the total development budget.

Technical support encompasses collecting, validating, and archiving the measurement data. Included in these activities are database management, library maintenance, execution of support tools, and high-level reporting of summary measurement data. These essential activities must be planned, supported, and carefully executed. In addition to the cost of personnel are the costs of acquiring and maintaining database software, support tools, and other automated processing aids (e.g., code analyzers).

In an organization of over 50 management, technical, and clerical personnel, any measurement program will require three to five full-time staff members to handle the necessary support tasks. A smaller organization, with perhaps only one project and a pilot measurement program, may wish to combine the support effort with configuration management (CM) or independent QA activities. Implementation of a separate technical support element may not be cost effective.

Experience within NASA has shown that the cost of the technical support for measurement programs involving 100 to 200 software developers or maintainers is approximately 7 percent of the total effort. That cost includes approximately five full-time data technicians and database support personnel, plus the costs of the DBMS and associated software tools and equipment. For larger measurement programs with 250 to 600 software personnel, experience indicates that only one additional full-time support person is required. Thus, for organizations with 50 to 600 developers and maintainers, the overhead cost is approximately 6 percent of the project cost. For organizations with approximately 500 to 1,000 software personnel, the overhead cost approaches 3 percent of the project cost or about seven full-time personnel added to the cost of tools and equipment.

The cost estimates are based on the assumption that an organization is actively working on 5 to 15 development or maintenance projects at any one time. The overall cost of the technical support component will vary significantly depending on the number of projects participating in the measurement program. An organization of 200 or 300 people actively working on a single large project will require much less support than the same organization with 20 active smaller projects. Limited experience with larger organizations of over 5,000 persons indicates that the technical support cost is essentially the same as for an organization of 500. As its size increases, an organization tends to collect measurement data at a less detailed level.

3.5.3 Cost of Analysis and Packaging

The cost of the analysis component of the measurement program ranges from 5 to 15 percent of the total project budget.

Analysis and packaging is the most critical part of the measurement program and the most costly of the three elements of cost overhead. Without a sufficient allocation of effort to this function, the measurement program cannot be a success. Packaging is the culmination of all measurement activities and the primary purpose for the measurement program.

Key activities associated with this element are

- Design of process studies (determining what is to be measured)
- Information analysis (e.g., analysis of data and synthesis of models)
- Project interaction (clarifying the purposes of measurement, training developers, providing feedback to projects)
- Packaging (producing standards, policies, and training programs and capturing assessments of analyzed processes)

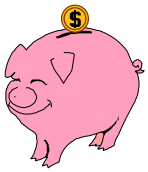
Plan to spend at least three times as much on data analysis and use as on data collection.

NASA experience shows that the cost of this element in successful measurement programs far exceeds the combined costs of the other two and is typically about three times the amount that the software projects spend providing data. A successful measurement program dictates that this cost be recognized and budgeted. For measurement programs involving 50 to 250 software developers or maintainers, the cost of this activity has consistently run from approximately 7 to 12 percent of the organization's total budget. Costs are incurred by the researchers who design studies and develop new concepts, by the process staff responsible for developing and writing standards, and by all the personnel required for analyzing, providing feedback, and developing improvement guidelines. The analysis and packaging portion of the measurement costs depends on the number of projects active within the organization. The figures provided here assume at least 10 active projects and an archive of data from at least 15 projects available for analysis. With fewer active projects, the analysis overhead would be smaller than indicated.

NASA's historical data indicate that organizations spending between $20 million and $30 million for development and maintenance projects have spent between $1 million and $3 million for extensive and mature analysis efforts (in fiscal year 1993 dollars). For efforts on a much larger scale, the measurement analysis must necessarily be conducted on a comparably higher level; consequently, the overhead percentage decreases significantly. An expenditure of an equivalent amount of analysis resources, plus a modest increase due to the size of the organization, need not exceed the 5 percent level for measurement programs of any size. Because application of the measurement data is the primary reason for the measurement program, adequate resources must be allocated for this critical measurement program element.

Chapter 4. Core Measures

Chapter Highlights

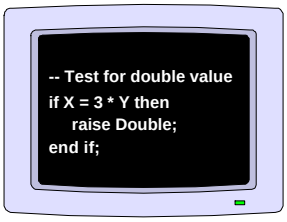


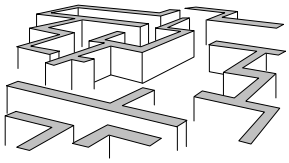
COST

- Reporting period dates
- Total effort
- Effort by development and maintenance activity

ERRORS

- Dates error reported and corrected
- Effort to isolate and correct the error
- Source and class of error



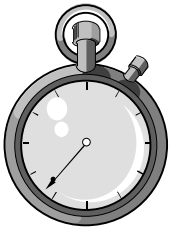


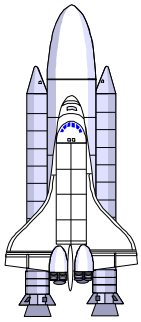
PROCESS CHARACTERISTICS

- Identification of programming languages
- Indication of the use of significant processes
- Description of measurement study goals

PROJECT DYNAMICS

- Changes to requirements
- Changes to code
- Growth of code
- Predicted characteristics





PROJECT CHARACTERISTICS

- Development dates
- Total effort
- Product size
- Component information
- Software classification

This chapter describes a set of core measures that any organization can use to begin a measurement program. There is no universal, generally applicable collection of measures that will satisfy the needs and characteristics of all organizations. However, on the basis of the experiences of mature measurement programs throughout NASA, a set of measures in the following five categories will typically be required by any software development and maintenance organization:

1. Cost
2. Errors
3. Process characteristics
4. Project dynamics
5. Project characteristics

Although organizations beginning a measurement program may want to use the core set as a baseline, they will soon find that additional information is required to satisfy their specific goals and that some of the core measures are not required. Each organization should use those measures that reflect its own goals. As its measurement program matures, the organization will recognize which measures support those goals and which provide no added value.

The recommended core measures in each of the categories exhibit the following important attributes. They

- Address the three key reasons for measurement
  1. Understanding
  2. Managing
  3. Guiding improvement
- Support both software development and software maintenance activities
- Are easy to collect and archive
- Are based on the experience of mature NASA measurement programs

The following sections provide further information on the core measures.

4.1 Cost

Cost is the most universal and commonly accepted measure for understanding and managing software processes and products. Consequently, cost data represent the most essential part of any measurement program. Although many development organizations assume that the cost data must be extensive and detailed to capture the overall cost characteristics of a software project adequately, the cost data should actually be easy to capture. If a programmer needs more than a few minutes each week (on the average) to record his or her effort, then the forms require too much data. As long as the managers are aware of the total amount of effort required for the software projects, an organization can gain a significant amount of insight by observing the trends

over time. The simplest, yet most critical, cost measure is the record of the total expenditures for a project.

4.1.1 Description

Collect effort data at least monthly.

Every project must capture staff effort data on a consistent, periodic basis. A monthly schedule is recommended, at a minimum; however, many major NASA measurement programs capture effort data biweekly or even weekly. The higher frequency requires little additional work and provides more project characterization detail.

Clarify the scope of effort data collection.

The scope of the effort data collection depends on the organization's goals. Each organization must determine precisely who will supply effort data, at what point during the software life cycle measurement will begin, and when data collection will terminate. Typically, effort data must be collected for all personnel who charge their time to the software project, specifically, technical, management, secretarial, and publications staff.

For every data reporting period, each individual must minimally report the total number of hours of effort and a breakout of the number of hours per activity (e.g., design, code, test, or other).

A decision concerning the reporting of unpaid extra hours of effort must be based on whether the intent is to measure the actual effort expended or the actual effort charged. Some organizations maintain separate records of unpaid overtime hours.

Within the SEL, every programmer and every first- or second-line manager provide effort data. Data collection starts when the functional requirements have been completed and the software life cycle begins with the requirements analysis phase.[6] For *development* projects, data collection continues until the system is turned over for operational use. For *maintenance* projects, data collection starts at the beginning of the operations phase and continues until the analysts determine that no additional value will be gained from further collection. Each maintenance project is judged on its own merits. Some may provide data for 1 year only, whereas others provide data until the software is retired.

4.1.2 Data Definition

When the measurement program is first established, personnel from the analysis component must define the activities to ensure clarity and internal consistency. Focus should be on using locally

[6] For all five categories of measures, the SEL begins to capture data no earlier than the beginning of the software requirements analysis phase. System requirements definition is normally performed by a different organization from the one that develops the software.

developed definitions for the activities. Excessive time should not be spent trying to be consistent with outside organizations.

All project personnel (e.g., programmers, managers, QA staff, CM staff, and testers) provide the data listed in Table 4-1. Additional resource data on the documentation effort (total hours by publications) and the clerical effort (total hours charged by secretarial support) may be extracted from project management accounting records, as long as there is a definition of scope and characteristics. The data must be consistent from project to project and should provide an accurate history of the cost required to produce and to maintain the software product.

Table 4-1. Data Provided Directly by Project Personnel

Data	Descriptions
All Effort	
Date	Date of the end of the reporting period
Total effort	Total hours charged to the project during that period
Development Activity Only	
Hours by development activity	Predesign Create design Read and review design Write code Read and review code Test code units Debugging Integration test Acceptance test Other
Maintenance Only	
Hours by maintenance class	Correction Enhancement Adaptation Other
Hours by maintenance activity	Isolation Change design Implementation Unit test and system test Acceptance test and benchmark test Other

The SEL Personnel Resources Forms (see Figures A-5 and A-6 in Appendix A) and the Weekly Maintenance Effort Form (see Figure A-13) are examples of forms used to capture effort data for development and maintenance projects, respectively. Programmers and managers typically complete a form every week. Both forms provide space for recording total hours and the distribution of hours by activities. To reduce questions and confusion, the definitions of the

activities are supplied on the forms. Other organizations may use different definitions as long as they are applied consistently throughout the organization's measurement program.

Figure 4-1 summarizes the life-cycle phases, sources, and frequency for cost data collection. Typically, organizations separate the costs of development and maintenance activities.

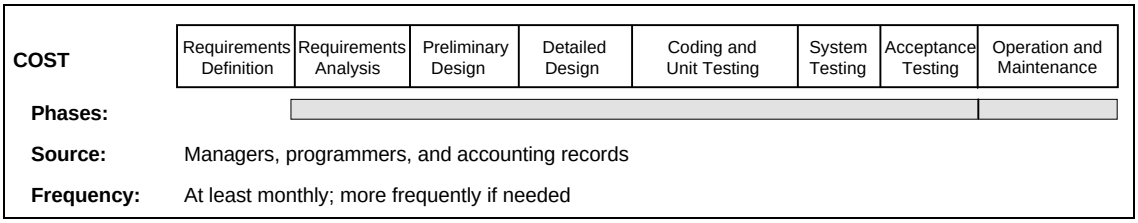


Figure 4-1. Cost Data Collection Summary

4.2 Errors

Error data make up the second most important category of core measures. A better understanding of the characteristics of software defects is necessary to support a goal of higher quality and greater reliability. Error data may minimally include only counts of defects detected during a specific life-cycle phase; at the other extreme, error data may include detailed descriptions of the characteristics of the errors and information on where the errors came from, how they were found, and how they were corrected. The level of detail must be driven by the goals and needs of the particular organization. This section recommends core error measures based on those collected within a successful measurement program in a medium-sized NASA organization.

4.2.1 Description

The core error measures consist of the

- Date the error was found
- Date the error was corrected
- Effort required to isolate and correct the error
- Source of the error
- Error class

When the measurement program is first established, the measurement analysts must define the scope of the error reporting activity.

Collect error data only for controlled software.

Error data should be captured only after a unit of software has been placed under configuration management control. This recommendation, which is based on 17 years of experience, may seem

counterintuitive. However, until CM checkout and checkin procedures have been established as prerequisites for making changes, consistent error reporting cannot be guaranteed. Within the SEL, a unit is turned over for configuration control only after it has been *coded*. Other NASA organizations (e.g., JPL) have reported significant improvements from collecting and analyzing data about defects detected and corrected during formal inspections of requirements documents (see Reference 26).

Do not expect to measure error correction effort precisely.

Programmers focusing on their technical activities may not be able to report the exact amount of time required for a particular change. Forms should allow them to estimate the approximate time expended in isolating and correcting an error.

4.2.2 Data Definition

After completing a software change, a programmer submits the appropriate change form with the data shown in Table 4-2. A change form is required whenever a controlled software component is modified, whether or not the detection of an error necessitated the change. Experience has shown that the process of reporting such changes enhances configuration management and that the information proves useful in modeling the dynamics of the software in an organization. In addition to the measures already cited, a maintenance change form must include the type of modification. As always, it is important to focus locally when defining the error classes.

Table 4-2. Change Data

Data	Descriptions
All Changes	
Date error reported	Year, month, and day
Date error corrected	Year, month, and day
Source of error	Requirements, specification, design, code, previous change, other
Class of error	Initialization, logic/control, interface, data, computational
Effort to isolate error	Approximate number of hours
Effort to implement change	Approximate number of hours
Maintenance Changes Only	
Type of modification	Correction, enhancement, adaptation

The SEL Change Report Form and the Maintenance Change Report Form (see Figures A-1 and A-4 in Appendix A) are examples of forms used to capture error data for development and maintenance projects, respectively. In either case, a single form is used to report both software errors detected and software changes to correct the errors. Programmers use only one form to report one error that requires changes to multiple components.

Figure 4-2 summarizes the life-cycle phases, sources, and frequency for error data collection.

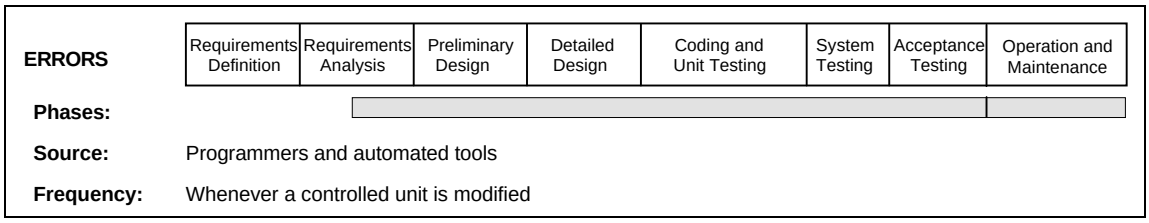


Figure 4-2. Error Data Collection Summary

4.3 Process Characteristics

> ***Do not expect to find generalized, well-defined process measures.***

Focusing on the process characteristics category of software measures allows investigation into the effectiveness of various software engineering methods and techniques. Looking at process characteristics also provides insight into which projects use related processes and can thus be grouped together within the measurement program to derive models and relationships or to guide improvements.

Because few process features are consistently defined and can be objectively measured, few core measures are recommended in this category. Rather than capturing extensive process characteristics, it is suggested that some basic information be collected about the development process used for the project being measured.

4.3.1 Description

The recommended core process measures are limited to the following three:

1. Identification of development language(s)
2. Indication of the use of specific processes or technology [e.g., the Cleanroom method or a particular computer-aided software engineering (CASE) tool]
3. Description of measurement study goals

Common descriptions of measures do not exist for such fundamental software engineering process elements as methodology, policies, automation, and management expertise. Therefore,

recommending that such measures be included in the core set is not useful. Measures such as these must be defined and analyzed locally for consistency with the organization's goals.

Do not expect to find a database of process measurements.

Detailed process descriptions cannot be stored in a database. Instead, important process information is often provided in papers and reports. For example, if an organization is studying the impact of using different testing strategies, the analysts must capture the detailed information about the results of applying different techniques and report on the results.

Understand the high-level process characteristics.

Before attempting to capture advanced process measurement data, an organization must have a clear understanding of the core process measures. Experience within the SEL has shown that the most important process characteristic is the choice of programming language; the availability of this information may provide further insight during the analysis of other measurement data.

4.3.2 Data Definition

Table 4-3 summarizes the core process characteristics measures. Figure 4-3 summarizes the life-cycle phases, sources, and frequency for process characteristics data collection.

Table 4-3. Process Characteristics Data

Data	Descriptions
Development language	Language name: percentage used Language name: percentage used ...
Important process characteristics (if any)	One-line textual description (e.g., "used Cleanroom")
Study goals	Brief description of the goals and results of the measurement study associated with the project

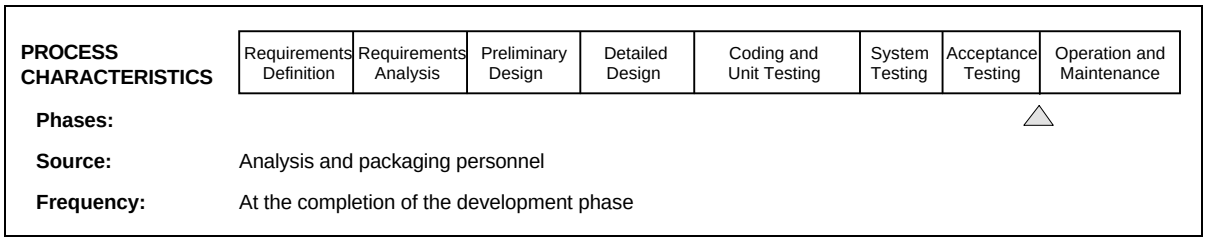


Figure 4-3. Process Characteristics Data Collection Summary

4.4 Project Dynamics

The next category of core measures—project dynamics—captures changes (to requirements, to controlled components, and in the estimates for completion) during the software life cycle. Experience has shown that such information aids management and improves understanding of the software process and product.

4.4.1 Description

The core measures in this category characterize observed changes in the project requirements and the product code, as well as updated estimates of the final project characteristics (see Section 4.5). These measures consist of

- Changes to requirements
- Changes to baseline code
- Growth in baseline code
- Predicted project characteristics

Requirements changes represent the overall stability of the software requirements and can be used effectively to manage the development effort and to improve understanding of the characteristics of the software problem definition in the local environment.

Records of changes to the code and the growth of the code provide insight into how the various phases of the life cycle affect the production of software, the most tangible product that a development process generates. Change measures are useful in managing ongoing configuration control processes, as well as in building models of the development process itself.

The measures of predicted project characteristics are excellent management aids and are useful for studying the cause and effect of changes, as well as process and problem complexity. The characteristics should be captured on a regular basis, at least monthly.

4.4.2 Data Definition

The Project Estimates Form (see Figure A-8 in Appendix A) is an example of a form used to provide predicted project characteristics at the start of the project and periodically throughout the life cycle. Table 4-4 summarizes the core project dynamics measures, and Figure 4-4 summarizes the life-cycle phases, sources, and frequency for project dynamics data collection.

Table 4-4. Project Dynamics Data

Data	Descriptions
Changes to requirements	Count and date of any change made to the baselined requirements specifications
Changes to code	Weekly count of the number of software components changed
Growth of code	Biweekly count of the total number of components and total lines of code in the controlled library
Predicted characteristics	Monthly record of the estimated completion dates and software size
Dates	End design End code End testing System completed
Size	Total components Total lines of code (new, reused, modified)
Effort	Total staff months (technical, management, support services)

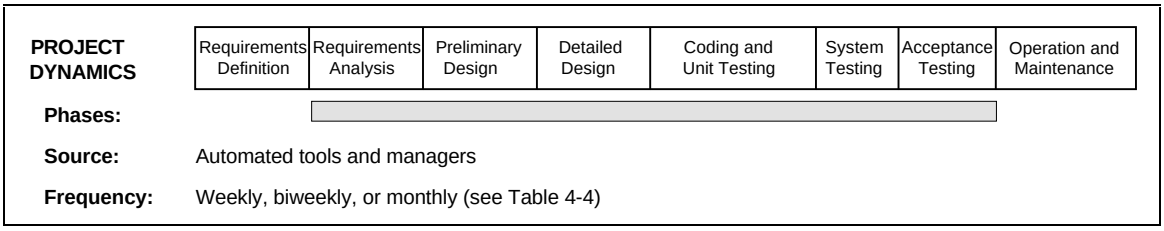


Figure 4-4. Project Dynamics Collection Summary

4.5 Project Characteristics

The core measures that characterize the completed project constitute another essential part of the measurement program. Organizations derive models and relationships from project characteristics in the historical database. Without a basic description of the overall software project effort, it is difficult to apply the other measurement information in a meaningful manner.

4.5.1 Description

The project characteristics can be broken down into five categories of core measures:

1. Development dates
2. Total effort
3. Project size
4. Component information
5. Software classification

Use simple definitions of life-cycle phases.

The important dates are the beginning and the end of each life-cycle phase and the final project completion date. If the organization is using a strict waterfall life cycle with nonoverlapping phases, then the end of a nonterminal phase is defined by the beginning of the subsequent phase. When a different life-cycle methodology is applied, the organization will have to adjust the structure of the project characteristics data. Each organization must determine how it wants to capture details of the key phase dates within the software life cycle. The simplest approach is to use the classical phase definitions of a standard life-cycle methodology. However, as long as an organization has its own consistent internal definitions, there is no overwhelming reason to adopt an external standard. Multiple releases can be treated as multiple projects or as a single project followed by maintenance enhancements.

The total effort expended on the project should be divided into hours used by programmers, managers, and support services. At the conclusion of the project, the totals should be determined from accounting information or another official source. The sum of the effort data collected during the development or maintenance project should be compared with the value obtained from the alternative source to cross-check the accuracy.

The core size measures are the total size of the software product and the total number of components within the product. NASA experience shows that archiving additional details about the origin of the code (e.g., whether it is new, reused, or modified) can lead to useful models.

Use lines of code to represent size.

NASA programs typically measure software size in terms of lines of code. Some authorities recommend other size measures [e.g., function points (see Reference 17)]. However, no other measure is as well understood or as easy to collect as lines of code.

This guidebook also recommends collecting size and origin information for software components and defines a software component as a separately compilable unit of software for the project being measured. Some organizations define components as subprograms or subsystems, which is fine as long as the organization applies that definition consistently and derives useful results. The SEL

captures the basic information for each separately compilable unit of source code and has found that the overhead required to extract the information using an automated tool is trivial. As a result, programmers can be freed from expending additional effort in providing that information.

The final category of project characteristics core measures is software classification. This measure is abstract and of limited value. Consequently, most organizations are advised to spend only limited effort collecting and analyzing classification data. Nevertheless, several NASA organizations have found a high-level classification scheme to be both adequate and useful. These organizations use three broadly defined classes:

1. Business or administrative applications
2. Scientific or engineering applications
3. Systems support

Other organizations may want to record more detailed classification data, such as

- Embedded versus nonembedded
- Real-time versus nonreal-time
- Secure versus nonsecure

4.5.2 Data Definition

The recording of project characteristics data can often be substantially automated to minimize the burden on the development and maintenance organization. Dates and effort, for example, are normally available from management accounting reports; automated tools frequently can be used to report size and component information, and the time and effort needed to indicate software classification is minimal. Table 4-5 summarizes the project characteristics data.

No universally accepted definition exists for the start and stop times of various phases, such as when a project starts or when a design ends. Experience within NASA has led to the use of phase dates as follows:

- *Start of software development*—delivery of system requirements documents
- *End of requirements analysis*—completion of specifications review
- *End of design*—completion of design review
- *End of coding*—completion of code and unit test
- *End of testing*—delivery to acceptance testing
- *End of development*—delivery to operations

Table 4-5. Project Characteristics Data

Data	Descriptions
Dates	
Phase start dates (year, month, and day)	Requirements analysis Design Implementation System test Acceptance test Cleanup Maintenance
End date	Project end
Effort	
Total hours	Project total Management personnel Technical personnel Support personnel (e.g., publications), if applicable
Size	
Project size (lines of code)	Delivered Developed Executable Comments New Extensively modified Slightly modified Reused
Other (count)	Number of components Pages of documentation
Component information (for each component)	
Component size (lines of code)	Total Executable
Component origin	New Extensively modified Slightly modified Reused
Software classification	Business/administrative Scientific/engineering Systems support

The effort data, compiled at the conclusion of the project, are used as part of the high-level summary information for the project. The information represents the total cost of the project broken down among developers, managers, and support services.

Table 4-5 lists several measures for lines of code. Consensus may never be reached on what constitutes a line of code. Therefore, to facilitate various forms of comparison and analysis, this guidebook recommends recording multiple values. The core measures include counts of

- *Total lines delivered*—every logical line, including comments, blanks, executable, and nonexecutable
- *Developed lines*—total lines with a reuse factor
- *Executable statements*—total number of executable statements
- *Comment lines*—total number of lines containing only comments or blanks

The SEL captures source lines of code in four categories:

1. *New*—code in new units
2. *Extensively modified*—code for reused units in which 25 percent or more of the lines were modified
3. *Slightly modified*—code for reused units in which fewer than 25 percent of the lines were modified
4. *Reused verbatim*—code for units that were reused with no changes

For estimation purposes, lines of code are often classified into two categories that combine newly written and extensively modified units as *new* code and slightly modified and verbatim code as *reused* code. Consequently, the SEL relationships (see Reference 9) for estimating developed lines are

FORTRAN developed lines = new lines + 20% of reused lines

Ada developed lines = new lines + 30% of reused lines

(See Sections 2.2.1 and 6.1.2 for more discussion of developed lines of code.)

