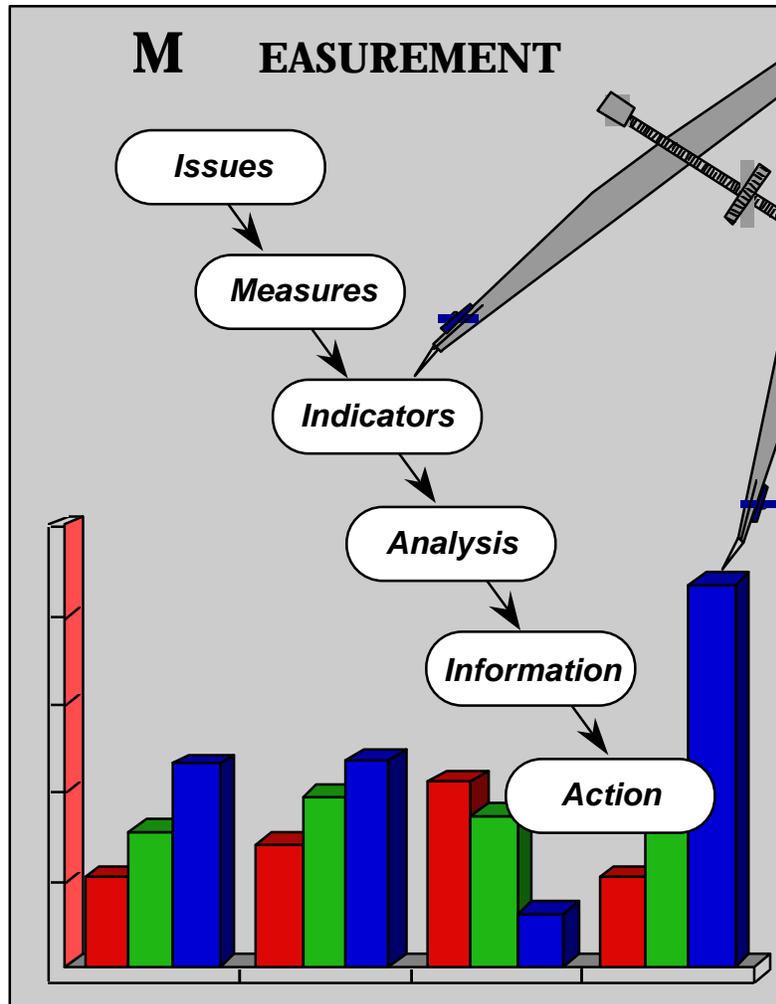


PRACTICAL

SOFTWARE



A guide to objective program insight

FOREWORD

One of the most challenging tasks in the Department of Defense is delivering a software intensive system that meets program cost, schedule, and performance objectives. With more of the capability in today's Weapons and Automated Information Systems implemented in software, effective management of the software development and support efforts has become critical to program success.

As a DoD Program Manager, the introduction of new acquisition requirements and new software technologies increases the need for more effective software management techniques. Key software issues have to be identified, prioritized, and managed. You have to have the right information to make informed decisions about the software issues throughout the course of the program.

Practical Software Measurement: A Guide to Objective Program Insight, was developed to help you meet these software management challenges. *PSM* describes how to define and implement a software measurement process to address the unique management and information needs of your program. The guidance in *Practical Software Measurement* is based on actual software measurement experience on successful DoD and industry programs. It represents the best practices used by measurement professionals within the software acquisition and engineering communities.

This version of *Practical Software Measurement* explains how to select the measures for your program and how to analyze the measurement results to manage your software issues. *Practical Software Measurement* Version 3.0, currently under development, will contain more extensive software analysis and estimation guidance. We welcome your contributions to *Practical Software Measurement*, and your participation in the project.

John McGarry
Naval Undersea Warfare Center

ACKNOWLEDGMENTS

The following software measurement professionals have been principal contributors in the development of *Practical Software Measurement: A Guide to Objective Program Insight*.

Elizabeth Bailey
INSTITUTE FOR DEFENSE ANALYSES

Cheryl Jones
NAVAL UNDERSEA WARFARE CENTER

David Card
SOFTWARE PRODUCTIVITY SOLUTIONS INC.

Beth Layman
SOFTWARE PRODUCTIVITY SOLUTIONS INC.

Joseph Dean
TECOLOTE RESEARCH INC.

John McGarry
NAVAL UNDERSEA WARFARE CENTER

The following software measurement professionals have participated in the development of *Practical Software Measurement: A Guide to Objective Program Insight*.

Bruce Allgood
US AIR FORCE
SOFTWARE TECHNOLOGY SUPPORT CENTER

Paul Janusz
US ARMY ARDEC

James Arthur
VIRGINIA POLYTECHNIC INSTITUTE

John Keddy
NAVAL UNDERSEA WARFARE CENTER

Lt. Col. Terrence Brotherton
INFORMATION RESOURCES MANAGEMENT COLLEGE

Kenneth Kelley
DEFENSE INFORMATION SYSTEMS AGENCY

Luke Campbell
NAVAL AIR WARFARE CENTER

Ron Larson
US NAVY PEO CU

Anita Carleton
SOFTWARE ENGINEERING INSTITUTE

Steven Law
DEFENSE INFORMATION SYSTEMS AGENCY

David Castellano
US ARMY ARDEC

Scott Lucero
US ARMY
OPERATIONAL TEST AND EVALUATION COMMAND

Carl Crawford
NAVAL SURFACE WARFARE CENTER

John Marciniak
KAMA SCIENCES CORPORATION

Deborah DeToma
GTE GOVERNMENT SYSTEMS CORPORATION

Charles McPherson
US ARMY MATERIEL COMMAND

Jim Dobbins
DEFENSE SYSTEMS MANAGEMENT COLLEGE

Richard Nance
VIRGINIA POLYTECHNIC INSTITUTE

David R. Erickson
US AIR FORCE
SOFTWARE TECHNOLOGY SUPPORT CENTER

Raymond Paul
OFFICE OF THE UNDER SECRETARY OF DEFENSE - A&T

Mike Falat
DEFENSE INFORMATION SYSTEMS AGENCY

Margaret Powell
NAVAL INFORMATION SYSTEMS MANAGEMENT CENTER

William Farr
NAVAL SURFACE WARFARE CENTER

Edward Primm
NAVAL SURFACE WARFARE CENTER

Stewart Fenick
US ARMY CECOM

Bryce Ragland
US AIR FORCE
SOFTWARE TECHNOLOGY SUPPORT CENTER

William Florac
SOFTWARE ENGINEERING INSTITUTE

Anthony Shumskas
BDM ENGINEERING SERVICES COMPANY

John Gaffney
SOFTWARE PRODUCTIVITY CONSORTIUM

Lynn Simms
LOGICON

Tony Guido
NAVAL AIR SYSTEMS COMMAND

Raghu Singh
SPACE AND NAVAL WARFARE SYSTEMS COMMAND

Fred Hall
INDEPENDENT ENGINEERING INC.

O.T. Smith
US AIR FORCE MATERIEL COMMAND

Robert Hegland
US ARMY
INFORMATION SYSTEMS SOFTWARE CENTER

George Stark
THE MITRE CORPORATION

Jeffrey Heimberger
SOFTWARE PRODUCTIVITY SOLUTIONS INC.

Stephen Thompson
US ARMY PEO STAMIS

Scott Hissam
LORAL DEFENSE SYSTEMS - EAST

Sharyn Tolochko
US ARMY ARDEC

Kevin Holt
DEFENSE LOGISTICS AGENCY

TABLE OF CONTENTS

PART 1 - THE SOFTWARE MEASUREMENT PROCESS

CHAPTER 1 - PROGRAM MANAGEMENT AND THE MEASUREMENT PROCESS..... 7

| | |
|---|-------------------------------------|
| 1.1 Managing a Software Intensive Program..... | 7 |
| 1.2 Overview of Measurement Process..... | Error! Bookmark not defined. |
| 1.3 Software Measurement Principles..... | 11 |
| 1.3.1 Program Issues and Objectives..... | 13 |
| 1.3.2 Developer's Software Process..... | 14 |
| 1.3.3 Low Level Data..... | 14 |
| 1.3.4 Independent Analysis Capability..... | 15 |
| 1.3.5 Structured Analysis Process..... | 15 |
| 1.3.6 Results in Program Context..... | 16 |
| 1.3.7 Life Cycle Integration..... | 17 |
| 1.3.8 Objective Communications..... | 18 |
| 1.3.9 Single-Program Analysis..... | 19 |
| 1.4 Measurement Implementation Considerations..... | 20 |

CHAPTER 2 – TAILORING SOFTWARE MEASURES..... 21

| | |
|---|-----------|
| 2.1 Measurement Tailoring Overview..... | 21 |
| 2.2 Identify and Prioritize Program Issues..... | 22 |
| 2.2.1 Program-Specific Issues..... | 23 |
| 2.2.2 Common Software Issues..... | 24 |
| 2.2.3 Identifying Program Issues..... | 25 |
| 2.2.4 Prioritizing Program Issues..... | 26 |
| 2.3 Select and Specify Program Measures..... | 27 |
| 2.3.1 Measurement Category Selection..... | 28 |
| 2.3.2 Measurement Selection Criteria..... | 28 |
| 2.3.3 Specifying Data and Implementation Requirements..... | 30 |
| 2.4 Integrate measures into the developer's process..... | 33 |
| 2.4.1 Characterizing the Software Environment..... | 34 |
| 2.4.2 Identifying Measurement Opportunities..... | 35 |
| 2.4.3 Developing a Software Measurement Plan..... | 36 |

CHAPTER 3 – APPLYING SOFTWARE MEASURES..... 39

| | |
|--|-----------|
| 3.1 Collect and Process Data..... | 40 |
| 3.1.1 Data Sources..... | 40 |
| 3.1.2 Reporting and Processing..... | 41 |
| 3.1.3 Normalization and Aggregation..... | 42 |
| 3.1.4 Data Verification..... | 42 |
| 3.2 Define and Generate indicators..... | 43 |
| 3.2.1 Basic Indicator Concepts..... | 44 |
| 3.2.2 Types of Indicators..... | 47 |

| | |
|--|------------------------------|
| 3.3 Analyze Issues..... | 49 |
| 3.4 Report Results..... | 57 |
| 3.5 Take Action..... | 58 |
| 3.6 Life Cycle Application..... | 59 |
| 3.6.1 Program Planning..... | 59 |
| 3.6.2 Development..... | 61 |
| 3.6.3 Software Support..... | 62 |
| CHAPTER 4 - IMPLEMENTING A MEASUREMENT PROCESS..... | 65 |
| 4.1 Measurement Implementation Overview..... | 65 |
| 4.2 Measurement Implementation Activities..... | 66 |
| 4.2.1 Obtain Organizational Support..... | 67 |
| 4.2.2 Define Measurement Responsibilities..... | 68 |
| 4.2.3 Provide Measurement Resources..... | 70 |
| 4.2.4 Initiate the Measurement Process..... | 74 |
| 4.3 Using the Measurement Results..... | 75 |
| 4.3.1 Program Development Viewpoint..... | 76 |
| 4.3.2 DoD Executive Management Viewpoint..... | 77 |
| 4.3.3 Process Improvement Viewpoint..... | 78 |
| 4.3.4 Lessons Learned..... | 78 |
| PART 2 - SELECTING AND SPECIFYING PROGRAM MEASURES | |
| CHAPTER 1- HOW TO SELECT AND SPECIFY PROGRAM MEASURES.. | 87 |
| 1.1 Introduction..... | 87 |
| 1.2 Identifying and Prioritizing Program Issues..... | 90 |
| 1.3 Selecting the Appropriate Measurement Categories..... | Error! Bookmark not defined. |
| 1.4 Selecting the Applicable Measures..... | 93 |
| 1.5 Specifying Measurement Data and Implementation Requirements..... | 95 |
| 1.6 Selecting and Specifying Measures for Existing Programs..... | 98 |
| CHAPTER 2 – DETAILED MEASUREMENT SELECTION AND SPECIFICATION INFORMATION..... | 101 |
| 2.1 Introduction..... | 101 |
| 2.2 How To Use the Measurement Tables..... | 101 |
| 2.2.1 Measurement Category Tables..... | 102 |
| 2.2.2 Measurement Description Tables..... | 104 |
| 2.2.3 General Measurement Specification Table..... | 107 |
| 2.2.4 Additional Implementation Guidance..... | 107 |
| 2.2.5 Measurement Selection and Specification Tables..... | 108 |
| CHAPTER 3 – MEASUREMENT SELECTION AND SPECIFICATION EXAMPLE..... | 173 |

| | |
|--|-----|
| 3.1 Program Scenario..... | 173 |
| 3.2 Measurement Selection Summary..... | 174 |

PART 3 - ANALYSIS TECHNIQUES AND EXAMPLES

| | |
|--|------------|
| CHAPTER 1 – MEASUREMENT APPLICATION OVERVIEW..... | 183 |
| 1.1 Collect and Process Data..... | 183 |
| 1.2 Define And Generate Indicators..... | 184 |
| 1.3 Analyze Issues..... | 185 |
| 1.4 Report Results..... | 185 |
| 1.5 Take Action..... | 186 |
| CHAPTER 2 – INDICATOR REPRESENTATION..... | 187 |
| CHAPTER 3 – SINGLE INDICATOR EXAMPLES..... | 191 |
| 3.x Indicator Name..... | 192 |
| 3.1 Milestone Progress Indicator..... | 194 |
| 3.2 Design Progress Indicator..... | 196 |
| 3.3 Schedule Variance Indicator..... | 198 |
| 3.4 Incremental Build Content Indicator..... | 200 |
| 3.5 Effort Allocation Indicator..... | 202 |
| 3.6 Staff Experience Indicator..... | 204 |
| 3.7 Cost Profile Indicator..... | 207 |
| 3.8 Resource Utilization Indicator..... | 209 |
| 3.9 Software Size Indicator..... | 211 |
| 3.10 Requirements Stability Indicator..... | 213 |
| 3.11 Response Time Indicator..... | 215 |
| 3.12 Problem Report Status Indicator..... | 217 |
| 3.13 Problem Report Aging Indicator..... | 218 |
| 3.14 Defect Density Indicator..... | 221 |
| 3.15 Software Complexity Indicator..... | 223 |
| 3.16 Software Process Maturity Indicator..... | 225 |
| 3.17 Software Productivity Indicator..... | 227 |
| 3.18 Rework Effort Indicator..... | 230 |
| 3.19 Software Origin Indicator..... | 233 |

| | |
|---|-------------|
| CHAPTER 4 – INTEGRATED INDICATOR EXAMPLES..... | .235 |
| 4.1 Design Completion Analysis..... | 236 |
| 4.2 Test Completion Analysis..... | 238 |
| 4.3 Readiness for Delivery Analysis..... | 240 |
| 4.4 Maintenance Analysis..... | 242 |

PART 4 - ACQUISITION AND CONTRACT IMPLEMENTATION GUIDANCE

| | |
|--|------------------------------|
| CHAPTER 1 – CONTRACT IMPLEMENTATION GUIDANCE..... | 251 |
| 1.1 Contract Planning and Preparation..... | 251 |
| 1.2 Proposal Evaluation..... | Error! Bookmark not defined. |
| 1.3 Negotiations..... | 252 |
| 1.4 Contract Modifications..... | 253 |

| | |
|--|------------|
| CHAPTER 2 – SAMPLE RFP WORDING..... | 255 |
|--|------------|

| | |
|--|------------|
| CHAPTER 3 – ADDITIONAL SAMPLE MATERIAL..... | 261 |
|--|------------|

PART 5 - SOFTWARE MEASUREMENT CASE STUDIES

| | |
|---------------------------------------|------------|
| WEAPONS SYSTEM CASE STUDY..... | 271 |
|---------------------------------------|------------|

| | |
|--|------------------------------|
| CHAPTER 1 - PROGRAM OVERVIEW..... | 275 |
| 1.1 Introduction..... | Error! Bookmark not defined. |
| 1.2 Program Technical Approach..... | 277 |
| 1.2.1 System Requirements Definition and Design Analysis..... | 277 |
| 1.2.2 DDG 51 C ⁴ I Baseline System Description..... | 278 |
| 1.2.3 System Requirements and Design Recommendations..... | 280 |
| 1.3 Program Management Approach..... | 282 |

| | |
|--|------------|
| CHAPTER 2 - PROGRAM PLANNING AND ACQUISITION..... | 285 |
| 2.1 Software Program Planning..... | 285 |
| 2.2 Software Acquisition..... | 288 |
| 2.2.1 Request for Proposal..... | 288 |
| 2.2.2 Proposal Evaluation..... | 289 |
| 2.2.3 Award..... | 291 |
| 2.2.4 Negotiations..... | 293 |

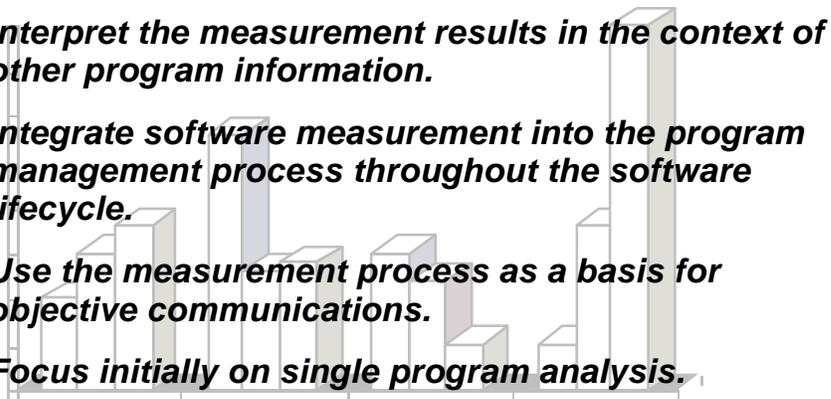
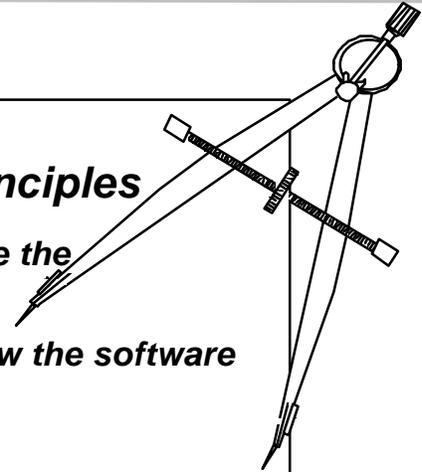
| | |
|---|------------|
| CHAPTER 3 - DEVELOPMENT PHASE..... | 297 |
|---|------------|

| | |
|--|------------|
| 3.1 Tracking Development Performance..... | 297 |
| 3.1.1 Software Measurement Overview..... | 297 |
| 3.1.2 Software Issue Identification and Analysis..... | 298 |
| 3.2 Revising The Development Plan..... | 307 |
| 3.3 Software Delivery..... | 309 |
| 3.4 Epilogue..... | 310 |
| | |
| AUTOMATED INFORMATION SYSTEM CASE STUDY..... | 313 |
| | |
| CHAPTER 1 - PROGRAM OVERVIEW..... | 317 |
| 1.1 Introduction..... | 317 |
| 1.2 Air Force Business Process Modernization Initiative..... | 319 |
| 1.3 Program Description..... | 320 |
| 1.4 System Architecture and Functionality..... | 322 |
| 1.4.1 Current Personnel System..... | 322 |
| 1.4.2 Military Automated Personnel System (MAPS)..... | 323 |
| | |
| CHAPTER 2 - GETTING THE PROGRAM UNDER CONTROL..... | 327 |
| 2.1 Evaluating the Software Development Plan..... | 327 |
| 2.2 Revising the Software Development Plan..... | 330 |
| 2.3 Tracking Performance Against the Revised Plan..... | 334 |
| | |
| CHAPTER 3 - EVALUATING READINESS FOR DELIVERY..... | 341 |
| 3.1 Increment 1..... | 341 |
| 3.2 Increment 2..... | 345 |
| | |
| CHAPTER 4 - INSTALLATION AND SOFTWARE SUPPORT..... | 349 |
| 4.1 Increment 1 Installation..... | 349 |
| 4.2 Software Support..... | 350 |
| 4.3 Epilogue..... | 353 |
| | |
| PART 6 - SUPPLEMENTAL INFORMATION | |
| | |
| GLOSSARY..... | 357 |
| | |
| ACRONYMS..... | 365 |
| | |
| BIBLIOGRAPHY..... | 367 |

| | |
|---|------------|
| Software Measurement References..... | 367 |
| Government Agency-Specific Software Measurement References..... | 372 |
| PSM RELATIONSHIP TO SPECIFIC DOD POLICIES..... | 373 |
| PSM PROJECT INFORMATION SUMMARY..... | 377 |
| Use of Practical Software Measurement..... | 378 |
| Project Contact Information..... | 378 |
| Version Description Summary..... | 381 |

Software Measurement Principles

- *Program issues and objectives drive the measurement requirements.*
- *The developer's process defines how the software is actually measured.*
- *Collect and analyze low level data.*
- *Implement an independent analysis capability.*
- *Use a structured analysis process to trace the measures to the decisions.*
- *Interpret the measurement results in the context of other program information.*
- *Integrate software measurement into the program management process throughout the software lifecycle.*
- *Use the measurement process as a basis for objective communications.*
- *Focus initially on single program analysis.*



SCOPE AND STRUCTURE OF THE GUIDE

This document, *Practical Software Measurement: A Guide to Objective Program Insight (PSM)*, is intended to provide a basic introduction to software measurement for Department of Defense (DoD) program managers and their technical staff responsible for implementing a software measurement program. *PSM* applies to both weapons systems and automated information systems. Although the *Guide* is written from a DoD perspective, the principles and approach of *PSM* apply equally well to large scale commercial and other government software programs.

While it supports DoD and commercial software process and acquisition standards, *PSM* does not depend on the adoption of any specific standard. It provides a flexible framework for integrating measurement into software management and development processes. The *Guide* covers three major topics:

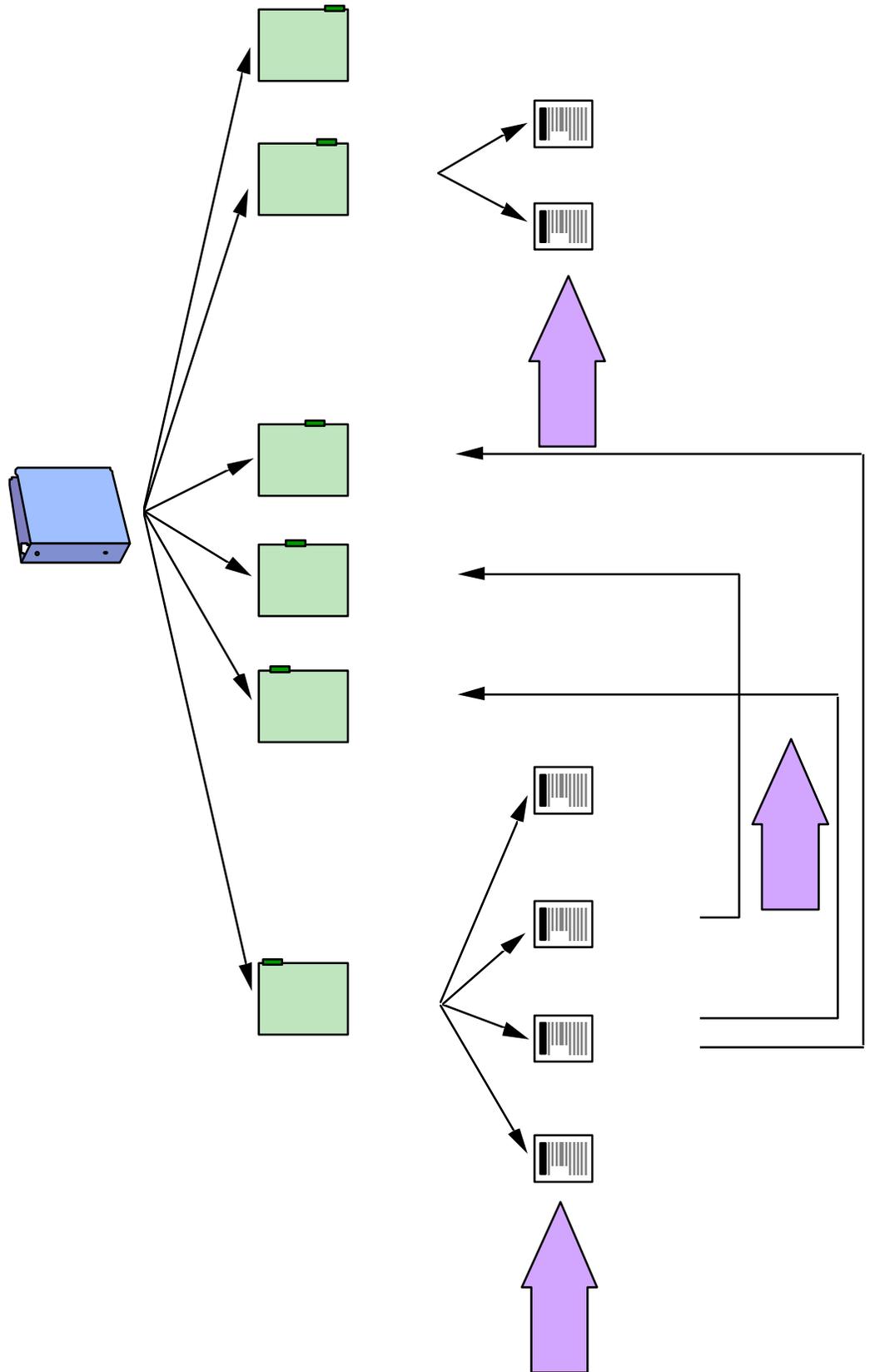
- tailoring the software measures to meet program needs
- applying software measures to obtain insight into program issues
- implementing a measurement process within an organization

The *Guide* addresses program management, with a focus on program tracking. While the measures used to achieve management insight and control over a program also are used for the purposes of process improvement and product engineering, those topics are not explored except where they are necessary to attain the program manager's goals. Even in the program management area, the *Guide* does not provide an exhaustive treatment of all possible measures. Instead, it focuses on the most commonly used measures and techniques.

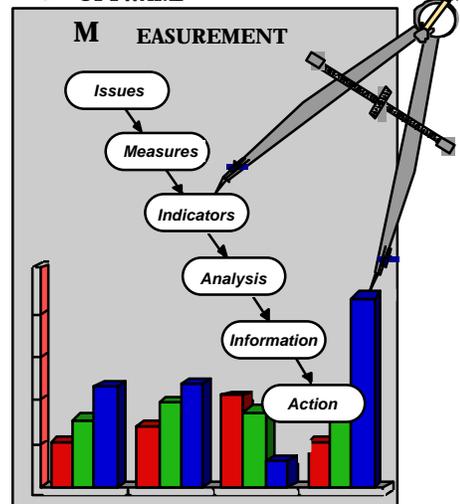
The *Guide* is organized into six parts that provide increasingly detailed treatments of the three topics: tailoring, applying, and implementing software measures. The six parts are as follows:

- Part 1 - *The Software Measurement Process*, describes a basic measurement process which can be applied to any program. In particular, it focuses on providing the DoD program manager with visibility into an ongoing program.
- Part 2 - *Selecting and Specifying Program Measures*, provides a series of tables that help the user to select the set of measures that most cost-effectively address the program's issues.
- Part 3 - *Analysis Techniques and Examples*, explains basic analysis techniques and provides sample analyses.
- Part 4 - *Acquisition and Contract Implementation*, provides examples of contract language and work break-down structures used to specify measurement requirements in two-party contractual situations.
- Part 5 - *Software Measurement Case Studies*, illustrate many of the key points made in the *Part 1 of the Guide*. The case studies include two complete examples of software measurement as applied to typical DoD programs. One example is a weapon system. The other is an automated information system.
- Part 6 - *Supplemental Information*, contains the glossary, acronyms, bibliography, project description, and comment form.

The following figure shows how these parts of the Guide address the elements of the measurement process. Part 1 introduces the basic concepts, principles, and terminology of *PSM*. Everyone should read this part of the *Guide*. Parts 2, 3, and 4 serve as detailed references to the program manager and measurement analyst for the performance of measurement functions. The reader should familiarize himself with the contents and organization of these sections, but need not read them in detail until performing the corresponding function. Part 5 illustrates the application of *PSM* in two typical program scenarios. Read at least the case study that most closely approximates the type of program you are involved with. Part 6 may be useful for clarification at any time.



PRACTICAL
S SOFTWARE



THE SOFTWARE MEASUREMENT PROCESS

PART 1

THE SOFTWARE MEASUREMENT PROCESS

*Measurement is a key element of successful management in every well-established engineering discipline. **Practical Software Measurement** presents a proven strategy for tailoring, applying, and implementing an effective measurement process for DoD software intensive Weapons System and Automated Information System (AIS) programs. The objective is to provide the DoD Program Manager with the software information required to make informed decisions which impact program schedule, cost, and technical objectives.*

PSM describes software measurement as a systematic, but flexible process which is an integral part of the overall program management structure. The PSM measurement process is issue driven. It is uniquely adapted to meet each program's specific information needs. The process is defined around a set of proven characteristics derived from actual experience on successful DoD programs. These characteristics, called software measurement principles, help to make the PSM measurement process an effective management tool, and not just another program management "requirement".

Part 1 of Practical Software Measurement describes the principles and techniques for tailoring, applying, and implementing an effective software measurement process. It presents a comprehensive view of the complete measurement approach in terms of "what" should be done. Other parts of the Guide contain detailed "how to" guidance for key measurement activities and supporting information.

This part of the Guide is organized into four chapters:

- Chapter 1 - Program Management and the Measurement Process, explains the relationship between measurement and management, and introduces the PSM software measurement principles.*
- Chapter 2 - Tailoring Software Measures, describes a sequential approach for tailoring measurement to*

directly address program specific software issues and objectives,

- *Chapter 3 - Applying Software Measures, describes a structured approach for converting software measurement data into actionable program management information.*
- *Chapter 4 - Implementing a Measurement Process, describes the activities required to get measurement into practice within an organization*

The PSM measurement process provides the foundation for making informed “software” program management decisions. It describes how to define and integrate program measurement requirements, how to collect and analyze measurement data, and how to implement the overall process into your organization.

TABLE OF CONTENTS

| | |
|--|-----------|
| CHAPTER 1 - PROGRAM MANAGEMENT AND THE MEASUREMENT PROCESS..... | 7 |
| 1.1 Managing a Software Intensive Program..... | 7 |
| 1.2 Overview of Measurement Process..... | 10 |
| 1.3 Software Measurement Principles..... | 11 |
| 1.3.1 Program Issues and Objectives..... | 13 |
| 1.3.2 Developer’s Software Process..... | 14 |
| 1.3.3 Low Level Data..... | 14 |
| 1.3.4 Independent Analysis Capability..... | 15 |
| 1.3.5 Structured Analysis Process..... | 16 |
| 1.3.6 Results in Program Context..... | 17 |
| 1.3.7 Life Cycle Integration..... | 17 |
| 1.3.8 Objective Communications..... | 18 |
| 1.3.9 Single-Program Analysis..... | 19 |
| 1.4 Measurement Implementation Considerations..... | 20 |
| CHAPTER 2 – TAILORING SOFTWARE MEASURES..... | 21 |
| 2.1 Measurement Tailoring Overview..... | 21 |
| 2.2 Identify and Prioritize Program Issues..... | 22 |
| 2.2.1 Program-Specific Issues..... | 23 |
| 2.2.2 Common Software Issues..... | 24 |
| 2.2.3 Identifying Program Issues..... | 25 |
| 2.2.4 Prioritizing Program Issues..... | 26 |
| 2.3 Select and Specify Program Measures..... | 27 |
| 2.3.1 Measurement Category Selection..... | 28 |
| 2.3.2 Measurement Selection Criteria..... | 28 |
| 2.3.3 Specifying Data and Implementation Requirements..... | 30 |
| 2.4 Integrate measures into the developer's process..... | 33 |
| 2.4.1 Characterizing the Software Environment..... | 34 |
| 2.4.2 Identifying Measurement Opportunities..... | 35 |
| 2.4.3 Developing a Software Measurement Plan..... | 36 |
| CHAPTER 3 – APPLYING SOFTWARE MEASURES..... | 39 |
| 3.1 Collect and Process Data..... | 40 |
| 3.1.1 Data Sources..... | 40 |
| 3.1.2 Reporting and Processing..... | 41 |
| 3.1.3 Normalization and Aggregation..... | 42 |
| 3.1.4 Data Verification..... | 42 |
| 3.2 Define and Generate indicators..... | 43 |
| 3.2.1 Basic Indicator Concepts..... | 44 |
| 3.2.2 Types of Indicators..... | 47 |
| 3.3 Analyze Issues..... | 49 |
| 3.4 Report Results..... | 57 |

| | |
|--|-----------|
| 3.5 Take Action..... | 58 |
| 3.6 Life Cycle Application..... | 60 |
| 3.6.1 Program Planning..... | 60 |
| 3.6.2 Development..... | 61 |
| 3.6.3 Software Support..... | 62 |
| | |
| CHAPTER 4 - IMPLEMENTING A MEASUREMENT PROCESS..... | 65 |
| 4.1 Measurement Implementation Overview..... | 65 |
| 4.2 Measurement Implementation Activities..... | 66 |
| 4.2.1 Obtain Organizational Support..... | 67 |
| 4.2.2 Define Measurement Responsibilities..... | 68 |
| 4.2.3 Provide Measurement Resources..... | 70 |
| 4.2.4 Initiate the Measurement Process..... | 74 |
| 4.3 Using the Measurement Results..... | 75 |
| 4.3.1 Program Development Viewpoint..... | 77 |
| 4.3.2 DoD Executive Management Viewpoint..... | 77 |
| 4.3.3 Process Improvement Viewpoint..... | 78 |
| 4.3.4 Lessons Learned..... | 78 |

CHAPTER 1 - PROGRAM MANAGEMENT AND THE MEASUREMENT PROCESS

Measurement is a key element of successful management in every well-established engineering discipline. This chapter introduces a flexible strategy for applying software measurement to improve program management effectiveness for Department of Defense (DoD) Weapons, and Automated Information Systems (AIS) systems. *Practical Software Measurement* presents a systematic approach that helps a program use software measurement to address specific program requirements. This chapter introduces nine basic software measurement principles that guide the implementation of a measurement process adapted to meet program needs.

1.1 MANAGING A SOFTWARE INTENSIVE PROGRAM

Much of the capability in today's DoD Weapons and Automated Information Systems is implemented with software. In the current acquisition environment, the ability of the Program Manager to effectively manage the critical software issues has become an important factor in a program's success. With the reductions in available resources and the use of new software technologies, the ability to successfully deliver a large and complex software system is increasingly challenging. New methods are required to help the DoD program manager plan, monitor and control the software processes and products which are now a large part of every program.

In both the DoD and in industry, software measurement has proven to be an effective tool in helping to manage software intensive programs. Software measurement, when integrated into the overall program management process, provides the information necessary to identify and manage the software issues which are inherent in every program. It helps the Program Manager identify specific problems, assess the impacts of these problems on program cost, schedule, and capability objectives, develop alternative solutions, and select the best approach for correcting the problems. Software measurement provides the insight a Program Manager needs to make the software decisions critical to program success.

Why should a Program Manager measure software? Recent changes in the DoD acquisition process have emphasized the need for better software management tools and techniques. Emphasis on the use of Commercial Off the Shelf (COTS) and reusable software components, and the implementation of Open System Architectures (OSA), is changing the way software is acquired and how systems are developed. New technologies and new development processes require that the Program Manager have better, and more objective software information to help make the day to day decisions which guide the program. The Integrated Product Team (IPT) approach is an effective technique for managing large systems within the DoD. The IPT approach requires continuous and effective communications within the program team to determine the best solutions to identified problems. Software measurement provides the objective information which is essential for such communications.

Software measurement helps the DoD Program Manager do a better job. It helps to define and implement more realistic software plans, and then to accurately monitor progress against those plans. It provides the information required to make key program decisions and take appropriate action. Specifically, software measurement provides objective software information to help the Program Manager:

- **Communicate Effectively Throughout The Program Organization-** This is one of the key benefits of software measurement. Objective software information reduces the ambiguity which generally surrounds the software issues on a DoD program. Measurement allows the software issues to be explicitly identified, prioritized, and shared at all levels of the organization, particularly between the Program Manager and the developer.
- **Identify And Correct Problems Early** Rather than waiting for something bad to happen, measurement implements a pro-active software management strategy. As part of the day to day program management process, measurement focuses on the early discovery and correction of software technical and management problems which are more difficult to address later in the program. Measurement helps the Program Manager focus on the key software issues throughout the program life cycle.

- **Make the Key Tradeoffs** Every program is constrained to some degree. Development schedules, resources, and system capability requirements all have to be managed together to make the program a success. With software intensive programs, decisions in one area always have an impact on the others. Measurement allows the Program Manager to objectively assess these impacts, and make the proper tradeoff decisions to best meet program objectives.
- **Track to Specific Program Objectives** Measurement, better than any other software management tool, accurately describes the status the software processes and products. It objectively represents the progress of the software activities and the quality of the software products. It helps to answer key questions such as, Is the development on schedule? and Is the software ready to deliver?
- **Manage to an Optimal Solution** No program ever has enough time or resources to meet all of the technical requirements. These constraints are amplified when changes to funding profiles and delivery dates occur over the course of a program. Software measurement helps the Program Manager to address these constraints. By objectively relating all of the software technical and management characteristics, measurement can identify and manage to an optimized set of objectives.
- **Defend and Justify Decisions** -The current DoD acquisition environment emphasizes successful program performance. A decreasing tolerance for failing programs, coupled with the need to accurately evaluate the performance of all government initiatives, requires that the Program Manger be able to effectively defend and justify his decisions. Measurement helps to do this. It helps the Program Manager focus on the key software issues, and provides the data required to explain how the issues were prioritized and managed.

Like any program management tool, software measurement cannot guarantee that a program will be successful. It does, however, help the Program Manager take a pro-active approach in dealing with the inevitable issues that are part of any software intensive program. Even more importantly, measurement establishes a basis for objective communications within the program team. This is essential when decisions which materially impact the outcome of a

program have to be made quickly, and correctly. *Software measurement helps the Program Manger to succeed.*

1.2 OVERVIEW OF MEASUREMENT PROCESS

How does an organization that wants to take advantage of the benefits of software measurement proceed? A number of specific measurement prescriptions have been offered to government and industry organizations with limited success. Rather than propose another fixed measurement scheme, this guide presents a flexible measurement approach. *PSM* views measurement as a process that must be adapted to the technical and management characteristics of each program. This measurement process is issue-driven. That is, it provides information about the specific issues and objectives important to program success.



As shown in Figure 1-1, the *PSM* approach defines three basic measurement activities necessary to get measurement into practice. The first two activities, *tailoring* measures to address program needs and *applying* measures to obtain insight into program issues, are the basic subprocesses of the measurement process. The third activity, *implementing* measurement, consists of the steps necessary to establish this measurement process within an organization.

The *tailoring* activity addresses the selection of an effective and economical set of measures for the program (for details see Chapter 2) . The *application* activity involves collecting, analyzing and acting upon the data defined in the tailoring activity (for details see Chapter 3). *PSM* recommends that these two activities be performed by a measurement analyst who is independent of the software developer.

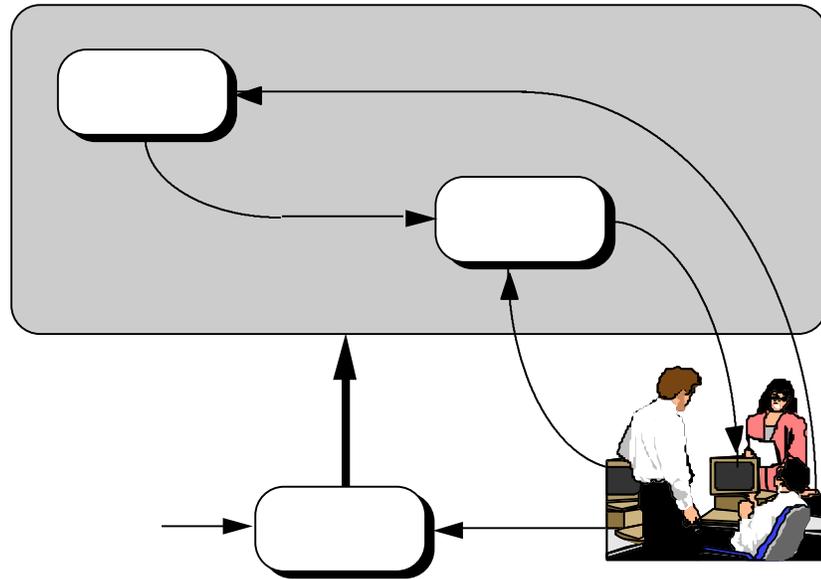


Figure 1-1. Basic Measurement Activities

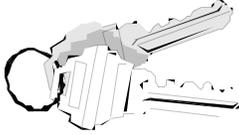
The *implementation* activity addresses the cultural and organizational changes necessary to establish a measurement process (for details see Chapter 4). Implementing a measurement process requires the support of program and organizational managers.

The measurement process must be integrated into the developer's software process. The nature of the developer's process determines the opportunities for measurement. Because the software process, itself, is dynamic - the measurement process also must change and adapt as the program evolves. This makes the activities of measurement tailoring and application iterative throughout the program life cycle. The issues, measures, and analysis techniques change over time.

