Technical Report
CMU/SEI-92-TR-11
ESD-TR-92-11

Software Measurement Concepts for Acquisition Program Managers

Developed by the

Software Acquisition Metrics Working Group

Prepared by

James A. Rozum

June 1992

Technical Report
CMU/SEI-92-TR-11
ESD-TR-92-11
June 1992

Software Measurement Concepts for Acquisition Program Managers

Developed by the

Software Acquisition Metrics Working Group

Prepared by
James A. Rozum

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

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

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

FOR THE COMMANDER

(signature on file)

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

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

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

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

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

This report was prepared for the

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

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

FOR THE COMMANDER

(signature on file)

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

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

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

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

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

This report was prepared for the

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

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

FOR THE COMMANDER

(signature on file)

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

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

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

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

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

Table of Contents

List of Figures

Acknowledgements

The SEI measurement efforts have depended on the participation of many people. The Software Process Measurement Project thanks the members of the Software Acquisition Metrics Working Group who contributed to the content and structure of this document. The SEI is indebted to them and to the organizations who sponsored their participation in this effort to incorporate measurement into the practices of acquisition program managers. Without their participation, the SEI could not have completed this task. The members of the Software Acquisition Metrics Working Group are:

A. Frank Ackerman
Institute for Zero Defect Software

Mark Amaya
McDonnell Douglas Corporation

Miguel Carrio
Teledyne Brown Engineering

Andrew Chruscicki
USAF Rome Laboratory

Charles Cox
US Naval Weapons Center

Joseph P. Dean
Tecolote Research, Inc.

Peter Dyson
Software Productivity Solutions

Stewart Fenick
US Army Communication and Electronics Command

Ronald Gulezian
Drexel University

Capt. Marc L. Hoffman
USAF - Air Force Systems Command

Steve Keller
Dynamics Research, Inc.

Jim Keng
McDonnell Douglas Corporation

Jeffrey Lasky
Rochester Institute of Technology

Janet MacLaughlin
US Naval Ocean Systems Center

John J. McGarry
US Naval Underwater Systems Center

Alfred H. Peschel
TRW, Systems Development Division

Sam T. Redwine, Jr.
Software Productivity Consortium

Alan J. Roberts
Defense Systems Management College

James A. Rozum
Software Engineering Institute

Herman P. Schultz
MITRE Corporation

Raghu P. Singh
US Navy Space and Warfare Command

William Spaulding
Dynamics Research, Inc.

The SEI Software Process Measurement Project thanks all the individuals who reviewed the October 1991 draft and took the time and care to provide written comments. Those comments were very important in transforming the draft into this technical report. People who provided comments are:

Emanuel Baker
Software Engineering Consultants, Inc.

Lyle Cocking
General Dynamics

Barry Corson
US Naval Air System Command

Dean Dubofsky
MITRE Corporation

Stewart Fenick
US Army Communication and Electronics Command

Paul Funch
MITRE Corporation

John Harding
Groupe Bull

Donna Hinrichs
US.Air Force

Theresa Huber
DSD Laboratories, Inc.

Charles Koch
US Naval Air Development Center

Harry Larson
Larbridge Enterprises

Donna Lindskog
University of Regina and SaskTel

John Marciniak
National Security Industries Association

Richard Murphy
Software Engineering Institute

Jacquelyn Nixon
US Marine Corps (MCTSSA)

Don O'Neill
O'Neill Consultants

Anne Quinn
DCMDM-EE

Donald Reifer
Reifer Consultants, Inc.

Norman Schneidewind
Naval Post Graduate School

Mark Servello
American Management Systems, Inc.

Terry Wilcox
Defense Plant Representative Office - General Dynamics

The project also thanks the members of the Measurement Steering Committee for their many thoughtful contributions. Their insight and advice have been invaluable. This committee consists of the following representatives from industry, government, and academia who have earned solid national and international reputations for their contributions to measurement and software management:

William Agresti
MITRE Corporation

Henry Block
University of Pittsburgh

David Card
Computer Sciences Corporation

Andrew Chruscicki
USAF Rome Laboratory

Samuel Conte
Purdue University

Bill Curtis
Software Engineering Institute

Joseph P. Dean
Tecolote Research

Stewart Fenick
US Army Communication and Electronics Command

John Harding
Groupe Bull

Frank McGarry
NASA Goddard Space Flight Center

John J. McGarry
US Naval Underwater Systems Center

Watts Humphrey
Software Engineering Institute

Richard Mitchell
US Naval Air Development Center

John Musa
AT&T Bell Laboratories

Alfred H. Peschel
TRW, Systems Development Division

Marshall Potter
Department of the Navy

Samuel T. Redwine, Jr.
Software Productivity Consortium

Kyle Rone
IBM Corporation

Seward (Ed) Smith
IBM Corporation

Norman Schneidewind
Naval Post Graduate School

Herman P. Schultz
MITRE Corporation

Robert Sulgrove
NCR Corporation

Ray Wolverton
Hughes Aircraft

As the members of the Software Acquisition Metrics Working Group prepared this report, they were aided in their activities by the able and professional support staff of the SEI. Special thanks are owed to Mary Beth Chrissis and Suzanne Couturiaux, who were instrumental in getting the report into final form; to Marcia Theoret and Lori Race of the Software Process Measurement Project, who coordinated many of our meeting activities and provided outstanding secretarial services; and to Helen Joyce and her assistants from Event Management, who so competently assured that meeting rooms, lodgings, and refreshments were there when we needed them.

Software Measurement Concepts for Acquisition Program Managers

Abstract. For program managers to effectively manage and control software development, they need to incorporate a measurement process into their decision making and reporting process. Measurement costs money, but it can also save money through early problem detection and objective clarification of critical software development issues. This report provides some basic concepts that program managers can use to help integrate measurement into the process for managing software development. It also provides an initial set of measures to help address common issues in software intensive acquisitions.

When the Software Acquisition Metrics Working Group first met in 1989, only a few reports existed on the subject of how program managers could use software measurement; now, other reports have been written. The goal of this report is not to compete with those reports, but to use them as starting points for expansion. This report should be viewed not as a standard but as containing guidelines and advice for program officers and managers starting to use software measurement in their own organizations.

1. Executive Summary

1.1 Why Should Acquisition Program Managers Use Software Measurement?

Software contracts are difficult to manage. There are many complex inter-relationships between the factors, assumptions, estimates, and unknowns during the planning and execution of software contracts. Contributing equally to the difficulty of managing the contracts are ever changing constraints, technology, and acquisition environments. These all contribute to the difficulty of making good, timely decisions that guide a program toward successful completion. With software measurement, program managers have information that leads to early insight of potential problems as well as information that can support decisions. Without software measurement, program managers do not have the insight to identify and resolve problems, and as a result, they make decisions to react to events and problems rather than to curtail those problems before they surface.

Managers use measurement as feedback to control their projects. Without measurement, there is usually no way to know the status of a project along its many dimensions: cost, schedule, product performance, supportability, quality, etc. Thus, it is difficult to know what actions to take, what decisions to make, or how to correct unexpected outcomes. To benefit from measurement, managers need to link software measurement to the goals and risks of the program [WALTON].

> "…You cannot just use numbers to control things. The numbers must properly represent the process being controlled, and they must be sufficiently well defined and verified to provide a reliable basis for action. While process measurements are essential for orderly improvement, careful planning and preparation are required or the results are likely to be disappointing." [HUMPHREY]

The successful application of software measurement depends on having well-established measurement goals. A software measurement process that identifies the issues and uses them to identify what data and measurements to collect will provide better insight into the key program issues and help program managers make informed decisions. No single measurement will meet a program manager's needs [BROOKS87]. Therefore, the program manager needs a framework for defining what data to collect and how to analyze the data after it is collected.

> "The data collection process must be driven by the information from questions that we formulate based on our needs. In short, know what question is to be answered before collecting data." [JURAN]

A myth exists that program managers can just collect data and plot graphs to form an effective measurement process. Although possibly true in a static, precedented software development environment, this is not the case for the majority of U.S. Department of Defense (DoD) software acquisitions. The measurement process must be dynamic because of the constantly changing issues and complexities in DoD contracting. To be effective for the program manager, the measurement process must be a complete Plan-Do-Check-Act (PDCA) process that is integrated into the program office structure and used as a constructive tool for communicating between the program office and the contractor [SHEWART]. The PDCA cycle that the program manager uses is described as follows:

Plan: Identify the issues or questions the program manager has and then determine the data and measures to be collected to address them.

Do: Collect data and, based on the issues identified and using baseline data, derive graphical representations (i.e., indicators) to better illustrate the data trends.

Check: Analyze the trends, graphs, data, etc. to better understand the issues and the performance towards their resolutions.

Act: Report the results, recommend improvements, and identify new issues and questions.

1.2 Organization of This Report

In general, the concepts and measures defined in this report are designed to benefit the program manager by providing insight into, and hence control over, a software contract's resources, progress, and technical issues. To effectively use the concepts and measures, the program manager will require personnel skilled in software engineering and development and in the acquisition process and environment.

The data definition and collection concepts provided in Chapter 2 of this report let program managers define their own goals, identify what they believe to be the issues and risks, and define what questions they want the measurement to help answer. The set of commonly used software measures provided in Chapter 3 helps acquisition program managers address issues that are common to most software contracts. An overview of measurement analysis techniques provided in Chapter 4 helps program managers better analyze the data they have collected.

1.3 Purpose and Audience

The primary objective in publishing this report is to accelerate the use of software measurement in the acquisition process by providing:

1) Measurement process concepts that support the implementation of software measurement in DoD programs.

2) A set of metrics that are useful in addressing common software development issues.

This report is written for DoD program managers who are responsible for managing and making decisions regarding a software contract. It is expected that the program manager is knowledgeable about and regularly uses DOD-STD-2167A [DOD2167A]. However, this report can be adapted and used by other government agencies or persons who have similar contract monitoring and oversight duties. In addition, contractors can use the process and measures described in this report to support their own management and development processes.

1.4 Scope

This report provides the basic concepts needed to start a software measurement process. The process can then be tailored and further elaborated to support a specific contract or program office. Follow-on efforts may include a detailed software measurement process for acquisition program offices including guidance on how to contractually implement software measurement.

The software measures described in this report are intended for use during the engineering and manufacturing phase of the system acquisition life cycle [DOD5000.1]; however, this does not preclude adapting them for use in earlier phases such as demonstration and validation or during later phases such as operational support. It also does not preclude adapting the measures for measuring how the process responds when applying process improvements or new technologies within a domain, e.g., reuse, object oriented design, etc.; nor does it preclude using the measures to guide and manage programs determined to already be in trouble.

1.5 Software Measurement Process Concepts

A software measurement process is a systematic method of measuring, assessing, and adjusting the software development process using objective data. Within such a systematic software measurement process, software data is collected based on a known or anticipated development issue, concern, or question. The data is analyzed with respect to the characteristics of the software development process and products, and used to assess progress, quality, and performance throughout the development. There are four key components to an effective measurement process:

- Defining clearly the software development issues and the software measures (data elements) that support insight to the issues.
- Processing the software data into graphs and tabular reports (indicators) that support issue analysis.
- Analyzing the indicators to provide insight into the issues.
- Using the results to implement improvements and identify new issues and questions.

Each of these components is driven by the issues and characteristics inherent to the program. For example, the decision of what specific data to collect is based on the list of issues to be addressed by measurement; the indicators developed from the measurement data are flexible and tailored by the development issues and related questions; and analysis techniques are chosen based on what information is desired and what question(s) need to be answered. Analysis can be directed at assessing the feasibility of development plans, identifying new issues, tracking issue improvement trends, projecting schedules based on performance to date, defining possible development tradeoffs, and evaluating the consistency and quality of development activities and products.

The measurement process and its components are implemented on an ongoing basis throughout the development. The defined data parameters are measured, processed, and analyzed within a flexible assessment structure. This allows for the measurement process to adjust to changing development activities and products throughout the life cycle. The key is that the measurement process is driven by the program's issues.

The government program office implements its own measurement process independent of the contractor. This includes collecting or receiving the actual data, not just the graphs or indicators. The government program office must use the data to perform its own analysis, independent of the contractor, then meet with the contractor to discuss and compare analysis results. This independence will foster the government's confidence in the data and a better understanding of the issues. This measurement process implies that the government program manager is actively involved in managing the contract.

For the software measurement process to be effective, it needs to be an integral part of the program manager's decision-making process. To accomplish this, insights gained from the measurements should be combined with program knowledge from other sources in the conduct of daily program management activities. It is the overall measurement process that adds value to the data for the program manager, not just the graphs or reports. The following steps are necessary to integrate the measurement into daily program management activities:

1. Collect measurements that are targeted to specific program issues and that will aid in understanding and improving the technical and management processes.

2. Use the measures together with other management information to improve insight into progress and risks. For example, when addressing schedule risk, look at related analyses of staffing and schedule progress.

3. Establish means to investigate both management and technical issues further. The software measurement process is often the starting point for obtaining insight into the entire program. To isolate specific causes of trends apparent from the measures, it is often necessary to ask additional questions. The value of the software measurement process in such cases is in identifying the questions that need to be asked.

4. Identify and implement changes to improve the program, including any needed improvements to the measurement process.

The measurement process needs to be consistent, yet flexible enough to address changes since they occur in all phases of software acquisitions. Change in the acquisition process is inevitable as the development proceeds and requirements become better defined: cost, performance, and schedule estimates are refined; personnel, machine, test, and support resource requirements are identified; and insights from the measures themselves are gained. As the issues change, the program manager should adapt the measurement process to address the changes. However, definitions of key data elements (e.g., lines of code) should be consistently used. The proposed software measurement process is different from a static approach by:

- Separating issues about the data (e.g., accuracy or timeliness) from the program issues by carefully defining and validating the data.
- Interpreting the data by using knowledge of the data, related program issues and risks, corrective action plans, and industry-wide software engineering experience (e.g., models or databases).
- Responding to changing program issues and needs during development by enhancing the measurements and data by iterative data collection and analyses.
- Producing end-item products from the measurement process (i.e., the indicators and accompanying analysis) that are objective and explainable.

To summarize, a software measurement process should:

- Provide insight into known software issues and identify new software issues.
- Use a consistent methodology that allows the software measurements to be objectively applied and evaluated throughout the program's life cycle.
- Allow the measures to be modified as the program's products and activities evolve.
- Enhance the program office's technical and program management processes through the use of quantitative data.

1.6 Common Program Issues

The measures provided in Chapter 3 can be used during the entire software development process. They were chosen because they seem to be the most useful measures, of those commonly accepted and used, to help the program manager address issues common to all software programs. As with the development of other system components, the key issues for software development relate to resource adequacy and expenditure, development progress and schedule, and the quality of the interim and final products. The nature of software development results in these three areas being strongly related. Issues with software quality, for example, can in many cases be directly attributable to schedule and resource constraints. This requires that the software measurement process address the relationships between several types of software data.

Issues related to the adequacy and expenditure of resources include:

- Staff availability and stability
- Funding adequacy
- Spending rate
- Allocation of resources to key activities (e.g., documentation, configuration management, or testing)
- Productivity rate assumptions
- Size estimate accuracy
- Subcontractor allocations
- Computer resource adequacy

The progress issue is simple; will the contract be completed on schedule, and if not, when will it be completed? Questions about the progress that program managers typically have include:

- Schedule (milestone commitments); is the contractor meeting commitments on time and within a prescribed level of quality?
- Development progress; what is the true (objective) progress of the contract?
- Schedule and forecasts to completion; are they realistic?
- Size estimates and forecasts; are they realistic?
- Functionality allocations; are they shifting from earlier to later builds?
- Productivity rate; is the planned productivity rate being achieved?
- Rework; are high levels of rework having an impact on progress?

The technical quality issues include both the quality of products being produced and the quality of the processes used to produce the products.` Here the program manager is concerned that interim products are stable and complete so that successive processes using those products do not compound mistakes (i.e., defects) and thereby drive up cost through added rework. Questions about the technical quality that program managers typically have revolve around issues regarding:

- Utilization of target computer resources, e.g., spare capacities and throughput.
- Problem reports, e.g., severity of problems, number of problems, timely resolution of problems, and high density levels of problems in products or processes.
- Completeness, e.g., product and process exit criteria.
- Product stability and volatility, e.g., requirements, design, and functionality allocations.
- Process stability and volatility, e.g., procedures and standards conformance.
- Rework, e.g., later processes finding defects caused by earlier processes.
- Defect rate, e.g., excessive defect rates could effect future progress.

The basic measurements discussed in Chapter 3 can be tailored by defining and collecting data that help address the issues identified as key to the program being managed. The measurements further described in Chapter 3 that have been found useful in addressing the common program issues are:

- **Software size:** Alerts the program manager to changes in the estimate(s) and/or actual software size. As the size grows beyond what was expected, the probability of the schedule and therefore the budget being exceeded also increases.

- **Effort:** Tracks the staff hours being expended by the contractor during the various development activities and for the entire project. Software development is a human intensive and dependent activity, making the effort expended the largest and least controllable cost variable [BROWN]. The effort measurements are particularly useful because it alerts the program manager to changes to and deviations from the plan that could drive the cost up and cause the budget to be exceeded.

- **Staff:** Gives the program manager insight into whether or not the contractor has a staff that is stable and able to do the job. The contractor's staff is tracked according to predefined and agreed upon labor categories. Also tracked are the losses (staff turnover) and additions of staff by labor category. Losses that result in a higher than expected staff turnover for the project may jeopardize the project's quality and expected productivity rates because replacement staff must be trained to become familiar with the software being developed and with the decisions that have been made.

- **Milestone performance:** Tracks the contractor's performance toward meeting commitments to complete activities and their formal milestones and interim events. If interim schedule commitments are met, the whole project is more likely to stay on schedule.

- **Development progress:** Tracks the progress of the development by quantifying and tracking the work completed. Work items should have predefined entry and exit criteria to determine when they are started and completed. The program manager counts those items that have been completed, knows how many need to be completed, and, therefore, has a quantitative method to determine how much work remains. This is done for each phase of the project by quantifying the work to be completed that produces interim products.

- **Software defects:** Tracks the evolving quality of the products as measured by the number of closed and open defects. Defects can be tracked by product (e.g., errors discovered during testing or peer reviews) or by process activity (e.g., action items from reviews or comments from a document review). A defect can be recorded and tracked against anything that would cause the contractor to rework an item that has passed through its exit criteria. Using this measure, the program manager can help determine the level of resources needed, the progress made, and the technical quality of the software and the processes used to develop it, as well as have an early indication of the requirements to support the software after it is delivered.

- **Computer resource utilization:** Gives the program manager early insight into whether or not the upper limits for computer resources will be exceeded. For example, the software may have performance requirements and may also need to allow for future expansion all within a known hardware configuration.

The intent is to provide measurements that give insight into issues. An argument can easily be made that each measure above provides insight into each of the three common issues (resources, progress, and technical quality). Figure 1-1 shows the issue to measurement relationships discussed in detail in Chapter 3. From Figure 1-1, a program manager can determine what measures are needed to give insight into his or her issue. The exception is the software defects measurement that will indicate levels of required rework, thus affecting the resources and progress issues, and can also indicate the technical quality of products and processes.

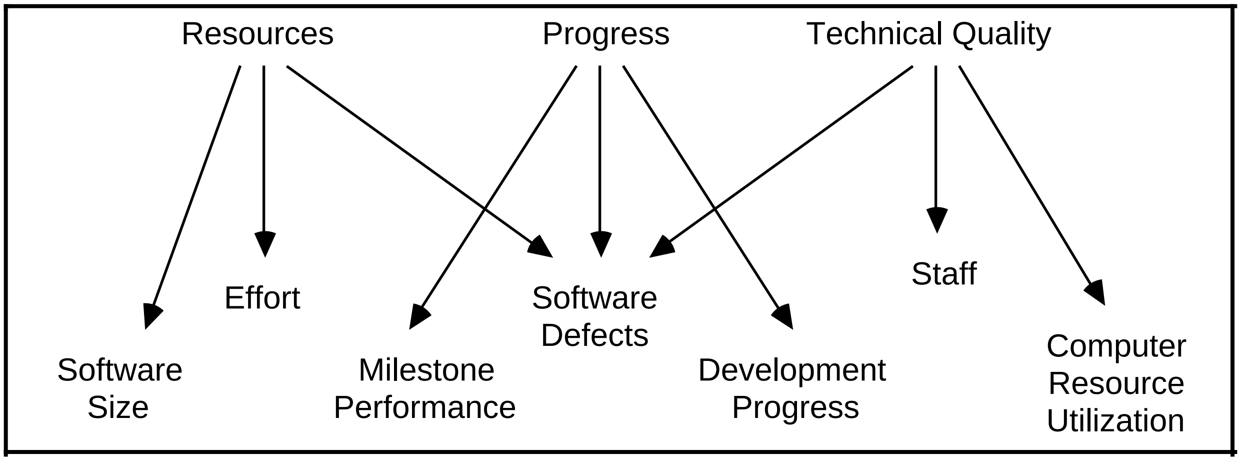


Figure 1-1 Cross Reference of Metrics to the Issues They Support

1.7 Constraints and Limitations

Software measures are valuable tools for gaining management and technical insight into a software program; however, they are not a panacea. To implement a software measurement system effectively, program managers must be aware of the following constraints and limitations associated with the application of software measurement:

- **Measurements are used as indicators, not as absolutes.** There is a strong temptation to seek absolute answers from measures. The role of measurement is to provide insights into the software development, which are based on objective data that might not otherwise be gained or be as timely. Often the measures prompt additional questions and insights that are not directly apparent from the measures themselves. For example, the manager may want to determine why the staff level is below what was planned. Perhaps there is some underlying program issue or perhaps the plan was inappropriate.

  This leads to a corollary constraint: Measurement cannot be applied in a vacuum. Insights gained through analysis of measures must be combined with program-specific knowledge to reach the correct conclusion.

- **Evaluations based on measurement are only as good as the input data.** Quality of the input data can be measured in three ways: timeliness, consistency, and accuracy. If the data isn't timely, it is of little use for making decisions associated with the current program status. Even If the data is timely, the data elements must be consistently defined and accurately collected. A deficiency in either of these areas can skew the measurements derived from the data and thus lead to false conclusions.

This leads to the following corollary constraint: Measures are representative of the software development process that produces them; i.e., the more mature the software development process, the more advanced the measurement process. A well-managed program with a well-defined data collection process that is an integral part of the developer's overall process will provide better data than a less mature software development and collection process.

- **Measurement must be understood to be used and be of value.** Measurement, like any other management or technical tool, must be understood by the program managers. This means understanding what the low-level measurement data represent and what the intentions are of the overall measurement process. The program manager must also learn when to look beyond the data and measurement process to understand what is really going on. For example, if there is a sharp decrease in problems being discovered at the same time there is a large increase in problem resolution and close-out, the initial conclusion might be that the number of latent problems is decreasing. However, in an environment constrained by labor resources, it is equally likely that the problem discovery rate dropped because engineering resources were moved from software problem detection (e.g., testing) to software problem correction.

- **Measurement should not be used against the contractor (or other organizations).** The measurement process requires a team effort. While it is necessary to impose contractual controls to implement software measurement on a contract, it is important to avoid making measurement an adversarial issue with the contractor. The contractor will be sensitive to the program manager's use of the measures to quantify its performance and will resist supporting the measurement process if the results are used against it. Instead, the measures should serve as the basis for interactive resolution of development process concerns. Measurement should be used as a problem identification and resolution tool targeted toward making the processes and products better. While measures may deal with personnel and organization data, use of this data should focus on constructive process-oriented decision making rather than blaming specific organizations or individuals.

- **Measurement cannot identify, explain, and predict everything.** An effective measurement process can identify and help explain many software program anomalies and can identify trends that support forecasting of performance. However, measurement cannot identify every potential problem nor explain every situation. It would not be practical or cost-effective to attempt a measurement process that tried to quantitatively characterize every aspect of a software program.

- **The measurement process cannot be exclusively done by the contractor.** When initially using measurement, program managers may be tempted simply to impose requirements on the contractor to collect data, generate and analyze measures, and deliver completed measurement graphs and reports. In the ideal scenario, a contractor already has a measurement process in place. So why should the program office get involved? For three reasons: 1) the measurement process is iterative; not all desired measures and display formats can be predetermined since issues and problems vary throughout the program's life cycle; 2) the natural tendency of the contractor will be to present the program in the best possible light; independent government analysis of the data is required to avoid possible misrepresentation of the program; and 3) the measurement process needs to be issue driven, and the contractor and government program office will have inherently different issues.

- **Causal use of direct comparisons of programs should be avoided.** Because no two programs are alike, it is inappropriate to simply use data from past programs as the estimates for current programs. Program managers can, however, characterize their programs by using past, similar programs to determine the realism of projected plans and costs. In this way, historical databases and other types of program comparisons can be used to help develop realistic estimates.

- **A single measurement should not be used.** No single measure can give the insight needed to answer or address all program issues. Most issues will require multiple (and seemingly unrelated) data items to characterize the issue. This, and the fact that measures are interrelated, implies that a program manager needs to correlate trends across measures.

2. Data Definition and Collection

The basis of an effective software measurement process is to get quantified, low-level data so that 1) software development issues can be identified and clarified early enough to adjust plans and mitigate problems, and 2) progress can be tracked against plans. The concepts below will help guide program managers in meeting these objectives.

2.1 Deciding Which Measures Are Required

The question of what data to collect and how to define that data is driven by the program manager's current and projected software issues and characteristics of the software development process and products. Chapter 3 gives a common set of software measures to consider. Those measures and suggested data inputs were chosen based on issues and problems that are common to all software developments.[1] To tailor or expand the set, the program office needs to identify issues not addressed by the measurement set, combine the issues into a common list, set the priorities for the issues, and determine which of the issues can be addressed with the software measurement process. The program manager can then determine what data is needed to support insight into the issues. However, not all issues can be predetermined or projected; therefore, the program manager also needs to include provisions that allow the process to be flexible so that it can be modified to provide insight into unforecasted issues.

Once the issues are identified, the program manager and the contractor must agree on definitions of the entry and exit criteria for the process and products, all data inputs, the standards for acceptance, the schedule and progress estimation methods, the collection methods, etc. before awarding the contract. For example, the program manager and the contractor must agree on the definition of source lines of code (SLOC) and how and when the SLOC will be estimated or counted (or both when applicable). This entire process and all of its decisions and agreements should then be written into the contract.

Once the data collection has started, the definitions should not be changed. The most important data concept is that of consistency in the definitions. Changing a definition after the data collection has started produces variations in the trends that could confound the analyses and camouflage performance or related problems. However, sometimes definitions will change. When they do, it becomes critically important that the government program manager understands the change and how the change affects data that has already been collected.

When determining what data to collect, the preferred data is that which is a direct result of the contractor's process. For example, count the number of computer software units to be

[1] Many studies and reports have documented the problems and issues regarding software development, including [USAF], [SEI89], [DEMARCO], [BROOKS82], [CONGRESS], and [PACKARD].

developed by looking in the preliminary design documentation. Limit the use of data that is below the granularity supported by the contractor's process. For example, when collecting software defect data and accompanying process information about the defects, use the contractor's process. To do otherwise might not be cost effective and may adversely affect the productivity and schedule of the contract. As the development progresses, new techniques, tools, etc. might emerge. Tailor the data to these techniques, but as the data collected changes, also change your understanding of what is included in the data. The data collected should be that which is a natural result of the process used. Avoid collecting data that is derived (i.e., data resulting from an algorithm) or ill-defined and cannot be traced back to the contractor's process. Ill-defined data confuses the analysis and possibly leads to incorrect conclusions. If data must be derived (e.g., productivity data), it is better for the program manager to derive the data than it is for the contractor to deliver the data already derived. In this way, the program manager retains confidence in the data and better understands the issues by developing the analysis from the low-level data items that best represent the processes and products.

The program manager must realize that most issues will require multiple (and seemingly unrelated) data items to characterize the issues. For example, if the issue is the amount of rework and how it is affecting the contractor's progress, the program manager would need to have data on progress, the number of items being reworked, how much effort is being spent on rework, how effective the process is at finding items that need rework early, etc. If an issue is how the amount of rework is affecting the budget or end quality of the product, a different set of data items would be collected. The data needed to characterize this issue is also dependent on the development phase (e.g., requirements, design, or coding). Using the rework example, determining what items to use for measuring rework will be different during the requirements phase than during the design phase.

The way a data item is partitioned can also help to improve the insight into issues. For example, the data could be partitioned by:

- Processor (i.e., target platforms)
- Language
- Organization or subcontractor
- Computer software configuration item (CSCI), computer software component (CSC), or computer software unit (CSU)
- Configuration (e.g., avionics of two different airplanes)
- Severity
- Incremental build
- Facility/Lab
- Work breakdown structure (WBS) element

The level of partitioning will have an impact on how the data is used and how useful the data will be. For instance, if effort is collected at the organizational level rather than at lower levels of a WBS, problems (such as certain modules or units not receiving enough testing) may be disguised. The partitions should highlight the area(s) of concern so that trends from the indicators can help the program manager determine if the issues are being mitigated. For another example, SLOC can be partitioned by newly developed, modified, or reused. If only total SLOC is measured, changes to where the SLOC comes from may go undetected. These changes may reflect significant differences in the effort and schedule required to complete the project. In general, the lower the level of the data, the better the probability of isolating problem areas. However, the lower the level of the data, the higher the cost of the measurement process. Therefore, tradeoffs will need to be made.

The contractor's plans, assumptions, models, and historical data provide the basis for using actual project data. There are four types of data that program managers will use:

- Historical
- Plan
- Actual
- Projections

The contractor will use historical data and assumptions to generate estimates. The estimates, along with other knowledge of the system and environment, are then used as the basis for planning the project. Actual data is collected monthly by the government program manager and used as the basis for comparing the contractor's performance to its plans and for projecting future trends. The government program manager also uses historical data to determine the realism of the contractor's plans. After the plan is accepted, the government program manager uses actual data to ensure that the contractor is managing the project according to the plan.

2.2 Collecting the Measurement Data

Plan data is collected from contract documentation (e.g., the statement of work, the contractor's proposal, or software development plan). The program manager has actual data collected or reported monthly starting the month after the contract begins and continuing until the contract ends. However, not all data is collected monthly throughout the contract. For example, actual computer resource utilization data is not available during early development (although prototypes can be developed to simulate performance if the resource utilization is critical).

Whenever data is collected, its time of collection and source should be noted. The contractor's data should be treated as proprietary to prevent abuses and reduced availability and to alleviate the contractor's concerns about how the data will be used. The contractor will already be concerned about the use of the data and using data as a tool to quantify its performance. If measurement becomes an adversarial issue, the quality and availability of the data will probably be greatly reduced. Therefore, the data should be used as a tool for improving communications with the contractor and as a basis for jointly identifying and resolving issues.

Along with the source and time, assumptions and other knowledge about the data should be noted. Later, when analyzing the data, the program manager will need the data and all the information available about the data to completely understand the information. For example, if actual effort expended is lower than expected, the contractor might have had uncontrollable problems such as staff awaiting security clearances or waiting for equipment to be delivered. Other types of information could signal potential problems (i.e., changes to plans).

The program manager can use tools to help collect, manage, and report the data. Such tools can automate many labor intensive and tedious tasks. Without tools, a measurement process may not be feasible, both from an economical and technical standpoint. The two primary types of tools are collection tools and databases.

Collection tools will facilitate the timeliness, accuracy, and efficiency of the process. Not all data can be collected with an automated tool, but data such as lines of code, effort, staff, and computer resource utilization benefit the measurement process when collected with an automated tool.

Databases expedite reporting and provide the overall efficiency needed to make the measurement process viable. A database allows the data to be stored, and it can include many different attributes that provide the program manager multiple views when addressing issues. At times, it may be best to share a database with the contractor or have the contractor maintain certain databases (e.g., a defect database). Here the preferred method is to have the contractor maintain a defect database with on-line access available to the program manager. In this case, the program manager has data on defects readily available and occasionally audits the information to validate the data and the process of updating the database.

Not all data can be obtained with an automated method; therefore, some other options available to the program manager include:

- Collecting data from other deliverables (e.g., plans, specifications, and status reports).
- Collecting data from activity output sources (e.g., configuration management or software quality assurance files).
- Collecting data that is the output of processes (e.g., inspections, peer reviews, audits, or formal reviews).
- Having data reported via a contract data requirements list (CDRL) item.

However data is obtained, it is best to use one primary method to collect the data and a different method to validate the data and primary collection method periodically. To lessen the cost of data collection and analysis, measures from all sources of CDRL items could be coordinated so that they are collected only once. For example, data and analysis from configuration management activities, performance monitoring, and cost or performance reporting may provide much of the information required for progress assessment. Also, the development of common definitions and parameters could extend beyond the data collection area.

2.3 Understanding the Data

Once data is collected, the program manager will need to analyze it to determine if identified issues are being mitigated and new issues are being identified. Before making determinations from the data, the program manager needs to consider the data and thoroughly understand what it represents. To understand the data:

- Use multiple sources to validate the accuracy of data and to identify differences and causes in seemingly identical data items. For example, when counting software defects by severity, spot check actual problem reports to ensure that the definition of the various levels of severity are being followed and are being properly recorded.
- Investigate the process for preparing the lower level data and understand what the data represents and how it was measured. When analyzing software size, for example, how are the estimates being prepared? What assumptions are being used to generate the estimates? Does the total represent a mix of estimates and actual data?
- Separate data and its related issues from the program issues. There will be issues about the data itself (sometimes actually negating the use of certain data items). Program managers shouldn't get bogged down in data issues, but instead, should focus on the program issues and the data items that they have confidence in to provide the necessary insight. For example, suppose that 50 problem reports are being generated per week and only 4 are being corrected. The program manager should not be bogged down with what 4 were fixed or what kinds of problems are included in the 50. The program manager should be concerned that the backlog of problems is rapidly increasing and that the development process itself may be out of control.
- Do not assume that data from different sources (e.g., SQA or subcontractors) is based on the same definition, even if predefined data definitions have been required. For example, for SLOC data, even though a specific definition was required, different organizations may modify that definition. Sometimes these modifications may be made for an acceptable reason such as an organization having an in-house code counter. In this scenario, the program manager must understand the definition and any differences or variability it will cause when using the data.
- Realize that development processes and products are dynamic and subject to change. Differences of data originating at different points in time may simply reflect the differences in the processes and products. For example, the number of integration tests may be greater than originally planned if the number of CSUs increases or more testing is recommended at a milestone review because of unforeseen complexities. The amount of change must be analyzed over time. Some change is natural and healthy and shows that the contractor is responsive to evolving program needs; however, too much change could indicate that the process is not stable or the original requirements were not adequate.

For example, Figure 2-1 shows monthly estimates for software size. There is some variability in the first nine estimates. Estimate ten, however, drops significantly. This should prompt the program manager to ask questions about the data and understand exactly what is being counted. The notation on the figure shows that a new methodology was used for estimate ten. The program manager needs to understand the new methodology, how it impacts other assumptions and data inputs, and how it affects plans that used the size estimate to make other estimates (e.g., effort and schedule planning). Estimate thirteen is also noticeably different in that it includes a significant amount of reused code. Again, the program manager needs to understand the data and the program assumptions used to generate the data.[2]

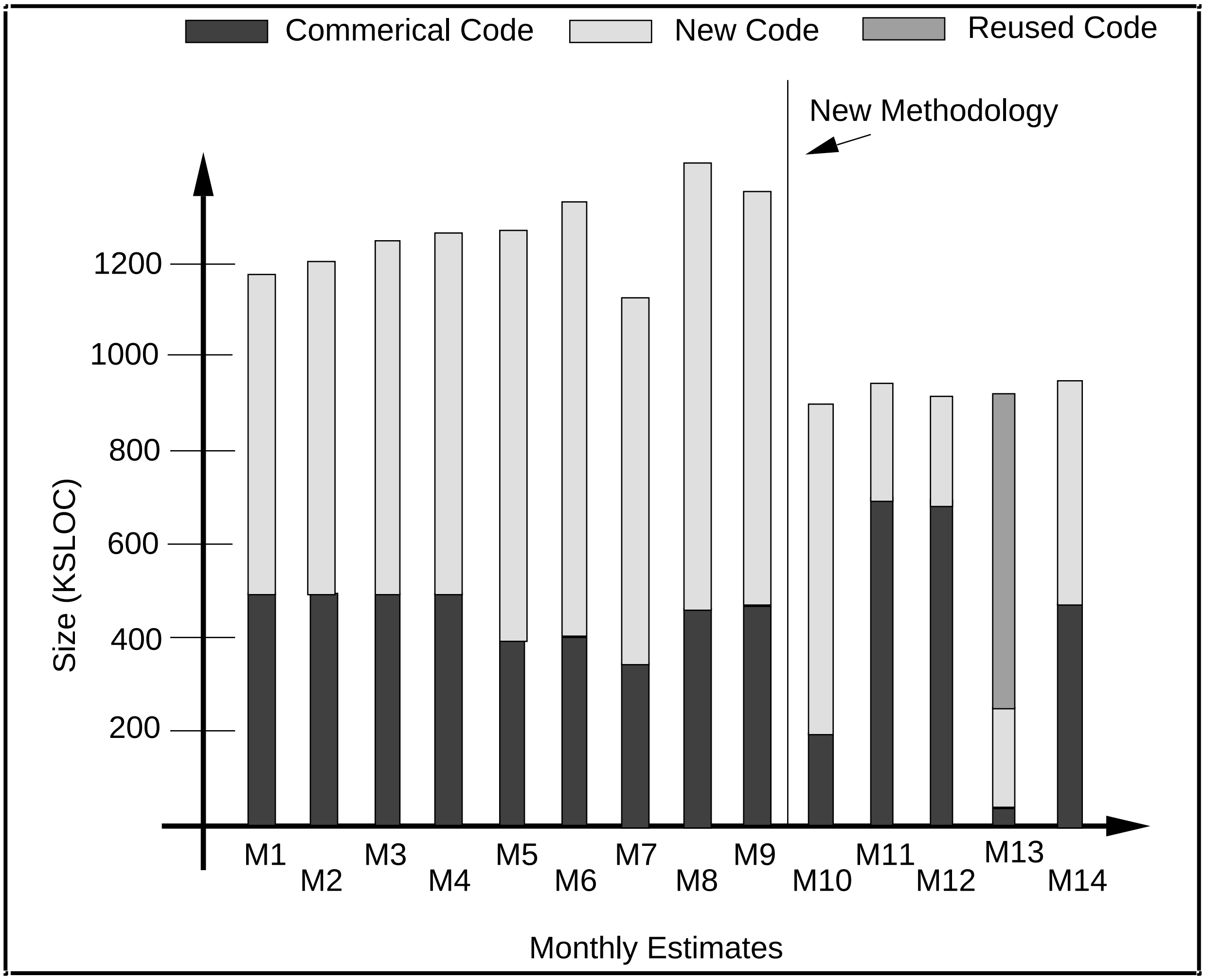


Figure 2-1 Example Illustrating the Need for Understanding Data

[2] Many comments were received that this example is unrealistic. However, this is data from a real program and is used to illustrate a point. It does reinforce the point that you MUST understand data.

2.4 Using the Data

Data is the basis for all analyses. Not only is it used to generate indicators and graphs, but assumptions, other knowledge, and understandings about it are used extensively during analysis to gain insight into issues.

Analysis of the data must involve all levels of staff in the development process in an integrated effort toward improving performance. Successful implementation of software measurement depends on the ongoing interaction between the contractor and the acquisition program office. All levels of staff in the development process must understand the analysis process, and the program manager must understand the analysis results in the context of what is happening on the program to improve his or her decision making abilities.

Measurement data need to be provided by the contractor, then processed and analyzed by both the contractor and the government. The contractor needs the data to manage its own process, to adjust its plans, and to provide meaningful status reports to the government. Experience also has shown that the government must have access to the basic data so it can independently analyze issues and interpret the results.