Specify which software is to be counted.

It is important to be specific about which software is to be included in the size counts. For example, it is usually appropriate to exclude throw-away prototypes, test harnesses, and commercial off-the-shelf (COTS) software from the reported totals.

Component information can provide insight into the overall development characteristics. Although the total amount of information may be extensive, it should be easy to compile at the conclusion of the project and can be almost completely retrieved via automated software tools such as code counters, auditors, or analyzers.

The Project Completion Statistics Form (see Figure A-7 in Appendix A) is an example of a form used for collecting project characteristics at the completion of a project. Figure 4-5 summarizes the life-cycle phases, sources, and frequency for project characteristics data collection.

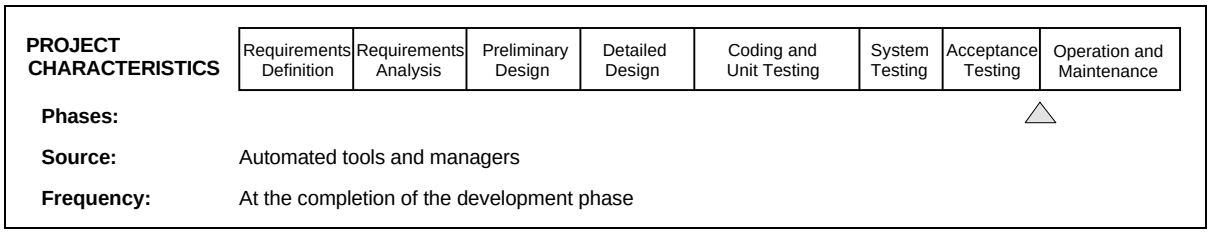


Figure 4-5. Project Characteristics Collection Summary

Chapter 5. Operation of a Measurement Program

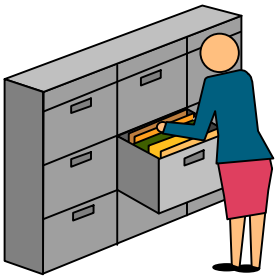
Chapter Highlights

DEVELOPMENT AND MAINTENANCE

- Providing data
- Participating in studies

TECHNICAL SUPPORT

- Collecting data
  - Interface with data providers
  - Definitions
- Storing data and assuring data quality
- Summarizing, reporting, and exporting data



ANALYSIS AND PACKAGING

- Designing studies
- Analyzing data
- Packaging the results
  - Policies and standards
  - Training
  - Automated tools
  - Reports
  - Updates

Having established a measurement program, the organization must shift its emphasis to operation. Chapter 3 introduced the three organizational components of a measurement program: development and maintenance, technical support, and analysis and packaging. After briefly describing mechanisms for collecting project data, this chapter expands on the operational responsibilities of those three components.

Figure 5-1 illustrates that mechanisms for data collection fall into the three primary categories listed below. Each category provides a particular type of data and requires a specific interface between pairs of organizational components.

1. *Printed forms*—The forms are designed by the analysis and packaging component, completed by the development and maintenance component, and submitted directly to the technical support component. All forms require the submitter to provide identifying information, such as the project name, the team member's name, and the date. In addition, each type of form is designed to provide some of the measures that satisfy the goals of the measurement program. Some forms request both objective data (directly observed) and subjective data (based on opinion). All require only short answers or the selection of options from a checklist. Appendix A includes a sample set of data collection forms used in the SEL and designed to provide the measurement data stored in the SEL's historical database. An organization establishing a measurement program can use these forms as a starting point in designing its own set of organization-specific forms.
2. *Automated tools*—Some data can be collected automatically and unobtrusively by software tools. For example, code analyzers and compilers can count lines of code; operating system accounting packages can supply data about processor and tool usage; and organizational accounting systems can typically report hours of effort by interfacing with the time card system.
3. *Personal interviews*—Some information can be captured only during personal interviews. Interviews are typically used to obtain subjective information about project status and to verify preliminary results of data analysis.

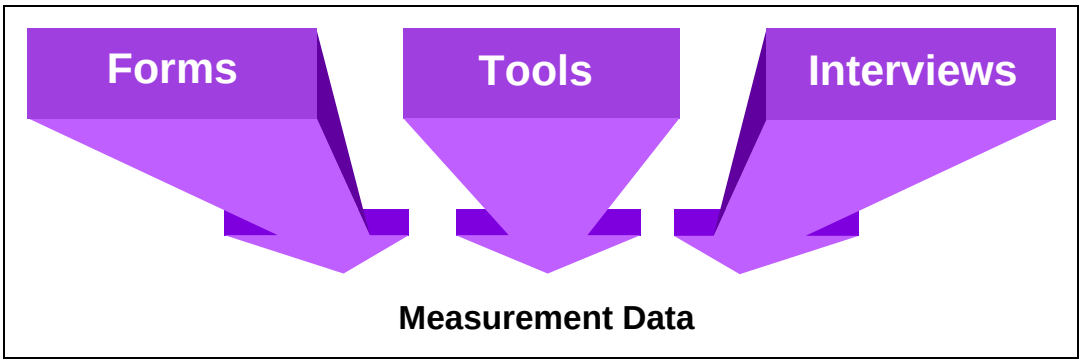


Figure 5-1. Three Data Collection Mechanisms

Occasionally, there may be other process and product information sources that do not fall neatly into one of the three categories. For example, personnel often have insights during document or code reviews. Any information that can be useful within the organization's measurement program should be exploited.

Figure 3-1 illustrates the operational relationships among the three components of the measurement program.

5.1 Development and Maintenance

Personnel whose primary responsibility is developing or maintaining software must not be burdened with heavy measurement program duties.[7] The measurement program must be designed so that it is deemed to be a help, not a hindrance, to development and maintenance personnel. The operational responsibilities of the development and maintenance component are

- Providing data
- Participating in studies

5.1.1 Providing Data

Project personnel are responsible for completing data forms that should have been designed for simplicity. At project initiation, the project characteristics (discussed in Chapter 4) are provided to establish a baseline. Throughout the life of the project, measures must be provided on a regular schedule, as agreed upon by the analysts and management. Possibly the most important data to be provided by the development team are the accurate final project statistics (see Figure A-7). These data are often overlooked in an immature measurement program.

The process for submitting completed forms must be equally simple. Developers and maintainers must be able to deliver forms to a specified, convenient location or hand them to a designated individual and then forget about them. A representative of the technical support component will be responsible for collecting the forms and initiating the data entry process.

Occasionally, developers and maintainers are asked to meet with the analysts. Although vitally important, these meetings must be brief and well planned so that they do not interfere with development and delivery schedules. Meetings may be feedback sessions for the purpose of verifying preliminary data analysis, interviews to gather additional project characteristics data or subjective information, or training sessions to reinforce the proper use of specific processes being applied by the developers.

[7] In most organizations, the managers of the development organization will continue to be responsible for collecting and applying certain data needed for ongoing program management activities without impact from the analysts. Some data collected in support of earned value analysis or planned versus actual budget information, for example, will continue to be collected and analyzed by managers and their project control support personnel. The role of the measurement analysts is to provide accurate models and relationships to support those management activities.

5.1.2 Participating in Studies

The analysts may ask the developers and maintainers to participate in the experimental use of some process, technique, tool, or model that is not part of the organization's standard practice. Such studies sometimes necessitate the use of new forms and typically require that development and maintenance personnel attend briefings or a training session on using the new process.

Most projects experience little, if any, process change driven by the analysts. For these projects, training is typically limited to discussions of new forms and new data reporting agreements. For projects that undergo significant process changes, however, training sessions are important to ensure that development and maintenance personnel thoroughly understand the new process and fully agree that the study supports the organizational goals. The study must be a cooperative team effort: analysts must provide regular feedback of interim results, and developers or maintainers must contribute their insight regarding the value and relevance of those results.

When development and maintenance personnel participate in such studies, they should always receive feedback from the analysts. At feedback sessions, developers and maintainers also have an opportunity to report their impressions of the degree of success derived from the innovation and to discuss any difficulties experienced in applying the new process.

5.2 Technical Support

The primary operational responsibilities of the technical support personnel are

- Collecting data
- Storing and quality assuring data
- Summarizing and reporting data

5.2.1 Collecting Data

Satisfactory collection of data by the technical support component depends on a clearly established interface with the development and maintenance component and on clearly defined terms and concepts provided by the analysis and packaging component.

Although many organizations put a great deal of effort into automating data collection, many years of experience have led to the following rule:

Do not expect to automate data collection.

Attempts to automate the data collection process should be limited. Because routine, manual data collection efforts add an overhead of only 1 to 2 percent (see Reference 18), automation may not result in a cost saving. In practice, extensive efforts to develop automated tools may actually increase cost to the total organization. It is more important to ensure that the amount of data is driven by specific organizational goals (which will also minimize the amount required) and that the data collection process is well defined and operationally smooth.

Regardless of the size of the automated data collection effort, it is essential that management communicate with the developers and maintainers about which parts of the process will be monitored electronically.

Interface With Data Providers

Technical support personnel must ensure that members of the management and technical staffs within the development and maintenance component understand their responsibilities with respect to furnishing the selected project measures. Technical support personnel must also communicate with the providers of the data to ensure that everyone understands the details of the collection requirements, for example,

- Which personnel are responsible for collecting and furnishing project measures
- How frequently the collection will occur
- Which portions of the software life cycle will be reflected in the data
- What type of personnel (management, technical, or administrative) will be included in level-of-effort measurements

Make providing data easy.

Personnel within the technical support component must make furnishing data as painless as possible for development and maintenance personnel to reduce the chances for aggravation and resentment on the part of those data providers. Publishing a list of technical support contacts can make it easy for the data providers to ask questions or deal with measurement problems. Making it obvious where to deposit the data forms and collecting them promptly to emphasize the importance of providing the forms on schedule are also useful tactics.

Definitions

To ensure that the data provided are based on a consistent understanding of the measurement terms and concepts, support personnel must supply concise, clear definitions to the development and maintenance personnel. It is the responsibility of the analysis and packaging component to write definitions that are consistent with organizational goals and locally understood ideas; however, the data collectors are responsible for furnishing the definitions to the data providers. The importance of focusing locally, rather than adhering to arbitrary industry-wide conventions, cannot be overemphasized.

5.2.2 Storing and Quality Assuring Data

The second important responsibility of the technical support component is storage of high-quality data. For project data to be used effectively in support of the goals of a measurement program, they must be complete and accurate as defined by QA procedures and readily available.

Data Storage

To be readily available, project data must be stored in an online database. This requirement leads to the next rule:

> ***Use commercially available tools.***

Using a COTS DBMS to support the organization's measurement program is highly recommended. The time and effort required to develop custom tools will outweigh their benefits. A relational DBMS will provide the most appropriate support for data retrieval and analysis using a variety of table combinations and user views. Spreadsheets, indexed sequential files, and even networked or hierarchical DBMSs are simply inadequate. See Reference 19 for a detailed description of a mature measurement database using a commercial DBMS.

Data Quality

The quality of the stored data must also be considered. From the perspective of the support component, data quality assurance is a two-step process:

1. *Verification of source data*—Discrepancies must be tracked to the source and corrected. This step includes checking that the
   a. Data forms have been submitted and are complete (i.e., all required values are provided).
   b. Values are of the specified type (e.g., numeric fields do not contain non-numeric values).
   c. Values are within specified ranges (e.g., the number of hours of effort per day per person is never greater than 24).
   d. Values are reported on the prescribed schedule.
2. *Verification of data in the database*—After the values have been entered into the database, a second check is performed to verify that the entries match the source value.

An organization with a mature measurement program may be able to use automated tools that allow developers to enter data directly into the database via online forms, thereby eliminating paper forms and the manual QA process. Although this approach may seem ideal, experience has shown that it often leads to unreliable data and that the cost of a manual process is relatively small.

Despite the quality assurance steps, the next rule still applies:

Expect measurement data to be flawed, inexact, and inconsistent.

The collection and verification processes are fallible, and some data will be incomplete and imperfect. In addition to the quality assurance activities performed by the technical support personnel, the analysts will subsequently have to determine the accuracy and usefulness of the data by cross-checking, back tracking, and general qualitative analysis.

5.2.3 Summarizing, Reporting, and Exporting Data

Technical support personnel are also responsible for producing and distributing reports and data summaries to data users in all three measurement program components. Occasionally, they are also responsible for exporting raw data to external organizations. Reports can be tabular or graphical, printed or displayed. Summary reports are designed to highlight particular trends or relationships.

Not all reports are generated by the support personnel, however. High-level data analysis reports, prepared by the analysis and packaging component, are discussed in the next section. Routine management reports of project control information remain the responsibility of management.

Many of the raw data and summary reports are generated on a regular schedule. These reports range from single-project summaries focused on a particular data type to multiple-project roll-ups that provide high-level statistics in a format compact enough to facilitate project-to-project comparisons. Support personnel distribute those reports to development and maintenance personnel to provide feedback on project measures. Analysis and packaging personnel also use the reports to identify projects and data to be used in studies and model generation.

Figure 5-2 provides an example of a regularly scheduled Project Summary Statistics report, showing actual data for projects in a NASA organization with a mature measurement program. The report also contains several questionable entries (e.g., 0.0 hours for support where there probably should be a positive value) and illustrates the rule that data may be flawed, inexact, or inconsistent.

The technical support component also generates some of the raw data and summary reports on an ad hoc basis, as requested by users of the data. Requests for specific data on specific projects come from both the development and maintenance component and the analysis and packaging component. Such reports also include low-level data dumps used by support personnel during the data verification process.

A related responsibility of the support component is preparing measurement data for export to another organization. Sharing data across domains and interpreting data out of context are normally not meaningful, as cautioned in the "focus locally" rule. Nevertheless, exporting data to another organization occasionally makes sense. For example, the organization may intend to use acquired data to support the establishment of its own measurement program. In addition to

Project Summary Statistics

9/13/93 07:23:39 Project Criteria : ALL

Project	Status	No. of Sub-systems	No. of Compo-nents	Total SLOC	New SLOC	Extensively Modified SLOC	Slightly Modified SLOC	Old SLOC	No. of Changes	Technical & Mgmt Hours	Support Services Hours
PROJECTA	INACTIVE	14	132	15500	11800	0	0	3700	2670	17715.0	1774.0
PROJECTB	INACTIVE	5	224	16000	14100	0	0	1900	213	5498.0	11.0
PROJECTC	INACTIVE	2	175	34902	34902	0	0	0	413	7965.3	0.0
PROJECTD	INACTIVE	2	415	41829	40201	450	1044	134	544	32083.4	4407.6
PROJECTE	INACTIVE	40	292	50911	45345	0	4673	893	1255	12588.0	1109.0
PROJECTF	INACTIVE	20	397	61178	49712	0	10364	1102	221	17039.0	3056.0
PROJECTG	INACTIVE	1	76	8547	8041	0	446	60	307	2285.0	0.0
PROJECTH	INACTIVE	11	494	81434	70951	0	0	10483	1776	17057.0	1875.0
PROJECTI	INACTIVE	11	267	72412	55289	1879	4184	11060	427	13214.6	1365.8
PROJECTJ	INACTIVE	14	930	178682	141084	16017	13647	7934	1494	49930.5	4312.9
PROJECTK	INACTIVE	4	322	36905	26986	0	7363	2556	412	12005.0	1524.5
PROJECTL	INACTIVE	6	244	52817	45825	1342	1156	4494	344	6106.3	0.0
*PROJECTM	INACTIVE	0	0	0	0	0	0	0	0	19208.9	3612.5
PROJECTN	ACT_DEV	0	0	0	0	0	0	0	0	59.0	0.0
PROJECTO	DISCONT	Incomplete data for this project									
PROJECTP	INACTIVE	11	278	26844	24367	0	2477	0	1177	10946.0	967.0
PROJECTQ	ACT_DEV	0	0	0	0	0	0	0	0	24662.2	3739.2
PROJECTR	INACTIVE	34	392	25731	25510	0	0	221	124	1514.0	0.0
PROJECTS	ACT_DEV	0	0	0	0	0	0	0	0	0.0	0.0

* Project data are not final

Figure 5-2. Project Summary Statistics

issuing a caveat about the danger of misinterpretation, support personnel must sanitize the data before export to preserve the confidentiality of the data providers. Sanitizing the data requires eliminating names of individuals and substituting generic project names for the mnemonics used to identify projects within the local environment.

5.3 Analysis and Packaging

Analysis and packaging responsibilities consist of

- Designing studies
- Analyzing project data
- Packaging results

The analysis and packaging component has the heaviest burden within the measurement program. The analysts must first design measurement studies to collect and analyze project data in support of the organization's process improvement goals. Next, they must use the data to develop and maintain organizational models, such as cost estimation models and error profiles, and to

determine the impact of new technologies, such as object-oriented design or code reading, on the organization. Finally, they must provide the derived information to the project organization in a useful form, such as guidebooks, tools, and training courses. The analysis and packaging effort should always be transparent to the development and maintenance projects providing the data. Developers have a right to understand why they are providing the data. Moreover, a clear understanding of the connection between the data they provide and the models and guidelines produced by the analysts leads to higher quality project data and a higher degree of confidence in the resulting products.

By analyzing and packaging measurement data, these personnel support the three reasons for establishing a measurement program:

1. *Understanding*—Analysts use routine data from the core measures to build models and relationships and to characterize the overall software processes and products.
2. *Managing*—Although the analysts do not play an active role in managing the software development and maintenance projects, they provide information and models to the development and maintenance personnel to improve the quality of project management.
3. *Guiding improvement*—Each project provides the analysts an opportunity to study the effect of a change and learn something from it. The goals for collecting specific measures are clearly defined in process study plans. These studies can range in scope from straightforward validation of the current organizational models to controlled investigations of the impact of introducing a new methodology. Data from projects with similar goals are analyzed and synthesized to produce models and to understand the impact of process changes. Beneficial new technologies and organizational process and product models are then packaged for use by the projects.

5.3.1 Designing Process Improvement Studies

On the basis of the overall goals of the organization and the characteristics of the individual projects, the analysts, working with the project leaders, prepare plans that define specific study goals and specify the data to be collected. Figure 5-3 provides an outline of a process study plan. In some cases, analysts prepare detailed plans for projects participating in the measurement program. In most cases, however, no significant changes will be proposed, and the study goals will be primarily to refine the understanding of the software process or product; routine measurement data will be sufficient, and no training will be needed. Many of the study plans will, therefore, be relatively brief, containing simple descriptions of the data to be collected, the analysis to be performed, and the study goals (e.g., "gain insight into the classes and origins of software errors").

Analysts must also prepare higher level organizational plans to coordinate the studies across projects and to ensure that all high-priority organizational goals are being addressed. They work closely with the organization's managers to choose appropriate projects for major studies.

Appendix B includes a sample process study plan. The plan summarizes key characteristics of the project, specifies study goals, identifies key questions to be answered by analyzing project data and information, and clearly defines the data to be provided by the project.

Process Study Plan for

{Project Name**}**

{Plan Originator Name**}**

{Date**}**

1. Project Description

Briefly describe the application and the project team.

2. Key Facts

Briefly state the life-cycle methodology, methods, schedule, project size, implementation language, and any other important details.

3. Goals of the Study

Explain the goals of this study.

4. Approach

Describe the steps planned to accomplish the goals.

5. Data Collection

Itemize the measurement data and information to be collected during the study.

Figure 5-3. Process Study Plan Outline

A key reason for a study is to assess and guide change. Any change, such as introducing a new method, tool, or language, may involve an element of risk, so any significant change to a standard development or maintenance process must be jointly approved by the analysts and the project manager. When asked by the analysts to introduce evolving technologies on a project, a manager must consider the risk, use common sense, be cautious, and even refuse the change if the risk is too great. Nevertheless, process studies are important to every organization, and each development or maintenance project is expected to add some amount of process information to the organization's experience base.

Just as the organization's high-level measurement plans must relate to its overall goals, a process study plan for a project (or for a related set of projects) must show a clear connection between the data being collected and the goals of the study. The sample plan in Appendix B was developed for an ongoing project within an organization that already had developed a high-level plan. It includes a high-level description of the approach for analyzing the project information and defines a study intended to support new organizational goals.

5.3.2 Analyzing Project Data

The analysts continually synthesize data from many projects to gain an understanding of both the product and process characteristics of the organization. They look for distinguishing project characteristics that identify subgroups within the organization—for example, all projects using the Ada language or all projects applying object-oriented requirements analysis and design methods. That effort results in a baseline set of process and product models for the organization and may reveal changes (to models and relationships) that are not the result of explicitly introducing new processes. Baseline analysis is a major effort, and it is a critical prerequisite for any analysis or

packaging of the results of individual project studies. Experience has shown that the baseline characteristics change slowly, even with the infusion of new processes. Therefore, packagers generate new handbooks and guidebooks only every 3 to 5 years.

Analysts also examine individual project data to determine how trends correlate with project successes and difficulties. They design the content of the high-level analysis reports and work with technical support personnel to establish the frequency for producing and distributing reports. These reports generally provide high-level summaries of project characteristics or support specific study objectives. Figure 5-4 provides an example of a high-level development project summary report. Figure 5-5 shows a similar report for a maintenance project.

Once the organization's processes and products have been characterized, the analysts shift their focus to assessing the impact of change. They compare current project measures with the organization's historical models to measure the impact of evolutionary changes introduced by either explicit changes to the software processes (such as a new method or tool) or external influences (such as changing the problem complexity). This analysis results in updated process, product, or management models.

Analysts also focus on determining the impact of new technologies and approaches introduced in major experiments. They compare experimental data with the historical baseline models to assess success or failure. Often the result indicates a guarded success, suggesting that continued study is needed to refine the technique and confirm success.

Figure 5-6 shows the results of a study designed to determine the impact on the distribution of effort across software activities and life-cycle phases when the Ada programming language was introduced into an organization that had previously relied on FORTRAN. After the organization had gained the experience of using Ada on nine projects, the models stabilized as shown in the figure (see Reference 10).

Although analysts use objective measurement extensively, they also depend heavily on subjective information gathered directly from project personnel and recorded in project history reports to help interpret the data. Each project has a unique set of drivers and circumstances that must be considered when interpreting the data. Chapter 6 addresses the analysis, application, and feedback of measurement information in more detail.