1.3 SOFTWARE MEASUREMENT PRINCIPLES

Each program is described by different management and technical characteristics, and by a specific set of software issues. To address the unique measurement requirements of each program, *PSM* explains how to tailor and apply a generally defined software measurement process to meet specific program information needs. To help do this, *PSM* provides nine principles that define the

characteristics of an effective measurement process. These principles are based upon actual measurement experience on successful programs.



As shown in Figure 1-2, the software measurement principles are the foundation for the application of software measurement for:

- **Program Management** ensuring that products are delivered on time, within budget, and are of acceptable quality.
- **Product Engineering** ensuring that products satisfy customer needs.
- **Process Improvement** ensuring that the process becomes more capable over time.

The same measurement process can be used to provide the information needed for all three applications, employing different measures to address differences. However, the *Guide* only addresses program management practices, techniques, and tools.

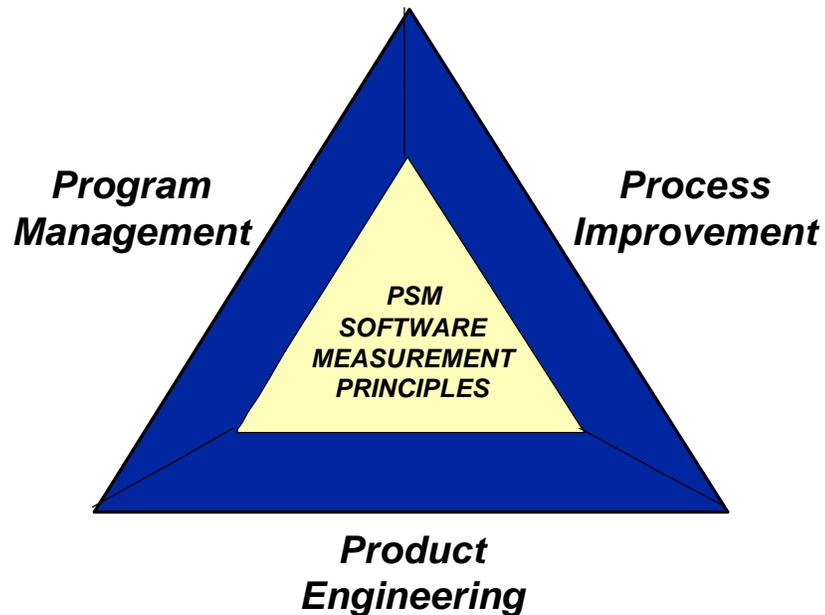


Figure 1-2. Scope of Software Measurement

The following subsections introduce each of the nine principles. Experience has shown that a measurement process that adheres to these principles is more likely to succeed. The following chapters explain in detail how these principles drive the tailoring, application, and implementation of the measurement process.

1.3.1 Program Issues and Objectives

Program issues and objectives drive the measurement requirements. The purpose of software measurement is to help management achieve program objectives, identify and track risks, satisfy constraints, and recognize problems early. These management concerns are referred to, collectively, as issues.

Note that issues are not necessarily problems, but rather they define areas where problems may occur. An initial set of issues are identified at the outset of the program. Thereafter, the issue set evolves and changes as the program progresses.

PSM emphasizes identifying program issues at the start of a program and then using the measurement process to provide insight to those issues. While a few issues are common to most or all programs, each program typically has some unique issues. Moreover, the priority of issues usually varies from program to program.

The six basic software issues addressed in this document are as follows:

- Schedule and Progress
- Resources and Cost
- Growth and Stability
- Product Quality
- Development Performance
- Technical Adequacy

At the start of a program, each of these issues is usually analyzed in terms of the *feasibility* of the plan. For example, the program manager may ask questions such as: Is this a reasonable size estimate? or, Can the software be completed with the proposed amount of effort and schedule? Once the program is underway, the manager's concern turns to *performance*. The key questions then

become ones such as: Is the program on schedule? or, Is the quality good enough?

It is important to note that software issues are not independent. For example, requirements growth may result in schedule delays or effort over-runs. Moreover, the impact of the addition of work to a program (size growth) may be masked, in terms of level of effort, by stretching out the schedule. Thus, it is important that issues be considered together, rather than individually, to get a true understanding of program status.

Focusing measurement attention on items that provide information about the program's issues also minimizes the effort required for the measurement process. Resources are not expended collecting data that may not get used.

1.3.2 Developer's Software Process

The developer's software process defines how the software is actually measured. The definition of a measurement process cannot be based solely on the objectives of the program manager. To collect measurement data in the most cost effective and useful manner, the measurement analyst must consider the software process of the developer. Program issues identify the information that the measurement process must derive from the data. The developer's software process determines what specific data items are to be collected and how that is to be accomplished.

One purpose of the measurement process is to provide insight into the performance of the developer. Thus, the measures collected must objectively represent the activities and products of the developer's software process. Consequently, the data better represents the software products and processes. Moreover, when choices are possible, the program manager should select measures that are normally collected by the software developer. This decision should also consider the software processes employed by any subcontractors.

1.3.3 Low Level Data

Collect and analyze low level data. The measurement process defined in *PSM* depends on the periodic collection, processing, and

analysis of measurement data rather than the review of pre-packaged reports. This data includes plans, changes to plans, and counts of actual activities, products, and expenditures. The program office should receive data from the developer at a low enough level of detail to allow for the isolation of problems (errors, delays, over-runs) by activity and component. This detail is commonly at the unit and software activity level as defined by the software architecture and work breakdown structure.

Indicators that address program issues are computed from measurement data collected by the developer. Most good software developers can provide a wide range of data items. The specific data items needed for program management depend on the program issues. When a proposed measure proves difficult to collect or doesn't provide the required information, an effective substitute may often be found by looking at related measures. Collecting low level data allows the measurement analyst to perform a variety of different analyses with the same data. It is a key requirement for defining a flexible measurement process.

1.3.4 Independent Analysis Capability

Implement an independent analysis capability. The program manager must have a measurement capability that is independent of the software developer's. This requirement is motivated by the recognition that communication can only occur when both parties have achieved an understanding of the data under discussion. The ideal situation involves a government measurement analyst in the program office who regularly receives low level raw data from the developer, analyzes it, and presents the results to the program manager. Alternatively, the independent analyst function may be provided by an Independent Verification and Validation (IV&V) contractor, matrix function, Systems Engineering and Technical Assistance contractor, or another organization independent of the developer.

Note that without an independent analysis capability, the delivery of low level data to the program office (see Section 1.3.3) has no value. Similarly, without low level data the ability of the measurement analyst to conduct an independent analysis will be seriously limited.

1.3.5 Structured Analysis Process

Use a structured analysis process to trace the measures to the decisions. Measurement-based conclusions and recommendations must be generated in a systematic manner to be accepted as a basis for management decision-making and action. Key concerns of management about such information is its traceability and repeatability. Traceability means that the conclusions and recommendations are generated from measurement data in a defined sequence of steps. Repeatability means that analysts following the same sequence of steps are likely to arrive at the same conclusions and recommendations. An ad-hoc analysis approach does not provide management with the confidence necessary to act on measurement information. For measurement to succeed, management must become an active participant in the measurement program and a regular consumer of measurement results.

This *Guide* describes a systematic method for using detailed data items to gain insight into high level issues. For example, a high level software issue is *schedule and progress*. A realistic question concerning this issue is whether the program is progressing on schedule. A complicating factor in assessing this issue is that a major program contains many different individual activities, where thousands of software units may be developed. Some of these units may be ahead of schedule and some behind. Hence, the overall status of the program is very difficult to determine without some systematic method for combining *quantitative data* from all these software units into information about progress.

A *measure* is a method of counting or otherwise quantifying some attribute of a software process or product. Measures alone do not provide much insight into issues. For example, two major deliverables of the typical program are software and documentation. Measuring the amount of software and documentation completed gives a sense that work is progressing; however, without comparing the work performed with the plan, we cannot tell whether the work is on schedule. Measures are used as *indicators* of software development and support status. These indicators provide insight into key program issues.

1.3.6 Results in Program Context

Interpret the measurement results in the context of other program information. Measurement provides an indication or warning that a problem may exist. No measurement result by itself is good or bad. For example, assume that the number of unit designs completed to date is lower than planned. This situation might occur because the program is not fully staffed, but while there is still time to add staff and recover. It might occur while the program is fully staffed because the developers' productivity is much lower than planned. The low score on a progress indicator does not necessarily mean that the program has a problem. However, it does signal that the program manager should pay attention to this issue *now*. He must collect additional information to evaluate the cause and severity of the situation to assess its probable impact on program success. Measurement results must be examined in the context of other information about the program to determine whether action is warranted, and what action to take.

Some aspects of, or contributors to, an issue may not easily be quantified. For example, getting the requirements right may depend on adequate interaction with the system's intended user. Even if production of the requirements document is on schedule it may not have the right content. Thus, *qualitative data* about the level of user interaction must be considered when assessing progress for this example.

1.3.7 Life Cycle Integration

Integrate software measurement into the program management process throughout the lifecycle. The issue-driven software measurement approach described in *Practical Software Measurement* applies throughout the software life cycle. For purposes of this document, we define three major phases: program planning, development, and software support. Four principal software activities occur within the development and software support phases. These are requirements analysis, design, implementation, and integration and testing. Measurement results must be provided at appropriate decision points throughout the life cycle.

Decisions made in one phase or activity affect the results of other phases and activities. Measurement provides insight into the current

phase, as well as helping to project the consequences of current actions into later phases. For example, the selection of a specific software developer during program planning affects the level of performance realized by the program during development and software support. Consequently, it is important to adopt a life cycle perspective when developing a measurement program. Over the course of the software life cycle, the issues of concern to program managers may change. However, the basic measurement principles still apply.

1.3.8 Objective Communications

Use the measurement process as a basis for objective communications. The measurement process cannot be conducted by the measurement analyst in isolation. At each step of defining the measurement requirements and analyzing the measurement data, the program manager and measurement analyst must communicate with the developer team. Most decisions that are based on the data will affect more than one party. A corrective action that is identified and planned in cooperation with the developer is more likely to succeed than one that is arbitrarily imposed by the program manager. Figure 1-3 shows the roles of measurement in communication.

The communication described in Figure 1-3 depends on measurement data that objectively represents the developer's software products and processes. The figure highlights the role of measurement in communication between the program and developer manager. Both the government and developer measurement analysts should be analyzing the same data for their respective managers. While there may be some differences between the issues of concern to the software program manager and the software developer, there should also be a high degree of commonality.

The concept of Integrated Product Design and Development (IPDD) and the functioning of an Integrated Product Team (IPT) depend on frequent and unambiguous communication about technical and management issues among all team members. Measurement provides an effective vehicle for this.

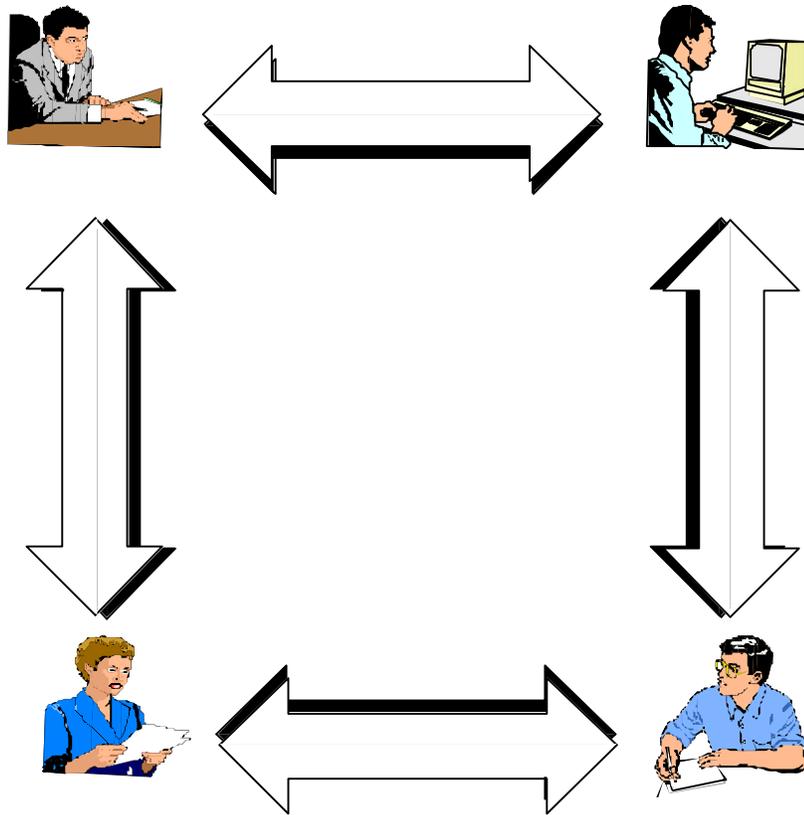


Figure 1-3. Role of Measurement in Communication

It is important to ensure that all parties use the same data and have a common understanding of the data definitions. Most data comes from the developer, so the burden is primarily on the program manager to understand the developer's software process and measures.

1.3.9 Single-Program Analysis

Focus initially on single program analysis. Program success means meeting program-specific objectives. While the larger organization (of which the program is a part) may have concerns and objectives that span multiple programs, this *Guide* stresses the need to measure and understand individual programs before attempting to make cross-program comparisons. The variety of measurement techniques and definitions used in current practice make more detailed cross-program comparisons difficult and time-consuming. The problems are compounded when programs from multiple organizations are involved. Nevertheless, at several steps in the

measurement process the analyst will need to refer to normative data and simple models based on multiple programs.

1.4 MEASUREMENT IMPLEMENTATION CONSIDERATIONS

Because the software measurement process is an integral part of the software process, many members of the organization play a role in it. Moreover, appropriate resources must be allocated in order for the measurement process to work effectively.

The most important roles in the software measurement process are the following:

- **Program Manager**- identifies issues, interprets and acts on measurement information. (The acquirer and developer may both have program managers.)
- **Measurement Analyst**- specifies measurement requirements, analyzes and reports results. (The acquirer and developer may both have measurement analysts.)
- **Developer Team**- may be a contractor or in-house developer, collects and packages data for the measurement analyst. (The *Guide* uses the term, developer, to refer to both developers and maintainers.)
- **Executive Manager**- has several program managers reporting, periodically reviews program status. (The acquirer and developer may both have executive managers.)

Figure 1-3 shows the relationships among measurement analysts and program managers.

Making sure that all participants in the measurement process understand and commit to their roles helps ensure that accurate data is provided that results in constructive communication among all parties.

Experience from a wide variety of commercial and government organizations shows that the cost of implementing and operating a measurement process of the type described in this *Guide* ranges from 1 to 5 percent of the program's software budget. This is a relatively small amount when compared to the cost of conventional review and documentation based program monitoring techniques.

CHAPTER 2 – TAILORING SOFTWARE MEASURES

The first activity of the software measurement process is to identify measurement requirements that address program issues. This activity includes identifying program issues, selecting and specifying measures, and integrating the measures into the software process. In some situations, the measurement requirements must be specified as contractual requirements through negotiation with the software developer. This *Guide* recommends an experience-based process for converting program issues into data requirements. This process and its component activities are discussed in this chapter.

2.1 MEASUREMENT TAILORING OVERVIEW

This section of the *Guide* explains how to determine measurement requirements and develop a program measurement plan. The objective of the measurement tailoring process is to define the set of measures that provides the greatest insight at the lowest cost. The tailoring process focuses effort and resources on getting the most important program information first. When implementing measurement on an existing program, give special consideration to existing data sources and measurement activities.

Measurement tailoring begins with issue identification. On an existing program the issues should be well understood. Program issues drive the entire measurement process. This includes the selection of measures to be collected, the analysis of measurement results, and management decision-making. Figure 2-1 illustrates the process recommended in this *Guide* for developing a measurement approach that addresses program issues. This process is a detailed definition of the measurement tailoring activity introduced in Figure 1-1.

The first step is to define the program-specific issues to be tracked. These are derived by combining a set of common software issues with program-specific issue criteria. The basic concern in this step is identifying the issues into which measurement can provide insight.

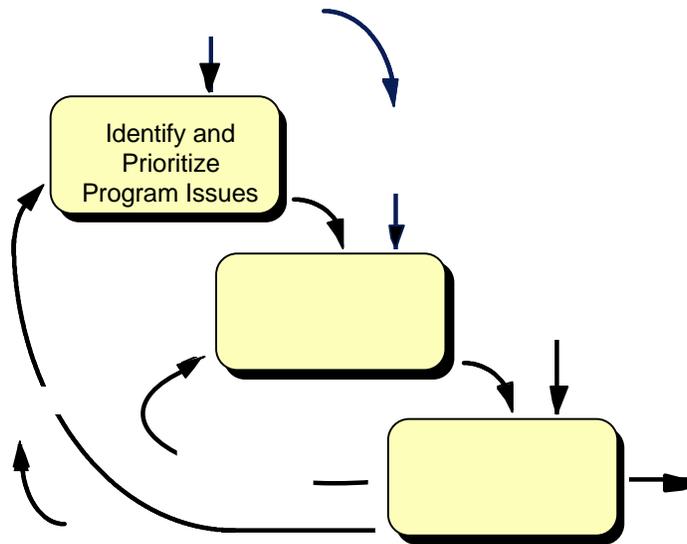


Figure 2-1. Measurement Tailoring Process

The next step is to define program-specific measures. The measures are selected by applying measurement selection criteria to measurement sets that map back to the common software issues. The basic concern in this step is finding measures appropriate to the issues.

The basic concern in the final step is integrating the measures into the developer's process. The software environment, development approach, and existing measurement mechanisms, if any, should be considered for their applicability to program needs. The results of this step are documented in a software development plan. The plan may be formal or informal.

The following sections explain each of these steps in more detail.

2.2 IDENTIFY AND PRIORITIZE PROGRAM ISSUES

An effective measurement process helps the program manager to be successful. It provides information the program manager can act on. This means that measurement must provide information pertinent to the achievement of program objectives. An objective is

a statement about the cost, schedule, functionality, quality, or performance that a program must achieve. Objectives may be directed downward by executive management or defined by the project manager in consultation with the prospective system user. Issues are current or potential problem areas that might impact the achievement of program objectives. Objectives and issues vary from program to program.

2.2.1 Program-Specific Issues

The basic concept underlying the *Practical Software Measurement* approach is that measures should be selected and organized to track program-specific issues. An issue is anything that might affect the achievement of program objectives. Issues include risks, constraints, and any other concerns. Some examples of issues are growth and stability, schedule and progress, and product quality. Over-runs or short-falls in these areas usually affect program success. Aggressive or unrealistic organizational goals might also be treated as program issues.



Specific issues can be defined at the outset of the program. These may be identified by considering the basic issues described in this *Guide*, as the result of a risk analysis, by relying on past experience, or by examination of executive management objectives.

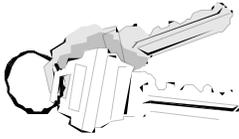
Issues may arise during program performance. New or evolving requirements, changes in technology, and other factors may result in the identification of derived issues as the program progresses.

Identifying something as an issue does not mean that it is a problem. An issue is something that might become a problem. Issues are identified in anticipation of problems, not just after a problem has occurred. The *Practical Software Measurement* approach emphasizes prevention and early detection of problems rather than waiting for problems to become critical.

2.2.2 Common Software Issues

Experience shows that some issues are basic or common to almost all programs. If you are not tracking these common issues, then you probably are not managing all of your program risks. The six basic software issues are as follows:

- **Schedule and Progress**- this issue relates to the completion of major milestones and individual work units. A program that falls behind schedule can usually only make it up by eliminating functionality or sacrificing quality.
- **Resources and Cost**- this issue relates to the balance between the work to be performed and personnel resources assigned to the project. A project that exceeds the budgeted effort usually can recover only by reducing software functionality or sacrificing quality.
- **Growth and Stability**- this issue relates to the stability of the functionality or capability required of the software. It also relates to the volume of software delivered to provide the required capability. Stability includes changes in scope or quantity. An increase in software size usually requires increasing the applied resources or extending the program schedule.
- **Product Quality**- this issue relates to the ability of the delivered product to support the user's needs without failure. Once a poor quality product is delivered, the burden of making it work usually falls on the maintainers.
- **Software Development Performance** this issue relates to the capability of the developer relative to program needs. A developer with a poor software development process or low productivity may have difficulty meeting an aggressive schedule and cost plan.
- **Technical Adequacy**- this issue relates to the viability of the proposed technical approach. It includes features such as software reuse, use of COTS software and components, and reliance on advanced software development processes. Cost increases and schedule delays may result if key elements of the proposed technical approach are not achieved.



PSM recommends using the six common issues in two ways. First, reviewing the common issues helps the program manager and measurement analyst to recognize related issues specific to their program. Second, the common issues are used to classify program-specific issues so that they can be mapped into the measurement selection structure discussed in Section 2.3.

2.2.3 Identifying Program Issues

The common software issues are a good starting point for identifying program-specific issues. While the common issues apply to all programs, their priority and exact wording are likely to be specific to each program. For example, a program that plans to make extensive use of COTS software may be more concerned with the schedule and progress of COTS software integration than with the quality of the COTS software (presuming that the COTS software was selected based on an evaluation that showed that it met user requirements.) On the other hand, a safety-critical system might have quality at the top of its priority list.

In addition to reviewing the six common software issues, the measurement analyst and program manager should consider other sources of information about potential program problem areas. Useful sources of information to consider when identifying issues include the following:

- **Risk Analysis**- the results of technical and management risk analyses should be considered in identifying program-specific issues. Risk analyses may point to potential requirements, technology, process, cost, and schedule issues.
- **Program Constraints and Assumptions** the program plan is based on assumptions about the performance of the software developer, availability of facilities, etc. Moreover, schedules and budgets may have inflexible constraints. If deviations from these assumptions and constraints could

threaten program success, then these areas should be identified as issues.

- **Leveraged Software Technologies** the program plan may depend on obtaining the benefits of a leveraging software technology such as reuse, COTS, or advanced programming languages. If program success depends on obtaining these benefits, then the effectiveness of this technology should be identified as an issue.
- **Product Acceptance Criteria** the user may impose stringent milestone or final acceptance criteria on the system to be delivered. If there is significant doubt about the system's capability to meet acceptance criteria, advertised objectives, or other external criteria, then satisfaction of these criteria should be identified as an issue.
- **Experience**- the manager's experience with similar past projects may suggest potential problem areas that should be identified as issues.

These sources of information, together with the common issues, help to identify program-specific issues. Each program issue should be stated in terms that are appropriate for that specific program. Focus on those aspects of the issue that are most important to the program. In the earlier example in this section, the schedule and progress issue was stated in terms of COTS software integration progress instead of design progress.

2.2.4 Prioritizing Program Issues

Programs may have many issues. Not all issues are equally important. Issues must be prioritized to determine where to focus the measurement effort. In general, more data should be collected and analyzed for important issues than for less important ones.

One useful way to prioritize issues is to classify them according to the following: 1) how likely that issue is to result in a problem, and 2) how much impact a problem in this area is likely to have on program success. Three categories of issues, in decreasing order of importance, are as follows:

- **Primary Issues**- likely to be problem areas, and the related problems are likely to have major impacts.
- **Secondary Issues**- likely to be problem areas, or the related problems are likely to have major impacts, but not both.
- **Peripheral Issues**- not likely to be problem areas, and the related problems are not likely to have major impacts.

Most programs cannot afford to track peripheral issues. Of course, this rating is subjective, so there may be a temptation to reduce measurement requirements by down-grading the priority of an issue. That temptation must be guarded against.

As an example, if the software budget is known to be a constraint from the outset of a program (the probability of a problem is high), and reducing functionality to fit available resources is not an option (the problem is likely to have serious consequences) then resources and cost are almost certain to be *primary* issue.

Issues (and their priorities) are dynamic. Additional issues may be identified once the program is underway. Also, things that were originally thought to be issues may be recognized as unimportant. Issues evolve as program concerns evolve. Thus, the measurement process has to change to keep pace. When defining a new or derived issue, remember to consider the probability of a problem arising and its likely impact before deciding to collect any additional data or regularly tracking the issue.

2.3 SELECT AND SPECIFY PROGRAM MEASURES

Once the program-specific issues have been identified and prioritized, appropriate measures must be selected to track them. Many different measures may apply to an issue. However, in most cases it is not practical to collect all or even most of the possible measures for an issue. Generally, more measures should be collected to track primary (high-priority) issues. Identification of the "best" set of measures for a program depends on a systematic evaluation of the potential measures with respect to the issues and relevant program characteristics.

For example, if *growth and stability* is selected as an issue, then requirements and software size measures will be needed to track it. The appropriate measure will depend on the nature of the program. For example, the language type and application domain influence the choice of size measure. Automated Information Systems may use function points to measure size. Weapons Systems are likely to find lines of code to be more useful.

PSM provides a three-part measurement selection and specification mechanism. First, issues are reviewed to identify the applicable measurement categories. Next, measures within the categories are reviewed for applicability. Finally, the data items and implementation requirements are specified for the selected measures.

2.3.1 Measurement Category Selection

A measurement category is a set of related measures. The measures within a category are derived similarly or address related software attributes. They provide similar information and answer similar questions. Figure 2-2 shows the types of questions to which each measurement category responds.

Use this table (or the corresponding detailed tables in Part 2) to find the measurement category (or categories) that most closely aligns with the formulation of the program-specific issue. For example, if the program-specific issue is "Progress of COTS software Integration", then the Work Unit Progress category is suggested because the issue involves a question about the progress of a specific activity (i.e., integration). If the program-specific issue was "Availability of Qualified Staff", then the Staff Profile category is suggested because our issue concerns not just the number of staff, but also their skills.

2.3.2 Measurement Selection Criteria

Once a measurement category has been selected, then the measurement selection criteria discussed below can be applied to identify the best measures for this program from among those in the indicated measurement category.

| Issue | Measurement Category | Questions Addressed |
|--------------------------------|--------------------------------------|---|
| Schedule and Progress | Milestone Performance | Is the program meeting scheduled milestones? |
| | Work Unit Progress | How are specific activities progressing? |
| | Schedule Performance | Is program spending meeting schedule goals? |
| | Incremental Capability | Is capability being delivered as scheduled? |
| Resources and Cost | Effort Allocation | Is effort being expended according to plan? |
| | Staff Profile | Are staff assigned according to plan? |
| | Cost Performance | Is program spending meeting budget goals? |
| | Environmental Availability | Are necessary facilities and equipment available as planned? |
| Growth and Stability | Product Size and Stability | Are the product size and content changing? |
| | Functional Size and Stability | Are the functionality and requirements changing? |
| | Target Computer Resource Utilization | Is the target computer system adequate? |
| Product Quality | Defect Profile | Is the software good enough for delivery to the user? |
| | Complexity | Is the software testable and maintainable? |
| Development Performance | Process Maturity | Will the developer be able to meet budgets and schedules? |
| | Productivity | Is the developer efficient enough to meet current commitments? |
| | Rework | How much breakage due to changes and errors has to be handled? |
| Technical Adequacy | Technology Impacts | Is the planned impact of the leveraged technology being realized? |

Figure 2-2. Questions Addressed by Categories

Some of the key criteria to consider when selecting measures include the following:

- **Measurement Effectiveness** how effective is the measure in providing the desired insight? Does the measure provide insight that relates to more than one issue? How difficult and effective have these measures been on past projects?
- **Domain Characteristics** are certain measures more likely to be used in a given domain? For example, response time is widely used to measure target computer resource utilization in AIS systems, while memory utilization is more widely used in weapons systems.
- **Program Management Practices** can existing management practices be leveraged to support measurement requirements? For example, is a scheduling system in use that provides one or more of the desired measures?
- **Cost and Availability**- what data should be readily available in this program context? How much effort will be required to extract and package the data for analysis? Extracting data from electronic sources usually costs less than manual collection.
- **Life Cycle Coverage**- does the measure apply to the life cycle phase under consideration? Does it apply to multiple life cycle phases?
- **External Requirements** has the overall organization or oversight authority imposed any related measurement requirements?
- **Size/Origin of Software**- does the size or scope of the software justify a larger investment in measurement? Does this measure make sense for this type of software (e.g., COTS)?

The tables in Part 2 provide an assessment of 45 different measures with respect to these criteria. This assessment is based on actual experience in applying measurement to large programs.

2.3.3 Specifying Data and Implementation Requirements

Once the measures have been selected, the appropriate level of detail for data collection for those measures must be decided upon.

The frequency and format of data deliveries must also be specified. Data may be reported less often than it is collected by the developer. Monthly reporting is common. The tables in Part 2 provide typical implementation requirements for common measures.

Potential candidates for measurement include all the products to be delivered by the program and all the processes used by the developer. Of course, these can be defined and measured at many different levels of detail (see Figure 2-3). However, unless these definitions and measures are coordinated appropriately, the measurement program may not produce meaningful results.



The program's work breakdown structure (WBS) provides a simple mechanism for defining and integrating the software activities (Figure 2-3 shows a typical hierarchy of software activities and components) and components to be measured. The WBS identifies all the hardware, software, data, and other products and services that must be delivered to complete the system. (See Part 4 for sample WBSs for AIS and weapons systems.) The WBS is used to break a project down into small tasks. Each of the small tasks is called a work package. Typically, each work package has a schedule, effort allocation, and quantity of work associated with it. Errors and rework may be by products of implementing a work package.

A work package could correspond to something as large as developing an entire Computer Software Configuration Item (CSCI) over a period of years or as small as testing a single unit within one week.

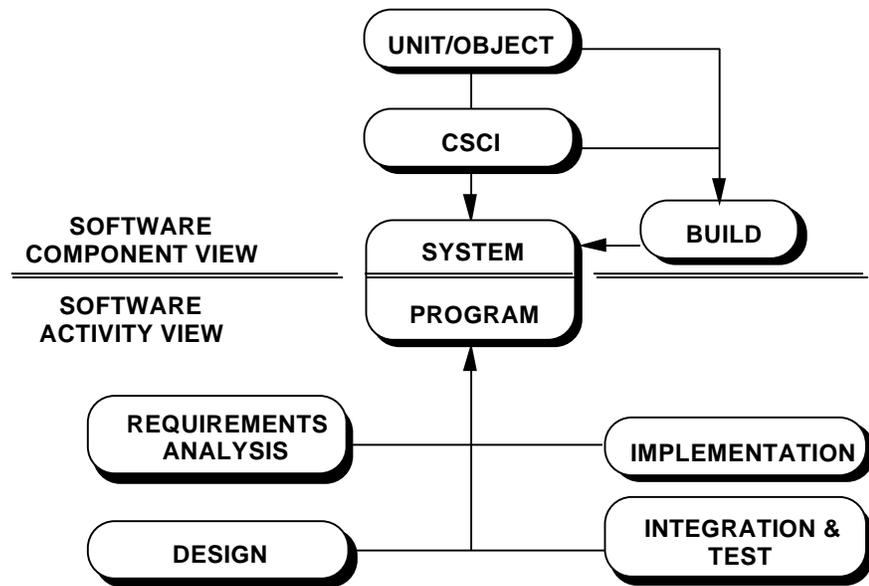


Figure 2-3. Activity and Component Aggregation

Most programs define work packages for each major activity (i.e., requirements analysis, design, implementation, integration and testing, and rework) for each CSCI. However, to adequately address specific program issues it may be necessary to collect one or more types of data at a more detailed level. Some of the factors that help define the appropriate level of data collection are as follows.

- Requirements and size data are normally tracked at least at the CSCI level. Consider tracking size at a lower level if the CSCIs are large.
- Progress is normally reported at the level of major activity (e.g., design). Consider tracking at the level of subactivities if the schedule is a long one.
- Keep data from subcontractors separate, especially if the subcontractors have significant software development responsibility, or a different development process.
- Maintain separate counts of size for each language type, including 4GLs and application generators, unless the languages are very comparable (for example, Fortran and Algol).
- Maintain separate counts of size, effort, and problem reports for each category of new development, reuse, and COTS software, especially if program success depends on realizing some specific benefit from these approaches.

- Keep separate counts for each priority category of problem report, especially if the program maintains a large backlog of problems.

All data collected must be consistent with the WBS. Different types of data may be collected at different levels of detail, but each must roll up into the same product elements. For example, it is hard to analyze productivity when effort data is collected using categories that do not map into the work packages against which size is measured. When defining a measurement program, the ability of the developer's cost accounting system to flexibly support detailed effort and cost reporting are important considerations.

In determining the proper level of detail for the measurement data to be collected, the measurement analyst must balance the cost of data collection, data processing, and analysis against the need for detailed insight into program issues. More detailed data allows greater flexibility for analysis in terms of defining new indicators and localizing the source of potential problems detected with the data. However, a greater level of detail also implies a greater volume of data and a greater cost to the measurement program. Nevertheless, more detailed data should be sought to track those issues defined to be most important. All of these recommendations for selection of measures and level of detail must be tempered with an understanding of the developer's process.

2.4 INTEGRATE MEASURES INTO THE DEVELOPER'S PROCESS

Up to this point the measurement selection process has largely been driven by "what" we need to know as defined by the issues. Now we need to look at "how" the measurement process will actually function with the program structure. The data readily available from the developer may not map exactly into our ideal measurement requirements as defined thus far.

The measures and implementation requirements selected in the preceding step form the basis for negotiations between the program manager and the developer about the specific data elements to be provided for analysis. This negotiation may be accomplished via a formal contracting process, or via a less formal agreement. The result of this step is a definitive statement of the measurement approach to be followed, often documented in an informal

measurement plan, or incorporated into the program management plan.

Adjusting the Program Manager's measurement requirements to the developer's process involves three tasks:

- Characterizing the software environment
- Identifying measurement opportunities
- Developing a software measurement plan

During the course of performing these tasks, the developer may propose changes to the program measurement requirements to better integrate the measures into the software process. The final plan is based on both the initial requirements and agreed-upon changes.

Part 4 of the *Guide* provides sample contract wording that helps implement these steps. A "contract" may be a formal contract, a Memorandum of Agreement (MOA), an Inter-Service Support Agreement (ISSA), or some other written agreement. The technical concepts discussed in this *Guide* are applicable to whichever type of contract is used.

The tasks required to integrate the measurement requirements into the software process are discussed below.

2.4.1 Characterizing the Software Environment

The developer's process has a major impact on the cost and effectiveness of a software measurement program. *One basic purpose of the measurement program is to provide insight into the developer's process.* Thus, it is important that the measures accurately represent the software process being used and the products being built. Some key factors to consider are as follows:

- The life cycle model or activity structure used to define the developer's process
- Product structure, including builds and releases defined by the developer
- Current measurement activities employed by the developer
- Software technology, including programming language, design language, etc.

- Planned source of software (COTS, GOTS, reuse, etc.)
- Management, review, testing, and inspection practices employed by the developer
- Engineering and management standards to be applied

Whenever possible, take advantage of the developer's current practices and existing data collection mechanisms. Avoid imposing new measurement requirements. Use the program's WBS, including product structure and activities, as the basis for measurement.



To the extent that the activities of the developer's process are well-defined, measuring them will provide useful information. An ad-hoc or ill-defined process makes it difficult to tell exactly what is being measured.

For many issues, the data available changes across life cycle activities. For example, during the software implementation stage, progress may be measured in terms of units designed and coded. During testing, progress may be measured in terms of tests attempted and tests passed. The measurement analyst must ensure that relevant measures and indicators are provided throughout the program's life cycle, making substitutions as appropriate.

Before measurement requirements are finally negotiated, the measurement analyst should use his or her understanding of the developer's process, as well as direct feedback from the developer to modify the target measurement set. Giving appropriate consideration to the developer's process helps to ensure that useful data is provided with the lowest impact and cost.

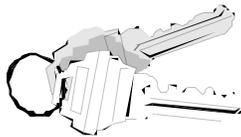
2.4.2 Identifying Measurement Opportunities

During measurement planning a high priority should be given to finding and taking advantage of any measurement mechanisms already operating within the development organization. *This is especially important when installing measurement on an existing program.* Give special attention to databases and tools supporting the following functions:

- management/scheduling
- financial/earned value/timecard
- planning/estimating
- configuration management
- problem tracking
- action item tracking
- inspection results
- development tools (e.g. CASE)
- measurement database

Extracting and delivering data from electronic sources such as these is usually more cost effective than manual (or paper forms-based) collection methods.

Most actual software data will come from the developer. However, initial planning data often is produced by the program management office. The source of the data will affect choices about the frequency and form of delivery.



As a result of characterizing the program environment and identifying measurement opportunities, changes to the measurement requirements previously defined may be proposed. Moreover, new issues may be identified that result in changes at the issue level as well. Thus, the measurement selection process is iterative. This iteration may be managed via a formal contracting process, a less formal agreement mechanism, or internal policy.

2.4.3 Developing a Software Measurement Plan

The final task in measurement selection is to develop a software measurement plan. The software measurement plan may be formal or informal. A formal plan may be produced as a separate document, but is commonly incorporated into the program's software management, development, or maintenance plan. Some elements of the plan may be specified in the Computer Life Cycle Management Plan. Part 4 provides additional detail on preparing a

formal plan as part of a contracting process and a sample measurement plan outline.

The plan is the result of adjusting the program manager's requirements to fit the developer's process.

Regardless of the formality of the measurement plan, it should incorporate the following information:

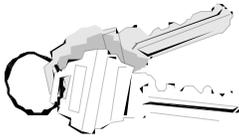
- **Issues and measures** selected.
- **Identification of data elements** consolidate the data elements required for all measures into a single list. Measures frequently share data elements.
- **Data definitions**- provide a complete and unambiguous definition of each data item. The checklists contained in the SEI Core measures may be helpful in this regard.
- **Data sources**- identify the specific sources (e.g., person, tool, report, activity) for all data items.
- **Level of measurement** determine the level of detail at which data items are to be collected and delivered for analysis.
- **Aggregation structure** define the hierarchy by which the low level data items will be combined to provide system, build, and program-level views. This structure should parallel the WBS.
- **Frequency of collection** specify how frequently data items are to be collected and delivered for analysis. Include plans and replans as well as actual data. This is typically monthly.
- **Method of delivery**- define the method for providing access to the data (e.g., common database, electronic media).
- **Communication and interfaces** identify the points of contact for all data sources, reports, and requests for clarification.
- **Frequency of analysis and reporting** determine the reviews and reports via which measurement results will be provided to the program. These should occur on a monthly or quarterly basis.

Most large programs will require the development of a unique software measurement plan. However, some organizations may be able to define a software measurement plan that covers many projects. This implies that a common measurement set can be

defined for the organization. A common measurement set only makes sense for programs that share the following traits:

- similar software issues
- common process (standards, practices)
- stable technology (languages, tools, platforms)
- similar application domains

Imposing a standard measurement set in situations where these conditions are not satisfied may burden the program with unnecessary measurement requirements while missing important issues that should be tracked.



A common data set or normalization scheme may be necessary for other types of analysis to support process improvement and business purposes. However, this *Guide* focuses on single program analysis. Recording the characteristics which drive decisions in the measurement selection process is important for figuring out how to normalize data for these purposes later.

In addition to the program-specific measurement needs discussed in this *Guide*, other users may have other valid needs that the program's measurement process must address. These other users include executive managers performing an oversight function and software engineering process groups working on process improvement issues. Most of the data needed by these other users originates from the program. Getting good data for executive review and process improvement depends on establishing an effective program-level measurement program.

Consider measurement requirements from all sources together when developing a program's measurement plan. This will enable the measurement analyst to minimize the redundancy and inefficiency that can result from multiple data collection efforts. Focusing on measures and analyses that benefit multiple users helps to maximize the value of the measurement process implemented.

CHAPTER 3 – APPLYING SOFTWARE MEASURES

This chapter explains how the measurement plan that results from the tailoring process described in Chapter 2 is applied during the program planning, development, and software support phases of the program life cycle. This chapter discusses the collection of the data, generation of indicators and reports, analysis of results, and the use of measurement information to support program management decisions and actions. Management support and participation throughout these activities are essential to the success of a measurement program. This chapter also describes how the focus of analysis changes across the program life cycle.

Figure 3-1 shows the major steps by which data is collected and converted into information which provides a basis for action by the program manager. This figure expands upon the measurement application subprocess defined in Figure 1-1. During measurement application (Figure 3-1) the specified measures are collected and analyzed to provide feedback on the issues. During this activity, questions may be raised and new issues may be identified, causing the process to iterate.

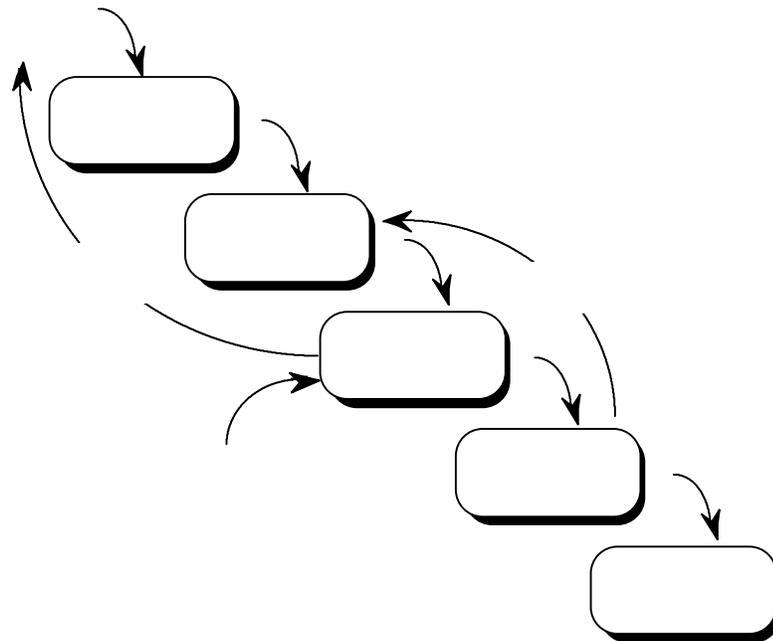


Figure 3-1. Measurement Application Process

3.1 COLLECT AND PROCESS DATA

Collecting and understanding the data is the first step towards analyzing program issues. Getting good data is the foundation of any measurement program. Almost all data originates with the software developer, including planned, actual, and historical data. Some of the concerns associated with data collection are the sources of data, reporting frequency and format, normalization and aggregation, and verification.