When processing the data, techniques that can help the program manager understand and analyze it include:

- Partitioning data to depict particular differences, e.g., by organization, processor, or CSCIs (computer software configuration items). For example, when using the effort measures, show the effort expended by subcontractor and CSCI.
- Separating inaccurate, inconsistent, or obsolete data from meaningful data. As the data is better understood, it will be realized that not all data items received are useful either because the items are outdated, low confidence levels in the validity of the data exist, or the data are inconsistent with other data items that are considered to be accurate and valid.
- Putting data sources (document, organization, date, etc.) on the graphs to support interpretations. Knowing the source and the date of the data will help the program manager to interpret it. However, if the contractor is explicitly or implicitly named, the program manager should handle the data as sensitive or proprietary.
- Putting major milestones on the graphs to support an overall view. To better understand how the data being displayed impacts or interplays with the overall process, major milestones (e.g., PDR and CDR) could be annotated on the time axis of graphs. This gives the program manger more insight into how trends in the data may impact the contract.
- Reviewing changes in plans or methods (e.g., build content and CSCI allocation) to determine applicable comparisons (i.e., data inputs to graphs). The program manager should preserve original baselines and assumptions so that adjusted plans accurately present progress history. When generating the graphs, the program manager should include the original baseline plan along with the currently approved plan and actual data. The program manager could also include other plans, both

approved and unapproved, for additional insights into the stability of the contract and plans. To do this, the graphs need to include the various plans as distinguishable curves as in Figure 2-2. This provides the program manager additional insight into the stability of the contract and plans.

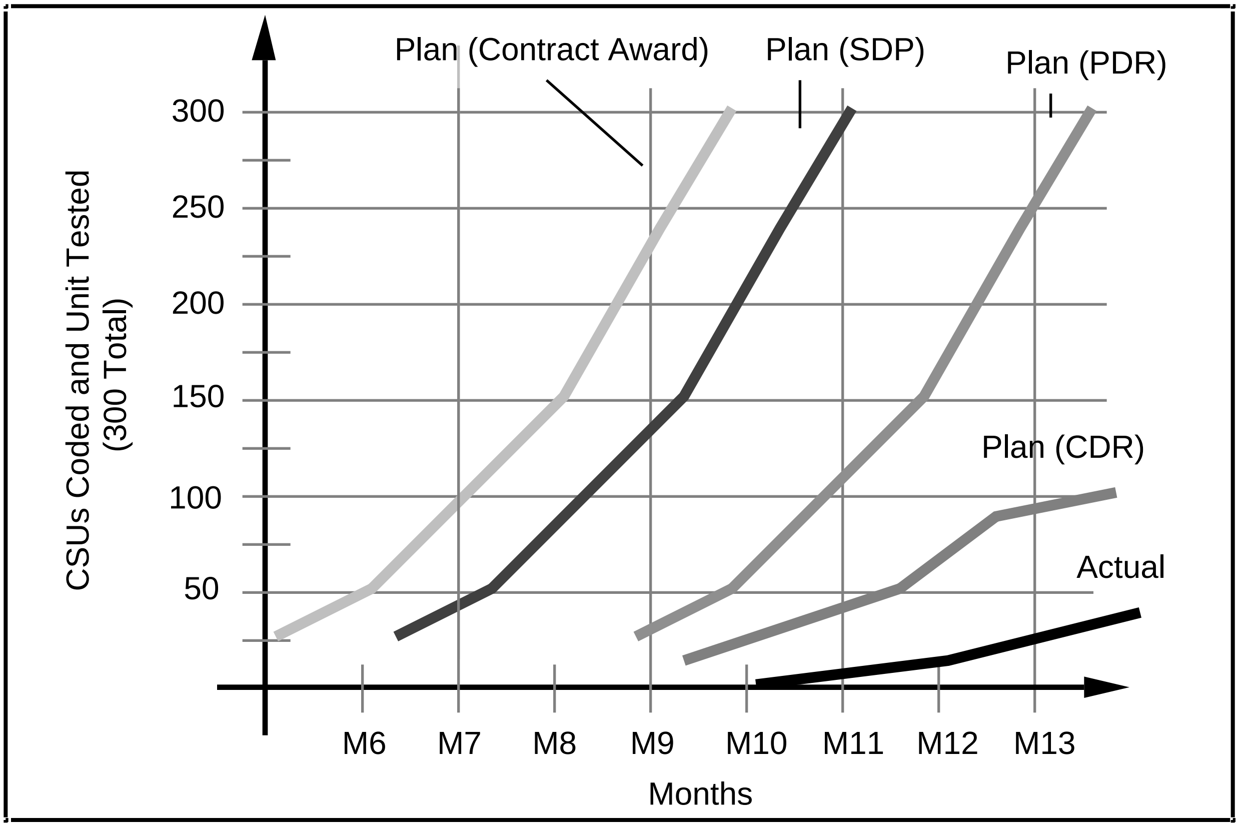


Figure 2-2 Example Showing How to Track Multiple Plans

In summary, the objective is to have an understandable set of measurement data that can be processed into different reports and graphs to address and help answer different issues and questions as they occur. To accomplish this, program managers should collect much of the data themselves and at a low enough level so that the data can be clearly understood and represented in many different ways.

3. Software Measures for Common Software Development Issues

This chapter highlights how a government program office might define, collect, and use data to address issues such as those outlined in Section 1.5.

3.1 Software Size

3.1.1 Purpose (Software Size)

Software size measurements give the program manager an indication of the size of the software being developed. Software size is used as an indication of the amount of work to be done and the amount of resources needed to do the work. The data collection records of size estimates and actual size, together with the assumptions from which the estimates were derived, can provide valuable historical data for improving the processes to estimate cost and schedule, thereby improving overall project management and planning.

3.1.2 Description (Software Size)

The program manager tracks the actual software size, compares the estimate(s) (both overall and for each incremental build), and then analyzes the trends for indications that the size of the software is growing or that functionality is moving from earlier to later increments. Estimates of various size attributes are compared with actual or new estimates monthly throughout the life cycle of the program. The basic size attributes are:

- Number of requirements: gives an early indication of the software size based on actual data.
- Number of CSUs: indicates the amount of work to be completed for each software increment.
- SLOC: gives an indication of the accuracy of the estimates by comparing actual values to estimates.

3.1.3 Data Inputs and Collection (Software Size)

The data inputs the program manager collects or has reported via a CDRL item for the software size measurements are:

- Number of distinct, functional requirements in the Software Requirements Specification (SRS) and Interface Requirements Specification (IRS): The number of requirements is partitioned by functional area for each CSCI. Starting with the system requirements, requirements data are collected from the System/Segment Specification (SSS) and then throughout the development process for each SRS and IRS.
- Number of CSUs as documented in the Software Development Plan (SDP) or the Software Design Document (SDD): The CSUs are tracked by CSCI/CSC for each build. The number of CSUs is estimated early in the process and is then tracked with actual data from the SDD(s).
- SLOC estimates for each CSCI and build and actual data from the source listing for each CSU: For each CSCI's incremental build, SLOC are partitioned by implementation language (e.g., Ada, C, or assembly) and by the amount of new, modified, and reused SLOC. The number of SLOC to be developed is estimated in the contractor's proposal. These estimates are then updated monthly or during reviews and are tracked and compared with the actual data as it becomes available.

3.1.4 Sample Indicators, Analyses, and Actions (Software Size)

Software size has a direct impact on the total development cost and schedule. A program manager can use the size measure to help answer the following questions:

- How much do the size estimates change over a period of time during the development process?
- How much do the actual data deviate from their estimated values?
- How does the trend of the estimates and the actual data affect the development process?
- Is the ratio of reused to new code changing and what are the implications to cost and schedule?

Major variations in the size data could indicate:

- Problems in the use, appropriateness, or validity of the model used to develop the estimates.
- Instability in requirements, design, or coding.
- Problems in understanding the system to be developed.
- An unrealistic original estimate for the system to be developed.
- Unachievable target productivity rates [BEAM].

Significant changes in estimates should trigger a risk assessment, preferably using size-based models to compare effort and schedule to planned values. For example, in Figure 3-1 the size of the software to be developed has nearly tripled from M1 to M6 (where M1 is the first month after contract award). Such a size increase could indicate that the contractor didn't understand the system to be developed, the requirements for the system to be built have changed significantly, or the original estimate was unrealistic. Such trends—and even ones of much lesser magnitude—indicate the estimated cost and schedule may not be met. Even if the estimated cost and schedules are met, the quality of the product could be lower than desired.

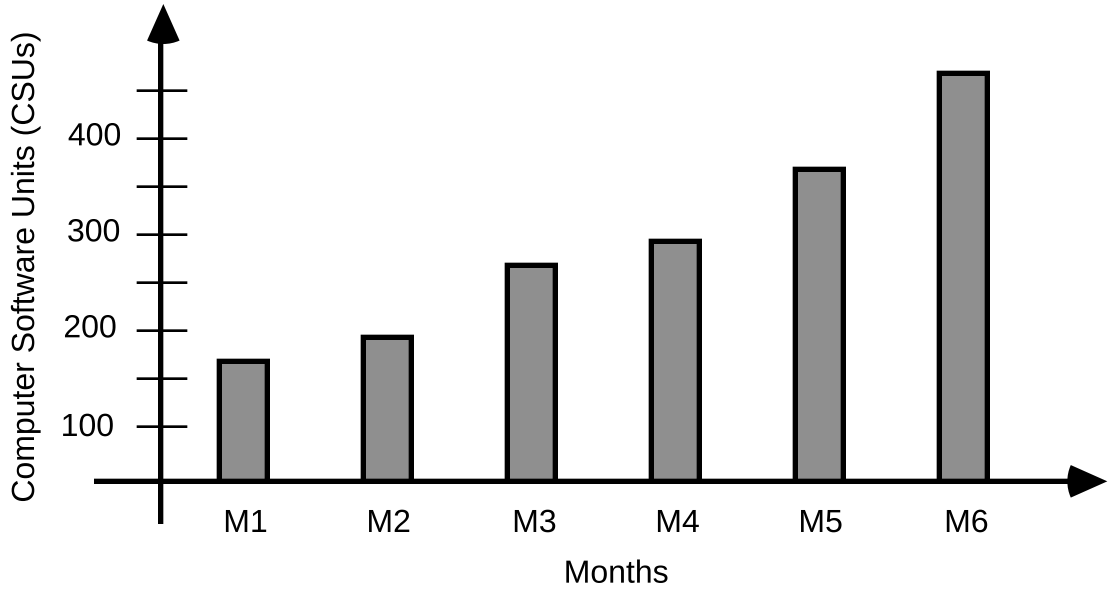


Figure 3-1 Sample Software Size Measure - CSU

The program manager uses the CSUs per build primarily to reveal postponement of functionality to meet schedule commitments. For example, in Figure 3-2 a fourth build shows up at M4 and the functionality of build number 1 is reduced. This and the magnitude of growth should signal a warning to the program manager that the contract's cost and schedule (and maybe the quality of the products) are at risk.

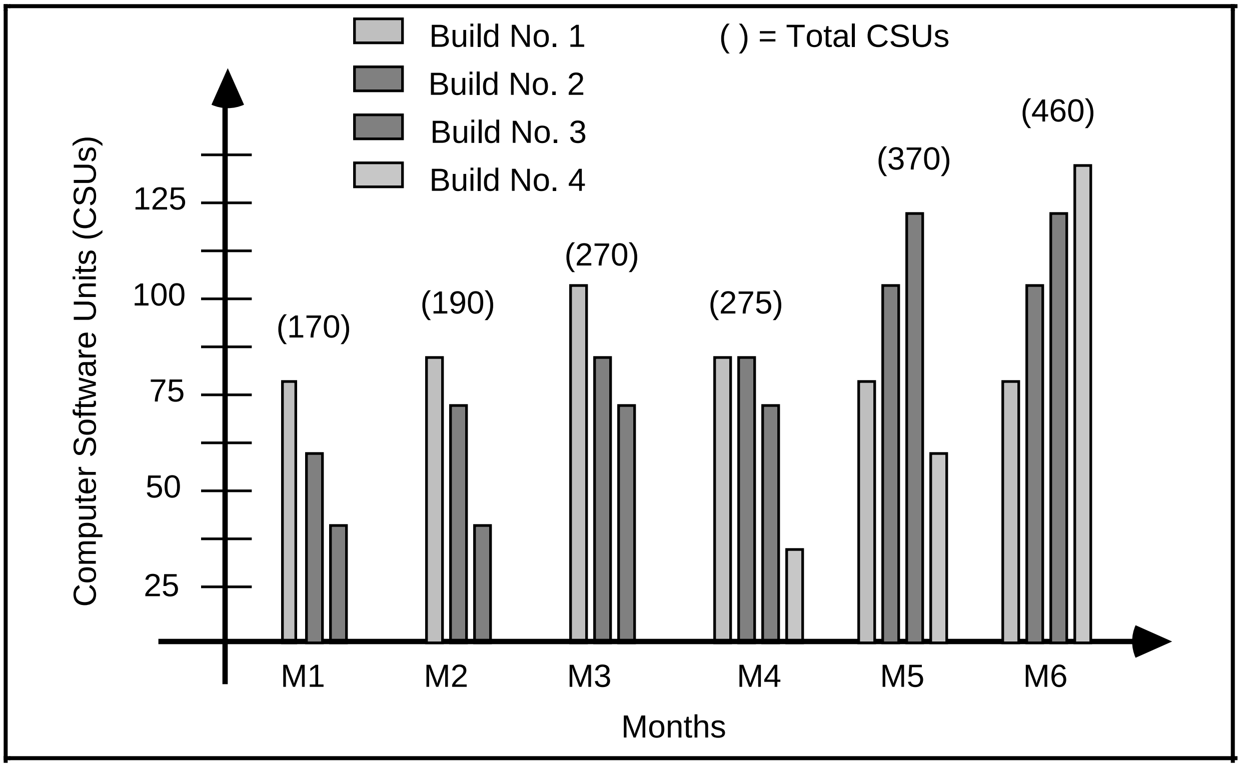


Figure 3-2 Sample Software Size Measure - CSU Per Build

Both of the figures indicate a contract with visible cost and schedule risks. In such cases, the likelihood of a major replan and possible contractual changes (e.g., engineering change proposals) increases. If funds and time are available and mistakes are not repeated, then contractual changes could be useful for replanning the project and mitigating some of the exposed risks.

To illustrate further the importance of tracking changes in plans, refer to Figure 3-3. Figure 3-3 shows the change of size estimates for each build from the second plan submitted to the seventh. Danger signs in this figure are the delaying of the development of code from earlier builds to a later build (i.e., build number 5) and the disappearance of the COTS (commercial off the shelf) code. Here, the program manager needs to investigate why there is such a postponement and what is being postponed. The program manager also must correlate the postponement with the other measures, most notably the development progress measures (Section 3.5). Such a postponement might have been predicted much earlier in the development by correlating the change in plans with other measures.

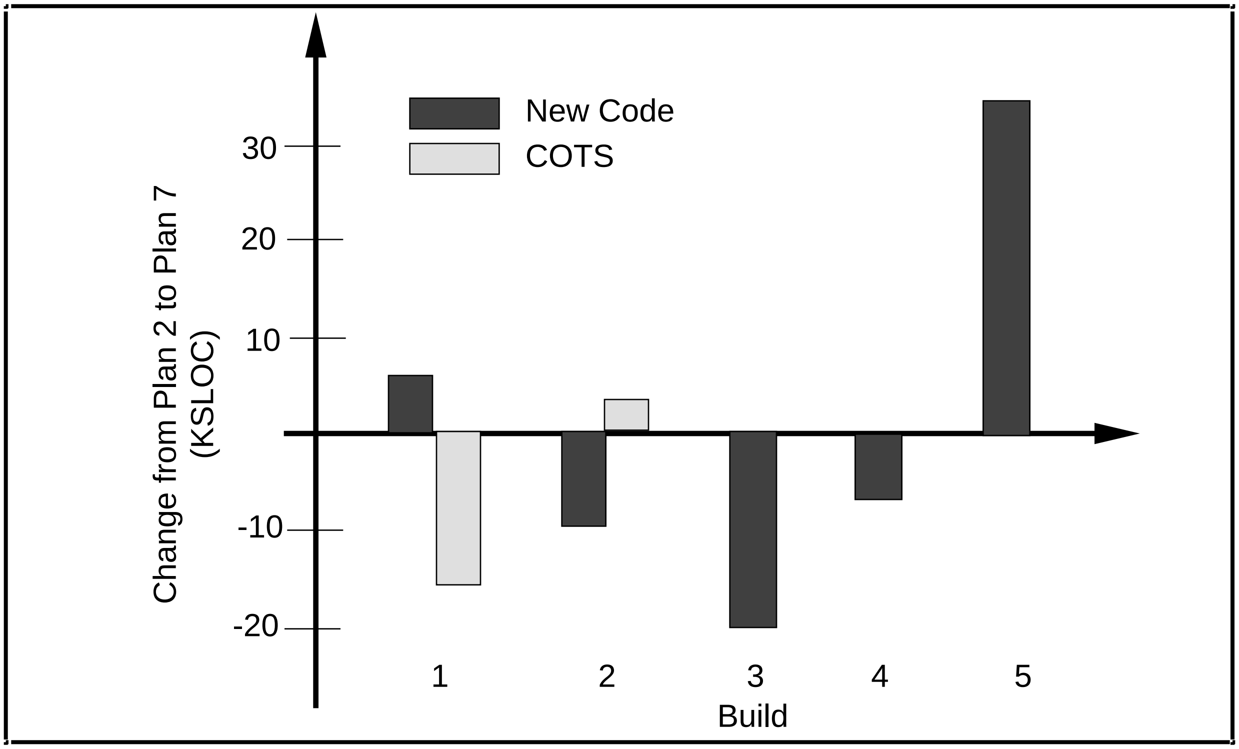


Figure 3-3 Example Showing the Postponement of Code Development From Plan 2 to Plan 7

Other uses of the size data include:

- Correlating it to other measures to determine its validity. For example, size data can be correlated with the effort measures (Section 3.2) or other size data (e.g., number of requirements).
- Correlating estimated SLOC with staffing plans. This allows the program manager to assess the realism of planned staffing levels using the size measures.
- Determining if actual productivity rates are deviating from those used to estimate the cost and schedule. If the rates used to estimate the cost and schedule are not realized, then the budget and schedule are at risk.

3.1.5 Other Measurements and Partitions (Software Size)

Based on program issues, the program manager may also want to consider the following software size measurements and partitions:

- SLOC for each processor
- Object code size
- Database size in terms of bytes, records, or fields
- Function points [IFPUG]
- Design language statements
- Design objects
- Pages of documentation
- Test cases
- Partitions by subcontractor

3.2 Effort

3.2.1 Purpose (Effort)

Effort measures show the relationship between planned and actual staff hours expended. They are used to monitor whether work products are being developed according to planned expenditures of resources. Effort measures allow the program manager to track the contractor's effort and to make inferences about the cost.

3.2.2 Description (Effort)

The program manager tracks the number of staff hours expended monthly starting at contract award and compares planned versus actual level of expenditures. Staff hours may be partitioned by direct labor and support staff categories, experience levels, discipline areas (SQA, testing, programming, etc.), or activity (requirements analysis, design, etc.).

3.2.3 Data Inputs and Collection (Effort)

The program manager has the actual staff hours expended reported each month. For each contractor labor category, the data inputs are reported via a CDRL item. (The labor categories are the same as those used for the staff measurements in Section 3.3.) From the contractor's proposal and development plans, the program manager extracts the planned staff hour expenditure rates for comparison with the actual data. The program manager should have actual staff hour expenditures reported monthly starting at contract award and continuing for the life of the contract. The data is collected for all non-government contributors. That is, it includes data for subcontractors, independent verification and validation (IV&V) contractors, separate test organizations, etc. To provide better insight into and control over the project, staff hours may be partitioned by:

- Development discipline area (SQA, configuration management, analysis, programming, etc.)
- Development activity (design, code and unit testing, etc.)
- WBS elements at the level that software activities are defined

3.2.4 Sample Indicators, Analyses, and Actions (Effort)

Each month the program manager collects the number of staff hours expended and aggregates the data to obtain the total staff hours expended for the contract.

The measures for monthly staff effort expended tracks staff hours expended against staff hours planned. Tracking effort monthly lends insights into the stability of the software development and the risks of not completing the project within cost and schedule. If the effort expended exceeds plans and work progress fails to meet schedule plans, the quality of the software may also be in jeopardy.

The measurements for the total staff effort expended tracks the total number of staff hours planned to be expended against the actual number expended up to a point in time. Information from the aggregated data is used to determine if the contractor is meeting the planned amount of effort. When used in conjunction with the development progress measures, the data can provide the planned and actual number of staff hours expended to complete each of the development activities. The primary reason for tracking the total effort is to be forewarned of cost overruns.

A variance between the planned number of staff hours and the actual number of staff hours expended will inevitably occur. How large must this variance be for the program manager to take action? The answer to this question depends on the issues that are important to the program. For example, consider a program that consists of building customized hardware where the software is needed in the early phases of the project to test the interaction between the hardware and software. In this scenario, changes to the schedule may not be tolerable. The program manager needs to determine how much of and how long an underexpenditure of staff hours can be tolerated to keep the project on schedule. To do this, the program manager must also use the milestone performance, software size, quality progress, and development progress measures to make inferences and management decisions.

Total effort expended data, when graphed, is usually in the form of a flattened S-curve, as depicted in Figure 3-4. The flattened S-curve reflects a smaller staff at the beginning of the project (a smaller sloped portion of the curve), a larger staff during the main part of the project (a more steeply sloped portion of the curve), and a reduction in staff at the end of the project (another smaller sloped portion of the curve).

The total effort expended data will most likely show a variance between staff hours planned and staff hours actually expended. However, because this is cumulative data, the difference between the plan and actual curves will not be large early in the project. For this reason, the program manager should take notice of small but steadily increasing differences between the curves.

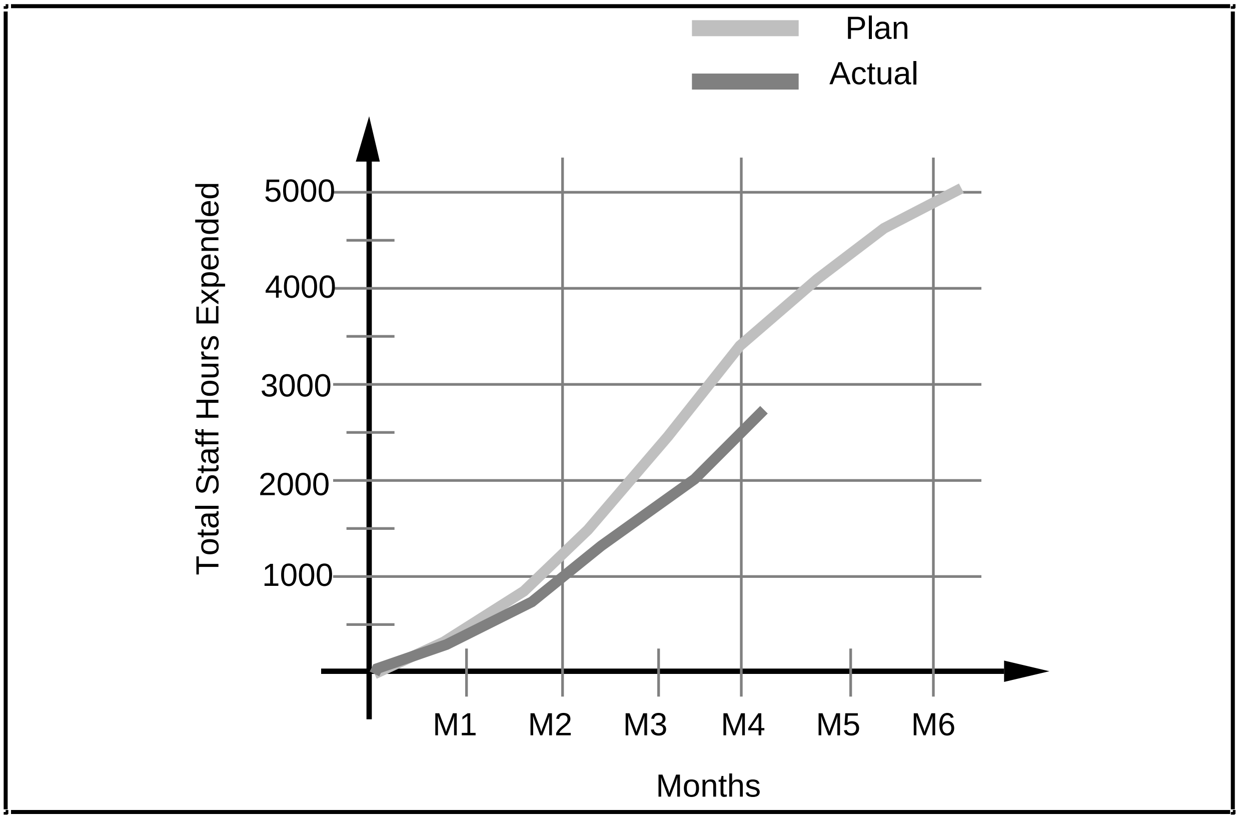


Figure 3-4 Sample Total Staff Effort Expended Measure

Figure 3-5 shows an example using the monthly effort data. The curve indicating the planned number of staff hours shows an orderly and achievable increase through requirements analysis and detailed design, a semi-constant level around a maximum during coding and unit testing, and an orderly decrease through integration and delivery. (Such a curve would exist for each iteration of activities.)

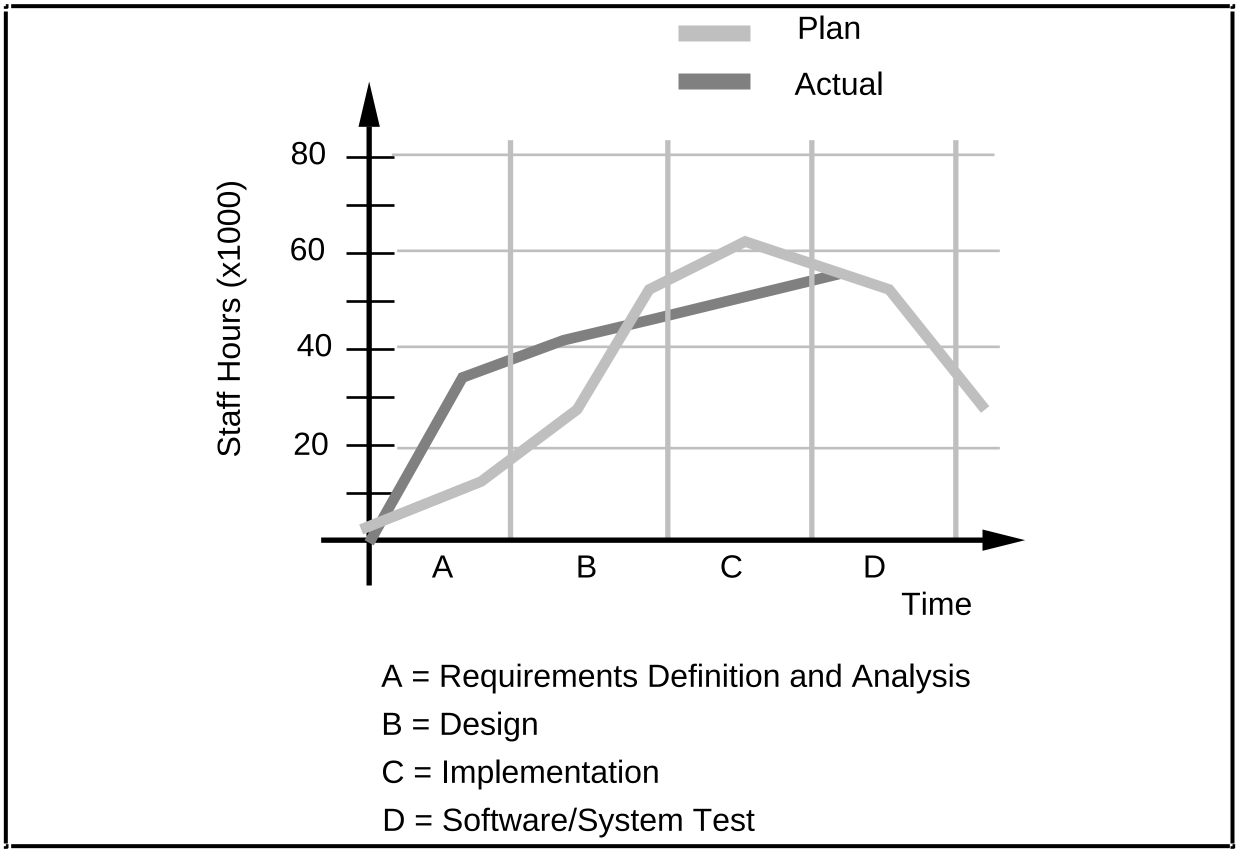


Figure 3-5 Sample Monthly Staff Effort Measure

Effort measures should be used with the measures for milestone performance, software size, software defects, staff levels, and development progress. If there is a significant underexpenditure of staff hours, the contractor may be having problems staffing the contract. Other possible reasons for underexpenditure of staff hours include:

- Overestimating the software size: The program manager should correlate effort measures with trends in the software size and staff measures. If the actual size data is tracking below the plan curve, it may indicate that the size was overestimated or that the contractor does not have an adequate number or the right experience mix of personnel on staff. When both trends occur simultaneously, the project is usually behind schedule.
- Insufficient development progress: By correlating the effort measures with the development progress measures, the program manager can determine if underexpenditure of staff hours is causing the development progress measures to track below the plan curve. The staff measures should also be investigated to determine whether the contractor has adequate staff to perform the task.
- Increasing levels of open problems: The program manager should correlate the effort measures with the software defects measures. If the difference between the number

of open and closed defects is increasing, additional staff may be needed or redirected to correct outstanding defects.

If there is significant overexpenditure of staff hours, the contractor may have been forced to absorb staff members from other projects that were ending. Other possible reasons for overexpenditure of staff hours include:

- Underestimating the size of the software: The program manager should correlate effort measures with the software size measures. If the size measures are tracking above the plan curve, it may indicate that the size of the software has grown well beyond the estimate, requiring the expenditure of more staff hours than originally planned [BOEHM87].
- Insufficient development progress: By correlating effort measures with the development and milestone performance measures, the program manager can determine if the contractor is trying to make up delays by adding staff to the contract. This may not be successful; according to one of Brooks' rules, "adding people to a late project just makes it later." [BROOKS82]
- Increasing number of defects: By correlating effort measures with the software defects measures, the program manager can determine if the contractor is adding staff to correct a growing number of problems. This may indicate a software quality problem.

3.2.5 Other Measurements and Partitions (Effort)

Depending on program issues, the program manager may also want to consider substituting staff weeks, staff months, or even staff years for staff hours on the y-axis. The conversion factor must be defined and documented (e.g., 156 staff hours equal one staff month) if a time period other than staff hours is used as the y-axis variable. If the unit of measure on the x and y-axes are of equivalent value (e.g., months and staff months) then the y-axis will correspond to the number of equivalent planned and actual full-time personnel on the project. (See Section 3.3.)

3.3 Staff

3.3.1 Purpose (Staff)

Staff measures give the program manager insight into the staffing labor categories used on the contract. The program manager uses this insight to track the contractor's ability to maintain a sufficient level of staffing to complete the contract [BOEHM87] and to track the amount of the contractor's staff turnover. When used in conjunction with the effort measures, the program manager also gains insight into the number of part-time people working on the contract and the amount of overtime being worked (regardless of whether or not it is compensated).

3.3.2 Description (Staff)

Typically, early development activities have a more experienced staff, i.e., a large percentage of staff at the higher labor categories. As development progresses, however, experienced staff get replaced by less experienced staff. Program managers track the actual staff levels of the contractor's labor categories and compare them to the levels that were planned to be used. This measurement is analyzed by looking at and determining the impact of the variances between the planned and actual staff levels. The program manager also tracks the number of people added and subtracted from the labor categories (i.e., staff turnover).

3.3.3 Data Inputs and Collection (Staff)

The program manager has the number of contractor staff by labor categories reported via a CDRL item. To provide better insight into and control over the project, the staff labor categories may be partitioned by:

- Development discipline area (SQA, configuration management, analysis, programming, etc.)
- Development activity (design, code and unit testing, etc.)
- WBS elements at the level that software activities are defined

The program manager can specify in the request for proposal (RFP) an experience profile for the labor categories for potential contractors to address. The labor category profiles and staffing levels proposed by the contractor are used by the program manager as the plan for comparison of actual data. The staffing levels should be planned for the length of the development process.

The program manager should also define how to count the staff turnover and require specific data on the turnover by labor categories and possibly the partitions above. The data is then collected for all non-government contributors. That is, it includes data for subcontractors, independent verification and validation contractors, separate test organizations, etc.

3.3.4 Sample Indicators, Analyses, and Actions (Staff)

The program manager generates graphs of the staff data similar to the one shown in Figure 3-6 to show average staffing levels and turnover on the contract. A chart like Figure 3-6 would be generated for each labor category and for the contract overall.

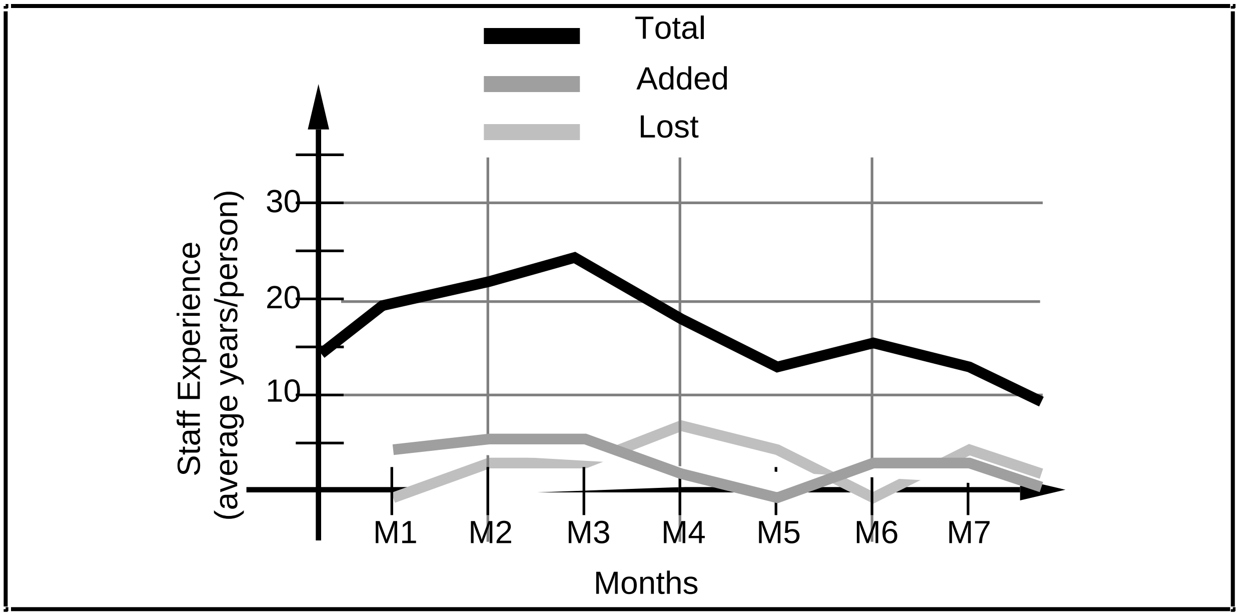


Figure 3-6 Sample Staff Measure

The total staffing level should show an overall decreasing level of experience (i.e., a decreasing percentage of staff at the higher labor categories) because typically more experienced personnel are assigned to the contract during the early development activities (e.g., requirements analysis) than during later activities. (During the testing activity, staff levels may rise but typically not as high as those experienced during requirements analysis.)

Insights into staff capabilities supported by these staff measures help the program manager to identify situations where a lack of staff may cause problems in the program's technical performance. Early insight into specific staff capabilities can also highlight staffing risk areas. Examples of analyses that can be performed with the staff experience measure are:

- Graphing the number of people charging by labor category (i.e., the number of staff per labor category) coupled with the number of hours expended by labor category (Section 3.2), shows the average hours expended per staff member. This information allows the program manager to assess how people are being used on the project. The use of too many part-time people may degrade the accountability and efficiency of a project. Likewise, the overuse of a limited number of people may degrade the quality of the products.

- Determining if actual staffing is consistent with the planned levels and if there is adequate commitment for using senior staff (i.e., staff from the higher labor categories).
- Correlating the staff measures with the measurements for development progress, milestone performance, and effort to determine if trends in these measures are related.

3.3.5 Other Measurements and Partitions (Staff)

Based on program issues, the program manager may also want to consider the following staff measurements and partitions of the staffing levels:

- Development activity (design, code and unit testing, etc.)
- CSCI
- Subcontractors

3.4 Milestone Performance

3.4.1 Purpose (Milestone Performance)

The milestone performance measures gives the program manager a comparison of actual milestone completions against established milestone commitments. These measurements quantify the contractor's performance toward meeting commitments for delivering products and completing milestones.

3.4.2 Description (Milestone Performance)

Milestone performance measures help the program manager graphically portray planned delivery dates, replanned delivery dates, and intermediate activities needed to meet the end delivery dates. The milestone performance measure tracks progress and delays for activities with respect to planned DOD-STD-2167A milestone events. These include system design review (SDR), software specification review (SSR), preliminary design review (PDR), critical design review (CDR), test readiness review (TRR), functional configuration audit (FCA), and physical configuration audit (PCA). It also tracks interim milestone events defined in the software development WBS leading up to the milestones. For each revision to the plans, associated schedule delays are indicated and tracked to determine the impact of the revisions and the delays associated with the revisions.

Other items that may affect milestone performance include tool development schedules and deliveries of government furnished equipment (GFE) and government furnished information

(GFI). If delivery of GFE or GFI is delayed, schedule and development progress may be affected. Therefore, the program manager may want the requested schedule data partitioned further to reveal the progress against these critical item dependencies.

3.4.3 Data Inputs and Collection (Milestone Performance)

The planned and actual data inputs the program manager collects or has reported via a CDRL item are:

- WBS activity start and completion dates
- DOD-STD-2167A milestone start and completion dates
- Key interim product milestone start and completion dates
- Progress to date (overall and for each milestone's interim activity)
- Estimated slip (overall and for each milestone's interim activity)

Objective entry and exit criteria for each event and activity must be defined and agreed on at contract award. It is also important to identify clearly the method of calculating progress to date, for estimated slip, and for the revision status of planned events.

WBS activity schedules and data on actual activity progress are collected monthly throughout the contract. The program manager notes the overall delay resulting from each plan revision when the revision is submitted and relates the delay to the original plan and all approved plans superseding the original.

3.4.4 Sample Indicators, Analyses, and Actions (Milestone Performance)

Experience has shown that late or unacceptable software products are good indicators of schedule risk. These late and unacceptable products are shown on milestone performance measurement graphs as slippage and can reveal contractor problems in maintaining the planned schedule. However, the measures only show the program manager that a deviation from the planned schedule exists; they do not explain why it exists. Milestone performance measures also depict overlapping project activities and the impact of rework and contingent risk abatement on schedule slippage through milestone completion estimates projected from actual data.

The inputs for the milestone performance measures may be displayed as a Gantt chart as in Figure 3-7. The chart shows planned events, actual events, and progress to date against planned activities. The chart shows a delay between planned completion and the actual activity or new estimate for completion. Large programs may have a set of tiered schedules for each system component. From such a figure, the program manager also looks for indications such as:

- Excessive overlap of activities where dependencies are expected (e.g., excessive amounts of coding activity completed before the design activity is completed). For example, in Figure 3-7, Activity 3 and Activity 4 are scheduled to start at approximately the same time. If Activity 4 depends on outputs from Activity 3, the program manager should ask what work will be done on Activity 4 before it receives the outputs from Activity 3.

- Successor activities starting before predecessor activities (e.g., coding starting before detailed design is started).
- Larger slips shown in predecessor activities than in successor activities. For example, in Figure 3-7, Activity 2 has slipped 40 days, yet Activity 3 is ahead of schedule 10 days. If Activity 3 depends on Activity 2, then the program manager needs to question how the 40 day slip will be made up.