5.3.3 Packaging the Results

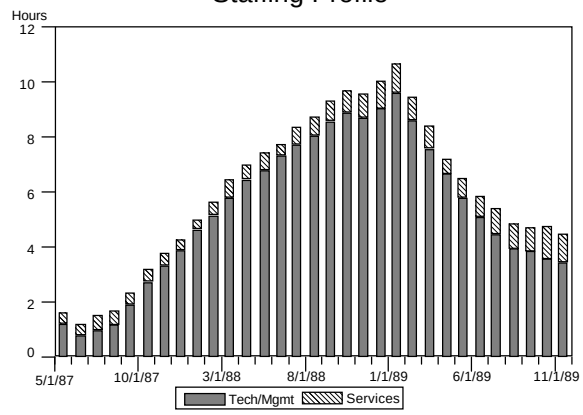
As analysts gain greater insight into the characteristics of the current software development and maintenance environment and the impacts of specific software methodologies on that environment, they must infuse that understanding back into the development organization, packaging the appropriate software practices for the problem domain in well-founded standards and policies so that they can be applied on ensuing projects. Packaging entails generating the following items:

- Software management policies and guidelines
- Software development and maintenance standards

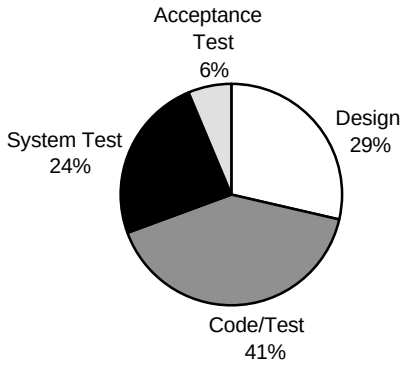
Data Summary for Project X

CHARACTERISTICS		PHASE DATES		LINES OF CODE		DERIVED MEASURES	
Project name: X		Requirements:	no date	Developed:	73,047	Productivity (SLOC/hour):	4.283
Primary language: Ada		Design:	10/26/87	Delivered:	81,434	Productivity (DLOC/hour):	4.774
Current phase: Inactive		Implementation:	01/27/88	New:	70,951	Reliability (errors/KDLOC):	4.642
Development computer: VAX		System test:	01/05/89	Reused:	10,483	Change rate (changes/KDLOC):	8.277
Components: 494		Acceptance test:	10/03/89	Modified:	0	CPU run rate (runs/KDLOC):	218.4
Changes: 674		Maintenance:	12/15/89			CPU use rate (CPU hours/KDLOC):	0.768
Errors: 378							
Total effort: 17,057 hours							

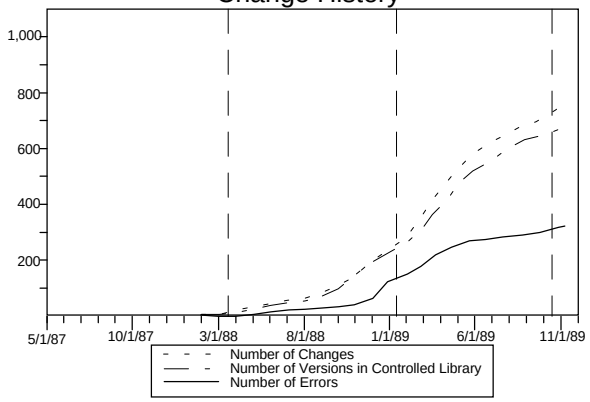
Staffing Profile



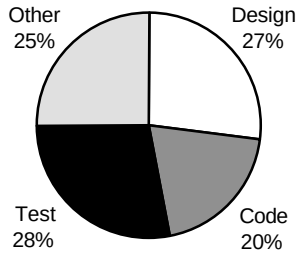
Effort by Calendar Phase



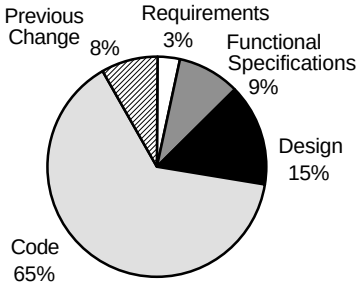
Change History



Effort by Activity



Error Source



Error Class

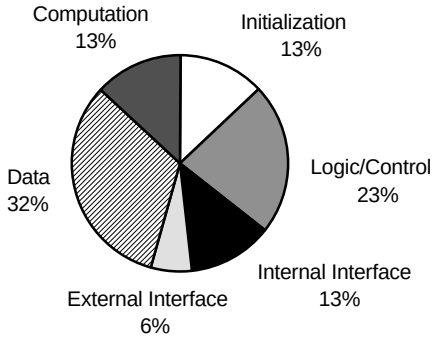


Figure 5-4. High-Level Development Project Summary Report

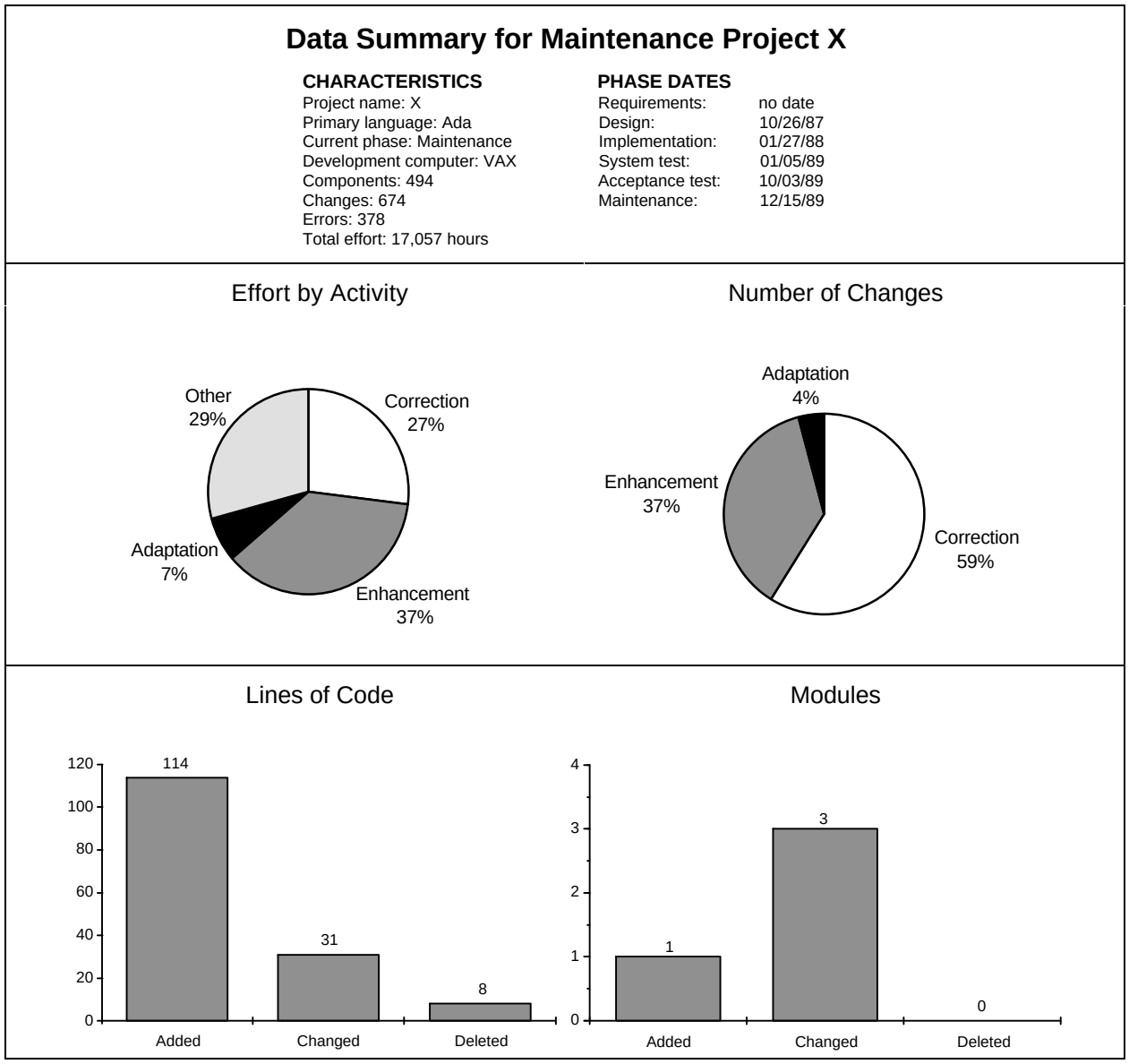


Figure 5-5. High-Level Maintenance Project Summary Report

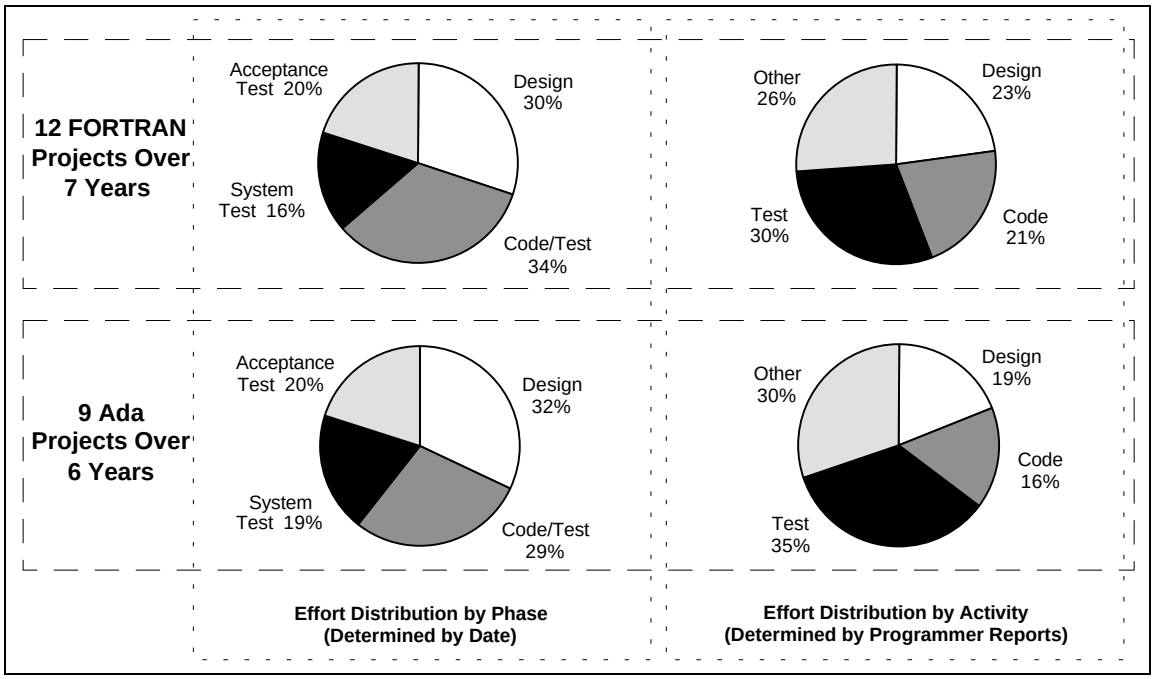


Figure 5-6. Impact of Ada on Effort Distribution

- Software training
- Tools and automated aids
- Reports of process studies
- Updates of packaged materials

Software Management Policies and Guidelines

Much of the information that has been collected and synthesized by the analysis component is fed back into the organization in the form of models, planning aids, and guidelines. When packaged into well-designed policies and guidebooks, this information can improve a manager's ability to plan a software project, monitor its progress, and ensure the quality of its products.

Management policies and guidelines provide the local scheduling, staffing, and cost estimation models that are needed for initial project planning as well as for re-estimation during the life of the project. NASA's *Manager's Handbook for Software Development* (Reference 10) contains guidelines and examples for using numerous models, such as

- Relationships relating effort to system size
- Effort and schedule distributions by phase
- Staffing profiles
- Productivity relationships

The key models used for gauging project progress and quality are organized and packaged together, preferably with the planning models, in a single reference source. Typical progress models include local profiles of software growth, computer use, and test completion. Quality models include error rates, reported and corrected software discrepancies, and software change rates. Figure 5-7 shows an example of an error rate model used to predict and track errors throughout the life cycle. The model was calibrated by measuring the error characteristics of over 25 projects with more than 5,000 errors reported. It depicts the typical rate of finding errors in code (four errors per KSLOC), during the system test phase (two errors per KSLOC), and during acceptance testing (one error per KSLOC), a reduction of 50 percent in each subsequent phase. Because no data were collected during the design phase, the error rate is zero. The variation was also computed, as shown in the figure. An actual error rate above the bounds of the model may be the result of misinterpreted requirements or may be caused by highly unreliable or complex software. An actual rate below the bounds may be the result of particularly well-built software, a relatively simple problem, or inadequate testing.

Every organization can and should produce a document containing the complete set of models, relationships, and management guidelines used within the organization. (See Reference 8 for an example of such a document.)

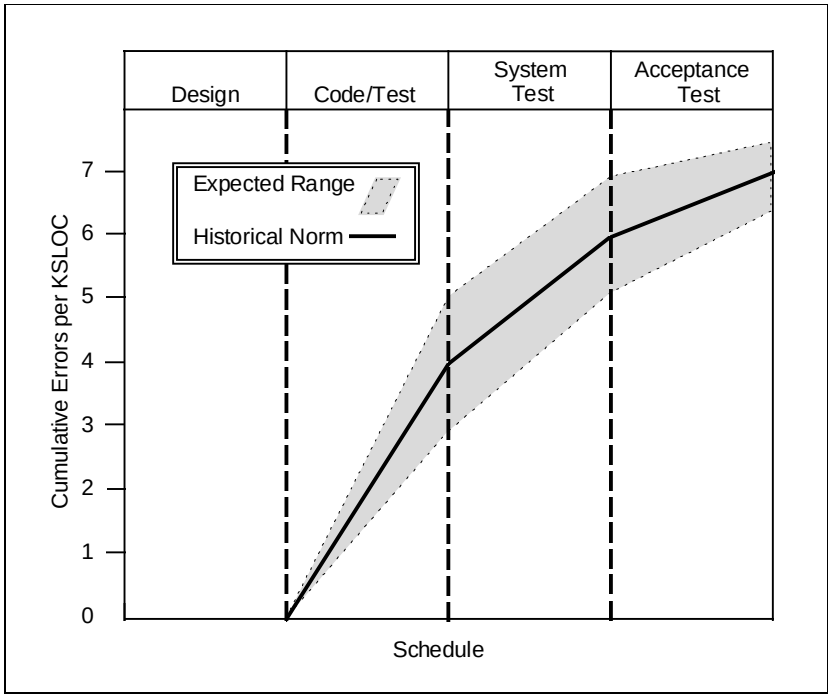


Figure 5-7. Sample Error Rate Model

Software Development and Maintenance Standards

In a mature measurement program, standards for software development and maintenance address each phase of the software life cycle, covering the entire range of technical activities. These standards define the products, methods, tools, data collection procedures, and certification criteria

that have been identified as beneficial to the organization. Separate, detailed standards characterize programming practices unique to the local environment or to a specific development language; they also address specialized techniques, such as the Cleanroom method or object-oriented design.

The most useful, high-quality software engineering standards are derived from the practices of the organization for which they are intended; that is, they are measurement driven. A standard requiring the use of processes that are incompatible with the organization's development and maintenance methodology cannot be successful.

Software Training

The organization's goals, environment, and measured experiences must drive the planning and execution of the training curriculum. Courses reflect the understanding of the characteristics of the local environment, and each course must respond to a specific need.

Training becomes essential when new technologies, standards, tools, or processes are infused into the software engineering environment. Personnel are more likely to accept a new approach when it has been introduced in well-organized stages within the interactive setting of a training course.

Training must be provided first to those who are participating in an experiment with a new technology and then to a wider audience as soon as the technology has been adopted for general use within the organization. A training program should also include courses that introduce new personnel to the software development and maintenance environment.

Tools and Automated Aids

Packaging personnel also build tools and other automated aids to facilitate software management, development, maintenance, or data collection processes. Such tools include

- Cost estimation aids based on local models
- Management aids that compare actual measured values with baseline estimates
- Design aids that are driven by experimental results indicating beneficial design approaches

In addition, more sophisticated tools may use the organization's extensive historical information for managing and for analysis. An example of such a tool is the Software Management Environment (SME) (Reference 20). It encapsulates experience (i.e., data, research results, and management knowledge) gained from past development projects in a practical tool designed to assist current software development managers in their day-to-day management and planning activities. The SME provides integrated graphical features that enable a manager to predict characteristics such as milestones, cost, and reliability; track software project parameters; compare the values of the parameters to past projects; analyze the differences between current and expected development patterns within the environment; and assess the overall quality of the project's development progress. Figure 5-8 illustrates the architecture and typical uses of such a tool.

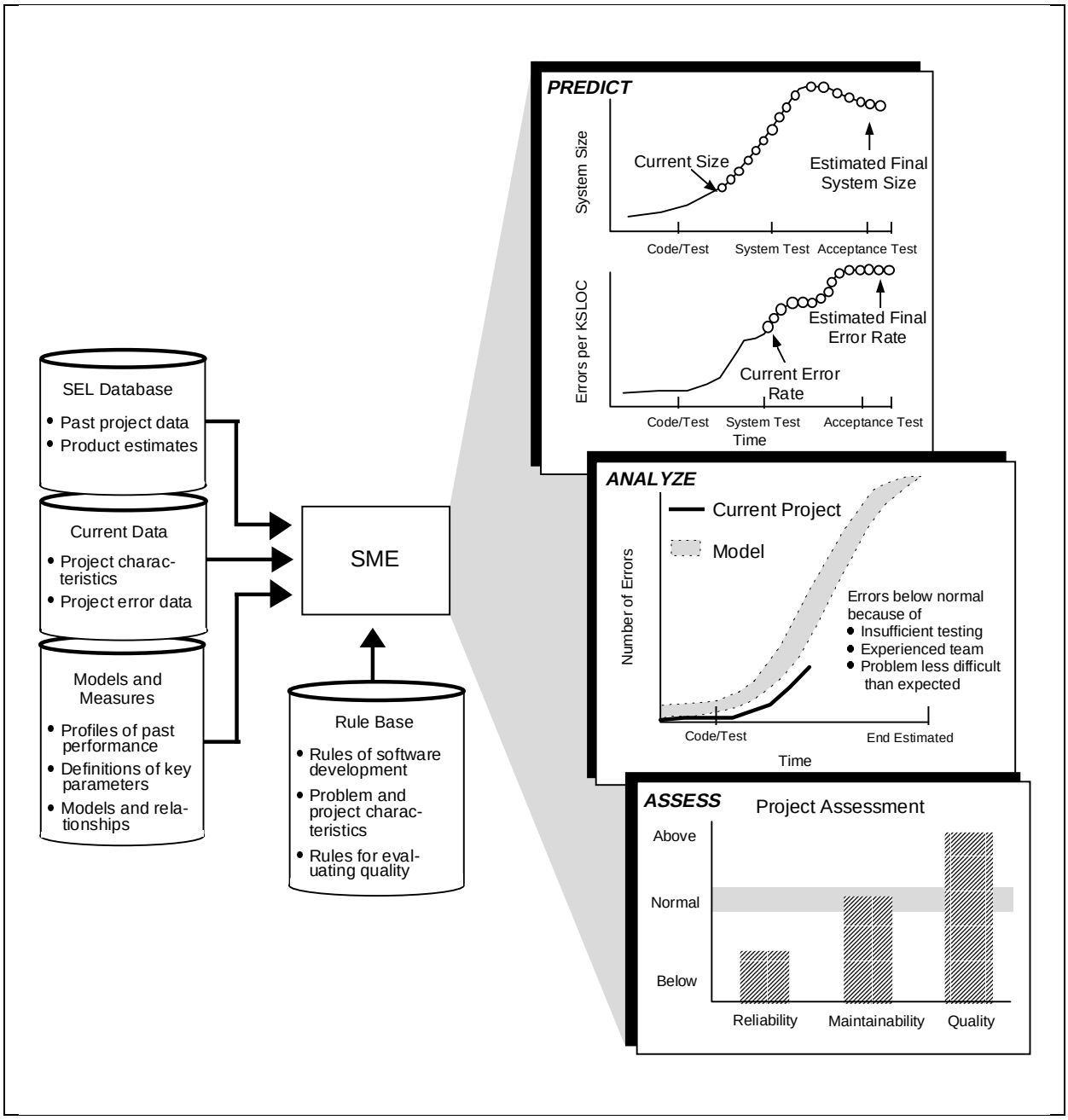


Figure 5-8. SME Architecture and Use

Reports of Process Studies

For each process study, analysts prepare one or more reports that address the goal, the methods employed, the results measured, and the conclusions drawn. Interim reports document partial results during lengthy or ongoing studies, and final reports are prepared immediately after the study is completed.

Final reports are vital sources of information when the time comes to integrate study recommendations with other standard practices before packaging them as policies, guidebooks, courses, or tools. An organization may, therefore, find it helpful to collect all study reports produced within a year into a single annual reference volume.

Some organizations repackage study reports for distribution outside the local environment as conference papers and presentations, thus gaining valuable feedback by subjecting the results to peer review. Such scrutiny can offer comparisons, suggest other interpretations or conclusions, and help improve the data collection procedures and analytical methods employed by the organization.

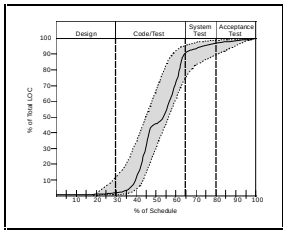
Updates of Packaged Materials

All packaged materials—policies, standards, course materials, tools, and study reports—must be maintained in an organizational repository. Together with the information in the measurement database, the repository of packaged materials functions as the memory of the organization. It is essential that the contents of the library be catalogued and that the catalog be kept up-to-date as new material is added. In the SEL, for example, a bibliography containing abstracts of all SEL documents is revised and republished annually.

The analysis and packaging component also updates guidebooks, training courses, policies, and tools on a regular basis to keep the organization abreast of current software engineering practices.

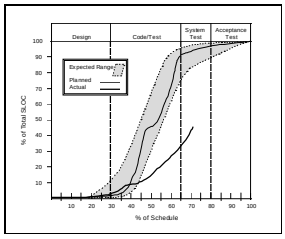
Chapter 6. Analysis, Application, and Feedback

Chapter Highlights



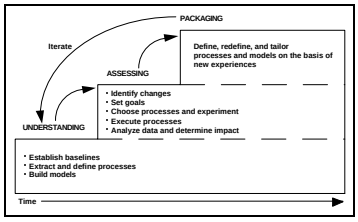
UNDERSTANDING

- Software attributes
- Cost characteristics
- Error characteristics
- Project dynamics



MANAGING

- Planning
- Assessing progress
- Evaluating processes



GUIDING IMPROVEMENT

- Measuring process impact
- Adapting process to local environment
- Eliminating processes with little value

This chapter describes specific approaches for using measurement information effectively. Software measurement programs must focus on the use of data rather than on their collection. Thus, the approach to using measurement data must be clearly defined, and the data must be analyzed and packaged in a suitable form. The effective use of measurement data is an outgrowth of the planning exercise that establishes the organization's goals, which drive the measurement activities.

The following sections address the analysis, application, and feedback of measurement information in the context of the three key reasons for establishing a measurement program, as discussed in Chapter 2:

1. Understanding
2. Managing
3. Guiding improvement

Examples drawn from experiences within NASA illustrate the important points. Because each organization's measurement goals may differ, the examples presented here may not relate directly to the needs of other organizations.

6.1 Understanding

The first reason for measurement—*understanding*—includes generating models of software engineering processes and the relationships among the process parameters. As an organization builds more models and relationships and refines them to improve their accuracy and reliability, its personnel develop more insight into the characteristics of the software processes and products.

True understanding requires qualitative analysis of objective and subjective measurement information, including examination for accuracy and checks for flawed, missing, or inconsistent data values. If used properly, subjective information is as valuable as objective counts. Unlike objective data, which are used in statistical analysis, subjective information reflects the experience of managers and developers within the organization's local environment. The resulting models and relationships, whether derived from objective or subjective information, are relevant only within the local environment.

The understanding process includes the following major measurement applications:

- Software attributes
- Cost characteristics
- Error characteristics
- Project dynamics

Increased understanding provides the foundation for building models and relationships and for developing the key information required for managing subsequent software development efforts.

The examples in this section depict various measurement applications that have proven beneficial to experienced measurement organizations. All of the models can be developed from the core measures described in Chapter 4. The example descriptions are by no means exhaustive. Finding the answers to the questions posed in Table 6-1 is an essential activity in applying measurement.

Basili's Goal/Question/Metric paradigm (References 23 and 24) provides the framework to relate the questions in Table 6-1 (and Table 6-5) to the goals and measures addressed in the examples that appear throughout the rest of the chapter. Any software organization will benefit from analyzing the fundamental information shown in these examples.

Table 6-1. Questions Leading to Understanding

Measurement Application	Understanding	Examples
Software Attributes	What languages are used, and how is the use evolving?	1
	What are the system sizes, reuse levels, and module profiles?	2
Cost Characteristics	What is the typical cost to develop my software?	3
	What percentages of my software resources are consumed in the various life-cycle phases and activities?	4
	How much is spent on maintenance, QA, CM, management, and documentation?	5
Error Characteristics	What are the error rates during development and maintenance?	6
	What types of errors are most prevalent?	7
	How do size and complexity affect error rates?	8
Project Dynamics	What is the expected rate of requirements changes during development?	9
	How fast does code grow during development, and how fast does it change?	

6.1.1 Software Attributes

Information about software attributes is easy to record and use but is too often overlooked. At a minimum, organizations should record the sizes, dates, and languages used on every project. Those basic characteristics are necessary for developing cost models, planning aids, and general management principles. Table 6-2 shows a subset of the actual data used in calculating the information shown in the examples that follow. For a more complete listing of the data, see Reference 9.

Example 1:
Language Evolution

Goal:	***Determine the language usage trend.***
Measures needed:	***Project dates, sizes, and languages.***
	(See Sections 4.3 and 4.5.)