As explained in Chapter 2, the data collected should reflect the nature of the software product and the developer's process. Be sure to include all contractors and subcontractors in the data collection effort. More mature developers are likely to be able to provide more types of data at greater levels of detail than less mature developers.

3.1.1 Data Sources

Software data comes from many sources. The program's software development plan is a primary source. This plan typically contains the budgets and schedules against which progress and expenditures will be compared. Data must be collected from both initial plans and later replans (including incremental changes to plans). As the program evolves, the corresponding actual data on problems, progress, size, and effort will become available.

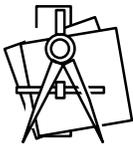
Many sources of data lie within the developer's process. Software problem counts and severities can be obtained from problem databases, of properly structured. Counts of hours expended by activity can be obtained from financial management records. (The capability of the developer's financial management system may limit the measurement analyst's ability to get detailed effort data.) Progress data usually comes from the detailed work plans maintained by technical managers and team leaders. Use of a project management tool facilitates data collection.

Counts of software units, lines of code, and changes to software and documents usually can be obtained from configuration management records and reports. Alternatively, a source code analyzer may be used. Product information, such as counts of lines of code or pages, can also easily be captured by recording them during inspections. Note that in all these cases, the most efficient method of collecting the desired data depends on the nature of the software developer's process.

3.1.2 Reporting and Processing

Data may be collected by the developer more frequently than it is reported to the program manager. The most common reporting intervals are monthly during requirements analysis, design, and implementation, then weekly during integration and testing. Integration and test data typically is reported more frequently because this period is relatively shorter. In any case, the reporting interval should not be longer than quarterly. Data reported less frequently is stale, and the opportunity for action has often passed by. Generally, analysis should occur soon after each delivery.

The developer should supply low level data directly to the program manager. Data should be provided in both hard copy and electronic form. When developing the schedule by which the developer provides data to the program manager, remember to allow adequate time for analysis between data delivery and the date the analysis results are required. The lag between data collection and reporting should be kept to a month or less.



One approach that helps assure timely provision of low-level data is to provide the measurement analyst with on-line access to the developer's software engineering database, if that database contains the necessary information.

3.1.3 Normalization and Aggregation

During data analysis, it may be helpful to combine or compare measures from different activities or CSCIs implemented in different languages. Normalizing data requires the definition of conversion rules or models. For example, to compare the productivity of different developers, it may be necessary to use a model that takes into account the effect of program schedule and size on productivity. Normalization has to be performed carefully. Any rules or models used must be validated with historical data.

The measurement analyst does not want to report all data to the program manager at the detailed level at which it is received. Consequently, it is often necessary to combine raw data from low level components into higher levels. Aggregating data requires the definition of the relationships among the measured objects, such as is provided by a WBS. For effective communication to occur, both the developer and program manager must understand and use the same aggregation and normalization rules.

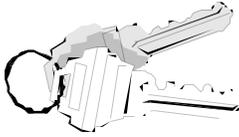
3.1.4 Data Verification

Getting useful measurement results depends on feeding good data into the analysis and reporting process. Data verification must consider both the accuracy of the data as it is recorded, as well as the fidelity with which it is transmitted. All data should be identified with its date of collection and source. Such identification helps to line up data with program events. Apply configuration management procedures (such as versions and dates) to electronic deliveries of datasets. This audit trail should be tested periodically to assess the integrity of the data collection process.

Once the data has been processed into a database or other storage medium, additional checks should be made. Compare a sample of values from the input data with the database contents. Previous values that are not expected to change should also be compared with current values. Additional checks may be automated. These include type checks, range checks, and completeness checks.

Developing and disseminating clear definitions of the desired data items helps ensure consistent data. Even seemingly obvious terms, like lines of code and staff-months of effort, need to be defined. For example, lines of code may be interpreted to mean all physical lines,

only non-comment lines, executable statements, or one of dozens of other variations. Even staff-months is ambiguous. The average number of hours worked per month varies from organization to organization. The categories of labor reported may also differ.



Data verification is complicated by the fact that some of the assumptions underlying the measurement program can change during the program life. Aggregation structures, product components, processes, and even definitions of measures may be updated as the program evolves. Sometimes, estimates and actuals are measured differently. Consider these possibilities during the data verification step.

You should be aware that even valid software engineering data is likely to be "noisy". Software engineering is a human-intensive activity; things seldom go exactly as planned. Because performance varies from week to week, you should be wary of "actual" data that exactly matches the "plan".

Any concerns about, or inconsistencies in, the data should be resolved via communication with the developer. Missing data, large changes in values, changes in the data structure should always be discussed to ensure that the measurement analyst understands the data.

3.2 DEFINE AND GENERATE INDICATORS

The next step after collecting and verifying the data is to define and generate the indicators that are the basis for analysis, reporting, and action. An indicator is a measure of combination of measures that provides insight into a software issue or concept. Most of the indicators discussed in *PSM* compare two measures, usually planned values versus actual values. The relationship between the two measures often can best be communicated graphically.

The measurement approach advocated in this *Guide* stresses the collection of low level data from which many different indicators

can be constructed. Such an approach allows a greater flexibility in analyzing issues and adapting to new issues as they arise. A measurement process that is based on the periodic delivery of only pre-defined graphs and tables does not have this flexibility.

Note that while some measures are closely associated with specific indicators, the *PSM* concept of an indicator helps the analyst to combine measures in many different ways.

Three sets of indicators typically are produced for each analysis cycle:

- Pre-defined indicators that are produced for every analysis cycle.
- Variations of the pre-defined indicators that provide additional detail to help localize problems.
- New indicators that respond to questions raised by the program manager or software measurement analyst during the current analysis cycle.

Most data necessary for producing these indicators should be supplied by the software developer. Most database and spreadsheet tools have the capability necessary to produce graphs like those discussed in this *Guide*. Usually, only summary reports are provided to the program manager on a regular basis, but detailed analyses must be studied by the measurement analyst and be available for discussion with the program manager if needed.

Good graphic displays of indicators facilitate communication of measurement results. Hence, graphs must not be too complex. Each graph should convey a clear message. It is better to have many graphs than many messages on one graph, especially when getting started. Part 3 of the *Guide* provides guidelines for developing effective graphs.

3.2.1 Basic Indicator Concepts

Issues usually cannot be measured directly. It is difficult to find a single measure that captures everything important about an issue. Thus, we must rely on (usually multiple) indicators. The indicators discussed in *PSM* are analysis tools, often represented as a graph or a table, that give insight to a particular issue. Indicators give

warnings of problems associated with issues. An important issue may be tracked with several indicators.

In most cases, insight into an issue cannot be obtained simply by collecting current data. That data must be compared with some notion or expectation of what the current data should be. That expectation may not be explicitly stated prior to the start of the measurement process. It may be a rule of thumb such as, "error rates usually go down as testing progresses". Since in the real world we seldom get exactly what we expect, we also need criteria to decide whether or not the difference between actual data and our expectation is sufficiently different to cause concern. The measurement indicators used in PSM generally consist of three parts:

- **Actual value of a measure or combination of measures**- actual current data such as hours of effort expended or lines of code produced to date.
- **Expected value of a measure or combination of measures**- planned value, quantitative objective, baseline, or historical value such as planned milestone dates, target level of reliability or required productivity.
- **Significance criteria**- rules of thumb and statistical techniques used to assess the difference (often called *variance*) between planned (expected) and actual (measured) values

Data provided by the developer will include planned and historical values, as well as corresponding measured actual values. During program execution, new indicators can often be defined by organizing collected measures in different ways.

Indicators can be used for predictive as well as for assessment purposes. Thus, a given indicator may be regarded from two points of view, based on how it is used:

- **Leading Indicators**- predict the future situation with respect to an issue. For example, requirement changes may be a leading indicator for developer effort. Changes in requirements usually result in a need for increased effort.
- **Current Indicators**- describe the current situation with respect to an issue. For example, staffing level describes the developer effort currently being expended by the program.

Note that the use of an indicator as leading or current is with respect to a specific issue. A given indicator may be current with respect to one issue and leading with respect to another issue. In order to define a leading indicator, the relationship between the activities or products measured by the leading indicator and those measured by current indicators must be understood.



Figure 3-2 illustrates the cascading relationship of typical software problems. Requirements changes drive increases in size. The increased size requires additional effort. The additional effort leads to schedule delays. Schedule pressure can cause a product to be delivered that is not fully tested and has documented problems that have not been corrected. These problems represent rework that requires additional effort in future releases or during maintenance.

These problems correspond to issues for which indicators can be defined. The earlier the situation described in Figure 3-2 is recognized and addressed, the greater the chances of program success. Thus, a program for which schedule was identified as a primary issue might also benefit from collecting effort and size measures as potential leading indicators of schedule.

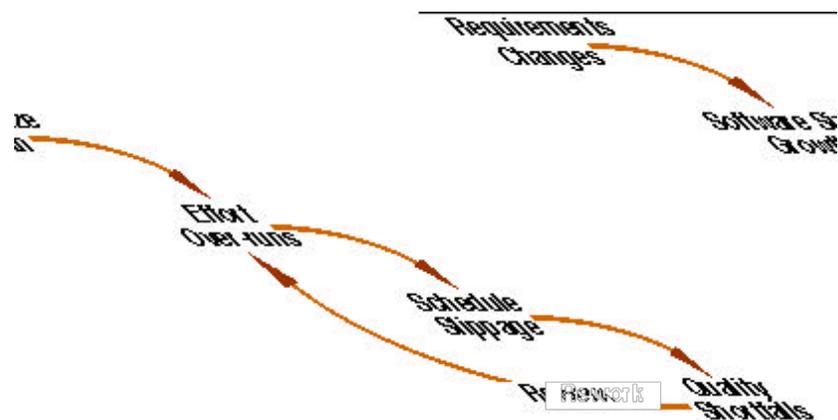


Figure 3-2. Typical Pattern of Software Development Problems

During analysis, each issue should be considered from two perspectives, feasibility and performance. **Feasibility** deals with the

accuracy and realism of plans, estimates, or assumptions associated with an issue. For example, an assessment of the feasibility of *funding and personnel resources* for a program must consider whether the proposed work can be accomplished with the proposed resources. **Performance** deals with adherence to plans, estimates, and assumptions associated with an issue. For example, an assessment of the funding and resources performance of a program must consider whether expenditures, such as personnel effort, conform to plan.

3.2.2 Types of Indicators

Indicators may also be classified into two general types, trend-based and limit-based, in terms of how they are graphed and analyzed. The primary distinction between the two is whether the expectation (target or plan) is relatively constant or changes over time. Both types of indicators should be constructed using raw data rather than percentages. Percentages are easily manipulated. The following subsections explain these indicator types in more detail.

3.2.2.1 Trend-Based Indicators

Trend-based indicators are used when the expected or planned value changes regularly over time. **Feasibility analysis** of a trend-based indicator involves determining whether the rate of work or other performance implied in the trend is actually achievable. **Performance analysis** consists of determining whether the actual program trend corresponds to the planned trend. Figure 3-3 shows an example of a trend-based indicator. In this example, a different goal or target for software *work units completed* has been set for each week. This is the program's implementation plan. Actuals are plotted on the same figure. The example of Figure 3-3 also shows an update to the plan due to size growth, an increase in the number of work items.

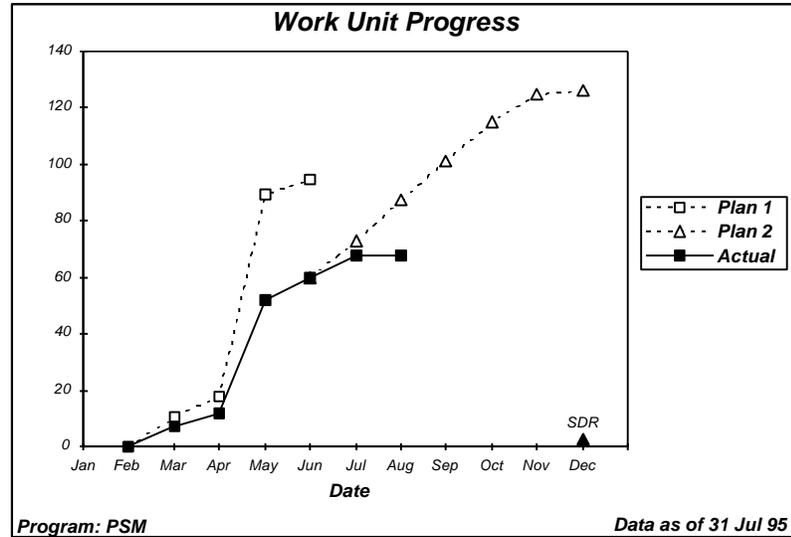


Figure 3-3. Trend Based Indicator Example

In addition to planned versus actual indicators, trends are used to represent work backlogs like problem reports. The amount of work to be completed (problem reports to be fixed) is not known in advance, so the plan (or target) is developed week by week as problems are discovered.

3.2.2.2 Limit-Based Indicators

Limit-based indicators are used when the expected or planned value remains relatively constant. A change in the limits or targets associated with these indicators usually involves a major replan or a change in expectations for the program. **Feasibility analysis** of a limit-based indicator requires determining whether the proposed limits are reasonable and soundly-based in fact. **Performance analysis** consists of determining whether the actual program performance trespasses its established bounds.

Limit-based indicators include measures for error rates, computer utilization targets, and productivity goals. Figure 3-4 shows an example of a limit-based indicator for software size. As long as the actual size remains within the planned limit (initial estimate plus acceptable error), performance is acceptable. Whenever actual values exceed the limit(s), the cause should be investigated.

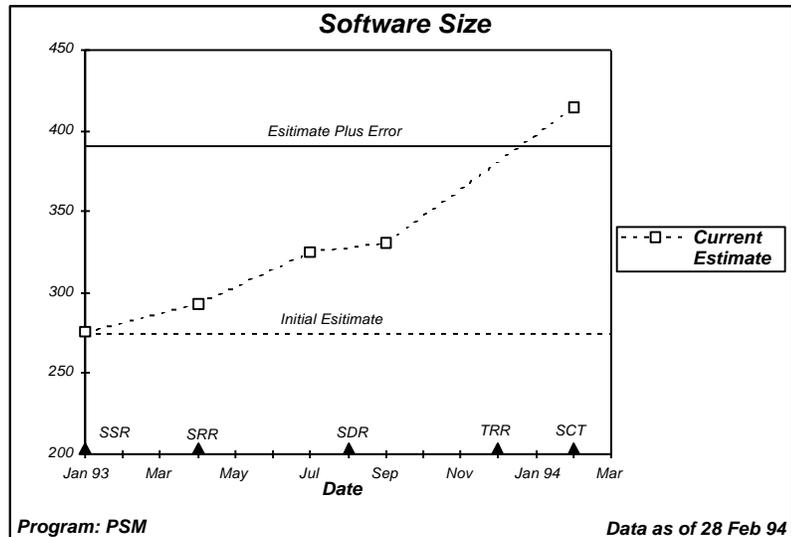


Figure 3-4. Limit-Based Indicator Example (Software Size)

Figure 3-3 in the preceding section also provides an example of a potential relationship between a limit-based and a trend-based indicator. The change in total work items is a change to a limit (the amount of work to be performed) which must be reflected in an update to the trend plan.

3.3 ANALYZE ISSUES

During this step, the indicators generated in the preceding step go through a systematic analysis process. This process results in assessment of the status of the program relative to the *known issues*. As shown in Figure 3-1, this analysis is based on both measurement and other program information. Only the integration of quantitative and qualitative data produces true program insight. The results of the analysis also are the basis for identifying *new issues* and taking corrective action on known issues.

The measurement process must be able to respond quickly to the information needs of program managers. Typical questions asked by program managers include the following:

- Can I trust the data?
- Is there really a problem?
- How big is the problem?
- What is the scope of the problem?

- What is causing the problem?
- Are there related problems?
- What should I expect to happen?
- What are my alternatives?
- What is the recommended course of action?
- When can I expect to see the results?

The measurement process must generate the answers to these questions.

During each analysis cycle, two types of analyses should be performed. **Feasibility analysis** is conducted to determine whether the software developer's plans and targets are achievable. **Performance analysis** is conducted to determine whether the developer is meeting the plans, assumptions, and targets. Sections 3.3.2 and 3.3.3, respectively, discuss feasibility analysis and performance analysis in more detail. The next section describes the general process in which either type of analysis can occur.

3.3.1 Basic Analysis Process

Analysis of measurement data tends to be a highly individualistic activity. However, the *credibility* and *completeness* of the analysis are enhanced when the analyst follows a *repeatable* process. Analysis results are more likely to be *useful* and the program manager will have a higher degree of *confidence* in them. This *Guide* will present the analysis activity from three perspectives: 1) tasks of steps that answer the Program Manager's questions, 2) analysis techniques (i.e. feasibility and performance) used during these tasks, and 3) life cycle phase. Figure 3-5 shows these perspectives.



The iterative nature of the analysis process at each step complicates the achievement of a thorough and repeatable analysis. The analyst may revisit earlier steps or jump ahead temporarily. However, following a basic sequence of tasks helps to ensure the effectiveness

of both feasibility and performance analysis. These tasks are as follows (see also Figure 3-5):

- Identification of Problems
- Assessment of Problem Impact
- Projection of Outcome
- Evaluation of Alternatives

Several of these steps will involve the collection of additional non-measurement data. Decisions cannot be based solely on software measurement data. Context information may be collected via developer feedback, audits, joint technical and management reviews, document reviews, and risk analyses. Gathering and integrating appropriate non-quantitative data is essential to the successful application of measurement.

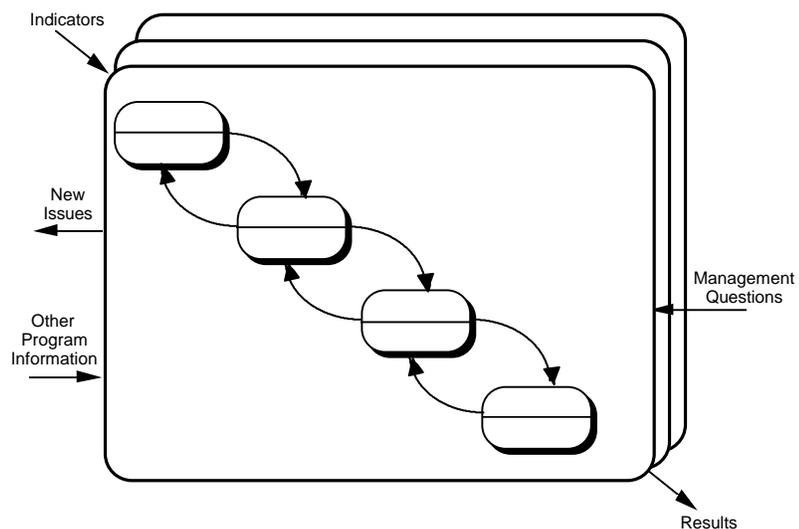


Figure 3-5. Issue Analysis Activities

3.3.1.1 Identification of Problems

Problems are recognized by detecting a difference between plans and actuals or between plans and other baselines. If the difference between these values exceeds the threshold of risk acceptable to management, then the situation should be investigated. Consider not just the absolute magnitude of the difference but also the trend. If a variance has been growing steadily larger month by month, it should be investigated even if it has not yet exceeded the threshold.

Both feasibility and performance analyses may be performed using a *single indicator technique*, one indicator at a time. However, because issues are not independent, we must also apply an *integrated analysis technique* using multiple indicators simultaneously. For example, a problem that should show up in one issue area (e.g., effort increases) may be disguised by an accommodation made in another issue area (e.g., schedules slip so that the increased effort does not result in a detectable increase in staff level).

These interactions among issues also suggest that a measurement program should never adopt a single issue focus. Single indicator analysis usually is performed first during the analysis cycle, followed by an integrated analysis.

Sometimes inconsistent, incorrect, or inaccurate data may cause an indicator to suggest a problem when none really exists. Discuss all data anomalies and other potential inconsistencies with the software developer. However, when multiple indicators point to a problem, it's usually not just a data problem.

3.3.1.2 Assessment of Problem Impact

The first step in assessing the impact of a problem is to localize the source of the anomaly detected and evaluate its scope. This may require additional focused data collection efforts or audits, but most measurement requirements should be satisfied with the existing data.

Sometimes a substantial difference between planned and actual values may be caused by outliers, which are values that don't appear to be consistent with the other data collected. For example, the average cyclomatic complexity of a component may be significantly higher than that of the rest of system due to one or two unusually complex units. Be careful to not make judgments about the whole system based on these outliers.

Once the source and scope of the problem has been identified the magnitude of its potential impact on program success can be assessed. The magnitude of the impact is not always proportional to the size of the difference between planned and actual. Sometimes, a small problem that arises in one issue area (e.g., size growth) may

have a ripple effect on another issue (causing, for example, effort over-runs). Rippling multiplies the effect of a problem.

Also consider the impact of any identified problems on program risk. A problem that does not by itself pose an obstacle to program success, but which increases the program's risk level should be managed carefully. A significant impact to an item on the critical path should always be of concern.

3.3.1.3 Projection of Outcome

Assessing the current impact of a problem helps to understand the probable outcome of the program. However, to get a complete picture of the significance of the problem its impact must be projected into the future. Eventual program outcomes can be predicted by projecting current trends as straight lines or by employing more sophisticated parametric estimation models for effort, size, and schedule.

Use these projection techniques to investigate the effects of changes in assumptions on program outcomes. Exploring these "what if" scenarios helps the measurement analyst to understand which factors most strongly influence program outcomes. Throughout these studies, keep in mind the imprecision inherent in such projections. Small differences in predicted outcomes are probably meaningless.

3.3.1.4 Evaluation of Alternatives

The information assembled via the preceding steps should enable the measurement analyst to evaluate alternative actions and make a recommendation to the program manager. The underlying problem and potential actions should be reviewed with the developer and modified as appropriate based on the developer's feedback. Consider the raw data, indicators, and context information about the program and recent events in reaching conclusions. Don't reach conclusions based on a single item of evidence, whether quantitative or subjective.

In deciding on a specific recommendation, consider the nature and effectiveness or impact of previously taken corrective actions. Avoid recommending a corrective action that will conflict with

previous actions or a corrective action that has already failed to work in a similar situation.

One result of the analysis process may be to identify a new issue and recommend the collection of additional data to track it. This will require the program to revisit the measurement tailoring process described in Chapter 2.

3.3.2 Feasibility Analysis

The four analysis steps just described can be performed to assess the feasibility of program plans. A feasibility analysis should be conducted with respect to an issue during the initial planning activity and at all subsequent replans. A program's failure may be the consequence of an overly ambitious plan as much as of poor performance.

The feasibility of a plan depends on the accuracy of assumptions and data as well as the effectiveness of the planning process. Some of the key considerations in determining the feasibility of a plan are as follows:

- **Basis for estimate-** How completely was the problem analyzed? How good is the historical data (e.g., past productivity)? Are the measures well-defined?
- **Realism of adjustments-** Do any adjustments for unique product or process factors (e.g., software development environment) reflect likely impacts rather than optimistic hopes?
- **Confidence in process-** Has the process that determines the plans or targets been used before? Did it give good results?
- **Changes in assumptions or environment** Have any significant changes occurred in the underlying assumptions or program environment which might affect the validity of the plan?
- **Comparison of program parameters** Are the performance levels or targets (e.g., productivity, quality) in the same range as those that have been achieved on similar programs?

Each part of the plan (such as size, schedule, staffing profile) may pass the tests above, but the plan may prove to be infeasible when considered as a whole. Figure 3-6 shows an overlay of a milestone (Gantt) chart on a staffing profile. (A similar effect could be

obtained by laying the charts side by side.) Note that the highly parallel design and implementation activities are scheduled during an interval of decreasing staffing. Thus, while the overall schedule may be adequate and the overall staffing sufficient, the allocation of staffing over time does not match the schedule. Following this plan, the program is sure to experience some periods where the staff exceeds the scheduled work and other periods where the staff is insufficient for the scheduled work.

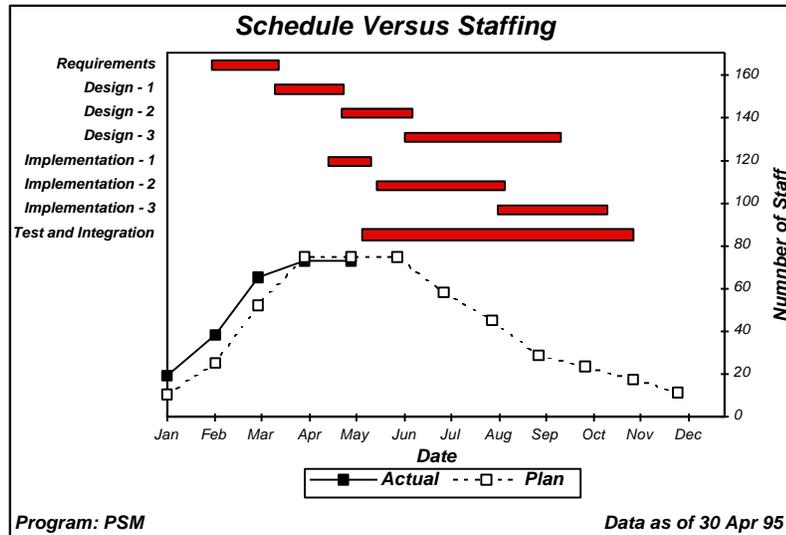


Figure 3-6. Software Development Personnel and Gantt Chart Analysis

3.3.3 Performance Analysis

Regardless of its feasibility or goodness, **once a program has committed to a plan, developer performance can be measured against the plan.** The program manager must pay close attention to how well the developer keeps to the plan.

Unfortunately, by the time a size, effort, or schedule performance problem is recognizable in a single indicator analysis, the problem has likely become one of major proportions. Thus, in evaluating performance the analyst must rely more on integrated analyses using multiple indicators. Some of the things to look for in such an analysis are as follows:

- **Leading indicators**- some indicators help to identify problems before they translate into a measurable schedule slip or cost over-run. For example, requirements changes usually precede size and effort

increases. Even if resources are not currently a problem on a program, a large number of requirements changes indicates that resources will become a problem if action is not taken.

- **Critical path items**- even if high level indicators suggest the program is moving ahead smoothly, delays and quality problems in a critical path item can have a ripple effect late in the program if not recognized and countered early.
- **Inconsistent trends**- sometimes two related indicators will suggest that different situations exist. Neither variance taken alone may be large enough to suggest a problem, but taken together they indicate that some element of the process is not working as planned.

Figure 3-7 shows an example of a problem made visible by detecting inconsistent trends. The figure overlays a design progress indicator with a problem report indicator. (The same effect can be seen by laying the graphs side by side.) While the measure of actual design progress appears to be only slightly behind the plan, the number of open problem reports is not going down. These open problem reports represent rework that must be completed before the design activity can be completed. Thus, the trends in these two indicators are inconsistent.

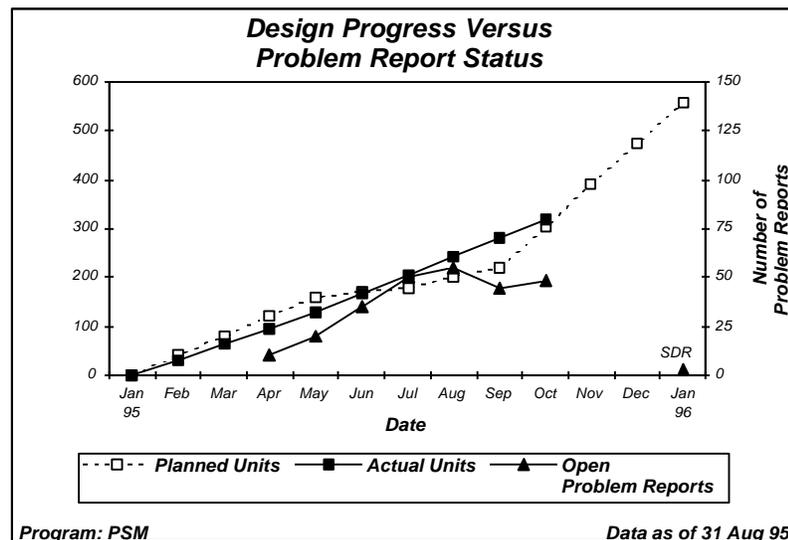


Figure 3-7. Development Progress and Problem Report Profile Correlation Example

Once the existence of a problem has been suggested by an analysis, the problem should be localized by examining indicators based on

more detailed data. In the example of Figure 3-7, the problem report indicator should be generated for each of the CSCIs within program XYZ. Identifying the specific source of the problem helps to determine the cause and select an appropriate corrective action.

Additional context information is needed to make valid interpretations as to the cause. For example, noting a discrepancy between the originally estimated software size and the current estimate (or actual) size does not provide enough information for management action. The size difference may result because 1) the size of the system was poorly estimated in the beginning, 2) significant requirements changes have occurred, or 3) changes were made in the way size is counted. Depending on the cause of the variance, different actions may be indicated.

3.4 REPORT RESULTS

The measurement analyst must regularly communicate the results of his or her analysis to the program manager. This communication is normally done via a briefing or report. The software measurement analyst should report the following:

- **Overall evaluation of program** status relative to the known program issues and projections of performance to completion
- **Identification of specific problems** location, cause, and impact of any problems identified in the analysis
- **Formulation of recommendations** alternative actions proposed for addressing the underlying problems identified in the analysis (with advantages and disadvantages of each)
- **Identification of potential new issues** nature of the problem or proposed actions may result in the identification of new issues that need to be tracked in the future.

Reporting and reviewing measurement results must be integrated into the management process. Two regular opportunities for management action are as follows:

- **Periodic status reviews**- concentrate on analyzing performance relative to plans and assumptions. Present system level graphs first. Only introduce more detailed

levels of analysis if a problem is identified. Do not expect to report a problem every time.

- **Major milestone reviews** consider feasibility and performance. Re-estimates should be prepared for effort, size, schedule, and other measures related to key program issues. The measurement analyst's assessment of these estimates should be presented at this review.

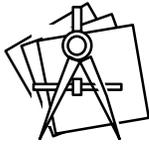
The reporting system should promote regular interaction and objective communication among the elements of the development team. (See Figure 1-3.) An effective vehicle for this is an online database and analysis capability accessible to both the developer and the Program Management team. Recognize that the measurement analysis report or briefing may contain proprietary or sensitive information. The measurement analyst must take appropriate steps to protect this information.

If possible, measurement results should be discussed with the appropriate software developer personnel prior to the formal review. This interaction provides an opportunity to discover events and qualitative information that helps explain what is happening in the data. **Measurement should be used for communication and understanding, not for punishment.**

The measurement analyst should record (and be prepared to explain) how analysis results and recommendations were arrived at. This may be need to justify decisions and trace recommendations back to the underlying data.

3.5 TAKE ACTION

The use of software measurement on a program does not require any special, additional management control functions. However, it does require that basic program management structures be in place. Measurement complements the existing planning and control activities. When management action is deemed appropriate based on measurement information, it should be implemented via the existing management structure and contractual mechanisms.



Measurement helps to recognize that a problem exists and to localize its cause. The identification of the underlying cause and selection of appropriate corrective action requires the application of good management and engineering judgment. **Action must be taken to realize any benefit from measurement.**

Sometimes the developer will recognize the problem and take action independently. At other times, the program manager will have to intervene. Examples of actions that might be taken by the program manager include the following:

- Extending the program schedule to maintain quality.
- Adding development resources to stay on schedule.
- Deleting functional capabilities to control costs.
- Changing the process to improve performance.
- Reallocating resources to support key activities.

Some of the actions listed above are significant and may not be possible. However, others attempt to optimize performance within the program's established constraints. Measurement can help the program manager to recognize and select the "best" course of action available.

Measurement assists in making predictions about likely program outcomes given different scenarios and actions. Current trends can be projected into the future. Historical data and qualitative experience from similar programs can also be very helpful in evaluating alternatives. All of these types of information help the program manager to arrive at the optimum decision within the bounds of program constraints.

Once a corrective action is initiated, additional indicators may be defined to assess the effectiveness of the action taken. Naturally, there is a delay between the start of a corrective action and the detection of its effects. Nevertheless, it is important to follow through to ensure that the desired outcome is realized. In most

cases new indicators to track actions can be defined using the data already collected.

3.6 LIFE CYCLE APPLICATION

While the philosophy of issues-oriented measurement and flexible analysis process advocated in this *Guide* applies throughout the life cycle, the issues, measures, and focus of analysis may change as the program progresses. This *Guide* adopts a three-phase life cycle model consisting of Program Planning, Development, and Software Support. This section discusses some of the unique measurement concerns in each life cycle phase.

3.6.1 Program Planning

During the Program Planning Phase, the program manager's primary concerns are assessing the feasibility of the program plans and selecting the most capable software developer for the job. Feasibility of plans should be assessed as described in Section 3.3.2 above. Two sets of plans must be analyzed as follows:

- **Program plan-** assess the required functionality, resources, and schedule defined for the program. Since it may be difficult to adjust the level of resources and schedule, the result of this assessment may be a quantification of risk rather than revised budgets and milestones.
- **Developer plan-** assess the developer's approach to satisfying the program plan in terms of required functionality, resources, and schedule. Also assess the technical approach, quality, and capability of each potential developer.

Since the overall functionality, resource and schedule envelope is established by the program, the technical approach, quality, and capability will be major criteria in the selection of the developer. Measurement-related information used to select the developer should include the following:

- **Past performance data-** the developer should be able to provide productivity and quality data from past projects. When comparing potential developers' past performance, be sure to compensate for differences in

how measures such as lines of code, errors, and effort are defined.

- **Overall process maturity** the measurement maturity of an organization is one dimension of its overall process maturity. Organizations with an ad-hoc process may have difficulty providing the basic measurement data described in this *Guide*.
- **Maturity of the measurement program** sometimes organizations that rate well in terms of overall process maturity have weak measurement programs. The ability of the developer to provide accurate and meaningful measurement data appropriate to the program issues must be considered.

Of course, the choice of a developer can not be based solely on measurement-related factors. The measurement capability of potential developers is just one more factor that needs to be considered along with the other technical, management, and experience factors on which a source selection is based.

3.6.2 Development

During the development phase, the program manager continues to be concerned with all six basic issues. Even developer capability needs to be tracked because it can change. For example, a high level of personnel turn-over could result in lower productivity. During this phase, the focus of analysis turns to performance relative to the plans, rather than the feasibility of the plans themselves. However, replans continue to be assessed for feasibility.

Development tasks often are categorized into four activities: Requirements Analysis, Design, Implementation, and Integration and Testing. Depending on the development model adopted for the program, these activities may be organized in different ways. Each new activity introduces new opportunities for measurement.

During the **requirements analysis** activity, the primary issues are *growth and stability*, *schedule and progress*, and *product quality*. The overall magnitude and stability of requirements can be tracked by counting requirements and changes to them. However, progress and quality are more difficult to measure during this phase. In part, this difficulty is caused by the ad-hoc nature of the requirements

process in many organizations. Measurement can only reflect the developer's process and product. It does not add structure.

The requirements process must be well defined to obtain meaningful measures. One effective requirements technique is to plan and conduct a series of reviews of parts of the requirements. This technique offers several opportunities for measurement. Completion of reviews can be tracked to assess progress. Action items and problems from the reviews can be tracked to assess quality.

During **design** and **implementation**, the focus is on *schedule and progress, product quality, and technical adequacy*. The program manager must continue to keep an eye on *growth and stability* to avoid surprises. Again, the opportunity to gain insight into program status depends on the structure of the developer's process. To the extent that the development process defines discrete design and implementation activities, progress is easier to measure. Sometimes progress comes at the expense of quality. The program manager needs to recognize and address that situation if it arises. During design and implementation, the adequacy of the developer's technical approach will be challenged. Any deficiencies must be recognized as soon as possible so that a work-around can be selected.

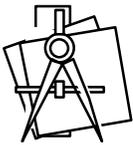
During **integration** and **testing**, the program focus is on getting the product ready to be deployed. This means evaluating *product quality*. The testing activity is often one of the shortest and most intense. Consequently, the measurement analyst must focus on providing rapid collection, analysis, and feedback to the program manager (especially on problem report status) so that effective decisions can be made. A weekly reporting interval often is used during this activity. In some cases, daily test progress and problem report status are provided. The determination of the reporting interval depends on many factors, but the measurement analyst should be prepared for this burst of activity during testing.

3.6.3 Software Support

Software support continues the transition in the program issues focus towards *product quality* and away from *size and growth*. Note that after deployment, when a system is normally in software support, large enhancements may take place that are really new

developments. These do not follow the usual “change and fix” process for software support.

The software support process may be implemented in many different ways. An organization different from the software developer often handles software support. That organization is likely to use a different management structure, personnel, and process than the developer. Even though the basic principles still apply in software support, the measurement program for the software support organization usually needs to be planned separately from that of the software developer. For example, during development work unit progress measures may be collected to track the design, coding, and integration and testing of components. However, during software support the unit of work tracked becomes the change request rather than the component.



During software support, problem reports and change requests may be handled individually or bundled together to define a new version of the software product. It is easier to measure and control the version-based process. However, the nature of the system being supported often dictates the version release strategy and other aspects of the software engineering process.

CHAPTER 4 - IMPLEMENTING A MEASUREMENT PROCESS

The previous chapters describe the software measurement process. This process includes the tailoring and application of software measures to address specific program issues. A well defined measurement process is of little value if it is not properly implemented within the organization. This chapter addresses how to do this, and describes four key measurement implementation activities. The chapter also addresses how measurement information can be used to support overall organizational requirements. Although this chapter approached measurement implementation from the perspective of a DoD acquisition organization, much of the guidance is applicable to any type of organization implementing measurement.

4.1 MEASUREMENT IMPLEMENTATION OVERVIEW

Implementing a measurement process within an organization is similar to implementing any new initiative or function. Measurement represents a significant change in how an organization does business, and the issues and concerns related to this change must be directly addressed.

There are four key activities which must take place to effectively introduce software measurement into an organization. These activities are depicted in Figure 4-1 and described as follows:

Obtain Organizational Support - This activity is concerned with generating support for software measurement at all levels throughout the organization. Management mandated measurement without organizational buy-in and support will seldom succeed. All members of the organization, at all levels, need to understand how measurement will directly benefit their programs and their own work processes.

Define Measurement Responsibilities - This activity involves establishing and assigning measurement related responsibility

within the organization. The key positions generally responsible for software measurement include the organizational and program managers, the measurement analyst, and other members of the technical and management staff who are involved with software acquisition and development activities. Clear definitions of who is responsible for what parts of the measurement process are important to successful implementation.

Provide Measurement Resources - This activity establishes the measurement resources required to implement the measurement process within the organization. These resources include tools and funding of the measurement effort

Initiate the Measurement Process - This activity involves transitioning the focus from *establishing* the measurement process to actually *applying* it within the context of a software program.

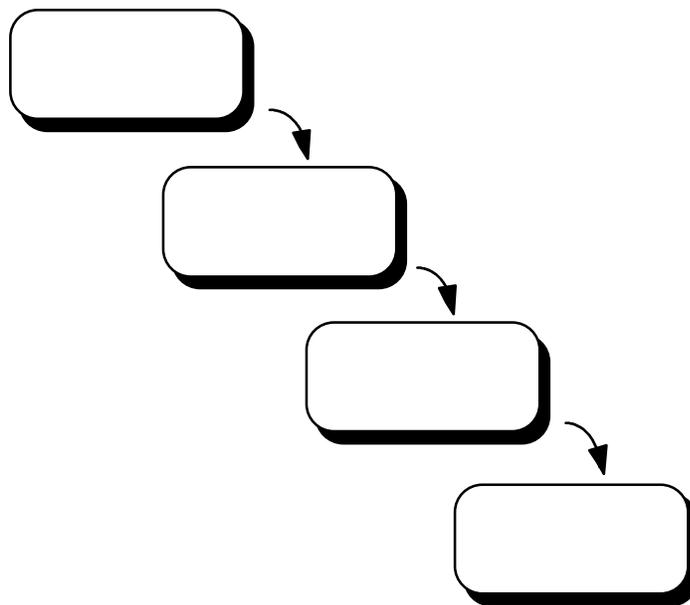


Figure 4-1. Measurement Implementation Activities

4.2 MEASUREMENT IMPLEMENTATION ACTIVITIES

Software measurement is a useful tool which can help most DoD organizations improve the management of their programs and help them meet organizational objectives. Like any tool, measurement

must be implemented correctly for it to be of any help. The following sections address the activities and issues related to implementing a measurement process.