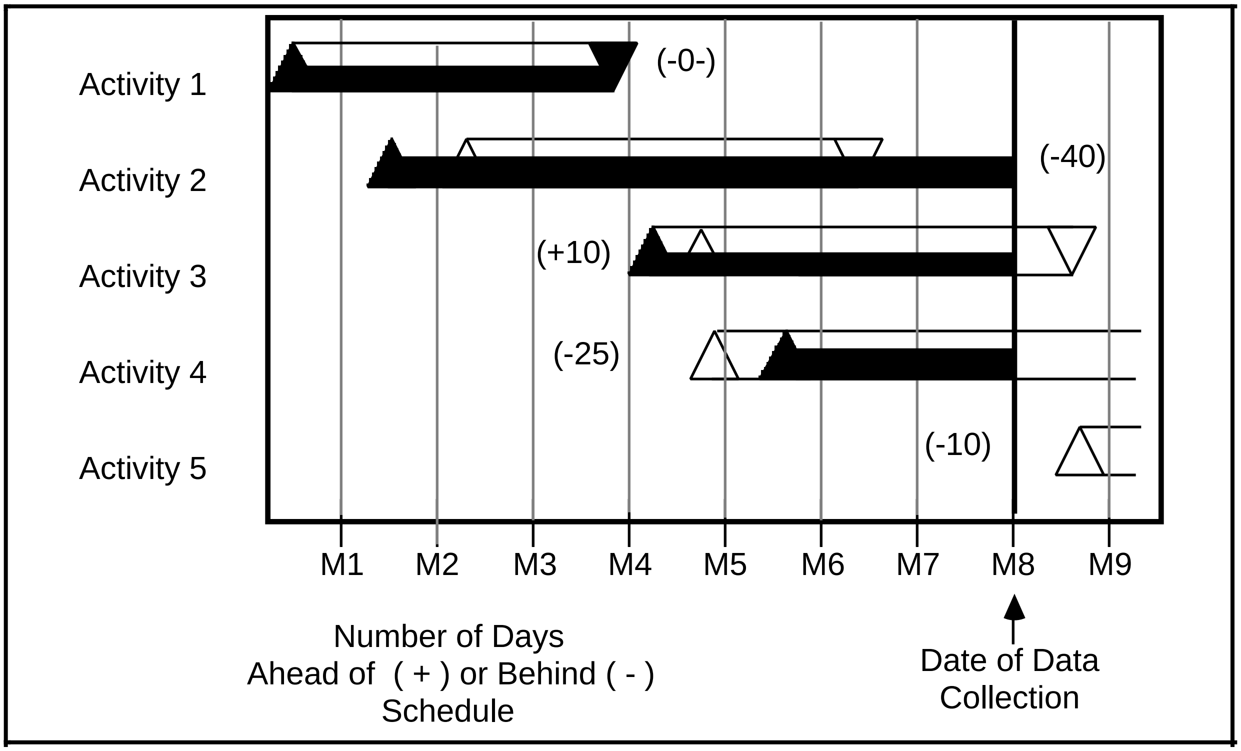


Figure 3-7 Sample Milestone Performance Measure

To determine the stability and progress of the schedule, the program manager could use a table similar to Figure 3-8 which includes for each revision, the date, time since last revision, and schedule change since the last revision. Records of milestone performance measures can be a valuable historical tool for improving schedule estimation, particularly for development efforts with characteristics similar to those of previous systems.

Schedule Revision #	Date	Time Between Revisions	Schedule Change From Last Revision
1	6/15/88	12 months	- 4 months
2	3/15/89	9 months	- 5 months
3	3/15/90	12 months	- 2 months

Contract award date: 6/15/87

Data collection date: 4/1/90

Figure 3-8 Example Table Illustrating Milestone Performance Stability

The program manager can also prepare a figure similar to Figure 3-9 to show schedule change data graphically. From Figure 3-9, the program manager can see overall slips in the schedule. When the program manager receives data such as that shown in the first quarter on Figure 3-9, he or she should be concerned and ask how the first milestone can be late while succeeding milestones are early. Such a scenario usually leads to data similar to that shown in the third quarter where all milestones are late.

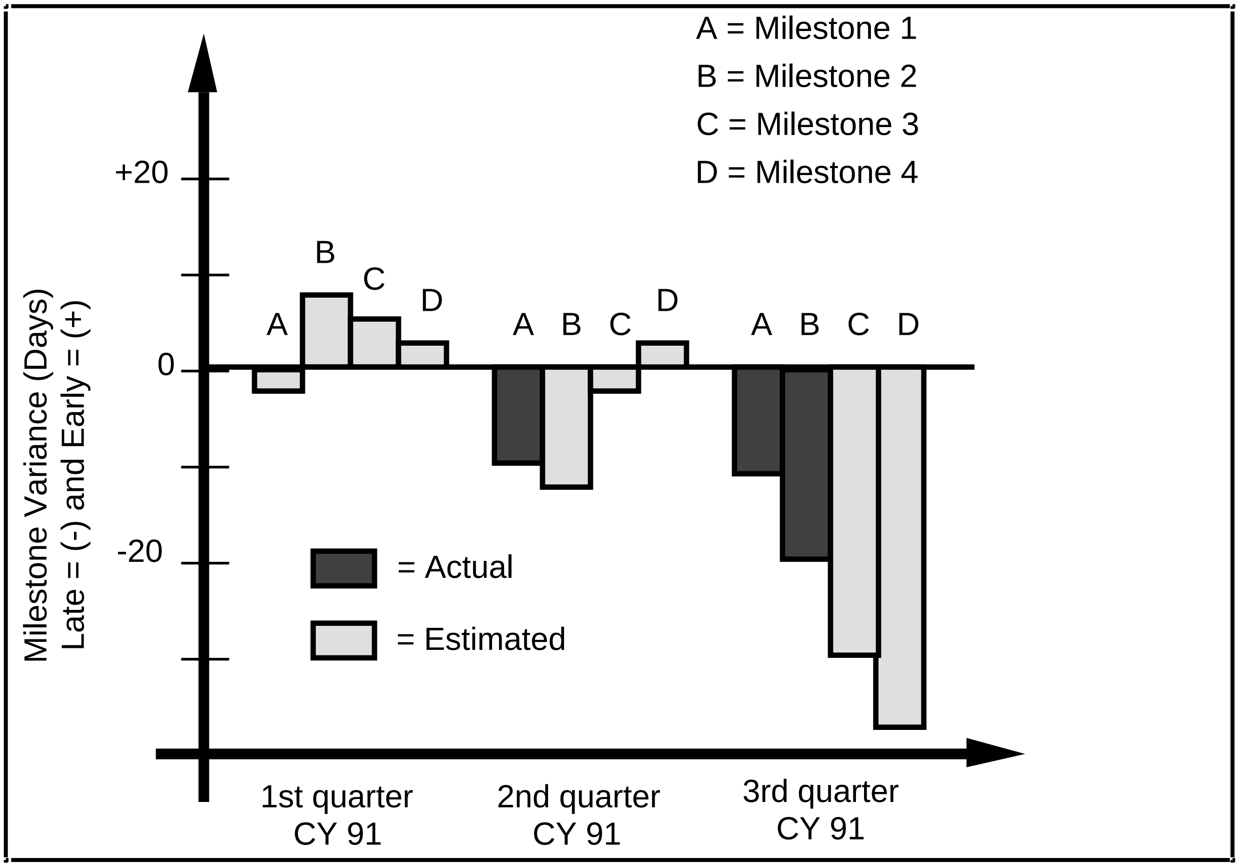


Figure 3-9 Example Measure Illustrating Milestone Performance Stability

Correlations with the defect measures should be made. Excessive emphasis on milestone performance could be counter productive causing the contractor to neglect quality in lieu of progress.

3.4.5 Other Measurements and Partitions (Milestone Performance)

The program manager could also use the following milestone performance measurements and partitions to reveal critical risk areas that might be masked by aggregation:

- Project status at different levels (e.g., CSC, CSCI, build, and total software)
- Project status for software engineering environment and tools
- Project status of GFE/GFI item deliveries
- Activities at lower levels of the WBS

Also, items on which activities depend and which may have a large impact on activity completion could be reported and tracked [CORI].

3.5 Development Progress

3.5.1 Purpose (Development Progress)

The development progress measures gives the program manager a quantitative indication of work progress. The measurement uses data on the planned and actual progress of software development activities to assess whether an activity is complete and the contractor is ready to proceed to successive activities.

3.5.2 Description (Development Progress)

The program manager applies the development progress measures across the major software development process activities, i.e., requirements analysis, preliminary and detailed design, code and unit test, integration and test, and formal test. The development progress measure is used to quantify the work to be done on key interim products of these activities, e.g., CSUs designed or coded. Each of these development activities is further broken down and quantified into work items that are small enough to allow progress to be seen monthly. The work items should be natural artifacts of the contractor's development process.

3.5.3 Data Inputs and Collection (Development Progress)

The program manager collects the following planned and actual data or has it reported monthly via a CDRL item:

- Requirements Analysis
    - Number of software requirements in the System/Segment Design Document (SSDD) that are documented in the SRS
    - Number of software requirements in the SSDD that are documented in the IRS
- Preliminary Design
    - Number of SRS and IRS requirements documented in the SDD
    - Number of SRS and IRS requirements documented in the IDD
- Detailed Design
    - Number of CSUs designed
- Code and Unit Test
    - Number of CSUs coded
    - Number of CSUs unit tested

- Integration and Test
  - Number of CSUs integrated
  - Number of CSC integration tests completed
- Formal Test
  - Number of requirements tested
  - Number of tests completed

The number of work items (e.g., requirements documented or CSUs designed) that are planned and reported must be sufficient to enable the program manager to determine intermediate progress; i.e., the granularity of the work items must be small enough to provide new data for each monthly report.

Tracking starts when an activity passes through its entry criteria. Tracking for each activity can either end when the exit criteria for the activity are completed or continue past the exit criteria as a measure of the rework for that activity [BOEHM87]. The plans should be identified by revision number. Data from past plans should be shown on the development progress measures.

3.5.4 Sample Indicators, Analyses, and Actions (Development Progress)

Early insight into deviations from planned progress is essential for early corrective action to be taken. The program manager's biggest advantage in using the development progress measures is that the true progress of work completed can be followed early in the development process. Moreover, these measures require the contractor to use a detailed level of planning to successfully use this measure.

A key to successfully applying this measure is having the entry and exit criteria properly defined for each of the data items reported. For example, the exit criteria for a CSU design to be counted as complete could be a successful design peer review and no open action items or defects against the CSU; for the code of a CSU to be counted as complete, an error-free compile and code inspection should be achieved; for the integration of a CSU to be counted complete, the CSU integration test should be successfully completed and no open defects should exist against the CSU. If an item's exit criteria have been met and it has been counted as complete, any additional work toward that item can be counted as rework.

Measures for projects progressing well will show a steady, consistent growth of completions as shown in Figure 3-10. Program managers can use a development progress measure similar to Figure 3-10 to determine 1) if any deviations of actual progress from the planned progress warrant corrective action; and 2) if the rate of progress is sufficient to meet the planned major milestone dates (i.e., activity completions) and, eventually, the planned date for product delivery.

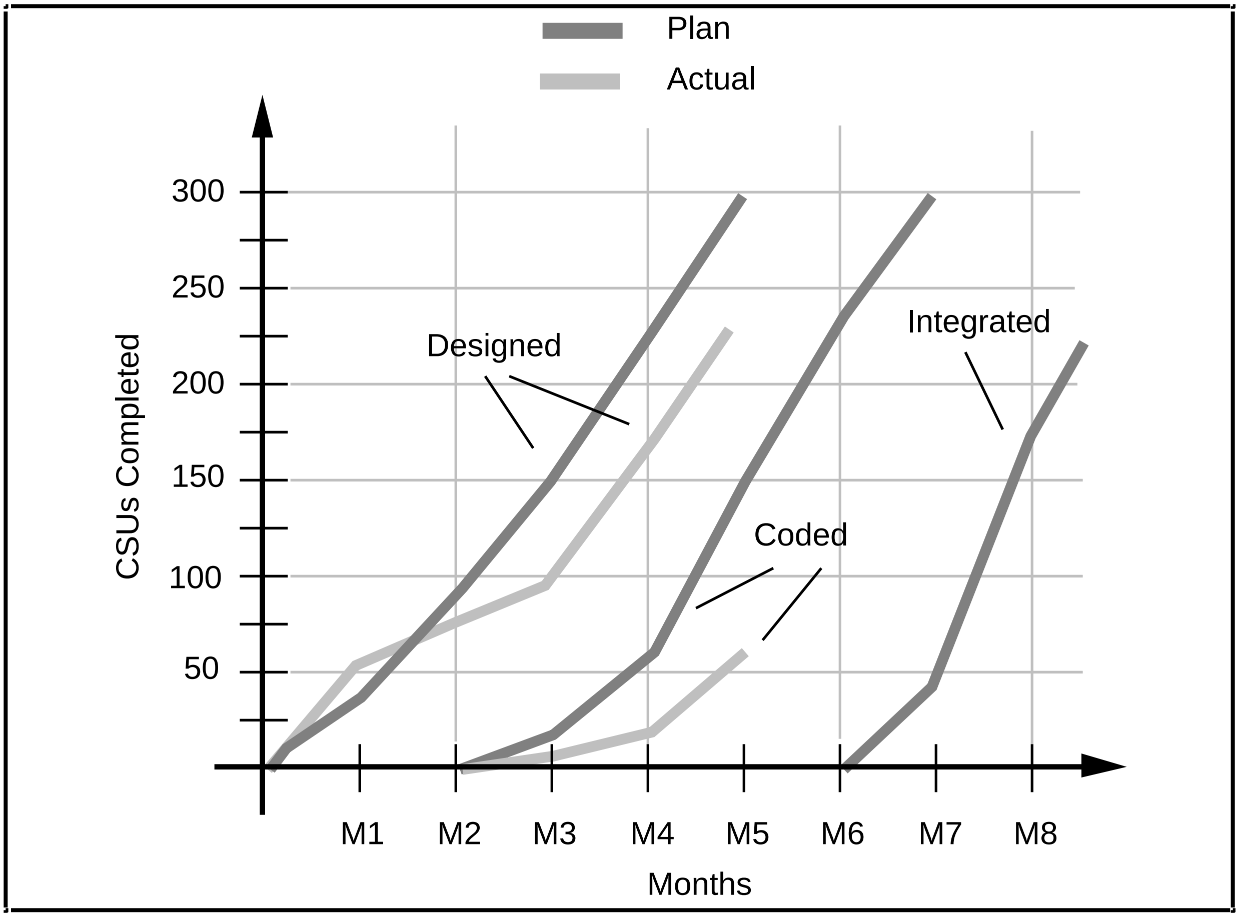


Figure 3-10 Sample Development Progress Measure

Development progress data can also be effectively displayed as shown in Figure 3-11. In Figure 3-11, the number of CSUs that have completed key process steps are tracked. For this measure to be effective, clearly defined exit criteria must be used to determine when a CSU has passed through such a checkpoint.

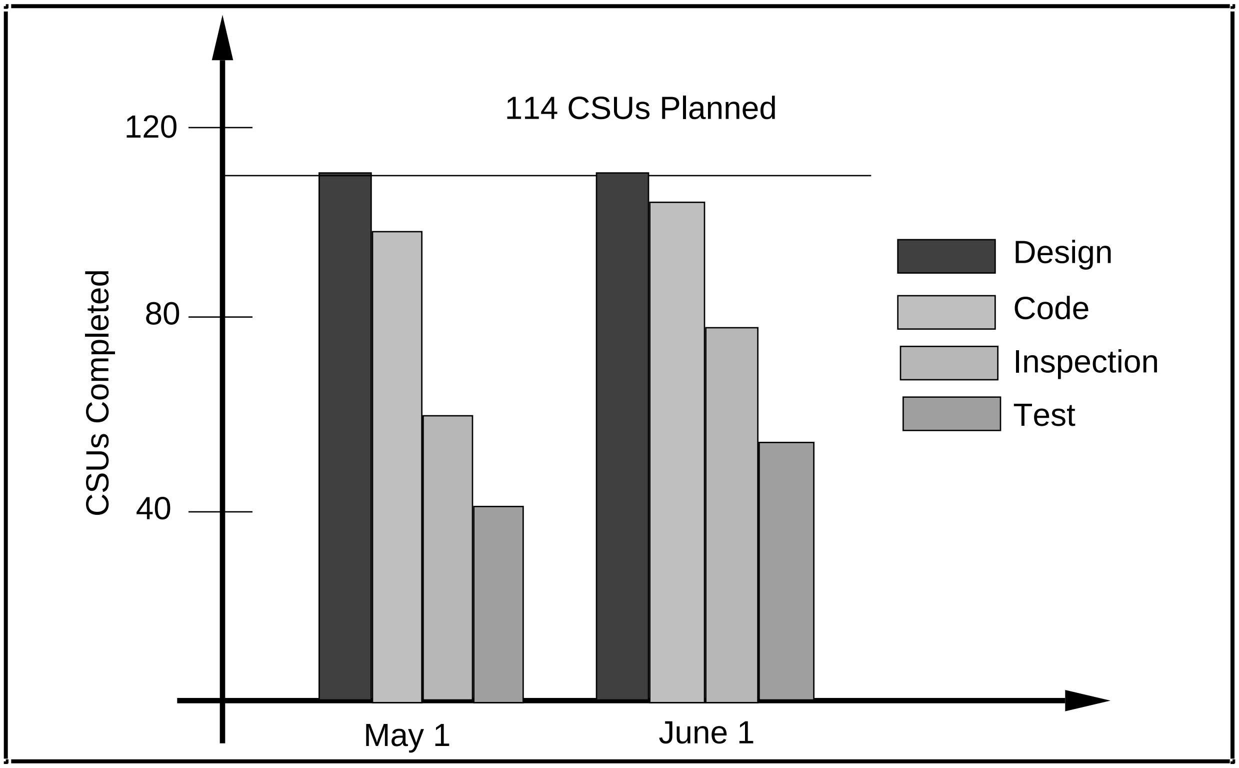


Figure 3-11 Example Showing CSUs That Have Completed Key Process Steps

Uses of the development progress measures include:

- Comparing it to the staff effort measures. Here the program manager compares the total number of work items and the distribution of the work items to the effort distribution to ensure that the distributions are proportional and represent the desired work item granularity.
- Comparing it to the milestone performance measures. Here, the program manager compares the progress shown for an activity in the development progress measures to the progress shown in the milestone performance measures.
- Assessing whether the planned schedule completion date is achievable given the deviation of the progress to date from the plan, i.e., the actual number of work items completed compared to the number that was planned to be completed to achieve the planned completion date. Two items that need to be analyzed to make such an assessment are 1) current deviation between the actual progress and the plan and 2) the rate of progress or slope of the curve for the actual completed work packages from the beginning of work to the last report.

To assess whether the planned schedule completion date is achievable, the program manager might want to:

- Ensure that the completion and exit criteria for milestones and work items are well-defined and verifiable (e.g., for milestones, the review is complete and action items are closed or the source code is submitted to configuration management for the code and unit test activity). If they are not, then true progress may be disguised by having work items reported as completed when work still remains.
- Take into account the quality of the completed work items, especially during the requirements and design stages where problems will be less costly to correct [BOEHM81]. If quality is questionable during the requirements and design activities, then the program manager should consider missing some early milestones to ensure a higher quality software product later. A rough judgment of the quality at these early stages can be obtained by correlating development progress with the quality progress measures.
- Make a gross schedule forecast from the development progress measure by taking the ratio of the planned completed and the actual completed work items multiplied by the total schedule duration. For example, suppose that 100 work items were planned to be completed by a certain date, but only 90 were completed and that the schedule duration for those 100 work items was 6 months. Those 90 items should have taken 5.4 months, thus the progress is at least 6/10ths of a month behind that which was expected. That is:

  if 100 work items = 6 months and 90 work items = x months then

  100/6 = 90/x or x = (6*90)/100 = 5.4

  Note, at least three reporting periods are probably needed before such a forecast is useful. Also, such a forecast assumes that the work items being counted take approximately the same amount of staff hours. If this is not the case, do not do this type of forecast.
- Extrapolate the rate of progress (the slope of the actual completed work item curve between the current report and the last report) to determine if the trend is converging toward or diverging from the planned completed work item curve. If the actual curve is significantly lower than the planned curve and the slope indicates further divergence, this is usually cause for concern.
- Assess the deviation between the planned and actual progress profiles. If the contractor is unable to report planned progress profiles soon after SDR, there may be insufficient requirements analysis and decomposition. If the deviation between planned and actual data is increasing, SSDD requirements may not be allocated to CSCIs and these requirements may not be translated or refined to complete, testable, and traceable SRS requirements.
- Track the number of tests successfully completed against scheduled tests as time progresses. Deviations from planned test progress should decrease to zero toward the end of the contract. If the deviation persists, or if there are repeat test failures, more serious requirements, design, and implementation problems could be the

cause. The quality progress measures should be evaluated to obtain insight into these problems.

3.5.5 Other Measurements and Partitions (Development Progress)

The program manager may also want to consider the following development progress measurements and partitions:

- Track by separate CSCIs or system/segment.
- Show multiple activities on the same graph (e.g., coding and integration testing).
- Show system test progress for critical software.
- Track by incremental release content (i.e., by build).
- Show verification status of requirements and design.

3.6 Software Defects

3.6.1 Purpose (Software Defects)

The software defect measures track the acceptability and problem content of the products. It gives the program manager information on the readiness of the software to proceed to the next phase. It also helps the program manager determine if the product is ready for review or release and if it will be necessary to rework a product that has exhibited problems before proceeding to the next activity. The program manager should make the collection of software defects a constructive effort and encourage the early identification and correction of defects.

3.6.2 Description (Software Defects)

The program manager uses the software defect measures to track the defects resulting from program activity. The program manager uses the information on defects to determine whether to move intermediate products forward to the next development activity or to continue the current activity and further mature the products. For example, if at PDR an inordinate number of action items results from the requirements documentation, the program manager may decide that the contractor needs to spend more time analyzing requirements, updating the documentation, and closing the action items from the review. The program manager may want to advise or redirect the contractor to re-do the requirements specification before continuing on to the design activity.

Defects are anomalies noted during the development process. They are tracked via software problem reports (SPRs). SPRs should be tracked for any product or work item that has passed through its exit criteria and was counted as completed by the development progress measures. Defects can be:

- Errors detected during testing. For example, the work item would be the coding of a unit whose exit criteria could be a successful compilation, unit test, and code inspection.
- Government comments on documents. For example, the work item could be the document, or a section of the document if it is a major document, whose exit criterion is its delivery.
- Action items from reviews. For example, the work item would be the product(s) being reviewed whose exit criterion is the review. Action items can result from major milestone reviews (e.g., PDR or CDR) or from internal or less formal reviews (e.g., inspections or peer reviews).

For program managers to measure the effort being consumed by rework, they need to require contractors to track the effort expended correcting defects.

3.6.3 Data Inputs and Collection (Software Defects)

The data inputs the program manager collects or has reported for the software defects measures are:

- Total number of defects opened and closed
- Number of defects opened and closed since the last report

Additionally, the data above should include the following information for each defect to allow proper analysis:

- Type (e.g., testing error, action item, or document comment)
- Classification and priority[3]
- Product in which the defect was found

Different types of defects should be tracked separately so they are distinguishable from each other (e.g., document comments and software testing errors should not be added together).

[3] For determining the classification and priority of defects, refer to Appendix C of DOD-STD-2167A [DOD2167A].

3.6.4 Sample Indicators, Analyses, and Actions (Software Defects)

The software defect measures can be used to determine the effectiveness of the software development process. For each product and type of defect, the program manager tracks:

- Reported defects
- Open defects: defects that have not been resolved
- Closed defects: defects that have been resolved and approved

From these, the program manager can determine the defect discovery rate and the current known quality of the software products. The program manager can use a chart similar to Figure 3-12 to make these determinations.

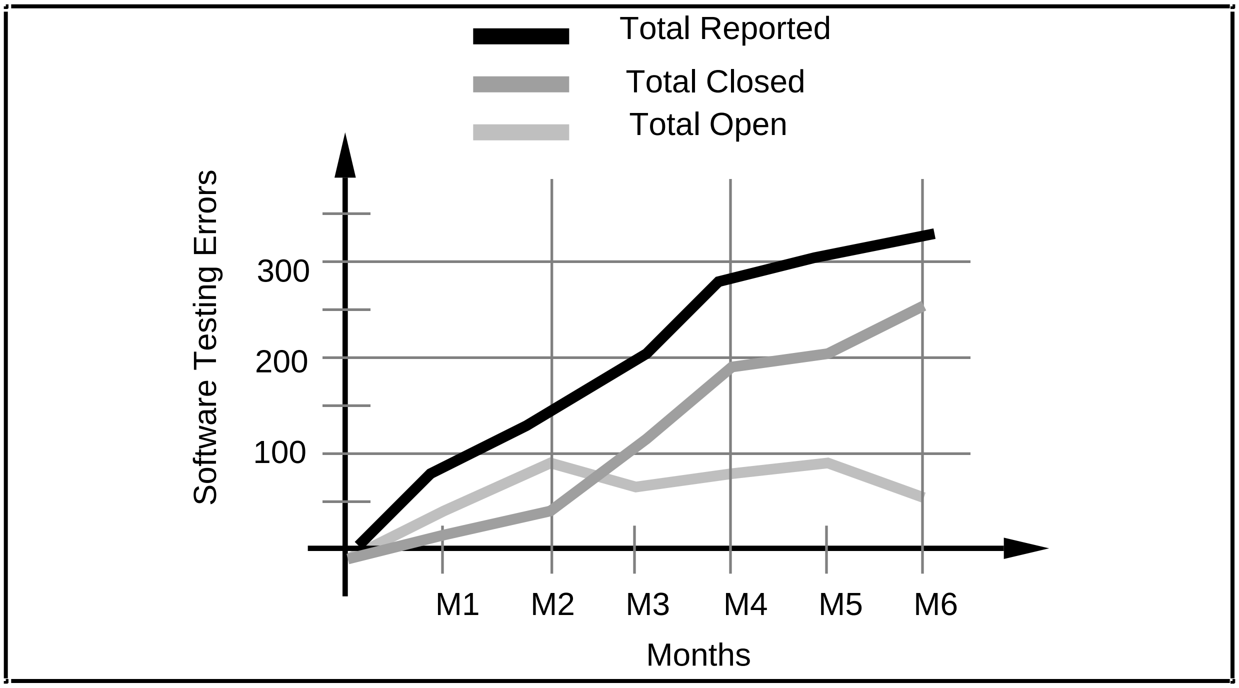


Figure 3-12 Sample Software Defects Measure

The number of defects reported can be used to determine the effort being consumed by rework by including the number of hours to process and correct each defect. The rate of closure and the trend of remaining unresolved defects can be used to measure the progress of the rework. The defects can be grouped and the progress of resolving the group(s) of defects can be tracked by the development progress measures.

The defect discovery rate can be tracked to determine the acceptability of products at a particular stage of the development cycle. When analyzing the discovery rate, the program manager needs to correlate the rate with the types of effort being applied to the contract. A declining number of reported defects may be caused by the reallocation of effort from discovering defects to correcting defects (e.g., the testing effort is being applied to correction activities and not to testing). A measure can be generated for other development resources (e.g., the number of development computers) and correlated to the defect discovery rate.

The program manager can also analyze the slope of each curve. If the slope of the open defect curve is positive, it could indicate that defects are being identified faster than they can be resolved; if the slope is negative, then the defect detection and correction process may be working or little effort is being expended to discover defects [BRETT], [MUSA].

Understanding both the data and what the data represents is clearly needed as shown in Figure 3-13. In this figure, the total number of defects discovered are being tracked. However, without knowing that the total includes the code related defects and non-code related defects, the program manager could be led to believe that many more defects are being discovered in the software than really exist. From such a figure, the program manager can also get information on the rate of discovery of defects. By correlating the figure to other measures such as the milestone performance and the effort measures, the program manager can get indications of impacts on the schedule.

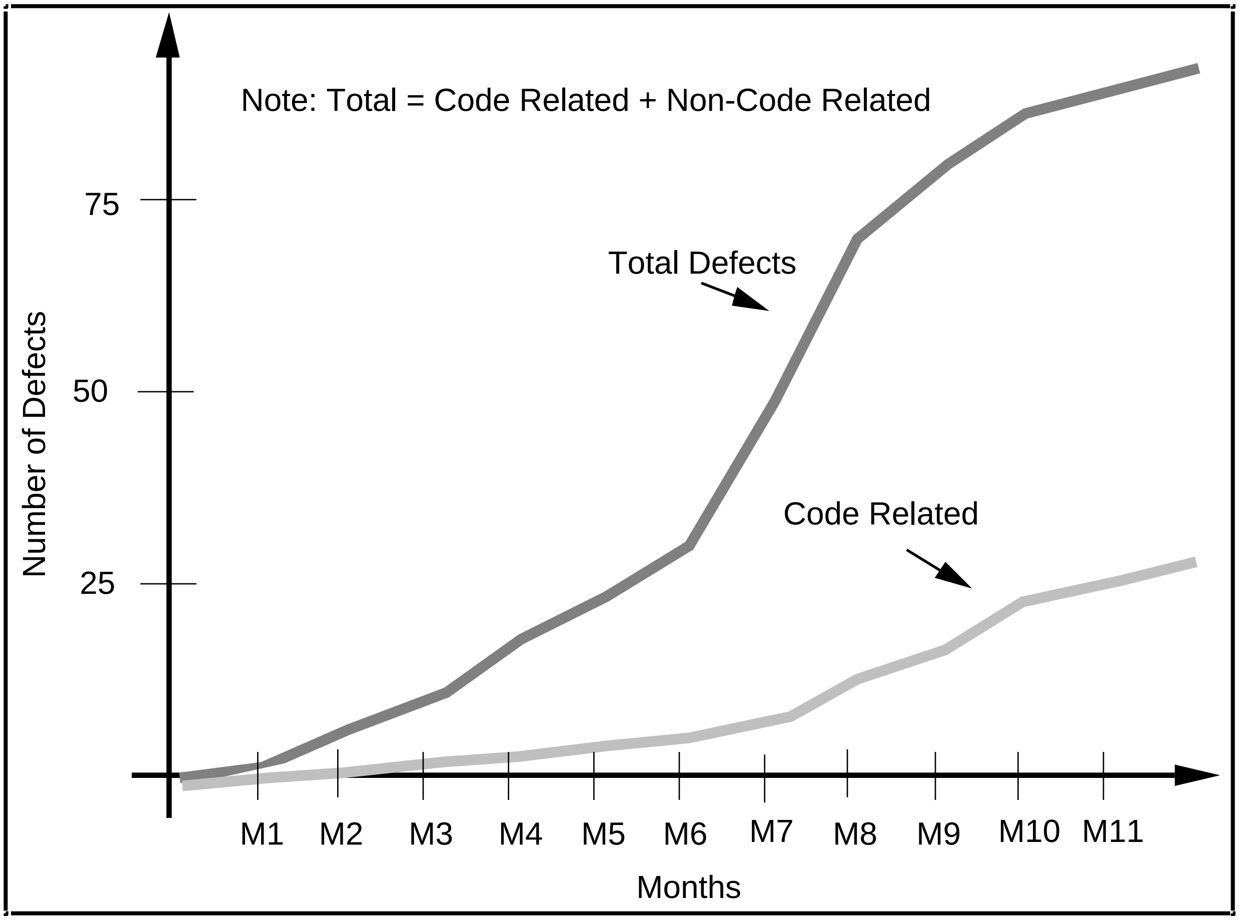


Figure 3-13 Example Illustrating the Need for Understanding Software Defects Discovered

Tracking the number of defects found during integration or acceptance testing can provide data for reliability models as well as information on the rate at which defects are being discovered during testing. By determining the software's defect density by normalizing the number of defects found during testing to the size measurement (Figure 3-14), the program manager also has an indication of testing adequacy and code quality.

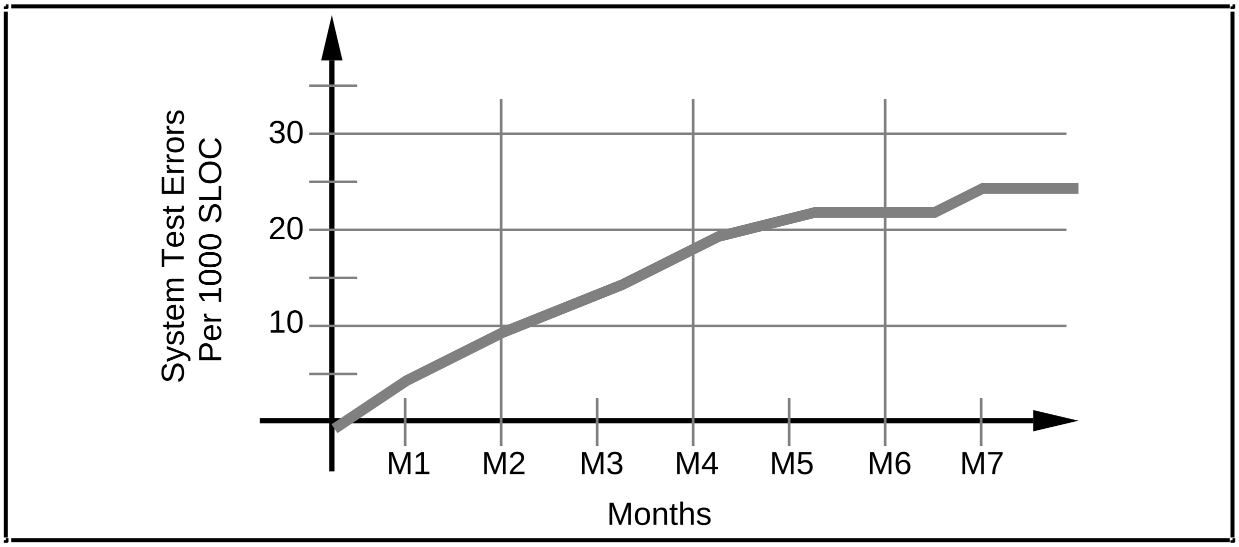


Figure 3-14 Sample Defect Density Measure

Increases in reported defects are frequently observed after major reviews (i.e., action items) and the start of testing activity (testing errors). If the increase is minor, it is necessary to investigate whether the product is of high quality or whether the review was ineffective.

The program manager needs to be concerned about the length of time that known defects remain open. The sooner defects can be detected, the lower the cost to correct them and the lower their impact on the schedule [BOEHM81]. The program manager needs to ensure that detected defects are corrected in a timely manner so that the risk impact is minimized. Figure 3-15 is an example indicator report that program managers can use to track the longevity of defect reports.

2167A Priority Levels	Number of Problem Reports That Have Been Open x Days				Totals
	x ≤ 30	30 < x ≤ 60	60 < x ≤ 90	x > 90	
Priority 1	2	1			3
Priority 2	3	1	1		5
Priority 3	3	2	1	1	7
Priority 4	4	3	3	2	12
Priority 5	8	6	3	3	20
Totals	20	13	8	6	47

Figure 3-15 Example Table Showing Longevity of Defects

Analyzing the root causes of defects can motivate an improvement process to prevent the introduction of errors [HUMPHREY89]. Analyzing the root causes of defects may also result in higher quality end-product and less product rework in later releases of a multi-release project.

The program manager can also use techniques such as Pareto analysis (as in Figure 3-16) to help isolate modules that are the most prone to error [ISHIKAWA]. If certain modules have more errors than others, those modules may be more complex, the functionality may not be completely understood [BURR], or the modules may be very large compared to those with few or no errors. The program manager could have the modules redeveloped (i.e., redesigned and recoded) or require more testing for the modules. For example, in Figure 3-16, CSCs D and F are responsible for 80 percent of the errors found. In this case, the program manager would have the contractor redesign those modules or apply more effort to them during testing.

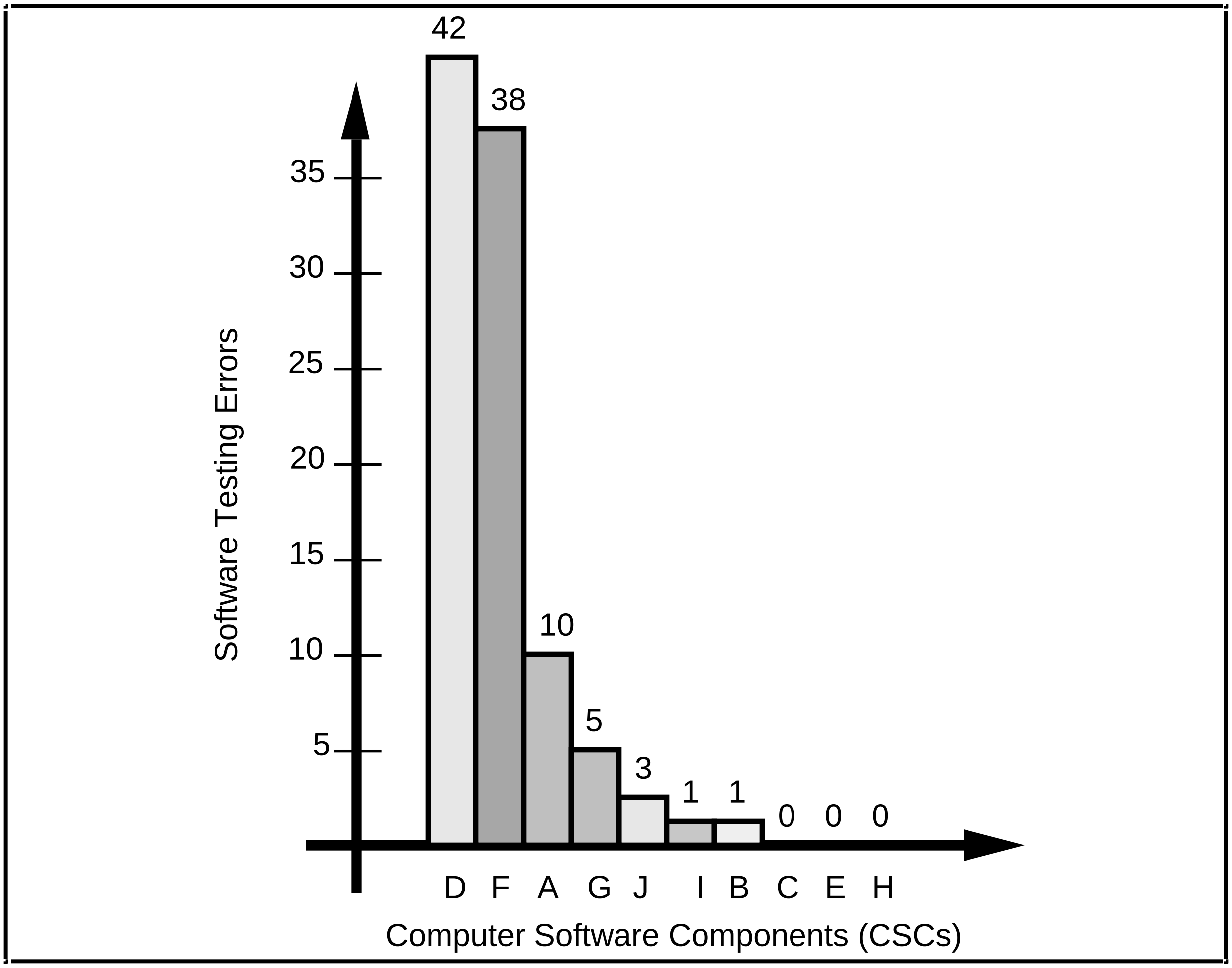


Figure 3-16 Example of Pareto Analysis Showing Defects Per CSC

3.6.5 Other Measurements and Partitions

Based on program issues, the program manager might also want to consider the following software defect measurements and partitions:

- Priority, severity, or criticality of defect
- Software language
- Development process or activity that caused the defect
- Development process or activity that found the defect
- Contractor or subcontractor
- Effort expended to close defects (or categories of defects, e.g., effort expended to close defects by various levels of defect severity)

3.7 Computer Resource Utilization

3.7.1 Purpose (Computer Resource Utilization)

Computer resource utilization (CRU) measures give the program manager an indication of the percentage of computer hardware resources used. The program manager is concerned about the use of computer resources because the software must operate within tangible hardware limits.[4] The program manager is also concerned that resources be available for future expansion of software functionality or for needed increases in software performance. The CRU measures track the use of a computer's processors, memory, mass storage devices, and input/output channel throughput. CRU growth should be reviewed and analyzed early in the program if spare capacities are of concern.

3.7.2 Description (Computer Resource Utilization)

The CRU measures track four categories of computer resources:

- Central processing unit (CPU) utilization: measures the percentage of available processing power used during worst-case software execution.
- Memory utilization: measures the percentage of total available computer memory process-resident software and data.
- I/O throughput: measures the speed and amount (number of bytes) of total throughput capacity used during worst-case data transfers.
- Mass storage utilization: measures the percentage of total storage used at peak residency.

The measures may be applied to development computer and target computer depending on criticality.

The program manager determines how much spare capacity is needed by considering the software's purpose and future. For example, if the software's purpose is an overnight processing of database reports for an inventory management system, spare capacities may be small because performance may not be an issue. In this case, expansion could be achieved by adding another computer. However, if the software is for the flight control system of a plane, the spare capacities may need to be large for increased performance or to allow for future expansion of the software because another computer is not easily added.