Table 6-2. Software Attribute Data

Project	Language	Development Period	New SLOC	Reused SLOC	Effort (Hours)
ISEEB	FORTRAN	10/76–09/77	43,955	11,282	15,262
SEASAT	FORTRAN	04/77–04/78	49,316	26,077	14,508
DEA	FORTRAN	09/79–06/81	45,004	22,321	19,475
ERBS	FORTRAN	05/82–04/84	137,739	21,402	49,476
GROAGSS	FORTRAN	08/85–03/89	204,151	32,242	54,755
GROSIM	FORTRAN	08/85–08/87	31,775	7,175	1,146
COBSIM	FORTRAN	01/86–08/87	47,167	5,650	49,931
GOADA	Ada	06/87–04/90	122,303	48,799	28,056
GOFOR	FORTRAN	06/87–09/89	25,042	12,001	12,804
GOESAGGS	FORTRAN	08/87–11/89	113,211	15,648	37,806
GOESIM	Ada	09/87–07/89	65,567	26,528	13,658
UARSAGSS	FORTRAN	11/87–09/90	269,722	33,404	89,514
ACME	FORTRAN	01/88–09/90	34,902	0	7,965
UARSTELS	Ada	02/88–12/89	44,441	23,707	11,526
EUVEAGSS	FORTRAN	10/88–09/90	55,149	193,860	21,658
EUVETELS	Ada	10/88–05/90	2,532	64,164	4,727
EUVEDSIM	Ada	10/88–09/90	57,107	126,910	20,775
SAMPEXTS	Ada	03/90–03/91	3,301	58,146	2,516
SAMPEX	FORTRAN	03/90–11/91	12,221	142,288	4,598
SAMPEXTP	FORTRAN	03/90–11/91	17,819	1,813	6,772
POWITS	Ada	03/90–05/92	20,954	47,153	11,695
TOMSTELS	Ada	04/92–09/93	1,768	50,527	6,915
FASTELS	Ada	08/92–10/93	5,306	59,417	7,874
FASTAGSS	FORTRAN	08/92–04/94	21,750	125,405	7,550
TOMSEP	FORTRAN	05/93–04/94	24,000	180,300	12,850

Language Usage Trend

Data recorded at NASA to track language usage on projects have provided insight into the trends within the organization and have led to better planning for programmer training. Figure 6-1 compares the language usage on projects completed before 1992 (and currently in maintenance) with those in development after 1992 (see Reference 21).[8]

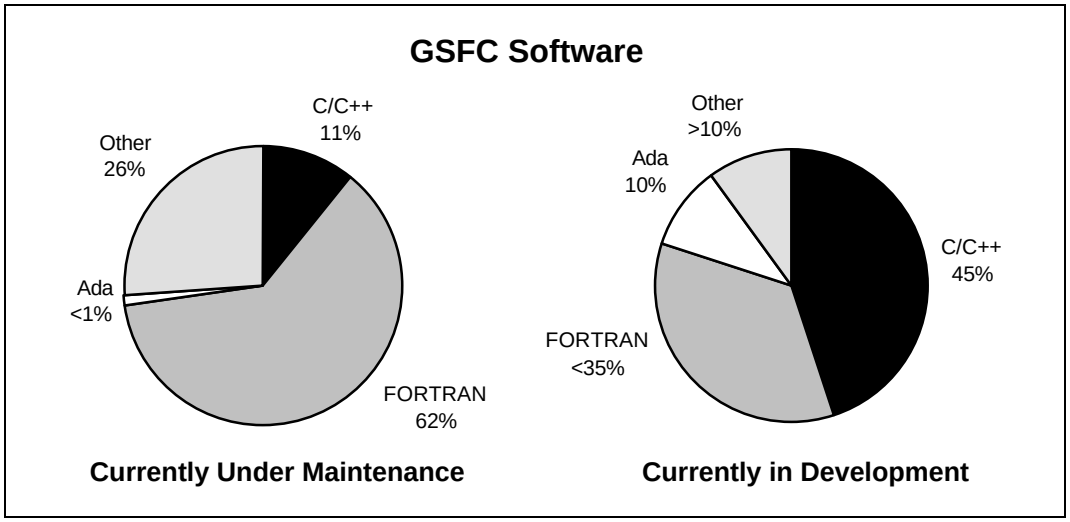


Figure 6-1. Language Usage Trend

Example 2:
Product Profiles

Goal:	***Determine the levels and trends of code reuse in projects.***
Measures needed:	***Project dates, sizes, and percentages of reuse.***
	Total effort on each project.
	(See Section 4.5.)

The characteristics of the source code itself can provide useful information about software projects. Too often this basic information, which is required to develop effective cost and planning models, is neither archived nor used effectively. Relatively simple historical models can be useful for managing and guiding improvements on projects. The information includes the typical size of projects and components; profiles of source code distributions among commentary, data definitions, and executable code; and resultant code reuse models.

[8] The percentages shown in the figure are derived from data collected from over 75 projects covering a span of 10 years. Table 6-2 represents only a small sample of those data.

Code Reuse Trend

Figure 6-2 shows trends derived from 11 FORTRAN and 8 Ada projects. The models were initially produced in 1989 for the early projects; more recent projects reflect a significantly higher percentage of reuse.

The basic source code information is needed not only for tracking changes in the code reuse level over time but, more importantly, for determining essential cost models for the local environment. The following section discusses how to derive cost-impact models of reuse.

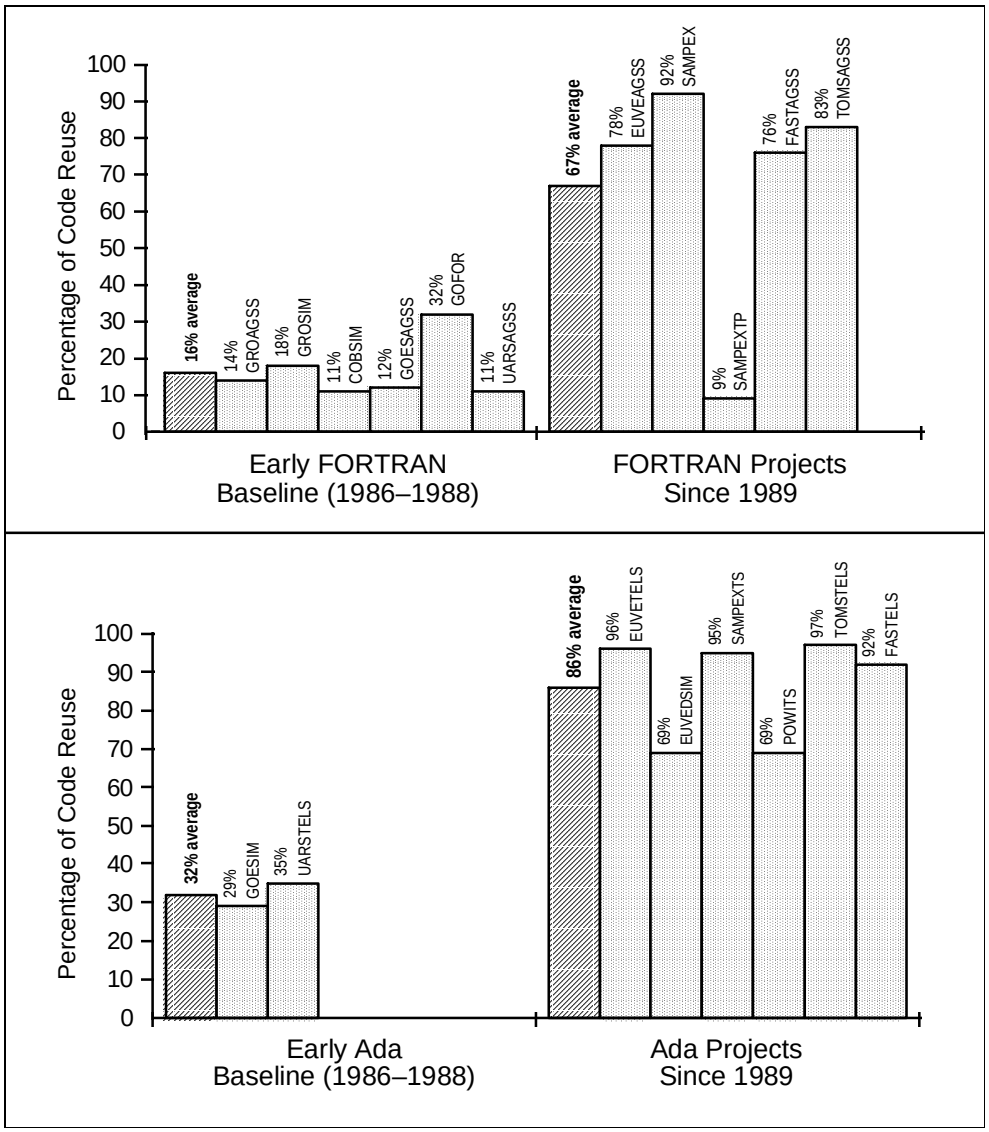


Figure 6-2. Code Reuse Trend

6.1.2 Cost Characteristics

Software cost characteristics are probably the most important set of attributes that contribute to an understanding of software. Cost characteristics include productivity, cost of phases, cost of activities, cost of changes, and many other attributes required for managing, planning, and monitoring software development and maintenance.

Example 3:
Cost Versus Size

Goals:	***Evaluate the cost of reusing code.*** ***Determine the cost of producing code in the organization.***
Measures needed:	***Project size, dates, reuse, and effort data.*** ***(See Section 4.5.)***

Cost of Reusing Code

Simple measures can be used to derive a local model for the cost of producing software. One major factor that must be analyzed is the impact of code reuse on cost. Borrowing code written for an earlier software project and adapting it for the current project usually requires less effort than writing entirely new code. Testing reused code also typically requires less effort, because most software errors in reused code have already been eliminated. Software projects using a significant amount of reused code usually require less overall effort than do projects with all code written from scratch.

Chapter 2 introduced the following relationship among the values of effort (cost of personnel), DLOC, and productivity:

$$\text{Effort (in hours)} = \text{DLOC} / \text{Productivity}$$

where

$$\text{DLOC} = \text{New SLOC} + \text{Reuse Cost Factor} \times \text{Reused SLOC}$$

The *reuse cost factor* is a weighting factor applied to reused source code. Several simplifying assumptions can be made to compute an approximate value for this factor. The most significant assumption is that all similar projects reflect approximately the same productivity; hence, the only variable is the cost of reuse. In this case, the similarity of the projects comes from their having been developed within the same environment and in the same language (FORTRAN). Although numerous other factors affect the cost of development, it is best to apply simple measures to arrive at an approximation before attempting detailed analysis of more complex factors.

Points derived from values in Table 6-2 can be plotted to illustrate the relationship between lines of code per hour and the reuse percentage as shown in Figure 6-3. Assuming that productivity (DLOC/Effort) is constant, the straight line fit to the DLOC points indicates that 20 percent is a reasonable approximation for the reuse cost factor for FORTRAN.

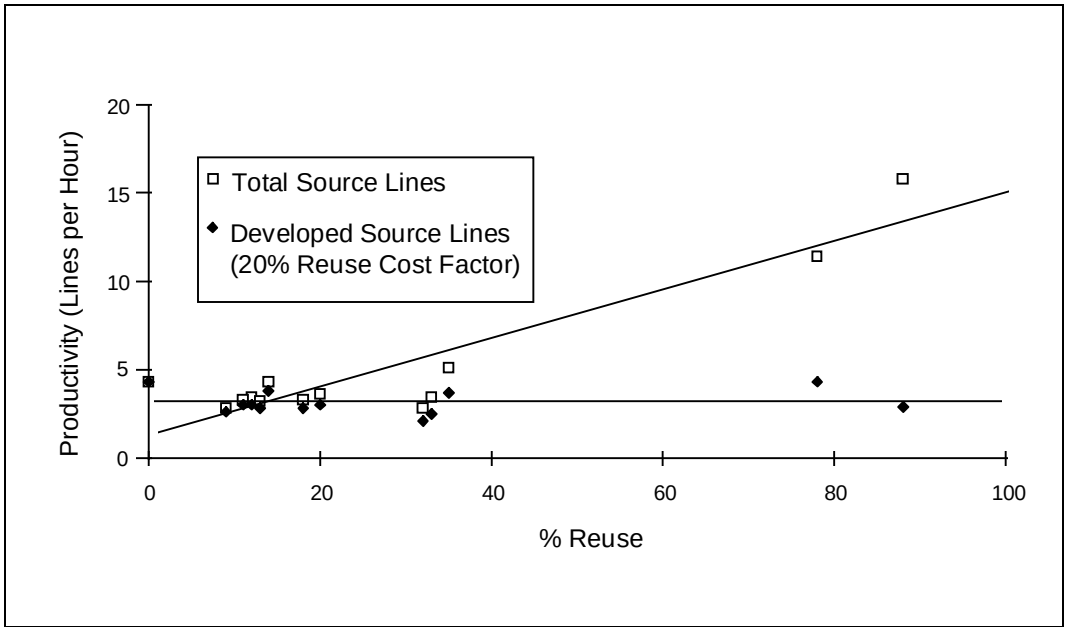


Figure 6-3. Derivation of 20 Percent Reuse Cost Factor for FORTRAN

Figure 6-4 shows a slightly different approach for Ada language projects. Analysts within the same environment studied size, effort, and reuse data from five projects developed between 1987 and 1990 to derive the Ada reuse cost factor. Attempting to produce a constant productivity value, they computed the productivity as DLOC per hour for each of the five projects while varying the reuse cost factors. In this case, the 30 percent factor resulted in the lowest standard deviation for the computed productivity values and was adopted for this organization.

Every organization can develop its own reuse cost factor with the simple measures listed in Table 6-2.

Cost of Producing Code

One of the most basic aspects of software engineering understanding is the ability to model the cost of a system on the basis of size or functionality. Section 2.2 discussed the basic estimation models, relating cost to software size, which have proven useful in one environment. Those models were derived by analyzing data from over 100 projects spanning many years and by making careful decisions about which projects to include in the baseline model and which to exclude. Organizations just starting to apply measurement should begin to establish cost models with their own data.

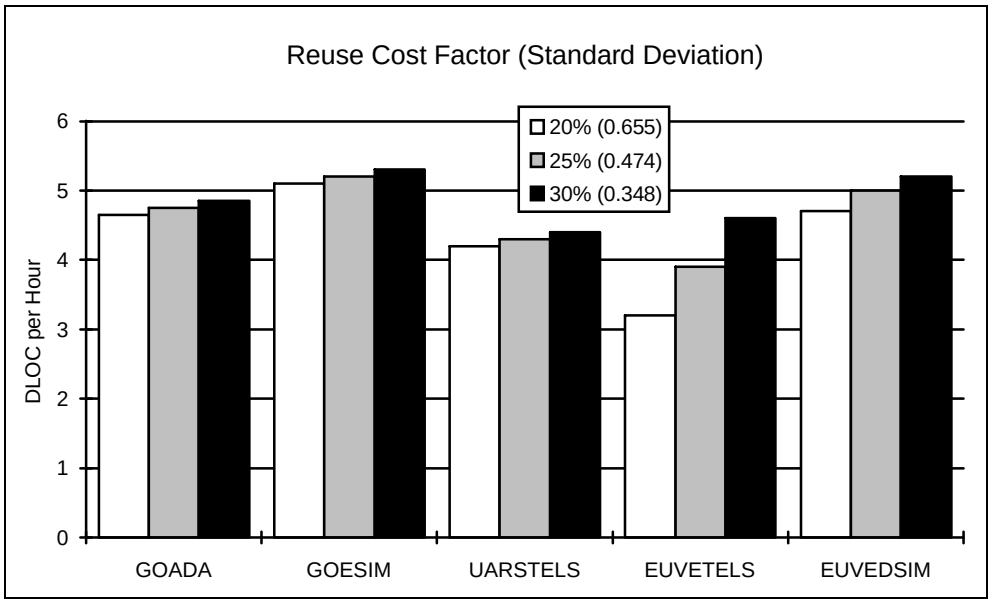


Figure 6-4. Derivation of 30 Percent Reuse Cost Factor for Ada

Example 4:
Effort Distribution

Goals:	***Determine the relative cost of each life-cycle phase.***
	Determine the characteristics of staffing profiles.
Measures needed:	***Project phase dates, effort data, and developer activity data.***
	(See Sections 4.1 and 4.5.)

Cost of Life-Cycle Phases

An effort distribution can be modeled in two ways:

1. By phase, to determine which phases of the life cycle consume what portion of the total effort
2. By activity, to determine what portion of effort is spent performing each defined software engineering activity

Figure 6-5 shows those two distributions of effort for the same set of development projects. The model of effort by life-cycle phase represents hours charged to a particular project during each phase as determined by the beginning and ending dates of the phases. The model of effort by activity represents all hours attributed to a particular activity, regardless of when in the life cycle it occurred. The four activities (design, code, test, and other) are determined by local process definitions. The "other" category includes

supporting efforts such as managing, training, attending meetings, and preparing documentation.

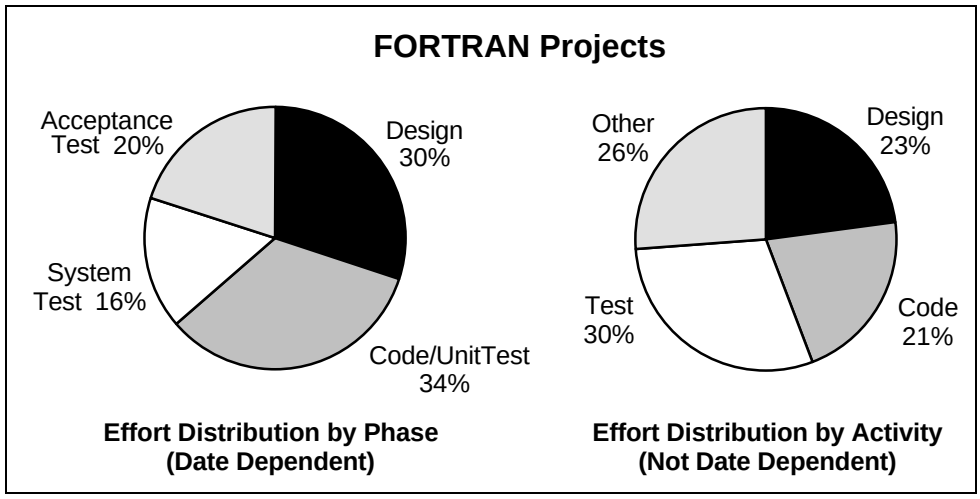


Figure 6-5. Effort Distribution Model

Staffing Profiles

Another use of effort data is to model the baseline staffing profile that reflects the development environment and the type of problem. In the SEL environment, where a substantial portion of the detailed requirements is not known until mid-implementation, the expected model resembles a doubly convex curve instead of the traditional, widely used Rayleigh curve (see Figure 6-6). The cause of this trend is not well understood, but it occurs repeatedly on flight dynamics projects in that environment. It is valuable for each software organization to produce its own staffing profile rather than to rely on a generic model that may have no relevance to the actual processes used at the local level.

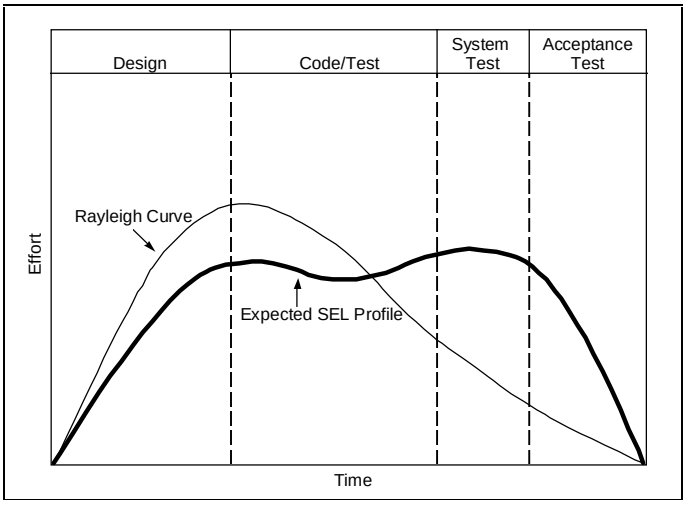


Figure 6-6. Staffing Profile Model

Example 5:
Cost of Major Activities

Goal:	***Build models of the cost of maintenance and other major activities, such as documentation and quality assurance.***
Measures needed:	***Developer activity data, effort, and software size.***
	(See Sections 4.1 and 4.5.)

Cost of Maintenance

Software maintenance includes three types of activities occurring after the system is delivered:

1. Correcting defects found during operational use
2. Making enhancements that improve or increase functionality
3. Adapting the software to changes in the operational environment, such as a new operating system or compiler

The SEL environment has two major types of systems under maintenance: *multiple-mission* systems, which support many spacecraft and have a software lifetime of from 10 to 30 years, and *single-mission* support systems, which run as long as the spacecraft are operational, typically from 2 to 7 years. Both types of systems are written primarily in FORTRAN on mainframes and are roughly the same magnitude in size (100–250 KSLOC). A large percentage of the maintenance effort is spent enhancing the system by modifying and recertifying existing components. SEL maintenance personnel add few new components and produce little new documentation. Average annual maintenance cost ranges from 1 to 23 percent of the total development cost of the original system. Table 6-3 includes analysis of representative data from several SEL systems under maintenance for at least 3 years. Some of the values are not available and some are questionable; nevertheless, analysis provides useful insights into the cost of maintenance.

On the basis of the above analysis, and in consideration of the high variation among systems, the SEL uses the conservative approach shown in Table 6-4 when estimating maintenance costs.

A general model of the overall cost of the development and maintenance of software can be of significant value for identifying more detailed breakdowns of cost by key activities. The data from projects depicted in Table 6-2 are used to determine the cost of several key activities.

Table 6-3. Analysis of Maintenance Effort Data

System	Type	Size (SLOC)	Development Effort (Hours)	Yearly Maintenance Effort History (Hours)				% Effort per Year
				1st	2nd	3rd	Average	
COBEAGSS	S	178,682	49,931	57	0	0	19	0.04
GROAGSS	S	236,393	54,755	496	370	370	412	1
GOESAGSS	S	128,859	13,658	607	159	950	572	4
EUVEAGSS	S	249,009	21,658	757	358	410	508	2
DCDR	M	75,894	28,419	n/a	4,000	4,000	4,000	5
ADG	M	113,455	45,890	n/a	6,000	6,000	6,000	13
CFE	M	98,021	30,452	n/a	2,000	2,000	2,000	2

NOTE: S = single mission system.
M = multiple mission system.

Table 6-4. Basis of Maintenance Costs Estimates

Project Type	Estimated Annual Maintenance Cost as a Percentage of Total System Development Cost
Single-mission systems	5%
Multiple-mission systems	15%

Costs of Documentation, Quality Assurance, and Configuration Management

The costs of support activities such as documentation, QA, and CM are determined from the development activity measures combined with the basic time reporting from the support organizations. These data are easy to collect in most software organizations. Figure 6-7 shows the data collected from one large NASA organization. A basic understanding of the cost of these activities is essential so that any change or attempt to plan for these efforts can be based on a solid foundation.

6.1.3 Error Characteristics

Understanding the characteristics of errors in the software products is just as important as understanding the cost of producing and maintaining software. The nature of software errors includes the error frequency, the cost of locating and removing errors, the severity of the errors, the most common causes of errors, and the processes most effective in identifying or preventing errors.

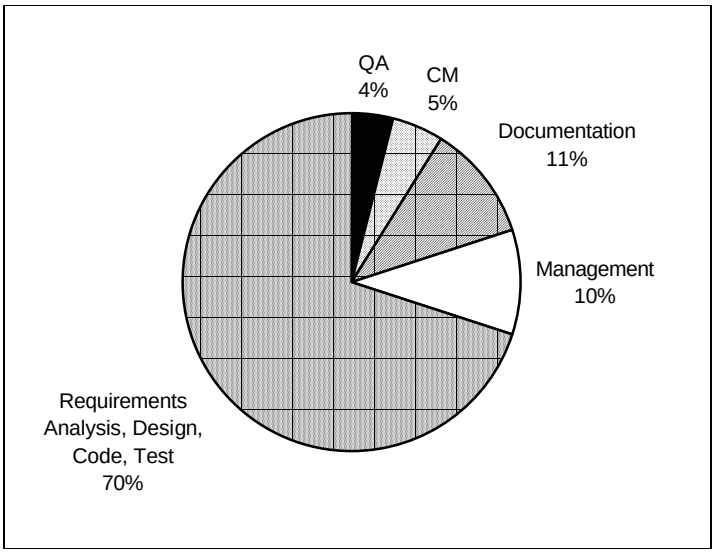


Figure 6-7. Typical Allocation of Software Project Resources

Example 6:
Error Rates

Goals:	***Determine the average rate of uncovering errors.***
	Determine which life-cycle phases yield the most errors.
	Compute the error rate in delivered software.
Measures needed:	***Project size, phase dates, and reported errors.***
	(See Sections 4.2 and 4.5.)

Error Rates by Phase

Figure 6-8 illustrates a model of the number of reported errors (normalized by the product size) over the various phases of the life cycle. This model combines product and process data and provides two types of information.

The first type is the absolute error rate expected in each phase. The rates shown here are based on SEL development projects from the mid-1980s. The model predicts about four errors per KSLOC during implementation, two during system testing, one during acceptance testing, and one-half during operation and maintenance. Those error rates by phase yield an overall average rate of seven errors per KSLOC during development. An analysis of more recent projects indicates that error rates are declining as improvements are made in the software process and technology.