4.2.1 Obtain Organizational Support

Implementing measurement in an organization represents a major cultural change. Fear always exists that the measurement results will be used improperly, to evaluate individual performance or to arbitrarily rank development organizations. There may be concern that measurement will highlight problems in a program or in an organization, problems which were not visible before the measurement process was implemented. Maybe the measurement analysis will show that the software development plan was unrealistic, or that only a portion of the software functionality will actually be delivered. These concerns are real, and to overcome them requires the support of all levels within the organization.

To successfully implement a measurement process, management support is critical. This goes beyond the senior managers saying that software measurement is “a good idea”. Management must take an active and public interest in the measurement process. They must be seen as supporting the process by providing adequate resources, asking for data and analyses, and acting on that analysis. The entire organization will then understand that measurement is important, and begin to actively support it as well. A measurement process requires enthusiastic leadership at the highest levels of the organization to make it work.

Many managers first learn about software measurement when some significant software “event” brings into question the way a program or organization is being managed. Others learn about it as a result of a policy directive or initiative. Few managers are first introduced to software measurement as an effective program management tool that can help to achieve defined program and organizational objectives. In many cases, management views measurement as “another thing to do” and as something that will require resources that are already committed.

The key benefits of measurement to the organization should be clearly identified. These include:

- objective insight into organizational issues and processes

- early detection and resolution of software problems
- the availability of objective information to identify and manage risk
- objective program team and organizational communications
- the ability to assess organizational performance
- the ability to objectively defend and justify program and organizational decisions

In addition to management support, measurement has to be adopted and supported at lower levels in the organization. Most people want to do a good job, and measurement can help them. Evaluating acquisition alternatives, assessing the feasibility of proposed software plans, and identifying the key areas of technical concern are all activities which involve the use of measurement. One of the important aspects of obtaining support for measurement throughout the organization is to ensure that everyone understands that the measurement results will be used to support organizational objectives, and not used to evaluate individual performance.

4.2.2 Define Measurement Responsibilities

The size and structure of each specific organization is directly related to how measurement responsibility is assigned. How many people are involved, and how the measurement tasks are actually allocated, vary considerably from organization to organization. In general, responsibility for implementing the measurement process is focused at different levels.

The primary responsibility for the measurement process is at the management level. In many DoD organizations, two different types of managers are involved in the acquisition and support of software intensive systems:

- **Executive Manager**- The executive manager, who in many cases is the Program Executive Officer (PEO), generally has responsibility for more than one program. The organizational manager's decisions materially affect all of the programs within the organization. Measurement helps the organization manager to determine the status of individual programs, and to make decisions which apply across the organization.

- **Program Manager-** The program manager has direct responsibility for a software intensive program. In most cases, the program manager is the primary user of measurement results. He is responsible for identifying and managing the software issues, and communicating with the program team, including the developer and senior levels of DoD management. The program manager uses measurement to make program decisions.

In some DoD organizations, the program manager is also the organizational manager. It is the program manager's responsibility to ensure that measurement is integrated into the program. Integration includes all of the activities which make measurement part of the overall program management and technical processes, including the identification of resources to support the measurement effort.

While management is responsible for integrating and using measurement within the organization, the program technical staff is usually assigned the day to day tasks related to tailoring and applying the measures. One of the key responsibilities is that of the measurement analyst. The measurement analyst has the primary responsibility for tailoring the measures, collecting and processing the measurement data, analyzing the measurement results, and reporting the results to management. The measurement analyst is the primary measurement point of contact with the developer. with respect to measurement. In short, the measurement analyst ensures that the measurement process is implemented properly, and that the program manager is getting the software information required to properly manage the program.

Depending on the size and scope of the program, the program office's measurement team can consist of a part-time measurement analyst or a multi-person team. The important thing is to have the primary measurement responsibility for the program assigned to a specific individual, and to allow that individual to interface directly with the development team. If established, the measurement analyst should be a member of the software engineering Integrated Product Team (IPT). Above all, the measurement analyst must be able to independently arrive at objective answers, and be able to convey those answers directly to the program manager.

Other members of the program office technical staff also have responsibility within the measurement process. They should each

understand how the process works and what information it can provide to them. They should also support measurement analysis efforts by helping to identify program events which may have an impact on interpreting the measurement data.

Although the development organization is not part of the program office staff, it plays an important role in the measurement process. Most of the software data used by both the developer and the program office comes from the developer. All users must understand how each measure is defined and what the data represents. What software WBS elements, for example, are included in the reported software effort data? The developer should also designate a key measurement point of contact to interface with the program office's measurement analyst on a regular basis. The developer's measurement point of contact should be part of any program software engineering IPT.

4.2.3 Provide Measurement Resources

Experience suggests that the measurement process will require from 1 to 5 percent of the total software program cost. Measurement costs include personnel and tools, as well as the cost for the developer to assemble and report the data. Most developers use software data internally to manage their programs. As such, the program office should not incur a considerable amount of additional cost for the data to be collected. If the developer does not collect software data, there should be some concern about the maturity of the underlying software process.

As with any initiative, there are some non-recurring startup costs associated with implementing a measurement process. These costs, which include both training and tools, diminish as measurement becomes a day to day activity within the organization. It is important to view the measurement process as a long-term resource within the organization. It should be self supporting, saving as much as it costs, within a relatively short time after it is established.

In some DoD organizations, the measurement costs for individual programs can be reduced by establishing the measurement team as an organizational resource. As long as there is a primary analyst assigned to work independently on each program, the measurement team can share resources, tools, and expertise.

4.2.3.1 Measurement Tools

Once the specific measurement requirements have been established, the tools used to collect, process, and analyze the data should be identified. On many smaller programs, the measurement process can be adequately supported using a personal computer with an common suite of integrated office software. On larger programs, or on programs which need to implement more advanced analysis techniques, additional measurement tools are usually required. When deciding what resources are required, the wrong thing to do is to purchase a specific tool before determining if it supports the information needs of the program. The types of software issues that need to be addressed and the characteristics of the measurement process drive the support tool requirements. The process should never be implemented around a pre-defined set of measurement tools.

Several different classes of tools are commonly applied in the measurement process. Many are used by the developer but may be accessed by the program office.

- **Database, Graphing, and Reporting Tools** These tools manage and store the measurement data and produce graphical and text based reports. Commercial personal computer database applications are generally adequate for most programs. For larger programs with extensive data management and storage requirements, consideration should be given to using more powerful applications.
- **Software Analysis and Modeling Tools** These tools provide enhanced graphics and software analysis capabilities generally unavailable from databases or spreadsheets. The category includes software cost estimation models, software reliability models, statistical analysis tools, and similar applications. These tools can be extremely valuable when implemented as part of the overall measurement process.
- **Measurement “Workstation” Tools** - These applications support user interaction at all levels of the organization by providing real time access to both measurement data and analysis results. They are very useful for summarizing and providing measurement information at the management level.
- **Schedule and Project Management Tools**- These tools assist in program scheduling progress tracking,

and critical path analysis. Some tools in this category can also track resource allocations and expenditures for identified activities.

- **Financial Management Tools**- These tools help to collect and store data related to labor and funds expenditures. Some tools in this category include cost accounting and earned value functions. In some cases existing financial management systems may not provide software specific data at an adequate level of detail. These systems may be difficult to modify.
- **Software Product Analysis Tools** These tools generate software product related data through automatic analysis of specific software products. Examples include software complexity analyzers, software size counting utilities, and similar product measurement oriented applications.
- **Software Data Collection Tools** These tools help to automatically extract software measurement data from systems which support the developer's software process. They can be commercial or locally developed utilities which access the developer's CASE tools, configuration management tools, and other software related systems. They are useful for providing the program office with direct access to the developer's measurement data.
- **Office Automation Tools** These tools provide standard office automation applications such as word processors, spreadsheets, and presentation graphics. They can effectively support basic measurement analysis activities and help to produce measurement related graphs and reports.

General guidelines for selecting tools to support the measurement process include the following:

- Select tools that support the measurement process as tailored to meet specific program needs. Do not build a process around the tools
- Evaluate tools that may already be available within the organization.
- Select tools that automate as much of the measurement process as possible. Automated data collection, data processing, analysis, and reporting tools can considerably improve the efficiency of the measurement process.

- Work closely with the developer to coordinate measurement tool selection and implementation especially with respect to electronic data transfer
- Select tools that simplify importing and exporting data easily between different formats
- Select tools that run on a common platform

On most programs, some manual data entry will usually be required. This should be kept to a minimum. It is usually more cost effective to implement commercially available tools and applications instead of developing them in-house. Data transfer utilities which provide direct access to the developer's measurement data in many cases are unique to each program. It is cost effective to implement these utilities rather than to rely on manual data transfer and entry.

4.2.3.2 Measurement Training

Personnel at all levels of the organization require appropriate software measurement training. Figure 4-2 summarizes the general training requirements for different personnel in the program organization.

Program managers require a good foundation in the basic concepts of software engineering and software measurement. They need to understand the capabilities and limitations of the measurement process measurement and how it can help them to meet their objectives.

| Job Function | | | Measurement Training Requirement |
|-----------------|--------------------------------|---------------------|----------------------------------|
| Program Manager | Technical Managers & Engineers | Measurement Analyst | |
| • | • | • | Software Engineering |
| • | • | • | Measurement Overview |
| | | • | Data Collection and Management |
| | | • | Measurement Analysis |

Figure 4-2. Measurement Training Requirements for Program Personnel

Program office technical managers and engineers require training in the basic concepts of software engineering and measurement. They

must understand how the data will be used within the program organization and how measurement will impact their own work

Measurement analysts need appropriate training and experience in software engineering, the measurement process, and in specific software measurement disciplines. Software engineering expertise is critical to the success of the measurement analyst. It provides the basis for interpreting and analyzing the data. Every measurement analyst should understand the activities and products inherent to the software process, and be able to relate program software issues to specific measures and analysis activities. Software estimation and modeling skills, and statistical analysis experience is required for more advanced analysis.

4.2.4 Initiate the Measurement Process

On most programs, some data collection and analysis occurs immediately after the decision is made to implement a measurement process. It is not unusual for all of the implementation activities to be taking place concurrently. A key requirement is to show how the measurement process can help address even the basic software issues and start to answer the Program Manager's questions. Even if the program is large, initially implementing a few key measures will provide important information that was not previously available.

One of the most important things to do is to establish an interface between the program office and the developer with respect to software measurement. Once established, this interface will become one of the important tools in the measurement process. Direct access to the developer allows the measurement analyst to freely address data issues, and allows for analysis feedback to be provided to the developer on a working level. In some instances, the program office - developer interface can be established as part of an IPT.

Just establishing a measurement process will not have an immediate impact on the program. As the measurement process is implemented, use of the measurement results will need to be "marketed" within the organization. At this point in time it is especially important to use the measurement results correctly. The data should be well defined, the analysis should be accurate, and the developer should have an opportunity to address the results.

After review by the Program Manager, the measurement information should be made available to the entire program team. This should include the developer. Discussion of the measurement results with the developer should focus on how the measurement results reflect what is actually happening on the program, and if new issues identified by the analysis are valid. The developer is important to the measurement process. If the developer is punished for poor measurement results, then the flow of data may be impeded or manipulated, resulting in the loss of program insight and communication.

The measurement process tends to impose a discipline on program software management activities. If the measurement process is implemented properly, the results will be used throughout the organization. It will provide insight into the program issues and help management to make informed software decisions.

4.3 USING THE MEASUREMENT RESULTS

The primary user of software measurement information is the individual program team. The team includes the DoD program office and technical support organizations, as well as the software developer and associated development organizations. Other organizations, particularly those with responsibility within the DoD acquisition structure, have a valid need for information which is available from the measurement process. Each of these information needs is somewhat unique. This is due to the fact that each organization has a different role with respect to the program, and must address different issues and questions. Figure 4-3 illustrates the “types” of measurement information requirements within the DoD structure. There are three primary viewpoints, the Program Development Team, DoD Executive Management, and Software Process Improvement groups.

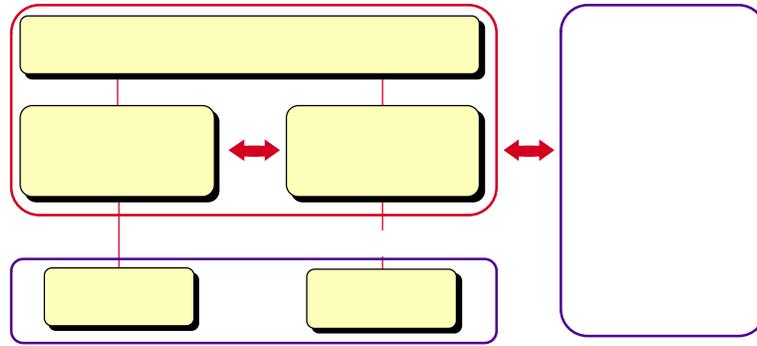


Figure 4-3. DoD Software Measurement Information Requirements

Not only is the program team the primary user of the software measurement information, it is the primary source for the software data and analysis results used throughout the structure. The program manager has considerable influence over how the data is used within his own program organization. In many instances, the program manager is required to provide data outside of the program development team. This can be of some concern, especially if there is some question as to how the information will be interpreted. Before using the measurement results, all organizations should have a clear understanding of how to interpret the information with respect to the specific program in question. This requires all users to understand what the data represents, how the analysis was conducted, and how does the measurement information relate to the context of the program. All users should understand the measurement process, especially its capabilities and limitations. The objective of the program measurement process, even at higher levels of the organization, remains the identification and management of software issues, not to grade or punish the program organizations or individual developers.

Measurement can be a powerful tool, but it can also be misused. Using measurement results to compare and rank different programs with respect to performance is a primary example. Software measurement is different for every program. The measures that are used and how they are defined are different, as are the overall technical and management processes that the measures represent. Even though there is a need to quantify program performance in a standard manner, in most cases a comparison of programs using the software measurement results will be invalid.

4.3.1 Program Development Viewpoint

The program development organization has two primary decision-makers that need measurement information: the DoD Program Manager, and the Development Manager. They use the measurement information in three ways:

- To analyze options and trade-offs
- To make program decisions
- To communicate program status

Integrated Product and Process Development (IPPD), implemented through Integrated Product Teams (IPTs), provides a natural mechanism for the use of measurement information. The purpose of the IPT is to make team decisions based on timely and objective data from the entire team, and software measurement information specifically supports this objective. Measurement information provides a basis for continuous feedback and discussion between the Government and the developer team.

One of the most important uses of measurement at the program level is to help define feasible software plans. The measurement process will quickly identify if a program is not tracking to plan. In many cases this is due to the plan being unrealistic. Using the measurement information to trade off and manage software cost schedule, and capability objectives and constraints helps to establish achievable goals for the program team. At the very least, the measurement information can be used to objectively address the constraints when they cannot be materially changed.

4.3.2 DoD Executive Management Viewpoint

There are many uses of the measurement information outside of the program organization. One of the most important is to satisfy DoD executive management reporting requirements. Software measurement can help in reporting the overall status of the program. Objective data gives external organizations confidence that the status of the program is accurately represented. Measurement information also assists the DoD Program Manager in coordinating with other joint or inter-related programs, particularly on issues such as schedule. It also helps him to show how the critical software portion of the program is being managed, and how he is determining the status of the software with respect to

readiness for operational test and delivery. Justifying decisions is easier when based on a repeatable process that uses measurement data. When DoD Management asks “Why did you decide to take this course of action?” the DoD Program Manager can pro-actively and objectively support his decision.

Oversight Organizations have special information needs. Using measurement to support oversight requirements is challenging, because the measurement results must be conveyed within the technical and management context of the software effort. Measurement can help by providing objective data that clearly relates the program’s status. Insightful analyses can help in understanding the type and criticality of the issues a program faces. More importantly, the measurement information can lead Oversight Organizations to ask the right questions

Comments and direction from all DoD management organizations should be fed back to the program measurement process. If there are upper level concerns about a particular software issue, measurement can be used at the program level to address it.

4.3.3 Process Improvement Viewpoint

Software measurement is also used outside of the program organization to support software process improvement. Software Engineering Process Groups (SEPG), in both the government acquisition and developer organizations, use the measurement data to help identify candidate areas for process improvement activities. Measurement also helps to evaluate the effects of process changes across an organization. Without measurement, an organization can have little confidence that it is improving.

4.3.4 Lessons Learned

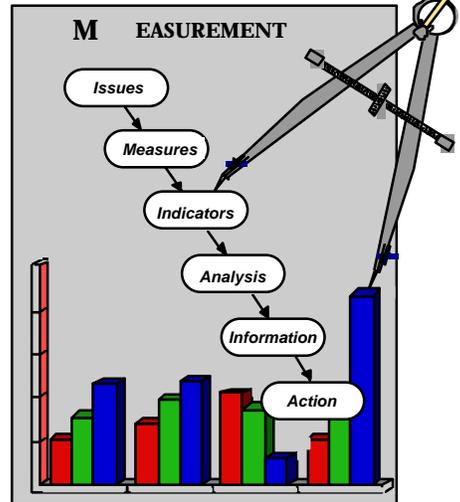
Figure 4-4 summarizes some of the important lessons learned in getting a measurement process started and then using its results.

| Lessons Learned | |
|--|--|
| Getting Started | Using Results |
| Ensure that everyone in the organization understands both the capabilities and limitations of the measurement process. | Do not allow anyone in the organization to use measurement to evaluate individual or workgroup performance. |
| Start small. Implement only a few measures to address key issues and show how the measurement results support both individual and management objectives | Make the measurement data and information available to everyone in the organization. This is a key approach in helping people to actually use the results. If the information is valid, people will find a way to use it |
| Ensure that only the required measures are implemented, based on the issues and objectives of the organization. If you don't need the data, don't collect it. The measurement process must be cost effective to succeed. | Do something early. A considerable amount of meaningful analysis can be performed with a minimal amount of data. Don't wait until all of the data is available to apply it. |
| Assign a key individual to implement the measurement process. This "measurement analyst" should be an integral part of the program team and should act as the primary interface with the developer with respect to software measurement. | Different levels within the same organization have different information needs. Organization managers may make investment decisions with respect to software process technology and tools while program managers make decisions about specific technologies used to best satisfy program objectives. Organizational issues and objectives do not always equate to those of a specific program. |
| The Program Manager should not incur significant costs from the program for the developer to collect software data. The unavailability of data may indicate a low level of maturity in the developer's software process. | Measurement should be made an integral part of the program or organization. Measurement should support the existing management and technical processes. Measurement should not be treated as an "add on" within the organization. |
| The measurement process can initially be implemented with basic, commercially available database, spreadsheet, word processing, and presentation graphics applications. More advanced tools can be added as required. | The program manager must be at least willing to listen to "bad news" resulting from the measurement analysis. Not every analysis result requires action. In some cases the recommended action is not feasible. Measurement is intended to help the program manager make a decision, not make it for him |
| All users at all levels must understand what the measurement data represents. This understanding is vital to the proper interpretation of the measurement analysis results. | Management should not try to "influence" the measurement results before they are reported. They should, however, understand how the reported results were arrived at and what they mean with respect to the associated software issues. |
| | Pro-actively use the measurement information to report program status. |

Figure 4-4. Lessons Learned for Measurement Implementation

PRACTICAL

SOFTWARE



SELECTING AND SPECIFYING PROGRAM MEASURES

PART 2

SELECTING AND SPECIFYING PROGRAM MEASURES

Part 1 of the Guide describes the overall measurement tailoring process, and explains each of the three associated tailoring activities. This part of the guide, Part 2, addresses the second step of the tailoring process in more detail. It shows how to actually use the PSM guidance to select the appropriate measures and to specify the related data and implementation requirements.

This part of the guide is organized into three chapters:

- Chapter 1, How to Select and Specify Program Measures - describes how to select from a set of proven measures based upon a prioritized list of program issues and questions. It explains how to use the detailed PSM measurement selection and specification information found in Chapter 2.*
- Chapter 2, Detailed Measurement Selection and Specification Information - packages measurement selection and specification experience derived from successful DoD programs into a series of tables which helps you to choose which measurement categories and individual measures are correct for your program. The tables also provide specification guidance for each measure which helps you to define associated data and implementation requirements.*
- Chapter 3, Measurement Selection and Specification Example - shows how the guidance in Chapters 2 and 3 are used to select and define the measures within a typical DoD program scenario.*

The PSM measurement selection and specification guidance is based upon actual implementation experience. It is a compilation of the best and most commonly used measurement practices which have helped DoD Program Managers achieve success on past programs.

TABLE OF CONTENTS

CHAPTER 1- HOW TO SELECT AND SPECIFY PROGRAM MEASURES..... 87

- 1.1 Introduction.....87**
- 1.2 Identifying and Prioritizing Program Issues.....90**
- 1.3 Selecting the Appropriate Measurement Categories.....91**
- 1.4 Selecting the Applicable Measures.....93**
- 1.5 Specifying Measurement Data and Implementation Requirements.....95**
- 1.6 Selecting and Specifying Measures for Existing Programs.....98**

CHAPTER 2 – DETAILED MEASUREMENT SELECTION AND SPECIFICATION INFORMATION..... 101

- 2.1 Introduction..... 101**
- 2.2 How To Use the Measurement Tables..... 101**
 - 2.2.1 Measurement Category Tables..... 102
 - 2.2.2 Measurement Description Tables..... 104
 - 2.2.3 General Measurement Specification Table..... 107
 - 2.2.4 Additional Implementation Guidance..... 107
 - 2.2.5 Measurement Selection and Specification Tables..... 108

CHAPTER 3 – MEASUREMENT SELECTION AND SPECIFICATION EXAMPLE.173

- 3.1 Program Scenario..... 173**
- 3.2 Measurement Selection Summary..... 174**

CHAPTER 1- HOW TO SELECT AND SPECIFY PROGRAM MEASURES

One of the most important aspects of implementing the measurement process is tailoring it to meet the specific needs of your program. There are three activities associated with tailoring the process. First, the issues which characterize the program must be identified and prioritized. Second, the software measures which best address these issues must be selected and the associated data and implementation requirements specified. Third, the selected measures, data requirements, and implementation requirements must be integrated into the software process. During integration, the measures and the requirements are revised to better reflect the characteristics of the software development environment. The result of the tailoring effort is a well defined measurement plan which directly addresses the program's unique information needs, and which can be implemented without materially impacting the developer's software process.

Part 1 of *Practical Software Measurement* explains how to identify and prioritize program specific issues. Once these issues are identified, the guidance provided in this chapter shows how to actually select the appropriate measures and how to specify the related data and implementation requirements.

The software measures presented in the Part 2 tables of the *Guide* are widely used for program management purposes. However, they are not meant to imply an exhaustive or required set of measures. *PSM* provides guidance for tailoring any measure, whether or not it is included in the Part 2 tables.

1.1 INTRODUCTION

To be effective, the measures that you select must directly address the specific software management and technical issues which characterize your program. Since every program is described by a unique set of issues, the applied measurement sets are also unique. Each measurement set must be tailored to meet the specific

information requirements and characteristics of each individual program.

The *PSM* approach used for selecting and specifying program measures is based upon the direct relationship between program issues, information needs, and the specific measures which provide the required information.

Three key *PSM* mechanisms which support the measurement selection and specification approach. These include: 1) a set of **common software issues**, which allows you to group closely related program issues into a manageable structure, 2) the definition of different **measurement categories** for each common issue, each of which groups the measures which provide similar types of information related to that issue, and 3) **detailed measurement descriptions**, which define each individual measure with respect to when it is best used, what data it provides, and how it should be implemented.

Figure 2-1 shows how the *PSM* mechanisms help you to select the measures that are most appropriate for your program.

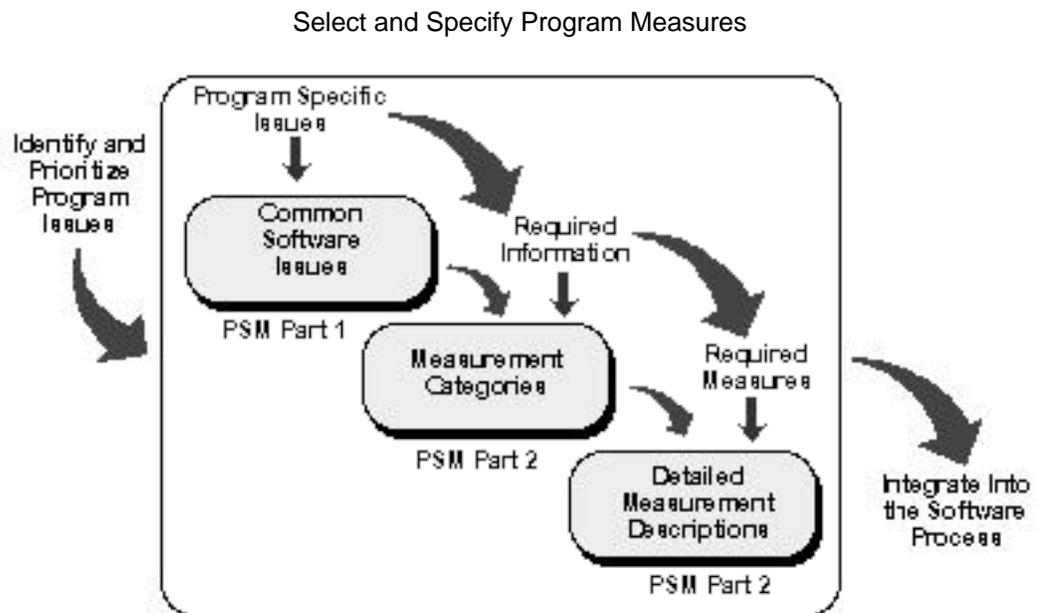


Figure 2-1 PSM Measurement Selection Mechanisms

The three *PSM* mechanisms help to map the issues to the measures, and subsequently to define the data and implementation requirements for each selected measure. To select the appropriate

measures, the specific program issues are prioritized and allocated to the *PSM* Common Issues as described in Part 1 of the Guide. The Measurement Categories which provide the types of information necessary to adequately address each of the defined Common Issues are then selected, using the measurement category tables contained in Chapter 2, *Detailed Measurement Selection and Specification Information*. The individual measures which comprise each Measurement Category are then reviewed for specific applicability, and after the appropriate measures are selected, associated data and implementation requirements are defined for each. This is accomplished using the detailed measure tables contained in Chapter 2.

Although most of the measurement requirements are usually defined when the software measurement process is initially tailored for a program, changes to the applied measurement set are sometimes required due to changes in the program's issues and information requirements. This is most common in large software development programs, and is a result of the changing priority of program issues due to normal changes in the management focus over the life cycle of the development, redirection of the program's objectives, or the identification of new risks and issues.

The *PSM* measurement selection and specification guidance is designed to simplify the mapping of the measures to the program issues. As such, measures which in reality support multiple software issues are listed under a single primary issue. As you use the guidance to select your measures, keep in mind that many of the measures do provide insight into more than one common issue.

The measures which are the foundation for the *PSM* measurement selection and specification approach are described in Chapter 2, *Detailed Measurement Selection and Specification Information*. These measures are not intended to represent an exhaustive list of program management measures. They are, however, measures which have repeatedly proven to be effective over a wide range of successful programs, and represent the best practices for addressing the issues faced by most DoD Program Managers responsible for software intensive systems.

No program should implement all of the measures listed in Chapter 2. Although the measures that are implemented are driven primarily by the issues which must be addressed, the overall characteristics of

the program and the software process also need to be taken into consideration during the selection process. The types of indicators and graphs also have a bearing on the measures which are selected. Although the *PSM* measurement process addresses measurement application (e.g. tailoring and analysis) separately, anticipation of the types of graphs and reports which may be needed helps to define which measures are required.

1.2 IDENTIFYING AND PRIORITIZING PROGRAM ISSUES

The first step in selecting and specifying the measures for any program is to identify and prioritize the specific program software issues which the measurement process has to address. As discussed in Part 1, there are many sources of information which help the Program Manager define these software issues. The issue or risk identification process which helps to do this is usually implemented, either formally or informally, on most DoD programs. The information which helps to identify and prioritize the program specific software issues is derived from risk analysis results, definition and recognition of program constraints and assumptions, identification of leveraged software processes and technologies which have a direct bearing on program success, and the overall experience of the Program Manager and the program team. External oversight and reporting requirements also influence the issues and questions that need to be addressed by measurement

As the above information is gathered and reviewed by the program team, an overall software issue profile which characterizes the program is developed. This issue profile generally includes the prioritized list of program management and technical risks, issues, and questions, and helps to focus the information and analysis needs within the program.

The prioritized list of program software issues is the starting point for selecting and specifying the required software measures. Rather than provide an exhaustive list of all potential software issues which map to all applicable measures, *PSM* allocates the identified program specific issues to a set of six Common Software Issues. These Common Software Issues are basic to all programs and represents the key software areas which must be managed on a day to day basis by DoD program managers. They help to simplify the mapping of program issues to the measures. This is accomplished

by first mapping the set of Common Issues to pre-defined measurement categories, and then to the measures listed within each category.

1.3 SELECTING THE APPROPRIATE MEASUREMENT CATEGORIES

Once the program specific software issues are identified, prioritized, and allocated to one of the six Common Software Issues, the program team will have a good understanding of the software questions they will be called upon to answer, and the types of information that they require. The second step in selecting and specifying the measures for the program is to select the measurement categories which include the measures that provide the needed information.

Figure 2-2 shows the complete mapping of the Common Issues to the *PSM* measurement categories and associated measures. Some common issues are comprised of only a single measurement category, and some measurement categories contain only a single measure. Since the *PSM* guidance is based upon actual measurement experience, the overall measurement selection structure is intended to be augmented and modified to meet individual program needs.

As shown in Figure 2-2, each measurement category is mapped to a single common issue. Within each common issue, the measurement categories are differentiated by the distinct types of information and questions that their respective measures can address. Under the common issue of Schedule and Progress, for example, there are four different measurement categories, Milestone Performance, Work Unit Progress, Schedule Performance and Incremental Capability. The measures in all of these categories address schedule and progress related issues, but they do so with different types of information at different levels of detail.

Milestone performance measures provide basic start and end dates for defined software activities and events. This is adequate for developing and reviewing Gantt-like schedules, but the measures do not address the degree of completion of the individual software activities and products at any point in time. More detailed schedule and progress information is provided by the measures in the Work Unit Progress measurement category. Earned value based schedule

Part 2 - Selecting and Specifying Program Measures

information, with schedule variances expressed in dollars, is provided by the measures allocated to Schedule Performance. Lastly, the measures which comprise the Incremental Capability category show whether or not planned software components are being completed as planned in an incremental type of software development.

| Issues - Categories - Measures Mapping | | |
|---|--|--|
| Issue | Category | Measure |
| Schedule and Progress | Milestone Performance | Milestone Dates |
| | Work Unit Progress | Components Designed Components Implemented Components Integrated and Tested Requirements Allocated Requirements Tested Test Cases Completed Paths Tested Problem Reports Resolved Reviews Completed Changes Implemented |
| | Schedule Performance Incremental Capability | Schedule Variance Build Content - Component Build Content - Function |
| Resources and Cost | Effort Profile | Effort |
| | Staff Profile | Staff Level Staff Experience Staff Turnover |
| | Cost Performance | Cost Variance Cost Profile |
| | Environment Availability | Resource Availability Dates Resource Utilization |
| Growth and Stability | Product Size and Stability | Lines of Code Components Words of Memory Database Size |
| | Functional Size and Stability | Requirements Function Points |
| | Target Computer Resource Utilization | CPU Utilization CPU Throughput I/O Utilization I/O Throughput Memory Utilization Storage Utilization Response Time |
| Product Quality | Defect Profile | Problem Report Trends Problem Report Aging Defect Density Failure Interval |
| | Complexity | Cyclomatic Complexity |
| Development Performance | Process Maturity | Capability Maturity Model Level |
| | Productivity | Product/Effort Ratio Function/Effort Ratio |
| | Rework | Rework Size Rework Effort |

| | | |
|---------------------------|--------------------|------|
| Technical Adequacy | Technology Impacts | None |
|---------------------------|--------------------|------|

Figure 2-2 Mapping Common Issues to Measurement Categories and Measures

The Measurement Category tables in Chapter 2 describe each *PSM* Measurement Category in detail. The tables define each category in terms of what information is provided by the included measures, and the applicability of the measures within the category to different types of programs and software processes. The tables also identify the limitations of the types of measures in the category. The tables help you to determine which of the measurement categories best satisfy the projected information requirements for the issues you have defined.

The Measurement Category tables in Chapter 2 are grouped with similar tables which describe each individual measure within that category. It is recommended that you review both the category and associated measurement tables together to help determine if the information provided by a particular measurement category will meet your needs. Always try to choose the measurement category which provides you with the best type of information needed to address your prioritized list of issues. For critical high priority issues, consider selecting more than one measurement category. This will provide different types of measures and measurement information, allowing for more in-depth analysis.

1.4 SELECTING THE APPLICABLE MEASURES

The third step in selecting and specifying program measures is to actually choose the measures which best address the specific program issues. In most cases, selection of the applicable software measures will be accomplished in conjunction with the selection of the appropriate measurement categories. The overall objective is to define measures which not only adequately address the identified issues, but are practical to implement given the management and technical characteristics of the program.

As discussed in Part 1 of *PSM*, there are a number of criteria which have a direct bearing on which measures are selected. The most

important is how effective the measure is in addressing the specific issue in question. Other factors include:

- the **effectiveness** of the measure in addressing multiple issues
- the **applicability** of the measure within the program domain
- how well the measure is **supported** by existing program management practices
- the **availability** of the associated measurement data and the cost to collect it
- the applicability of the measure to the **life cycle phase** of the program
- the **usefulness** of the measure in addressing external reporting requirements
- the overall **size and scope** of the program, including the derivation and source of the software

The Measurement Description tables in Chapter 2 explain each of the *PSM* measures in detail. Each table includes criteria for determining whether or not to select the measure. This information is found in the measurement definition and selection guidance portions of the tables.

In most cases the selection process will require that tradeoffs be made among the measurement selection criteria. A given measure, for example, may directly address a high priority program issue, but be too costly to implement in terms of time and resources. Similarly, on a large program you may have to limit the number of measures that you apply in order to adequately address the high priority issues for all of the software. Some measures, when used in conjunction with other specific measures, support key analysis techniques. Milestone Dates, Labor Hours, and Lines of Code, for example, are the measures used to calculate and analyze software development performance in terms of productivity.

In general, measures from different measurement categories within the same common issues can be substituted with some degree of effectiveness. Also, measures which are categorized under different common issues can provide additional insight into the issue in

question. Obviously, it is better to use a substitute measure than to select a measure that cannot be implemented.

After the initial set of measures is selected, it should be reviewed to ensure that the high priority issues are addressed, and that there is adequate coverage across all of the identified issues. For some unique issues, none of the *PSM* measures may provide adequate information. In these cases, more advanced or different measures than those provided in the *PSM* guidance should be defined and specified. The bibliography contained in Part 6 provides potential sources for other measures.

1.5 SPECIFYING MEASUREMENT DATA AND IMPLEMENTATION REQUIREMENTS

The last step in the measurement selection and specification process is to define the data and implementation requirements for each of the selected measures. The *PSM* guidance that supports this activity is also included in the Chapter 2 Measurement Description tables in the Specification Guidance portion of the tables.

The tables include a list of the data items which are typically collected for each measure, the typical levels at which the data is collected and reported, the software activities to which the measure applies, and other pertinent information. The purpose of this information is to help identify those data and implementation requirements which should be considered in specifying the measure.

The specification information provided in a Measurement Description table is focused on the single measure being described. A set of general implementation requirements, applicable to all measures, is listed in a separate table in Chapter 2, and titled General Measurement Specification. After reviewing the individual measurement specification guidance on the individual tables for the selected measures, the specification guidance in the General Measurement Specification table should be addressed.

The general specification guidance outlines the requirements related to defining and collecting measurement data. These requirements help to define the overall measurement implementation approach on the program and help to convey to the developer how the measurement plan should be implemented. The general requirements include the following:

- **Data Types** - Measurement data representing plans, changes to plans, and actuals for each measure should be collected and reported. Plans and estimates should be updated regularly by the developer. Effective insight can be derived early in the program by analyzing how the planning data is changing. Extremely stable plans may indicate that the developer is not adjusting to actual program events. For many programs some plans and estimates are difficult to collect due to limitations in the software process. Not everyone, for example, can adequately project the number of expected problem reports to be found. In these cases trends based on the periodic actual data may be adequate to support the measurement analysis requirements.
- **Measurement Definitions** - During the integration portion of measurement tailoring, the developer identifies the actual measurement definitions and methodologies that will be used for each specified measure. These definitions sometimes vary over the course of the program, as software processes are modified and updated. Changes to the definition and interpretation of any measure should be defined by the developer and relayed to the program office. In many cases this information is included in the periodic delivery of the measurement data. For many measures, such as lines of code, the estimation methodologies and the way the actuals are counted may be different. This can sometimes result in variances between plans and actuals which are measurement, not performance related. Such estimation inconsistencies should be identified. Many measures require that both the estimation and actual counting methodologies be defined, as well as the “exit” criteria for measuring actuals. Definition of the measures is extremely important, as it provides the basis for correct interpretation of the associated data.
- **Data Dates** - For each measure, both the date that the measurement data was collected and the date that it is reported should be identified. This allows the timeliness of the data to be assessed, and supports the correlation of related measurement data during analysis. On a productivity calculation, for example, the time period during which the number of lines of code produced should correspond to the time period during which the labor hours used to produce them are counted. The difference between the date the data was collected and the date

the data was provided to the program office should be minimized. This allows for timely analysis and feedback on the issues.

- **Collection Periodicity** - Measurement data should be collected on a periodic, not event driven basis. This is generally monthly on most programs but can be adjusted as necessary. Problem report data, for example, is collected and reported on a weekly and sometimes daily basis during integration and test.
- **Measurement Scope** - If more than one organization is involved in developing the software for a program, measurement data should be collected from each and identified by source. This is usually the case when there are one or more software subcontractors working under a prime contractor. In many instances the individual organizations have very different software processes which result in different measurement definitions for the same measure. This prevents the combination and aggregation of some types of measurement data from the different organizations. In these cases the data from a given organization must be managed and analyzed separately. For example, a system level productivity calculation may be invalid if different subcontractors count labor hours and software size differently. In some cases, different measures will be used by different organizations to address similar issues.
- **Program Phase** - The measures which are selected and integrated into the program should generally be applied to all life cycle phases, including program planning, development, and software support. For most measures, the planning data will be available initially, followed by actual data as the program progresses and planned software process activities are implemented. Even when actual data is available, the related measurement plans and estimates should be continuously updated.
- **Data Reporting Mechanisms** - The reporting mechanisms for delivering data from the developer vary based upon the actual measures which are selected and the internal software and program management processes of both the developer and the program office. The data for many measures, such as problem reports, is usually available from an existing configuration management database which can be accessed on a real-time

basis. In other cases, such as with effort, size, and schedule measures, the data can easily be formatted into electronic media and delivered accordingly. Some data may need to be delivered in hard copy format. During the integration portion of the tailoring process, the developer identifies the mechanisms which are available. Every effort should be given to establishing the interfaces required to electronically transfer the data on a periodic basis.

- When the measures have been selected and the associated data and implementation requirements are complete, this information is conveyed to the development organization for integration into the software process. During integration, the developer suggests revisions to better align the program office's measurement requirements with the software process, and defines the actual measurement methodologies and data reporting mechanisms for each measure. This information essentially defines the program measurement plan.

The actual software measurement selection and specification process is dynamic. Although it is described as a step by step approach, many of the activities take place concurrently. The *PSM* guidance represents current practice, but is intended to be modified and augmented as required to meet new or unique program needs.

1.6 SELECTING AND SPECIFYING MEASURES FOR EXISTING PROGRAMS

The *PSM* measurement selection and specification guidance is generally structured to support a sequential tailoring of the measurement process. In some instances, the need to implement a measurement process is driven by a significant program event or issue which must to be immediately supported by objective software information. In other cases, new policy guidance or other external requirements such as a major milestone review may make it necessary to implement measurement on a program which has already been initiated.

If the measurement process is to be implemented on an existing program, the *PSM* selection and specification guidance can be used to help identify the measures and associated data which can usually be derived from the existing software process. The overall approach still begins with the identification and prioritization of the specific program issues. In all likelihood, key issues and problems

have already been identified, and the immediate objective is to identify what information can be used to analyze the issues and provide meaningful information to the Program Manager.

In general, most programs have some existing measures or data available which correspond to the following *PSM* Common Issues and associated Measurement Categories:

- **Schedule and Progress**
 - Milestone Performance
 - Schedule Performance
- **Resources and Cost**
 - Effort Allocation
 - Cost Performance
- **Growth and Stability**
 - Product Size and Stability
- **Product Quality**
 - Defect Profile

The *PSM* measurement description tables can be used to identify and define the measures and data available on an existing program. The measures which are required to address key issues but are not available can also be identified.

In general, the available measures on a program just implementing a measurement process are available at higher levels of detail. Software labor hours, for example, may only be available at a system or organizational level, and may not be allocated to the software component (i.e., CSCI) or activity (Design) levels. Similarly, software sizes may be available in terms of the number of components, and not in lines of code or function points. A significant amount of analysis can usually be performed with the existing data. As the measurement process is implemented, deficiencies with respect to required measures and collecting and reporting mechanisms can be identified and corrected.

The most important aspect of implementing the measurement process on an existing program is to start with the measures and data which are available while focusing on the application of a consistent approach for data collection, analysis, and reporting.