[4] In some systems, the program manager may choose not to use this indicator because it is not important to the system. However, in mission critical computer resource (MCCR) applications, the program manager should include it.

3.7.3 Data Inputs and Collection (Computer Resource Utilization)

For the CRU measures, the program manager has the following data inputs reported via a CDRL item:

- Total available capacity of each resource
- Current measurements
- Estimates projected to a designated milestone for each resource

Example of the units for each computer resource data input are:

- CPU throughput capacity: millions of instructions per second (MIPS)
- Memory: the kilobytes or megabytes for each type of memory (e.g., RAM or ROM)
- I/O throughput: bytes per second (BPS)
- Mass storage devices: kilobytes or megabytes for each device

The program manager determines the required spare capacities and has estimates made for the use of the resources until actual data are available. Estimates should be replaced with simulated results and actual data as they become available. Extremely important to the program manager are the method of calculating the data and the assumptions used to develop resource loadings for the "worst-case scenario." The program manager needs to ensure that the method used is valid and verifiable. Examples of methods include analysis by comparison, simulation analysis, or demonstration of the possible scenarios of resource use.

The earliest estimates of processor throughput availability should be carefully specified in terms of instruction mix and other benchmark assumptions until these estimates can be replaced by measurements.

Monthly tracking against plans should start at PDR and continue throughout the development effort.

3.7.4 Sample Indicators, Analysis, and Actions (Computer Resource Utilization)

Early insight into the resource demands of the software will highlight any inadequacy of the planned resources. Early CRU assessments are vital for long lead time procurements of special purpose processors, memories, buses, and mass storage devices. This is particularly important when there are physical or other constraints on the amount of resources that can be provided. Excessive use of computer resources contributes to increased schedule delay, increase of development cost, lowered reliability, potential loss of system functionality, and expensive software or system redesign.

The program manager tracks the CRU measures using a graph similar to Figure 3-17. Spare capacity levels for target CPU should be established to allow for software upgrades after the software has been released. The program manager monitors the use of resources to verify that spare capacities are sufficient. For systems where resources are critical, a spare capacity requirement lower than 50 percent must be carefully weighed in terms of cost, feasibility, and operational suitability. If a spare capacity (i.e., reserve) lower than 50 percent is established, there should be rigorous formal tracking and proactive risk management. Sometimes system engineering tradeoffs do not permit reasonable spare capacity targets. The program manager might then manage this risk formally by requiring quantitative allocations of resource capacity to CSCIs, CSCs, and management of these allocated resources.

When use of resources is near the specified upper bound for physically constrained computer resources, this measure should be correlated with the schedule and development progress measures to assess the potential impact of any necessary redesign.

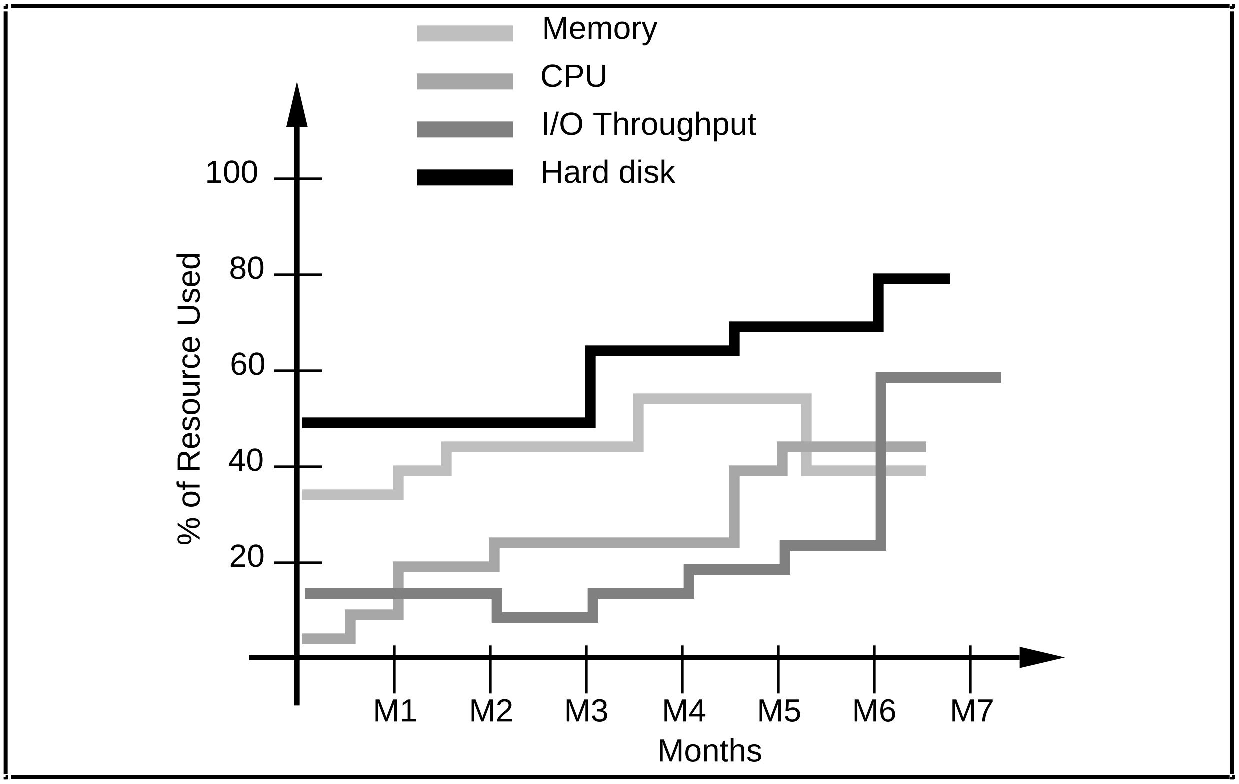


Figure 3-17 Sample Computer Resource Utilization Measure

The I/O resource is the hardest to measure fully because there may be many I/O measurements to consider. These include disk channel capacities and rates for serial data, parallel data, disk access, printers, plotters, local network data, I/O data bus, and special peripheral data. The I/O resource to be tracked must be defined if it can adversely affect system operation.

When system utilities cannot provide the CPU processor time used during execution of specific tasks, the program manager should consider the use of a "time burner" stub during design to simulate and estimate via a prototype the worst-case amount of time a subroutine is expected to take to execute after it has been fully coded. During verification testing, such a stub can be used as a test driver to set up a high priority task and increase appropriate time delays while looking for system degradation effects.

When the estimated use of resources exceeds management criteria for spare capacities, the following actions might be considered.

- Optimizing the software: the software and/or storage devices could be optimized using commercially available utility software.
- Redesigning the software: time critical functions could be redesigned and recoded in assembly language; time "hogging" functions could be redesigned for optimal processing; and time consuming data transfers could be redesigned using double buffering design techniques.
- Adding computer resources: if feasible, faster processors, more memory, larger disks, etc. could replace slower or smaller devices.
- Accepting loss of mission or support functionality: if necessary, functionality may need to be deferred to later releases or eliminated.
- Changing system requirements: as a last resort, the system requirements could be revised and alternatives considered. The system may not be possible within certain usage constraints.

Carefully defined and collected CRU data can provide a valuable historical basis for improving the accuracy of future estimates of computer resources, the effectiveness of methods for data collection and estimation, and the choices for management reaction criteria.

Other measurement correlations that the program manager can use include:

- Check data from the defect measures regarding defects related to resource utilizations.
- Investigate rework to determine if resource utilization problems are causing unplanned staff growth and excessive effort expenditures.
- Scrutinize size allocations to determine if size growth is, or will be, within hardware resource limits.
- Probe schedule and progress reports to assure that resource utilization plans are not impacting delivery dates and commitments.

3.7.5 Other Measurements and Partitions (Computer Resource Utilization)

The program manager might also want to consider the following CRU measurements and partitions:

- Absolute counts instead of percentages of CRU units to highlight changes in total reallocated available resources.
- No-load and average-load in addition to worst-case assumptions for resource loading scenarios to highlight the impact of different scenarios of resource loading.
- Separate reports on use of resources in a multi-resource architecture that has dedicated functions to highlight the impact of different dedicated functions on CRU.

4. Other Sample Analysis Techniques

A good software development process yields good quantitative data. Good quantitative data, in turn, provides information that contributes to successful software program management. Analyzing measurement data will provide the program manager insight into the software development process throughout the development life cycle.

Based on the program manager's issues, indicators are derived from low-level measurement data and analyzed to gain insights into the program. The program manager tracks the plans and estimates against the actual data as it becomes available. The program manager then extrapolates trends in the actual data to estimate future performance and progress and to determine if the trends mitigate known risks or expose new ones. Several analysis techniques can be used to maximize the insight achieved. The other techniques discussed are:

- Trend analysis. For example, plot and analyze the number of CSUs completing unit test.
- Multiple metric relationship analysis. For example, plot the current staffing plan through completion and compare it to the scheduled tasking to assess whether the planned staffing is realistic and adequate.
- Modeling input data analysis. For example, use a model such as the constructive cost model (COCOMO) or other commercially available tools to generate estimates and to extrapolate data to predict future performance.
- Thresholds and warning limits. For example, set thresholds around planning curves, then analyze the variability of the actual data using the thresholds as warning signals when the actual data approaches or crosses them.

4.1 Trend Analysis

Trend analysis is a basic technique for gaining insight into a program. Consider the example trend graph shown in Figure 4-1 which illustrates the number of problem reports and indicates their status. One purpose of this measure is to support the management and assessment of cost and schedule risk (i.e., if a large number of unresolved problems is allowed to accrue, cost and schedule overruns may result by the time the problem reports are finally addressed).

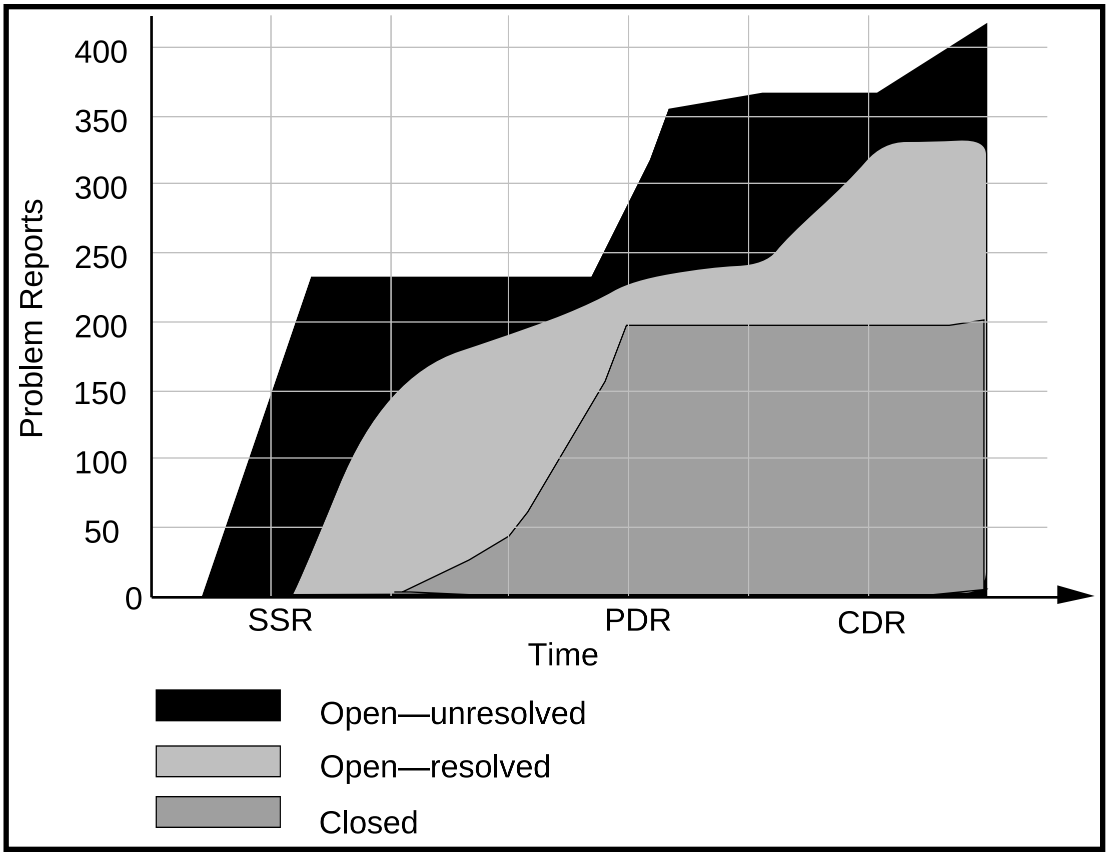


Figure 4-1 Example of Single Parameter Trend Analysis

Based on this graph, the program manager can assess whether problem report status poses a cost or schedule risk to the program. An analysis of the graph shows a large number of new problem reports around SSR, with successively smaller jumps around PDR and CDR. It also shows that the resolution rate has nearly kept pace with the new problem report identification rate. The one possible cause for concern is the closure rate, which shows that half of the total problem reports remain open, and none have closed since PDR. In the example, the failure to close problem reports should be investigated; however, given that most of the open problem reports have been resolved, they do not appear to be a significant cause of risk.

Trend analysis can also be used to compare plans with actual data. These analyses can encompass data for plans, plan changes, actuals, actual changes, projections, and projection changes. In addition to providing specific insight into the aspect of the program being measured, planned versus actual measures provide an indication of the maturity and reliability of the planning and estimating process.

Consider the CSU development progress measure shown in Figure 4-2. This measure shows planned, replanned, and actual completion rates for design, code/unit test, and integration of CSUs. The measure also shows rework. The primary uses of this measure are to assess the schedule risk and measure real technical progress. This graph shows that there was one replan just prior to CDR, which appears to have been caused by a 10 percent increase in the total CSUs defined. Actual design completion slipped another month beyond the replan. CDR, coding, and integration all began as originally scheduled. The main problem appears to be in integration progress, which is below the plan and progressing at a declining rate. The level of redesign and recoding during integration also raises a warning flag. Armed with this measure , the program manager can ask specific questions (such as queries into the nature of the rework performed) to identify the source of the integration problem.

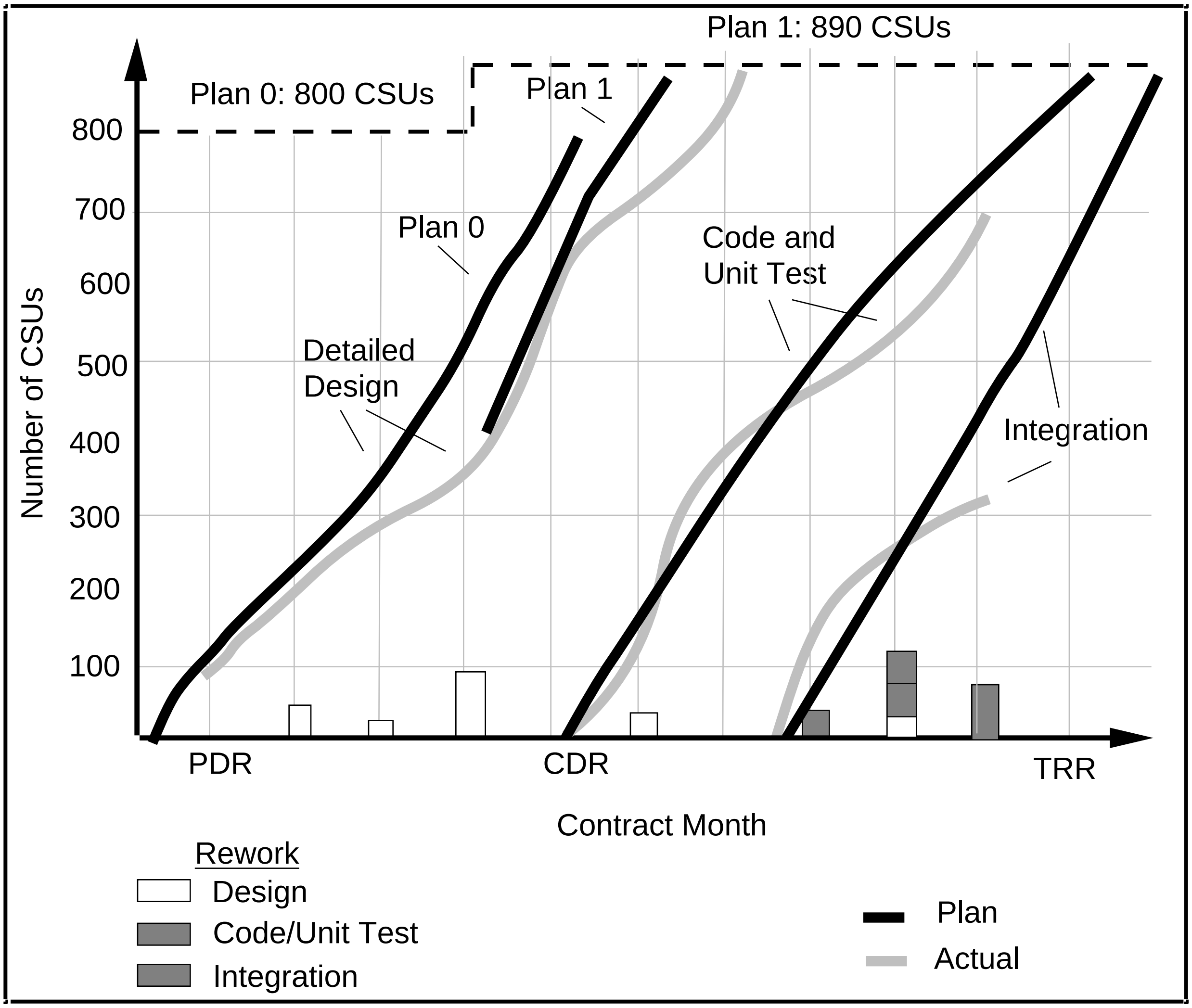


Figure 4-2 Example of Multiple Parameter Trend Analysis

4.2 Multiple Metric Relationship Analysis

The program manager should correlate the trends from all of the measures before making decisions based on them. Analysis of relationships between multiple measures is an essential tool for correlating such trends and for obtaining and confirming program insights. Often the benefit of analyzing the relationship between multiple measures is greater than the sum of the individual benefits of the same measures. An example of this is shown in Figure 4-3, where the number of software development personnel overlays the schedule. Notice that the design and implementation activities planned are highly parallel even as planned staffing is decreasing. Assuming the budget is based on the planned staffing shown, this program exhibits significant cost risk, since it is unrealistic to assume that the schedule can be maintained with the planned decrease in staffing. Also, the program has been able to stay on schedule by using more staff than planned, but just as schedule activities intensify, the staffing levels are beginning to drop below the planned level. If each of these measures were only considered independently, it is possible that the program manager would overlook the apparent inconsistency in the plans.

As shown by the example, the technique is simple to use, yet very powerful and valuable as a tool for assessing plans. Measures based on data from plans will generally exhibit one of three relationships with each other:

- Positive trend relationships, where both measures are expected to track in a consistent direction. Where the expected tracking does not occur, such as in the example above, further investigation by the program manager is warranted.
- Inverse relationships, where there is an underlying tradeoff implicit in the things being measured. A simple example is estimated cost and estimated SLOC. If the developer determines that a portion of the originally planned level of reusable software will have to be new SLOC, a negative cost impact would be expected.
- Independence, where there is no presumed relationship between the measures.

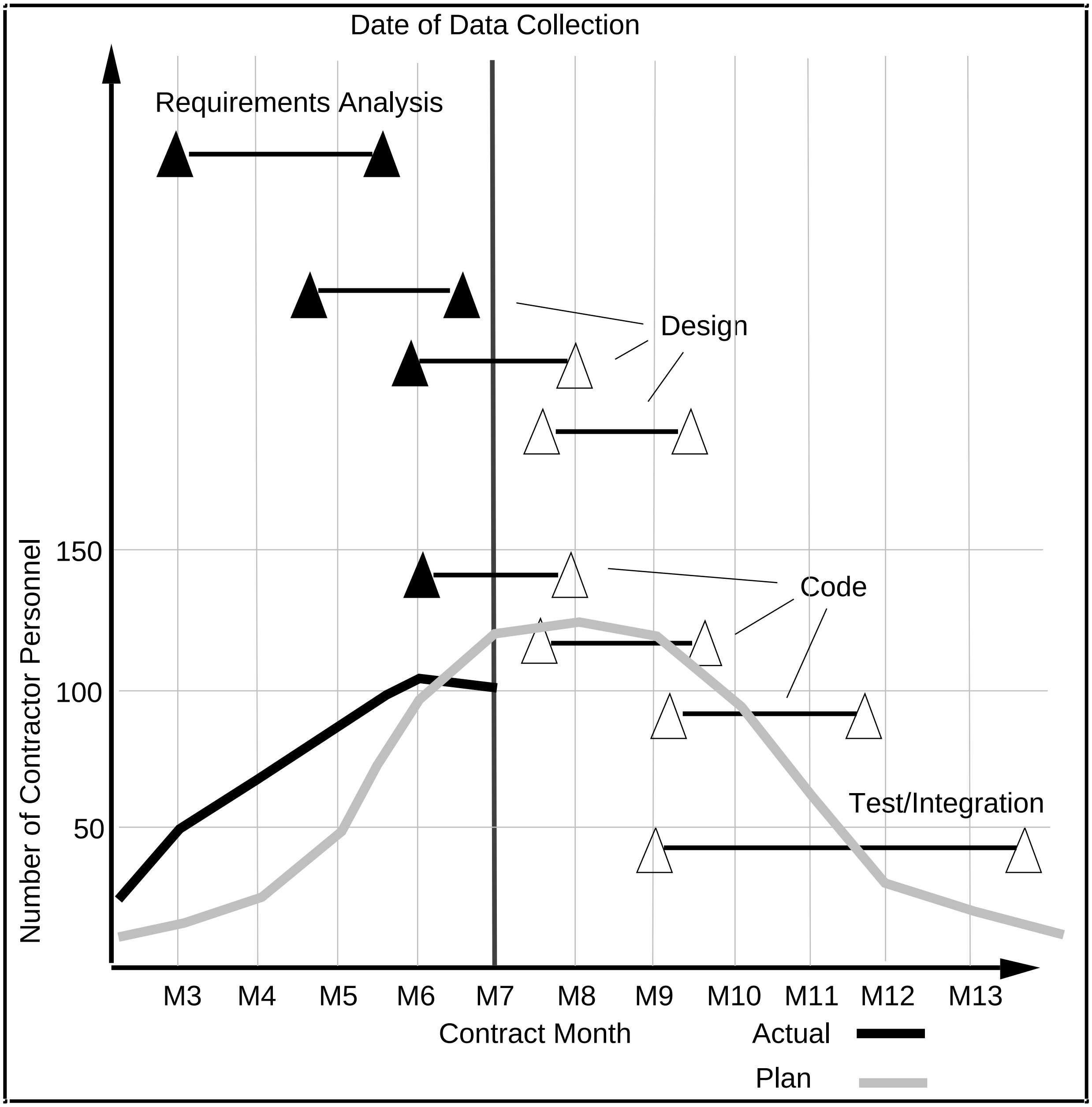


Figure 4-3 Example of Multi-Metric Relationship Analysis

4.3 Modeling Input Data Analysis

Use of models fits naturally with software measurement. Models provide specialized algorithms for predicting or estimating certain key characteristics of a software program. The most common examples are cost models and reliability models. Often these models can use the same data that is being collected for use in generating the other measures discussed previously.

Models often use data from similar, past projects as bases for comparisons and to guide decisions on the input data. Such historical data is critical in validating the model used. But because no two programs are alike, care must be used to not overemphasize the model's output. Because of the emphasis on understanding inputs to models, modeling becomes an important analysis tool for understanding contractor estimates and forecasts.

Cost models typically provide predictions of cost and schedule based on a set of input parameters, such as lines of code, labor rates, quality requirements, application characteristics, environment characteristics, and demonstrated past performance. While the actual value of certain input parameters (especially lines of code) is not available until late in the program, early estimates of these parameters have been successfully used with many cost models.

Reliability models typically estimate the number of operational failures that will occur per unit of time. The primary drawback of the most common reliability models is the unavailability of input parameter data (or reasonable estimates) until late in the development of a software system. Most reliability models use test results (failure data during operational testing) as the key input parameter. For this reason, the main benefit of reliability models occurs in the testing stages and later during post deployment software support. A typical application of a reliability model is as an aid in determining when testing can stop.

An example analysis using a cost model (in this case COCOMO) is shown in Figure 4-4. This example shows the results of independent predictions of cost and schedule by the software development contractor, the government, and an independent estimator. Given the significant difference in the predicted cost and schedule, further investigation of the underlying assumptions is warranted. Figure 4-5 shows a possible result of such an investigation, based on the government versus contractor assumptions of the input parameters driving the cost model. An item by item review with the contractor can be conducted to understand and evaluate the rationale for the input values used.

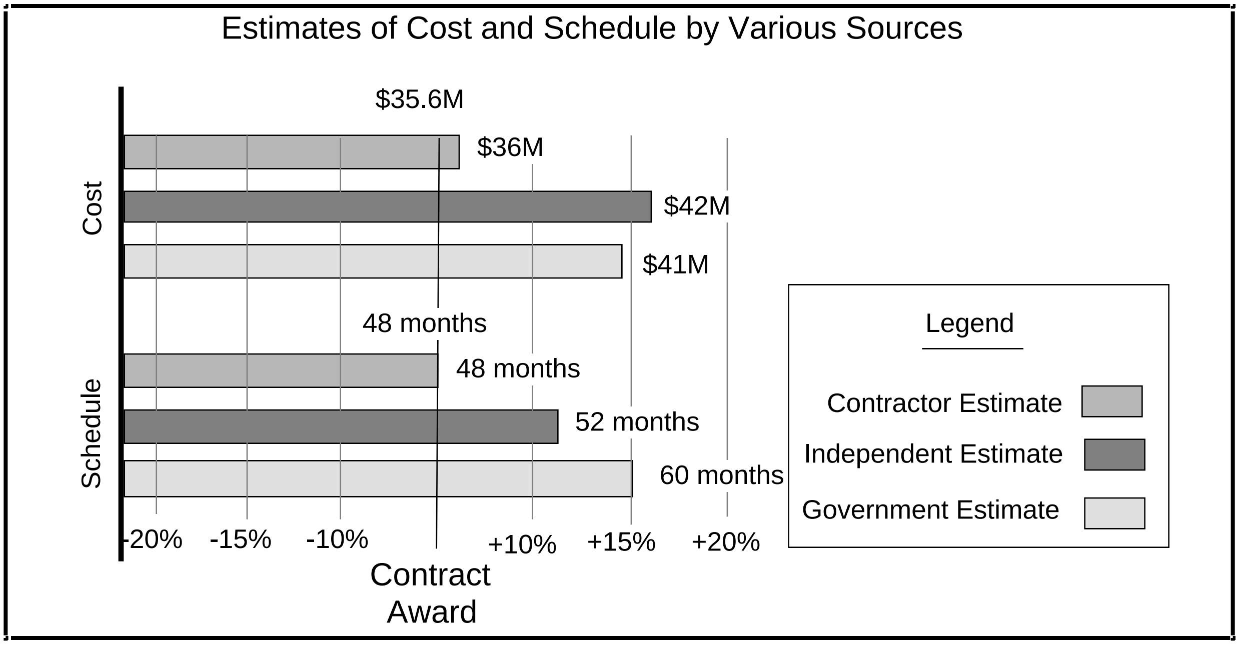


Figure 4-4 Example of Modeling Input Data Analysis Using COCOMO

COCOMO Model Analysis

Driver assumptions used by government and contractor for their estimate

	Contr	Gov't
ACAP	HI	NML
AEXP	NML	NML
PCAP	NML	LO
VEXP	NML	LO
LEXP	NML	LO
Size =	200K	250K

	Contr	Gov't
RELY	HI	HI
DATA	NML	HI
CPLX	HI	HI
TIME	NML	NML
STOR	NML	NML
Mode =	Semi	Semi

	Contr	Gov't
VIRT	HI	HI
TURN	NML	HI
MODP	HI	NML
TOOL	LO	LO
SCED	NML	NML

Staff Months

	Contr	Gov't
RP	4.8	233.3
PD	201.0	298.1
DD	331.5	618.1
CT	466.1	1048.6
IT	498.9	1368.9
Total	1601.9	3566.8

Cost (%M)

Contr	Gov't
.786	1.750
1.507	2.236
2.486	4.636
3.496	7.865
3.739	10.264
12.015	26.750

Schedule (Months)

Contr	Gov't
6.5	5.1
8.2	7.4
7.8	9.2
8.2	10.8
13.0	21.7
43.6	54.3

Staff

Contr	Gov't
16.2	45.5
24.6	40.3
42.6	66.9
57.1	96.8
38.2	63.1
36.7	65.7

Figure 4-5 Example of Modeling Input Data Analysis Results of Government Versus Contractor Estimates

4.4 Thresholds and Warning Limits

The program manager can also use an analysis method that is similar to statistical process control to help analyze the variability of the data received based on preconceived ideas of when an identified issue has become (or is about to become) a problem. The basic technique uses an adaptation of statistical process control charts. However, the program manger usually does not have a sample of data from which to apply statistical methods such as control limits; instead, the program manager uses subjective thresholds of variability about the plan data. That is, the program manager uses the plan data (or if plan data does not exist, determines a goal, e.g., the number of software defects) and applies ranges of variability whose boundaries are predetermined thresholds. These charts then allow the program manger to see how the contractor is performing relative to the thresholds regarding each issue.

Each measure would have an upper and lower threshold (UT and LT) and upper and lower warning limits (UWL and LWL). The program manager would predetermine the threshold limits and warning limits. These predeterminations could be dependent upon the priority of the issue, the amount of risk that is associated with an issue, and the criticality or impact an issue might have on the eventual outcome of the system (i.e., the potential loss). In Figure 4-6, the UT and LT are set at plus and minus 20 percent of the plan data and the UWL and LWL are set at plus and minus 10 percent. In practice, the thresholds and warning limits do not have to be symmetrical, e.g., an upper limit could be plus 10 percent and a lower limit could be minus 25 percent.

In Figure 4-6, productivity is plotted as staff hours per thousand function points as an example of how to apply the technique. Whenever the actual data curve falls outside the warning levels, the program manager should interpret it as a signal that potential problems are starting to show up and further investigation is needed. Seemingly good trends, such as the actual data curve being higher than the UWL, should also be investigated. The program manager investigates trends by asking probing questions and analyzing the data and contract information more closely.

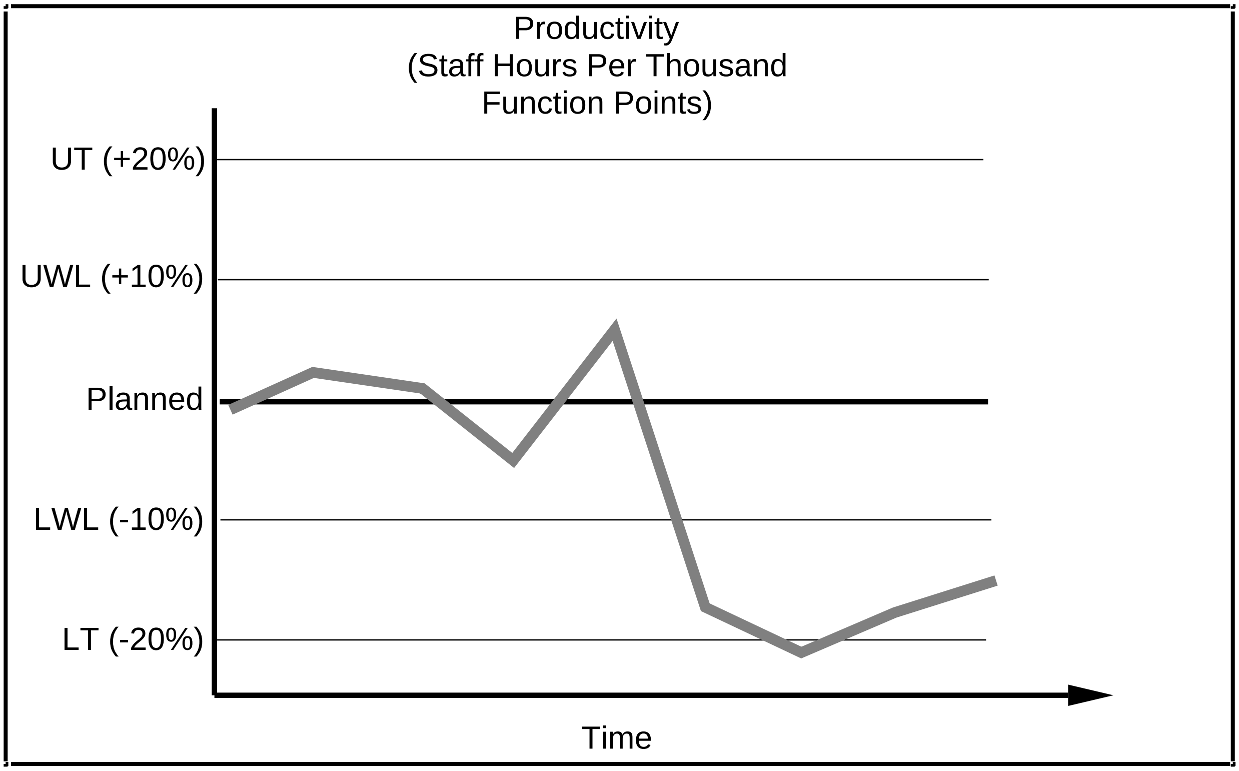


Figure 4-6 Example of Statistical Analysis Using Productivity

References

[AFSC] Software Management Metrics, AFSC Pamphlet 800-43, August 31, 1990.

[BASILI] Basili, V. and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environment," *IEEE Transactions on Software Engineering*, June 1988.

[BEAM] Beam, W. R., J. D. Palmer, and A. P. Sage, "Systems Engineering for Software Productivity," *IEEE Transactions on Systems, Man, and Cybernetics*, March/April 1987.

[BOEHM81] Boehm, B., *Software Engineering Economics*. Englewood Cliffs, New Jersey: Prentice-Hall, 1981.

[BOEHM87] Boehm, B., "Understanding and Controlling Software Costs," *IEEE Transactions on Software Engineering*, October 1988.

[BRETT] Brettschneider, R., "Is Your Software Ready for Release," *IEEE Software*, July 1989.

[BROOKS82] Brooks, F. P. Jr., *The Mythical Man-Month: Essays on Software Engineering*. Reading, Massachusetts: Addison-Wesley Publishing Co., 1982.

[BROOKS87] Brooks, F. P., "No Silver Bullet: Essence and Accidents on Software Engineering," *IEEE Computer*, April 1987.

[BROWN] Brown, J. R. and M. Lipow, *The Quantitative Measurement of Safety and Reliability*, TRW Report QR 1776, August 1973.

[BURR] Burr, T., "The Tools of Quality Part VI: Pareto Charts," *Quality Progress*, November 1990.

[CONGRESS] "Bugs in the Program: Problems in Federal Government Computer Software Development and Regulation," Staff Study by the Subcommittee on Investigations and Oversight transmitted to the Committee on Science, Space, and Technology - U.S. House of Representatives 101st Congress 1st Session, September 1989.

[CORI] Cori, K., "Fundamentals of Master Scheduling for the Project Manager," *Project Management Journal*, June 1985.

[DEMARCO] DeMarco, T., *Controlling Software Projects*. New York: Yourdan Press, 1982.

[DOD2167A] *Military Standard - Defense System Software Development*, DOD-STD-2167A, February 1988.

[DOD5000.1] *Department of Defense Directive - Major Systems Acquisitions,* DOD-DIR 5000.1, February 1991.

[HUMPHREY] Humphrey, W. S., *Managing the Software Process*. Reading, Massachusetts: Addison-Wesley Publishing Co., 1989.

[IEEE1045] *Standard for Software Productivity Metrics*, IEEE P1045/D4.0, December 1990.

[IFPUG] *Function Points as Assets-Reporting to Management,* The International Function Point User Group, February 1991.

[ISHIKAWA] Ishikawa, K., *Guide to Quality Control.* Tokyo: Noridca International Limited, 1989.

[JURAN] The Juran Institute, "The Tools of Quality; Part V: Check Lists," *Quality Progress*, October 1990.

[MUSA] Musa, J. D., A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*. New York: McGraw Hill, 1987.

[ONEILL] O'Neill, D. and C. Ingram, "Software Inspections Tutorial," *SEI Technical Review*, 1988.

[PACKARD] Packard, D., *Quest for Excellence, Final Report to the President,* D. Packard on behalf of the President's Blue Ribbon Commission on Defense Management, Superintendent of Documents, GPO, Washington, D.C., 1986.

[SCHULTZ] Schultz, H., *Software Management Metrics*, ESD-TR-88-001, May 1988.

[SEI89] *Proceedings of the Workshop on Executive Software Issues August 2-3 and November 18, 1988*, SEI Technical Report, CMU/SEI-89-TR-6, ADA 206779, January 1989.

[SHEWART] Shewart, W. A., *Statistical Method From the Viewpoint of Quality Control*, The Graduate School - The Department of Agriculture, Washington D.C., 1939.

[USAF] *Report of the DoD Joint Service Task Force on Software Problems*, Lt. Col. Larry E. Druffel, USAF - Task Force Chairman, July 1982.

[WALTON] Walton, M., *The Deming Management Method*. New York: The Putnam Publishing Group, 1986.

Acronyms

BPS	Bytes Per Second
CDR	Critical Design Review
CDRL	Contract Data Requirements List
CM	Configuration Management
COCOMO	Constructive Cost Model
COTS	Commercial Off the Shelf
CPU	Central Processing Unit
CRU	Computer Resource Utilization
CSC	Computer Software Component
CSCI	Computer Software Configuration Item
CSU	Computer Software Unit
DoD	Department of Defense
FCA	Functional Configuration Audit
GFE	Government Furnished Equipment
GFI	Government Furnished Information
IDD	Interface Design Document
IEEE	Institute of Electrical and Electronics Engineers
I/O	Input/Output
IRS	Interface Requirements Specification
IV&V	Independent Verification and Validation
KSLOC	Thousands of Source Lines of Code
LT	Lower Threshold
LWL	Lower Warning Limit
MCCR	Mission Critical Computer Resources
MIPS	Millions of Instructions Per Second

PCA	Physical Configuration Audit
PDCA	Plan-Do-Check-Act
PDR	Preliminary Design Review
RAM	Random Access Memory
ROM	Read Only Memory
RFP	Request For Proposal
SAMWG	Software Acquisition Metrics Working Group
SDD	Software Design Document
SDP	Software Development Plan
SDR	System Design Review
SEI	Software Engineering Institute
SLOC	Source Lines of Code
SPR	Software Problem Report
SQA	Software Quality Assurance
SRS	Software Requirements Specification
SSR	Software Specification Review
SSDD	System/Segment Design Document
SSS	System/Segment Specification
TR	Technical Report
TRR	Test Readiness Review
USAF	United States Air Force
UT	Upper Threshold
UWL	Upper Warning Limit
WBS	Work Breakdown Structure

Table of Contents

List of Figures

Acknowledgements

The SEI measurement efforts have depended on the participation of many people. The Software Process Measurement Project thanks the members of the Software Acquisition Metrics Working Group who contributed to the content and structure of this document. The SEI is indebted to them and to the organizations who sponsored their participation in this effort to incorporate measurement into the practices of acquisition program managers. Without their participation, the SEI could not have completed this task. The members of the Software Acquisition Metrics Working Group are:

A. Frank Ackerman
Institute for Zero Defect Software

Mark Amaya
McDonnell Douglas Corporation

Miguel Carrio
Teledyne Brown Engineering

Andrew Chruscicki
USAF Rome Laboratory

Charles Cox
US Naval Weapons Center

Joseph P. Dean
Tecolote Research, Inc.

Peter Dyson
Software Productivity Solutions

Stewart Fenick
US Army Communication and Electronics Command

Ronald Gulezian
Drexel University

Capt. Marc L. Hoffman
USAF - Air Force Systems Command

Steve Keller
Dynamics Research, Inc.