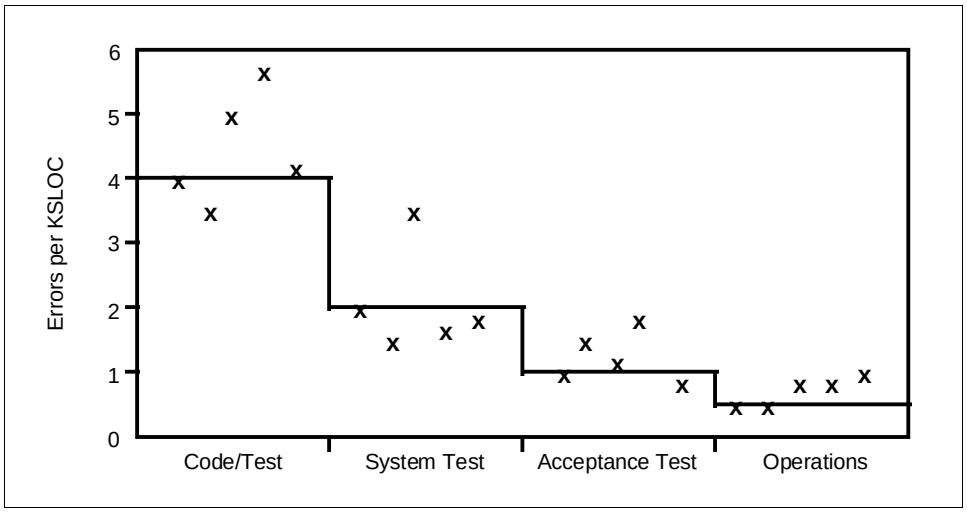


Figure 6-8. Error Detection Rate by Phase

The second piece of information is that error detection rates are halved in each subsequent phase. In the SEL, this trend seems to be independent of the actual rate values, because the 50 percent reduction by phase is holding true even as recent error rates have declined.

Example 7:
Error Classes

Goal:	***Determine what types of errors occur most often.***
Measures needed:	***Reported error information.***
	(See Section 4.2.)

Types of Errors

Figure 6-9 depicts two models of error class distribution. The model on the left shows the distribution of errors among five classes for a sample of projects implemented in FORTRAN. A manager can use such a model (introduced in Section 2.2.1) to help focus attention where it is most needed during reviews and inspections. In addition, this type of baseline can show which profiles seem to be consistent across differing project characteristics, such as in the choice of development language.

The model on the right shows the distribution across the same classes of errors for Ada projects in the same environment. Contrary to expectation, there is little difference in the error class profiles between the FORTRAN and Ada development efforts. One possible interpretation of this result is that the organization's overall life-cycle methodology and the experience of the people in that environment are stronger influences on process profiles than any one specific technology.

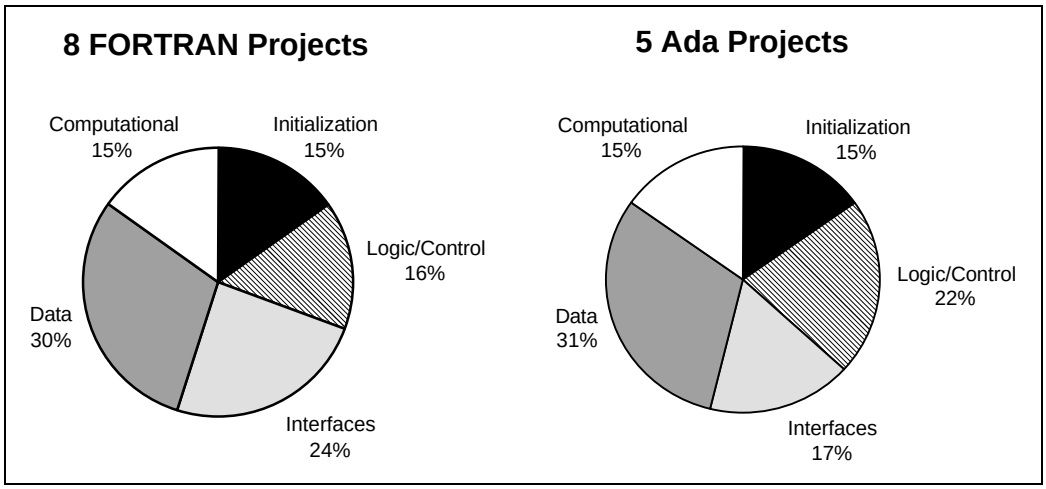


Figure 6-9. Comparative Error Class Distributions

Example 8:
Errors Versus Size and Complexity

Goals:	***Determine if error rates increase as module size increases.***
	Determine if error rates increase as module complexity increases.
Measures needed:	***Error reports by module, module size, and module complexity.***
	(See Sections 4.2 and 4.5.)

Many measures proposed in the literature attempt to model errors or effort as some function of program or design complexity. Two of the most prevalent sets are Halstead's software science measures and McCabe's cyclomatic complexity number. A 1983 SEL study (see Reference 22) examined the relative effectiveness of those measures and simpler software size measures (SLOC) in identifying error-prone modules. A linear analysis of various scatter plots using 412 modules failed to support the commonly held belief that larger or more complex systems have higher error rates.

Figure 6-10 shows that error rates actually decreased as both size and complexity increased for the large sample set in this environment.[9] However, more extensive analysis revealed that this unexpected trend occurred for only the limited set of modules used in the earlier study. When the sample size was increased, the trend reversed, suggesting that it is wise to be cautious of drawing conclusions from limited analysis.

[9] Module complexity can be derived from an analysis of completed software.

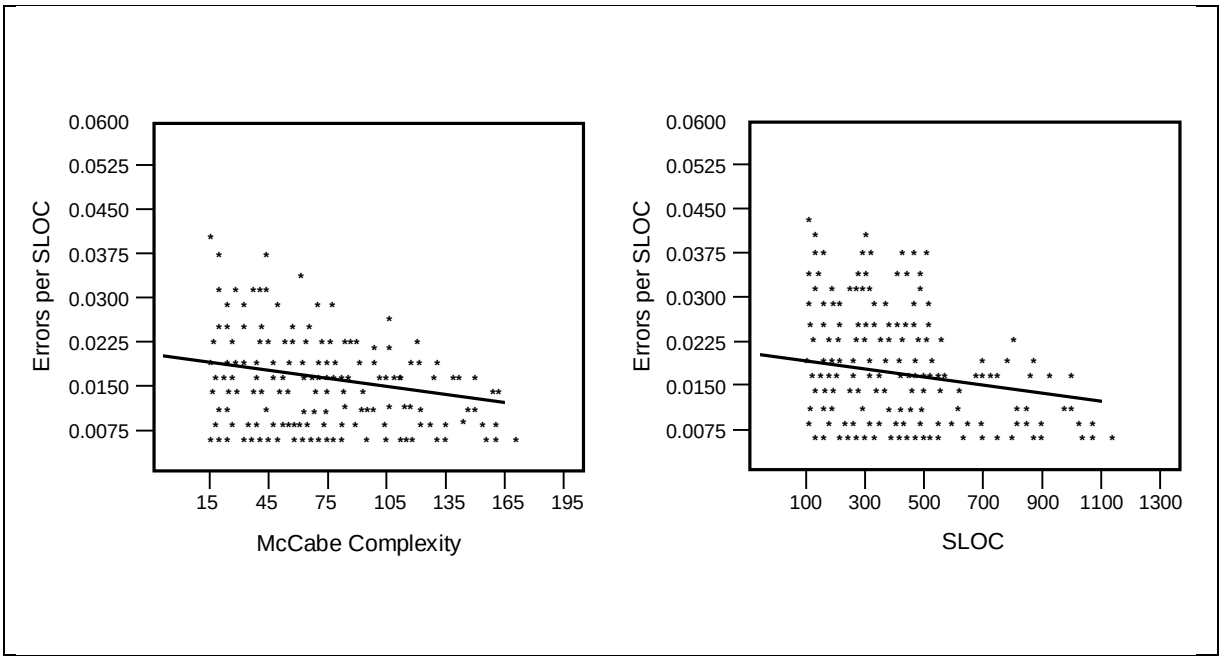


Figure 6-10. Cyclomatic Complexity and SLOC as Indicators of Errors (Preliminary Analysis)

6.1.4 Project Dynamics

An analysis of project dynamics data can give managers useful insight into changes to requirements, to controlled components, and in the estimates to completion.

Example 9:
Growth Rate Dynamics

Goal:	***Derive a model that characterizes the local rate of code production.***
Measures needed:	***Phase dates and weekly count of completed code.*** ***(See Section 4.4.)***

The growth rate of the source code in the configuration-controlled library closely reflects the completeness of the requirements product and some aspects of the software process. In the SEL environment, periods of sharp growth in SLOC are separated by periods of more moderate growth, as shown in Figure 6-11. This phenomenon reflects the SEL approach of implementing systems in multiple builds. The model also shows that, in response to requirements changes, 10 percent of the code is typically produced after the start of system testing. The uncertainty band highlights the typical variation expected with this model.

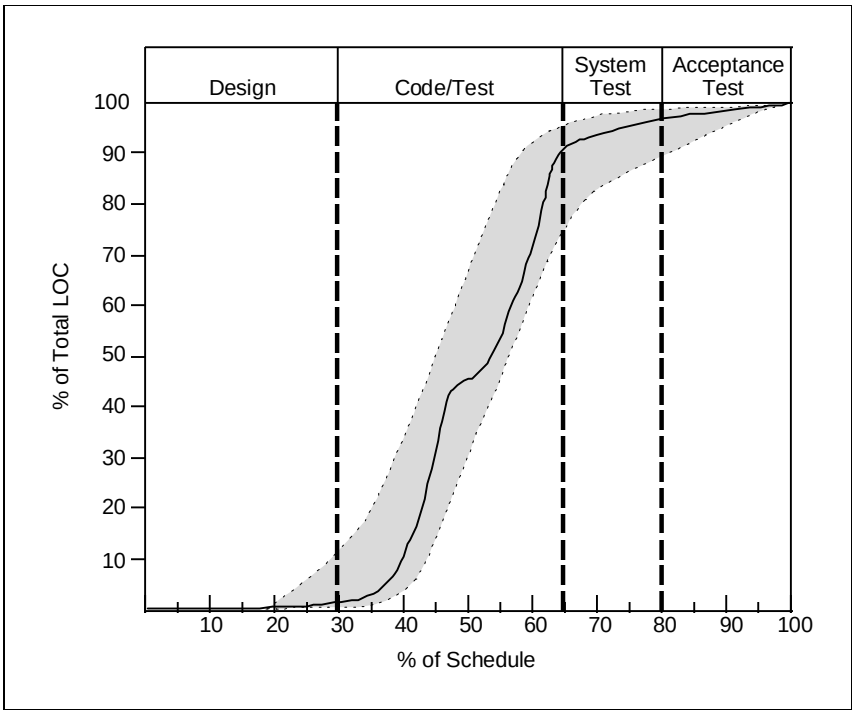


Figure 6-11. Growth Rate Model

6.2 Managing

The management activities of planning, estimating, tracking, and validating models, introduced in Section 2.2, all require insight into the characteristics of the organization's software engineering environment and processes. Measurement data extracted during the development and maintenance phases will provide quantitative insight into whether a project is progressing as expected.

An analysis of the following types of measurement information can lead to better management decision making:

- *Planned versus actual values.* Tracking ongoing progress requires not only the actual data but also planning data based on estimates from local models. Candidates for such analysis include effort, errors, software changes, software size, and software growth.
- *Convergence of estimates.* A manager should expect to revise estimates periodically. The frequency of revisions can be based on the pattern of the data being tracked. If the actuals are deviating from the current plan, more frequent updates are needed. The successive estimates themselves should eventually converge and not vary wildly from one estimate to another.
- *Error history and classes of errors.* An analysis of error data can pinpoint problems in the quality of development or maintenance processes. Possible focus areas include design or code inspections, training, and requirements management. Data from relatively few projects can be effectively used in this manner.

An effective measurement program enhances management activities:

- *Planning*. Historical information, along with estimates of the current project, enable the manager to prepare schedules, budgets, and implementation strategies.
- *Assessing progress.* Measures indicate whether projected schedules, cost, and quality will be met and also show whether changes are required.
- *Evaluating processes.* The manager needs insight into whether a selected software engineering process is being applied correctly and how it is manifested in the final product.

Using the information gained from tracking software measures, managers have numerous options for addressing possible progress or quality problems. Those options include adjusting staff, adding resources, changing processes, replanning, and enforcing a process, among others. Table 6-5 lists the examples presented in this section, which are derived from actual data on NASA software projects.

Table 6-5. Questions Supporting Management Activities

Measurement Application	Managing	Examples
Planning	What is my basis for estimating cost, schedule, and effort?	10
	What is my basis for projecting code growth and change? What is my organization's model of expected error rate?	11
Assessing Progress	Is my project development proceeding as expected?	12
	How stable are the requirements and design?	13
	Is my original staffing estimate on track?	14
	Are we correcting defects faster than they are detected? When will testing be complete?	15
	Are we producing high-quality and reliable software?	16
Evaluating Processes	Are our standard processes being applied properly? Are they having the expected effects?	17

6.2.1 Planning

A software manager's major responsibilities include effective planning at the start of a project. The manager must estimate cost, schedules, and effort; define the processes; and initiate a mechanism for tracking against the plan. The major application of measurement information for the planning phase is to make use of the derived models, relationships, and insights gained from measurement understanding efforts.

Example 10:
Projected Cost, Scheduling, and Phases

Goal:	***Estimate cost, schedule, effort, and errors.***
Measures needed:	***Project size estimate, models, and relationships.***
	(See Sections 2.2, 6.1.2, and 6.1.3.)

Although estimating the size of a new project is not easy, most organizations have an approach for producing a reasonable size estimate in SLOC. Once that size estimate has been calculated, the derived models for cost, schedule, effort, and other project characteristics can be used in the planning phase. The models described in Section 6.1 are used to derive more detailed estimates of a project based on the size estimate. The following example depicts the planning for an AGSS project whose initial size estimate is 150 KSLOC of FORTRAN code, of which 90 KSLOC is estimated to be new and 60 KSLOC is estimated to be reused from other systems.

The manager computes DLOC as

$$\begin{aligned} \text{DLOC} &= \text{New SLOC} + (\text{Reuse Cost Factor} \times \text{Reused SLOC}) \\ &= 90\text{K} + (0.2 \times 60\text{K}) \\ &= 102\text{K} \end{aligned}$$

Using a productivity rate of 3.2 DLOC per hour (see Chapter 2)

$$\begin{aligned} \text{Effort} &= \text{DLOC} / \text{Productivity} \\ &= 102 \text{ KDLOC} / (3.2 \text{ DLOC per hour}) \\ &= 31{,}875 \text{ hours} \\ &= 206 \text{ months} \end{aligned}$$

The manager next distributes the effort across the life-cycle phases (see Table 6-6) using the percentages shown in Figure 6-5 and estimates the duration of the development using the relationship introduced in Chapter 2:

$$\begin{aligned} \text{Duration} &= 4.9(\text{Effort})^{0.3} \\ &= 4.9(206 \text{ months})^{0.3} \\ &= 24.2 \text{ months} \end{aligned}$$

Figure 6-8 tells the manager to estimate 7 errors per KSLOC during development; for 150 KSLOC, the estimate is 1,050 errors distributed as shown in Table 6-6, with 75 additional errors estimated to be detected in the operational system.

Table 6-6. Project Planning Estimates

Activity	Estimate
Development Effort	
Design (30%)	62 staff-months
Code/unit test (34%)	70 staff-months
System test (16%)	33 staff-months
Acceptance test (20%)	41 staff-months
Total	206 staff-months
Duration	24.2 months
Errors	
Code/unit test	600 errors
System test	300 errors
Acceptance test	150 errors
Total development	1,050 errors
Errors	
Operations	75 errors
Annual maintenance effort	31 staff-months
Documentation effort	23 staff-months

Assuming that the system is intended to support multiple missions, the estimated annual maintenance effort (derived from Table 6-4) is 31 staff-months.

Finally, the cost of support activities can be derived from Figure 6-7. Table 6-6 shows the estimated cost of the documentation effort.

Example 11:
Project Dynamics

Goal:	***Determine the expected growth rate, change rate, and error rate of source code.***
Measures needed:	***Project size estimate, models, and relationships.***
	(See Sections 2.1 and 6.1.)

The project manager introduced in the previous example can use models derived from historical data to project the expected rate of source code growth, as well as the expected change rate and error rates of the software. Each new project will always strive to attain lower error rates; however, until those lower rates are packaged into new organizational

models, the manager should use the current historical models. Figure 6-12 illustrates the planning charts derived from the models discussed in Sections 2.1 and 6.1.

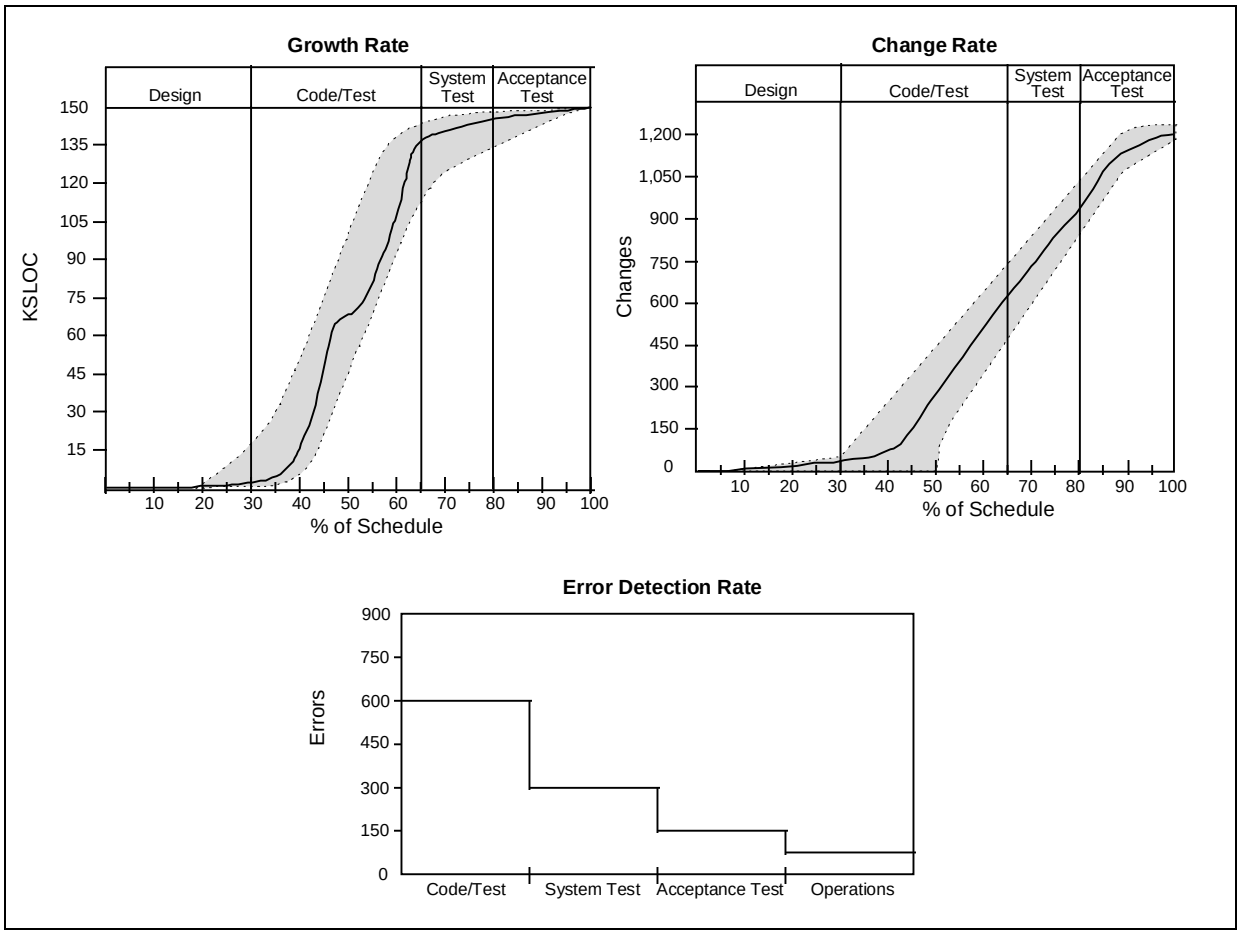


Figure 6-12. Planning Project Dynamics

Estimating the final software size is the most understood and useful basis for project planning, and the basic historical models derived during the understanding stage of a measurement program are the most important planning aids. As an organization completes more detailed analyses of the local environment, additional models will provide even more accurate planning data. Such parameters as problem complexity, team experience, maturity of the development environment, schedule constraints, and many others are all valid considerations during the planning activity. Until the measurement program provides some guidance on the effect of such parameters, project planning should rely primarily on lines of code estimates, along with the basic historical models.

6.2.2 Assessing Progress

A second important management responsibility is to assess the progress of the development and maintenance activity. Project managers must track the activities and interpret any deviations from the historical models. Although experience is the best asset for carrying out this responsibility,

several measures are helpful. The standard earned-value systems, which aid in analyzing the rate of resources consumed compared to planned completed products, are effective for supporting progress tracking. Along with earned-value techniques, other software measures can provide additional insights into development progress.

Example 12:
Tracking Code Production

Goal:	***Determine whether development is progressing as expected.***
Measures needed:	***Biweekly count of source library size, manager's updated at-completion estimates.*** ***(See Section 4.4.)***

An analysis of historical data enables the derivation of such profiles as the expected rate of code growth in the controlled library (see Figure 6-11). Using such a model, a project manager can determine whether code production is proceeding normally or is deviating from the expected range of values. As with other models, a project's deviation from the growth-rate model simply means that the project is doing something differently. For example, a project reusing a large amount of existing code may show an unexpectedly sharp jump early in the code phase when reused code is placed in the configured library. Figure 6-13 shows an example in which code growth made several jumps resulting from reuse but then followed the model derived for the local environment.

Example 13:
Tracking Software Changes

Goal:	***Determine whether requirements and design are stable.***
Measures needed:	***Changes to source code and manager's project estimates.*** ***(See Section 4.4.)***

By tracking the changes made to the controlled source library, a manager can identify unstable requirements or design. Plotting the behavior of a current project's change rate against the organization's predictive model indicates whether the project is on track or is deviating. Exaggerated flat spots (periods without changes) or large jumps (many changes made at the same time) in the data should raise flags for further investigation. Some deviations may be readily explained; for example, during testing, changes are often grouped and incorporated into the configured software at the same time, thus causing a large jump in the weekly change rate.

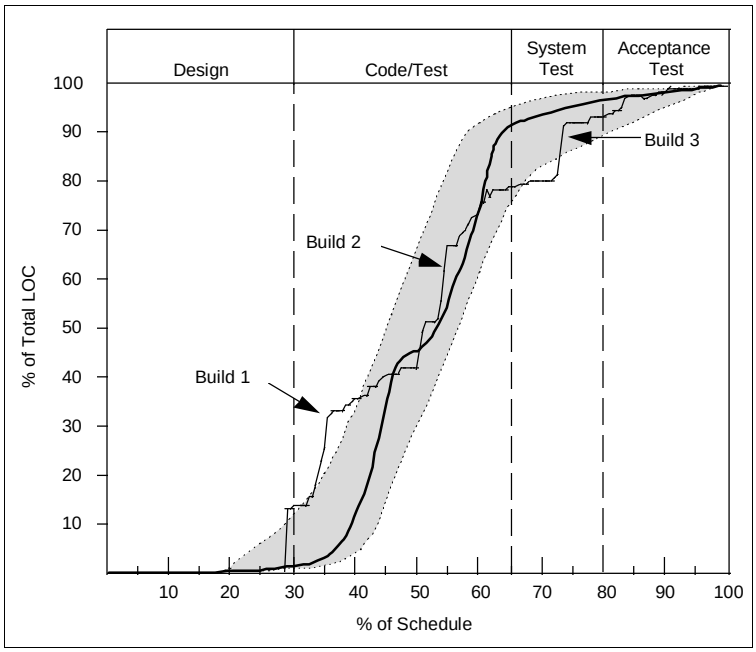


Figure 6-13. Growth Rate Deviation

Figure 6-14 presents an example from actual data for a project that experienced a higher than normal change rate. The requirements for this 130-KSLOC system were highly unstable, resulting in a deviation from the existing model (introduced in Figure 6-12). By recognizing the change rate early, managers could compensate by tightening CM procedures to maintain the quality and the schedule.