CHAPTER 2 – DETAILED MEASUREMENT SELECTION AND SPECIFICATION INFORMATION

Chapter 1 explained the *PSM* approach for selecting and specifying program measures, and the use of Common Software Issues, Measurement Categories, and Detailed Measurement Descriptions in the selection process. This chapter provides the detailed information needed to actually determine which measures to use, and to define the data and implementation requirements for the measures that you select. This information is contained in a set of comprehensive Measurement Category and Measurement Description tables. Tables are provided for each Common Software Issue. The information which comprises the Measurement Tables is derived from actual measurement experience on successful DoD programs.

2.1 INTRODUCTION

The Measurement Category and Measurement Description Tables provide the detailed information that you need to select and specify the measures for your program. The *PSM* selection and specification approach is based upon the direct relationship between program issues, information needs, and the specific measures which provide the required information. To implement this approach, *PSM* defines a simple mapping from the Common Software Issues to related Measurement Categories, and then to individual Measures. This mapping is depicted in 2-2. The structure of the Measurement Tables in this chapter follows this mapping, and guides the user in selecting first the measurement categories, and then the specific measures which address the identified program issues. After the measures are selected, the tables further provide the information that is required to specify the data and implementation requirements for each selected measure.

2.2 HOW TO USE THE MEASUREMENT TABLES

Two types of Measurement Tables are provided, Measurement Category Tables and individual Measurement Description Tables.

2.2.1 Measurement Category Tables

The Measurement Category Tables help you to determine if the measures in a specific category provide the type of information required to adequately address the issue in question. These tables should be reviewed for each Common Software Issue which has been identified as being relevant to the program. If the category provides the type of information that is required, the Measurement Description Tables within that category should then be reviewed to select specific measures. In most cases the Measurement Category Tables and the Measurement Description tables are reviewed concurrently.

Figure 2-3 is a “roadmap” to the information contained in a Measurement Category Table. The following is a description of the type of information provided in each section of the table. The information in each Measurement Category Table section applies to all measures within the category.

- **Measurement Category and Issue** - this section identifies the Measurement Category and the corresponding Common Software Issue.
- **Definition and Description** - this section provides a description of the types of measurement information provided by the measures which comprise the Measurement Category, and indicates how this measurement information is used.
- **Program Application** - this section provides information which helps to identify if the measures in the category are applicable to specific types of programs. The information addresses applicability with respect to functional domain, size, and life-cycle phase of the program.
- **Measures Included In this Category** - this section lists the measures which are included in the Measurement Category. In some cases this is a single measure.
- **Limitations** - this section addresses the general limitations of the measures in the category. The information helps to determine if the measures provide the type of measurement information that is required.

- **Related Measurement Categories** - this section references other *PSM* measurement categories which contain measures which are useful if implemented in conjunction with the measures in the current category. These related categories provide information which supports a more complete analysis of the issue in question.
- **Additional Information** - this section provides supplementary information which applies to the measures included in the Measurement Category. This information may define concepts or terms used in the measures or provide amplifying selection guidance. This section is not included for all Measurement Category tables.
- **Example Indicator** - Part 3 of *PSM* includes sample graphs of measurement indicators derived from selected measures in each of the 14 Measurement Categories. This section indicates which measure or measures from the category were selected for use in Part 3.

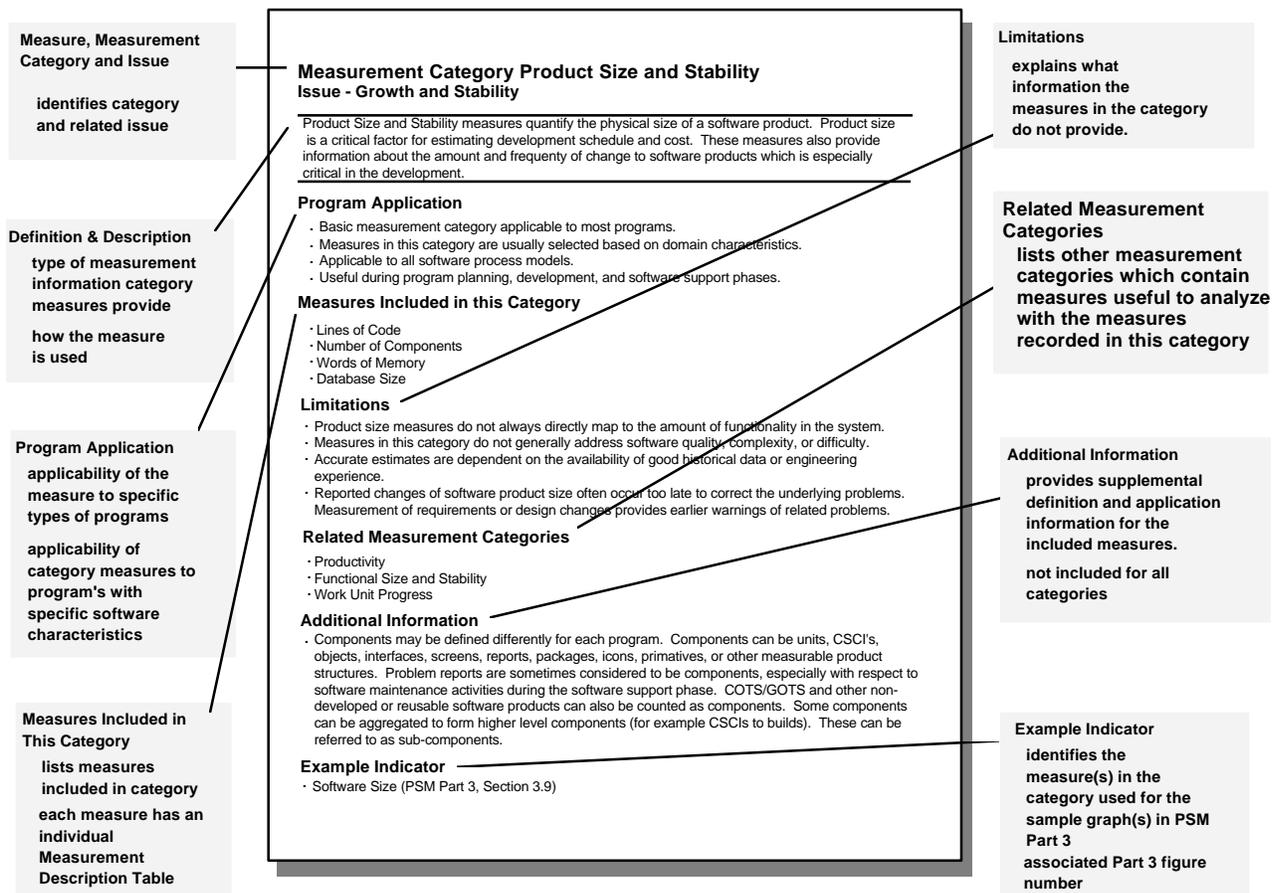


Figure 2-3 Measurement Category Table “Roadmap”

2.2.2 Measurement Description Tables

The Measurement Description Tables serve two purposes. As such, they contain two types of information. The first type of information is called selection guidance. This information helps to determine if a measure will effectively address an identified issue, and if the measure is applicable given the characteristics of the program and the nature of the associated management and technical processes. The tables also provide a second type of information called specification guidance. This information is used to define the specific data and implementation requirements for each selected measure.

Some specification guidance is common to all measures. Rather than repeat this information in every Measurement Description Table, it is summarized in a single General Measurement Specification Table. This table is a unique Measurement Description Table which applies to all measures. It is intended to be used in conjunction with each of the individual Measurement Description Tables when specifying measurement data and implementation requirements.

Figure 2-4 is a “roadmap” to the information contained in a Measurement Description Table. The following is a summary of the type of information provided in each section of the table.

- **Measure, Measurement Category, and Issue** - this section identifies the specific Measure, the associated Measurement Category, and the corresponding Common Software Issue.
- **Definition and Description** - this section provides a definition of the measure and a description of the measurement information that it provides. It also explains how the measure is used, and how effective the measure is in addressing related issues. This section addresses both selection and specification guidance.
- **Program Application** - this section provides selection guidance information which helps to identify if the measure is applicable to specific types of programs. The information addresses applicability with respect to functional domain,

software size, scope, type and origin, and life-cycle phase of the program. It specifically addresses application of the measure to real-time, data intensive, and other systems. It also identifies the life cycle phases in which the measure is most useful. The overall use of the measure within the DoD and industry is also addressed.

- **Process Integration** - this section provides selection guidance information which helps to determine if the measure is applicable to specific program and technical management processes. The information addresses particular program management practices, data availability and cost, and other process characteristics.
- **Usually Applied During** - this section provides selection guidance information which identifies if the measure is applicable to a particular software process activity. These activities are defined to include requirements analysis, design, implementation, and integration and test. These activities can take place during any phase of the software life cycle, and can occur concurrently during the same phase. They should not be construed to be sequential in nature. The information in this section also addresses the type of data (estimates or actuals) which is generally available with respect to the identified software activities.
- **Data Items - Additional Information (Optional)**- this section provides additional information to help specify data items.
- **Data Items Typically Collected** - this section provides specification guidance which identifies the attributes of the measures which are typically measured and collected.
- **Typical Collection Level** - this section provides specification guidance which identifies the software activity and design levels at which the developer typically collects the data items for the measure.
- **Typical Reporting Level** - this section provides specification guidance which identifies the software activity and design levels at which the data elements are reported by the developer. These are not necessarily the same as the collection levels.

- **Count Actuals Based On** - this section provides specification guidance which identifies typical activities or exit criteria for the listed data elements. This information helps to determine when a measure is counted as an actual, or when an activity or event is complete. Normally only one of these options is specified.
- **This Measure Answers Questions Such As** - this section lists the typical questions that are addressed by the measure.

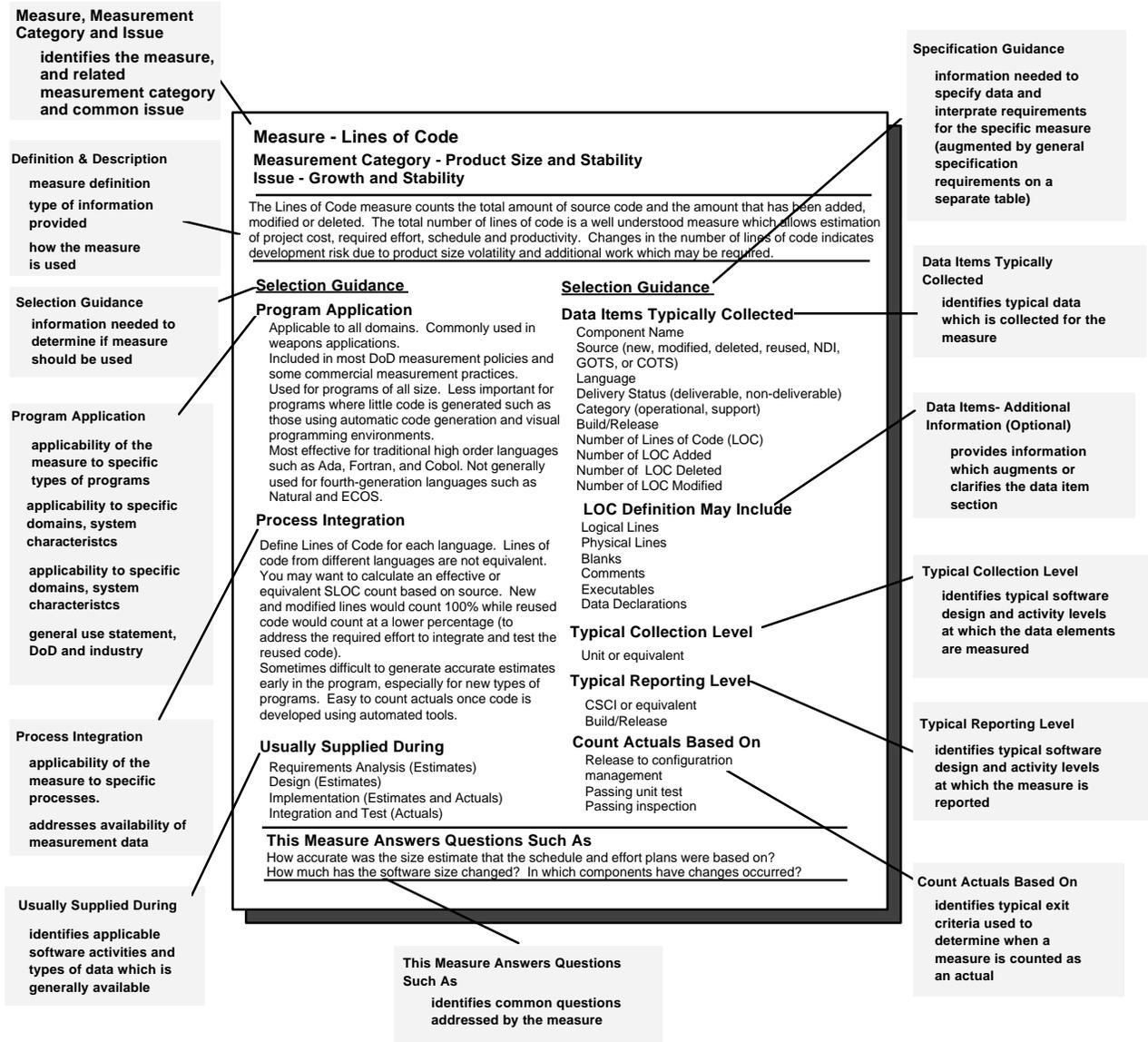


Figure 2-4. Measurement Description Table

2.2.3 General Measurement Specification Table

The General Measurement Specification Table should be used in conjunction with the individual Measurement Description tables when specifying the data and implementation requirements for measures which have been selected for implementation. The included specification guidance applies to all measures in all Measurement Categories under all Common Issues. It summarizes the general specification requirements described in Section 1.5 of this part of the Guide.

2.2.4 Additional Implementation Guidance

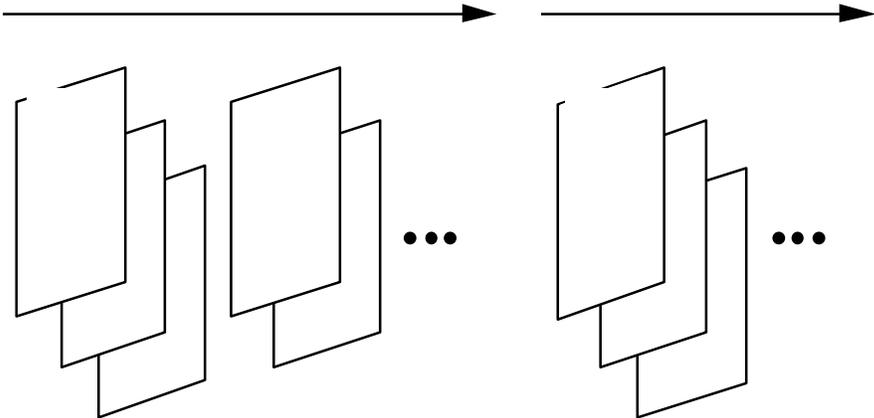
Most of the measures listed in the Measurement Description Tables are basic measures which quantify a single software attribute. Some of the measures, such as those which fall under the common issues of Product Quality and Development Performance, are actually composite measures which are derived using measures which are listed elsewhere. Productivity, for example, is a composite Development Performance measure which is calculated using the Product Size measure under the Growth and Stability issue and the Labor Hours measure under the Resources and Cost issue. The composite measures, which also include Defect Density, are included on separate tables since they are widely used to address different issues than the more basic related measures.

There are no specific measures listed in the Technical Impact category under the common issue of Technical Adequacy. Measures in this category should be defined on a program by program basis to provide insight into the software technologies and processes which are critical to the success of the program. Measures in this category are generally defined to track those software technologies which are highly leveraged. Many of the Technical Adequacy measures are derivatives of measures categorized under other common issues. For example, if a program's planned cost and schedule is based upon large increases in development productivity due to the use of a substantial amount of reused software, a measure could be defined which provides information on the relative growth of the reused vs. the newly developed code. Growth in the new code with concurrent reductions in the planned amount of reused code may indicate that

the reused code may not satisfy the requirements as expected, and that the actual productivity may be much less than anticipated.

2.2.5 Measurement Selection and Specification Tables

Measurement categories are grouped by the six common issues. Each category table is followed immediately by the tables for its constituent measures.



Measurement Category General Measurement Specification

Measurement Category - All

Issue - All

This table provides measurement specification guidance applicable to all measures, and augments the guidance found in the individual Measurement Description Tables. It provides information which helps to define overall data and implementation requirements for all selected measures.

Specification Guidance

- **Data Items** - The data elements for each selected measure and the levels of collection and reporting for each should be identified.
- **Data Types** - Measurement data representing plans, changes to plans, and actuals for each measure should be collected and reported. Plans and estimates should be updated on a regular basis.
- **Measurement Definitions**- The developer should identify the actual measurement definitions and methodologies that will be used for each specified measure. If these change over the course of the program, the definitions and associated interpretations should be updated and provided to the program office. Differences in the estimation methodologies and the way the actuals are counted for each individual measure should be identified. The “exit” criteria for counting actuals should be defined for each measure.
- **Data Dates** - For each measure, but the date that the measurement data was collected and the date that it is reported should be identified. The data should be provided in a timely manner. The difference between the date the data was collected and the date the data was provided to the program office should be minimized.
- **Collection Periodicity**- Measurement data should be collected on a periodic, not event driven basis. This is generally monthly on most programs but can be adjusted as necessary. The periodicity may have to be modified for selected measures due to software process constraints.
- **Measurement Scope**- If more than one organization is involved in developing the software for a program, measurement data should be collected from each and identified by source. Different definitions for the same measures should be identified.
- **Program Phase**- The measures which are selected and integrated into the program should generally be applied to all life cycle phases, including program planning, development, and software support. Throughout all phases measurement plans and estimates should be continuously updated and reported.
- **Data Reporting Mechanisms**- The reporting mechanisms for delivering data to the program office from the developer should be identified for each measure. Every effort should be given to establishing the interfaces required to electronically transfer the data on a periodic basis.

Measurement Category - Milestone Performance Issue - Schedule and Progress

The Milestone Performance measures provide basic schedule and progress information for key software development activities and events. The measures also help to identify and assess dependencies among software development activities and events. Monitoring changes in schedules allows the program manager to assess the risk in achieving future milestones.

Program Application

- Basic measurement category applicable to most programs.
- Applicable to all software process models.
- Useful during program planning, development, and software support phases.

Measures Included in this Category

- Milestone Dates

Limitations

- The measures in this category do not address the degree of individual activity completion or the amount of effort to complete a scheduled activity or task.
- These measures do not address the relative importance of key activities (except for the identification of critical path activities).

Related Measurement Categories

- Work Unit Progress
- Productivity
- Schedule Performance

Example Indicator

- Milestone Dates (PSM Part 3, Section 3.1)

Measure - Milestone Dates

Measurement Category - Milestone Performance

Issue - Schedule and Progress

The Milestone Dates measure consists of the start and end dates for software activities and events. The measure provides an easy to understand view of the status of scheduled software activities and events. Comparison of plan and actual milestone dates provides useful insight into both significant and repetitive schedule slips at the software activity level.

Selection Guidance

Program Application

- Basic measure applicable to all domains.
- Included in most DoD measurement policies and commercial measurement practices.
- Generally applicable to all sizes and types of programs.
- Useful during program planning, development, and software support phases. Some software support programs may be considered level of effort tasks and may not have associated milestones.

Process Integration

- Required data is generally easily obtained from program scheduling systems and/or documentation. Data should be focused on software activities and events, particularly those affecting the critical path or risk items.
- If dependency data is collected, slips in related activities can be more easily and accurately projected and assessed.

Usually Applied During

- Requirements Analysis (Estimates and Actuals)
- Design (Estimates and Actuals)
- Implementation (Estimates and Actuals)
- Integration and Test (Estimates and Actuals)

Specification Guidance

Data Items Typically Collected

- Activity or Event Name
- Component Name
- Start Date
- End Date
- Build/Release
- Responsible Organization
- Dependent Activity or Event Name

Typical Collection Level

- CSCI or equivalent
- Key software activities and events

Typical Reporting Level

- CSCI or equivalent
- Key software activities and events
- Build/Release

Count Actuals Based On

- Customer sign-off
- Action items closed
- Documents baselined
- Milestone review held
- Successful completion of tasks

This Measure Answers Questions Such As

- Is the current schedule realistic?
 - How many activities are concurrently scheduled?
 - How often has the schedule changed?
-

Measurement Category - Work Unit Progress

Issue - Schedule and Progress

Work Unit Progress measures address progress based on the completion of work units that combine incrementally to form a complete software activity or product. If objective completion criteria are defined, Work Unit Progress measures are extremely effective for assessing progress at any point in the program. They are also useful for projecting completion dates for the activity or product.

Program Application

- Basic measurement category applicable to most programs.
- Applies to all software process models.
- Useful during development and software support phases.

Measures Included in this Category

- Components Designed
- Components Implemented
- Components Integrated and Tested
- Requirements Allocated
- Requirements Tested
- Test Cases Completed
- Paths Tested
- Problem Reports Resolved
- Reviews Completed
- Changes Implemented

Limitations

- These measures do not weight difficult or critical activities or products. All activities are usually assumed to be of the same level of importance.

Related Measurement Categories

- Milestone Performance
- Effort
- Product Size and Stability
- Functional Size and Stability

Additional Information

- Components may be defined differently for each program. Components can be units, CSCIs, objects, interfaces, screens, reports, packages, icons, primitives, or other measurable product structures. Problem reports are sometimes considered to be components, especially with respect to software maintenance activities during the software support phase. COTS/GOTS and other non-developed or reusable software products can also be counted as components. Some components can be aggregated to form higher level components (for example, units to CSCIs to builds). These can be referred to as sub-components.

Example Indicator

- Design Progress (PSM Part 3, Section 3.2)

Measure - Components Designed

Measurement Category - Work Unit Progress

Issue - Schedule and Progress

The Components Designed measure counts the number of software components that have completed preliminary or detailed design. Early in design, planing changes should be expected as the design matures. Later in the process, an increase in the planned number of components can be an indication of unplanned or excessive growth. A comparison of planned and actual components is very effective for assessing design progress.

Selection Guidance

Program Application

- Applicable to all domains.
- Used on medium to large programs.
- Useful during development and software support phases. Not generally used on programs without a design activity such as software support programs which are focused on problem resolution or COTS integration programs.

Process Integration

- Easier to collect if formal reviews, inspections, or walkthroughs are included in the development process.
- Data sometimes available from configuration management systems or design tools.
- Data is generally available if there is a mature and disciplined design process

Usually Applied During

- Requirements Analysis (Estimates)
- Design (Estimates and Actuals)

Specification Guidance

Data Items Typically Collected

- Component Name
- Number of Sub-Components Completing Design
- Build/Release

Typical Collection Level

- Unit or equivalent

Typical Reporting Level

- CSCI or equivalent
- Build/Release

Count Actuals Based On

- Completion of component design reviews, inspections, or walkthroughs
- Release to configuration management
- Resolution of action items

This Measure Answers Questions Such As

- Are components completing design as scheduled?
 - Is the planned rate of design completion realistic?
 - What components are behind schedule?
-

Measure - Components Implemented

Measurement Category - Work Unit Progress

Issue - Schedule and Progress

The Components Implemented measure counts the number of software components that have been coded or modified and have completed unit test. An increase in the number of planned components is an indication of unplanned or excessive growth. A comparison of planned and actual components is one of the most effective measures of implementation progress.

Selection Guidance

Program Application

- Applicable to all domains.
- Used on medium to large programs.
- Useful during development and software support phases. Not generally used on programs without a implementation activity such as COTS integration programs or programs using automatic code generation.

Process Integration

- Data sometimes available from the configuration management system.
- Easy to collect, data is generally available.

Usually Applied During

- Design (Estimates)
- Implementation (Estimates and Actuals)

Specification Guidance

Data Items Typically Collected

- Component Name
- Number of Sub-Components Implemented
- Build/Release

Typical Collection Level

- Unit or equivalent

Typical Reporting Level

- CSCI or equivalent
- Build/Release

Count Actuals Based On

- Passing inspection
- Passing unit test
- Release to configuration management

This Measure Answers Questions Such As

- Are components implemented as scheduled?
 - Is the planned rate of implementation progress realistic?
 - What components are behind schedule?
-

Measure - Components Integrated and Tested

Measurement Category - Work Unit Progress

Issue - Schedule and Progress

The Components Integrated and Tested measure counts the number of software components that have been successfully integrated and tested. The measure is an indication of component integration and test progress.

Selection Guidance

Program Application

- Applicable to all domains.
- This measure is important when integrating COTS and reusable software components.
- Useful during development and software support phases.

Process Integration

- Requires a disciplined testing process with separate tests per component(s) allocated to defined test sequences.
- Can be applied for each unique test sequence (i.e. CSCI test, integration test, system test), including “dry-runs”
- Generally one of the more difficult work unit progress measures to collect since most integration and test activities are based on requirements or functions instead of components.

Usually Applied During

- Implementation (Estimates)
- Integration and Test (Estimates and Actuals)

Specification Guidance

Data Items Typically Collected

- Component Name
- Number of Sub-Components Integrated and Tested
- Number of Sub-Components Integrated and Tested Successfully
- Test Sequence Name
- Build/Release

Typical Collection Level

- Unit or equivalent

Typical Reporting Level

- CSCI or equivalent
- Test Sequence
- Build/Release

Count Actuals Based On

- Successfully passing all component tests in the appropriate test sequence

This Measure Answers Questions Such As

- Are components completing integration and test as scheduled?
 - Is the planned rate of integration and test completion realistic?
 - What components are behind schedule?
-

Measure - Requirements Allocated

Measurement Category - Work Unit Progress

Issue - Schedule and Progress

The Requirements Allocated measure counts the number of defined requirements which have been allocated to software design components and test cases. The measure is an indication of software design progress.

Selection Guidance

Program Application

- Applicable to all domains.
- Useful during development and software support phases. Not generally used on programs without a requirements analysis or design activity such as software support programs which are focused on problem resolution.

Process Integration

- Not all requirements are directly testable. Some are verified by inspection.
- There may not be a direct relationship between design components and test cases. Requirements may need to be allocated separately.
- Requires a good requirements traceability process. If an automated design tool is used, the data is more readily available.
- This is normally difficult to collect. It often requires a lot of manual effort.

Usually Applied During

- Requirements Analysis (Estimates)
- Design (Estimates and Actuals)

Specification Guidance

Data Items Typically Collected

- Requirements Identifier
- Design Component Name
- Test Case Identifier

Typical Collection Level

- Requirement

Typical Reporting Level

- CSCI or equivalent

Count Actuals Based On

- Completion of specification review
- Baselineing of specifications
- Baselineing of Requirements Traceability Matrix

This Measure Answers Questions Such As

- Have all of the requirements been allocated to software design components?
 - Which requirements are validated by which tests?
 - How many requirements are directly testable?
-

Measure - Requirements Tested

Measurement Category - Work Unit Progress

Issue - Schedule and Progress

The Requirements Tested measure counts of the number of stated and derived requirements that have been successfully tested. The measure addresses the degree to which required functionality has been successfully demonstrated against the specified requirements, as well as the amount of testing that has been performed. This measure provides an excellent measure of test progress. This measure is also known as "Breadth of Testing".

Selection Guidance

Program Application

- Applicable to all domains.
- Generally applicable to all sizes and types of programs, except those in which requirements cannot be traced to test cases.
- Useful during development and software support phases.

Process Integration

- Requires disciplined requirements traceability and testing processes to implement successfully. Allocated requirements should be testable and mapped to tests.
- Can be applied for each unique test sequence (i.e. CSCI test, integration test, system test), including "dry-runs".
- One of the more difficult work unit progress measures to collect since requirements often do not map to test procedures. It is also sometimes difficult to objectively determine if a requirement has been successfully tested.
- Some requirements may not be testable until late in the testing process. Others are not directly testable.

Usually Applied During

- Implementation (Estimates)
- Integration and Test (Estimates and Actuals)

Specification Guidance

Data Items Typically Collected

- Number of Requirements
- Number of Requirements Tested
- Number of Requirements Tested Successfully
- Test Sequence Name
- Build/Release

Typical Collection Level

- Requirement

Typical Reporting Level

- Test Sequence
- Build/Release

Count Actuals Based On

- Successful completion of all tests in the appropriate test sequence

This Measure Answers Questions Such As

- Are the requirements being tested as scheduled?
 - Has the testing been successful?
 - What requirements (functions) are behind schedule?
 - How much of the functionality has been tested?
-

Measure - Test Cases Completed

Measurement Category - Work Unit Progress

Issue - Schedule and Progress

The Test Cases Completed measure counts the number of test cases that have been attempted and those that have been completed successfully. This measure can be used in conjunction with the Requirements Tested measure to evaluate test progress. This measure allows assessment of software quality, based on the proportion of attempted test cases that are successfully executed. This measure is one of the best measures of test progress.

Selection Guidance

Program Application

- Applicable to all domains.
- Generally applicable to all sizes and types of programs.
- Useful during development and software support phases.

Process Integration

- Need disciplined test planning and tracking processes to implement successfully.
- Can be applied for each unique test sequence (i.e. CSCI test, integration test, system test), including “dry-runs”.
- There should be a mapping between defined test cases and requirements. This allows an analysis of which functions are passing test and which are not.
- Easy to collect. Most programs define and allocate a quantifiable number of test cases to each software test sequence.

Usually Applied During

- Implementation (Estimates and Actuals)
- Integration and test (Estimates and Actuals)

This Measure Answers Questions Such As

- Is test progress sufficient to meet the schedule?
 - Is the planned rate of testing realistic?
 - What functions are behind schedule?
-

Specification Guidance

Data Items Typically Collected

- Test Sequence Name
- Number of Test Cases
- Number of Test Cases Attempted
- Number of Test Cases Passed
- Build/Release

Alternates to Test Cases Include

- Test Procedures
- Test Steps

Typical Collection Level

- Test Case

Typical Reporting Level

- Test Sequence
- Build/Release

Count Actuals Based On

- Successful completion of each test case in the appropriate test sequence

Measure - Paths Tested

Measurement Category - Work Unit Progress

Issue - Schedule and Progress

The Paths Tested measure counts the number of logical paths successfully tested. The measure reports the degree to which the software has been successfully demonstrated and indicates the amount of testing that has been performed. This measure is also called "Depth of Testing".

Selection Guidance

Program Application

- Applicable to all domains.
- Applicable to most types of programs. Especially important for those with high reliability requirements, security implications, or catastrophic failure potential.
- Not generally used for COTS or reused code.
- Useful during development and software support phases.

Process Integration

- Usually applied on a cumulative basis across all test sequences (i.e. CSCI test, integration test, system test).
- Often used in conjunction with Cyclomatic Complexity.
- Difficult to collect - requires the use of test tools that can verify test paths covered. These test tools often require instrumentation of the code.
- Difficult to use on large programs due to the large number of paths.

Usually Applied During

- Implementation (Estimates and Actuals)
- Integration and Test (Actuals)

Specification Guidance

Data Items Typically Collected

- Component Name
- Number of Paths
- Number of Paths Tested
- Number of Paths Tested Successfully
- Test Sequence
- Build/Release

Alternatives to Paths Include

- Executable Statements
- Decisions

Typical Collection Level

- CSCI or equivalent

Typical Reporting Level

- CSCI or equivalent
- Test Sequence
- Build/Release

Count Actuals Based On

- Successful completion of each test in the appropriate test sequence

This Measure Answers Questions Such As

- Have all of the paths been successfully tested?
 - What are the minimum number of test cases required to completely test the software?
 - What percentage of the paths are represented in the testing approach?
-

Measure - Problem Reports Resolved

Measurement Category - Work Unit Progress

Issue - Schedule and Progress

The Problem Reports Resolved measure counts the number of software problems reported and resolved. This measure provides an indication of product maturity and readiness for delivery. The rates at which problem reports are written and closed can be used in a straight-line estimate of test completion. This measure can also be used as an indication of the efficiency of the problem resolution process.

Selection Guidance

Program Application

- Applicable to all domains.
- Applicable to all sizes and types of programs.
- Useful during development and software support phases.

Process Integration

- Many programs have acceptance criteria based on the number of open problem reports, by priority. This measure is useful in tracking to those requirements.
- The amount of test activity has a significant impact on this measure. Test personnel generally alternate between testing and fixing problems. You may want to normalize this measure using some measure of Test Progress.
- Data is generally available. Data is easier to collect when an automated problem tracking system is used.
- On development programs, data is generally available during integration and test. Problem report data is more difficult to collect earlier (during requirements analysis, design, and implementation), because the formal problem reporting system is usually not in place and rigidly enforced. When this data is available, it provides very good progress information.

Usually Applied During

- Requirements Analysis (Estimates and Actuals)
- Design (Estimates and Actuals)
- Implementation (Estimates and Actuals)
- Integration and Test (Estimates and Actuals)

This Measure Answers Questions Such As

- Are known problem reports being closed at a sufficient rate to meet the test completion date?
 - Is the product maturing (Is the problem report discovery rate going down)?
 - When will testing be complete?
 - What components have the most open problem reports?
-

Specification Guidance

Data Items Typically Collected

- Component Name
- Priority
- Number of Software Problems Reported
- Number of Software Problems Resolved

Typical Collection Level

- CSCI or equivalent

Typical Reporting Level

- CSCI or equivalent

Count Actuals Based On

- Fix developed
- Fix implemented
- Fix integrated
- Fix tested

Measure - Reviews Completed

Measurement Category - Work Unit Progress

Issue - Schedule and Progress

The Reviews Completed measure counts the number of reviews successfully completed, including both internal developer and program manager reviews. The measure provides an indication of progress in completing review activities.

Selection Guidance

Program Application

- Applicable to all domains.
- Used on medium to large programs. Not generally used on COTS and reusable software components.
- Useful during development and software support phases.

Process Integration

- Easy to collect if formal reviews are a part of the development process.

Usually Applied During

- Requirements Analysis (Estimates and Actuals)
- Design (Estimates and Actuals)
- Implementation (Estimates and Actuals)

Specification Guidance

Data Items Typically Collected

- Component Name
- Number of Reviews
- Number of Reviews Scheduled
- Number of Reviews Completed Successfully
- Build/Release

Alternatives to Reviews Include

- Inspections
- Walkthroughs

Typical Collection Level

- Unit or equivalent

Typical Reporting Level

- CSCI or equivalent
- Build/Release

Count Actuals Based On

- Completion of review
- Resolution of all associated action items

This Measure Answers Questions Such As

- Are development review activities progressing as scheduled?
 - Do the completed products meet the defined standards (Are components passing the reviews)?
 - What components have failed their review?
-

Measure - Changes Implemented

Measurement Category - Work Unit Progress

Issue - Schedule and Progress

The Changes Implemented measure counts the number of change requests affecting a product. The measure provides an indication of the amount of rework required and performed. It only identifies the number of changes, and does not report on the functional impact of changes or the amount of effort required to implement them.

Selection Guidance

Program Application

- Applicable to all domains.
- Applicable to all sizes of programs. Useful during the development phase. Not generally used for integration programs incorporating COTS and reused code. Often used for programs in the software support phase.

Process Integration

- Often used on iterative developments such as prototyping.
- Data should be available from most programs .

Usually Applied During

- Requirements Analysis (Actuals)
- Design (Actuals)
- Implementation (Actuals)
- Integration and Test (Actuals)

Specification Guidance

Data Items Typically Collected

- Priority
- Number of Software Change Requests Reported
- Number of Software Change Requests Completed

Options to Change Requests

- Enhancements
- Corrective Action Reports

Typical Collection Level

- Change Request

Typical Reporting Level

- System

Count Actuals Based On

- Change implemented
- Change integrated
- Change tested

This Measure Answers Questions Such As

- How many change requests have impacted the software?
 - Are change requests being implemented at a sufficient rate to meet schedule?
 - Is the trend of new change requests decreasing as the program nears completion?
-

Measurement Category - Schedule Performance

Issue - Schedule and Progress

Schedule Performance measures address earned value by comparing the budgeted cost of work performed to the budgeted cost of work scheduled. These measures can be used to identify critical path issues, schedule conflicts, or potential cost overruns.

Program Application

- Measurement category applicable to most programs.
- Applies to all software process models.
- Used during development and software support phases.

Measures Included in this Category

- Schedule Variance

Limitations

- Schedule progress is a product of the validity of the schedule, the availability of funding and other resources, and personnel resources. These factors are not identified by schedule performance measures. Other measurement categories provide better information about software schedule and progress.
- Schedule performance systems can be difficult to establish for software. A detailed software WBS must be developed that includes quantifiable exit criteria.
- Measurement of software performance is often difficult, due to insufficient detail in the software Work Breakdown Structure (WBS) and associated problems with reporting of actual progress.

Related Measurement Categories

- Milestone Performance
- Cost Performance
- Effort

Example Indicator

- Schedule Variance (PSM Part 3, Section 3.3)

Measure - Schedule Variance

Measurement Category - Schedule Performance

Issue - Schedule and Progress

The Schedule Variance measure is the difference between the budgeted cost of work performed and the budgeted cost of work scheduled, for each WBS element. The measure reports schedule progress, in terms of variance from original cost earned value estimates.

Selection Guidance

Program Application

- Applicable to all domains.
- Applicable to any program that uses an earned value cost accounting system. The DoD defined Cost/Schedule Control System Criteria (C/SCSC) apply to programs based on size and cost.
- Useful during development and software support phases.
- Limited in applicability if costs are planned and expended on a level of effort basis.

Process Integration

- C/SCSC data is required on most large DoD programs, so it is often readily available. This data should be based on a validated cost accounting system.
- This can be difficult to track without an automated system tied to the accounting function.
- This data tends to lag other measurement information due to formal reporting requirements.

Usually Applied During

- Requirements Analysis (Estimates and Actuals)
- Design (Estimates and Actuals)
- Implementation (Estimates and Actuals)
- Integration and Test (Estimates and Actuals)

This Measure Answers Questions Such As

- Is the schedule being met?
 - How far ahead/behind schedule is the program?
 - What WBS elements or tasks are behind/ahead of schedule?
-

Specification Guidance

Data Items Typically Collected

- WBS or Task Element
- Budgeted Cost of Work Scheduled (BCWS)
- Budgeted Cost of Work Performed (BCWP)

Typical Collection Level

- WBS or task element

Typical Reporting Level

- WBS or task element

Count Actuals Based On

- WBS element complete (to defined exit criteria)
- WBS element percent complete (based on engineering judgment)
- WBS element percent complete (based on underlying objective measures)

Measurement Category - Incremental Capability Issue - Schedule and Progress

Incremental Capability measures count the functional or product content associated with each incremental delivery. An incremental delivery may be a product shipped to a customer or it may be an internal build delivered to the next phase of development. These measures are used to determine if capability is being developed as scheduled or being delayed to future deliveries.

Program Application

- Measurement category applicable to programs that have multiple deliveries.
- Applies to software process models based on incremental development.
- Useful during development and software support phases.

Measures Included in this Category

- Build Content - Component
- Build Content - Function

Limitations

- Incremental software development often results in release of software with incomplete functions. It is sometimes difficult to determine if all of the planned capability is completed in any given increment.
- Requires a straightforward mapping of function or component to the increment. Difficult to collect and assess if measured components or functions are partitioned across increments.

Related Measurement Categories

- Product Size and Stability
- Functional Size and Stability
- Productivity

Example Indicator

- Incremental Build Content (PSM Part 3, Section 3.4)

Measure - Build Content - Component

Measurement Category - Incremental Capability

Issue - Schedule and Progress

The Build Content - Component measure identifies the components which are included in incremental builds. The measure indicates progress in the incremental products. Build content will often be deferred or removed in order to preserve the scheduled delivery date. It is easier to track incorporation of capability by component (rather than by function), since it is relatively easy to specify whether or not a component has been integrated. However, this provides less information, since the correlation between components and functionality is not always well defined.

Selection Guidance

Program Application

- Applicable to all domains.
- Generally applicable to all sizes and types of programs.
- Useful during development and software support phases.

Process Integration

- Requires a formal, detailed list of content by increment. This content must be defined at the component level.
- Easy to collect, especially if the program has a detailed tracking mechanism.
- To effectively measure the content of the software at the components build/release level, the lower level subcomponents which comprise the build/release must individually be complete with respect to defined criteria.

Usually Applied During

- Implementation (Estimates)
- Integration and Test (Estimates and Actuals)

Specification Guidance

Data Items Typically Collected

- Build/Release
- Component Name
- Number of Sub-Components
- Number of Sub-Components Integrated Successfully

Typical Collection Level

- Unit or equivalent

Typical Reporting Level

- CSCI or equivalent
- Build/Release

Count Actuals Based On

- Successful integration
- Successful testing

This Measure Answers Questions Such As

- Are components being incorporated as scheduled?
 - Will each increment contain the specified components?
 - Which components have to be deferred or eliminated?
 - What components have been added?
 - Is development risk being deferred?
-

Measure - Build Content - Function

Measurement Category - Incremental Capability

Issue - Schedule and Progress

The Build Content - Function measure identifies the content of incremental builds. The measure indicates the progress in the incorporation of incremental functionality. Build content will often be deferred or removed in order to preserve the scheduled delivery date.

Selection Guidance

Program Application

- Applicable to all domains.
- Generally applicable to all sizes and types of programs.
- Useful during development and software support phases.

Process Integration

- Requires a formal, detailed list of functions by increment.
- Feasible to collect if the program has a detailed tracking mechanism.
- It is often difficult to identify whether a function is incorporated in its entirety. A considerable amount of testing and analysis must be done to determine if all aspects of a function are incorporated.