Jim Keng
McDonnell Douglas Corporation

Jeffrey Lasky
Rochester Institute of Technology

Janet MacLaughlin
US Naval Ocean Systems Center

John J. McGarry
US Naval Underwater Systems Center

Alfred H. Peschel
TRW, Systems Development Division

Sam T. Redwine, Jr.
Software Productivity Consortium

Alan J. Roberts
Defense Systems Management College

James A. Rozum
Software Engineering Institute

Herman P. Schultz
MITRE Corporation

Raghu P. Singh
US Navy Space and Warfare Command

William Spaulding
Dynamics Research, Inc.

The SEI Software Process Measurement Project thanks all the individuals who reviewed the October 1991 draft and took the time and care to provide written comments. Those comments were very important in transforming the draft into this technical report. People who provided comments are:

Emanuel Baker
Software Engineering Consultants, Inc.

Lyle Cocking
General Dynamics

Barry Corson
US Naval Air System Command

Dean Dubofsky
MITRE Corporation

Stewart Fenick
US Army Communication and Electronics Command

Paul Funch
MITRE Corporation

John Harding
Groupe Bull

Donna Hinrichs
US.Air Force

Theresa Huber
DSD Laboratories, Inc.

Charles Koch
US Naval Air Development Center

Harry Larson
Larbridge Enterprises

Donna Lindskog
University of Regina and SaskTel

John Marciniak
National Security Industries Association

Richard Murphy
Software Engineering Institute

Jacquelyn Nixon
US Marine Corps (MCTSSA)

Don O'Neill
O'Neill Consultants

Anne Quinn
DCMDM-EE

Donald Reifer
Reifer Consultants, Inc.

Norman Schneidewind
Naval Post Graduate School

Mark Servello
American Management Systems, Inc.

Terry Wilcox
Defense Plant Representative Office - General Dynamics

The project also thanks the members of the Measurement Steering Committee for their many thoughtful contributions. Their insight and advice have been invaluable. This committee consists of the following representatives from industry, government, and academia who have earned solid national and international reputations for their contributions to measurement and software management:

William Agresti
MITRE Corporation

Henry Block
University of Pittsburgh

David Card
Computer Sciences Corporation

Andrew Chruscicki
USAF Rome Laboratory

Samuel Conte
Purdue University

Bill Curtis
Software Engineering Institute

Joseph P. Dean
Tecolote Research

Stewart Fenick
US Army Communication and Electronics Command

John Harding
Groupe Bull

Frank McGarry
NASA Goddard Space Flight Center

John J. McGarry
US Naval Underwater Systems Center

Watts Humphrey
Software Engineering Institute

Richard Mitchell
US Naval Air Development Center

John Musa
AT&T Bell Laboratories

Alfred H. Peschel
TRW, Systems Development Division

Marshall Potter
Department of the Navy

Samuel T. Redwine, Jr.
Software Productivity Consortium

Kyle Rone
IBM Corporation

Seward (Ed) Smith
IBM Corporation

Norman Schneidewind
Naval Post Graduate School

Herman P. Schultz
MITRE Corporation

Robert Sulgrove
NCR Corporation

Ray Wolverton
Hughes Aircraft

As the members of the Software Acquisition Metrics Working Group prepared this report, they were aided in their activities by the able and professional support staff of the SEI. Special thanks are owed to Mary Beth Chrissis and Suzanne Couturiaux, who were instrumental in getting the report into final form; to Marcia Theoret and Lori Race of the Software Process Measurement Project, who coordinated many of our meeting activities and provided outstanding secretarial services; and to Helen Joyce and her assistants from Event Management, who so competently assured that meeting rooms, lodgings, and refreshments were there when we needed them.

Software Measurement Concepts for Acquisition Program Managers

Abstract. For program managers to effectively manage and control software development, they need to incorporate a measurement process into their decision making and reporting process. Measurement costs money, but it can also save money through early problem detection and objective clarification of critical software development issues. This report provides some basic concepts that program managers can use to help integrate measurement into the process for managing software development. It also provides an initial set of measures to help address common issues in software intensive acquisitions.

When the Software Acquisition Metrics Working Group first met in 1989, only a few reports existed on the subject of how program managers could use software measurement; now, other reports have been written. The goal of this report is not to compete with those reports, but to use them as starting points for expansion. This report should be viewed not as a standard but as containing guidelines and advice for program officers and managers starting to use software measurement in their own organizations.

1. Executive Summary

1.1 Why Should Acquisition Program Managers Use Software Measurement?

Software contracts are difficult to manage. There are many complex inter-relationships between the factors, assumptions, estimates, and unknowns during the planning and execution of software contracts. Contributing equally to the difficulty of managing the contracts are ever changing constraints, technology, and acquisition environments. These all contribute to the difficulty of making good, timely decisions that guide a program toward successful completion. With software measurement, program managers have information that leads to early insight of potential problems as well as information that can support decisions. Without software measurement, program managers do not have the insight to identify and resolve problems, and as a result, they make decisions to react to events and problems rather than to curtail those problems before they surface.

Managers use measurement as feedback to control their projects. Without measurement, there is usually no way to know the status of a project along its many dimensions: cost, schedule, product performance, supportability, quality, etc. Thus, it is difficult to know what actions to take, what decisions to make, or how to correct unexpected outcomes. To benefit from measurement, managers need to link software measurement to the goals and risks of the program [WALTON].

> "…You cannot just use numbers to control things. The numbers must properly represent the process being controlled, and they must be sufficiently well defined and verified to provide a reliable basis for action. While process measurements are essential for orderly improvement, careful planning and preparation are required or the results are likely to be disappointing." [HUMPHREY]

The successful application of software measurement depends on having well-established measurement goals. A software measurement process that identifies the issues and uses them to identify what data and measurements to collect will provide better insight into the key program issues and help program managers make informed decisions. No single measurement will meet a program manager's needs [BROOKS87]. Therefore, the program manager needs a framework for defining what data to collect and how to analyze the data after it is collected.

> "The data collection process must be driven by the information from questions that we formulate based on our needs. In short, know what question is to be answered before collecting data." [JURAN]

A myth exists that program managers can just collect data and plot graphs to form an effective measurement process. Although possibly true in a static, precedented software development environment, this is not the case for the majority of U.S. Department of Defense (DoD) software acquisitions. The measurement process must be dynamic because of the constantly changing issues and complexities in DoD contracting. To be effective for the program manager, the measurement process must be a complete Plan-Do-Check-Act (PDCA) process that is integrated into the program office structure and used as a constructive tool for communicating between the program office and the contractor [SHEWART]. The PDCA cycle that the program manager uses is described as follows:

Plan: Identify the issues or questions the program manager has and then determine the data and measures to be collected to address them.

Do: Collect data and, based on the issues identified and using baseline data, derive graphical representations (i.e., indicators) to better illustrate the data trends.

Check: Analyze the trends, graphs, data, etc. to better understand the issues and the performance towards their resolutions.

Act: Report the results, recommend improvements, and identify new issues and questions.

1.2 Organization of This Report

In general, the concepts and measures defined in this report are designed to benefit the program manager by providing insight into, and hence control over, a software contract's resources, progress, and technical issues. To effectively use the concepts and measures, the program manager will require personnel skilled in software engineering and development and in the acquisition process and environment.

The data definition and collection concepts provided in Chapter 2 of this report let program managers define their own goals, identify what they believe to be the issues and risks, and define what questions they want the measurement to help answer. The set of commonly used software measures provided in Chapter 3 helps acquisition program managers address issues that are common to most software contracts. An overview of measurement analysis techniques provided in Chapter 4 helps program managers better analyze the data they have collected.

1.3 Purpose and Audience

The primary objective in publishing this report is to accelerate the use of software measurement in the acquisition process by providing:

1) Measurement process concepts that support the implementation of software measurement in DoD programs.

2) A set of metrics that are useful in addressing common software development issues.

This report is written for DoD program managers who are responsible for managing and making decisions regarding a software contract. It is expected that the program manager is knowledgeable about and regularly uses DOD-STD-2167A [DOD2167A]. However, this report can be adapted and used by other government agencies or persons who have similar contract monitoring and oversight duties. In addition, contractors can use the process and measures described in this report to support their own management and development processes.

1.4 Scope

This report provides the basic concepts needed to start a software measurement process. The process can then be tailored and further elaborated to support a specific contract or program office. Follow-on efforts may include a detailed software measurement process for acquisition program offices including guidance on how to contractually implement software measurement.

The software measures described in this report are intended for use during the engineering and manufacturing phase of the system acquisition life cycle [DOD5000.1]; however, this does not preclude adapting them for use in earlier phases such as demonstration and validation or during later phases such as operational support. It also does not preclude adapting the measures for measuring how the process responds when applying process improvements or new technologies within a domain, e.g., reuse, object oriented design, etc.; nor does it preclude using the measures to guide and manage programs determined to already be in trouble.

1.5 Software Measurement Process Concepts

A software measurement process is a systematic method of measuring, assessing, and adjusting the software development process using objective data. Within such a systematic software measurement process, software data is collected based on a known or anticipated development issue, concern, or question. The data is analyzed with respect to the characteristics of the software development process and products, and used to assess progress, quality, and performance throughout the development. There are four key components to an effective measurement process:

- Defining clearly the software development issues and the software measures (data elements) that support insight to the issues.
- Processing the software data into graphs and tabular reports (indicators) that support issue analysis.
- Analyzing the indicators to provide insight into the issues.
- Using the results to implement improvements and identify new issues and questions.

Each of these components is driven by the issues and characteristics inherent to the program. For example, the decision of what specific data to collect is based on the list of issues to be addressed by measurement; the indicators developed from the measurement data are flexible and tailored by the development issues and related questions; and analysis techniques are chosen based on what information is desired and what question(s) need to be answered. Analysis can be directed at assessing the feasibility of development plans, identifying new issues, tracking issue improvement trends, projecting schedules based on performance to date, defining possible development tradeoffs, and evaluating the consistency and quality of development activities and products.

The measurement process and its components are implemented on an ongoing basis throughout the development. The defined data parameters are measured, processed, and analyzed within a flexible assessment structure. This allows for the measurement process to adjust to changing development activities and products throughout the life cycle. The key is that the measurement process is driven by the program's issues.

The government program office implements its own measurement process independent of the contractor. This includes collecting or receiving the actual data, not just the graphs or indicators. The government program office must use the data to perform its own analysis, independent of the contractor, then meet with the contractor to discuss and compare analysis results. This independence will foster the government's confidence in the data and a better understanding of the issues. This measurement process implies that the government program manager is actively involved in managing the contract.

For the software measurement process to be effective, it needs to be an integral part of the program manager's decision-making process. To accomplish this, insights gained from the measurements should be combined with program knowledge from other sources in the conduct of daily program management activities. It is the overall measurement process that adds value to the data for the program manager, not just the graphs or reports. The following steps are necessary to integrate the measurement into daily program management activities:

1. Collect measurements that are targeted to specific program issues and that will aid in understanding and improving the technical and management processes.

2. Use the measures together with other management information to improve insight into progress and risks. For example, when addressing schedule risk, look at related analyses of staffing and schedule progress.

3. Establish means to investigate both management and technical issues further. The software measurement process is often the starting point for obtaining insight into the entire program. To isolate specific causes of trends apparent from the measures, it is often necessary to ask additional questions. The value of the software measurement process in such cases is in identifying the questions that need to be asked.

4. Identify and implement changes to improve the program, including any needed improvements to the measurement process.

The measurement process needs to be consistent, yet flexible enough to address changes since they occur in all phases of software acquisitions. Change in the acquisition process is inevitable as the development proceeds and requirements become better defined: cost, performance, and schedule estimates are refined; personnel, machine, test, and support resource requirements are identified; and insights from the measures themselves are gained. As the issues change, the program manager should adapt the measurement process to address the changes. However, definitions of key data elements (e.g., lines of code) should be consistently used. The proposed software measurement process is different from a static approach by:

- Separating issues about the data (e.g., accuracy or timeliness) from the program issues by carefully defining and validating the data.
- Interpreting the data by using knowledge of the data, related program issues and risks, corrective action plans, and industry-wide software engineering experience (e.g., models or databases).
- Responding to changing program issues and needs during development by enhancing the measurements and data by iterative data collection and analyses.
- Producing end-item products from the measurement process (i.e., the indicators and accompanying analysis) that are objective and explainable.

To summarize, a software measurement process should:

- Provide insight into known software issues and identify new software issues.
- Use a consistent methodology that allows the software measurements to be objectively applied and evaluated throughout the program’s life cycle.
- Allow the measures to be modified as the program’s products and activities evolve.
- Enhance the program office’s technical and program management processes through the use of quantitative data.

1.6 Common Program Issues

The measures provided in Chapter 3 can be used during the entire software development process. They were chosen because they seem to be the most useful measures, of those commonly accepted and used, to help the program manager address issues common to all software programs. As with the development of other system components, the key issues for software development relate to resource adequacy and expenditure, development progress and schedule, and the quality of the interim and final products. The nature of software development results in these three areas being strongly related. Issues with software quality, for example, can in many cases be directly attributable to schedule and resource constraints. This requires that the software measurement process address the relationships between several types of software data.

Issues related to the adequacy and expenditure of resources include:

- Staff availability and stability
- Funding adequacy
- Spending rate
- Allocation of resources to key activities (e.g., documentation, configuration management, or testing)
- Productivity rate assumptions
- Size estimate accuracy
- Subcontractor allocations
- Computer resource adequacy

The progress issue is simple; will the contract be completed on schedule, and if not, when will it be completed? Questions about the progress that program managers typically have include:

- Schedule (milestone commitments); is the contractor meeting commitments on time and within a prescribed level of quality?
- Development progress; what is the true (objective) progress of the contract?
- Schedule and forecasts to completion; are they realistic?
- Size estimates and forecasts; are they realistic?
- Functionality allocations; are they shifting from earlier to later builds?
- Productivity rate; is the planned productivity rate being achieved?
- Rework; are high levels of rework having an impact on progress?

The technical quality issues include both the quality of products being produced and the quality of the processes used to produce the products.` Here the program manager is concerned that interim products are stable and complete so that successive processes using those products do not compound mistakes (i.e., defects) and thereby drive up cost through added rework. Questions about the technical quality that program managers typically have revolve around issues regarding:

- Utilization of target computer resources, e.g., spare capacities and throughput.
- Problem reports, e.g., severity of problems, number of problems, timely resolution of problems, and high density levels of problems in products or processes.
- Completeness, e.g., product and process exit criteria.
- Product stability and volatility, e.g., requirements, design, and functionality allocations.
- Process stability and volatility, e.g., procedures and standards conformance.
- Rework, e.g., later processes finding defects caused by earlier processes.
- Defect rate, e.g., excessive defect rates could effect future progress.

The basic measurements discussed in Chapter 3 can be tailored by defining and collecting data that help address the issues identified as key to the program being managed. The measurements further described in Chapter 3 that have been found useful in addressing the common program issues are:

- **Software size:** Alerts the program manager to changes in the estimate(s) and/or actual software size. As the size grows beyond what was expected, the probability of the schedule and therefore the budget being exceeded also increases.

- **Effort:** Tracks the staff hours being expended by the contractor during the various development activities and for the entire project. Software development is a human intensive and dependent activity, making the effort expended the largest and least controllable cost variable [BROWN]. The effort measurements are particularly useful because it alerts the program manager to changes to and deviations from the plan that could drive the cost up and cause the budget to be exceeded.

- **Staff:** Gives the program manager insight into whether or not the contractor has a staff that is stable and able to do the job. The contractor's staff is tracked according to predefined and agreed upon labor categories. Also tracked are the losses (staff turnover) and additions of staff by labor category. Losses that result in a higher than expected staff turnover for the project may jeopardize the project's quality and expected productivity rates because replacement staff must be trained to become familiar with the software being developed and with the decisions that have been made.

- **Milestone performance:** Tracks the contractor's performance toward meeting commitments to complete activities and their formal milestones and interim events. If interim schedule commitments are met, the whole project is more likely to stay on schedule.

- **Development progress:** Tracks the progress of the development by quantifying and tracking the work completed. Work items should have predefined entry and exit criteria to determine when they are started and completed. The program manager counts those items that have been completed, knows how many need to be completed, and, therefore, has a quantitative method to determine how much work remains. This is done for each phase of the project by quantifying the work to be completed that produces interim products.

- **Software defects:** Tracks the evolving quality of the products as measured by the number of closed and open defects. Defects can be tracked by product (e.g., errors discovered during testing or peer reviews) or by process activity (e.g., action items from reviews or comments from a document review). A defect can be recorded and tracked against anything that would cause the contractor to rework an item that has passed through its exit criteria. Using this measure, the program manager can help determine the level of resources needed, the progress made, and the technical quality of the software and the processes used to develop it, as well as have an early indication of the requirements to support the software after it is delivered.

- **Computer resource utilization:** Gives the program manager early insight into whether or not the upper limits for computer resources will be exceeded. For example, the software may have performance requirements and may also need to allow for future expansion all within a known hardware configuration.

The intent is to provide measurements that give insight into issues. An argument can easily be made that each measure above provides insight into each of the three common issues (resources, progress, and technical quality). Figure 1-1 shows the issue to measurement relationships discussed in detail in Chapter 3. From Figure 1-1, a program manager can determine what measures are needed to give insight into his or her issue. The exception is the software defects measurement that will indicate levels of required rework, thus affecting the resources and progress issues, and can also indicate the technical quality of products and processes.

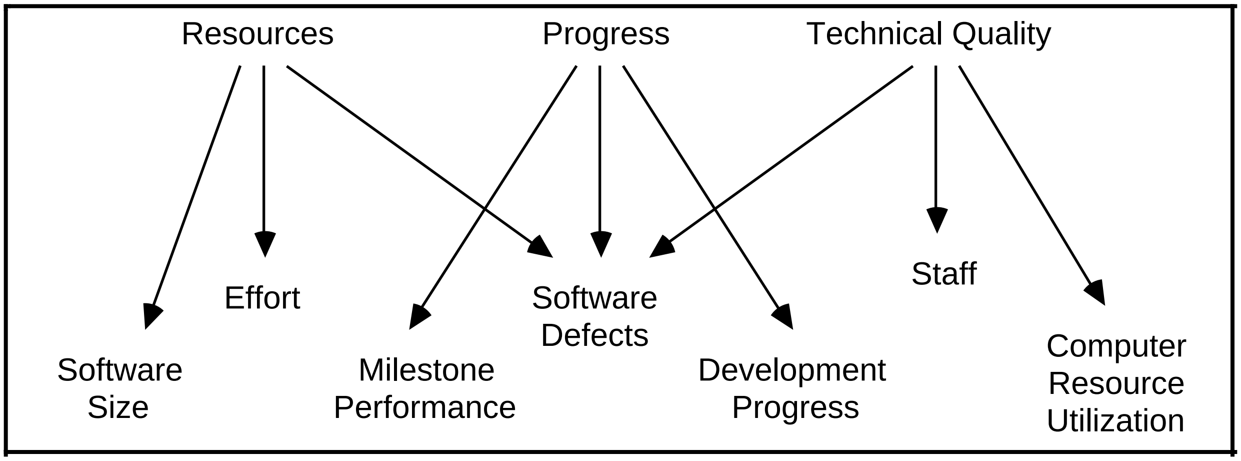


Figure 1-1 Cross Reference of Metrics to the Issues They Support

1.7 Constraints and Limitations

Software measures are valuable tools for gaining management and technical insight into a software program; however, they are not a panacea. To implement a software measurement system effectively, program managers must be aware of the following constraints and limitations associated with the application of software measurement:

- **Measurements are used as indicators, not as absolutes.** There is a strong temptation to seek absolute answers from measures. The role of measurement is to provide insights into the software development, which are based on objective data that might not otherwise be gained or be as timely. Often the measures prompt additional questions and insights that are not directly apparent from the measures themselves. For example, the manager may want to determine why the staff level is below what was planned. Perhaps there is some underlying program issue or perhaps the plan was inappropriate.

  This leads to a corollary constraint: Measurement cannot be applied in a vacuum. Insights gained through analysis of measures must be combined with program-specific knowledge to reach the correct conclusion.

- **Evaluations based on measurement are only as good as the input data.** Quality of the input data can be measured in three ways: timeliness, consistency, and accuracy. If the data isn’t timely, it is of little use for making decisions associated with the current program status. Even If the data is timely, the data elements must be consistently defined and accurately collected. A deficiency in either of these areas can skew the measurements derived from the data and thus lead to false conclusions.

This leads to the following corollary constraint: Measures are representative of the software development process that produces them; i.e., the more mature the software development process, the more advanced the measurement process. A well-managed program with a well-defined data collection process that is an integral part of the developer's overall process will provide better data than a less mature software development and collection process.

- **Measurement must be understood to be used and be of value.** Measurement, like any other management or technical tool, must be understood by the program managers. This means understanding what the low-level measurement data represent and what the intentions are of the overall measurement process. The program manager must also learn when to look beyond the data and measurement process to understand what is really going on. For example, if there is a sharp decrease in problems being discovered at the same time there is a large increase in problem resolution and close-out, the initial conclusion might be that the number of latent problems is decreasing. However, in an environment constrained by labor resources, it is equally likely that the problem discovery rate dropped because engineering resources were moved from software problem detection (e.g., testing) to software problem correction.

- **Measurement should not be used against the contractor (or other organizations).** The measurement process requires a team effort. While it is necessary to impose contractual controls to implement software measurement on a contract, it is important to avoid making measurement an adversarial issue with the contractor. The contractor will be sensitive to the program manager's use of the measures to quantify its performance and will resist supporting the measurement process if the results are used against it. Instead, the measures should serve as the basis for interactive resolution of development process concerns. Measurement should be used as a problem identification and resolution tool targeted toward making the processes and products better. While measures may deal with personnel and organization data, use of this data should focus on constructive process-oriented decision making rather than blaming specific organizations or individuals.

- **Measurement cannot identify, explain, and predict everything.** An effective measurement process can identify and help explain many software program anomalies and can identify trends that support forecasting of performance. However, measurement cannot identify every potential problem nor explain every situation. It would not be practical or cost-effective to attempt a measurement process that tried to quantitatively characterize every aspect of a software program.

- **The measurement process cannot be exclusively done by the contractor.** When initially using measurement, program managers may be tempted simply to impose requirements on the contractor to collect data, generate and analyze measures, and deliver completed measurement graphs and reports. In the ideal scenario, a contractor already has a measurement process in place. So why should the program office get involved? For three reasons: 1) the measurement process is iterative; not all desired measures and display formats can be predetermined since issues and problems vary throughout the program's life cycle; 2) the natural tendency of the contractor will be to present the program in the best possible light; independent government analysis of the data is required to avoid possible misrepresentation of the program; and 3) the measurement process needs to be issue driven, and the contractor and government program office will have inherently different issues.

- **Causal use of direct comparisons of programs should be avoided.** Because no two programs are alike, it is inappropriate to simply use data from past programs as the estimates for current programs. Program managers can, however, characterize their programs by using past, similar programs to determine the realism of projected plans and costs. In this way, historical databases and other types of program comparisons can be used to help develop realistic estimates.

- **A single measurement should not be used.** No single measure can give the insight needed to answer or address all program issues. Most issues will require multiple (and seemingly unrelated) data items to characterize the issue. This, and the fact that measures are interrelated, implies that a program manager needs to correlate trends across measures.

2. Data Definition and Collection

The basis of an effective software measurement process is to get quantified, low-level data so that 1) software development issues can be identified and clarified early enough to adjust plans and mitigate problems, and 2) progress can be tracked against plans. The concepts below will help guide program managers in meeting these objectives.

2.1 Deciding Which Measures Are Required

The question of what data to collect and how to define that data is driven by the program manager's current and projected software issues and characteristics of the software development process and products. Chapter 3 gives a common set of software measures to consider. Those measures and suggested data inputs were chosen based on issues and problems that are common to all software developments.[1] To tailor or expand the set, the program office needs to identify issues not addressed by the measurement set, combine the issues into a common list, set the priorities for the issues, and determine which of the issues can be addressed with the software measurement process. The program manager can then determine what data is needed to support insight into the issues. However, not all issues can be predetermined or projected; therefore, the program manager also needs to include provisions that allow the process to be flexible so that it can be modified to provide insight into unforecasted issues.

Once the issues are identified, the program manager and the contractor must agree on definitions of the entry and exit criteria for the process and products, all data inputs, the standards for acceptance, the schedule and progress estimation methods, the collection methods, etc. before awarding the contract. For example, the program manager and the contractor must agree on the definition of source lines of code (SLOC) and how and when the SLOC will be estimated or counted (or both when applicable). This entire process and all of its decisions and agreements should then be written into the contract.

Once the data collection has started, the definitions should not be changed. The most important data concept is that of consistency in the definitions. Changing a definition after the data collection has started produces variations in the trends that could confound the analyses and camouflage performance or related problems. However, sometimes definitions will change. When they do, it becomes critically important that the government program manager understands the change and how the change affects data that has already been collected.

When determining what data to collect, the preferred data is that which is a direct result of the contractor's process. For example, count the number of computer software units to be

[1] Many studies and reports have documented the problems and issues regarding software development, including [USAF], [SEI89], [DEMARCO], [BROOKS82], [CONGRESS], and [PACKARD].

developed by looking in the preliminary design documentation. Limit the use of data that is below the granularity supported by the contractor's process. For example, when collecting software defect data and accompanying process information about the defects, use the contractor's process. To do otherwise might not be cost effective and may adversely affect the productivity and schedule of the contract. As the development progresses, new techniques, tools, etc. might emerge. Tailor the data to these techniques, but as the data collected changes, also change your understanding of what is included in the data. The data collected should be that which is a natural result of the process used. Avoid collecting data that is derived (i.e., data resulting from an algorithm) or ill-defined and cannot be traced back to the contractor's process. Ill-defined data confuses the analysis and possibly leads to incorrect conclusions. If data must be derived (e.g., productivity data), it is better for the program manager to derive the data than it is for the contractor to deliver the data already derived. In this way, the program manager retains confidence in the data and better understands the issues by developing the analysis from the low-level data items that best represent the processes and products.

The program manager must realize that most issues will require multiple (and seemingly unrelated) data items to characterize the issues. For example, if the issue is the amount of rework and how it is affecting the contractor's progress, the program manager would need to have data on progress, the number of items being reworked, how much effort is being spent on rework, how effective the process is at finding items that need rework early, etc. If an issue is how the amount of rework is affecting the budget or end quality of the product, a different set of data items would be collected. The data needed to characterize this issue is also dependent on the development phase (e.g., requirements, design, or coding). Using the rework example, determining what items to use for measuring rework will be different during the requirements phase than during the design phase.

The way a data item is partitioned can also help to improve the insight into issues. For example, the data could be partitioned by:

- Processor (i.e., target platforms)
- Language
- Organization or subcontractor
- Computer software configuration item (CSCI), computer software component (CSC), or computer software unit (CSU)
- Configuration (e.g., avionics of two different airplanes)
- Severity
- Incremental build
- Facility/Lab
- Work breakdown structure (WBS) element

The level of partitioning will have an impact on how the data is used and how useful the data will be. For instance, if effort is collected at the organizational level rather than at lower levels of a WBS, problems (such as certain modules or units not receiving enough testing) may be disguised. The partitions should highlight the area(s) of concern so that trends from the indicators can help the program manager determine if the issues are being mitigated. For another example, SLOC can be partitioned by newly developed, modified, or reused. If only total SLOC is measured, changes to where the SLOC comes from may go undetected. These changes may reflect significant differences in the effort and schedule required to complete the project. In general, the lower the level of the data, the better the probability of isolating problem areas. However, the lower the level of the data, the higher the cost of the measurement process. Therefore, tradeoffs will need to be made.

The contractor's plans, assumptions, models, and historical data provide the basis for using actual project data. There are four types of data that program managers will use:

- Historical
- Plan
- Actual
- Projections

The contractor will use historical data and assumptions to generate estimates. The estimates, along with other knowledge of the system and environment, are then used as the basis for planning the project. Actual data is collected monthly by the government program manager and used as the basis for comparing the contractor's performance to its plans and for projecting future trends. The government program manager also uses historical data to determine the realism of the contractor's plans. After the plan is accepted, the government program manager uses actual data to ensure that the contractor is managing the project according to the plan.

2.2 Collecting the Measurement Data

Plan data is collected from contract documentation (e.g., the statement of work, the contractor's proposal, or software development plan). The program manager has actual data collected or reported monthly starting the month after the contract begins and continuing until the contract ends. However, not all data is collected monthly throughout the contract. For example, actual computer resource utilization data is not available during early development (although prototypes can be developed to simulate performance if the resource utilization is critical).

Whenever data is collected, its time of collection and source should be noted. The contractor's data should be treated as proprietary to prevent abuses and reduced availability and to alleviate the contractor's concerns about how the data will be used. The contractor will already be concerned about the use of the data and using data as a tool to quantify its performance. If measurement becomes an adversarial issue, the quality and availability of the data will probably be greatly reduced. Therefore, the data should be used as a tool for improving communications with the contractor and as a basis for jointly identifying and resolving issues.

Along with the source and time, assumptions and other knowledge about the data should be noted. Later, when analyzing the data, the program manager will need the data and all the information available about the data to completely understand the information. For example, if actual effort expended is lower than expected, the contractor might have had uncontrollable problems such as staff awaiting security clearances or waiting for equipment to be delivered. Other types of information could signal potential problems (i.e., changes to plans).

The program manager can use tools to help collect, manage, and report the data. Such tools can automate many labor intensive and tedious tasks. Without tools, a measurement process may not be feasible, both from an economical and technical standpoint. The two primary types of tools are collection tools and databases.

Collection tools will facilitate the timeliness, accuracy, and efficiency of the process. Not all data can be collected with an automated tool, but data such as lines of code, effort, staff, and computer resource utilization benefit the measurement process when collected with an automated tool.

Databases expedite reporting and provide the overall efficiency needed to make the measurement process viable. A database allows the data to be stored, and it can include many different attributes that provide the program manager multiple views when addressing issues. At times, it may be best to share a database with the contractor or have the contractor maintain certain databases (e.g., a defect database). Here the preferred method is to have the contractor maintain a defect database with on-line access available to the program manager. In this case, the program manager has data on defects readily available and occasionally audits the information to validate the data and the process of updating the database.

Not all data can be obtained with an automated method; therefore, some other options available to the program manager include:

- Collecting data from other deliverables (e.g., plans, specifications, and status reports).
- Collecting data from activity output sources (e.g., configuration management or software quality assurance files).
- Collecting data that is the output of processes (e.g., inspections, peer reviews, audits, or formal reviews).
- Having data reported via a contract data requirements list (CDRL) item.

However data is obtained, it is best to use one primary method to collect the data and a different method to validate the data and primary collection method periodically. To lessen the cost of data collection and analysis, measures from all sources of CDRL items could be coordinated so that they are collected only once. For example, data and analysis from configuration management activities, performance monitoring, and cost or performance reporting may provide much of the information required for progress assessment. Also, the development of common definitions and parameters could extend beyond the data collection area.

2.3 Understanding the Data

Once data is collected, the program manager will need to analyze it to determine if identified issues are being mitigated and new issues are being identified. Before making determinations from the data, the program manager needs to consider the data and thoroughly understand what it represents. To understand the data:

- Use multiple sources to validate the accuracy of data and to identify differences and causes in seemingly identical data items. For example, when counting software defects by severity, spot check actual problem reports to ensure that the definition of the various levels of severity are being followed and are being properly recorded.
- Investigate the process for preparing the lower level data and understand what the data represents and how it was measured. When analyzing software size, for example, how are the estimates being prepared? What assumptions are being used to generate the estimates? Does the total represent a mix of estimates and actual data?
- Separate data and its related issues from the program issues. There will be issues about the data itself (sometimes actually negating the use of certain data items). Program managers shouldn't get bogged down in data issues, but instead, should focus on the program issues and the data items that they have confidence in to provide the necessary insight. For example, suppose that 50 problem reports are being generated per week and only 4 are being corrected. The program manager should not be bogged down with what 4 were fixed or what kinds of problems are included in the 50. The program manager should be concerned that the backlog of problems is rapidly increasing and that the development process itself may be out of control.
- Do not assume that data from different sources (e.g., SQA or subcontractors) is based on the same definition, even if predefined data definitions have been required. For example, for SLOC data, even though a specific definition was required, different organizations may modify that definition. Sometimes these modifications may be made for an acceptable reason such as an organization having an in-house code counter. In this scenario, the program manager must understand the definition and any differences or variability it will cause when using the data.
- Realize that development processes and products are dynamic and subject to change. Differences of data originating at different points in time may simply reflect the differences in the processes and products. For example, the number of integration tests may be greater than originally planned if the number of CSUs increases or more testing is recommended at a milestone review because of unforeseen complexities. The amount of change must be analyzed over time. Some change is natural and healthy and shows that the contractor is responsive to evolving program needs; however, too much change could indicate that the process is not stable or the original requirements were not adequate.

For example, Figure 2-1 shows monthly estimates for software size. There is some variability in the first nine estimates. Estimate ten, however, drops significantly. This should prompt the program manager to ask questions about the data and understand exactly what is being counted. The notation on the figure shows that a new methodology was used for estimate ten. The program manager needs to understand the new methodology, how it impacts other assumptions and data inputs, and how it affects plans that used the size estimate to make other estimates (e.g., effort and schedule planning). Estimate thirteen is also noticeably different in that it includes a significant amount of reused code. Again, the program manager needs to understand the data and the program assumptions used to generate the data.[2]

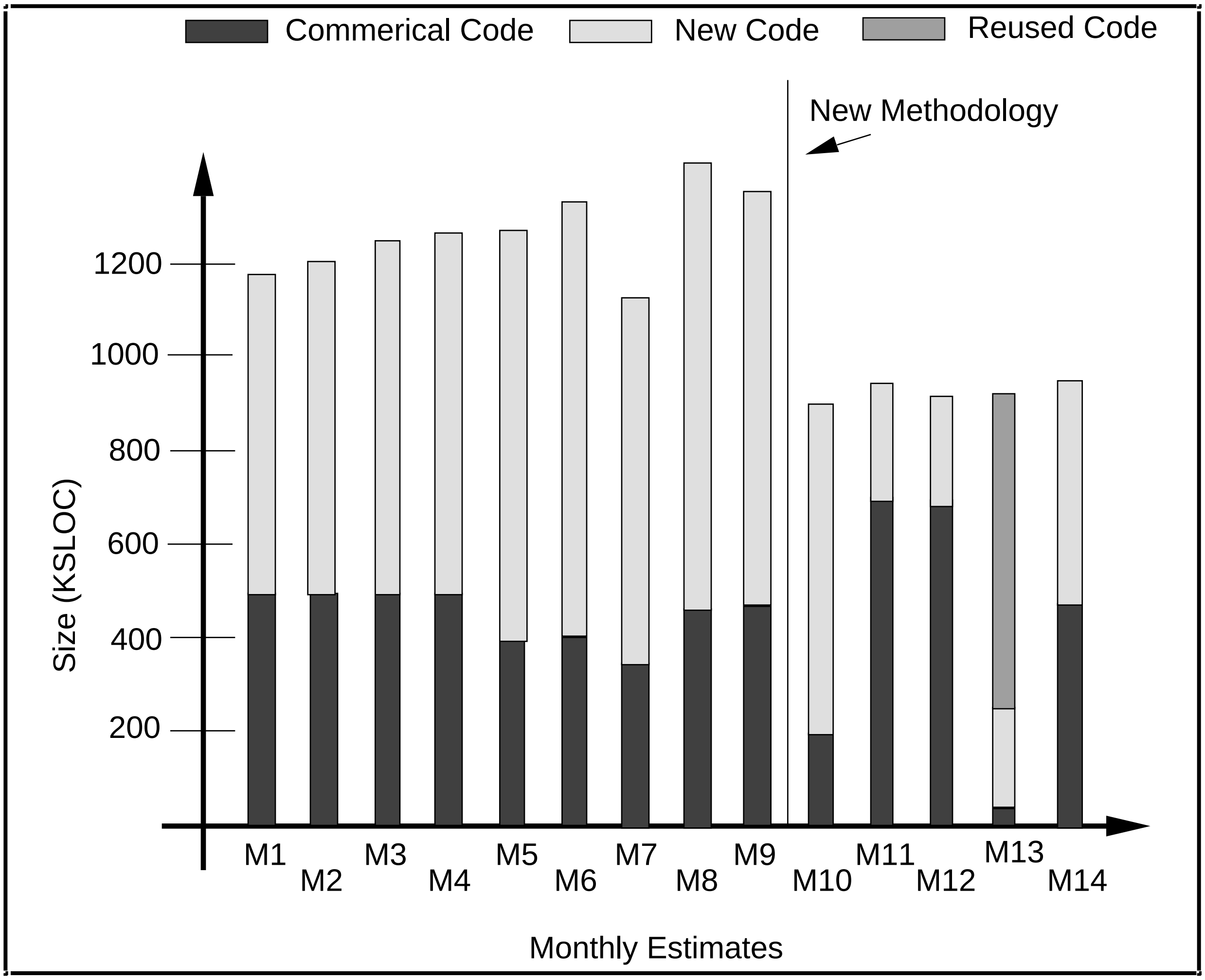


Figure 2-1 Example Illustrating the Need for Understanding Data

[2] Many comments were received that this example is unrealistic. However, this is data from a real program and is used to illustrate a point. It does reinforce the point that you MUST understand data.

2.4 Using the Data

Data is the basis for all analyses. Not only is it used to generate indicators and graphs, but assumptions, other knowledge, and understandings about it are used extensively during analysis to gain insight into issues.

Analysis of the data must involve all levels of staff in the development process in an integrated effort toward improving performance. Successful implementation of software measurement depends on the ongoing interaction between the contractor and the acquisition program office. All levels of staff in the development process must understand the analysis process, and the program manager must understand the analysis results in the context of what is happening on the program to improve his or her decision making abilities.

Measurement data need to be provided by the contractor, then processed and analyzed by both the contractor and the government. The contractor needs the data to manage its own process, to adjust its plans, and to provide meaningful status reports to the government. Experience also has shown that the government must have access to the basic data so it can independently analyze issues and interpret the results.

When processing the data, techniques that can help the program manager understand and analyze it include:

- Partitioning data to depict particular differences, e.g., by organization, processor, or CSCIs (computer software configuration items). For example, when using the effort measures, show the effort expended by subcontractor and CSCI.
- Separating inaccurate, inconsistent, or obsolete data from meaningful data. As the data is better understood, it will be realized that not all data items received are useful either because the items are outdated, low confidence levels in the validity of the data exist, or the data are inconsistent with other data items that are considered to be accurate and valid.
- Putting data sources (document, organization, date, etc.) on the graphs to support interpretations. Knowing the source and the date of the data will help the program manager to interpret it. However, if the contractor is explicitly or implicitly named, the program manager should handle the data as sensitive or proprietary.
- Putting major milestones on the graphs to support an overall view. To better understand how the data being displayed impacts or interplays with the overall process, major milestones (e.g., PDR and CDR) could be annotated on the time axis of graphs. This gives the program manger more insight into how trends in the data may impact the contract.
- Reviewing changes in plans or methods (e.g., build content and CSCI allocation) to determine applicable comparisons (i.e., data inputs to graphs). The program manager should preserve original baselines and assumptions so that adjusted plans accurately present progress history. When generating the graphs, the program manager should include the original baseline plan along with the currently approved plan and actual data. The program manager could also include other plans, both

approved and unapproved, for additional insights into the stability of the contract and plans. To do this, the graphs need to include the various plans as distinguishable curves as in Figure 2-2. This provides the program manager additional insight into the stability of the contract and plans.