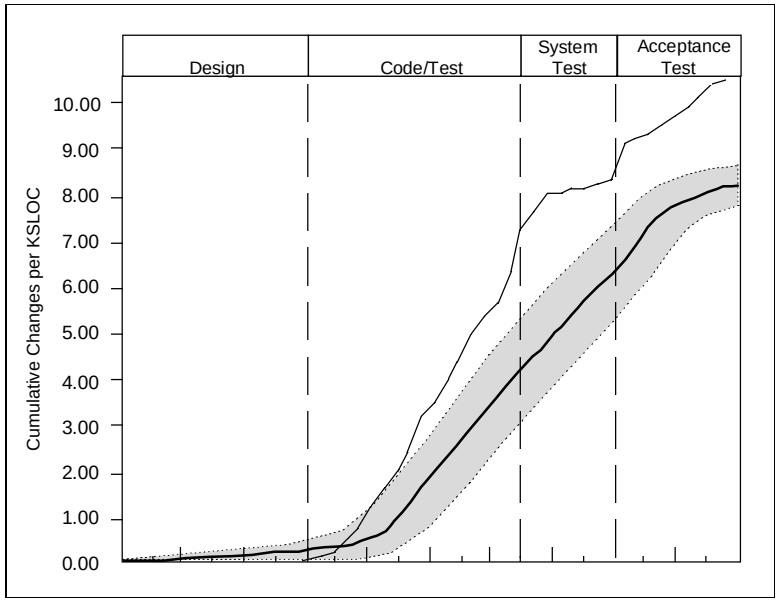


Figure 6-14. Change Rate Deviation

Example 14:
Tracking Staff Effort

Goal:	***Determine whether replanning is necessary.***
Measures needed:	***Initial project plan and weekly effort data.***
	(See Sections 4.1 and 4.4.)

By using the expected effort distribution and staffing profile over the life-cycle phases, a manager can predict the total cost and schedule based on the effort spent to date. If more effort than was planned is required to complete the design of a system, the remaining phases will probably require proportionately more effort. After determining why a deviation occurred, a manager can make an informed response by adjusting staffing, increasing the schedule, or scaling back functionality.

Deviations in effort expenditures can also raise quality flags. If all milestones are being met on an understaffed project, the team may appear to be highly productive, but the product quality may be suffering. In such a case, the manager should not automatically reduce effort predictions. An audit of design and code products, using both effort data and error data, can support an informed decision about whether to add staff to compensate for work not thoroughly completed in earlier phases.

Figure 6-15 presents an example of the use of measurement data in monitoring a project to determine whether replanning is necessary. Effort data were a key factor in management's detection and correction of several problems that would have jeopardized this project's eventual success.

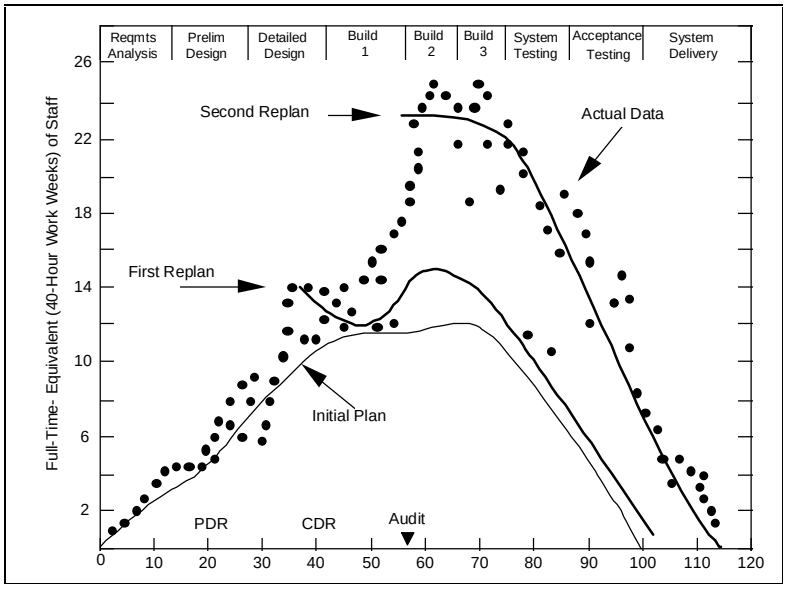


Figure 6-15. Staff Effort Deviation

The original staffing plan was based on an underestimation of the system size. Toward the end of the design phase, 40 percent more effort than planned was regularly required, indicating that the system had grown and that replanning was necessary. Although the manager's estimates of size did not reflect the significant increase, the staffing profile indicated that the system was probably much larger than anticipated. The required effort continued to grow, however, in spite of the new plan that projected a leveling off and then a decline. A subsequent audit revealed that an unusually high number of requirements were still unresolved or changing, resulting in excessive rework. As a part of the corrective action, a second replanning activity was needed.

Example 15:
Tracking Test Progress

Goal:	***Determine whether the testing phase is progressing as expected.***
Measures needed:	***Failure report data and change data.*** ***(See Section 4.2.)***

By consistently tracking reported versus fixed discrepancies, a manager gains insight into software reliability, testing progress, and staffing problems. The open failure reports should decline as testing progresses unless the project is understaffed or the software has many defects.

When the "open" curve falls below the "fixed" curve, defects are being corrected faster than new ones are reported. At that time, a manager can more confidently predict the completion of the testing phase. Figure 6-16 shows an example of discrepancy tracking that gave the manager an early indication of poor software quality (at Week 15). Staff members were added to increase the error-correction rate (during Weeks 20 through 35), and the system attained stability (at Week 35).

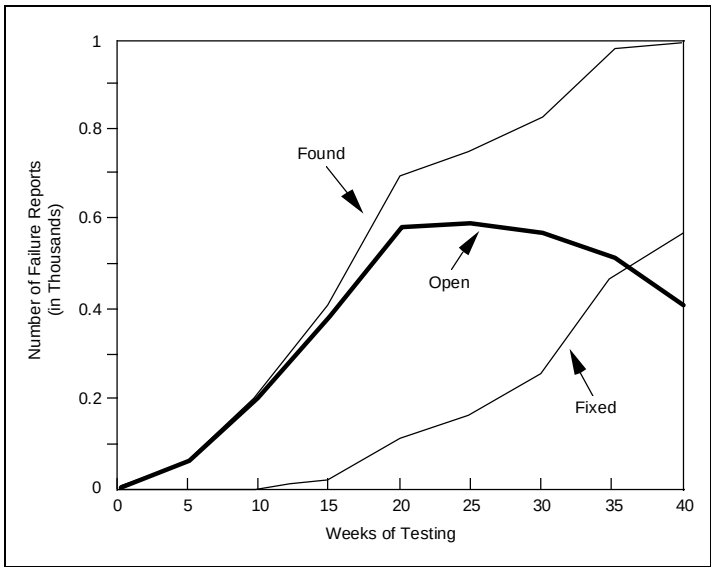


Figure 6-16. Tracking Discrepancies

Example 16:
Tracking Software Errors

Goal:	***Determine the quality of the software.***
Measures needed:	***Error report data, historical models, and size estimates.***
	(See Sections 4.2, 4.4, 5.3.3, and 6.1.3.)

One commonly used measure of software quality is the software error rate. Tracking the project's error rate against an organization's historical model can provide a simple estimate of the predicted quality of the delivered software. A consistent understanding of what to count as an error enables the organization to make reasonable predictions of the number of errors to be uncovered, as well as when they will be found.

The model in Figure 6-8 indicates that detected errors were reduced by half in subsequent phases following coding and unit testing. By estimating the total size of the software and by tracking the errors detected during the coding and unit testing phase, the project manager can both observe the quality of the existing system relative to the model and also project the quality of the delivered software.

Figure 6-17 is another view of the same model showing the cumulative errors detected throughout the life cycle (see also Figure 5-7). The model compares error rates reported during the coding and early test phases of an actual NASA project. The error rate can deviate from the model for many reasons, including the possibility that the development team is not reporting errors. However, it is still worthwhile to track the errors and to assume that the information is reasonably reliable. The example indicates that the projected quality or reliability (based on the predicted error rate) is an improvement over the average local project; indeed, in this case the project turned out to be an exceptionally reliable system.

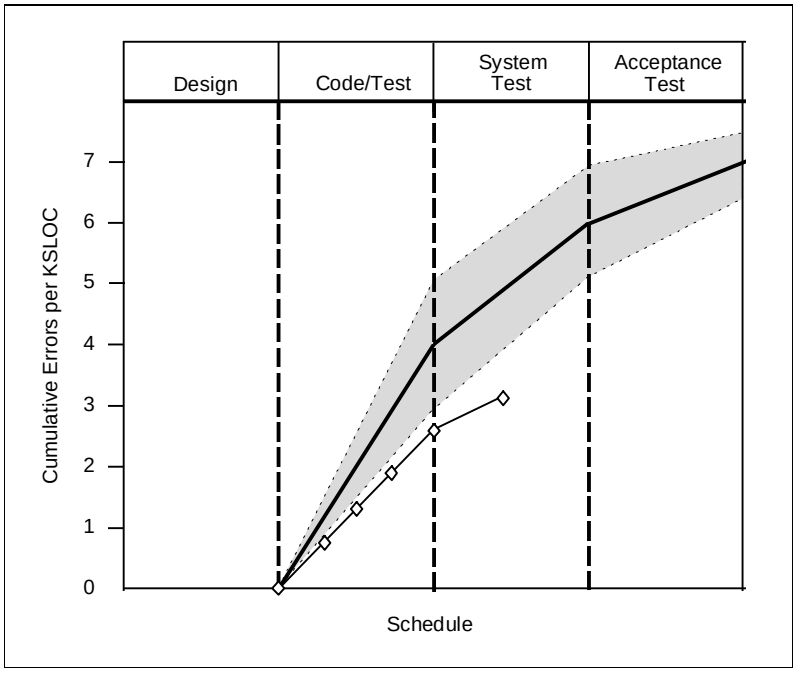


Figure 6-17. Projecting Software Quality

6.2.3 Evaluating Processes

A third responsibility of the software manager is to determine whether the project's standard software processes are, in fact, being used, and if there is any impact on the product. Project personnel may fail to apply a standard process because of inadequate training, team inexperience, misunderstandings, or lack of enforcement. Whatever the reasons, the manager must try to determine whether the defined process is being used.

Example 17:
Source Code Growth

Goal:	***Determine whether the Cleanroom method is being applied.***
Measures needed:	***Project phase date estimates, completed source code, and historical models.*** ***(See Section 4.4.)***

One characteristic of the Cleanroom method is an increased emphasis on source code reading before the code is released for system integration. This emphasis can be confirmed by tracking the source code growth and observing two phenomena:

1. A delay in the phasing of the code completion profile
2. A significant step function profile of the code completion rate caused by the strict incremental development of Cleanroom

The sample plot in Figure 6-18 is based on actual data from an organization's first use of the Cleanroom method. The data exhibited both expected phenomena, suggesting that the Cleanroom method was indeed part of the project process. Such measurement analysis is useful only to identify occasions when expected differences do not occur, so that the manager can try to determine the cause.

By tracking the values of process parameters, the manager can determine whether the process is helping to attain the organization's goals. If not, the manager should consider changing the process. The following section discusses using measurement to guide process improvement.

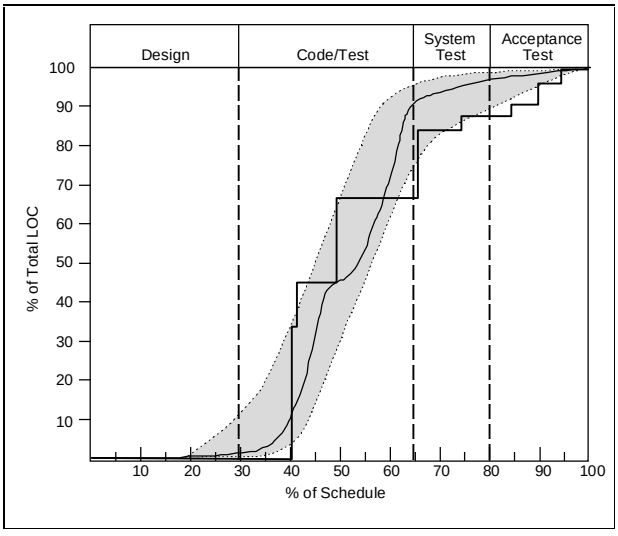


Figure 6-18. Impact of the Cleanroom Method on Software Growth

6.3 Guiding Improvement

One key reason for software measurement is to guide continual improvement in the organization's products and services. The same measurement activities that support understanding and managing can provide a basis for improvement.

To be able to assess how a process change affects a product, the manager must measure both the processes applied and the products developed. Two key analyses must be performed:

1. Verify that the process under study shows the expected measured behavior (either changed or similar to other processes).
2. Compare ongoing activities with the baseline measures developed to establish an understanding.

A specific innovation may result in many changes to process elements, some helpful and others not. Experience on subsequent projects is needed to adapt the process change to an environment. The types of adaptations include the following:

- Eliminate processes that provide little or no value.
- Accentuate processes that help.
- Determine the impact of specific techniques.
- Write new policies, standards, and procedures.
- Tailor processes for specific needs.

The two examples in this section illustrate the application of measurement for guiding improvement. Additional examples are provided in NASA's *Software Process Improvement Guidebook* (Reference 25).

Example 18:
Cleanroom

Assume that an organization's goal is to decrease the error rate in delivered software while maintaining (or possibly improving) the level of productivity. The organization must understand the current software engineering process, using historical data to establish a baseline for its error rate and productivity measures.

In this example, the organization has decided to change the process by introducing the Cleanroom method (see Reference 13). Cleanroom focuses on achieving higher reliability by preventing defects. Because the organization's primary goal is to reduce the error rate, there is no concern that the Cleanroom method does not address reuse, portability, maintainability, or many other process and product characteristics.

As the organization develops new products using the modified process, which incorporates the Cleanroom method, it must continue to collect data for both process and product measures and look for possible changes. Keep in mind that a *change* is not always an *improvement*; it must be possible to measure two things: (1) that a difference exists between the original and the changed product and (2) that the new product is better than the original. Table 6-7 lists the measures that are important indicators for this example and

summarizes their usage. Other software process and product characteristics, such as schedule, maintainability, and amount of reuse, may also reveal deviations beyond the expected baseline ranges. Such deviations must be investigated to determine whether the effect is related to the introduction of the Cleanroom method.

Table 6-7. Indicators of Change Attributable to Cleanroom

Measure	Type	Indicator
Cost		
Effort	Product	Expectation: Cleanroom should not decrease productivity.
Effort distribution	Process	Expectation: Cleanroom may show increased design time.
Size		
Software size	Product	Expectation: Cleanroom should have no impact.
Size growth	Process	Expectation: Cleanroom may affect measured profile.
Number of Errors	Product	Expectation: Cleanroom should increase reliability.

To observe changes, the organization must analyze the measurement data at regular intervals during the Cleanroom development period and compare the results with the baseline. For example, Figure 6-19a compares the results of measuring development activities on several SEL projects that used the Cleanroom method against the current baseline activity profile in the same organization. The slight changes in the effort distribution profiles suggest that the new method may have affected the development process, but the difference in percentages is not conclusive. A closer look (see Figure 6-19b) at the subactivities within the "code" category reveals more substantial differences and provides clear evidence of an impact on the relative percentages of the code writing and code reading processes.

During the Cleanroom experiment (see Reference 14), the SEL also compared another measure, software size growth, with the baseline. Figure 6-18 illustrates the marked differences between the profiles. The Cleanroom profile exhibits a more pronounced stepwise growth pattern, which results from the higher number of software builds required by the Cleanroom method. Whereas developers typically used two or three builds on projects that made up the baseline, they used from five to eight builds during the Cleanroom experiment.

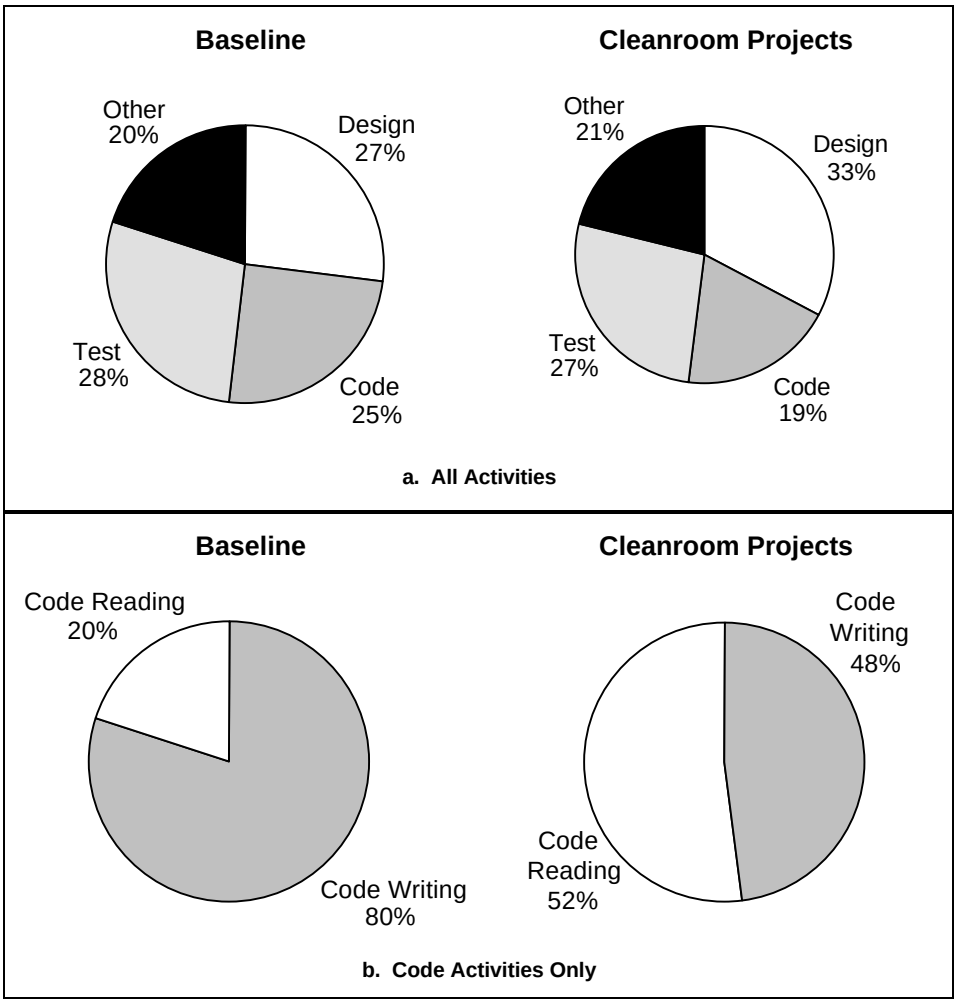


Figure 6-19. Impact of the Cleanroom Method on Effort Distribution

Both of the measures discussed above—effort by activity and software growth—are strong initial indicators that the Cleanroom method has indeed changed the process. Those process measures alone cannot, however, prove that the change has benefited the product. To determine that the change is an improvement requires an analysis of measures based on the project goals, specifically, higher product reliability (that is, lower error rates) and stable productivity. Table 6-8 shows the error rate and productivity measures for the baseline and experimental projects using the Cleanroom method. (The Cleanroom experiment includes data through the system testing phase and excludes acceptance testing; baseline values shown in the table have been adjusted to represent the same portions of the life cycle.)

The results of the experiment appear to provide preliminary evidence of the expected improvement in reliability after introducing the Cleanroom method and may also indicate an improvement in productivity. Two conclusions can be drawn:

1. Process measures can verify that adopting a new technology has affected the baseline process.

Table 6-8. Impact of the Cleanroom Method on Reliability and Productivity

Data Source	Error Rate (Errors per KDLOC)	Productivity (DLOC per Day)
Baseline	5.3	26
Cleanroom 1	4.3	40
Cleanroom 2	3.1	28
Cleanroom 3	6.0	20

2. Product measures can quantify the impact (positive, negative, or none) of a new technology on the product.

Both types of measures can then be used to model the new process and expand the experience baseline.

Example 19:
Independent Verification and Validation

Not all process changes result in measured product benefits. In 1981, the SEL studied a testing approach using an independent verification and validation (IV&V) process. IV&V promised to improve error detection and correction by finding errors earlier in the development cycle, thus reducing cost and increasing overall reliability with no negative impact on productivity. Determining the effect of this testing process on reliability and cost were two major study goals. Table 6-9 lists the measures that are important indicators for this example and summarizes the use of each.

Measurement analysts selected two projects for IV&V study and two similar ones for use as baseline comparison efforts. For this study, the activities performed by the IV&V team included the following:

- Verifying requirements and design
- Performing independent system testing
- Ensuring consistency from requirements to testing
- Reporting all findings

The next series of figures shows the measured results of the study.

Table 6-9. Indicators of Change Attributable to IV&V

Measure	Type	Indicator
Cost		
Effort	Product	Expectation: Cost of IV&V effort would be offset by reductions in error correction effort and decreases in system and acceptance test effort.
Effort distribution	Process	Expectation: IV&V process would show increased effort in early phases.
Staffing profile	Process	Expectation: Greater startup staffing for IV&V would affect profile model.
Errors		
Number	Product	Expectation: IV&V process would increase reliability.
Source	Process	Expectation: The number of requirements and design errors found in later phases would decrease.

Figure 6-20 illustrates the effect of IV&V on requirements and design errors. Requirements ambiguities and misinterpretations were reduced by 87 percent. The results show relatively little effect on design errors, however, especially on complex design errors.

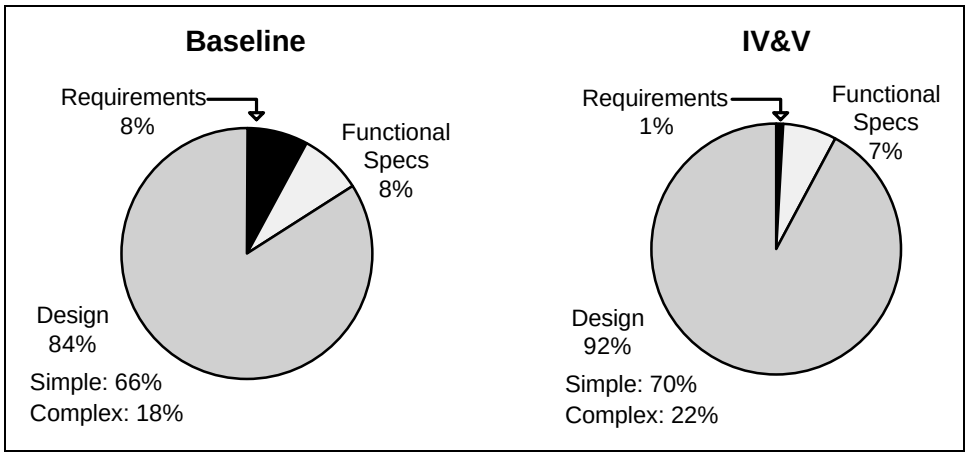


Figure 6-20. Impact of IV&V on Requirements and Design Errors

Figure 6-21 depicts the percentage of errors found after the start of acceptance testing. The IV&V projects exhibited a slight decrease in such errors but showed no significant increase in the early detection of errors.

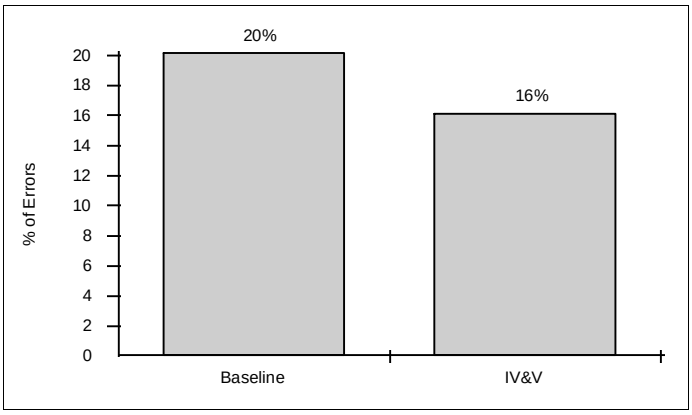


Figure 6-21. Percentage of Errors Found After Starting Acceptance Testing

Figure 6-22 shows the error rates by phase; the rates in the operations phase are the key indicators of IV&V effectiveness. The baseline error rate during operations is 0.5 errors per KSLOC; however, the error rate for the IV&V projects was slightly higher.