Usually Applied During

- Design (Estimates)
- Implementation (Estimates)
- Integration and Test (Estimates and Actuals)

Specification Guidance

Data Items Typically Collected

- Build/Release
- Function Name
- Number of Sub-Functions
- Number of Sub-Functions Integrated Successfully

Typical Collection Level

- Function or equivalent

Typical Reporting Level

- Function or equivalent
- Build/Release

Count Actuals Based On

- Successful integration
- Successful testing

This Measure Answers Questions Such As

- Is functionality being incorporated as scheduled?
 - Will each increment contain the specified functionality?
 - Which functionality has to be deferred?
-

Measurement Category - Effort Profile

Issue - Resources and Cost

Effort Profile measures identify the amount of effort expended on defined software activities or products over time. These measures may be used to assess the adequacy of planned effort and analyze the actual allocation of labor. They are essential to evaluating software development productivity. These measures are especially critical since software is a very labor intensive process.

Program Application

- Basic measurement category applicable to most programs.
- Applicable to all software process models.
- Useful during program planning, development, and software support phases.

Measures Included in this Category

- Effort

Limitations

- The utility and timeliness of the measures are generally limited by the structure and capabilities of the financial system, which may be difficult to change.
- Measures are not always available at lower levels of product and activity detail.
- Actual effort, especially uncompensated overtime, may not be reported.

Additional Information

- Software activities typically include system engineering, software engineering, system design, software design, software documentation, coding, unit test, CSCI integration and test, build/release integration and test, system integration and test, software integration and test, software program management, configuration management, and quality assurance.

Example Indicator

- Effort Allocation (PSM Part 3, Section 3.5)

Measure - Effort

Measurement Category - Effort Profile

Issue - Resources and Cost

The Effort measure counts the number of hours of effort applied to software tasks. This is a straightforward, generally understood measure. It can be categorized by activity as well as by product. This measure usually correlates directly with software cost, but can also be used to address other common issues including Schedule and Progress and Development Performance.

Selection Guidance

Program Application

- Basic measure applicable to all domains.
- Included in most DoD measurement policies and commercial measurement practices.
- Generally applicable to all sizes and types of programs.
- Useful during program planning, development, and software support phases. Some software support programs with fixed staffing levels may not track this measure.

Process Integration

- Data usually derived from a financial accounting and reporting system and/or separate time card system.
- All labor hours applied to the software tasks should be collected, including overtime. The overtime data is sometimes difficult to collect.
- Most effective when financial accounting and reporting systems are directly tied to software products and activities at a low level of detail.
- If labor hours are not explicitly provided, data may be approximated from staffing and/or cost data. Labor hours are sometimes considered proprietary data.
- The labor categories and activities that comprise the software tasks must be explicitly defined for each organization.
- Labor Hours may also be reported as Days, Weeks, or Months with associated conversions.

Usually Applied During

- Requirements Analysis (Estimates and Actuals)
- Design (Estimates and Actuals)
- Implementation (Estimates and Actuals)
- Integration and Test (Estimates and Actuals)

This Measure Answers Questions Such As

- Are development resources being applied according to plan?
 - Are certain tasks or activities taking more/less effort than expected?
 - Is the effort profile realistic?
-

Specification Guidance

Data Items Typically Collected

- Organization
- WBS or Task Element
- Labor Category
- Number of Labor Hours

Typical Collection Level

- WBS or task element

Typical Reporting Level

- WBS or task element
- Organization

Count Actuals Based On

- End of financial reporting period

Measurement Category - Staff Profile Issue - Resources and Cost

Staff Profile measures characterize the number and experience of personnel assigned to a program. These measures also can be used to evaluate the rate at which people are added and removed from a program.

Program Application

- Measurement category applicable to most programs.
- Applies to all software process models.
- Useful during program planning, development, and software support phases.

Measures Included in this Category

- Staff Level
- Staff Experience
- Staff Turnover

Limitations

- Measures may not capture the total effort applied to a program because they do not distinguish between full and part-time personnel. Effort Allocation provides a more precise indicator of total effort applied.

Related Measurement Categories

- Effort
- Milestone Performance

Example Indicator

- Staff Experience (PSM Part 3, Section 3.6)

Measure - Staff Level

Measurement Category - Staff Profile

Issue - Resources and Cost

The Staff Level measure counts the total number of personnel allocated to software related activities. The measure is used to determine if sufficient personnel are available. It can also provide an early indication of possible schedule slips and cost overruns or underruns.

Selection Guidance

Program Application

- Applicable to all domains.
- Generally applicable to all sizes and types of programs.
- Useful during program planning, development, and software support phases. Some software support programs with fixed staffing levels may not track this measure.

Process Integration

- Data should be available from most programs. The Labor Hours measure provides more detailed information.
- Total staffing is generally available from most programs at the system level. Counting software personnel may be difficult because they may not be allocated to the project on a full-time basis or they may not be assigned to strictly software related tasks.

Usually Applied During

- Requirements Analysis (Estimates and Actuals)
- Design (Estimates and Actuals)
- Implementation (Estimates and Actuals)
- Integration and Test (Estimates and Actuals)

This Measure Answers Questions Such As

- Are sufficient development resources available and applied?
 - Are certain activities or functions taking more staff than expected?
-

Specification Guidance

Data Items Typically Collected

- Organization
- Activity
- Labor Category
- Number of Personnel

Typical Collection Level

- Activity

Typical Reporting Level

- Organization

Count Actuals Based On

- End of financial reporting period

Measure - Staff Experience

Measurement Category - Staff Profile

Issue - Resources and Cost

The Staff Experience measure counts the total number of software personnel with experience in defined areas. The measure is used to determine whether sufficient experienced personnel are available and used. The experience factors are based on the requirements of each individual program (such as domain or language). Experience is usually measured in years, which does not always equate to capability.

Selection Guidance

Program Application

- Applicable to all domains.
- Applicable to programs that require particular expertise to complete.
- Useful during program planning, development, and software support phases.

Process Integration

- Requires a personnel database that maintains experience data.
- Difficult to collect and keep up-to-date as people are added/removed from a project. Generally has to be done manually.

Usually Applied During

- Requirements Analysis (Actuals)
- Design (Actuals)
- Implementation (Actuals)
- Integration and Test (Actuals)

Specification Guidance

Data Items Typically Collected

- Organization
- Experience Factor
- Number of Personnel
- Number of Years of Experience

Typical Experience Factors

- Language
- System Engineering
- Domain
- Hardware
- Application
- Platform

Typical Collection Level

- Organization

Typical Reporting Level

- Organization

Count Actuals Based On

- Prior to contract award
- Annual performance evaluation

This Measure Answers Questions Such As

- Are sufficient experienced personnel available?
 - Will additional training be required?
-

Measure - Staff Turnover

Measurement Category - Staff Profile

Issue - Resources and Cost

The Staff Turnover measure counts staff losses and gains. A large amount of turnover impacts learning curves, productivity, and the ability of the software developer to build the system with the resources provided within cost and schedule. This measure is most effective when used in conjunction with the Staff Experience measure. Losses of more experienced personnel are more critical.

Selection Guidance

Program Application

- Applicable to all domains.
- Applicable to programs of all sizes and types.
- Useful during development and software support phases.

Process Integration

- Very difficult to collect on contractual programs - most organizations consider this proprietary information. May be more readily available on in-house programs.
- It is useful to categorize the number of personnel lost into planned and unplanned losses.

Usually Applied During

- Requirements Analysis (Actuals)
- Design (Actuals)
- Implementation (Actuals)
- Integration and Test (Actuals)

Specification Guidance

Data Items Typically Collected

- Organization
- Number of Personnel
- Number of Personnel Gained (per period)
- Number of Personnel Lost (per period)

Typical Collection Level

- Organization

Typical Reporting Level

- Organization

Count Actuals Based On

- End of financial reporting period
- Organization restructuring or new organizational charts
- End of program activities or milestones

This Measure Answers Questions Such As

- How many people have been added/have left the program?
 - How are the experience levels being affected by the turnover rates?
 - What areas are most affected by turnover?
-

Measurement Category - Cost Performance Issue - Resources and Cost

Cost Performance measures report the difference between budgeted and actual costs for a specific product or activity. They are used to assess whether the program can be completed within cost constraints and to identify potential cost overruns.

Program Application

- Measurement category applicable to most programs.
- Required for major DoD programs.
- Applicable to all software process models.
- Useful during program planning, development and software support phases.

Measures Included in this Category

- Cost Variance
- Cost Profile

Limitations

- Cost performance systems can be difficult to establish for software. A detailed software WBS must be developed that includes quantifiable exit criteria.
- Cost is not generally the best measure of software performance due to insufficient detail in the software WBS and associated problems with the reporting of actual progress.

Related Measurement Categories

- Milestone Performance
- Effort

Example Indicator

- Cost Profile (PSM Part 3, Section 3.7)

Measure - Cost Variance

Measurement Category - Cost Performance

Issue - Resources and Cost

The Cost Variance measure is a comparison between the cost of work performed and the budget, based on dollars budgeted per WBS element. The measure can be used to identify cost overruns and underruns.

Selection Guidance

Program Application

- Applicable to all domains.
- Applicable to any program that uses an earned value cost accounting system. The DoD defined Cost/Schedule Control System Criteria (C/SCSC) apply to programs based on size and cost.
- Useful during development and software support phases.
- Limited in applicability if costs are planned and expended on a level of effort basis.

Process Integration

- C/SCSC data is required on most large DoD contracts, so it is often readily available. This data should be based on a validated cost accounting system. If this data is not required, then the cost profile measure can be used instead.
- This can be difficult to track without an automated system tied to the accounting department.
- This data tends to lag other measurement information

Usually Applied During

- Requirements Analysis (Estimates and Actuals)
- Design (Estimates and Actuals)
- Implementation (Estimates and Actuals)
- Integration and Test (Estimates and Actuals)

This Measure Answers Questions Such As

- Are program costs in accordance with budgets?
 - What is the projected completion cost?
 - What WBS elements or tasks have the greatest variance?
-

Specification Guidance

Data Items Typically Collected

- WBS or Task Element
- Budgeted Cost of Work Performed (BCWP)
- Actual Cost of Work Performed (ACWP)

Typical Collection Level

- WBS or task element

Typical Reporting Level

- WBS or task element

Count Actuals Based On

- WBS element complete (to defined exit criteria)
- WBS element percent complete (based on engineering judgment) due to formal reporting requirements.

Measure - Cost Profile

Measurement Category - Cost Performance

Issue - Resources and Cost

The Cost Profile measure counts budgeted and expended cost. The measure provides information about the amount of money expended on a program, compared to budgets.

Selection Guidance

Program Application

- Applicable to all domains.
- Applicable to programs of all sizes and types. Used to evaluate costs for those programs that do not use cost/schedule control system criteria (C/SCSC).
- Useful during program planning, development, and software support phases.

Process Integration

- Data should come from an automated accounting system. This data tends to lag other measurement information due to formal reporting requirements.
- Should be relatively easy to collect at a high level. Not all programs, however, will define software WBS elements to a sufficient level of detail.

Usually Applied During

- Requirements Analysis (Estimates and Actuals)
- Design (Estimates and Actuals)
- Implementation (Estimates and Actuals)
- Integration and Test (Estimates and Actuals)

Specification Guidance

Data Items Typically Collected

- WBS or Task Element
- Cost (dollars)

Typical Collection Level

- WBS or task element

Typical Reporting Level

- WBS or task element

Count Actuals Based On

- WBS element complete (to defined exit criteria)
- WBS element percent complete (based on engineering judgment)

This Measure Answers Questions Such As

- Are program costs in accordance with budgets?
 - Will the target budget be achieved or will there be an overrun or surplus?
-

Measure - Environmental Availability

Issue - Resources and Cost

Environmental Availability measures indicate the availability and utilization of tool and facility resources. Resources include those used for development, integration and test, file build, maintenance or operations. Recommended for programs in which key resources are shared with or provided by other programs or are suspected from the outset to be inadequate. These measures are used to address the adequacy of resources.

Program Application

- Measurement category applicable to all programs with resource constraints.
- Applies to all software process models
- Useful during program planning, development, and software support phases.

Measures Included in this Category

- Resource Availability Dates
- Resource Utilization

Limitations

- These measures do not address whether resources are used most effectively.

Related Measurement Categories

- Schedule Performance
- Productivity
- Process Maturity

Example Indicator

- Resource Utilization Indicator (PSM Part 3, Section 3.8)

Measure - Resource Availability Dates

Measurement Category - Environment Availability

Issue - Resources and Cost

The Resource Availability Dates measure lists the dates for the availability of key resources. The measure is used to determine if key resources are available when needed to support development and testing. It can be integrated in the milestone dates measure.

Selection Guidance

Program Application

- Applicable to all domains.
- More important for programs with constrained support resources.
- Useful during development and software support phases.

Process Integration

- Required data is generally easily obtained from program scheduling systems or documentation.
- Resources may include software, hardware, integration and test facilities, tools, other equipment, or office space service
Normally only key resources are tracked. Personnel resources are not included in this measure - they are tracked with Staff Profile.
- Include both government-furnished and developer-furnished resources.

Usually Applied During

- Requirements Analysis (Estimates and Actuals)
- Design (Estimates and Actuals)
- Implementation (Estimates and Actuals)
- Integration and Test (Estimates and Actuals)

This Measure Answers Questions Such As

- Are key resources available when needed?
 - Is the availability of support resources impacting progress?
-

Specification Guidance

Data Items Typically Collected

- Resource Name
- Availability Date

Typical Collection Level

- Resource

Typical Reporting Level

- Resource

Count Actuals Based On

- Demonstration of the intended

Measure - Resource Utilization

Measurement Category - Environment Availability

Issue - Resources and Cost

The Resource Utilization measure counts of the hours of resource time scheduled, available, not available due to maintenance downtime, and used. It is used on programs that have resource constraints, and is usually focused only on key resources. This measure provides an indication of whether key resources are sufficient and if they are used effectively.

Selection Guidance

Program Application

- Applicable to all domains.
- More important for programs with constrained support resources. Especially important during integration and test activities.
- Useful during development and software support phases.

Process Integration

- Relatively easy to collect at a high level. Easier to collect if a resource monitor or resource scheduling system is in place.
- Resources may include software, hardware, integration and test facilities, tools, and other equipment. Normally only key resources are tracked.
- Include both government-furnished and developer-furnished resources.

Usually Applied During

- Requirements Analysis (Estimates and Actuals)
- Design (Estimates and Actuals)
- Implementation (Estimates and Actuals)
- Integration and Test (Estimates and Actuals)

This Measure Answers Questions Such As

- Are sufficient resources available?
 - How efficiently are resources being used?
-

Specification Guidance

Data Items Typically Collected

- Resource Name
- Scheduled Hours
- Available Hours
- Used hours
- Hours Unavailable due to Maintenance

Typical Collection Level

- Resource

Typical Reporting Level

- Resource

Count Actuals Based On

- End of reporting period

Measurement Category - Product Size and Stability

Issue - Growth and Stability

Product Size and Stability measures quantify the physical size of a software product. Product size is a critical factor for estimating development schedule and cost. These measures also provide information about the amount and frequency of change to software products which is especially critical late in the development.

Program Application

- Basic measurement category applicable to most programs.
- Measures in this category are usually selected based on domain characteristics.
- Applicable to all software process models.
- Useful during program planning, development, and software support phases.

Measures Included in this Category

- Lines of Code
- Number of Components
- Words of Memory
- Database Size

Limitations

- Product size measures do not always directly map to the amount of functionality in the system.
- Measures in this category do not generally address software quality, complexity, or difficulty.
- Accurate estimates are dependent on the availability of good historical data or engineering experience.
- Reported changes of software product size often occur too late to correct the underlying problems. Measurement of requirements or design changes provide earlier warnings of related problems.

Related Measurement Categories

- Productivity
- Functional Size and Stability
- Work Unit Progress

Additional Information

- Components may be defined differently for each program. Components can be units, CSCIs, objects, interfaces, screens, reports, packages, icons, primitives, or other measurable product structures. Problem reports are sometimes considered to be components, especially with respect to software maintenance activities during the software support phase. COTS/GOTS and other non-developed or reusable software products can also be counted as components. Some components can be aggregated to form higher level components (for example, units to CSCIs to builds). These can be referred to as sub-components.

Example Indicator

- Software Size (PSM Part 3, Section 3.9)

Measure - Lines of Code

Measurement Category - Product Size and Stability

Issue - Growth and Stability

The Lines of Code measure counts the total amount of source code and the amount that has been added, modified, or deleted. The total number of lines of code is a well understood measure which allows estimation of project cost, required effort, schedule, and productivity. Changes in the number of lines of code indicate development risk due to product size volatility and additional work which may be required.

Selection Guidance

Program Application

- Applicable to all domains. Commonly used in weapons applications. Included in most DoD measurement policies and some commercial measurement practices.
- Used for programs of all size. Less important for programs where little code is generated such as those using automatic code generation and visual programming environments.
- Most effective for traditional high order languages such as Ada, FORTRAN, and Cobol. Not generally used for fourth-generation languages such as Natural and ECOS.
- Not usually tracked for COTS software unless changes are made to the source code.
- Useful during program planning, development, and software support phases.

Process Integration

- Define Lines of Code for each language. Lines of code from different languages are not equivalent.
- You may want to calculate an effective or equivalent SLOC count based on source. New and modified lines would count at 100% while reused code would count at a lower percentage (to address the required effort to integrate and test the reused code).
- Sometimes difficult to generate accurate estimates early in the program, especially for new types of programs. Easy to count actuals once code is developed using automated tools.
- Estimates should be updated on a regular basis.
- Can be difficult estimating and tracking lines of code by source (new, modified, retained, deleted, NDI, GOTS, or COTS).
- Actuals can easily be counted using automated tools.

Usually Applied During

- Requirements Analysis (Estimates)
- Design (Estimates)
- Implementation (Estimates and Actuals)
- Integration and Test (Actuals)

Specification Guidance

Data Items Typically Collected

- Component Name
- Source (new, modified, deleted, reused, NDI, GOTS, or COTS)
- Language
- Delivery Status (deliverable, non-deliverable)
- Category (operational, support)
- Build/Release
- Number of Lines of Code (LOC)
- Number of LOC Added
- Number of LOC Deleted
- Number of LOC Modified

LOC Definition May Include

- Logical Lines
- Physical Lines
- Blanks
- Comments
- Executables
- Data Declarations

Typical Collection Level

- Unit or equivalent

Typical Reporting Level

- CSCI or equivalent
- Build/Release

Count Actuals Based On

- Release to configuration management
- Passing unit test
- Passing inspection

This Measure Answers Questions Such As

- How accurate was the size estimate that the schedule and effort plans were based on?
 - How much has the software size changed? In which components have changes occurred?
 - Has the size allocated to each incremental build changed? Is functionality slipping to later builds?
-

Measure - Number of Components

Measurement Category - Product Size and Stability

Issue - Growth and Stability

The Number of Components measure counts the number of elementary software components in a software product, and the number that are added, modified, or deleted. The total number of components defines the size of the software product. Changes in the number of estimated and actual components indicates risk due to product size volatility and additional work which may be required. Reporting the number of design components provides product size information earlier than other size measures, such as Lines of Code or Function Points.

Selection Guidance

Program Application

- Applicable to all application domains, generally with different component definitions.
- Applicable to all size and type programs.
- Useful during development and software support phases.

Process Integration

- Requires a well defined and consistent component allocation structure (i.e. Unit to CSCI to Build).
- Required data is generally easy to obtain from software design tools, configuration management tools, or documentation.
- Deleted and added components are relatively easy to collect - modified components are often not tracked.
- Volatility in the planned number of components may represent instability in the requirements in the design of the software.

Usually Applied During

- Requirements Analysis (Estimates)
- Design (Estimates and Actuals)
- Implementation (Estimates and Actuals)
- Integration and Test (Actuals)

Specification Guidance

Data Items Typically Collected

- Component Name
- Source (new, modified, deleted, reused, NDI, GOTS, or COTS)
- Language
- Delivery Status (deliverable, non-deliverable)
- Category (operational, support)
- Build/Release
- Number of Components
- Number of Components Added
- Number of Components Deleted
- Number of Components Modified

Typical Collection Level

- Unit or equivalent

Typical Reporting Level

- CSCI or equivalent
- Build/Release

Count Actuals Based On

- Release to configuration management
- Passing unit test
- Passing inspection

This Measure Answers Questions Such As

- How many components need to be implemented and tested?
 - How much has the approved software baseline changed?
 - Have the components allocated to each incremental build changed? Is functionality slipping to later builds?
-

Measure - Words of Memory

Measurement Category - Product Size and Stability

Issue - Growth and Stability

The Words of Memory measure counts the number of words used in main memory, in relation to total memory capacity. This measure provides a basis to estimate if sufficient memory will be available to execute the software in the expected operational scenarios.

Selection Guidance

Program Application

- Most commonly used for weapons systems.
- Used on any program with severe memory constraints such as avionics or on-board flight software.
- For many programs the amount of memory reserved is part of the defined exit criteria.
- Useful during development and software support phases.

Process Integration

- Requires an automated tool that measures usage based on a defined operational profile. This is often difficult to collect.
- Estimation may be based on modeling or by assuming a translation factor between lines of code and words of memory.

Usually Applied During

- Requirements Analysis (Estimates)
- Design (Estimates)
- Implementation (Estimates)
- Integration and Test (Estimates and Actuals)

Specification Guidance

Data Items Typically Collected

- Processor Name
- Number of Words of Memory
- Number of Words of Memory Used

Typical Collection Level

- Processor

Typical Reporting Level

- Processor

Count Actuals Based On

- Completion of integration
- During Test Readiness Review (TRR)
- Prior to delivery

This Measure Answers Questions Such As

- How much spare memory capacity is there?
 - Does the memory need to be upgraded?
-

Measure - Database Size

Measurement Category - Product Size and Stability

Issue - Growth and Stability

The Database Size measure counts the number of words, records, or tables (elements) in each database. The measure indicates how much data must be handled by the system.

Selection Guidance

Program Application

- Applicable to all domains. Often used for AIS programs.
- Used for any program with a significant database. Especially important for those with performance constraints.
- Useful during development and software support phases.

Process Integration

- In order to estimate the size of a database, you must develop an operational profile. This is generally a manual process which can be difficult. Actuals are relatively easy to collect.

Usually Applied During

- Requirements Analysis (Estimates)
- Design (Estimates)
- Implementation (Estimates and Actuals)
- Integration and Test (Actuals)

Specification Guidance

Data Items Typically Collected

- Database Name
- Number of Tables, Words, or Bytes
- Number of Records or Entries

Typical Collection Level

- Database

Typical Reporting Level

- Database

Count Actuals Based On

- Schema design released to configuration management
- Schema implementation released to configuration management

This Measure Answers Questions Such As

- How much data has to be handled by the system?
 - How many different data types have to be addressed?
-

Measurement Category - Functional Size and Stability

Issue - Growth and Stability

Functional Size and Stability measures quantify the functionality of a software product. Functional size may be used to estimate development schedule and cost. These measures also provide information about the amount and frequency of change to software functionality which is especially critical late in the development. Functional changes generally correlate to effort, cost, schedule, and product size changes.

Program Application

- Measurement category applicable to most programs.
- Applicable to all software process models.
- Useful during program planning, development, and software support phases.

Measures Included in this Category

- Requirements
- Function Points

Limitations

- Data collection requires a defined method or tool and is often labor intensive.
- Since data is usually collected manually, variations can be expected from different measurement sources.

Related Measurement Categories

- Target Computer Resource Utilization
- Complexity
- Product Size and Stability
- Work Unit Progress

Example Indicator

- Requirements Stability (PSM Part 3, Section 3.10)

Measure - Requirements

Measurement Category - Functional Size and Stability

Issue - Growth and Stability

The Requirements measure counts the number of requirements in the software and interface specifications, and the number of these requirements that are added, modified, or deleted. The measure provides information on the total number of requirements, and the development risk due to volatility in requirements or functional growth.

Selection Guidance

Program Application

- Applicable to all domains.
- Applicable to any program that tracks requirements. Useful for any size and type of program.
- Useful during program planning, development, and software support phases.
- Effective for both non-developed (COTS/GOTS/Reuse and newly developed software).

Process Integration

- Requires a good requirements traceability process. If an automated design tool is used, the data is more readily available.
- Count changes against a baseline which is under formal configuration control. Both stated and derived requirements may be included.
- To evaluate stability, a good definition of the impacts of each change is required.
- It is sometimes difficult to specifically define a "requirement". A consistently applied definition makes this measure more effective.

Usually Applied During

- Requirements Analysis (Estimates and Actuals)
- Design (Estimates and Actuals)
- Implementation (Estimates and Actuals)
- Integration and Test (Actuals)

Specification Guidance

Data Items Typically Collected

- Component Name
- Build/Release
- Number of Requirements
- Number of Requirements Added
- Number of Requirements Deleted
- Number of Requirements Modified
- Source of Change (developer program manager)

Typical Collection Level

- Requirement
- Component

Typical Reporting Level

- Build/Release

Count Actuals Based On

- Passing requirements inspection
- Release to configuration management

This Measure Answers Questions Such As

- Have the requirements allocated to each incremental build changed? Are requirements being deferred to later builds?
 - How much has software functionality changed? Which components have been affected the most?
-

Measure - Function Points

Measurement Category - Functional Size and Stability

Issue - Growth and Stability

The Function Points measure provides a weighted count of the number of external inputs and outputs, logical internal files and interfaces, and inquiries. This measure determines the functional size of software to support an early estimate of the required level of effort. It can also be used to support productivity assessments

Selection Guidance

Program Application

- Applicable to all domains. Commonly used in AIS applications.
- Not usually tracked for COTS or reused software.
- Useful during development and software support phases.

Process Integration

- Requires a design process compatible with function points.
- Should be based on a defined method such as the IFPUG function point counting practices manual.
- Usually requires formal training.
- Requires a well defined set of work products to describe the requirements and design.
- Very labor intensive to estimate - automated tools are scarce and have not been validated.

Usually Applied During

- Requirements Analysis (Estimates)
- Design (Estimates and Actuals)
- Implementation (Actuals)
- Integration and Test (Actuals)

Specification Guidance

Data Items Typically Collected

- Component Name
- Source (new, modified, deleted, reused, NDI, GOTS, or COTS)
- Build/Release
- Number of Function Points

Typical Collection Level

- CSCI or equivalent

Typical Reporting Level

- CSCI or equivalent
- Build/Release

Count Actuals Based On

- Completion of design documentation
- Release to configuration management
- Passing design documentation inspections

This Measure Answers Questions Such As

- How big is the software product?
 - How much work is there to be done?
 - How much functionality is in the software?
-

Measurement Category - Target Computer Resource Utilization Issue - Growth and Stability

Target Computer Resource Utilization measures are used to assess the adequacy of the target hardware. High computer resource utilization can have serious impacts on software performance, cost, schedule, and supportability. High utilization may require hardware changes or software redesign. During development, reserve capacity is often defined to allow for future growth due to changes or additional requirements

Program Application

- Measurement category applicable to programs with target hardware resource constraints.
- Applicable to all software process models.
- Useful during development and software support phases.

Measures Included in this Category

- CPU Utilization
- CPU Throughput
- I/O Utilization
- I/O Throughput
- Memory Utilization
- Storage Utilization
- Response Time

Limitations

- These measures are often difficult to define, estimate, and collect. Some computer systems do provide automated status reporting of some of the measures in this category.

Related Measurement Categories

- Product Size and Stability
- Complexity
- Rework

Example Indicator

- Response Time (PSM Part 3, Section 3.11)

Measure - CPU Utilization

Measurement Category - Target Computer Resource Utilization

Issue - Growth and Stability

The CPU Utilization measure counts the estimated or actual proportion of time the CPU is busy during a measured time period. This measure indicates whether sufficient CPU resources will be available to support operational processing. This measure is also used to evaluate whether CPU reserve capacity will be sufficient for high-usage operations or for added functionality.

Selection Guidance

Program Application

- Applicable to all domains. Primarily used for weapon systems.
- Useful for any program with a dedicated processor and critical performance requirements. Not generally used on programs running on shared processors.
- Useful during development and software support phases.

Process Integration

- Requires a tool that measures usage based on a defined operational profile during a measured period of time.
- The operational profile (load levels) has a significant impact on this measure. Tests should include both normal and stress levels of operation.
- Estimates are very difficult to derive and require significant simulation or modeling support. Estimates must be developed early to impact design decisions.
- Actual processor utilization is often provided as an overhead function of an operating system and is more easily obtained.

Usually Applied During

- Design (Estimates)
- Implementation (Estimates and Actuals)
- Integration and Test (Actuals)

This Measure Answers Questions SuchAs

- Have sufficient CPU resources been provided?
 - Do CPU estimates appear reasonable? Have large increases occurred?
 - Can the CPU resources support additional functionality?
-

Specification Guidance

Data Items Typically Collected

- CPU Name
- Operational Profile
- Time CPU is Busy
- Measured Time Period
- Specified CPU Utilization Limit

Typical Collection Level

- CPU

Typical Reporting Level

- CPU
- Target HWC1

Count Actuals Based On

- Integrated system test
- Stress/endurance test

Measure - CPU Throughput

Measurement Category - Target Computer Resource Utilization

Issue - Growth and Stability

The CPU Throughput measure provides an estimate or actual count of the number of processing tasks that can be completed in a specified period of time. This measure provides an indication of whether or not the software can support the system's operational processing requirements.

Selection Guidance

Program Application

- Applicable to all domains. Primarily used for weapon systems.
- Useful for any program with a dedicated processor and critical timing requirements. Not generally used on programs running on shared processors.
- Useful during development and software support phases.

Process Integration

- Actuals can be based on real-time observation or may require a tool that measures task completion based on a defined operational profile. This data is generally easier to collect.
- The operational profile has a significant impact on this measure. Tests should include both normal and stress levels of operation.
- Estimates are very difficult to derive and require significant simulation or modeling support. Estimates must be developed early to impact design decisions.
- The measurement methodology for CPU throughput is critical for meaningful results. In many cases the measure is based on average CPU throughput. The averaging period used is therefore important.

Usually Applied During

- Design (Estimates)
- Implementation (Estimates and Actuals)
- Integration and Test (Actuals)

This Measure Answers Questions Such As

- Have sufficient CPU resources been acquired?
 - Do CPU estimates appear reasonable? Have large increases occurred?
-

Specification Guidance

Data Items Typically Collected

- CPU Name
- Operational Profile
- Number of Requests for Service
- Number of Requests for Service Completed
- Measured Time Period
- Specified CPU Throughput Limit

Typical Collection Level

- CPU

Typical Reporting Level

- CPU
- Target HWCI

Count Actuals Based On

- Integrated system test
- Stress/endurance test

Measure - I/O Utilization

Measurement Category - Target Computer Resource Utilization

Issue - Growth and Stability

The I/O Utilization measure calculates the proportion of time the I/O resources are busy during a measured time period. This measure indicates whether I/O resources are sufficient to support operational processing requirements.

Selection Guidance

Program Application

- Applicable to all domains. Primarily used for weapon systems.
- Critical for high traffic systems.
- Network I/O may also be measured under this measure.
- Useful during development and software support phases.

Process Integration

- Actual measurement requires a tool that measures usage based on a defined operational profile during a measured period of time. Actuals are relatively easy to collect.
- The operational profile has a significant impact on this measure. Tests should include both normal and stress levels of operation.
- Estimates are very difficult to derive and require significant simulation or modeling support. Estimates must be developed early to impact design decisions.

Usually Applied During

- Design (Estimates)
- Implementation (Estimates and Actuals)
- Integration and Test (Actuals)

Specification Guidance

Data Items Typically Collected

- I/O Channel Name
- Operational Profile
- Time I/O Resource is Busy
- Time I/O Resource is Available
- Measured Time Period
- Specified I/O Channel Utilization Limit

Typical Collection Level

- I/O Resource

Typical Reporting Level

- I/O Resource
- Target HWC1

Count Actuals Based On

- Integrated system test
- Stress/endurance test

This Measure Answers Questions SuchAs

- Do the I/O resources allow adequate data traffic flow?
 - Can additional data traffic be provided after system delivery?
 - Should I/O resources be expanded?
-

Measure - I/O Throughput

Measurement Category - Target Computer Resource Utilization

Issue - Growth and Stability

The I/O Throughput measure reports the rate at which the I/O resources send and receive data, according to the number of data packets (bytes, words, etc.) successfully sent or received during a measured time period. This measure indicates whether the I/O resources are sufficient to support the system's operational processing requirements.

Selection Guidance

Program Application

- Applicable to all domains. Primarily used for weapon systems.
- Critical for high traffic systems.
- Network I/O may also be measured using this measure.
- Useful during development and software support phases.

Process Integration

- Actual measurement requires a tool that measures usage based on a defined operational profile during a measured period of time. This is relatively easy to collect.
- The operational profile has a significant impact on this measure. Tests should include both normal and stress levels of operation.
- Estimates are very difficult to derive and require significant simulation or modeling support. Estimates must be developed early to impact design decisions.
- The measurement methodology for I/O throughput is critical for meaningful results. In many cases the measure is based on average I/O throughput. The averaging period used is therefore important.

Usually Applied During

- Design (Estimates)
- Implementation (Estimates and Actuals)
- Integration and Test (Actuals)

This Measure Answers Questions Such As

- Can the software design handle the required amount of system data in the allocated time?
 - Can the software handle additional system data after delivery?
-

Specification Guidance

Data Items Typically Collected

- I/O Resource Name
- Operational Profile
- Number of Data Packets
- Number of Data Packets Successfully Sent
- Number of Data Packets Successfully Received
- Measured Time Period
- Specified I/O Throughput Limit

Typical Collection Level

- I/O Resource

Typical Reporting Level

- I/O Resource
- Target HWCI

Count Actuals Based On

- Integrated system test
- Stress/endurance test

Measure - Memory Utilization

Measurement Category - Target Computer Resource Utilization

Issue - Growth and Stability

The Memory Utilization measure indicates the proportion of memory which is used during a measured time period. This measure addresses random access memory (RAM), read only memory (ROM), or any other form of electronic, volatile memory. This measure specifically excludes all types of magnetic and optical media (e.g. disk, tape, CD-ROM, etc.). This measure provides an indication of whether the memory resources can support the system's operational processing requirements.

Selection Guidance

Program Application

- Applicable to all domains. Primarily used for weapon systems.
- Critical for memory constrained systems.
- Useful during development and software support phases.

Process Integration

- Measure and monitor different types of memory (e.g. RAM, ROM) separately. Specify the size of a word (e.g. 16 bit, 32 bit, etc.) for each memory type.
- Actual measurement requires a tool that measures usage based on a defined operational profile during a measured time period or task. This is relatively easy to collect.
- The operational profile has a significant impact on this measure. Tests should include both normal and stress levels of operation.
- Estimates are very difficult to derive and require significant simulation or modeling support. Estimates must be developed early to impact design decisions.

Usually Applied During

- Design (Estimates)
- Implementation (Estimates and Actuals)
- Integration and Test (Actuals)

Specification Guidance

Data Items Typically Collected

- Memory Name
- Operational Profile
- Memory Available
- Memory Used
- Measured Time Period
- Specified Memory Utilization Limit

Typical Collection Level

- CPU

Typical Reporting Level

- CPU
- Target HWC1

Count Actuals Based On

- Integrated system test
- Stress/endurance test

This Measure Answers Questions SuchAs

- Will the software fit in the processors?
 - Can the software size increase after system delivery as needed to incorporate new functionality?
 - What is the risk that system errors will be caused by lack of storage space?
-

Measure - Storage Utilization

Measurement Category - Target Computer Resource Utilization

Issue - Growth and Stability

The Storage Utilization measure reports the proportion of storage capacity used. The measure provides an indication of whether storage resources are sufficient to store programs and/or the anticipated volume of operational data generated by the system. The term "storage" refers to magnetic and optical media (e.g. disk, tapes, hard drives, CD-ROM, etc.), but specifically excludes all types of random access memory (RAM), read only memory (ROM), or any other forms of electronic memory.

Selection Guidance

Program Application

- Applicable to all domains. Primarily used for weapon systems.
- Critical for storage constrained systems.
- Useful during development and software support phases.

Process Integration

- Measure and monitor different types of storage (e.g. disk, tape) separately. Specify the size of a word (e.g. 16 bits, 32 bits, etc.) for each storage type.
- Actuals are easy to measure. Estimates are often based on product size.

Usually Applied During

- Design (Estimates)
- Implementation (Estimates and Actuals)
- Integration and Test (Actuals)

This Measure Answers Questions Such As

- Have sufficient storage resources been provided?
 - Do storage estimates appear adequate?
 - What is the expansion capacity?
-

Specification Guidance

Data Items Typically Collected

- Storage Name
- Storage Available
- Storage Used
- Specified Storage Utilization Limit

Typical Collection Level

- Storage Device

Typical Reporting Level

- Storage Device
- Target HWCI

Count Actuals Based On

- Integrated system test
- Stress/endurance test

Measure - Response Time

Measurement Category - Target Computer Resource Utilization

Issue - Growth and Stability

The Response Time Measure reports the amount of time required to process a request. The measure counts the time between initiation of a request for service and the conclusion of that service. It provides an indication of whether the target computer system responds in a timely manner.

Selection Guidance

Program Application

- Applicable to all domains. Used extensively on AIS systems.
- Critical for programs with specified response time requirements. Especially critical for real-time programs.
- Useful during development and software support phases.

Process Integration

- Actuals can be based on real-time observation or may require a tool that measures request completion based on a defined operational profile. This data is generally easier to collect.
- The operational profile has a significant impact on this measure. Tests should include both normal and stress levels of operation.
- This measure must be collected at a low level in order to provide a good characterization of the level of service provided.

Usually Applied During

- Design (Estimates)
- Implementation (Estimates and Actuals)
- Integration and Test (Actuals)

This Measure Answers Questions Such As

- Is the target computer system sufficient to meet response requirements?
 - How long do certain services take?
 - Does the software operate efficiently?
-

Specification Guidance

Data Items Typically Collected

- HWCI Name
- Operational Profile
- Service Name
- Service Initiation Time
- Service Completion Time
- Maximum Allowable Service Time

Typical Collection Level

- Service

Typical Reporting Level

- Service

Count Actuals Based On

- Integrated system test
- Stress/endurance test

Measurement Category - Defect Profile Issue - Product Quality

Defect Profile measures identify the number of problem reports, defects, and failures in the software products and/or processes. Defect Profile measures are some of the best measures for monitoring integration and test progress. These measures also provide an indication of product quality.

Program Application

- Basic measurement category applicable to most programs.
- Applicable to all software process models.
- Useful during development and software support phases.

Measures Included in this Category

- Problem Report Trends
- Problem Report Aging
- Defect Density
- Failure Interval

Limitations

- Measures in this category do not always address the effort which is required to implement the changes. It is possible to have one change that has a major impact on all facets of the program, or multiple changes with minimal impact.

Related Measurement Categories

- Work Unit Progress
- Rework
- Product Size and Stability

Additional Information

- A defect is a product's non-conformance with its specification. A problem report is a documented description of a defect, unusual occurrence, observation, or failure that requires investigation and may involve software modifications. Not all problem reports identify valid software problems. A valid software problem may be associated with multiple defects.
- While commonly tracked during testing, defect profile measures are extremely useful when they are applied during software requirements analysis and design.

Example Indicator

- Problem Report Status (PSM Part 3, Section 3.12)
- Problem Report Aging (PSM Part 3, Section 3.13)
- Defect Density (PSM Part 3, Section 3.14)

Measure - Problem Report Trends

Measurement Category - Defect Profile

Issue - Product Quality

The Problem Report (PR) Trends measure quantifies the number, status, and priority of problems reported. It provides very useful information on the ability of a developer to find and fix defects. The quantity of PRs reported reflects the amount of development rework (quality). Arrival rates can indicate product maturity (a decrease should occur as testing is completed). Closure rates are an indication of progress and can be used to predict test completion.

Selection Guidance

Program Application

- Basic measure applicable to all domains.
- Included in most DoD measurement policies and commercial measurement practices.
- Applicable to all sizes and types of programs.
- Useful during development and software support phases.

Process Integration

- Requires a disciplined problem reporting process.
- This measure is generally available during integration and test. It is beneficial, however, to begin problem tracking earlier. Potential areas for tracking include requirements, design, code, and unit test inspections, unit tests, CSCI and build level integration and testing, and system level testing.
- The status codes used on a program should address at a minimum which problem reports are open and closed.
- Easy to collect actuals when an automated problem reporting system is used. Many programs do not estimate the number of problem reports expected.
- The number of discovered problem reports should be considered relative to the amount of discovery activity (number of inspections, amount of testing, etc.)
- Many programs use the number of open problem reports, by priority categories, as a measure of readiness for test/delivery.

Usually Applied During

- Requirements Analysis (Estimates and Actuals)
- Design (Estimates and Actuals)
- Implementation (Estimates and Actuals)
- Integration and Test (Estimates and Actuals)

This Measure Answers Questions Such As

- How many (critical) problem reports have been written?
 - Do defect arrival and closure rates support the scheduled completion date of integration and test?
-

Specification Guidance

Data Items Typically Collected

- Component Name
- Priority
- Status Code
- Number of Problem Reports
- Build/Release
- Discovery Activity

Typical Collection Level

- CSCI or equivalent

Typical Reporting Level

- CSCI or equivalent
- Build/Release

Count Actuals Based On

- Successfully tested
- Successfully integrated
- Delivery to field
- Problem report documented
- Problem report approved by configuration control board

Measure - Problem Report Aging

Measurement Category - Defect Profile

Issue - Product Quality

The Problem Report Aging measure reports the length of time that each problem report (PR) has remained open. The measure is used to determine whether progress is being made in fixing problems. It helps assess whether or not software rework is being deferred.