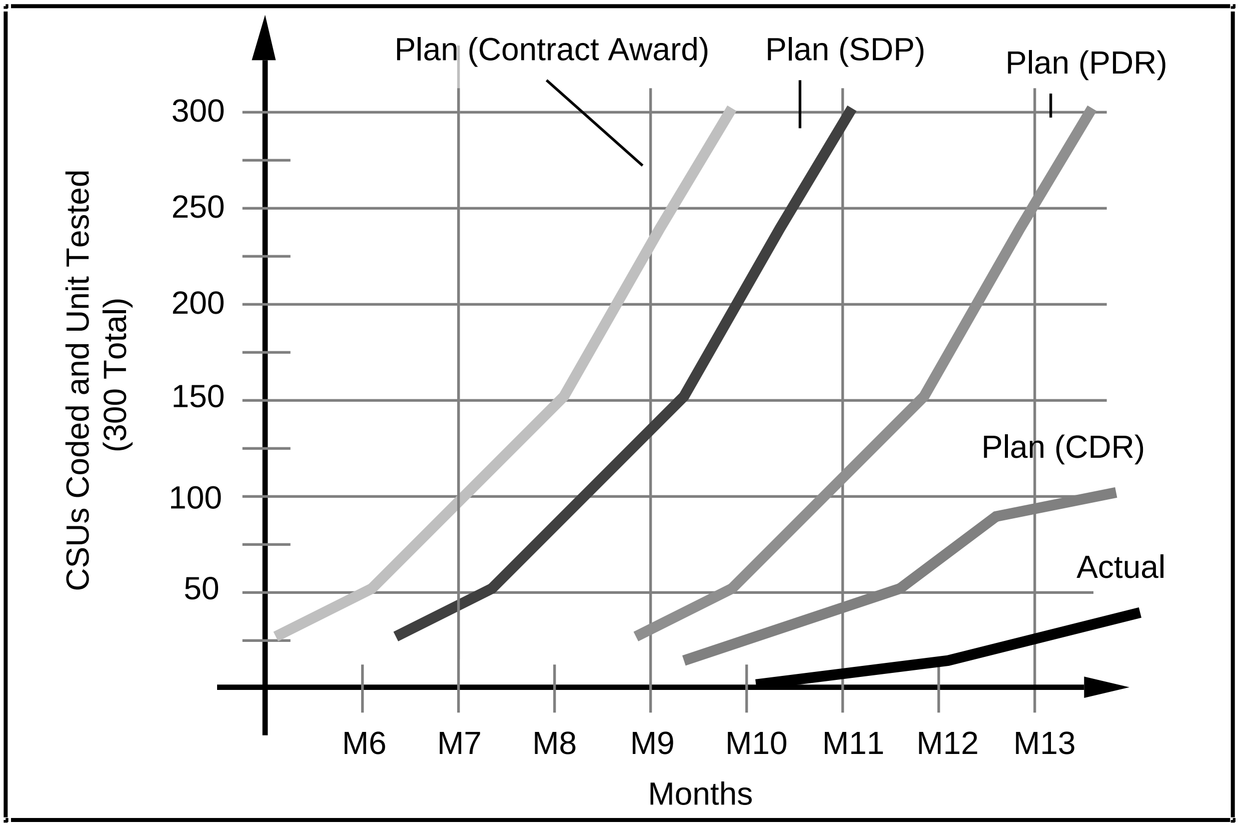


Figure 2-2 Example Showing How to Track Multiple Plans

In summary, the objective is to have an understandable set of measurement data that can be processed into different reports and graphs to address and help answer different issues and questions as they occur. To accomplish this, program managers should collect much of the data themselves and at a low enough level so that the data can be clearly understood and represented in many different ways.

3. Software Measures for Common Software Development Issues

This chapter highlights how a government program office might define, collect, and use data to address issues such as those outlined in Section 1.5.

3.1 Software Size

3.1.1 Purpose (Software Size)

Software size measurements give the program manager an indication of the size of the software being developed. Software size is used as an indication of the amount of work to be done and the amount of resources needed to do the work. The data collection records of size estimates and actual size, together with the assumptions from which the estimates were derived, can provide valuable historical data for improving the processes to estimate cost and schedule, thereby improving overall project management and planning.

3.1.2 Description (Software Size)

The program manager tracks the actual software size, compares the estimate(s) (both overall and for each incremental build), and then analyzes the trends for indications that the size of the software is growing or that functionality is moving from earlier to later increments. Estimates of various size attributes are compared with actual or new estimates monthly throughout the life cycle of the program. The basic size attributes are:

- Number of requirements: gives an early indication of the software size based on actual data.
- Number of CSUs: indicates the amount of work to be completed for each software increment.
- SLOC: gives an indication of the accuracy of the estimates by comparing actual values to estimates.

3.1.3 Data Inputs and Collection (Software Size)

The data inputs the program manager collects or has reported via a CDRL item for the software size measurements are:

- Number of distinct, functional requirements in the Software Requirements Specification (SRS) and Interface Requirements Specification (IRS): The number of requirements is partitioned by functional area for each CSCI. Starting with the system requirements, requirements data are collected from the System/Segment Specification (SSS) and then throughout the development process for each SRS and IRS.
- Number of CSUs as documented in the Software Development Plan (SDP) or the Software Design Document (SDD): The CSUs are tracked by CSCI/CSC for each build. The number of CSUs is estimated early in the process and is then tracked with actual data from the SDD(s).
- SLOC estimates for each CSCI and build and actual data from the source listing for each CSU: For each CSCI's incremental build, SLOC are partitioned by implementation language (e.g., Ada, C, or assembly) and by the amount of new, modified, and reused SLOC. The number of SLOC to be developed is estimated in the contractor's proposal. These estimates are then updated monthly or during reviews and are tracked and compared with the actual data as it becomes available.

3.1.4 Sample Indicators, Analyses, and Actions (Software Size)

Software size has a direct impact on the total development cost and schedule. A program manager can use the size measure to help answer the following questions:

- How much do the size estimates change over a period of time during the development process?
- How much do the actual data deviate from their estimated values?
- How does the trend of the estimates and the actual data affect the development process?
- Is the ratio of reused to new code changing and what are the implications to cost and schedule?

Major variations in the size data could indicate:

- Problems in the use, appropriateness, or validity of the model used to develop the estimates.
- Instability in requirements, design, or coding.
- Problems in understanding the system to be developed.
- An unrealistic original estimate for the system to be developed.
- Unachievable target productivity rates [BEAM].

Significant changes in estimates should trigger a risk assessment, preferably using size-based models to compare effort and schedule to planned values. For example, in Figure 3-1 the size of the software to be developed has nearly tripled from M1 to M6 (where M1 is the first month after contract award). Such a size increase could indicate that the contractor didn't understand the system to be developed, the requirements for the system to be built have changed significantly, or the original estimate was unrealistic. Such trends—and even ones of much lesser magnitude—indicate the estimated cost and schedule may not be met. Even if the estimated cost and schedules are met, the quality of the product could be lower than desired.

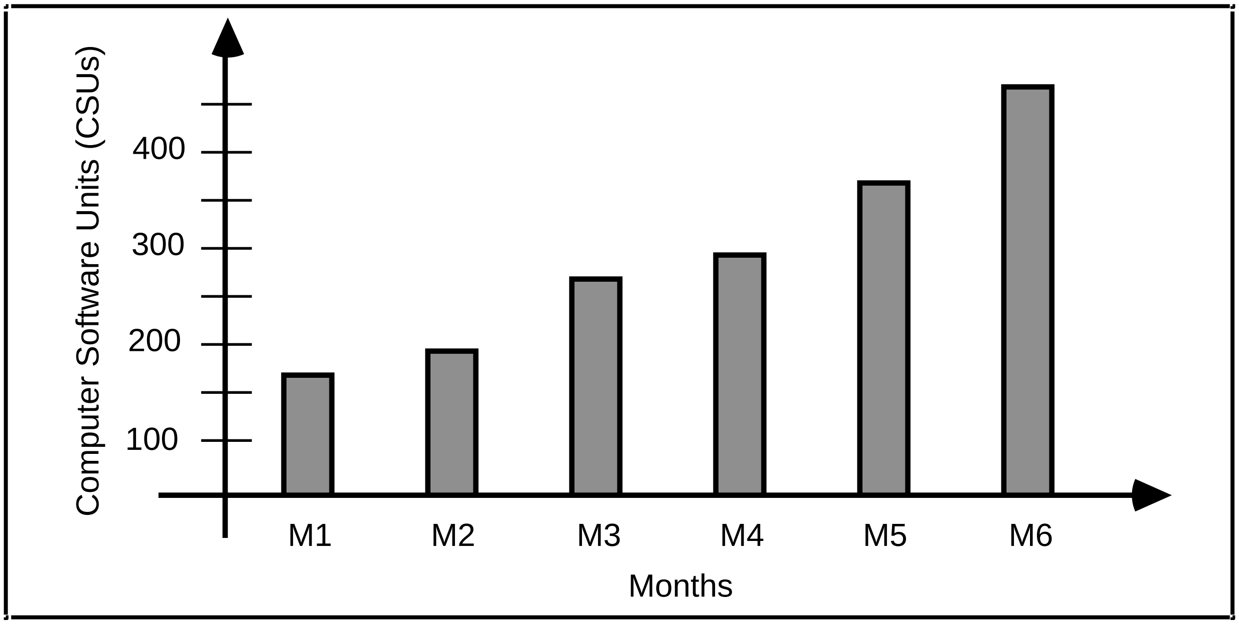


Figure 3-1 Sample Software Size Measure - CSU

The program manager uses the CSUs per build primarily to reveal postponement of functionality to meet schedule commitments. For example, in Figure 3-2 a fourth build shows up at M4 and the functionality of build number 1 is reduced. This and the magnitude of growth should signal a warning to the program manager that the contract's cost and schedule (and maybe the quality of the products) are at risk.

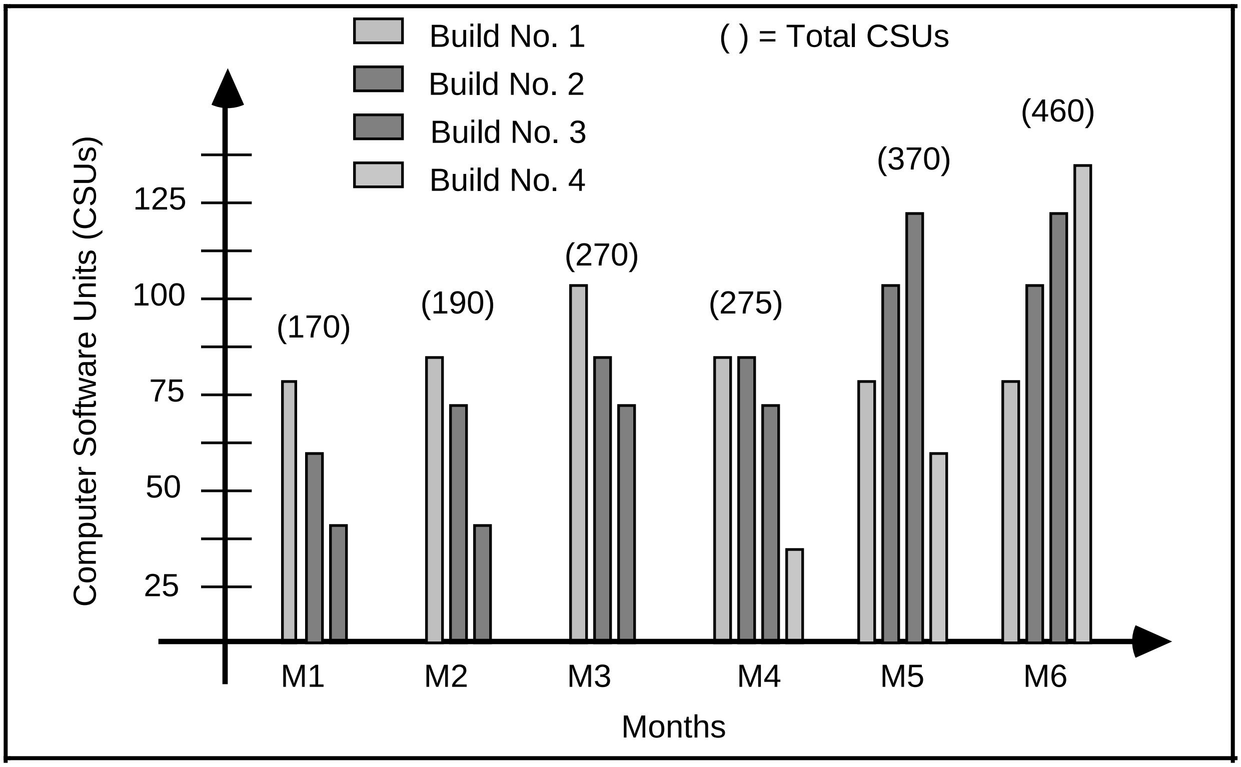


Figure 3-2 Sample Software Size Measure - CSU Per Build

Both of the figures indicate a contract with visible cost and schedule risks. In such cases, the likelihood of a major replan and possible contractual changes (e.g., engineering change proposals) increases. If funds and time are available and mistakes are not repeated, then contractual changes could be useful for replanning the project and mitigating some of the exposed risks.

To illustrate further the importance of tracking changes in plans, refer to Figure 3-3. Figure 3-3 shows the change of size estimates for each build from the second plan submitted to the seventh. Danger signs in this figure are the delaying of the development of code from earlier builds to a later build (i.e., build number 5) and the disappearance of the COTS (commercial off the shelf) code. Here, the program manager needs to investigate why there is such a postponement and what is being postponed. The program manager also must correlate the postponement with the other measures, most notably the development progress measures (Section 3.5). Such a postponement might have been predicted much earlier in the development by correlating the change in plans with other measures.

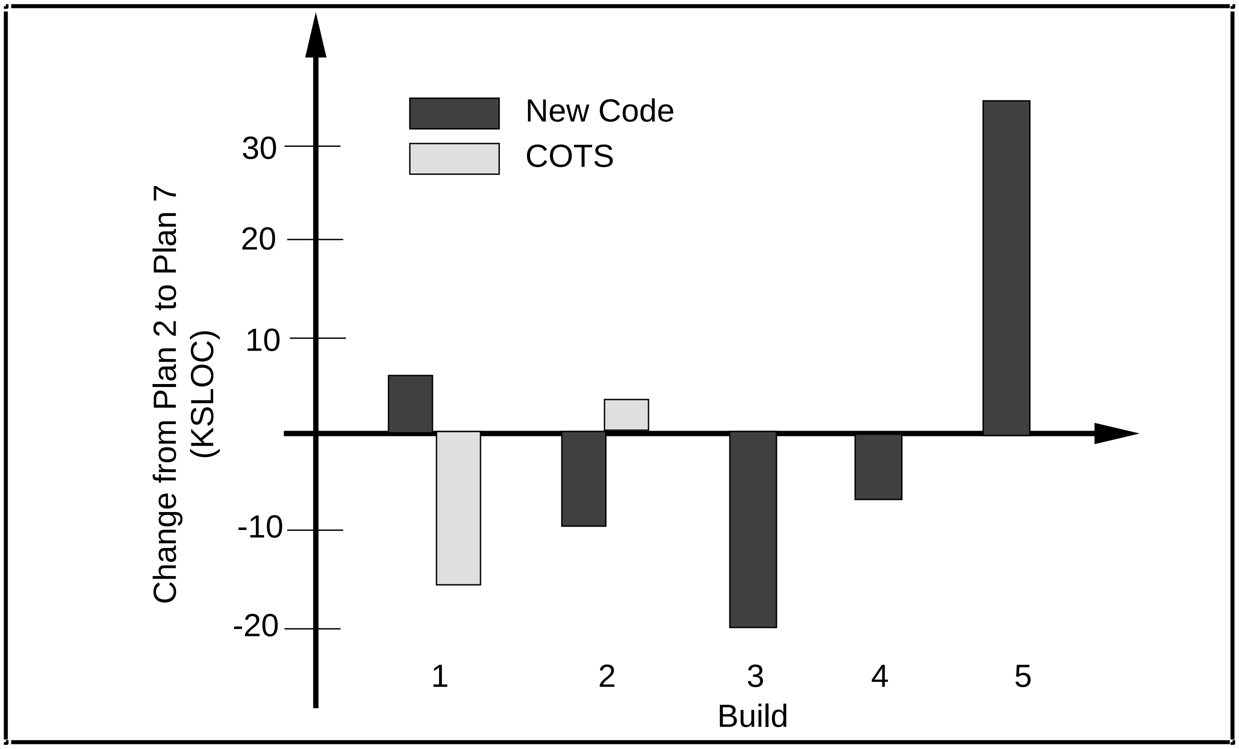


Figure 3-3 Example Showing the Postponement of Code Development From Plan 2 to Plan 7

Other uses of the size data include:

- Correlating it to other measures to determine its validity. For example, size data can be correlated with the effort measures (Section 3.2) or other size data (e.g., number of requirements).
- Correlating estimated SLOC with staffing plans. This allows the program manager to assess the realism of planned staffing levels using the size measures.
- Determining if actual productivity rates are deviating from those used to estimate the cost and schedule. If the rates used to estimate the cost and schedule are not realized, then the budget and schedule are at risk.

3.1.5 Other Measurements and Partitions (Software Size)

Based on program issues, the program manager may also want to consider the following software size measurements and partitions:

- SLOC for each processor
- Object code size
- Database size in terms of bytes, records, or fields
- Function points [IFPUG]
- Design language statements
- Design objects
- Pages of documentation
- Test cases
- Partitions by subcontractor

3.2 Effort

3.2.1 Purpose (Effort)

Effort measures show the relationship between planned and actual staff hours expended. They are used to monitor whether work products are being developed according to planned expenditures of resources. Effort measures allow the program manager to track the contractor's effort and to make inferences about the cost.

3.2.2 Description (Effort)

The program manager tracks the number of staff hours expended monthly starting at contract award and compares planned versus actual level of expenditures. Staff hours may be partitioned by direct labor and support staff categories, experience levels, discipline areas (SQA, testing, programming, etc.), or activity (requirements analysis, design, etc.).

3.2.3 Data Inputs and Collection (Effort)

The program manager has the actual staff hours expended reported each month. For each contractor labor category, the data inputs are reported via a CDRL item. (The labor categories are the same as those used for the staff measurements in Section 3.3.) From the contractor's proposal and development plans, the program manager extracts the planned staff hour expenditure rates for comparison with the actual data. The program manager should have actual staff hour expenditures reported monthly starting at contract award and continuing for the life of the contract. The data is collected for all non-government contributors. That is, it includes data for subcontractors, independent verification and validation (IV&V) contractors, separate test organizations, etc. To provide better insight into and control over the project, staff hours may be partitioned by:

- Development discipline area (SQA, configuration management, analysis, programming, etc.)
- Development activity (design, code and unit testing, etc.)
- WBS elements at the level that software activities are defined

3.2.4 Sample Indicators, Analyses, and Actions (Effort)

Each month the program manager collects the number of staff hours expended and aggregates the data to obtain the total staff hours expended for the contract.

The measures for monthly staff effort expended tracks staff hours expended against staff hours planned. Tracking effort monthly lends insights into the stability of the software development and the risks of not completing the project within cost and schedule. If the effort expended exceeds plans and work progress fails to meet schedule plans, the quality of the software may also be in jeopardy.

The measurements for the total staff effort expended tracks the total number of staff hours planned to be expended against the actual number expended up to a point in time. Information from the aggregated data is used to determine if the contractor is meeting the planned amount of effort. When used in conjunction with the development progress measures, the data can provide the planned and actual number of staff hours expended to complete each of the development activities. The primary reason for tracking the total effort is to be forewarned of cost overruns.

A variance between the planned number of staff hours and the actual number of staff hours expended will inevitably occur. How large must this variance be for the program manager to take action? The answer to this question depends on the issues that are important to the program. For example, consider a program that consists of building customized hardware where the software is needed in the early phases of the project to test the interaction between the hardware and software. In this scenario, changes to the schedule may not be tolerable. The program manager needs to determine how much of and how long an underexpenditure of staff hours can be tolerated to keep the project on schedule. To do this, the program manager must also use the milestone performance, software size, quality progress, and development progress measures to make inferences and management decisions.

Total effort expended data, when graphed, is usually in the form of a flattened S-curve, as depicted in Figure 3-4. The flattened S-curve reflects a smaller staff at the beginning of the project (a smaller sloped portion of the curve), a larger staff during the main part of the project (a more steeply sloped portion of the curve), and a reduction in staff at the end of the project (another smaller sloped portion of the curve).

The total effort expended data will most likely show a variance between staff hours planned and staff hours actually expended. However, because this is cumulative data, the difference between the plan and actual curves will not be large early in the project. For this reason, the program manager should take notice of small but steadily increasing differences between the curves.

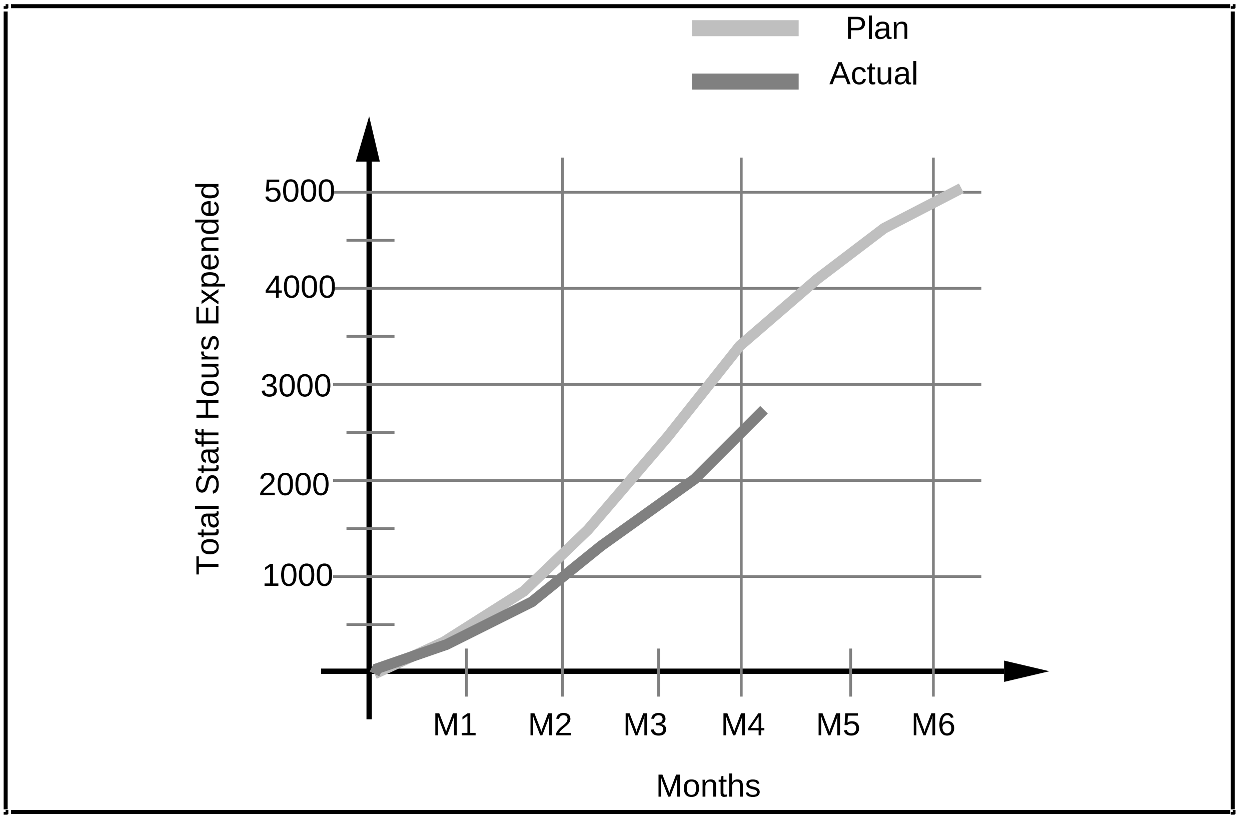


Figure 3-4 Sample Total Staff Effort Expended Measure

Figure 3-5 shows an example using the monthly effort data. The curve indicating the planned number of staff hours shows an orderly and achievable increase through requirements analysis and detailed design, a semi-constant level around a maximum during coding and unit testing, and an orderly decrease through integration and delivery. (Such a curve would exist for each iteration of activities.)

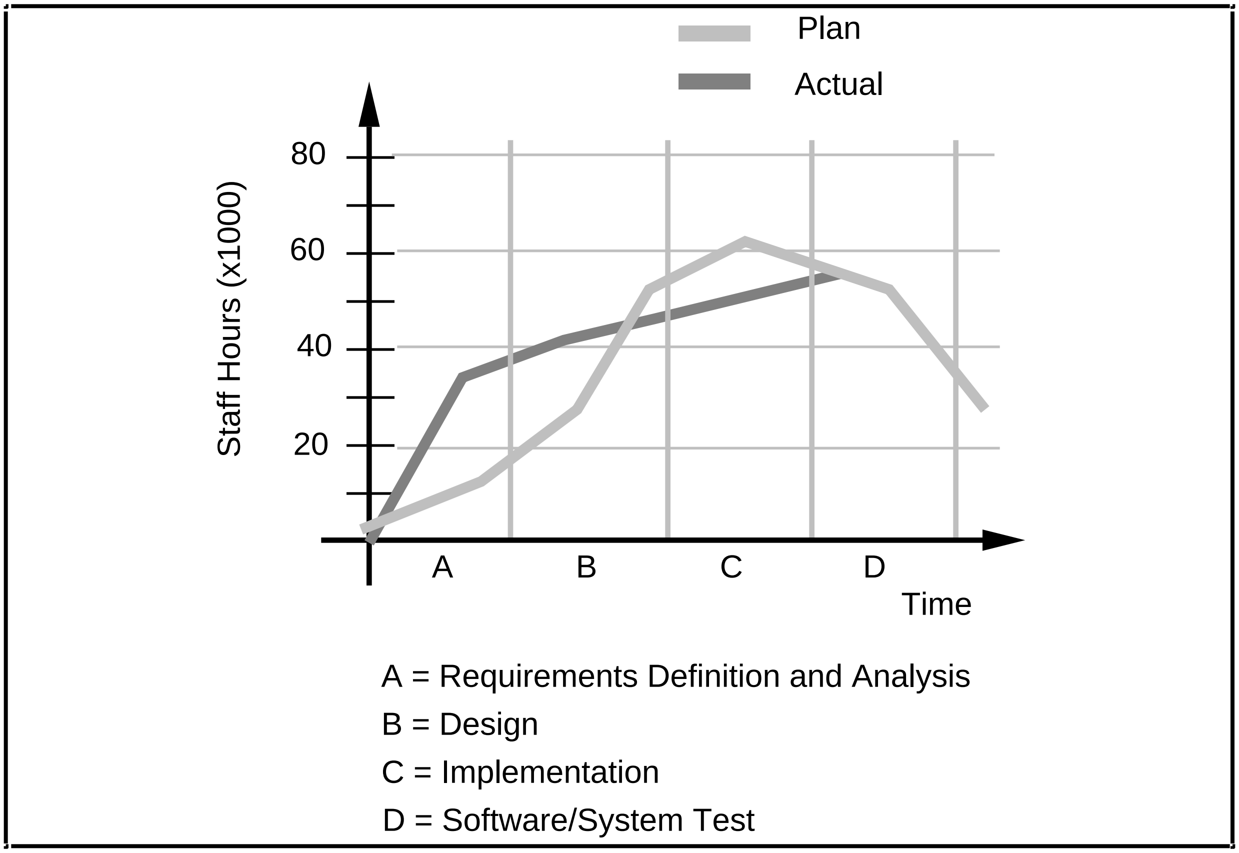


Figure 3-5 Sample Monthly Staff Effort Measure

Effort measures should be used with the measures for milestone performance, software size, software defects, staff levels, and development progress. If there is a significant underexpenditure of staff hours, the contractor may be having problems staffing the contract. Other possible reasons for underexpenditure of staff hours include:

- Overestimating the software size: The program manager should correlate effort measures with trends in the software size and staff measures. If the actual size data is tracking below the plan curve, it may indicate that the size was overestimated or that the contractor does not have an adequate number or the right experience mix of personnel on staff. When both trends occur simultaneously, the project is usually behind schedule.
- Insufficient development progress: By correlating the effort measures with the development progress measures, the program manager can determine if underexpenditure of staff hours is causing the development progress measures to track below the plan curve. The staff measures should also be investigated to determine whether the contractor has adequate staff to perform the task.
- Increasing levels of open problems: The program manager should correlate the effort measures with the software defects measures. If the difference between the number

of open and closed defects is increasing, additional staff may be needed or redirected to correct outstanding defects.

If there is significant overexpenditure of staff hours, the contractor may have been forced to absorb staff members from other projects that were ending. Other possible reasons for overexpenditure of staff hours include:

- Underestimating the size of the software: The program manager should correlate effort measures with the software size measures. If the size measures are tracking above the plan curve, it may indicate that the size of the software has grown well beyond the estimate, requiring the expenditure of more staff hours than originally planned [BOEHM87].
- Insufficient development progress: By correlating effort measures with the development and milestone performance measures, the program manager can determine if the contractor is trying to make up delays by adding staff to the contract. This may not be successful; according to one of Brooks' rules, "adding people to a late project just makes it later." [BROOKS82]
- Increasing number of defects: By correlating effort measures with the software defects measures, the program manager can determine if the contractor is adding staff to correct a growing number of problems. This may indicate a software quality problem.

3.2.5 Other Measurements and Partitions (Effort)

Depending on program issues, the program manager may also want to consider substituting staff weeks, staff months, or even staff years for staff hours on the y-axis. The conversion factor must be defined and documented (e.g., 156 staff hours equal one staff month) if a time period other than staff hours is used as the y-axis variable. If the unit of measure on the x and y-axes are of equivalent value (e.g., months and staff months) then the y-axis will correspond to the number of equivalent planned and actual full-time personnel on the project. (See Section 3.3.)

3.3 Staff

3.3.1 Purpose (Staff)

Staff measures give the program manager insight into the staffing labor categories used on the contract. The program manager uses this insight to track the contractor's ability to maintain a sufficient level of staffing to complete the contract [BOEHM87] and to track the amount of the contractor's staff turnover. When used in conjunction with the effort measures, the program manager also gains insight into the number of part-time people working on the contract and the amount of overtime being worked (regardless of whether or not it is compensated).

3.3.2 Description (Staff)

Typically, early development activities have a more experienced staff, i.e., a large percentage of staff at the higher labor categories. As development progresses, however, experienced staff get replaced by less experienced staff. Program managers track the actual staff levels of the contractor's labor categories and compare them to the levels that were planned to be used. This measurement is analyzed by looking at and determining the impact of the variances between the planned and actual staff levels. The program manager also tracks the number of people added and subtracted from the labor categories (i.e., staff turnover).

3.3.3 Data Inputs and Collection (Staff)

The program manager has the number of contractor staff by labor categories reported via a CDRL item. To provide better insight into and control over the project, the staff labor categories may be partitioned by:

- Development discipline area (SQA, configuration management, analysis, programming, etc.)
- Development activity (design, code and unit testing, etc.)
- WBS elements at the level that software activities are defined

The program manager can specify in the request for proposal (RFP) an experience profile for the labor categories for potential contractors to address. The labor category profiles and staffing levels proposed by the contractor are used by the program manager as the plan for comparison of actual data. The staffing levels should be planned for the length of the development process.

The program manager should also define how to count the staff turnover and require specific data on the turnover by labor categories and possibly the partitions above. The data is then collected for all non-government contributors. That is, it includes data for subcontractors, independent verification and validation contractors, separate test organizations, etc.

3.3.4 Sample Indicators, Analyses, and Actions (Staff)

The program manager generates graphs of the staff data similar to the one shown in Figure 3-6 to show average staffing levels and turnover on the contract. A chart like Figure 3-6 would be generated for each labor category and for the contract overall.

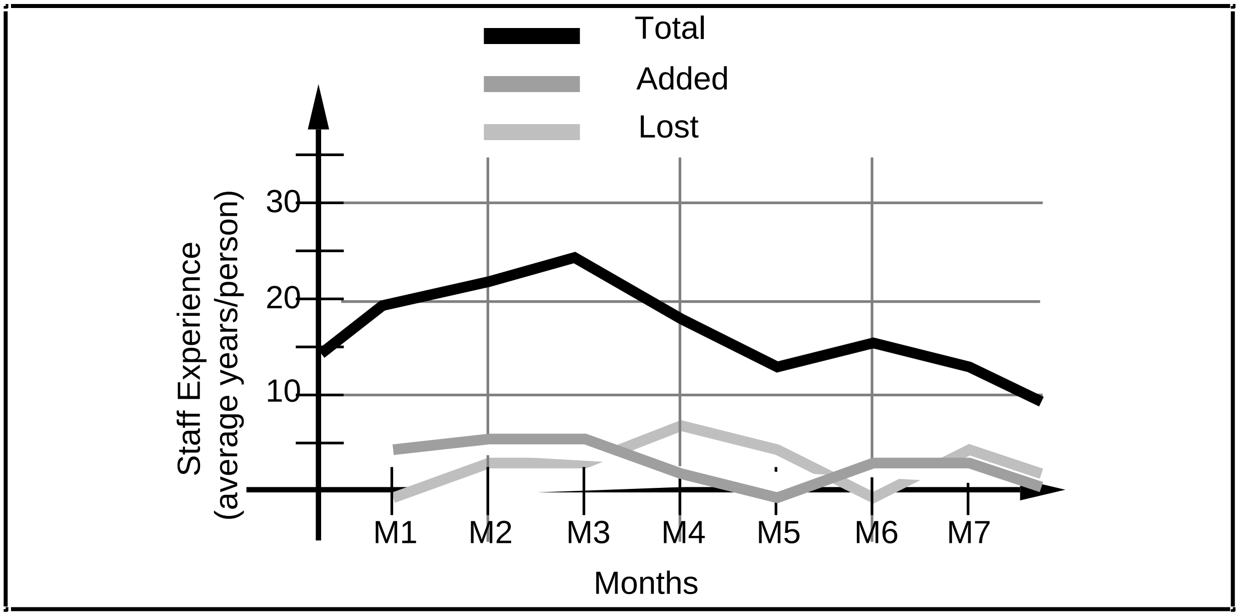


Figure 3-6 Sample Staff Measure

The total staffing level should show an overall decreasing level of experience (i.e., a decreasing percentage of staff at the higher labor categories) because typically more experienced personnel are assigned to the contract during the early development activities (e.g., requirements analysis) than during later activities. (During the testing activity, staff levels may rise but typically not as high as those experienced during requirements analysis.)

Insights into staff capabilities supported by these staff measures help the program manager to identify situations where a lack of staff may cause problems in the program's technical performance. Early insight into specific staff capabilities can also highlight staffing risk areas. Examples of analyses that can be performed with the staff experience measure are:

- Graphing the number of people charging by labor category (i.e., the number of staff per labor category) coupled with the number of hours expended by labor category (Section 3.2), shows the average hours expended per staff member. This information allows the program manager to assess how people are being used on the project. The use of too many part-time people may degrade the accountability and efficiency of a project. Likewise, the overuse of a limited number of people may degrade the quality of the products.

- Determining if actual staffing is consistent with the planned levels and if there is adequate commitment for using senior staff (i.e., staff from the higher labor categories).
- Correlating the staff measures with the measurements for development progress, milestone performance, and effort to determine if trends in these measures are related.

3.3.5 Other Measurements and Partitions (Staff)

Based on program issues, the program manager may also want to consider the following staff measurements and partitions of the staffing levels:

- Development activity (design, code and unit testing, etc.)
- CSCI
- Subcontractors

3.4 Milestone Performance

3.4.1 Purpose (Milestone Performance)

The milestone performance measures gives the program manager a comparison of actual milestone completions against established milestone commitments. These measurements quantify the contractor's performance toward meeting commitments for delivering products and completing milestones.

3.4.2 Description (Milestone Performance)

Milestone performance measures help the program manager graphically portray planned delivery dates, replanned delivery dates, and intermediate activities needed to meet the end delivery dates. The milestone performance measure tracks progress and delays for activities with respect to planned DOD-STD-2167A milestone events. These include system design review (SDR), software specification review (SSR), preliminary design review (PDR), critical design review (CDR), test readiness review (TRR), functional configuration audit (FCA), and physical configuration audit (PCA). It also tracks interim milestone events defined in the software development WBS leading up to the milestones. For each revision to the plans, associated schedule delays are indicated and tracked to determine the impact of the revisions and the delays associated with the revisions.

Other items that may affect milestone performance include tool development schedules and deliveries of government furnished equipment (GFE) and government furnished information

(GFI). If delivery of GFE or GFI is delayed, schedule and development progress may be affected. Therefore, the program manager may want the requested schedule data partitioned further to reveal the progress against these critical item dependencies.

3.4.3 Data Inputs and Collection (Milestone Performance)

The planned and actual data inputs the program manager collects or has reported via a CDRL item are:

- WBS activity start and completion dates
- DOD-STD-2167A milestone start and completion dates
- Key interim product milestone start and completion dates
- Progress to date (overall and for each milestone's interim activity)
- Estimated slip (overall and for each milestone's interim activity)

Objective entry and exit criteria for each event and activity must be defined and agreed on at contract award. It is also important to identify clearly the method of calculating progress to date, for estimated slip, and for the revision status of planned events.

WBS activity schedules and data on actual activity progress are collected monthly throughout the contract. The program manager notes the overall delay resulting from each plan revision when the revision is submitted and relates the delay to the original plan and all approved plans superseding the original.

3.4.4 Sample Indicators, Analyses, and Actions (Milestone Performance)

Experience has shown that late or unacceptable software products are good indicators of schedule risk. These late and unacceptable products are shown on milestone performance measurement graphs as slippage and can reveal contractor problems in maintaining the planned schedule. However, the measures only show the program manager that a deviation from the planned schedule exists; they do not explain why it exists. Milestone performance measures also depict overlapping project activities and the impact of rework and contingent risk abatement on schedule slippage through milestone completion estimates projected from actual data.

The inputs for the milestone performance measures may be displayed as a Gantt chart as in Figure 3-7. The chart shows planned events, actual events, and progress to date against planned activities. The chart shows a delay between planned completion and the actual activity or new estimate for completion. Large programs may have a set of tiered schedules for each system component. From such a figure, the program manager also looks for indications such as:

- Excessive overlap of activities where dependencies are expected (e.g., excessive amounts of coding activity completed before the design activity is completed). For example, in Figure 3-7, Activity 3 and Activity 4 are scheduled to start at approximately the same time. If Activity 4 depends on outputs from Activity 3, the program manager should ask what work will be done on Activity 4 before it receives the outputs from Activity 3.

- Successor activities starting before predecessor activities (e.g., coding starting before detailed design is started).
- Larger slips shown in predecessor activities than in successor activities. For example, in Figure 3-7, Activity 2 has slipped 40 days, yet Activity 3 is ahead of schedule 10 days. If Activity 3 depends on Activity 2, then the program manager needs to question how the 40 day slip will be made up.

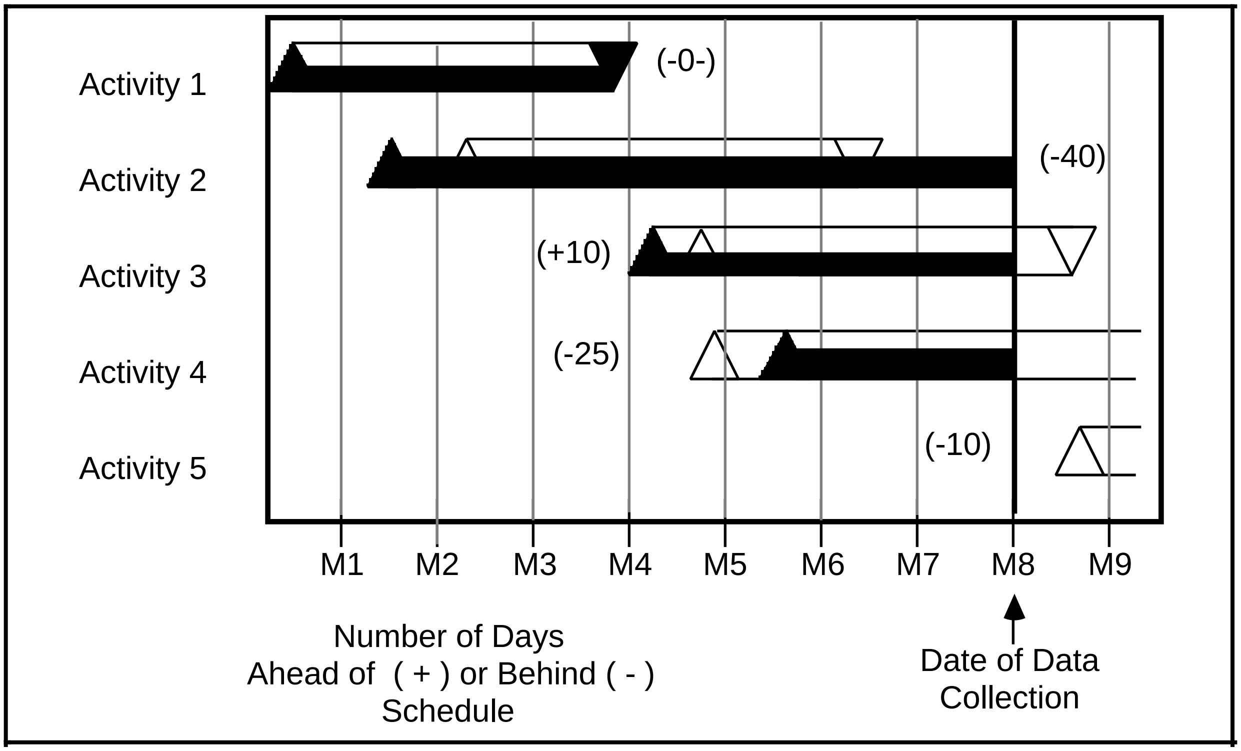


Figure 3-7 Sample Milestone Performance Measure

To determine the stability and progress of the schedule, the program manager could use a table similar to Figure 3-8 which includes for each revision, the date, time since last revision, and schedule change since the last revision. Records of milestone performance measures can be a valuable historical tool for improving schedule estimation, particularly for development efforts with characteristics similar to those of previous systems.