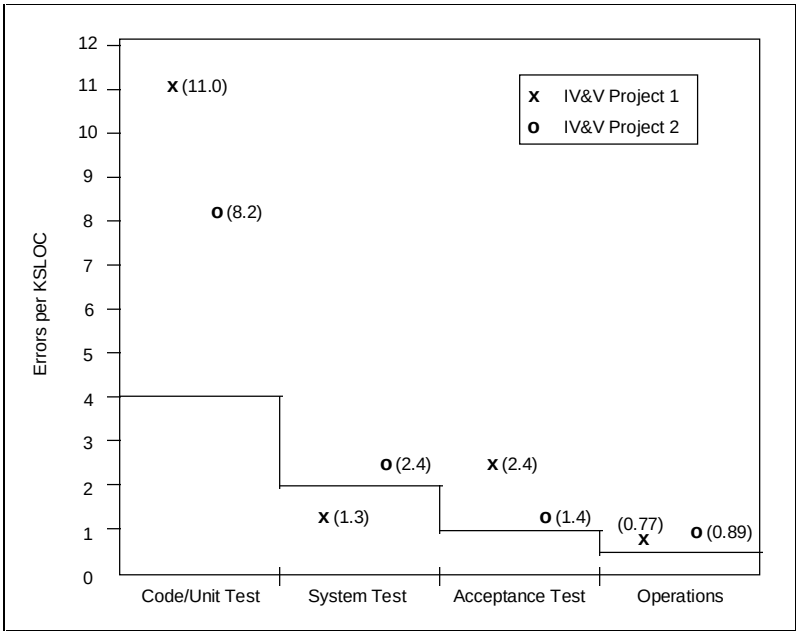


Figure 6-22. IV&V Error Rates by Phase

The final indicators for this experiment were effort distribution and overall cost. Figure 6-23 shows that process change in the effort distribution by phase did occur with the IV&V projects. According to expectation, developers' design effort slightly decreased; however, the substantial increase in coding and unit testing was somewhat surprising.

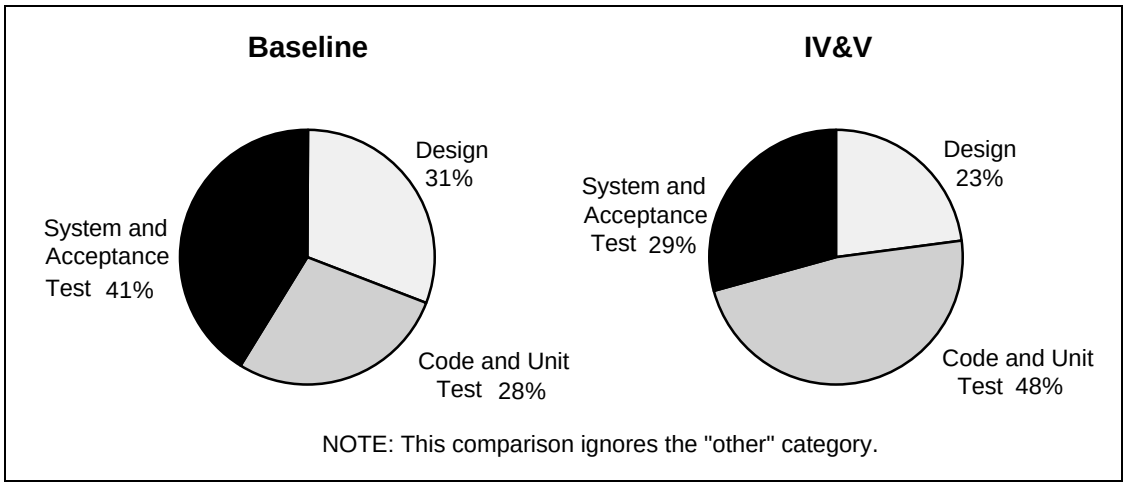


Figure 6-23. Impact of IV&V on Effort Distribution

Figure 6-24 shows the impact of the IV&V process in two areas: the overhead of the IV&V team itself and the increased cost to the development team because of their interactions with a new group. Together, the overall cost increased by 85 percent, an unacceptably high cost to pay for no measurable increase in overall product quality.

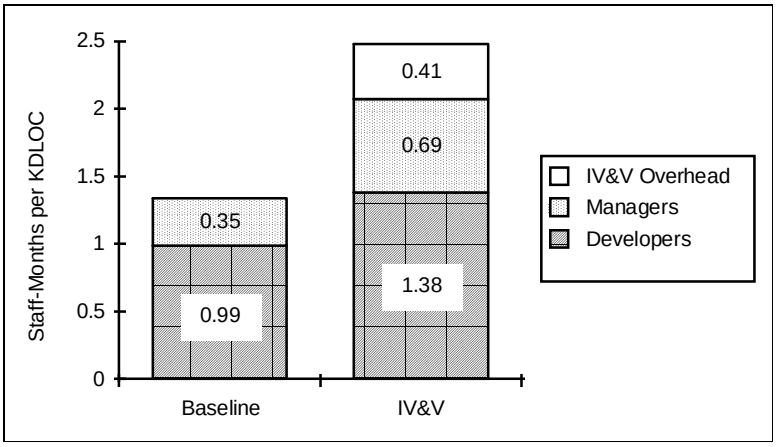


Figure 6-24. Impact of IV&V on Cost

This example is not intended to indicate that IV&V technology is never beneficial. On projects requiring extremely high levels of safety and reliability, the benefits of IV&V can often outweigh the added cost. The cited software project was a ground-based, non-life-critical system for which the extra safety was not worth the added overhead. Every organization must judge the appropriateness of a potential software process change within the context of the local environment and the organization's goals.

Chapter 7. Experience-Based Guidelines

Chapter Highlights

MEASUREMENT GUIDELINES

- ✔ The goal is application of results, not data collection.
- ✔ The focus should be on self-improvement, not external comparison.
- ✔ Measurement data are inexact.
- ✔ Interpretation is limited by analysts' abilities.
- ✔ Measurement should not threaten personnel.
- ✔ Automation of measurement has limits.

The following guidelines are precautionary notes for any software organization that plans to include software measurement as part of its development process. Some of these guidelines have been repeated several times throughout this document. Although some may seem counterintuitive, each has been derived from the experiences of extensive, mature measurement programs.

Guideline 1:
Data collection should not be the dominant element of process improvement; application of measures is the goal.

Focusing on collecting data rather than on analyzing and applying the data wastes time, effort, and energy. Although many organizations are convinced that measurement is a useful addition to their software development and maintenance activities, they do not fully plan for the use, benefits, and applications of the collected measures. As a result, the measurement program focuses on defining the list of measures to be collected and the forms that will be used to collect the data, rather than on the specific goals of the measurement efforts.

Having specific and clearly defined goals facilitates the task of determining which data are required. For example, if a goal is to determine error class distribution characteristics for each phase of the software life cycle, then data must be gathered on what classes of errors occur in what phases.

Experience in major mature measurement programs has shown that at least three times as much effort should be spent on analyzing and using collected data as on the data collection process itself. Focusing on data collection is a common mistake, similar to that of focusing on the development of "lessons learned" from software efforts rather than on applying previous lessons learned. More software lessons-learned reports are written than are ever read or used.

Software developers who are asked to collect data have the right to know how the data will be used and how that use will benefit their organization. Plans for analysis and application of the data must be well developed before the collection process is initiated. A measurement program that focuses on the collection, as opposed to the application, of the measurement data will fail.

Guideline 2:
The focus of a measurement program must be self improvement, not external comparison.

Because the primary reasons for measurement are to guide, manage, and improve within specific software domains, the analysis and use of any measurement information must logically focus on local improvement. Little emphasis should be placed on comparing local results and information with that from other domains, because combining data across dissimilar domains rarely produces meaningful results. In fact, organizations rarely define specific goals requiring external comparison.

There are two significant corollaries to this guideline:

1. Define standard terminology locally instead of generating widely accepted standard definitions. For example, provide a standard local definition of a line of code, because there is no universally accepted definition.

2. Use measurement data locally. Combining measurement data into larger, broader information centers has never proved beneficial and consumes unnecessary effort. Focus, instead, on producing higher quality, local data centers.

Guideline 3:
Measurement data are fallible, inconsistent, and incomplete.

Measurement programs that rely significantly on the high accuracy of raw software measurement data are probably doomed to failure. Because of the nature of the measurement process and the vast number of uncertainties that are part of it, the measurement data will always be inexact.

Relying primarily on the statistical analysis of the data collected for software development is a serious mistake. Collection of measurement data is one small component of the overall set of factors required to analyze software and software technologies effectively. The following additional factors must be considered:

- *Subjective information*—The general observations and assessments of developers, managers, and analysts are as vital as the objective data collected via forms and tools.

- *Context of the information*—Each set of data must be analyzed within a well-understood and defined context. Attempting to analyze larger and larger sets of measurement data adds to the confusion and difficulty of putting each set of data in its appropriate class of interpretation.

- *Qualitative analysis*—Because of the ever present danger that measures are erroneous, biased, or missing, each analysis and application of measurement data must include an analysis of the quality of the information. The measurement data characteristics must first be determined by analyzing patterns, inconsistencies, gaps, and accuracy. Any interpretation of measurement data results must include compensation for the quality of the data.

- *Defined goals*—Successful analysis of available data requires that the analyst first understand the goals that motivated the data collection. By understanding the goals of the measurement efforts, an analyst can interpret data gaps, biases, definitions, and even levels of accuracy. The goals will significantly influence the quality, consistency, and level of detail of the data analysis.

Because of the limited accuracy of measurement data, overdependence on statistical analysis of these data can lead to erroneous conclusions and wasted efforts. Although statistical analysis is a powerful mechanism for determining the strengths and weaknesses of collected measures and providing insight into the meaning of the data, it must be used as only one limited tool toward the goal of meaningful application of measurement data.

Another potential pitfall exists in the use of subjective data to characterize software development. Many measurement programs attempt to characterize the processes of each development project by recording a rating factor for several process elements such as "degree of use of modern programming practices," "experience of the team," "complexity of the problem," or "quality of the environment." Although successful analysis of measurement data must consider the context, problem, domain, and other factors, extensive studies within NASA measurement programs have repeatedly failed to show any value in analyzing such rating information. Because there are many inconsistencies in the definition and interpretation of terms such as "problem complexity" or "modern programming practices" and because of the inconsistencies in the value judgments of the people doing the ratings, the use of measurement data should be limited to providing a general understanding of the project—nothing more.

Guideline 4:
The capability to qualify a process or product with measurement data is limited by the abilities of the analysts.

Measurement data must be interpreted properly to provide meaningful results. For example, if an analyst cannot clearly and precisely define "software complexity," then no tool or measure can determine if software is too complex. There is a danger in expecting that a large amount of data combined with some software tool will provide a manager or analyst with a clear representation of software quality. The data and tool can represent only what the manager or analyst interprets as quality.

Inexperienced measurement programs occasionally assume the existence of a generally accepted threshold defining the boundary between acceptable and unacceptable values for some measures. For example, a program unit that is larger than some predetermined code size might be deemed undesirable. Similar thresholds are sometimes assumed for complexity, error rate, change rate, test failure rate, and many other measures. Establishing control limits for comparing measurement values is important, but the limits must be computed on the basis of local experience. It should not be assumed that there is some predefined threshold that defines an absolute boundary of acceptable values for local measures.

Guideline 5:
Personnel treat measurement as an annoyance, not a significant threat.

One of the most obvious and important guidelines for any measurement program is to emphasize consideration for the concerns of development and maintenance personnel. Measurement programs should not be used to qualify or characterize differences between individuals providing measurement data. If confidentiality is assured, project personnel will provide requested measurement information as freely as they provide other documentation or reports.

Experience has shown that, as long as managers ensure that measurements will never be used to evaluate performance or rate programmers, the development and maintenance teams will treat measurement responsibilities as just one additional task that is a part of their job.

Guideline 6:
Automation of measurement has limits.

Nearly every measurement program starts with two well-intentioned goals:

1. Measurement will be nonintrusive.
2. Measurement will be automated.

The process of measurement, however, cannot be totally automated. Essential human processes cannot be replaced by automated tools unless the measurement program is limited to a high-level survey activity, because the opinions, insight, and focus of individual programmers and managers are necessary to carry out effective measurement programs.

Tools can automate a limited set of routine processes for counting such measures as code size, code growth, errors, and computer usage; however, insight into the reasons for errors, changes, and problems requires human intervention. Without that insight and the verification of measurement information, collected data are of limited value.

One NASA organization with a mature measurement program uses the automated tools listed in Table 7-1.

Table 7-1. Examples of Automated Measurement Support Tools

Tool	Use
Code analyzers	Record code characteristics at project completion
DBMS tools	Store, validate, and retrieve information
CM tools	Provide counts of changes to source code
Operating system accounting tools	Provide computer usage data

This same organization has found that many other measures must be compiled manually; some examples are listed in Figure 7-1.

Even a well-defined and focused measurement program requires manual intervention. Because the team provides only the limited amount of information needed to satisfy the organizational goals, however, the measurement program will have a correspondingly limited intrusive impact on the development and maintenance organization.

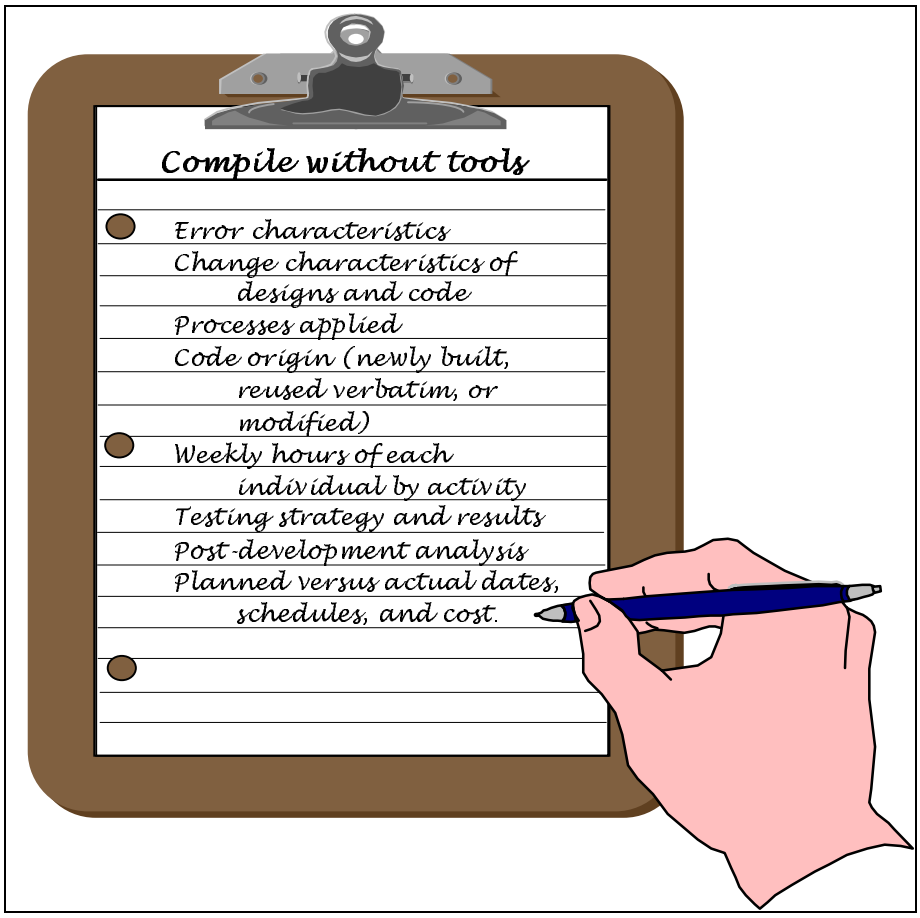


Figure 7-1. Examples of Measures Collected Manually

Appendix A. Sample Data Collection Forms

This appendix contains many of the data collection forms that are used within the NASA GSFC SEL measurement program. Reference 19 provides a detailed guide to using all of the SEL forms. An organization establishing a new measurement program may want to base its own set of forms on the samples. Table A-1 summarizes the purpose of the forms, which appear in alphabetical order on the following pages.

Table A-1. SEL Data Collection Forms

Name	Purpose
Change Report Form	Records information on changed units; is filled out each time a configured unit is modified
Component Origination Form	Provides information on software units as they are entered into the project's configured library
Development Status Form	Provides a record of the current status of the project parameters; is filled out by the project manager on a regular basis
Maintenance Change Report Form	Characterizes the maintenance performed in response to a change request
Personnel Resources Form	Provides information on hours spent on a project and how the effort was distributed; is filled out weekly by software developers or maintainers
Project Completion Statistics Form	Records final project statistics
Project Estimates Form	Records the completion estimates for project parameters; is filled out by project managers
Project Startup Form	Records general project information collected at the project startup meeting
Services/Products Form	Records use of computer resources, growth history, and services effort; is completed weekly
Subjective Evaluation Form	Records opinions that characterize project problems, processes, environment, resources, and products
Subsystem Information Form	Provides subsystem information at preliminary design review and whenever a new subsystem is created
Weekly Maintenance Effort Form	Records hours expended on maintenance activities

CHANGE REPORT FORM

Name:_____ **Approved by:**_____

Project:_____ **Date:**_____

Section A - Identification

Describe the change: (What, why, how) _____

_____

_____

_____

_____

_____

Effect: What components are changed?

Prefix	Name	Version

(Attach list if more space is needed)

Effort: What additional components were examined in determining what change was needed?

_____

_____

_____

_____

_____

Location of developer's source files:_____

	month	day	year
Need for change determined on:			
Change completed (incorporated into system):			

Check here if change involves Ada components. (If so, complete questions on reverse side.) ☐

	1 hr/less	1 hr/1 day	1/3 days	> 3 days
Effort in person time to isolate the change (or error):				
Effort in person time to implement the change (or correction):				

Section B - All Changes

Type of Change (Check one)

- ☐ Error correction
- ☐ Planned enhancement
- ☐ Implementation of requirements change
- ☐ Improvement of clarity, maintainability, or documentation
- ☐ Improvement of user services
- ☐ Insertion/deletion of debug code
- ☐ Optimization of time/space/ accuracy
- ☐ Adaptation to environment change
- ☐ Other (Describe below)

Effects of Change

Y	N	
☐	☐	Was the change or correction to one and only one component? (Must match Effect in Section A)
☐	☐	Did you look at any other component? (Must match Effort in Section A)
☐	☐	Did you have to be aware of parameters passed explicitly or implicitly (e.g., COMMON blocks) to or from the changed components?

Section C - For Error Corrections Only

Source of Error (Check one)

- ☐ **Requirements**
- ☐ **Functional specifications**
- ☐ **Design**
- ☐ **Code**
- ☐ **Previous change**

Class of Error (Check most applicable)*

- ☐ **Initialization**
- ☐ **Logic/control structure (e.g., flow of control incorrect)**
- ☐ **Interface (internal) (module-to-module communication)**
- ☐ **Interface (external) (module to external communication)**
- ☐ **Data (value or structure) (e.g., wrong variable used)**
- ☐ **Computational (e.g., error in math expression)**

***If two are equally applicable, check the one higher on the list.**

Characteristics (Check Y or N for all)

Y	N	
☐	☐	**Omission error (e.g., something was left out)**
☐	☐	**Commission error (e.g., something incorrect was included)**
☐	☐	**Error was created by transcription (clerical)**

For Librarian's Use Only

Number:_____
Date:_____
Entered by:_____
Checked by:_____

JANUARY 1994

Figure A-1. Change Report Form (1 of 2)

CHANGE REPORT FORM
Ada Project Additional Information

1. Check which Ada feature(s) was involved in this change (Check all that apply)

- ☐ Data typing
- ☐ Subprograms
- ☐ Exceptions
- ☐ Generics
- ☐ Program structure and packaging
- ☐ Tasking
- ☐ System-dependent features
- ☐ Other, please specify_____ (e.g., I/O, Ada statements)

2. For an error involving Ada components:

a. Does the compiler documentation or the language reference manual explain the feature clearly? _____ (Y/N)

b. Which of the following is most true? (Check one)

- ☐ Understood features separately but not interaction
- ☐ Understood features, but did not apply correctly
- ☐ Did not understand features fully
- ☐ Confused feature with feature in another language

c. Which of the following resources provided the information needed to correct the error? (Check all that apply)

- ☐ Class notes
- ☐ Ada reference manual
- ☐ Own project team member
- ☐ Own memory
- ☐ Someone not on team
- ☐ Other

d. Which tools, if any, aided in the detection or correction of this error? (Check all that apply)

- ☐ Compiler
- ☐ Symbolic debugger
- ☐ Language-sensitive editor
- ☐ CMS
- ☐ Source Code Analyzer
- ☐ P&CA (Performance and Coverage Analyzer)
- ☐ DEC test manager
- ☐ Other, specify_____

3. Provide any other information about the interaction of Ada and this change that you feel might aid in evaluating the change and using Ada

6201G(13)-13

NOVEMBER 1991

Figure A-1. Change Report Form (2 of 2)

COMPONENT ORIGINATION FORM

Identification

Name:_____

Project: _____ Date:_____

Subsystem Prefix: _____

Component Name:_____

Configuration Management Information

Date entered into controlled library (supplied by configuration manager): _____

Library or directory containing developer's source file:_____

Member name:_____

Relative Difficulty of Developing Component

Please indicate your judgment by circling one of the numbers below.

Easy		Medium		Hard
1	2	3	4	5

Origin

If the component was modified or derived from a different project, please indicate the approximate amount of change and from where it was acquired; if it was coded new (from detailed design) indicate NEW.

_____ NEW
_____ Extensively modified (more than 25% of statements changed)
_____ Slightly modified
_____ Old (unchanged)

For Librarian's Use Only

Number: _____
Date:_____
Entered by:_____
Checked by:_____

If not new, what project or library is it from?_____
Component or member name: _____

Type of Component (Check one only)

_____ INCLUDE file (e.g., COMMON)
_____ Control language (e.g., JCL, DCL, CLIST)
_____ ALC (assembler code)
_____ FORTRAN source
_____ Pascal source
_____ C source
_____ NAMELIST or parameter list
_____ Display identification (e.g., GESS, FDAF)
_____ Menu definition or help
_____ Reference data files
_____ BLOCK DATA file
_____ Ada subprogram specification
_____ Ada subprogram body
_____ Ada package specification
_____ Ada package body
_____ Ada task body
_____ Ada generic instantiation
_____ Ada generic specification
_____ Ada generic body
_____ Other

Purpose of Executable Component

For executable code, please identify the major purpose or purposes of this component. (Check all that apply).

_____ I/O processing
_____ Algorithmic/computational
_____ Data transfer
_____ Logic/decision
_____ Control module
_____ Interface to operating system
_____ Process abstraction
_____ Data abstraction

NOVEMBER 1991

Figure A-2. Component Origination Form

DEVELOPMENT STATUS FORM

Name: _____

Project: _____ Date: _____

Please complete the section(s) that is appropriate for the current status of the project.

Design Status	
Planned total number of components to be designed (New, modified, and reused)	
Number of components designed (Prolog and PDL have been completed)	

Code Status	
Planned total number of components to be coded (New, modified, and reused)	
Number of components completed (Added to controlled library)	

Testing Status	System Test	Acceptance Test
Total number of separate tests planned		
Number of tests executed at least one time		
Number of tests passed		

Discrepancy Tracking Status (from beginning of system testing)	
Total number of discrepancies reported	
Total number of discrepancies resolved	

Specification Modification Status (from beginning of requirements analysis)	
Total number of specification modifications received	
Total number of specification modifications completed (implemented)	

Requirements Questions Status (from beginning of requirements analysis)	
Total number of questions submitted to analysts	
Total number of questions answered by analysts	

Check here if there are no changes ☐

For Librarian's Use Only
Number: _____
Date: _____
Entered by: _____
Checked by: _____

6201G(39)-8

NOVEMBER 1991

Figure A-3. Development Status Form

MAINTENANCE CHANGE REPORT FORM

For Librarian's Use Only

Number: _____
Date: _____
Entered by: _____
Checked by: _____

Name: _____ **OSMR Number:** _____

Project: _____ **Date:** _____

SECTION A: Change Request Information

Functional Description of Change: _____

What was the type of modification?

- ____ Correction
- ____ Enhancement
- ____ Adaptation

What caused the change?