Selection Guidance

Program Application

- Applicable to all domains.
- Applicable to all sizes and types of programs.
- Useful during development and software support phases.

Process Integration

- Requires a disciplined problem reporting process.
- Easy to collect actuals when an automated problem reporting system is used. Most programs do not estimate problem report aging.

Usually Applied During

- Requirements Analysis (Actuals)
- Design (Actuals)
- Implementation (Actuals)
- Integration and Test (Actuals)

Specification Guidance

Data Items Typically Collected

- Problem Report Name
- Component Name
- Priority
- Status Code
- Discovery Date
- Closure Date
- Build/Release

Typical Collection Level

- Problem Report

Typical Reporting Level

- CSCI or equivalent
- Build/Release

Count Actuals Based On

- Successfully tested
- Successfully integrated
- Delivery to field
- Problem report documented
- Problem report approved by configuration control board

This Measure Answers Questions Such As

- How long does it take to close a PR?
 - Is the developer closing known problems in a timely manner? (How long have open PRs remained open?)
 - Are the problems which are more difficult to fix being deferred?
-

Measure - Defect Density

Measurement Category - Defect Profile

Issue - Product Quality

The Defect Density measure is a ratio of the number of defects written against a component relative to the size of that component. Either a product or function oriented size measure can be used. The measure helps identify components with the highest concentration of defects. These components often become candidates for additional reviews or testing, or may need to be re-written. Trends in the overall quality of a system can also be monitored with this measure.

Selection Guidance

Program Application

- Applicable to all domains.
- Applicable to all sizes and types of programs.
- Useful during development and software support phases.

Process Integration

- Requires a disciplined problem reporting process and a method for measuring software size.
- Requires the allocation of defect and size data to the associated component affected.
- In order to use functional measures of size, requirements or function points must be allocated to the associated components.
- Actuals are relatively easy to collect. Most programs do not estimate defect density.

Usually Applied During

- Requirements Analysis (Actuals)
- Design (Actuals)
- Implementation (Actuals)
- Integration and Test (Actuals)

Specification Guidance

Data Items Typically Collected

- Component Name
- Number of Defects
- Priority
- Number of Lines of Code
- Source (new, modified, deleted, reused, NDI, GOTS, or COTS)
- Language
- Build/Release

Size May be Measured As

- Lines of Code
- Components
- Requirements
- Function Points

Typical Collection Level

- CSCI or equivalent

Typical Reporting Level

- CSCI or equivalent
- Build/Release

Count Actuals Based On

- Defects documented
- Successfully integrated
- Successfully tested
- Delivery to field

This Measure Answers Questions Such As

- What is the quality of the software?
 - Which components have a disproportionate amount defects?
 - Which components require additional testing or review?
 - Which components are candidates for rework?
-

Measure - Failure Interval

Measurement Category - Defect Profile

Issue - Product Quality

The Failure Interval measure specifies the time between each report of a software failure. The measure is used as an indicator of the length of time that a program can be expected to run without a software failure (during production systems operation). The measure provides insight into how the software affects overall system reliability. This measure can be used as an input to reliability prediction models.

Selection Guidance

Program Application

- Applicable to all domains.
- Applicable to any program with reliability requirements.
- Useful during development in system or operational test. Used throughout software support based on reported operational failures.

Process Integration

- Requires a disciplined failure tracking process. Easier to collect if an automated system is used. Data can be gathered from test logs or incident reports.
- Consider what priority of failures to include.
- Be sure to exclude non-software failures.
- Some programs specify threshold limits for software reliability.

Usually Applied During

- Integration and Test (Actuals)

Specification Guidance

Data Items Typically Collected

- Failure Identifier
- Date/Time Stamp
- Operational Hours to Failure
- Priority

Typical Collection Level

- Build/Release

Typical Reporting Level

- Build/Release

Count Actuals Based On

- Failure documented
- Failure Validated

This Measure Answers Questions Such As

- What is the program's expected operational reliability?
 - How often will software failures occur during operation of the system?
-

Measurement Category - Complexity

Issue - Product Quality

Complexity measures quantify the structure of software components, based on the number and intricacy of interfaces and branches, the degree of nesting, the types of data structures, and other characteristics. Complex components are generally harder to test, are more difficult to maintain, and sometimes contain more defects than less complex components. Complexity measures provide indications of the need to redesign and the relative amount of testing required for any component.

Program Application

- Measurement category applicable to programs with long-term software support requirements.
- Applicable to most software process models.
- Useful during development and software support phases.

Measures Included in this Category

- Cyclomatic Complexity

Limitations

- Data is not generally available until after a component has been coded (although some CASE tools measure design complexity). Reducing complexity requires rework to redesign or recode the software.
- The interpretation of complexity is different for various high order languages.
- Some components must be complex to meet specified functional and performance requirements. The measures do not account for this.

Related Measurement Categories

- Defect Profile
- Product Size and Stability
- Rework

Example Indicator

- Software Complexity (PSM Part 3, Section 3.15)

Measure - Cyclomatic Complexity

Measurement Category - Complexity

Issue - Product Quality

The Cyclomatic Complexity measure counts the number of unique logical paths contained in a software component. This measure helps assess both code quality and the amount of testing required. A high complexity rating is often indicative of a high defect rate. Programs with high complexity may require additional reviews, testing, or may need to be re-written.

Selection Guidance

Program Application

- Applicable to all domains.
- Applicable to programs with testability, reliability, or maintainability concerns.
- Not generally used for COTS or reused code. Not generally used on software from automatic code generators or visual programming environments.
- Useful during development and software support phases.

Process Integration

- Cyclomatic complexity does not differentiate between types of control flow. A CASE statement counts as high complexity even though it is easier to use and understand than a series of conditional statements.
- Cyclomatic complexity does not address data structures.
- Operational requirements may require efficient, highly complex code.
- Relatively easy to collect actuals when automated tools are available (e.g. for Ada, C, C++). Estimates are generally not derived, but a desired threshold may be specified.

Usually Applied During

- Design (Actuals)
- Implementation (Actuals)
- Integration and Test (Actuals)

Specification Guidance

Data Items Typically Collected

- Component Name
- Cyclomatic Complexity Rating
- Build/Release

Typical Collection Level

- Unit or equivalent

Typical Reporting Level

- CSCI or equivalent
- Build/Release

Count Actuals Based On

- Release to configuration management
- Passing unit test
- Passing inspection

This Measure Answers Questions Such As

- How many complex components exist in this program?
 - Which components are the most complex?
 - Which components should be subjected to additional testing?
-

Measurement Category - Process Maturity Issue - Development Performance

Process Maturity measures address the capability of the software development processes within an organization. The measures may be used to predict the ability of an organization to best address the issues and constraints of a development program. These measures may also be used internally as part of a process improvement function.

Program Application

- Measurement category applicable to most programs.
- Applicable to all software process models.
- Useful during program planning.

Measures Included in this Category

- SEI Capability Maturity Model Level

Limitations

- These measures may be obtained through a formal assessment for certification or through an informal self-evaluation. Only the results of a formal certification should be accepted for source selection. A formal certification requires an investment to achieve required capabilities and to complete the certification process. A strong management commitment is essential.
- Process capability is often determined at an organization level. That capability may not be carried to the department or project levels, especially when there are significant program cost and schedule constraints.
- Process capability may help to select an adequate developer, but actual performance may vary considerably among developers at the same maturity level.
- A high level of software process maturity does not guarantee program success.
- There is subjectivity in the determination of process maturity.

Related Measurement Categories

- Environment Availability
- Productivity
- Rework

Example Indicator

- Software Process Maturity (PSM Part 3, Section 3.16)

Measure - Capability Maturity Model Level

Measurement Category - Process Maturity

Issue - Development Performance

The Capability Maturity Model (CMM) Level measure reports the rating (1-5) of a software development organization's software development process, as defined by the Software Engineering Institute. The measure is the result of a formal assessment of the organization's project management and software engineering capabilities. It is often used during the source selection process to evaluate competing developers.

Selection Guidance

Program Application

- Applicable to all domains.
- Normally applied at the organizational level.
- Useful during program planning, development and software support phases.

Process Integration

- Requires formal training and a very structured assessment approach. Requires a significant amount of time and effort.
- Assessment may be formally conducted by an external assessor, or a self-evaluation can be performed.
- Rating may be used during source selection to help select a developer. Assessment may be used as part of a process improvement program.

Usually Applied During

- Not applicable.

Specification Guidance

Data Items Typically Collected

- Organization Name
- CMM Rating

Typical Collection Level

- Program

Typical Reporting Level

- Organization

Count Actuals Based On

- Prior to contract award
- Annual performance evaluation

This Measure Answers Questions Such As

- Does a developer meet minimum development capability requirements?
 - What is the developer's current software development capability?
 - What project management and software engineering practices can be improved?
-

Measurement Category - Productivity

Issue - Development Performance

Productivity measures identify the amount of software product produced per unit of effort. Productivity measures are widely used as an indication of whether or not a program has adequate funding and schedule relative to the amount of software to be developed. Assessments of actual productivity provide an indication of whether the developer is producing code at a sufficient rate.

Program Application

- Measurement category applicable to most programs.
- Applicable to all software process models.
- Useful during program planning, development, and software support phases.
- While not explicitly included in most DoD measurement policies and commercial measurement practices, the data necessary to calculate these measures are generally included.

Measures Included in this Category

- Product Size/Effort Ratio
- Functional Size/Effort Ratio

Limitations

- Productivity measures cannot be compared to each other, unless the same definitions are used for the amount of product or function (in the same language) and effort (same labor categories included). This is probably the most misused measure.
- Measures in this category may not address software quality, complexity, or difficulty.
- Actual software productivity for different programs developed by the same organization can vary considerably. A high productivity on one program does not guarantee a high productivity for others.

Related Measurement Categories

- Product Size and Stability
- Functional Size and Stability
- Effort
- Milestone Performance

Example Indicator

- Software Productivity (PSM Part 3, Section 3.17)

Measure - ProductSize/Effort Ratio

Measurement Category - Productivity

Issue - Development Performance

The Product Size/Effort Ratio measure specifies the amount of software product produced relative to the amount of effort expended. This common measure of productivity is used as a basic input to project planning and also helps evaluate whether performance levels are sufficient to meet cost/schedule estimates.

Selection Guidance

Program Application

- Applicable to all domains. Commonly used in Weapons systems.
- Used for programs of all size. Less important for programs where little code is generated such as those using automatic code generation and visual programming environments.
- Not generally used for COTS or reused software.
- Estimates are often used during program planning. Both estimates and actuals are used during development and software support that focuses on incorporation of new functionality. Not generally used for maintenance programs focused on problem resolution.

Process Integration

- In order to compare productivities from different programs, the same definitions for size and effort must be used. For size, the same measure (e.g. Lines of Code) must be used as well as the same definition (e.g. logical lines). For the effort measure, the same labor categories and software activities must be included.
- The environment, language, tools, and personnel experience will affect productivity achieved.
- Productivity can also be calculated using software cost models. Many of these models include schedules as part of the productivity equation.
- To validly calculate productivity, the effort measure must correlate directly with the size measure. If, for example, effort for a component is covered but the component's size is not, productivity will be lower.

Usually Applied During

- Requirements Analysis (Estimates and Actuals)
- Design (Estimates and Actuals)
- Implementation (Estimates and Actuals)
- Integration and Test (Estimates and Actuals)

This Measure Answers Questions Such As

- Is the developer producing the product at a sufficient rate to meet the completion date?
 - How efficient is the developer at producing the software product?
 - Is the planned/required software productivity rate realistic?
-

Specification Guidance

Data Items Typically Collected

- Organization Name
- Build/Release
- Product Size (from Product Size and Stability measure)
- Language
- Effort

Typical Collection Level

- Build/Release
- Program
- Organization

Typical Reporting Level

- Build/Release
- Program
- Organization

Count Actuals Based On

- Completion of Build/Release
- Components Implemented
- Components Integrated and Tested

Measure - Functional Size/Effort Ratio

Measurement Category - Productivity

Issue - Development Performance

The Functional Size/Effort Ratio measure specifies the amount of functionality provided relative to the amount of effort expended. This measure is used as a basic input to project planning and also helps evaluate whether performance levels are sufficient to meet cost/schedule estimates.

Selection Guidance

Program Application

- Applicable to all domains. Commonly used in AIS systems.
- Useful when product size measures are not available. Useful during program planning, development, and software support phases

Process Integration

- In order to compare productivities from different programs, the same definitions for size and effort must be used. For size, the same measure (e.g. Function Points) must be used as well as the same counting practices. For the effort measure, the same labor categories and software activities must be included.
- The environment, language, tools, and personnel experience will affect productivity achieved.
- Productivity can also be calculated using software cost models. Many of these models include schedule as part of the productivity equation.
- To validly calculate productivity, the effort measure must correlate directly with the size measure. If, for example, effort for a function is covered but the functional size is not, productivity will be lower.
- Useful early in the program, before actual product size data is available.

Usually Applied During

- Requirements Analysis (Estimates and Actuals)
- Design (Estimates and Actuals)
- Implementation (Estimates and Actuals)
- Integration and Test (Estimates and Actuals)

This Measure Answers Questions Such As

- Is the developer producing the software at a sufficient rate to meet the completion date?
 - How efficient is the developer at producing the software?
-

Specification Guidance

Data Items Typically Collected

- Organization Name
- Build/Release
- Functional Size (from Functional Size and Stability measure)
- Labor Hours (from Labor Hours measure)

Typical Collection Level

- Build/Release
- Program
- Organization

Typical Reporting Level

- Build/Release
- Program
- Organization

Count Actuals Based On

- Completion of Build/Release
- Functions Implemented
- Functions Integrated and Tested

Measurement Category - Rework Issue - Development Performance

Rework measures address the amount of rework due to defects in completed work products (documents, design, code, test plans, testing, etc.). Rework measures are used to evaluate the quality of the software products and development process. They provide information on how much software must be recoded and how much effort is required for corrections.

Program Application

- Measurement category applicable to most programs.
- Applicable to most software process models. Not generally used in rapid prototype processes.
- Useful during development and software support phases.

Measures Included in this Category

- Rework Size
- Rework Effort

Limitations

- Data collection is difficult and often labor intensive.
- Most accounting systems do not include rework effort in separate accounts (in order to track rework effort at least one cost account needs to be added).
- Requires a consistent process for effort allocation to rework/non-rework categories.

Related Measurement Categories

- Product Size and Stability
- Defect Profile
- Complexity

Example Indicator

- Rework Effort (PSM Part 3, Section 3.18)

Measure - Rework Size

Measurement Category - Rework

Issue - Development Performance

The Rework Size measure counts the number of lines of code that must be changed to fix identified defects. This measure helps in assessing the quality of the initial development effort, by indicating the amount of total code which must undergo rework.

Selection Guidance

Program Application

- Applicable to all domains.
- Applicable to most development processes. In a rapid prototype process, it is only applicable to the 'final' version of the software product.
- Not generally used for non-developed code such as COTS.
- Useful during development and software support phases.

Process Integration

- Very difficult to collect. Most configuration management systems do not collect information on changes to the size of code or reason for the change (rework).
- Rework size should only include code changed to correct defects. Changes due to enhancements are not rework.
- Rework cost and schedule should be included in the development plan

Usually Applied During

- Implementation (Actuals)
- Integration and Test (Actuals)

Specification Guidance

Data Items Typically Collected

- Component Name
- Number of Lines of Code Changed Due to Rework

Size May be Measured As

- Lines of Code
- Components

Typical Collection Level

- Unit or equivalent

Typical Reporting Level

- CSCI or equivalent

Count Actuals Based On

- Release to configuration management
- Passing inspection
- Passing unit test

This Measure Answers Questions Such As

- How much code had to be changed as a result of correcting defects?
 - What was the quality of the initial development effort?
 - Is the amount of rework impacting cost and schedule?
-

Measure - Rework Effort

Measurement Category - Rework

Issue - Development Performance

The Rework Effort measure counts the amount of work effort expended to find and fix software defects. Rework effort may be expended to fix any software product, including those related to requirements analysts, design, code, etc. This measure helps assess the quality of the initial development effort, and identify products and software activities requiring the most rework.

Selection Guidance

Program Application

- Applicable to all domains.
- Applicable to most development processes. In a rapid prototype process, it is only applicable to the 'final' version of the software product.
- Not generally used for effort associated with non-developed code such as COTS.
- Useful during development and software support phases.

Process Integration

- Difficult to collect. Some cost accounting systems do not collect information on rework effort.
- For basic tracking, a single WBS/cost account should be created to track all rework effort (per organization). For more advanced tracking, multiple WBS/cost accounts should be created to track rework at the component and/or activity level.
- Rework effort should only include effort associated with correcting defects. Effort expended due to incorporation of enhancements is not rework.
- Rework cost and schedule should be included in the development plan.

Usually Applied During

- Requirements Analysis (Actuals)
- Design (Actuals)
- Implementation (Actuals)
- Integration and Test (Actuals)

This Measure Answers Questions Such As

- How much effort was expended on fixing defects in the software product?
 - Which software development activity required the most rework?
 - Is the amount of rework impacting cost and schedule?
-

Specification Guidance

Data Items Typically Collected

- Organization
- WBS or Task Element
- Labor Hours

Typical Collection Level

- WBS or task element

Typical Reporting Level

- WBS or task element
- Organization

Count Actuals Based On

- End of financial reporting period

Measurement Category - Technology Impacts

Issue - Technical Adequacy

Technology Impacts measures quantify the positive or negative impacts of new technology used on the program. They are defined and selected to track the effect of highly leveraged software technologies. They can include functionality delivered, the amount of code developed, the defect discovery rates, and required replans. Technology Impact measures provide an indication of the relative effects of developing or maintaining software in different environments.

Program Application

- Measurement category applicable to many programs.
- Applicable to all software process models.
- Useful during program planning, development, and software support phases.

Measures Included in this Category

- No pre-defined measures

Limitations

- It is very difficult to attribute problem impacts to one particular software technology. Measures in this category, however, do provide objective insight.

Related Measurement Categories

- Productivity
- Defect Profile
- Product Size and Stability
- Functional Size and Stability
- Milestone Performance

Example Indicator

- Software Origin (PSM Part 3, Section 3.19)

CHAPTER 3 – MEASUREMENT SELECTION AND SPECIFICATION EXAMPLE

Chapters 1 and 2 provide the guidance and detailed information required to select and specify program measures. This chapter provides an example program scenario to show how to use this information to actually select a set of software measures using. Other examples of measurement selection and specification are provided in the Case Studies (Part 5).

3.1 PROGRAM SCENARIO

During the program planning phase of a large weapons system software upgrade, the program office learned that the updated system would have to be deployed earlier than originally planned. The planning efforts completed to date clearly indicated that there were already some significant constraints with respect to schedule, and this change increased the risk even further. Given that the program was already getting a considerable amount of external visibility, the Program Manager decided that he would rely on a well implemented measurement process to provide him with the software information that he felt he would need to properly manage the program issues.

The program was required to follow many of the DoD acquisition reform requirements, and as such, a number of new technical and management approaches were going to be implemented. The Program Manager established several planning related Integrated Product Teams (IPT), and formally tracked the issues and risks associated in the program.

The primary risk to the program was the short development schedule. The program had originally been “sold” on the new mission capabilities and the use of advanced technologies, which increased the overall technical risk of the software development, and now the need to deliver the system earlier than expected was of increased concern.

The software engineering IPT was involved in tailoring the measurement process. To start, the key characteristics of the

program were identified and documented. This information is summarized as follows:

- Large real-time weapons system
- Existing system baseline
- Approximately 1.5 Million lines of source to be implemented
- Multiple software languages - Ada, C, and Assembly
- Multiple developers working under a prime contractor responsible for system integration
- Average software processmaturity across all organizations
- Constrained funding limits

Due to the schedule risk and the large amount of functionality that had to be implemented in a short time, the program office required that the developer maximize the use Commercial Off the Shelf (COTS) software components, reuse a considerable amount of existing legacy software, adopt an open systems architecture, and apply commercial software process standards.

3.2 MEASUREMENT SELECTION SUMMARY

Following the *PSM* measurement selection and specification approach, the program office prioritized the program specific software issues and allocated them to the appropriate Common Issues. The measurement analyst then reviewed the *PSM* Measurement Tables and determined the key measurement categories and associated measures that were needed to best provide the required information. Figure 2-5 summarizes the results of their selection process, and lists the primary measures that were selected.

| Measurement | | |
|-------------------------|-------------------------------|---|
| Issues | Categories | Measures |
| Schedule and Progress | Milestone Performance | Milestone Dates |
| | Work Unit Progress | Components Integrated and Tested Requirements Allocated Requirements Tested Problem Reports Resolved |
| | Incremental Capability | Build Content - Function |
| Resources and Cost | Effort Profile | Effort Allocation |
| Growth and Stability | Functional Size and Stability | Requirements |
| Product Quality | Defect Profile | Problem Report Trends |
| Development Performance | Productivity | Product/Effort Ratio |
| Technical Adequacy | Technology Impacts | Code Growth by Source |

Figure 3-1 Issues, Measurement Categories, and Selected Measures

The software engineering IPT felt that the importance of the schedule and progress issue required the implementation of a number of related measures. Along with Milestone Dates, which would already be available from the program management process, they selected a number of measures in the Work Unit Progress category. These measures would provide incremental completion information for each software activity. In this way the program team would be able to track the progress of each key activity to completion. The Work Unit Progress measures that were selected were the ones most useful in tracking the development activities given the large amount of COTS and reused code that would be implemented. Thus, the focus was on the selection of requirements oriented measures, and measures which provided progress information for integration and test rather than for design and implementation. This approach provided more useful information given the extent of COTS and reused software to be used.

Since the development plan was based on highly leveraged software technology, there were issues with respect to the actual impact of

the COTS and reused code on the development schedule and overall development productivity. As such, the team selected the Product Size/Effort Ratio measure under the Productivity category to measure and evaluate overall development performance. They also defined a Technology Impact measure which would show if the relative amount of developed to non-developed code was changing. Both of these measures required the use of lines of code for non-developmental software. The Labor Hours measure was selected to support the productivity assessment.

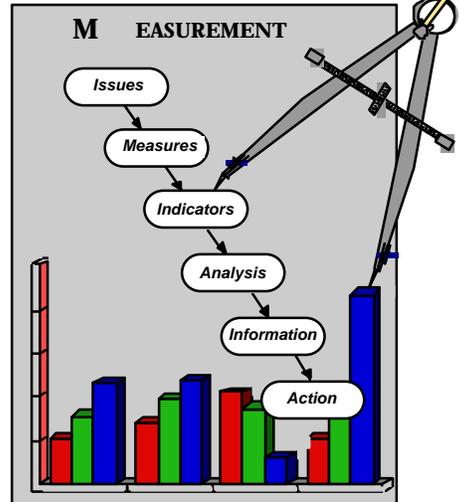
Given the constrained time frame and the large amount of functionality which had to be delivered, the team felt that requirements growth would be a major issue. Any significant growth in the requirements could significantly impact cost and schedule. They therefore selected the Requirements measure under the category of Functional Size and Stability.

Given the overall planning constraints and the criticality of the new functions, product quality was of concern. To address this, both the Problem Report Trends and Problem Report Aging measures were selected, to be applied to all software, inclusive of the COTS and reused code.

At the completion of the measurement selection process the IPT had defined a basic list of measures which directly addressed the issues that the program faced. After each measure was specified, the overall set of measurement requirements was conveyed to the developer for integration into the software process.

PRACTICAL

SOFTWARE



ANALYSIS TECHNIQUES AND EXAMPLES

PART 3

ANALYSIS TECHNIQUES AND EXAMPLES

Part 2 of Practical Software Measurement stresses the importance of selecting the measures which best address the software issues on a particular program. This part of PSM explains how to actually apply measurement: using measurement to gain insight into the issues that are of greatest concern for the program. Measurement is only useful when it provides information about the issues which helps to make program-related software decisions.

Part 1 of Practical Software Measurement describes the overall measurement process and the various techniques a measurement analyst should use when generating and analyzing indicators. This part of the Guide provides more specific guidance on defining indicators and generating visual representations of indicators (i.e., graphs). It provides examples of how measurement indicators can be generated and applied to analyze software issues.

This part of the Guide is organized into four chapters:

- Chapter 1 - Measurement Application Overview, summarizes the process described in Part 1 for collecting, analyzing, and reporting measurement data and information.*
- Chapter 2 - Indicator Representation, describes how to produce graphs and reports which help to visually represent an issue.*
- Chapter 3 - Single Indicator Examples, includes, for each measurement category described in PSM, an example of how a measurement indicator can be defined and analyzed.*
- Chapter 4 - Integrated Indicator Examples, includes examples of how different indicators can be used together to analyze specific program issues.*

The application guidance provided in this part of PSM is based on actual experience. Many of the examples used are taken directly from actual DoD programs.

TABLE OF CONTENTS

| | |
|--|------------|
| CHAPTER 1 – MEASUREMENT APPLICATION OVERVIEW..... | 183 |
| 1.1 Collect and Process Data..... | 183 |
| 1.2 Define And Generate Indicators..... | 184 |
| 1.3 Analyze Issues..... | 185 |
| 1.4 Report Results..... | 185 |
| 1.5 Take Action..... | 186 |
| CHAPTER 2 – INDICATOR REPRESENTATION..... | 187 |
| CHAPTER 3 – SINGLE INDICATOR EXAMPLES..... | 191 |
| 3.x Indicator Name..... | 192 |
| 3.1 Milestone Progress Indicator..... | 194 |
| 3.2 Design Progress Indicator..... | 196 |
| 3.3 Schedule Variance Indicator..... | 198 |
| 3.4 Incremental Build Content Indicator..... | 200 |
| 3.5 Effort Allocation Indicator..... | 202 |
| 3.6 Staff Experience Indicator..... | 204 |
| 3.7 Cost Profile Indicator..... | 207 |
| 3.8 Resource Utilization Indicator..... | 209 |
| 3.9 Software Size Indicator..... | 211 |
| 3.10 Requirements Stability Indicator..... | 213 |
| 3.11 Response Time Indicator..... | 215 |
| 3.12 Problem Report Status Indicator..... | 217 |
| 3.13 Problem Report Aging Indicator..... | 218 |
| 3.14 Defect Density Indicator..... | 221 |
| 3.15 Software Complexity Indicator..... | 223 |
| 3.16 Software Process Maturity Indicator..... | 225 |
| 3.17 Software Productivity Indicator..... | 227 |
| 3.18 Rework Effort Indicator..... | 230 |
| 3.19 Software Origin Indicator..... | 233 |
| CHAPTER 4 – INTEGRATED INDICATOR EXAMPLES..... | 235 |
| 4.1 Design Completion Analysis..... | 236 |
| 4.2 Test Completion Analysis..... | 238 |
| 4.3 Readiness for Delivery Analysis..... | 240 |
| 4.4 Maintenance Analysis..... | 242 |

CHAPTER 1 – MEASUREMENT APPLICATION OVERVIEW

Utilizing measurement data to analyze program issues is an iterative, versus a one-time, process. Data is continuously collected and measurement indicators are defined and generated at regular intervals throughout a program. Although each issue may require different data and different indicators, the basic analysis process is the same. Figure 1-1 describes this process:

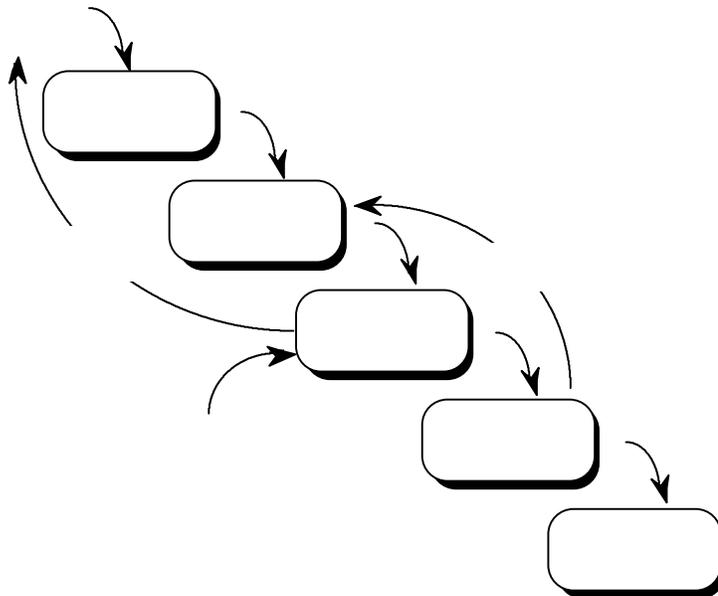


Figure 1-1 Measurement Application process

This version of *PSM* primarily uses examples to illustrate how indicators are defined and applied to help identify and resolve program issues. The next version of the Guide (v3.0) will provide more detailed guidance on how issues can be systematically analyzed

1.1 COLLECT AND PROCESS DATA

After the measurement process is tailored for a program, the specific measures, associated data, and the implementation requirement should be defined. However, how data is actually

collected is very dependent on the developer's software process, project organization, and existing tools. Planning data will most often be collected from project scheduling tools, spreadsheets, or forms. Actual data will most often be collected from sources such as project tracking tools, problem and defect tracking databases, static and dynamic analysis tools, time reporting systems, and configuration management systems.

Data may be reported at different levels of detail and at different frequencies than it was collected. However, data at a reasonably low-level of aggregation should be supplied to the program management office in order to allow the required analyses to be conducted. The program management office should verify the data before using it to assess program status. Verification involves looking for missing data items, cross-checking the data for accuracy, and ensuring data is being provided at the proper level of detail. Evaluating the accuracy and integrity of the data being supplied actually reveals a lot about the developer's process and the issues that may later impact the program.

1.2 DEFINE AND GENERATE INDICATORS

Measurement indicators are the primary mechanisms used for issue analysis and reporting. An indicator is a measure or combination of measures that provides insight into a software issue or concept. Multiple indicators may often be needed to thoroughly understand the status of an issue. Many measurement indicators are produced by using an *analysis technique* that compares one or more *measured values* to corresponding *expected values*

Measured values are the actual measurement data collected and reported by the developer. Examples include hours of effort expended or lines of code produced. *Expected values* are planned or historical measurement data such as planned milestone dates, target level of reliability or required productivity. An expected value may also reflect a standard rule of thumb or threshold, such as the generally recognized rule of thumb which recognizes 10 as the maximum desired value for a component's cyclomatic complexity score. A series of data points is often provided for both measured and expected values.

An indicator is produced by applying an *analysis technique* to the data. The technique usually involves the application of a graphing technique or a mathematical operation, or both, to the data, which results in a comparison of the measured and expected values. Various indicators may need to be defined and generated at various times throughout a program to effectively analyze an issue. The combination of measurement data used, the issue being analyzed, and the insight desired all influence the generation of an indicator. The name of the indicator should reflect these elements.

1.3 ANALYZE ISSUES

Analyzing a software issue involves using indicators to identify unexpected situations. Then, information received from the analysis is coupled with other program information and personal experience. Insight is achieved by combining information from these three sources; this insight is then used to assess the impact the situation may have on desired program outcomes.

Issue analysis also involves using problem solving skills to gain an understanding of why the situation exists so that the proper corrective action can be initiated. As information is gathered regarding an issue, actual findings will often lead to new and different types of analyses, utilizing new and different indicators. In this regard, the analysis process must be dynamic so that the underlying causes of problems can be localized and identified.

Rules of thumb are often used to evaluate whether a variance between measured and expected values represents a situation requiring further action. For instance, lessons learned from past projects in one organization may dictate that whenever a variance in either schedule or budget is greater than 10 percent, the program manager should investigate the situation and take corrective action. Organizations should attempt to define analysis rules of thumb wherever possible, as this increases the likelihood that the proper level of analysis will be performed consistently for each program.

1.4 REPORT RESULTS

Once the status of an issue is understood, the findings and recommendations should be reported to program management.

This is normally done via a briefing or report. The following information should be communicated:

- Overall status
- Specific situations or problems discovered
- Recommendations
- Identification of potential new issues

Results should be used by the program manager at program status review meetings, at major milestone reviews, and ideally throughout the program. Adequate time should be allocated in advance to collect and process the data, analyze the issues, prepare reports, and conduct the briefings. Care should be taken to insure that all information conveyed in the reports can be explained. The developer should also be briefed on the analysis results.

1.5 TAKE ACTION

Action must be taken to realize any benefit from measurement. The goal of project measurement is to identify problems and take corrective action *in time to affect the outcome of the project.* Actions may be initiated by either the developer or the program manager.

Actions, once taken, should be tracked to assess the effectiveness of the action and to ensure that the action does positively affect outcomes.

CHAPTER 2 – INDICATOR REPRESENTATION

Measurement indicators can provide valuable insight into a particular issue. Indicators help to explain both what **is** happening and what **may** happen with respect to the issue area. Simple charting techniques can be used to produce graphical representations of the indicators. The ability to extract the pertinent information contained in the measurement data can be improved with proper selection and use of these charting techniques. The two most commonly used charting techniques are described below.

Line Charts, sometimes called run charts, provide a way to represent a series of measurement data values over time. A series may contain either measured or expected values; each value in the series is reported for a specified point in time. Values are plotted as points on the graph, and the values are connected with lines to help show progress or a trend. For example, a line chart may include one series of planned values which shows the cumulative number of units scheduled to complete coding and unit testing each month, over a six month period of time. As units are actually completed, a second series of values is tallied and added to the graph each month to allow a comparison between measured and expected values.

Bar Charts provides a way to represent the count or frequency of a set of components or events. Bars are typically drawn vertically with the Y-axis indicating the units or events being counted. Each bar contains data associated with a class or grouping of data. Understanding the distribution of the data across the groups is often useful. For example, the bar might represent the number of defects detected: 1) for each product component, or 2) within each phase of the software development life cycle. Sets of bars can also be used to compare two series, such as measured and expected values. Histograms and Pareto charts are two special types of bar charts.

Good graphic displays of indicators facilitate communication of measurement results. Therefore, graphs must not be too complex. Each graph should convey a clear message. It is better to have many graphs than many messages on one graph, especially when getting started. Some guidelines for developing effective graphs include the following:

- Provide a **descriptive title** identifying the program or system name, type of data, and component or CSCI (if applicable) represented by the data.
- Show an **as-of** line or date indicating the reporting period represented by this data. Many graphs will show plans or projections beyond the as-of date.
- **Axis labels** should include type of units and scale markers (e.g., dates or counts).
- Provide **indicators of major milestones** that correspond to the interval plotted when showing time trends.
- Use the **connect-the-dots technique** rather than curve-fitting to show trends.
- Use **contrasting styles** for lines, bars, and data points that represent different groups of data.
- **Label line, bar, and data points** directly on the figure, if possible. Otherwise, use a key that associates a label with each contrasting style of line, bar, or data point.
- Identify the **source of the data** Include the version number of documents.
- **Use similar conventions for all reports** For example, always use solid boxes for actuals and open boxes for plans.
- Adjust the horizontal axis to **show the expected range** of the data plotted.
- **Label significant events and trends** in the data.
- Be careful that the **use of percentages** does not hide significant trends in the data.
- Use the **same axes** on both graphs when comparing two graphs.

Figure 3-2 shows a graph that illustrates the guidelines listed above.

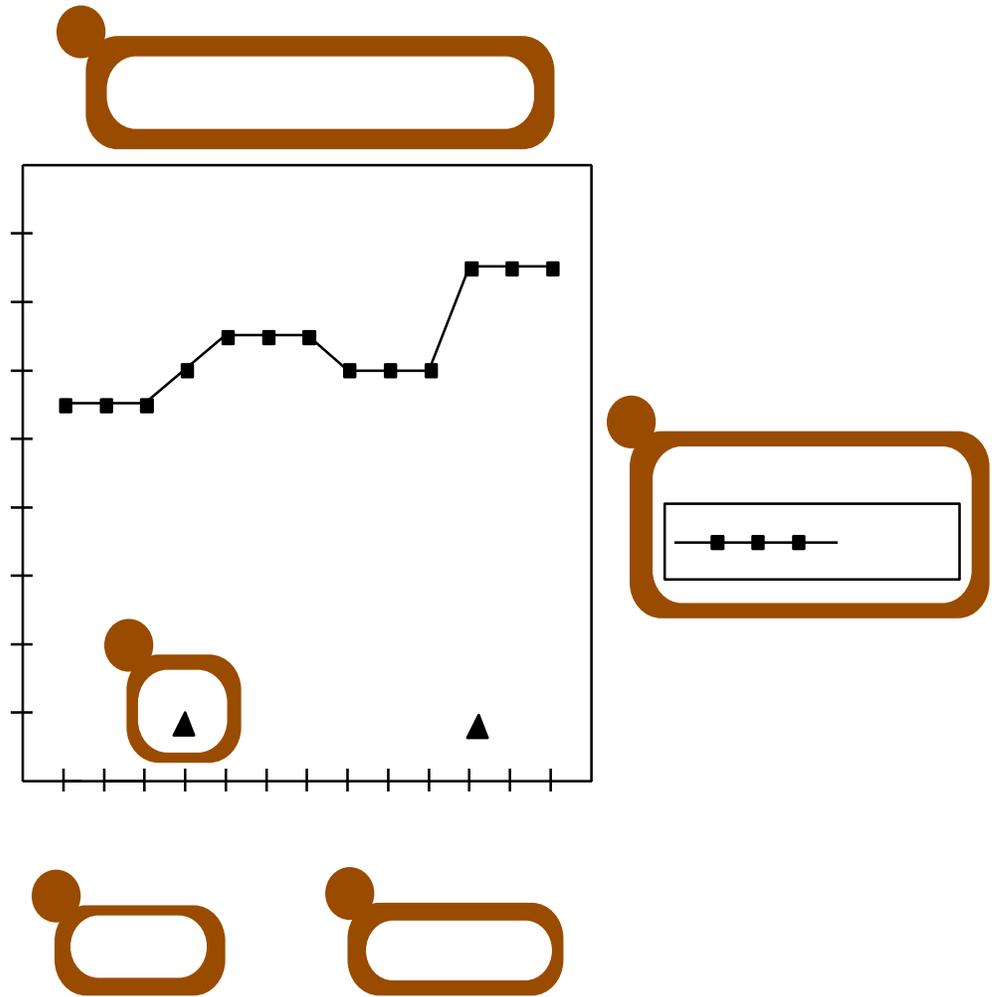


Figure 3-2. Size Requirements

CHAPTER 3 – SINGLE INDICATOR EXAMPLES

This chapter contains examples of single indicators used to analyze the issues covered in this document. *These are examples only and do not represent a definitive set that should be applied to all programs.* At least one sample indicator has been included for each PSM measurement category. Examples are presented using a two-page format, which contains general descriptions, a visual representation of the indicator produced from detailed measurement data, and brief explanations of how the indicator was generated and how the corresponding issue might be analyzed. Many of the examples include more than one graph. The first two pages describe the standard two-page format used throughout the remainder of the chapter .