Schedule Revision #	Date	Time Between Revisions	Schedule Change From Last Revision
1	6/15/88	12 months	- 4 months
2	3/15/89	9 months	- 5 months
3	3/15/90	12 months	- 2 months

Contract award date: 6/15/87

Data collection date: 4/1/90

Figure 3-8 Example Table Illustrating Milestone Performance Stability

The program manager can also prepare a figure similar to Figure 3-9 to show schedule change data graphically. From Figure 3-9, the program manager can see overall slips in the schedule. When the program manager receives data such as that shown in the first quarter on Figure 3-9, he or she should be concerned and ask how the first milestone can be late while succeeding milestones are early. Such a scenario usually leads to data similar to that shown in the third quarter where all milestones are late.

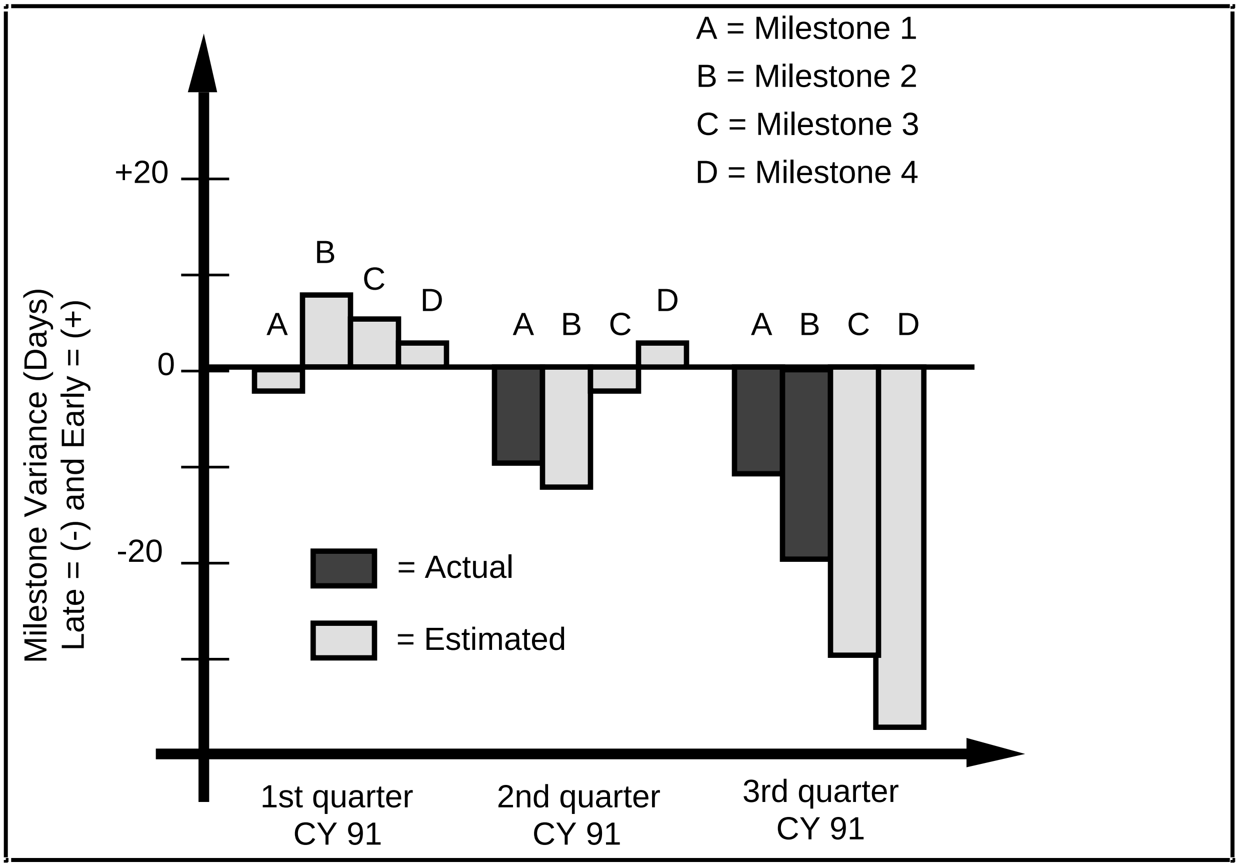


Figure 3-9 Example Measure Illustrating Milestone Performance Stability

Correlations with the defect measures should be made. Excessive emphasis on milestone performance could be counter productive causing the contractor to neglect quality in lieu of progress.

3.4.5 Other Measurements and Partitions (Milestone Performance)

The program manager could also use the following milestone performance measurements and partitions to reveal critical risk areas that might be masked by aggregation:

- Project status at different levels (e.g., CSC, CSCI, build, and total software)
- Project status for software engineering environment and tools
- Project status of GFE/GFI item deliveries
- Activities at lower levels of the WBS

Also, items on which activities depend and which may have a large impact on activity completion could be reported and tracked [CORI].

3.5 Development Progress

3.5.1 Purpose (Development Progress)

The development progress measures gives the program manager a quantitative indication of work progress. The measurement uses data on the planned and actual progress of software development activities to assess whether an activity is complete and the contractor is ready to proceed to successive activities.

3.5.2 Description (Development Progress)

The program manager applies the development progress measures across the major software development process activities, i.e., requirements analysis, preliminary and detailed design, code and unit test, integration and test, and formal test. The development progress measure is used to quantify the work to be done on key interim products of these activities, e.g., CSUs designed or coded. Each of these development activities is further broken down and quantified into work items that are small enough to allow progress to be seen monthly. The work items should be natural artifacts of the contractor's development process.

3.5.3 Data Inputs and Collection (Development Progress)

The program manager collects the following planned and actual data or has it reported monthly via a CDRL item:

- Requirements Analysis
  - Number of software requirements in the System/Segment Design Document (SSDD) that are documented in the SRS
  - Number of software requirements in the SSDD that are documented in the IRS
- Preliminary Design
  - Number of SRS and IRS requirements documented in the SDD
  - Number of SRS and IRS requirements documented in the IDD
- Detailed Design
  - Number of CSUs designed
- Code and Unit Test
  - Number of CSUs coded
  - Number of CSUs unit tested

- Integration and Test
  - Number of CSUs integrated
  - Number of CSC integration tests completed
- Formal Test
  - Number of requirements tested
  - Number of tests completed

The number of work items (e.g., requirements documented or CSUs designed) that are planned and reported must be sufficient to enable the program manager to determine intermediate progress; i.e., the granularity of the work items must be small enough to provide new data for each monthly report.

Tracking starts when an activity passes through its entry criteria. Tracking for each activity can either end when the exit criteria for the activity are completed or continue past the exit criteria as a measure of the rework for that activity [BOEHM87]. The plans should be identified by revision number. Data from past plans should be shown on the development progress measures.

3.5.4 Sample Indicators, Analyses, and Actions (Development Progress)

Early insight into deviations from planned progress is essential for early corrective action to be taken. The program manager's biggest advantage in using the development progress measures is that the true progress of work completed can be followed early in the development process. Moreover, these measures require the contractor to use a detailed level of planning to successfully use this measure.

A key to successfully applying this measure is having the entry and exit criteria properly defined for each of the data items reported. For example, the exit criteria for a CSU design to be counted as complete could be a successful design peer review and no open action items or defects against the CSU; for the code of a CSU to be counted as complete, an error-free compile and code inspection should be achieved; for the integration of a CSU to be counted complete, the CSU integration test should be successfully completed and no open defects should exist against the CSU. If an item's exit criteria have been met and it has been counted as complete, any additional work toward that item can be counted as rework.

Measures for projects progressing well will show a steady, consistent growth of completions as shown in Figure 3-10. Program managers can use a development progress measure similar to Figure 3-10 to determine 1) if any deviations of actual progress from the planned progress warrant corrective action; and 2) if the rate of progress is sufficient to meet the planned major milestone dates (i.e., activity completions) and, eventually, the planned date for product delivery.

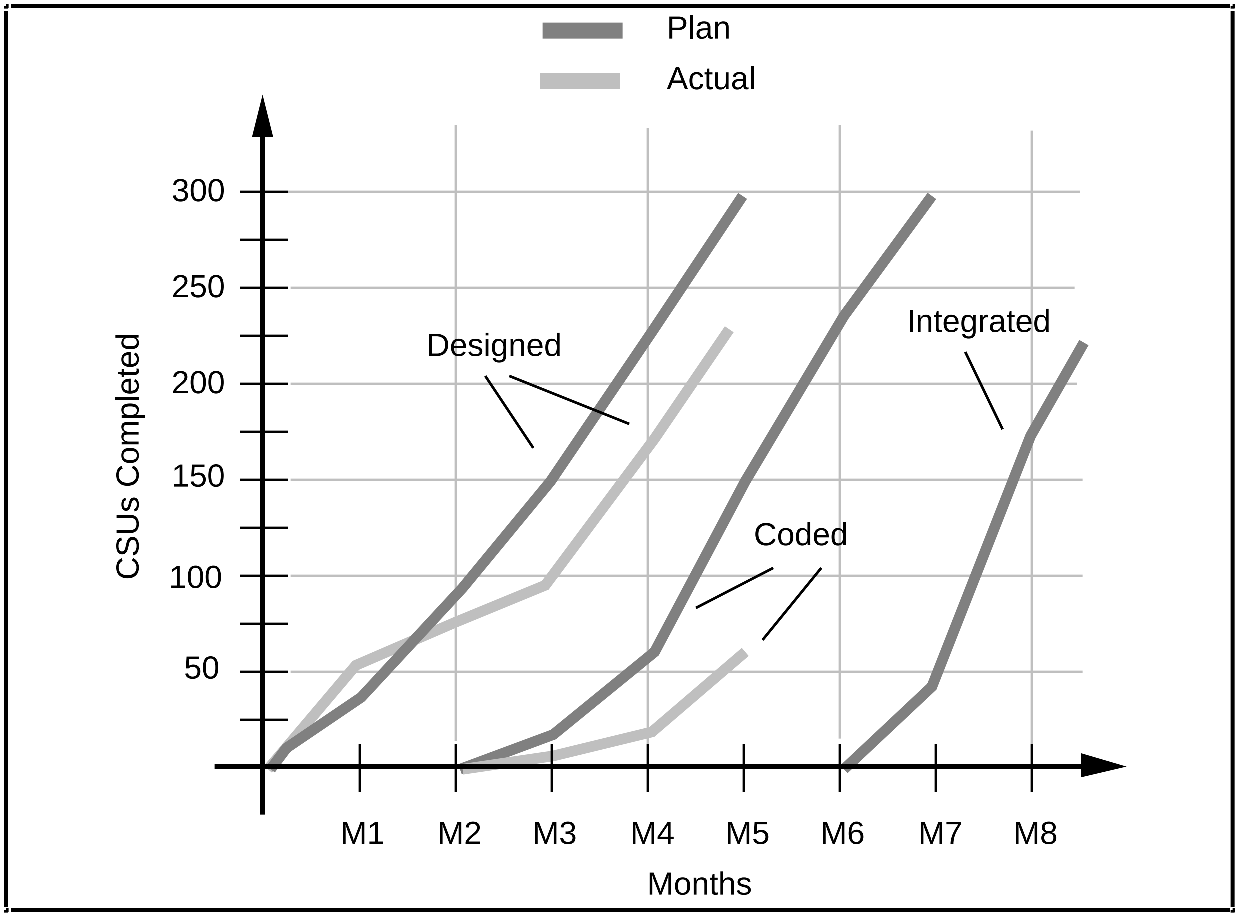


Figure 3-10 Sample Development Progress Measure

Development progress data can also be effectively displayed as shown in Figure 3-11. In Figure 3-11, the number of CSUs that have completed key process steps are tracked. For this measure to be effective, clearly defined exit criteria must be used to determine when a CSU has passed through such a checkpoint.

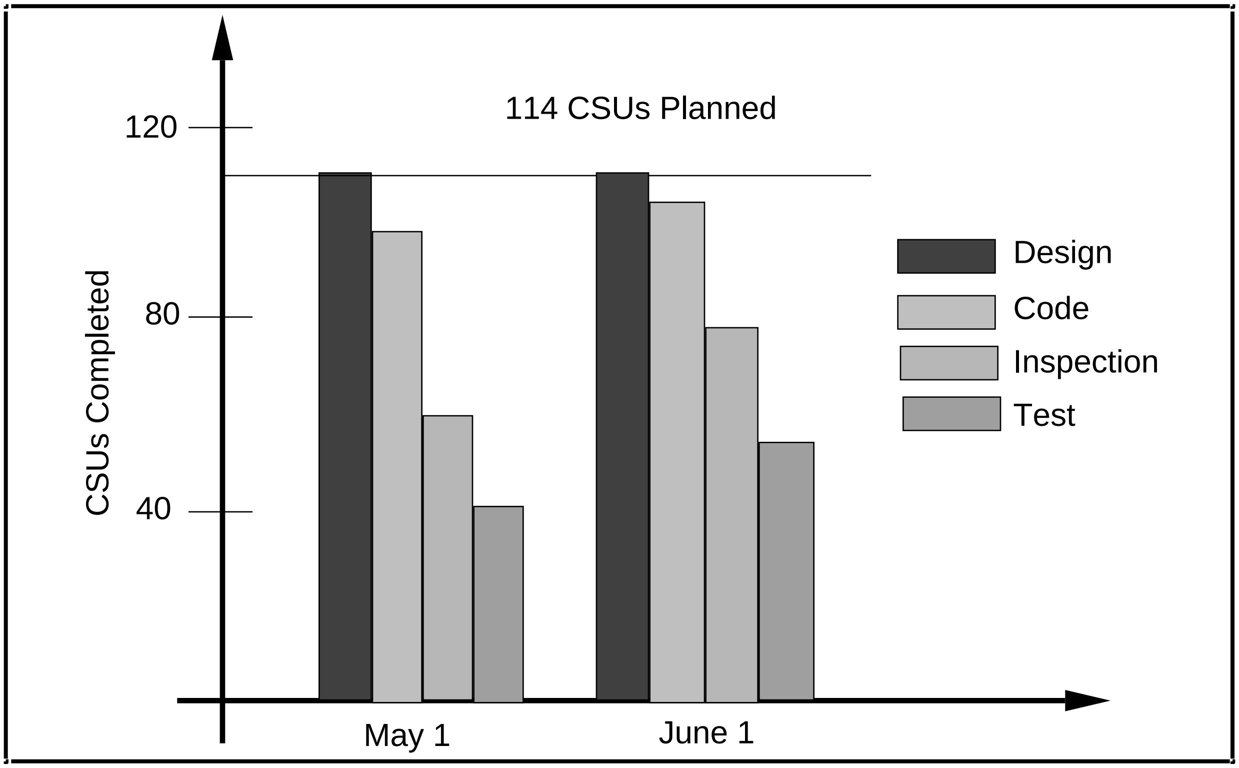


Figure 3-11 Example Showing CSUs That Have Completed Key Process Steps

Uses of the development progress measures include:

- Comparing it to the staff effort measures. Here the program manager compares the total number of work items and the distribution of the work items to the effort distribution to ensure that the distributions are proportional and represent the desired work item granularity.
- Comparing it to the milestone performance measures. Here, the program manager compares the progress shown for an activity in the development progress measures to the progress shown in the milestone performance measures.
- Assessing whether the planned schedule completion date is achievable given the deviation of the progress to date from the plan, i.e., the actual number of work items completed compared to the number that was planned to be completed to achieve the planned completion date. Two items that need to be analyzed to make such an assessment are 1) current deviation between the actual progress and the plan and 2) the rate of progress or slope of the curve for the actual completed work packages from the beginning of work to the last report.

To assess whether the planned schedule completion date is achievable, the program manager might want to:

- Ensure that the completion and exit criteria for milestones and work items are well-defined and verifiable (e.g., for milestones, the review is complete and action items are closed or the source code is submitted to configuration management for the code and unit test activity). If they are not, then true progress may be disguised by having work items reported as completed when work still remains.
- Take into account the quality of the completed work items, especially during the requirements and design stages where problems will be less costly to correct [BOEHM81]. If quality is questionable during the requirements and design activities, then the program manager should consider missing some early milestones to ensure a higher quality software product later. A rough judgment of the quality at these early stages can be obtained by correlating development progress with the quality progress measures.
- Make a gross schedule forecast from the development progress measure by taking the ratio of the planned completed and the actual completed work items multiplied by the total schedule duration. For example, suppose that 100 work items were planned to be completed by a certain date, but only 90 were completed and that the schedule duration for those 100 work items was 6 months. Those 90 items should have taken 5.4 months, thus the progress is at least 6/10ths of a month behind that which was expected. That is:

  if 100 work items = 6 months and 90 work items = x months then

  $100/6 = 90/x$ or $x = (6*90)/100 = 5.4$

  Note, at least three reporting periods are probably needed before such a forecast is useful. Also, such a forecast assumes that the work items being counted take approximately the same amount of staff hours. If this is not the case, do not do this type of forecast.
- Extrapolate the rate of progress (the slope of the actual completed work item curve between the current report and the last report) to determine if the trend is converging toward or diverging from the planned completed work item curve. If the actual curve is significantly lower than the planned curve and the slope indicates further divergence, this is usually cause for concern.
- Assess the deviation between the planned and actual progress profiles. If the contractor is unable to report planned progress profiles soon after SDR, there may be insufficient requirements analysis and decomposition. If the deviation between planned and actual data is increasing, SSDD requirements may not be allocated to CSCIs and these requirements may not be translated or refined to complete, testable, and traceable SRS requirements.
- Track the number of tests successfully completed against scheduled tests as time progresses. Deviations from planned test progress should decrease to zero toward the end of the contract. If the deviation persists, or if there are repeat test failures, more serious requirements, design, and implementation problems could be the

cause. The quality progress measures should be evaluated to obtain insight into these problems.

3.5.5 Other Measurements and Partitions (Development Progress)

The program manager may also want to consider the following development progress measurements and partitions:

- Track by separate CSCIs or system/segment.
- Show multiple activities on the same graph (e.g., coding and integration testing).
- Show system test progress for critical software.
- Track by incremental release content (i.e., by build).
- Show verification status of requirements and design.

3.6 Software Defects

3.6.1 Purpose (Software Defects)

The software defect measures track the acceptability and problem content of the products. It gives the program manager information on the readiness of the software to proceed to the next phase. It also helps the program manager determine if the product is ready for review or release and if it will be necessary to rework a product that has exhibited problems before proceeding to the next activity. The program manager should make the collection of software defects a constructive effort and encourage the early identification and correction of defects.

3.6.2 Description (Software Defects)

The program manager uses the software defect measures to track the defects resulting from program activity. The program manager uses the information on defects to determine whether to move intermediate products forward to the next development activity or to continue the current activity and further mature the products. For example, if at PDR an inordinate number of action items results from the requirements documentation, the program manager may decide that the contractor needs to spend more time analyzing requirements, updating the documentation, and closing the action items from the review. The program manager may want to advise or redirect the contractor to re-do the requirements specification before continuing on to the design activity.

Defects are anomalies noted during the development process. They are tracked via software problem reports (SPRs). SPRs should be tracked for any product or work item that has passed through its exit criteria and was counted as completed by the development progress measures. Defects can be:

- Errors detected during testing. For example, the work item would be the coding of a unit whose exit criteria could be a successful compilation, unit test, and code inspection.
- Government comments on documents. For example, the work item could be the document, or a section of the document if it is a major document, whose exit criterion is its delivery.
- Action items from reviews. For example, the work item would be the product(s) being reviewed whose exit criterion is the review. Action items can result from major milestone reviews (e.g., PDR or CDR) or from internal or less formal reviews (e.g., inspections or peer reviews).

For program managers to measure the effort being consumed by rework, they need to require contractors to track the effort expended correcting defects.

3.6.3 Data Inputs and Collection (Software Defects)

The data inputs the program manager collects or has reported for the software defects measures are:

- Total number of defects opened and closed
- Number of defects opened and closed since the last report

Additionally, the data above should include the following information for each defect to allow proper analysis:

- Type (e.g., testing error, action item, or document comment)
- Classification and priority[3]
- Product in which the defect was found

Different types of defects should be tracked separately so they are distinguishable from each other (e.g., document comments and software testing errors should not be added together).

[3] For determining the classification and priority of defects, refer to Appendix C of DOD-STD-2167A [DOD2167A].

3.6.4 Sample Indicators, Analyses, and Actions (Software Defects)

The software defect measures can be used to determine the effectiveness of the software development process. For each product and type of defect, the program manager tracks:

- Reported defects
- Open defects: defects that have not been resolved
- Closed defects: defects that have been resolved and approved

From these, the program manager can determine the defect discovery rate and the current known quality of the software products. The program manager can use a chart similar to Figure 3-12 to make these determinations.

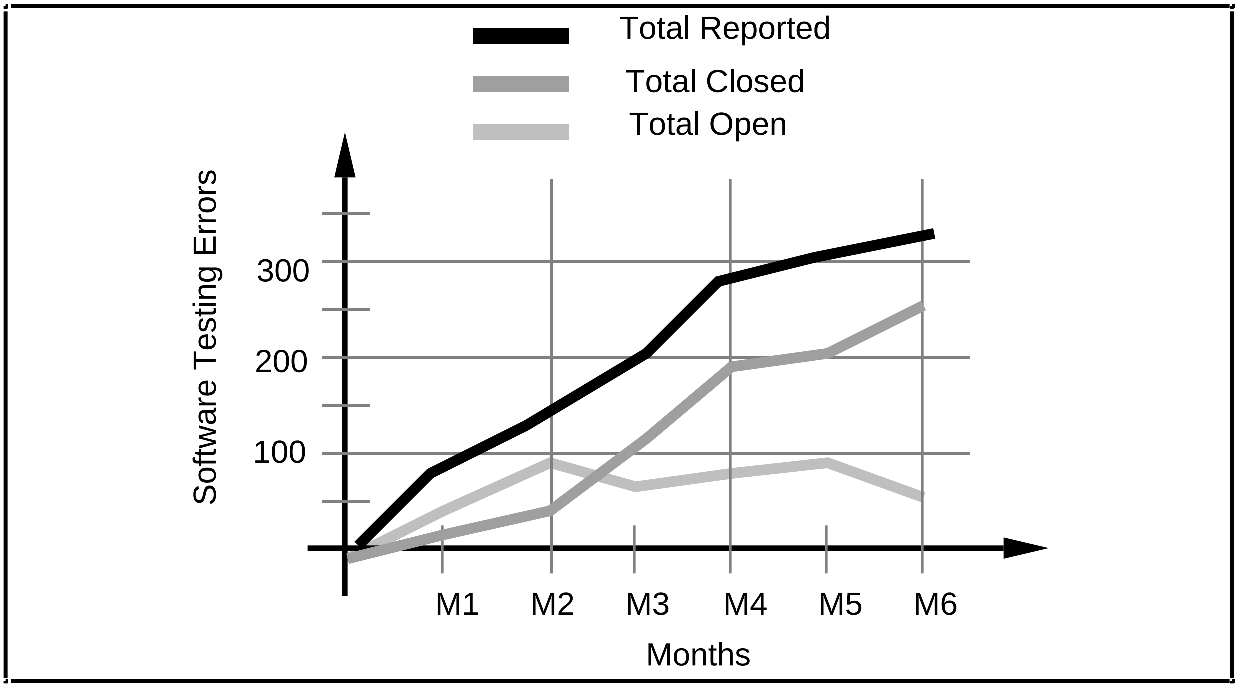


Figure 3-12 Sample Software Defects Measure

The number of defects reported can be used to determine the effort being consumed by rework by including the number of hours to process and correct each defect. The rate of closure and the trend of remaining unresolved defects can be used to measure the progress of the rework. The defects can be grouped and the progress of resolving the group(s) of defects can be tracked by the development progress measures.

The defect discovery rate can be tracked to determine the acceptability of products at a particular stage of the development cycle. When analyzing the discovery rate, the program manager needs to correlate the rate with the types of effort being applied to the contract. A declining number of reported defects may be caused by the reallocation of effort from discovering defects to correcting defects (e.g., the testing effort is being applied to correction activities and not to testing). A measure can be generated for other development resources (e.g., the number of development computers) and correlated to the defect discovery rate.

The program manager can also analyze the slope of each curve. If the slope of the open defect curve is positive, it could indicate that defects are being identified faster than they can be resolved; if the slope is negative, then the defect detection and correction process may be working or little effort is being expended to discover defects [BRETT], [MUSA].

Understanding both the data and what the data represents is clearly needed as shown in Figure 3-13. In this figure, the total number of defects discovered are being tracked. However, without knowing that the total includes the code related defects and non-code related defects, the program manager could be led to believe that many more defects are being discovered in the software than really exist. From such a figure, the program manager can also get information on the rate of discovery of defects. By correlating the figure to other measures such as the milestone performance and the effort measures, the program manager can get indications of impacts on the schedule.

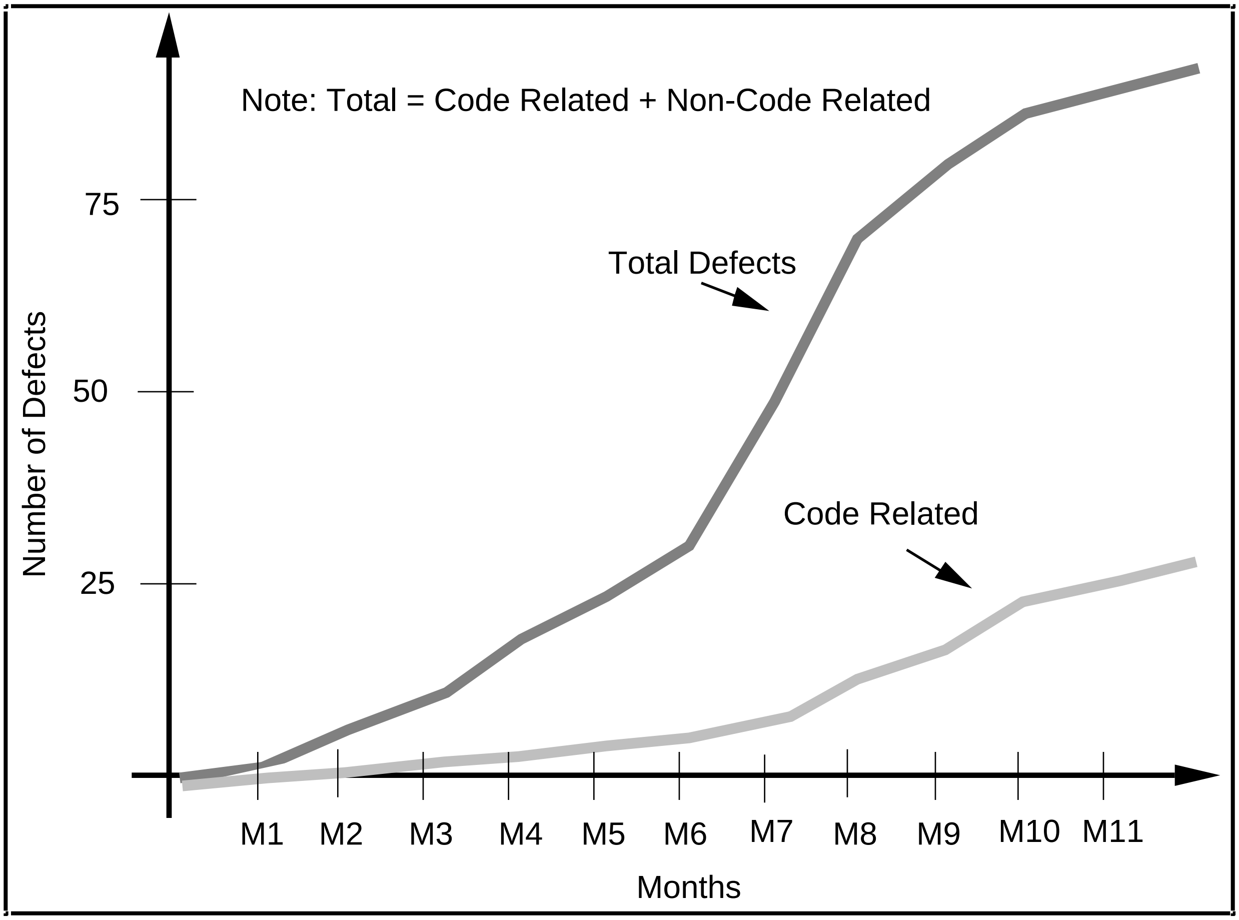


Figure 3-13 Example Illustrating the Need for Understanding Software Defects Discovered

Tracking the number of defects found during integration or acceptance testing can provide data for reliability models as well as information on the rate at which defects are being discovered during testing. By determining the software's defect density by normalizing the number of defects found during testing to the size measurement (Figure 3-14), the program manager also has an indication of testing adequacy and code quality.

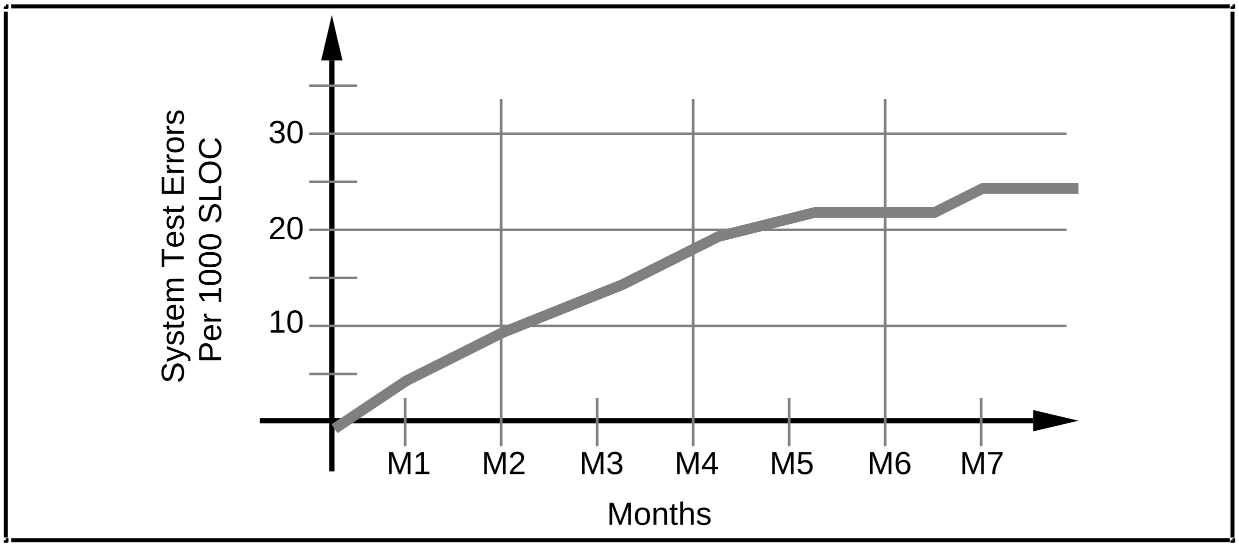


Figure 3-14 Sample Defect Density Measure

Increases in reported defects are frequently observed after major reviews (i.e., action items) and the start of testing activity (testing errors). If the increase is minor, it is necessary to investigate whether the product is of high quality or whether the review was ineffective.

The program manager needs to be concerned about the length of time that known defects remain open. The sooner defects can be detected, the lower the cost to correct them and the lower their impact on the schedule [BOEHM81]. The program manager needs to ensure that detected defects are corrected in a timely manner so that the risk impact is minimized. Figure 3-15 is an example indicator report that program managers can use to track the longevity of defect reports.

2167A Priority Levels	Number of Problem Reports That Have Been Open x Days				
	x ≤ 30	30 < x ≤ 60	60 < x ≤ 90	x > 90	Totals
Priority 1	2	1			3
Priority 2	3	1	1		5
Priority 3	3	2	1	1	7
Priority 4	4	3	3	2	12
Priority 5	8	6	3	3	20
Totals	20	13	8	6	47

Figure 3-15 Example Table Showing Longevity of Defects

Analyzing the root causes of defects can motivate an improvement process to prevent the introduction of errors [HUMPHREY89]. Analyzing the root causes of defects may also result in higher quality end-product and less product rework in later releases of a multi-release project.

The program manager can also use techniques such as Pareto analysis (as in Figure 3-16) to help isolate modules that are the most prone to error [ISHIKAWA]. If certain modules have more errors than others, those modules may be more complex, the functionality may not be completely understood [BURR], or the modules may be very large compared to those with few or no errors. The program manager could have the modules redeveloped (i.e., redesigned and recoded) or require more testing for the modules. For example, in Figure 3-16, CSCs D and F are responsible for 80 percent of the errors found. In this case, the program manager would have the contractor redesign those modules or apply more effort to them during testing.

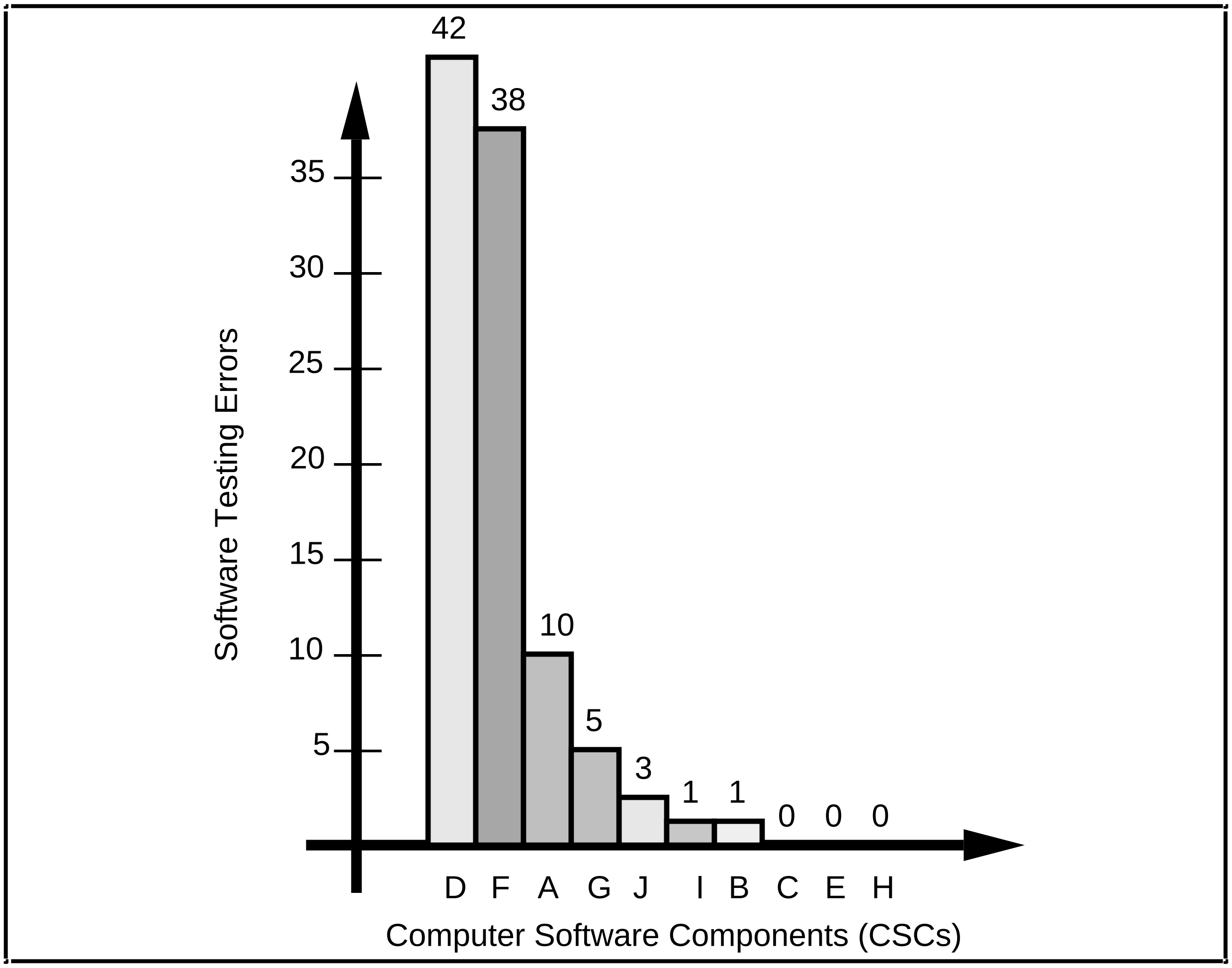


Figure 3-16 Example of Pareto Analysis Showing Defects Per CSC

3.6.5 Other Measurements and Partitions

Based on program issues, the program manager might also want to consider the following software defect measurements and partitions:

- Priority, severity, or criticality of defect
- Software language
- Development process or activity that caused the defect
- Development process or activity that found the defect
- Contractor or subcontractor
- Effort expended to close defects (or categories of defects, e.g., effort expended to close defects by various levels of defect severity)

3.7 Computer Resource Utilization

3.7.1 Purpose (Computer Resource Utilization)

Computer resource utilization (CRU) measures give the program manager an indication of the percentage of computer hardware resources used. The program manager is concerned about the use of computer resources because the software must operate within tangible hardware limits.[4] The program manager is also concerned that resources be available for future expansion of software functionality or for needed increases in software performance. The CRU measures track the use of a computer's processors, memory, mass storage devices, and input/output channel throughput. CRU growth should be reviewed and analyzed early in the program if spare capacities are of concern.

3.7.2 Description (Computer Resource Utilization)

The CRU measures track four categories of computer resources:

- Central processing unit (CPU) utilization: measures the percentage of available processing power used during worst-case software execution.
- Memory utilization: measures the percentage of total available computer memory process-resident software and data.
- I/O throughput: measures the speed and amount (number of bytes) of total throughput capacity used during worst-case data transfers.
- Mass storage utilization: measures the percentage of total storage used at peak residency.

The measures may be applied to development computer and target computer depending on criticality.

The program manager determines how much spare capacity is needed by considering the software's purpose and future. For example, if the software's purpose is an overnight processing of database reports for an inventory management system, spare capacities may be small because performance may not be an issue. In this case, expansion could be achieved by adding another computer. However, if the software is for the flight control system of a plane, the spare capacities may need to be large for increased performance or to allow for future expansion of the software because another computer is not easily added.

[4] In some systems, the program manager may choose not to use this indicator because it is not important to the system. However, in mission critical computer resource (MCCR) applications, the program manager should include it.

3.7.3 Data Inputs and Collection (Computer Resource Utilization)

For the CRU measures, the program manager has the following data inputs reported via a CDRL item:

- Total available capacity of each resource
- Current measurements
- Estimates projected to a designated milestone for each resource

Example of the units for each computer resource data input are:

- CPU throughput capacity: millions of instructions per second (MIPS)
- Memory: the kilobytes or megabytes for each type of memory (e.g., RAM or ROM)
- I/O throughput: bytes per second (BPS)
- Mass storage devices: kilobytes or megabytes for each device

The program manager determines the required spare capacities and has estimates made for the use of the resources until actual data are available. Estimates should be replaced with simulated results and actual data as they become available. Extremely important to the program manager are the method of calculating the data and the assumptions used to develop resource loadings for the "worst-case scenario." The program manager needs to ensure that the method used is valid and verifiable. Examples of methods include analysis by comparison, simulation analysis, or demonstration of the possible scenarios of resource use.

The earliest estimates of processor throughput availability should be carefully specified in terms of instruction mix and other benchmark assumptions until these estimates can be replaced by measurements.

Monthly tracking against plans should start at PDR and continue throughout the development effort.

3.7.4 Sample Indicators, Analysis, and Actions (Computer Resource Utilization)

Early insight into the resource demands of the software will highlight any inadequacy of the planned resources. Early CRU assessments are vital for long lead time procurements of special purpose processors, memories, buses, and mass storage devices. This is particularly important when there are physical or other constraints on the amount of resources that can be provided. Excessive use of computer resources contributes to increased schedule delay, increase of development cost, lowered reliability, potential loss of system functionality, and expensive software or system redesign.