- ____ Requirements/specifications
- ____ Software design
- ____ Code
- ____ Previous change
- ____ Other

SECTION B: Change Implementation Information

Components Added/Changed/Deleted: _____

	< 1hr	1 hr to 1 day	1 day to 1 week	1 week to 1 month	> 1 month
Estimate effort spent isolating/determining the change:					
Estimate effort to design, implement, and test the change:					

Check all changed objects:

- ____ Requirements/Specifications Document
- ____ Design Document
- ____ Code
- ____ System Description
- ____ User's Guide
- ____ Other

If code changed, characterize the change (check most applicable):

- ____ Initialization
- ____ Logic/control structure (e.g., changed flow of control)
- ____ Interface (internal) (module to module communication)
- ____ Interface (external) (module-to-external communication)
- ____ Data (value or structure) (e.g., variable or value changed)
- ____ Computational (e.g., change of math expression)
- ____ Other (none of the above apply)

Estimate the number of lines of code (including comments): ____ added ____ changed ____ deleted

Enter the number of components: ____ added ____ changed ____ deleted

Enter the number of the added components that are: ____ totally new ____ totally reused ____ reused with modifications

6201G(39)-12

NOVEMBER 1991

Figure A-4. Maintenance Change Report Form

Personnel Resources Form

Name:_____

Project:_____ Date (Friday): _____

SECTION A: Total Hours Spent on Project for the Week:_____

SECTION B: Hours By Activity (Total of hours in Section B should equal total hours in Section A)

Activity	Activity Definitions	Hours
Predesign	Understanding the concepts of the system. Any work prior to the actual design (such as requirements analysis).	
Create Design	Development of the system, subsystem, or components design. Includes development of PDL, design diagrams, etc.	
Read/Review Design	Hours spent reading or reviewing design. Includes design meetings, formal and informal reviews, or walkthroughs.	
Write Code	Actually coding system components. Includes both desk and terminal code development.	
Read/Review Code	Code reading for any purpose other than isolation of errors.	
Test Code Units	Testing individual components of the system. Includes writing test drivers.	
Debugging	Hours spent finding a known error in the system and developing a solution. Includes generation and execution of tests associated with finding the error.	
Integration Test	Writing and executing tests that integrate system components, including system tests.	
Acceptance Test	Running/supporting acceptance testing.	
Other	Other hours spent on the project not covered above. Includes management, meetings, training hours, notebooks, system descriptions, user's guides, etc.	

SECTION C: Effort On Specific Activities (Need not add to A)
(Some hours may be counted in more than one area; view each activity separately)

Rework: Estimate of total hours spent that were caused by unplanned changes or errors. Includes effort caused by unplanned changes to specifications, erroneous or changed design, errors or unplanned changes to code, changes to documents. (This includes all hours spent debugging.) ☐

Enhancing/Refining/Optimizing: Estimate of total hours spent improving the efficiency or clarity of design, or code, or documentation. These are not caused by required changes or errors in the system. ☐

Documenting: Hours spent on any documentation of the system. Includes development of design documents, prologs, in-line commentary, test plans, system descriptions, user's guides, or any other system documentation. ☐

Reuse: Hours spent in an effort to reuse components of the system. Includes effort in looking at other system(s) design, code, or documentation. Count total hours in searching, applying, and testing. ☐

For Librarian's Use Only

Number: _____

Date: _____

Entered by: _____

Checked by: _____

NOVEMBER 1991

Figure A-5. Personnel Resources Form

Personnel Resources Form

(CLEANROOM VERSION)

Name:_____

Project:_____ Date (Friday): _____

SECTION A: Total Hours Spent on Project for the Week:_____

SECTION B: Hours By Activity (Total of hours in Section B should equal total hours in Section A)

Activity	Activity Definitions	Hours
Predesign	Understanding the concepts of the system. Any work prior to the actual design (such as requirements analysis).	
Pretest	Developing a test plan and building the test environment. Includes generating test cases, generating JCL, compiling components, building libraries, and defining inputs and probabilities.	
Create Design	Development of the system, subsystem, or components design. Includes box structure decomposition, stepwise refinement, development of PDL, design diagrams, etc.	
Verify/Review Design	Includes design meetings, formal and informal reviews, and walkthroughs.	
Write Code	Actually coding system components. Includes both desk and terminal code development.	
Read/Review Code	Code reading for any purpose other than isolation of errors. Includes verifying and reviewing code for correctness.	
Independent Test	Executing and evaluating tests of system components.	
Response to SFR	Isolating a tester-reported problem and developing a solution. Includes writing and reviewing design or code to isolate and correct a tester-reported problem.	
Acceptance Test	Running/supporting acceptance testing.	
Other	Other hours spent on the project not covered above. Includes management, meetings, training hours, notebooks, system descriptions, user's guides, etc.	

SECTION C: Effort On Specific Activities

Methodology Understanding/Discussion: Estimate the total hours spent learning, discussing, reviewing or attempting to understand cleanroom-related methods and techniques. Includes all time spent in training. ☐

For Librarian's Use Only
Number: _____
Date: _____
Entered by: _____
Checked by: _____

NOVEMBER 1991

Figure A-6. Personnel Resources Form (Cleanroom Version)

PROJECT COMPLETION STATISTICS FORM

Name: _____

Project:_____ Date: _____

Phase Dates (Saturdays)	
Phase	Start Date
Requirements Definition	
Design	
Implementation	
System Test	
Acceptance Test	
Cleanup	
Maintenance	
Project End	

Staff Resource Statistics	
Technical and Management Hours	
Services Hours	

Computer Resource Statistics		
Computer	CPU hours	No. of runs

Project Size Statistics			
General Parameters		Source Lines of Code	
Number of subsystems		Total	
Number of components		New	
Number of changes		Slightly Modified	
Pages of documentation		Extensively Modified	
		Old	
		Comments	

Executable Modules		Executable Statements		Statements	
Total		Total		Total	
New		New		New	
Slightly Modified		Slightly Modified		Slightly Modified	
Extensively Modified		Extensively Modified		Extensively Modified	
Old		Old		Old	

Note: All of the values on this form are to be actual values at the completion of the project. The values entered by hand by SEL personnel reflect the data collected by the SEL during the course of the project. Update these according to project records and supply values for all blank fields.

For Librarian's Use Only

Number:_____

Date:_____

Entered by:_____

Checked by:_____

6201G(39)-11

NOVEMBER 1991

Figure A-7. Project Completion Statistics Form

PROJECT ESTIMATES FORM

Name: _____

Project: _____ **Date:** _____

Phase Dates (Saturdays)	
Phase	**Start Date**
Requirements Definition	
Design	
Implementation	
System Test	
Acceptance Test	
Cleanup	
Project End	

Staff Resource Estimates	
Programmer Hours	
Management Hours	
Services Hours	

Project Size Estimates	
Number of subsystems	
Number of components	
Source Lines of Code	
Total	
New	
Modified	
Old	

Note: **All of the values on this form are to be estimates of projected values at completion of the project. This form should be submitted with updated estimates every 6 to 8 weeks during the course of the project.**

For Librarian's Use Only
Number: _____
Date: _____
Entered by: _____
Checked by: _____

6201G(13)-16

NOVEMBER 1991

Figure A-8. Project Estimates Form

PROJECT STARTUP FORM

Name: _____

Project: _____ **Date:** _____

PLEASE PROVIDE ALL AVAILABLE INFORMATION

Project Full Name: _____

Project Type: _____

Contacts: _____

Language: _____

Computer System: _____

Account: _____

Task Number: _____

Forms To Be Collected: (Circle forms that apply)

PEF PRF CLPRF DSF SPF SIF COF CCF CRF SEF PCSF WMEF MCRF

General Notes: _____

Personnel Names (indicate with * if not in database):

6201G(13)-36

NOVEMBER 1991

Figure A-9. Project Startup Form

SERVICES/PRODUCTS FORM

Project:_____

Date (Friday):_____

COMPUTER RESOURCES

Computer	CPU Hours	No. of Runs

GROWTH HISTORY

Components	
Changes	
Lines of Code	

SERVICES EFFORT

Service	Hours
Tech Pubs	
Secretary	
Proj Mgmt	
Other	

For Librarian's Use Only

Number:_____

Date:_____

Entered by:_____

Checked by:_____

6201G(13)-08

NOVEMBER 1991

Figure A-10. Services/Products Form

SUBJECTIVE EVALUATION FORM

Name: _____

Project: _____ Date: _____

Indicate response by circling the corresponding numeric ranking.

I. PROBLEM CHARACTERISTICS

1. Assess the intrinsic difficulty or complexity of the problem that was addressed by the software development.

   1 2 3 4 5
   Easy Average Difficult

2. How tight were schedule constraints on project?

   1 2 3 4 5
   Loose Average Tight

3. How stable were requirements over development period?

   1 2 3 4 5
   Loose Average High

4. Assess the overall quality of the requirements specification documents, including their clarity, accuracy, consistency, and completeness.

   1 2 3 4 5
   Low Average High

5. How extensive were documentation requirements?

   1 2 3 4 5
   Low Average High

6. How rigorous were formal review requirements?

   1 2 3 4 5
   Low Average High

II. PERSONNEL CHARACTERISTICS: TECHNICAL STAFF

7. Assess overall quality and ability of development team.

   1 2 3 4 5
   Low Average High

8. How would you characterize the development team's experience and familiarity with the application area of the project?

   1 2 3 4 5
   Low Average High

9. Assess the development team's experience and familiarity with the development environment (hardware and support software).

   1 2 3 4 5
   Low Average High

10. How stable was the composition of the development team over the duration of the project?

    1 2 3 4 5
    Loose Average High

FOR LIBRARIAN'S USE ONLY

Number: _____ Entered by: _____

Date: _____ Checked by: _____

6201G(13)-29

NOVEMBER 1991

Figure A-11. Subjective Evaluation Form (1 of 3)

SUBJECTIVE EVALUATION FORM

III. PERSONNEL CHARACTERISTICS: TECHNICAL MANAGEMENT

11. Assess the overall performance of project management.

1	2	3	4	5
Low		Average		High

12. Assess project management's experience and familiarity with the application.

1	2	3	4	5
Low		Average		High

13. How stable was project management during the project?

1	2	3	4	5
Low		Average		High

14. What degree of disciplined project planning was used?

1	2	3	4	5
Low		Average		High

15. To what degree were project plans followed?

1	2	3	4	5
Low		Average		High

IV. PROCESS CHARACTERISTICS

16. To what extent did the development team use modern programming practices (PDL, top-down development, structured programming, and code reading)?

1	2	3	4	5
Low		Average		High

17. To what extent did the development team use well-defined or disciplined procedures to record specification modifications, requirements questions and answers, and interface agreements?

1	2	3	4	5
Low		Average		High

18. To what extent did the development team use a well-defined or disciplined requirements analysis methodology?

1	2	3	4	5
Low		Average		High

19. To what extent did the development team use a well-defined or disciplined design methodology?

1	2	3	4	5
Low		Average		High

20. To what extent did the development team use a well-defined or disciplined testing methodology?

1	2	3	4	5
Low		Average		High

IV. PROCESS CHARACTERISTICS

21. What software tools were used by the development team? Check all that apply from the list that follows and identify any other tools that were used but are not listed.

- ☐ Compiler
- ☐ Linker
- ☐ Editor
- ☐ Graphic display builder
- ☐ Requirements language processor
- ☐ Structured analysis support tool
- ☐ PDL processor
- ☐ ISPF
- ☐ SAP
- ☐ CAT
- ☐ PANVALET
- ☐ Test coverage tool
- ☐ Interface checker (RXVP80, etc.)
- ☐ Language-sensitive editor
- ☐ Symbolic debugger
- ☐ Configuration Management Tool (CMS, etc.)
- ☐ Others (identify by name and function)

22. To what extent did the development team prepare and follow test plans?

1	2	3	4	5
Low		Average		High

6201G(13)-30

Figure A-11. Subjective Evaluation Form (2 of 3)

SUBJECTIVE EVALUATION FORM

IV. PROCESS CHARACTERISTICS (CONT'D)

23. To what extent did the development team use well-defined and disciplined quality assurance procedures (reviews, inspections, and walkthroughs)?

1	2	3	4	5
Low		Average		High

24. To what extent did development team use well-defined or disciplined configuration management procedures?

1	2	3	4	5
Low		Average		High

V. ENVIRONMENT CHARACTERISTICS

25. How would you characterize the development team's degree of access to the development system?

1	2	3	4	5
Low		Average		High

26. What was the ratio of programmers to terminals?

1	2	3	4	5
8:1	4:1	2:1	1:1	1:2

27. To what degree was the development team constrained by the size of main memory or direct-access storage available on the development system?

1	2	3	4	5
Low		Average		High

28. Assess the system response time: were the turnaround times experienced by the team satisfactory in light of the size and nature of the jobs?

1	2	3	4	5
Poor		Average		Very Good

29. How stable was the hardware and system support software (including language processors) during the project?

1	2	3	4	5
Low		Average		High

30. Assess the effectiveness of the software tools.

1	2	3	4	5
Low		Average		High

VI. PRODUCT CHARACTERISTICS

31. To what degree does the delivered software provide the capabilities specified in the requirements?

1	2	3	4	5
Low		Average		High

32. Assess the quality of the delivered software product.

1	2	3	4	5
Low		Average		High

33. Assess the quality of the design that is present in the software product.

1	2	3	4	5
Low		Average		High

34. Assess the quality and completeness of the delivered system documentation.

1	2	3	4	5
Low		Average		High

35. To what degree were software products delivered on time?

1	2	3	4	5
Low		Average		High

36. Assess smoothness or relative ease of acceptance testing.

1	2	3	4	5
Low		Average		High

6201G(13)-31

Figure A-11. Subjective Evaluation Form (3 of 3)

SUBSYSTEM INFORMATION FORM

Name:_____

Project: _____ Date: _____

Add New Subsystems

Subsystem Prefix	Subsystem Name	Subsystem Function

Change Existing Subsystems

Old Subsystem Prefix (Must exist in the database)	Action (R - Rename, D - Delete)	New Subsystem Prefix (Must not exist in the database)

This form is to be completed by the time of the Preliminary Design Review (PDR). An update must be submitted each time a new subsystem is defined thereafter. This form is also to be used when a subsystem is renamed or deleted.

Subsystem Prefix: A prefix of 2 to 5 characters used to identify the subsystem when naming components

Subsystem Name: A descriptive name of up to 40 characters

Subsystem Function: Enter the most appropriate function code from the list of functions below:

USERINT:	User Interface
DPDC:	Data Processing/Data Conversion
REALTIME:	Real-time Control
MATHCOMP:	Mathematical/Computational
GRAPH:	Graphics and Special Device Support
CPEXEC:	Control Processing/Executive
SYSSERV:	System Services

For Librarian's Use Only

Number: _____

Date:_____

Entered by:_____

Checked by:_____

6201G(13)-39

NOVEMBER 1991

Figure A-12. Subsystem Information Form

WEEKLY MAINTENANCE EFFORT FORM

Name: _____

Project: _____ **Date (Friday):** _____

For Librarian's Use Only

Number: _____

Date: _____

Entered by: _____

Checked by: _____

Section A – Total Hours Spent on Maintenance

(Includes time spent on all maintenance activities for the project excluding writing specification modifications)

Section B – Hours By Class of Maintenance

(Total of hours in Section B should equal total hours in Section A)

Class	Definition	Hours
Correction	Hours spent on all maintenance associated with a system failure.	
Enhancement	Hours spent on all maintenance associated with modifying the system due to a requirements change. Includes adding, deleting, or modifying system features as a result of a requirements change.	
Adaptation	Hours spent on all maintenance associated with modifying a system to adapt to a change in hardware, system software, or environmental characteristics.	
Other	Other hours spent on the project (related to maintenance) not covered above. Includes management, meetings, etc.	

Section C – Hours By Maintenance Activity

(Total of hours in Section C should equal total hours in Section A)

Activity	Activity Definitions	Hours
Isolation	Hours spent understanding the failure or request for enhancement or adaptation.	
Change Design	Hours spent actually redesigning the system based on an understanding of the necessary change.	
Implementation	Hours spent changing the system to complete the necessary change. This includes changing not only the code, but the associated documentation.	
Unit Test/ System Test	Hours spent testing the changed or added components. Includes hours spent testing the integration of the components.	
Acceptance/ Benchmark Test	Hours spent acceptance testing or benchmark testing the modified system.	
Other	Other hours spent on the project (related to maintenance) not covered above. Includes management, meetings, etc.	

6201G(39)-10

NOVEMBER 1991

Figure A-13. Weekly Maintenance Effort Form

Appendix B. Sample Process Study Plan

SEL Representative Study Plan for

SOHOTELS

October 11, 1993

1. Project Description

The Solar and Heliospheric Observatory Telemetry Simulator (SOHOTELS) software development project will provide simulated telemetry and engineering data for use in testing the SOHO Attitude Ground Support System (AGSS). SOHOTELS is being developed by a team of four GSFC personnel in Ada on the STL VAX 8820. The project is reusing design, code, and data files from several previous projects but primarily from the Solar, Anomalous, and Magnetospheric Particle Explorer Telemetry Simulator (SAMPEXTS).

The SOHOTELS team held a combined preliminary design review (PDR) and critical design review (CDR) in April 1993. In their detailed design document, the SOHOTELS team stated the following goals for the development effort:

- To maximize reuse of existing code
- Where reuse is not possible, to develop code that will be as reusable as possible
- To make sure performance does not suffer when code is reused

2. Key Facts

SOHOTELS is being implemented in three builds so that it can be used to generate data for the early phases of the AGSS (which is a Cleanroom project). Build development and independent acceptance testing are being conducted in parallel. At present, the test team has finished testing SOHOTELS Build 1. The development team expects to complete Build 2 and deliver it to the independent test team by the end of the week.

SOHOTELS consists of six subsystems. As of June, the estimated total number of components was 435, of which 396 (91 percent) have currently been completed. Total SLOC for SOHOTELS was estimated at 67.6K SLOC, with 46.6K SLOC of code to be reused verbatim and 15.7K SLOC to be reused with modifications. As of September 13, 1993, there were 65.4K SLOC in the SOHOTELS system, or 97 percent of the estimated total.

The SOHOTELS task leader is currently re-estimating the size of the system because SOHOTELS will be more complex than was originally predicted. The new estimates will also include SLOC for the schema files that are being developed.

The phase start dates for SOHOTELS are

September 9, 1992	Requirements Definition
October 3, 1992	Design
May 1, 1993	Code and Unit Test

June 26, 1993	Acceptance Test
May 7, 1993	Cleanup

3. Goals of the Study

The study goals for SOHOTELS are

- To validate the SEL's recommended tailoring of the development life cycle for high-reuse Ada projects
- To refine SEL models for high-reuse software development projects in Ada, specifically
  - Effort (per DLOC, by phase and by activity)
  - Schedule (duration for telemetry simulators and by phase)
  - Errors (number per KSLOC/DLOC)
  - Classes of errors (e.g., initialization errors, data errors)
  - Growth in schedule estimates and size estimates (from initial estimates to completion and from PDR/CDR to completion)

4. Approach

The following steps will be taken to accomplish the study goals:

- Understand which of the standard development processes are being followed (per Reference 10) and which have been tailored for the SOHOTELS project. Ensure that information is entered into the SEL database that will allow SOHOTELS data to be correctly interpreted in light of this tailoring.
- Analyze project/build characteristics, effort and schedule estimates, effort and schedule actuals, and error data on a monthly basis while development is ongoing.
- At project completion, plot the effort, schedule, error rate, and estimate data. Compare these plots with current SEL models and with plots from other high-reuse projects in Ada. Compare and contrast the error-class data with data from FORTRAN projects, from Ada projects with low reuse, and from other high-reuse Ada projects.

5. Data Collection

To address these study goals, the following standard set of SEL data for Ada projects will be collected:

- Size, effort, and schedule estimates (Project Estimates Forms)
- Weekly development effort (Personnel Resources Forms)
- Growth data (Component Origination Forms and SEL librarians)
- Change and error data (Change Report Forms and SEL librarians)

Appendix C. List of Rules

Abbreviations and Acronyms

AGSS	attitude ground support system
CASE	computer-aided software engineering
CDR	critical design review
CM	configuration management
CMM	Capability Maturity Model
Code Q	Office of Safety and Mission Assurance (NASA)
COTS	commercial off-the-shelf
CPU	central processing unit
DBMS	database management system
DLOC	developed lines of code
GSFC	Goddard Space Flight Center
IV&V	independent verification and validation
JSC	Johnson Space Center
KDLOC	1,000 developed lines of code
KSLOC	1,000 source lines of code
NASA	National Aeronautics and Space Administration
PDR	preliminary design review
QA	quality assurance
R&D	research and development
SAMPEXTS	Solar, Anomalous, and Magnetospheric Particle Explorer Telemetry Simulator
SEI	Software Engineering Institute
SEL	Software Engineering Laboratory
SLOC	source lines of code
SME	Software Management Environment
SOHOTELS	Solar and Heliospheric Observatory Telemetry Simulator

References

1. Grady, R. B., and Caswell, D. L., Software Metrics: Establishing a Company-Wide Program. Prentice-Hall Inc., Englewood Cliffs, NJ, 1989.

2. NASA, DA3 Software Development Metrics Handbook, Version 2.1, JSC-25519, Office of the Assistant Director for Program Support, Mission Operations Directorate, Johnson Space Center, Houston, April 1992.

3. ———, DA3 Software Sustaining Engineering Metrics Handbook, Version 2.0, JSC-26010, Office of the Assistant Director for Program Support, Mission Operations Directorate, Johnson Space Center, Houston, December 1992.

4. ———, DA3 Development Project Metrics Handbook, Version 5.0, JSC-36112, Office of the Assistant Director for Program Support, Mission Operations Directorate, Johnson Space Center, Houston, March 1993.

5. Musa, J. D., Iannino, A., and Okumuto, K., Software Reliability: Measurement, Prediction, Application, McGraw-Hill, New York, 1987.

6. Rifkin, S., and Cox, C., Measurement in Practice, CMU/SEI-91-TR-16, Software Engineering Institute, Carnegie Mellon University, August 1991.

7. Daskalantonakis, M. K., "A Practical View of Software Measurement and Implementation Experiences With Motorola," IEEE Transactions on Software Engineering, Volume SE-18, November 1992.

8. Decker, W., Hendrick, R., and Valett, J., Software Engineering Laboratory Relationships, Models, and Management Rules, SEL-91-001, Software Engineering Laboratory, NASA/GSFC, February 1991.

9. Condon, S., Regardie, M., Stark, M., and Waligora, S., Cost and Schedule Estimation Study Report, SEL-93-002, Software Engineering Laboratory, NASA/GSFC, November 1993.

10. Landis, L., McGarry, F., Waligora, S., et al., Manager's Handbook for Software Development (Revision 1), SEL-84-101, Software Engineering Laboratory, NASA/GSFC, November 1990.

11. Paulk, M. C., Curtis, B., Chrissis, M. B., and Weber, C. V., Capability Maturity Model for Software, Version 1.1, CMU/SEI-93-TR-24, Software Engineering Institute, Carnegie Mellon University, February 1993.

12. McGarry, F., and Jeletic, K., "Process Improvement as an Investment: Measuring Its Worth," Proceedings of the Eighteenth Annual Software Engineering Workshop, SEL-93-003, NASA/GSFC, December 1993.

13. Currit, P. A., Dyer, M., and Mills, H. D., "Certifying the Reliability of Software," IEEE Transactions on Software Engineering, Vol. SE-12, No. 1, January 1986, pp. 3–11.

14. Basili, V. R., and Green, S., "Software Process Evolution at the SEL," IEEE Software, Vol. 11, No. 4, July 1994, pp. 58–66.

15. Rombach, H. D., Ulery, B. T., and Valett, J. D., "Toward Full Life Cycle Control: Adding Maintenance Measurement to the SEL," Journal of Systems and Software, Vol. 18, 1992, pp. 125–138.

16. Caldiera, G., McGarry, F., Waligora, S., Jeletic, K., and Basili, V. R., Software Process Improvement Guidebook, NASA-GB-002-94, Software Engineering Program, 1994.

17. International Function Point Users Group, Function Point Counting Practices Manual, Release 3.2, Westerville, Ohio, 1991.

18. McGarry, F., "Experimental Software Engineering: Seventeen Years of Lessons in the SEL," Proceedings of the Seventeenth Annual Software Engineering Workshop, SEL-92-004, NASA/GSFC, December 1992.

19. Heller, G., Valett, J., and Wild, M., Data Collection Procedures for the Software Engineering Laboratory Database, SEL-92-002, Software Engineering Laboratory, NASA/GSFC, March 1992.

20. Decker, W. and Valett, J., Software Management Environment (SME) Concepts and Architecture, SEL-89-003, Software Engineering Laboratory, NASA/GSFC, August 1989.

21. Hall, D., Sinclair, C., and McGarry, F., Profile of Software at the Goddard Space Flight Center, NASA-RPT-002-94, Software Engineering Program, April 1994.

22. Basili, V. R., and Perricone, B. T. "Software Errors and Complexity: An Empirical Investigation," Communications of the ACM, January 1984, Vol. 27, No. 1.

23. Basili, V. R., and Weiss, D. M. "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, November 1984.

24. Basili, V. R., and Rombach, H. D. "The TAME Project: Towards Improvement-Oriented Software Environments," IEEE Transactions on Software Engineering, June 1988.

25. Caldiera, G., Jeletic, K., McGarry, F., Pajerski, R., et al., Software Process Improvement Guidebook, NASA-GB-001-95, Software Engineering Program, 1995.

26. Kelly, J. C., Sherif, J. S., and Hops, J., "An Analysis of Defect Densities Found During Software Inspections," Journal of Systems and Software, Vol. 17, No. 2, February 1992.