The program manager tracks the CRU measures using a graph similar to Figure 3-17. Spare capacity levels for target CPU should be established to allow for software upgrades after the software has been released. The program manager monitors the use of resources to verify that spare capacities are sufficient. For systems where resources are critical, a spare capacity requirement lower than 50 percent must be carefully weighed in terms of cost, feasibility, and operational suitability. If a spare capacity (i.e., reserve) lower than 50 percent is established, there should be rigorous formal tracking and proactive risk management. Sometimes system engineering tradeoffs do not permit reasonable spare capacity targets. The program manager might then manage this risk formally by requiring quantitative allocations of resource capacity to CSCIs, CSCs, and management of these allocated resources.

When use of resources is near the specified upper bound for physically constrained computer resources, this measure should be correlated with the schedule and development progress measures to assess the potential impact of any necessary redesign.

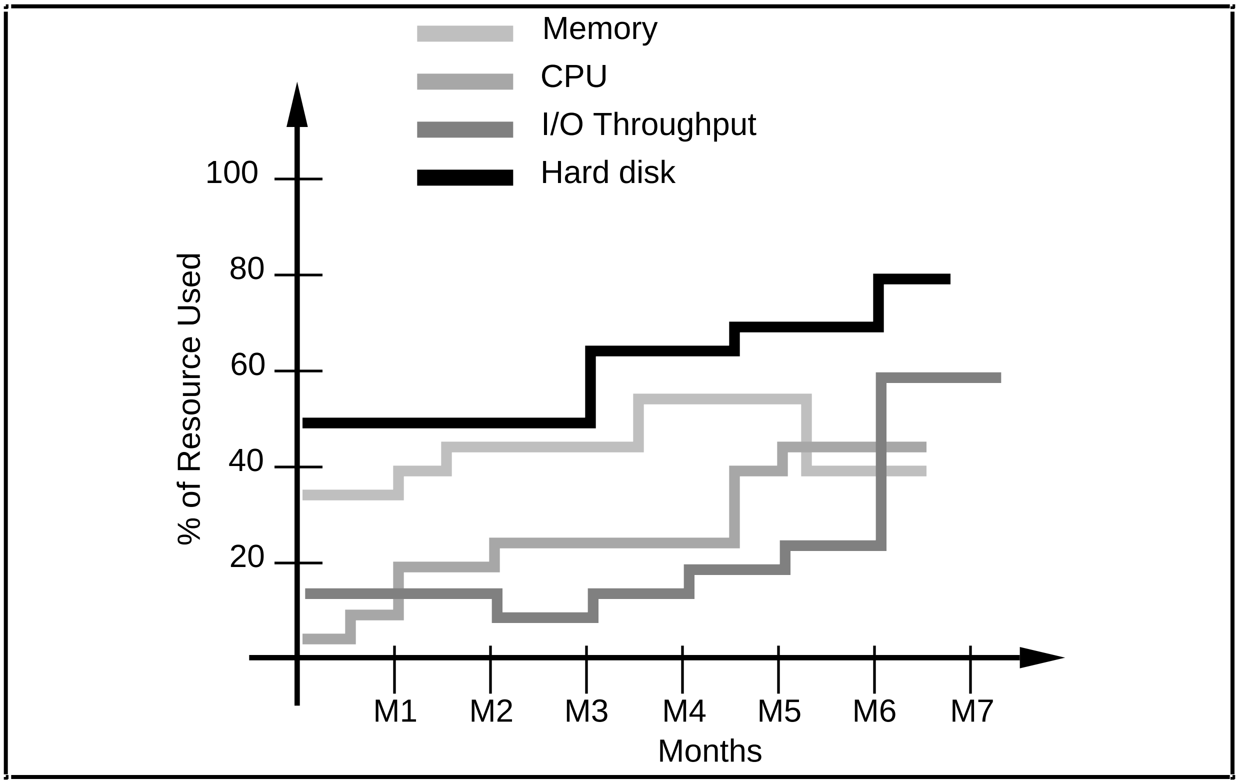


Figure 3-17 Sample Computer Resource Utilization Measure

The I/O resource is the hardest to measure fully because there may be many I/O measurements to consider. These include disk channel capacities and rates for serial data, parallel data, disk access, printers, plotters, local network data, I/O data bus, and special peripheral data. The I/O resource to be tracked must be defined if it can adversely affect system operation.

When system utilities cannot provide the CPU processor time used during execution of specific tasks, the program manager should consider the use of a "time burner" stub during design to simulate and estimate via a prototype the worst-case amount of time a subroutine is expected to take to execute after it has been fully coded. During verification testing, such a stub can be used as a test driver to set up a high priority task and increase appropriate time delays while looking for system degradation effects.

When the estimated use of resources exceeds management criteria for spare capacities, the following actions might be considered.

- Optimizing the software: the software and/or storage devices could be optimized using commercially available utility software.
- Redesigning the software: time critical functions could be redesigned and recoded in assembly language; time "hogging" functions could be redesigned for optimal processing; and time consuming data transfers could be redesigned using double buffering design techniques.
- Adding computer resources: if feasible, faster processors, more memory, larger disks, etc. could replace slower or smaller devices.
- Accepting loss of mission or support functionality: if necessary, functionality may need to be deferred to later releases or eliminated.
- Changing system requirements: as a last resort, the system requirements could be revised and alternatives considered. The system may not be possible within certain usage constraints.

Carefully defined and collected CRU data can provide a valuable historical basis for improving the accuracy of future estimates of computer resources, the effectiveness of methods for data collection and estimation, and the choices for management reaction criteria.

Other measurement correlations that the program manager can use include:

- Check data from the defect measures regarding defects related to resource utilizations.
- Investigate rework to determine if resource utilization problems are causing unplanned staff growth and excessive effort expenditures.
- Scrutinize size allocations to determine if size growth is, or will be, within hardware resource limits.
- Probe schedule and progress reports to assure that resource utilization plans are not impacting delivery dates and commitments.

3.7.5 Other Measurements and Partitions (Computer Resource Utilization)

The program manager might also want to consider the following CRU measurements and partitions:

- Absolute counts instead of percentages of CRU units to highlight changes in total reallocated available resources.
- No-load and average-load in addition to worst-case assumptions for resource loading scenarios to highlight the impact of different scenarios of resource loading.
- Separate reports on use of resources in a multi-resource architecture that has dedicated functions to highlight the impact of different dedicated functions on CRU.

4. Other Sample Analysis Techniques

A good software development process yields good quantitative data. Good quantitative data, in turn, provides information that contributes to successful software program management. Analyzing measurement data will provide the program manager insight into the software development process throughout the development life cycle.

Based on the program manager's issues, indicators are derived from low-level measurement data and analyzed to gain insights into the program. The program manager tracks the plans and estimates against the actual data as it becomes available. The program manager then extrapolates trends in the actual data to estimate future performance and progress and to determine if the trends mitigate known risks or expose new ones. Several analysis techniques can be used to maximize the insight achieved. The other techniques discussed are:

- Trend analysis. For example, plot and analyze the number of CSUs completing unit test.
- Multiple metric relationship analysis. For example, plot the current staffing plan through completion and compare it to the scheduled tasking to assess whether the planned staffing is realistic and adequate.
- Modeling input data analysis. For example, use a model such as the constructive cost model (COCOMO) or other commercially available tools to generate estimates and to extrapolate data to predict future performance.
- Thresholds and warning limits. For example, set thresholds around planning curves, then analyze the variability of the actual data using the thresholds as warning signals when the actual data approaches or crosses them.

4.1 Trend Analysis

Trend analysis is a basic technique for gaining insight into a program. Consider the example trend graph shown in Figure 4-1 which illustrates the number of problem reports and indicates their status. One purpose of this measure is to support the management and assessment of cost and schedule risk (i.e., if a large number of unresolved problems is allowed to accrue, cost and schedule overruns may result by the time the problem reports are finally addressed).

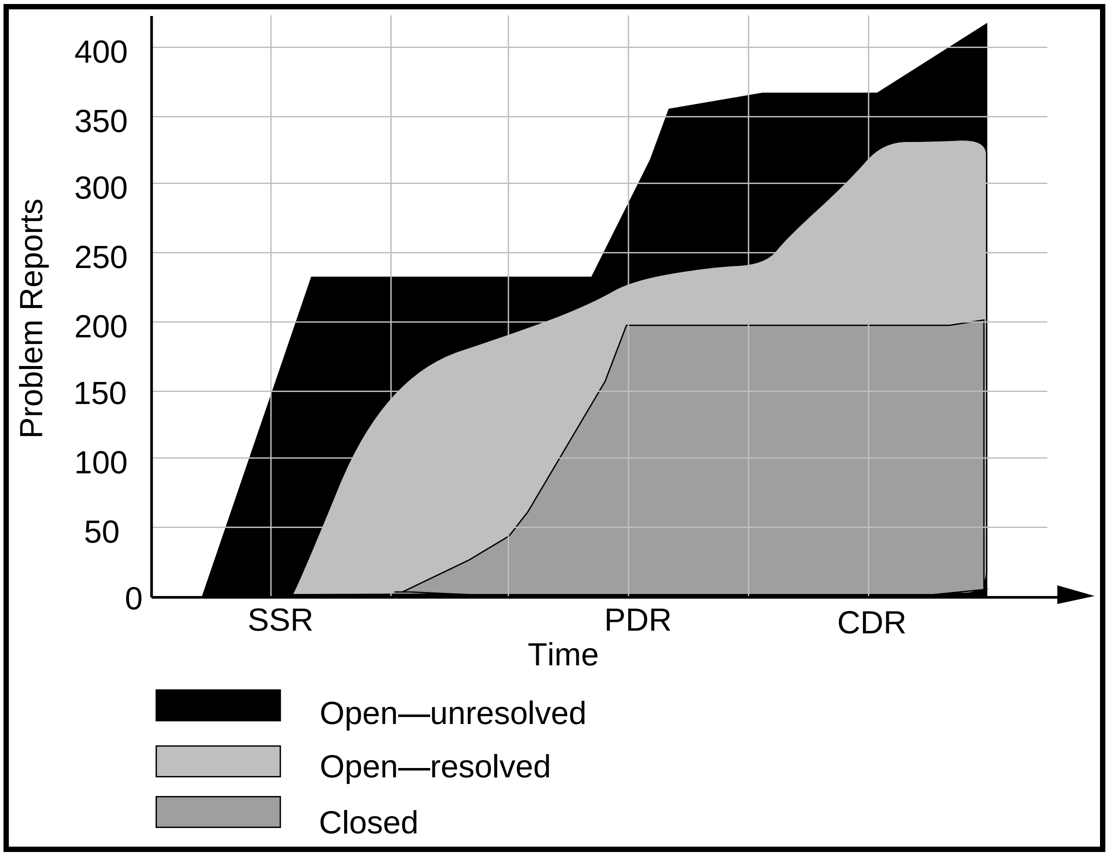


Figure 4-1 Example of Single Parameter Trend Analysis

Based on this graph, the program manager can assess whether problem report status poses a cost or schedule risk to the program. An analysis of the graph shows a large number of new problem reports around SSR, with successively smaller jumps around PDR and CDR. It also shows that the resolution rate has nearly kept pace with the new problem report identification rate. The one possible cause for concern is the closure rate, which shows that half of the total problem reports remain open, and none have closed since PDR. In the example, the failure to close problem reports should be investigated; however, given that most of the open problem reports have been resolved, they do not appear to be a significant cause of risk.

Trend analysis can also be used to compare plans with actual data. These analyses can encompass data for plans, plan changes, actuals, actual changes, projections, and projection changes. In addition to providing specific insight into the aspect of the program being measured, planned versus actual measures provide an indication of the maturity and reliability of the planning and estimating process.

Consider the CSU development progress measure shown in Figure 4-2. This measure shows planned, replanned, and actual completion rates for design, code/unit test, and integration of CSUs. The measure also shows rework. The primary uses of this measure are to assess the schedule risk and measure real technical progress. This graph shows that there was one replan just prior to CDR, which appears to have been caused by a 10 percent increase in the total CSUs defined. Actual design completion slipped another month beyond the replan. CDR, coding, and integration all began as originally scheduled. The main problem appears to be in integration progress, which is below the plan and progressing at a declining rate. The level of redesign and recoding during integration also raises a warning flag. Armed with this measure , the program manager can ask specific questions (such as queries into the nature of the rework performed) to identify the source of the integration problem.

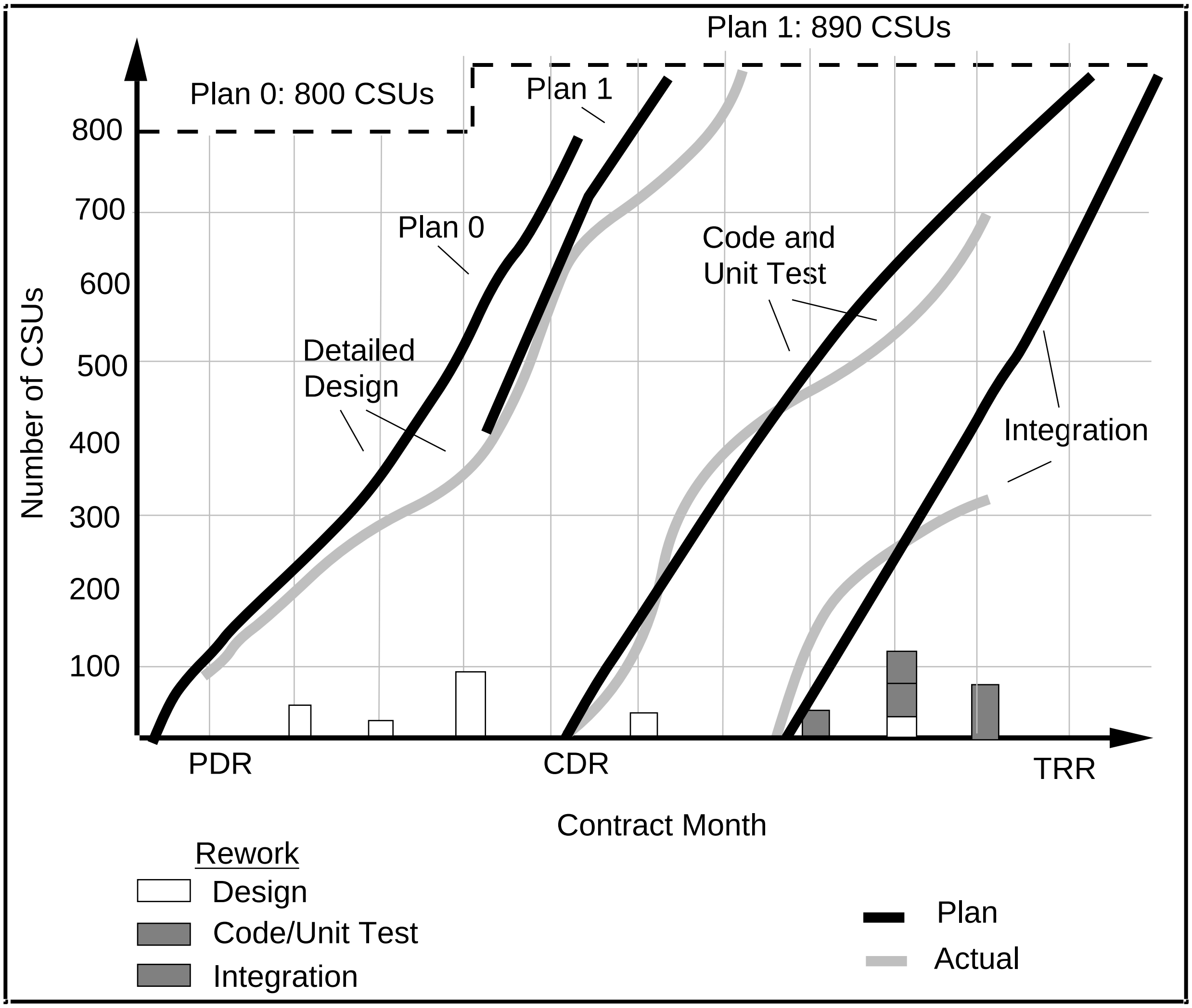


Figure 4-2 Example of Multiple Parameter Trend Analysis

4.2 Multiple Metric Relationship Analysis

The program manager should correlate the trends from all of the measures before making decisions based on them. Analysis of relationships between multiple measures is an essential tool for correlating such trends and for obtaining and confirming program insights. Often the benefit of analyzing the relationship between multiple measures is greater than the sum of the individual benefits of the same measures. An example of this is shown in Figure 4-3, where the number of software development personnel overlays the schedule. Notice that the design and implementation activities planned are highly parallel even as planned staffing is decreasing. Assuming the budget is based on the planned staffing shown, this program exhibits significant cost risk, since it is unrealistic to assume that the schedule can be maintained with the planned decrease in staffing. Also, the program has been able to stay on schedule by using more staff than planned, but just as schedule activities intensify, the staffing levels are beginning to drop below the planned level. If each of these measures were only considered independently, it is possible that the program manager would overlook the apparent inconsistency in the plans.

As shown by the example, the technique is simple to use, yet very powerful and valuable as a tool for assessing plans. Measures based on data from plans will generally exhibit one of three relationships with each other:

- Positive trend relationships, where both measures are expected to track in a consistent direction. Where the expected tracking does not occur, such as in the example above, further investigation by the program manager is warranted.
- Inverse relationships, where there is an underlying tradeoff implicit in the things being measured. A simple example is estimated cost and estimated SLOC. If the developer determines that a portion of the originally planned level of reusable software will have to be new SLOC, a negative cost impact would be expected.
- Independence, where there is no presumed relationship between the measures.

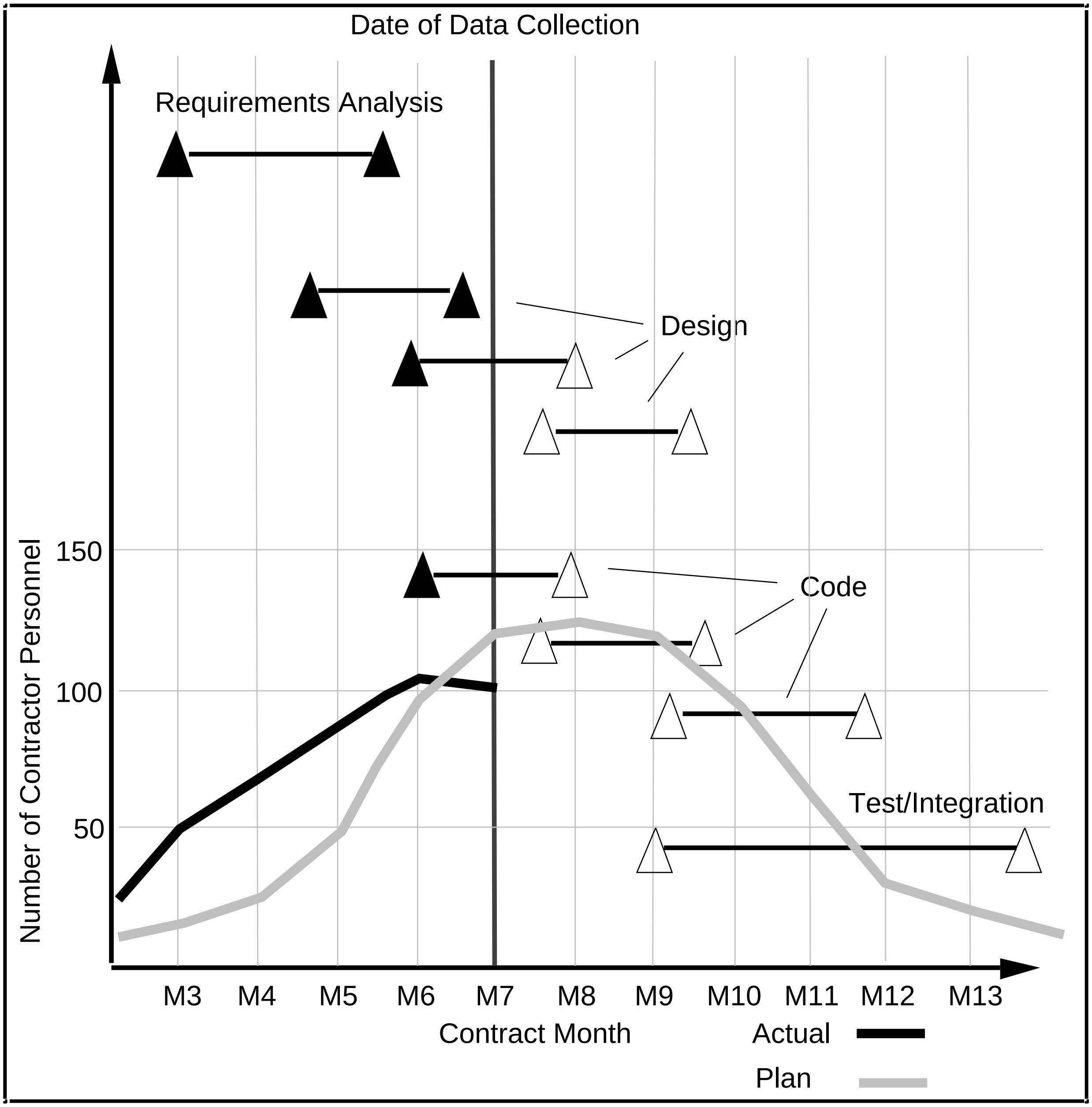


Figure 4-3 Example of Multi-Metric Relationship Analysis

4.3 Modeling Input Data Analysis

Use of models fits naturally with software measurement. Models provide specialized algorithms for predicting or estimating certain key characteristics of a software program. The most common examples are cost models and reliability models. Often these models can use the same data that is being collected for use in generating the other measures discussed previously.

Models often use data from similar, past projects as bases for comparisons and to guide decisions on the input data. Such historical data is critical in validating the model used. But because no two programs are alike, care must be used to not overemphasize the model's output. Because of the emphasis on understanding inputs to models, modeling becomes an important analysis tool for understanding contractor estimates and forecasts.

Cost models typically provide predictions of cost and schedule based on a set of input parameters, such as lines of code, labor rates, quality requirements, application characteristics, environment characteristics, and demonstrated past performance. While the actual value of certain input parameters (especially lines of code) is not available until late in the program, early estimates of these parameters have been successfully used with many cost models.

Reliability models typically estimate the number of operational failures that will occur per unit of time. The primary drawback of the most common reliability models is the unavailability of input parameter data (or reasonable estimates) until late in the development of a software system. Most reliability models use test results (failure data during operational testing) as the key input parameter. For this reason, the main benefit of reliability models occurs in the testing stages and later during post deployment software support. A typical application of a reliability model is as an aid in determining when testing can stop.

An example analysis using a cost model (in this case COCOMO) is shown in Figure 4-4. This example shows the results of independent predictions of cost and schedule by the software development contractor, the government, and an independent estimator. Given the significant difference in the predicted cost and schedule, further investigation of the underlying assumptions is warranted. Figure 4-5 shows a possible result of such an investigation, based on the government versus contractor assumptions of the input parameters driving the cost model. An item by item review with the contractor can be conducted to understand and evaluate the rationale for the input values used.

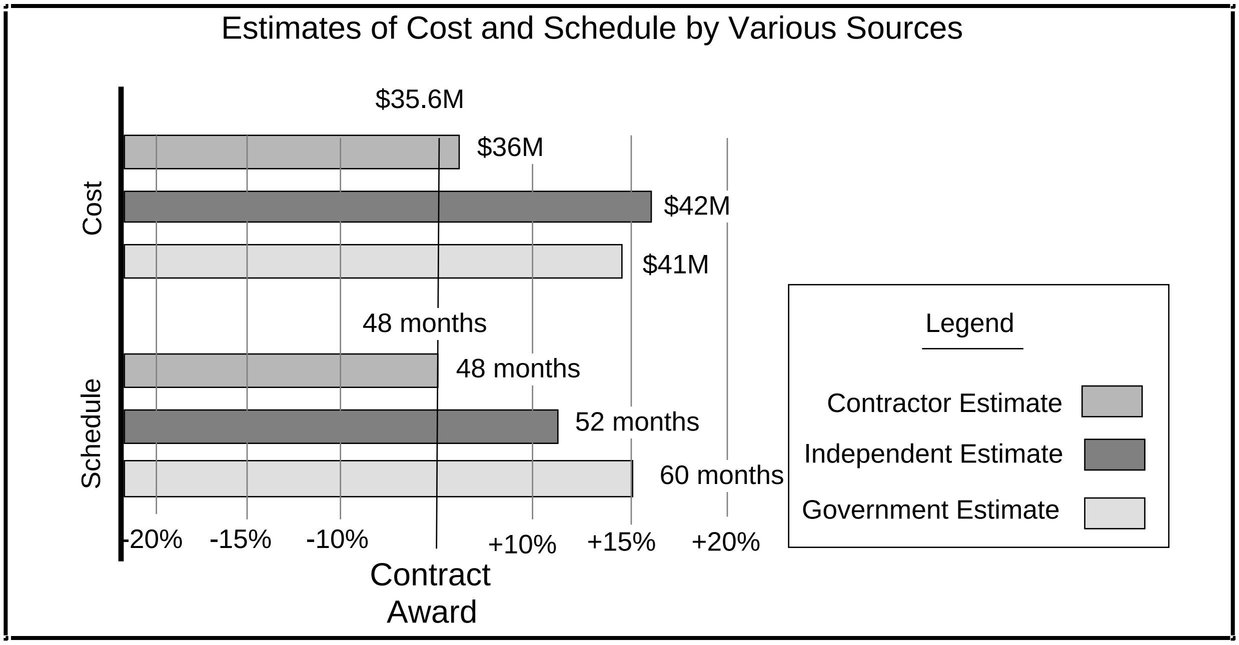
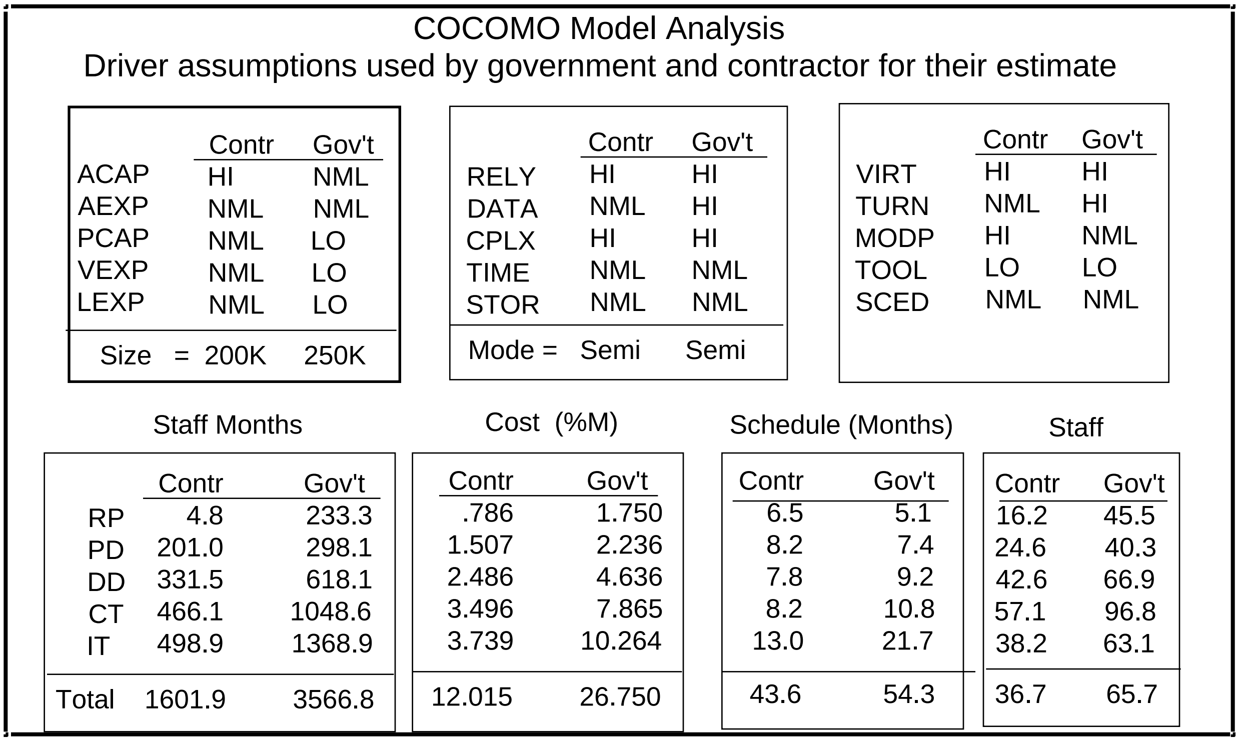


Figure 4-4 Example of Modeling Input Data Analysis Using COCOMO



COCOMO Model Analysis

Driver assumptions used by government and contractor for their estimate

	Contr	Gov't
ACAP	HI	NML
AEXP	NML	NML
PCAP	NML	LO
VEXP	NML	LO
LEXP	NML	LO
Size =	200K	250K

	Contr	Gov't
RELY	HI	HI
DATA	NML	HI
CPLX	HI	HI
TIME	NML	NML
STOR	NML	NML
Mode =	Semi	Semi

	Contr	Gov't
VIRT	HI	HI
TURN	NML	HI
MODP	HI	NML
TOOL	LO	LO
SCED	NML	NML

	Staff Months		Cost (%M)		Schedule (Months)		Staff	
	Contr	Gov't	Contr	Gov't	Contr	Gov't	Contr	Gov't
RP	4.8	233.3	.786	1.750	6.5	5.1	16.2	45.5
PD	201.0	298.1	1.507	2.236	8.2	7.4	24.6	40.3
DD	331.5	618.1	2.486	4.636	7.8	9.2	42.6	66.9
CT	466.1	1048.6	3.496	7.865	8.2	10.8	57.1	96.8
IT	498.9	1368.9	3.739	10.264	13.0	21.7	38.2	63.1
Total	1601.9	3566.8	12.015	26.750	43.6	54.3	36.7	65.7

Figure 4-5 Example of Modeling Input Data Analysis Results of Government Versus Contractor Estimates

4.4 Thresholds and Warning Limits

The program manager can also use an analysis method that is similar to statistical process control to help analyze the variability of the data received based on preconceived ideas of when an identified issue has become (or is about to become) a problem. The basic technique uses an adaptation of statistical process control charts. However, the program manger usually does not have a sample of data from which to apply statistical methods such as control limits; instead, the program manager uses subjective thresholds of variability about the plan data. That is, the program manager uses the plan data (or if plan data does not exist, determines a goal, e.g., the number of software defects) and applies ranges of variability whose boundaries are predetermined thresholds. These charts then allow the program manger to see how the contractor is performing relative to the thresholds regarding each issue.

Each measure would have an upper and lower threshold (UT and LT) and upper and lower warning limits (UWL and LWL). The program manager would predetermine the threshold limits and warning limits. These predeterminations could be dependent upon the priority of the issue, the amount of risk that is associated with an issue, and the criticality or impact an issue might have on the eventual outcome of the system (i.e., the potential loss). In Figure 4-6, the UT and LT are set at plus and minus 20 percent of the plan data and the UWL and LWL are set at plus and minus 10 percent. In practice, the thresholds and warning limits do not have to be symmetrical, e.g., an upper limit could be plus 10 percent and a lower limit could be minus 25 percent.

In Figure 4-6, productivity is plotted as staff hours per thousand function points as an example of how to apply the technique. Whenever the actual data curve falls outside the warning levels, the program manager should interpret it as a signal that potential problems are starting to show up and further investigation is needed. Seemingly good trends, such as the actual data curve being higher than the UWL, should also be investigated. The program manager investigates trends by asking probing questions and analyzing the data and contract information more closely.

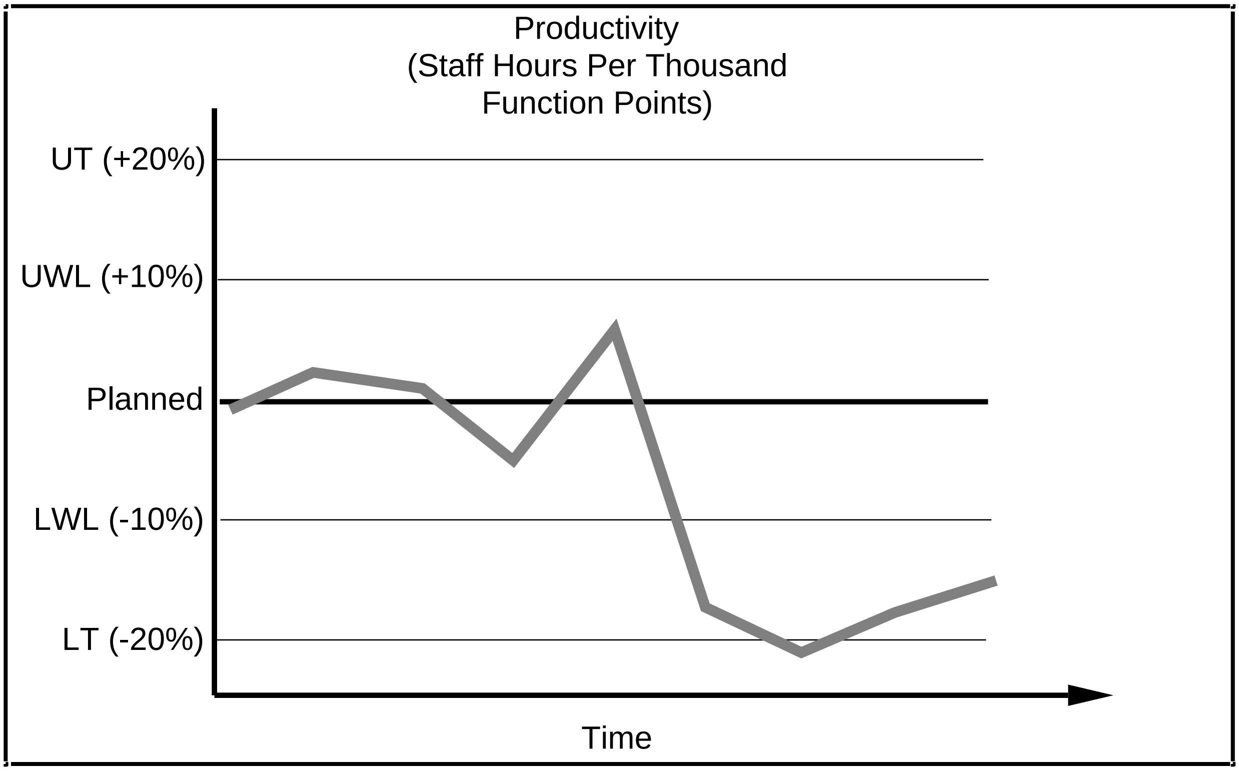


Figure 4-6 Example of Statistical Analysis Using Productivity

References

[AFSC] Software Management Metrics, AFSC Pamphlet 800-43, August 31, 1990.

[BASILI] Basili, V. and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environment," *IEEE Transactions on Software Engineering*, June 1988.

[BEAM] Beam, W. R., J. D. Palmer, and A. P. Sage, "Systems Engineering for Software Productivity," *IEEE Transactions on Systems, Man, and Cybernetics*, March/April 1987.

[BOEHM81] Boehm, B., *Software Engineering Economics*. Englewood Cliffs, New Jersey: Prentice-Hall, 1981.

[BOEHM87] Boehm, B., "Understanding and Controlling Software Costs," *IEEE Transactions on Software Engineering*, October 1988.

[BRETT] Brettschneider, R., "Is Your Software Ready for Release," *IEEE Software*, July 1989.

[BROOKS82] Brooks, F. P. Jr., *The Mythical Man-Month: Essays on Software Engineering*. Reading, Massachusetts: Addison-Wesley Publishing Co., 1982.

[BROOKS87] Brooks, F. P., "No Silver Bullet: Essence and Accidents on Software Engineering," *IEEE Computer*, April 1987.

[BROWN] Brown, J. R. and M. Lipow, *The Quantitative Measurement of Safety and Reliability*, TRW Report QR 1776, August 1973.

[BURR] Burr, T., "The Tools of Quality Part VI: Pareto Charts," *Quality Progress*, November 1990.

[CONGRESS] "Bugs in the Program: Problems in Federal Government Computer Software Development and Regulation," Staff Study by the Subcommittee on Investigations and Oversight transmitted to the Committee on Science, Space, and Technology - U.S. House of Representatives 101st Congress 1st Session, September 1989.

[CORI] Cori, K., "Fundamentals of Master Scheduling for the Project Manager," *Project Management Journal*, June 1985.

[DEMARCO] DeMarco, T., *Controlling Software Projects*. New York: Yourdan Press, 1982.

[DOD2167A] *Military Standard - Defense System Software Development*, DOD-STD-2167A, February 1988.

[DOD5000.1] *Department of Defense Directive - Major Systems Acquisitions,* DOD-DIR 5000.1, February 1991.

[HUMPHREY] Humphrey, W. S., *Managing the Software Process*. Reading, Massachusetts: Addison-Wesley Publishing Co., 1989.

[IEEE1045] *Standard for Software Productivity Metrics*, IEEE P1045/D4.0, December 1990.

[IFPUG] *Function Points as Assets-Reporting to Management,* The International Function Point User Group, February 1991.

[ISHIKAWA] Ishikawa, K., *Guide to Quality Control.* Tokyo: Noridca International Limited, 1989.

[JURAN] The Juran Institute, "The Tools of Quality; Part V: Check Lists," *Quality Progress*, October 1990.

[MUSA] Musa, J. D., A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*. New York: McGraw Hill, 1987.

[ONEILL] O'Neill, D. and C. Ingram, "Software Inspections Tutorial," *SEI Technical Review*, 1988.

[PACKARD] Packard, D., *Quest for Excellence, Final Report to the President,* D. Packard on behalf of the President's Blue Ribbon Commission on Defense Management, Superintendent of Documents, GPO, Washington, D.C., 1986.

[SCHULTZ] Schultz, H., *Software Management Metrics*, ESD-TR-88-001, May 1988.

[SEI89] *Proceedings of the Workshop on Executive Software Issues August 2-3 and November 18, 1988*, SEI Technical Report, CMU/SEI-89-TR-6, ADA 206779, January 1989.

[SHEWART] Shewart, W. A., *Statistical Method From the Viewpoint of Quality Control*, The Graduate School - The Department of Agriculture, Washington D.C., 1939.

[USAF] *Report of the DoD Joint Service Task Force on Software Problems*, Lt. Col. Larry E. Druffel, USAF - Task Force Chairman, July 1982.

[WALTON] Walton, M., *The Deming Management Method*. New York: The Putnam Publishing Group, 1986.

Acronyms

BPS	Bytes Per Second
CDR	Critical Design Review
CDRL	Contract Data Requirements List
CM	Configuration Management
COCOMO	Constructive Cost Model
COTS	Commercial Off the Shelf
CPU	Central Processing Unit
CRU	Computer Resource Utilization
CSC	Computer Software Component
CSCI	Computer Software Configuration Item
CSU	Computer Software Unit
DoD	Department of Defense
FCA	Functional Configuration Audit
GFE	Government Furnished Equipment
GFI	Government Furnished Information
IDD	Interface Design Document
IEEE	Institute of Electrical and Electronics Engineers
I/O	Input/Output
IRS	Interface Requirements Specification
IV&V	Independent Verification and Validation
KSLOC	Thousands of Source Lines of Code
LT	Lower Threshold
LWL	Lower Warning Limit
MCCR	Mission Critical Computer Resources
MIPS	Millions of Instructions Per Second

PCA	Physical Configuration Audit
PDCA	Plan-Do-Check-Act
PDR	Preliminary Design Review
RAM	Random Access Memory
ROM	Read Only Memory
RFP	Request For Proposal
SAMWG	Software Acquisition Metrics Working Group
SDD	Software Design Document
SDP	Software Development Plan
SDR	System Design Review
SEI	Software Engineering Institute
SLOC	Source Lines of Code
SPR	Software Problem Report
SQA	Software Quality Assurance
SRS	Software Requirements Specification
SSR	Software Specification Review
SSDD	System/Segment Design Document
SSS	System/Segment Specification
TR	Technical Report
TRR	Test Readiness Review
USAF	United States Air Force
UT	Upper Threshold
UWL	Upper Warning Limit
WBS	Work Breakdown Structure