The following indicators are included:

| Issue | Indicator | Identifier |
|-----------------------|---------------------------------|------------|
| Schedule and Progress | Milestone Performance | 3.1 |
| | Design Progress | 3.2 |
| | Schedule Variance | 3.3 |
| | Components Delivered | 3.4 |
| Resources and Cost | Effort Allocation | 3.5 |
| | (Staff) Domain Experience | 3.6 |
| | Cost Profile | 3.7 |
| | Test Lab (Resource) Utilization | 3.8 |
| Growth and Stability | Software Size | 3.9 |
| | Requirements Stability | 3.10 |
| | Response Time | 3.11 |
| Product Quality | Problem Report Status | 3.12 |
| | Problem Report Aging | 3.13 |
| | Defect Density | 3.14 |
| | Code Complexity | 3.15 |
| Performance | Process Maturity | 3.16 |
| Development | Productivity | 3.17 |
| | Rework Effort | 3.18 |
| Technical Adequacy | Software Origin | 3.19 |

3.X INDICATOR NAME

| | |
|-----------------------------|---|
| Issue | <i>Issue Name. The name reflects the measurement data used, the issue being analyzed, and the insight desired.</i> |
| Category | <i>Category Name.</i> |
| Selected Measure | <i>Name of the measure selected for use in this example.</i> |
| Description | <i>A description of the selected indicator, including its purpose and the questions it can help answer.</i> |
| Example Graph | <i>A description of the sample graph(s) on the opposite page, including how it was produced. Note: Some examples contain an analysis of the indicator at more than one level of detail and therefore contain more than one graph.</i> |
| Feasibility Analysis | <i>Instructions for evaluating the feasibility of the planned values used in this example.</i> |
| Performance Analysis | <i>A description of how the indicator depicted in the example might be analyzed to obtain information about the corresponding issue.</i> |
| Lessons Learned | <i>Helpful information such as the suggested reporting level, how much variance is typically considered acceptable, and which factors often interfere with analysis of this indicator.</i> |

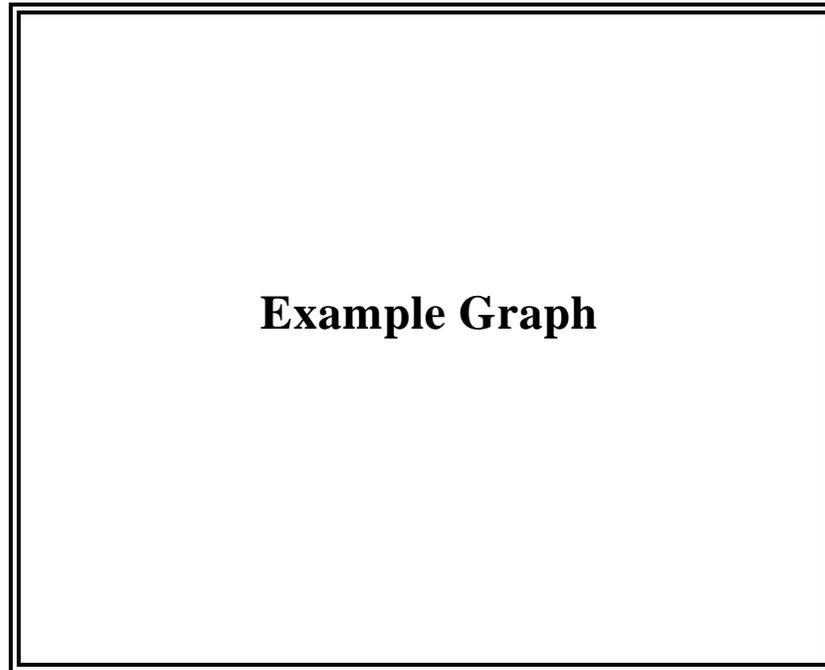


Figure 3-xa. Appropriate Graph Caption

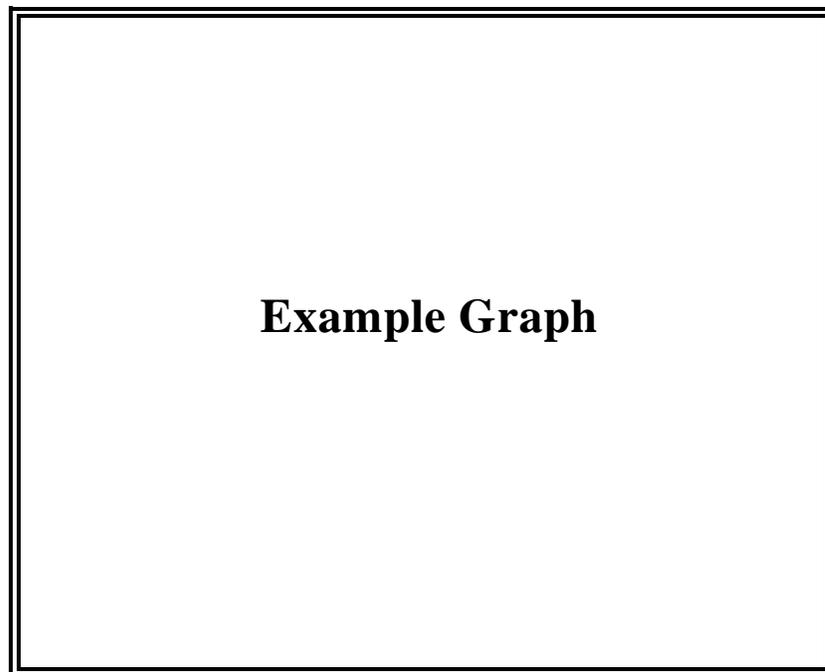


Figure 3-xb. Appropriate Graph Caption

3.1 MILESTONE PROGRESS INDICATOR

| | |
|-----------------------------|---|
| Issue | Schedule and Progress |
| Category | Milestone Performance |
| Selected Measure | Milestone Dates |
| Description | Helps identify the current status of major project events, and allows assessment of the impact of potential or actual schedule slips on future activities and milestones. |
| Example Graph | A Gantt chart was used to present the information. Milestones symbols were derived from single dates while start and stop dates were used to produce activity bars for major phase-level activities. An “as of” line was added to help identify which actuals were included in this chart. |
| Feasibility Analysis | Evaluate each activity’s planned start and end dates for reasonableness. The evaluation should include an assessment of whether all activities are included, what activities affect the critical path, and the amount of overlap between various activities. |
| Performance Analysis | Figure 3-1 shows a delayed program start resulting in a significant slip in Build 1 of the software. Based on known dependencies, the slips projected for Build 2 activities and milestones have been incorporated in the chart. Further analysis of staffing levels, work unit progress, and defect rates should help uncover the reasons for any further schedule slips. The impact of these schedule slips must be evaluated in light of project priorities and constraints. |
| Lessons Learned | Slips in activities and milestones on the critical path are of greatest concern due to the ripple effect in the later parts of the schedule. Ensure the graph contains a sufficient level of detail to monitor progress. If multiple builds or releases are planned, there should be separate activities and milestones for each build/release. |

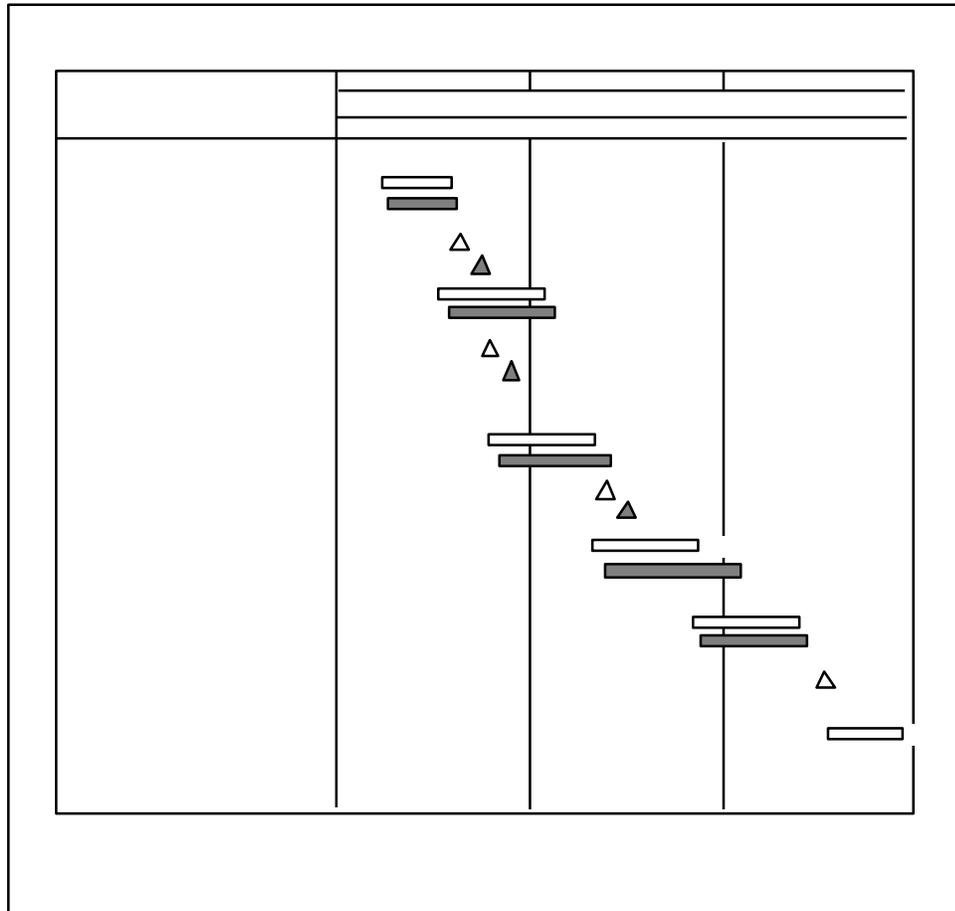


FIGURE 3-1. SOFTWARE DEVELOPMENT MILESTONE PROGRESS

3.1 DESIGN PROGRESS INDICATOR

| | |
|-----------------------------|--|
| Issue | Schedule and Progress |
| Category | Work Unit Progress |
| Selected Measure | Components Designed |
| Description | Helps identify or predict schedule slips and uncover design size growth, by comparing the number of units completing design to the number of units scheduled for design completion over time. |
| Example Graph | <p>Overall design progress (Figure 3-2a) was graphed using a line chart containing cumulative measures for the original plan, the recent replan, and the actual units designed to date. Each point is calculated by adding the number of units allocated for the reporting period to their respective cumulative totals from the last reporting period.</p> <p>A bar chart (Figure 3-2b) was used to perform a more detailed analysis of design progress by CSCI. The second bar in each series represents the number of units that should be completed “as of” the reporting date, and provides the most meaningful comparison against actual progress.</p> |
| Feasibility Analysis | Check to ensure that initial design plans and any replans reflect the total number of CSUs as documented for the system. Look for a slope that is unusually steep. Also evaluate the planned rate of design completion in light of project realities such as staffing levels, experience, and requirements volatility. |
| Performance Analysis | Figure 3-2a indicates that design progress was behind the original plan at the end of April, resulting in a replan of the overall activity. Actual design progress has remained fairly close to the new plan. Further analysis at the CSCI level (Figure 3-2b) reveals that, while progress on the units for CSCI A and C is close to plan, less than one third of the units planned to date for CSCI B have completed design. Additional analyses of CSCI B’s staffing levels and experience, rework effort, and changing requirements should help identify the cause of this deviation. |
| Lessons Learned | Be careful of major changes in the rate of progress. Once an actual trend line is established, it is very difficult to modify that rate of completion. A 10% cumulative or 20% per period actual deviation from the plan should be considered significant. |

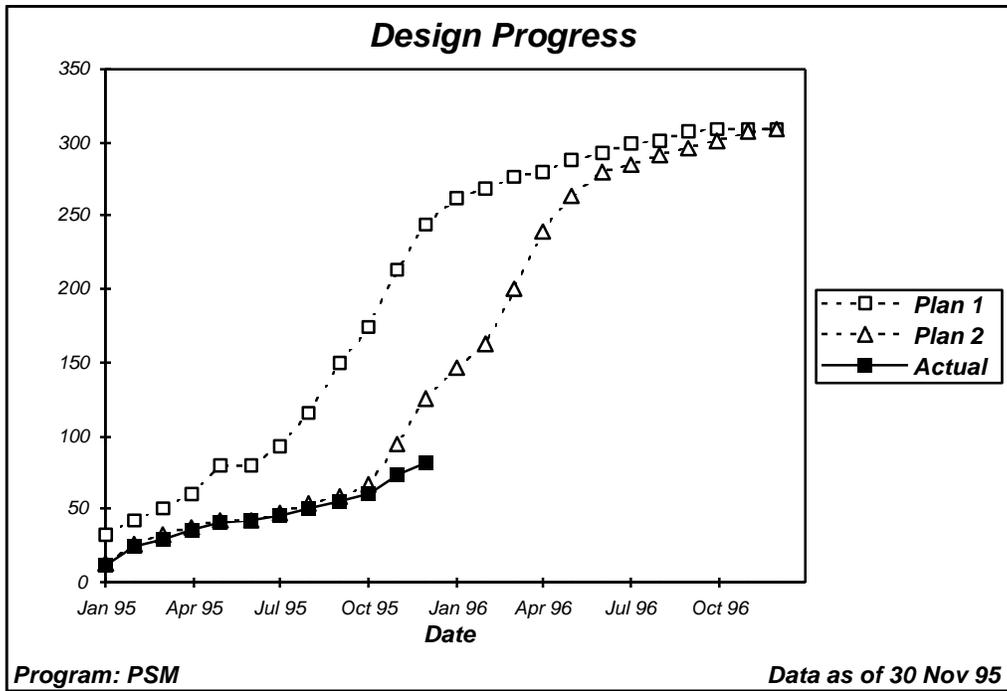


Figure 3-2a. Design Progress by Date

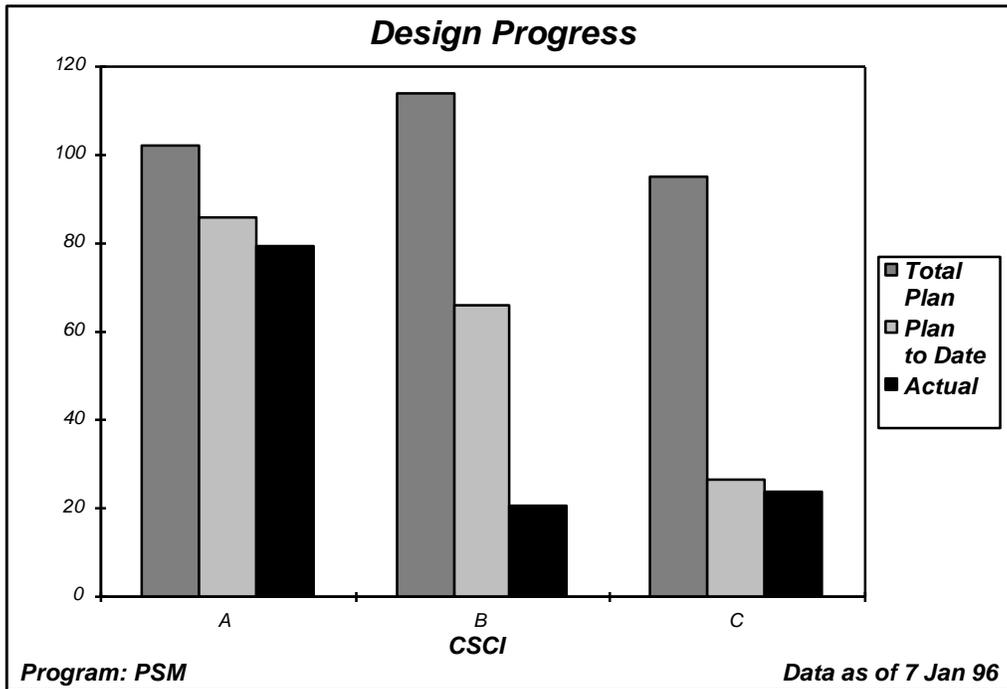


Figure 3-2b. Design Progress by CSCI

3.3 SCHEDULE VARIANCE INDICATOR

| | |
|-----------------------------|---|
| Issue | Schedule and Progress |
| Category | Schedule Performance |
| Selected Measure | Schedule Variance |
| Description | Provides an indication of schedule progress based on dollars budgeted per WBS element. The measure addresses the developer's ability to complete scheduled activities within the planned time frame and indicates the extent that the developer is ahead or behind schedule. |
| Example Graph | The schedule variance (SV) was graphed using a line chart. This is calculated as $SV = BCWP - BCWS$ where BCWP is the budgeted cost of work performed and BCWS is the budgeted cost of work scheduled. Data below the 0 line is an indication that the program is behind schedule, while values above the line indicate the program is ahead of schedule. |
| Feasibility Analysis | Not applicable. |
| Performance Analysis | Figure 3-3 indicates that initial progress was behind schedule until August. The sudden, dramatic drop in September is the greatest concern, however. Did a large number of people leave the program? Further analysis of staffing levels and work progress should help identify the cause. A replan may be necessary. |
| Lessons Learned | Investigate large deviations from the schedule. Schedule variance lines that continue to decrease over multiple months should be monitored closely. When schedule variance becomes large, a rebaseline should occur and a realistic plan should be established. |

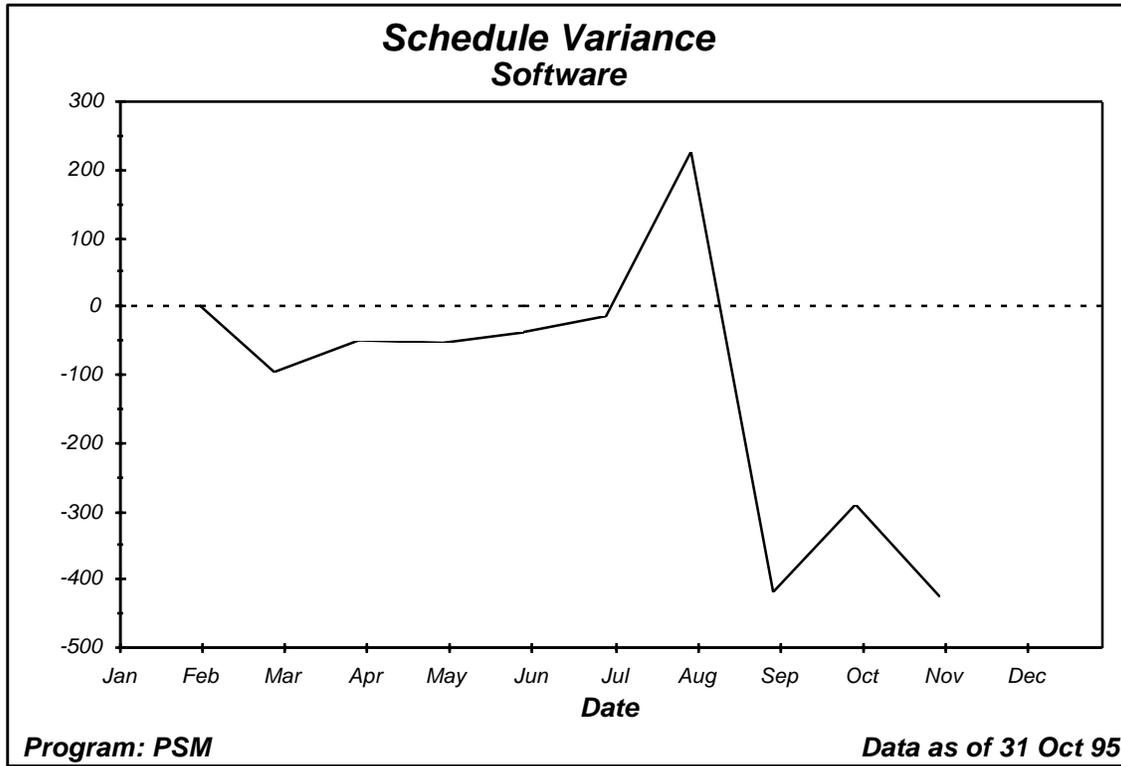


Figure 3-3. Schedule Variance

3.4 INCREMENTAL BUILD CONTENT INDICATOR

| | |
|-----------------------------|--|
| Issue | Schedule and Progress |
| Category | Incremental Capability |
| Selected Measure | Build Content - Component |
| Description | When multiple builds are planned, this indicator helps determine if capability is being delivered to the customer or to integration and test on schedule. The graph compares the number of components in each build that should have been delivered to date, against the number actually delivered. |
| Example Graph | The bar chart was produced by summarizing, for each build, the number of components 1) originally planned for delivery to date, 2) planned for delivery to date based on the latest plan, and 3) actually delivered to integration and test. |
| Feasibility Analysis | Evaluate whether the distribution of components across incremental builds look is reasonable, considering overlapping work effort and the likelihood of slippage. Also, ensure that the sum of each build's planned number of components is equal to the total number of components scheduled for the final release. |
| Performance Analysis | Figure 3-4 shows that components in both Builds 1 and 2 were deferred to Build 3, increasing its size by over 30%. While all components have been integrated to date, it is likely that these deferrals will result in delays in testing and may impact customer delivery milestones. Analyze test schedule and progress data to further assess the impact of these deferrals. |
| Lessons Learned | Deferments to later builds without adjustments to the schedule are of greatest concern. A 5% or greater variance in a single build or a 10% variance across two or more builds should be considered significant. Also, make sure that only components accounted for in the planned figures are included in actual counts. |

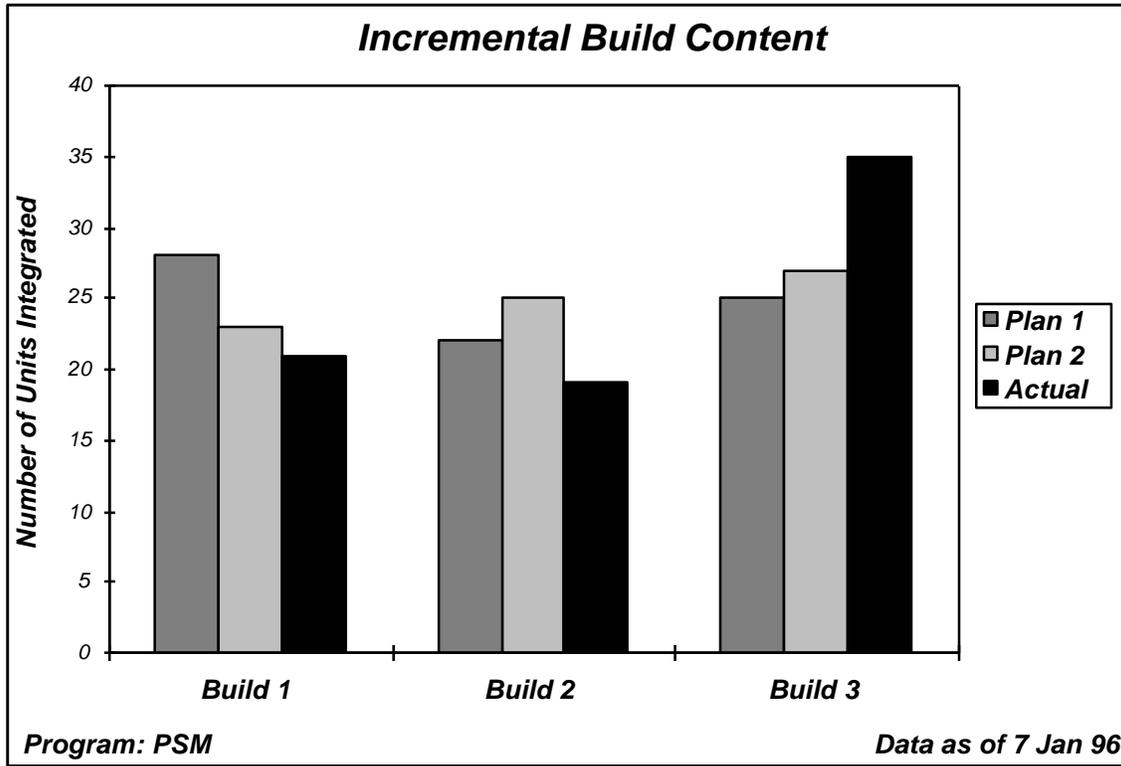


Figure 3-4. Incremental Build Content

3.5 EFFORT ALLOCATION INDICATOR

| | |
|-----------------------------|---|
| Issue | Resources and Cost |
| Category | Effort Profile |
| Selected Measure | Effort |
| Description | Used to assess the adequacy of planned effort and analyze the actual allocation of labor to development activities |
| Example Graph | <p>Total software effort was graphed using a line chart (Figure 3-5a) containing measures from the original plan, the April '95 replan, and actual staff months expended to date.</p> <p>A bar chart (Figure 3-5b) was used to obtain a more detailed view of effort allocation by software process activity. The current plan for each activity and the associated actual data was graphed as of the last reporting period.</p> |
| Feasibility Analysis | Evaluate whether the planned effort distribution is realistic. Additionally, check that the distribution of effort between the development activities is realistic. Insure that enough effort has been allocated to early requirements and design activities and to later testing activities, as these areas are often underestimated. |
| Performance Analysis | Figure 3-5a shows that actuals were initially below the original plan for several months. The developer had problems staffing the program due to delays in another program from which personnel were due to transfer. A replan was implemented, and actuals matched the new plan for several months, but then exceeded it. To assess the causes of this overrun, Figure 3-5b was drawn. This showed that additional effort was expended during software design. Further analysis of staffing and experience levels indicated that this was due to the developer's inexperience with the domain. |
| Lessons Learned | Check the rate of changes in effort data. Large numbers of people normally cannot be effectively added within a very short period. Large overruns during integration and test may be indicative of quality problems with the code - there may be significant defects that are delaying completion. |

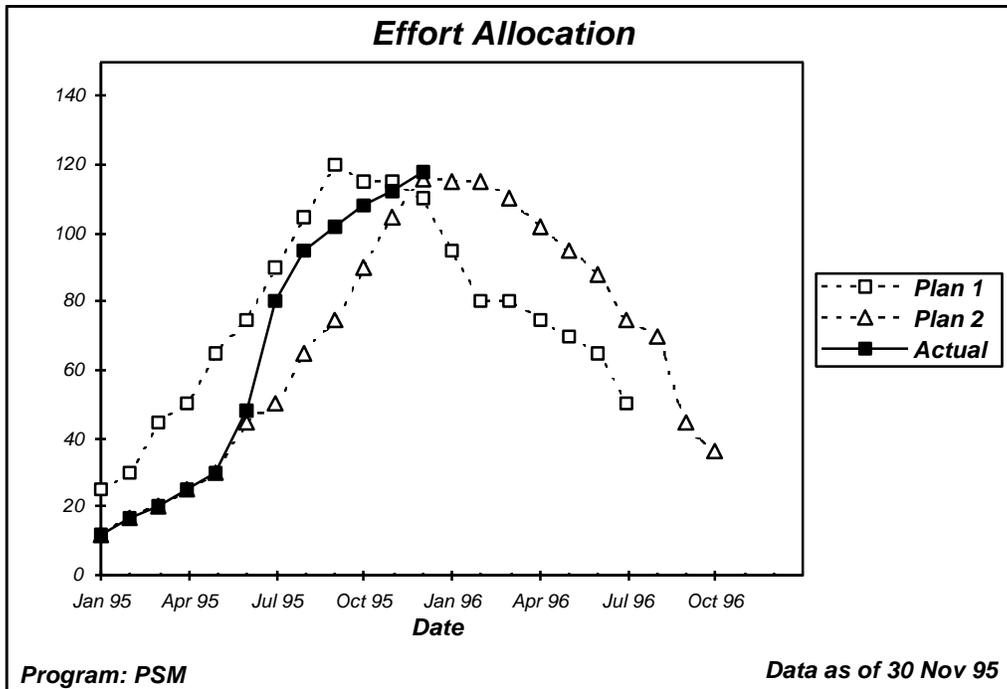


Figure 3-5a. Effort Allocation by Date

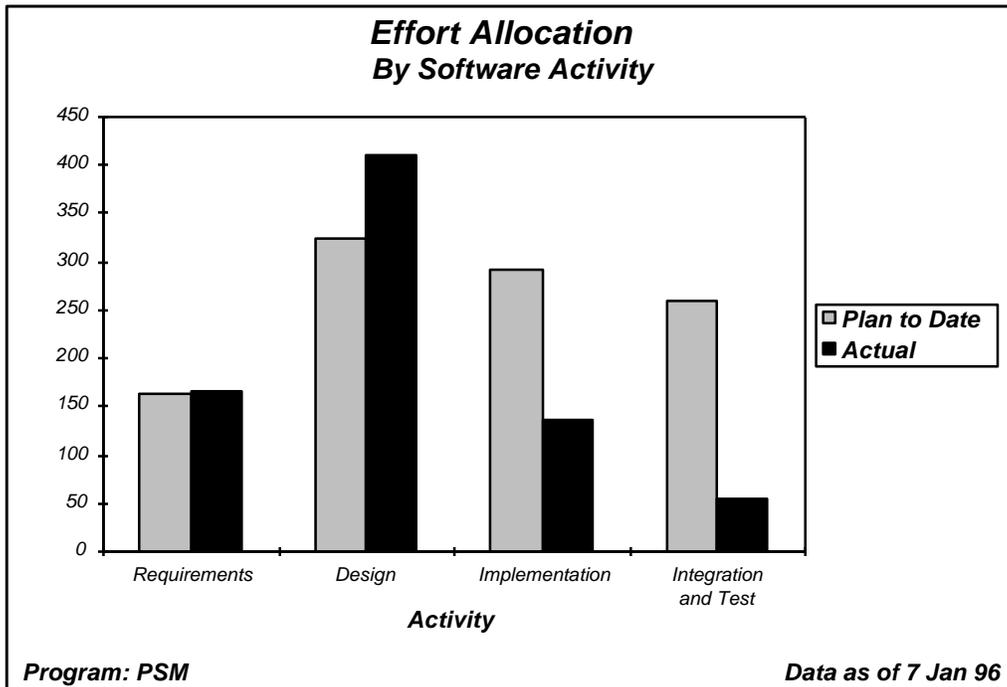


Figure 3-5b. Effort Allocation by Activity

3.6 STAFF EXPERIENCE INDICATOR

| | |
|-----------------------------|---|
| Issue | Resources and Cost |
| Category | Staff Profile |
| Selected Measure | Staff Experience |
| Description | Used to assess whether the personnel assigned to the project possess the domain experience necessary to produce a system that meets customer needs. The graph compares the development staff's years of real-time distributed systems experience to contract requirements. |
| Example Graph | Both series in the histogram were produced by sorting the development staff's experience data by the real-time distributed systems experience data item, and then tallying the number of staff members in each of six experience categories. This produced a distribution of experience levels which could be charted and compared to contract requirements. Since contract requirements specify an average number of years of real-time distributed systems experience (3 years), the staff's average at the current time was also calculated and displayed on the graph. |
| Feasibility Analysis | Using historical information from similar projects, assess whether the staff experience requirements to be specified in the contract are reasonable. Evaluate both what is possible (given the number of people available with relevant domain experience and their backgrounds in developing technology solutions) and what is needed. |
| Performance Analysis | Figure 3-6 shows that the development organization proposed and started the project with a staff reporting, on average, 3.43 years of real-time distributed systems experience. In order to further investigate recent schedule slippage and low productivity new staff experience data was requested. The new data reveals that, while staff size has remained constant in spite of turnover, experience levels of replacement staff members' have dropped. Average experience is now only 2.43 years. Additional analysis should be performed of skill requirements for the tasks remaining, and staff allocations. The program manager should decide if some staff members should be replaced with more experienced personnel or whether the experienced members can be leveraged to avoid further staffing changes. |

**Lessons
Learned**

Analysis of staff experience is usually only performed at major milestones on large projects, unless other analyses point to a staffing problem. Ensure that years of experience data is kept up to date. Be sure that experience obtained on the current project is considered when interim analyses are performed.

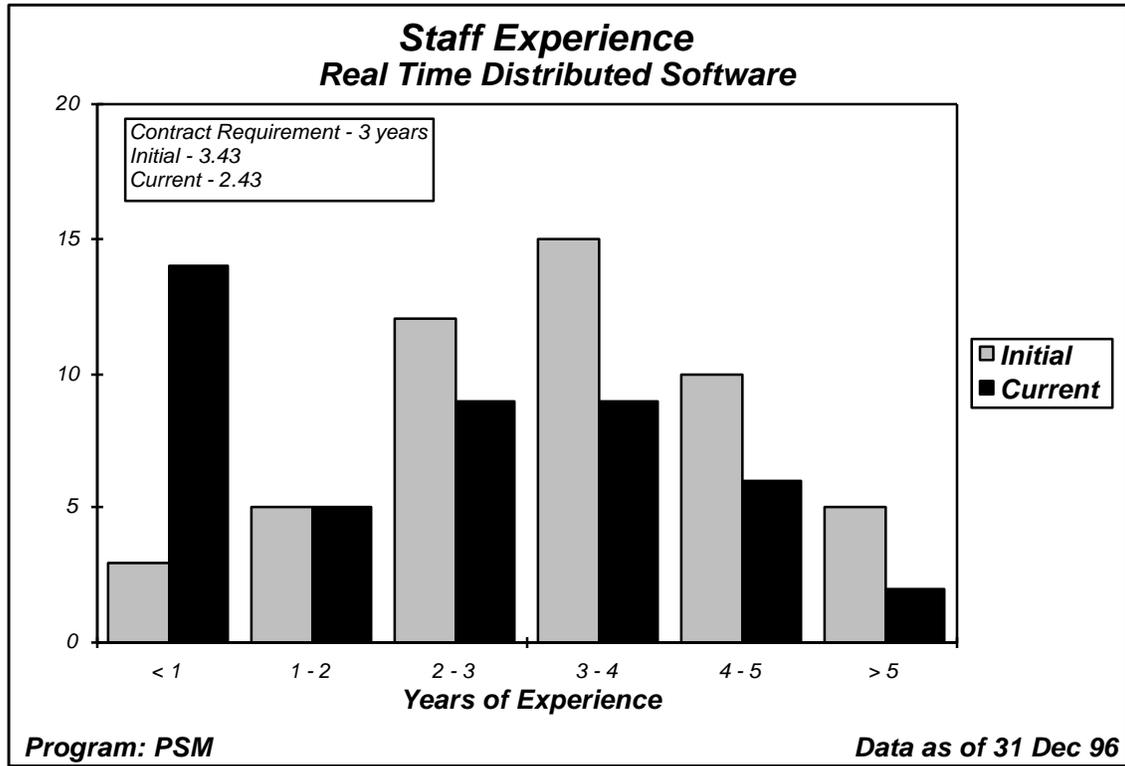


Figure 3-6. Staff Experience

3.7 COST PROFILE INDICATOR

| | |
|-----------------------------|---|
| Issue | Resources and Cost |
| Category | Cost Performance |
| Selected Measure | Cost Profile |
| Description | Used to evaluate costs based on planned versus expended costs. Used to assess whether a program can be expected to be completed within cost constraints. |
| Example Graph | Figure 3-7 shows a line chart used to present the cost information. In addition to cumulative plan and actual cost, the graph also contains the budgeted cost (top static line), and the funding profile (funding provided in increments) |
| Feasibility Analysis | Ensure that the planned cost is realistic over the period of performance. Large changes in the rate per period should be evaluated for feasibility. Figure 3-7 shows a relatively consistent planned expenditure rate. The funding profile should be evaluated to ensure that adequate funding has been provided to meet planned costs. Any delays in funding should be assessed for impact on the program. |
| Performance Analysis | Figure 3-7 shows a funding problem in March '95. Some development activities had to be delayed until the funding problem was resolved and additional funds were provided. Actuals were initially below plan, but are now tracking close to planned costs. Questions to ask about variances and overruns include: Are overruns due to activities costing more than planned? Is work beginning ahead of schedule? If actual cost is below plan, does that mean that the program is behind schedule or have activities cost less than planned? Investigating these other issues will help isolate the problem. |
| Lessons Learned | Evaluate major changes in the rate of actual cost expenditures. Since software development is a very labor intensive activity, this data should track closely to effort data. |

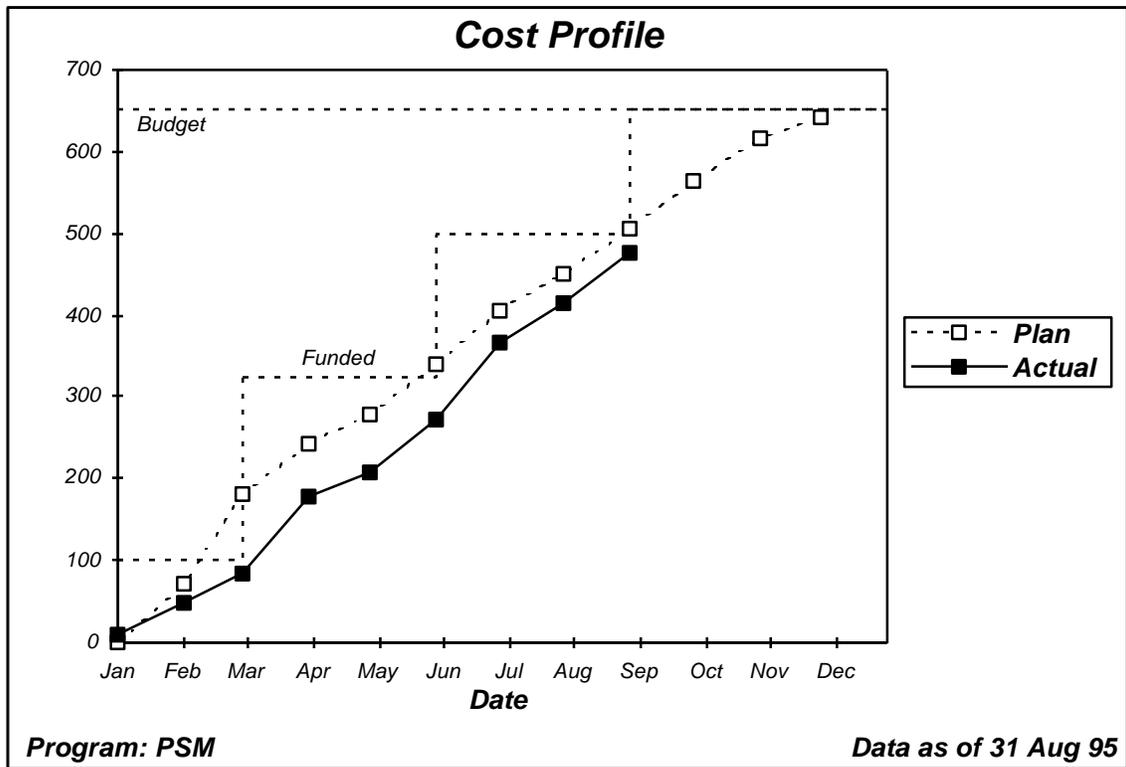


Figure 3-7. Cost Profile

3.8 RESOURCE UTILIZATION INDICATOR

| | |
|-----------------------------|---|
| Issue | Resources and Cost |
| Category | Environment Availability |
| Selected Measure | Resource Utilization |
| Description | Helps determine whether the test facilities needed to test the system are available and being used. |
| Example Graph | A line chart was produced containing four distinct utilization measures: 1) planned test facility availability (based on facility predictions), 2) actual test facility availability to date (based on total time minus actual maintenance downtime), 3) scheduled project utilization (based on project schedule), and 4) actual project utilization to date (based on project hours logged). |
| Feasibility Analysis | Check that the scheduled utilization of the test facility for this program is achievable given predicted test facility availability. Ensure that usage by other programs and scheduled downtime have been accounted for in availability figures. Ensure that predictions are consistent with recent past history. Evaluate the risks which may arise in scheduling the test facilities resources if testing is delayed. |
| Performance Analysis | Analysis of Figure 3-8 shows that testing at the facility started one month late. Also, a shortfall in the facility's availability in September appears to have impacted progress that month. Since the actual hours used to date are significantly below planned, a replan is probably needed. In addition, the cause of the shortfall in availability should be investigated to help reduce changes in future availability. Also, review testing progress to date in order to gain a more complete analysis of the situation. |
| Lessons Learned | Unexpected variances in either resource utilization or availability should be investigated. This may help prevent future problems. |

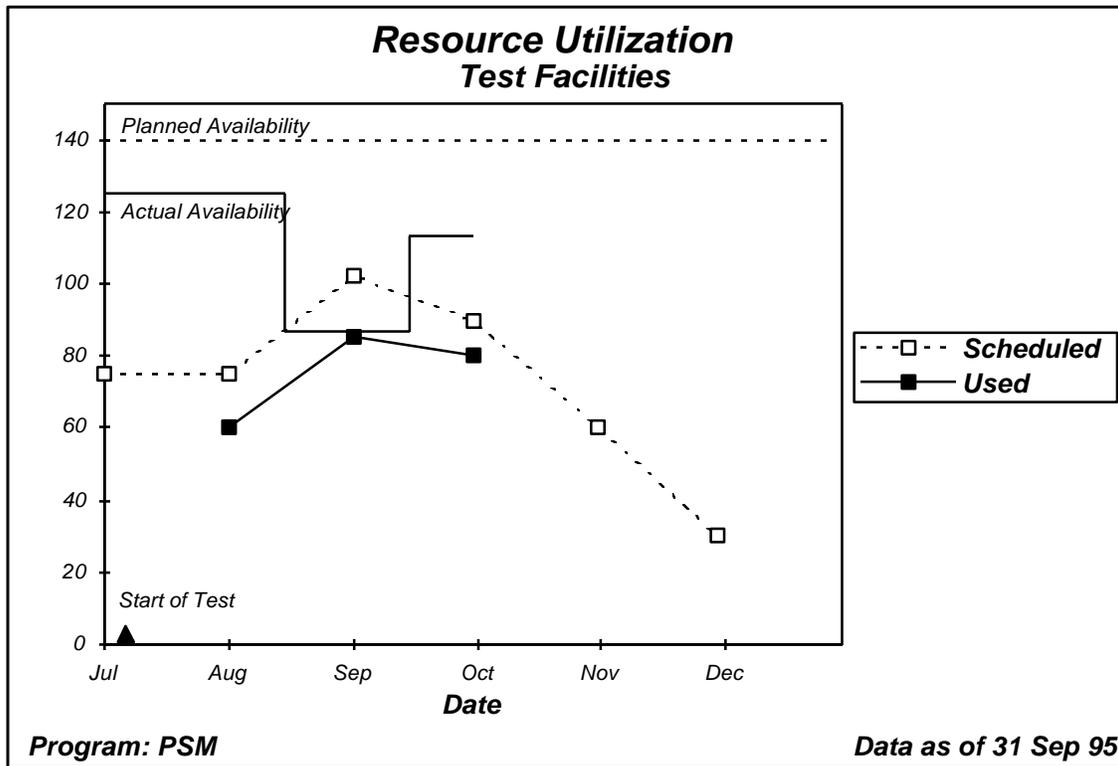


Figure 3-8. Resource Utilization

3.9 SOFTWARE SIZE INDICATOR

| | |
|-----------------------------|---|
| Issue | Growth and Stability |
| Category | Product Size and Stability |
| Selected Measure | Lines of Code |
| Description | Provides an estimate of software size, which is the major variable used to estimate software development effort and schedule. Used to monitor progress by comparing actual code developed and modified over time to plans for code development and growth. Unplanned additions and changes to code can adversely influence schedules and costs. |
| Example Graph | A line chart was used to show changes over time to: 1) overall software size estimates; and 2) actual size growth as the development proceeds. Size is measured in source lines of code. A corresponding bar chart shows the size breakdown by CSCI, and reflects the changes due to replans. |
| Feasibility Analysis | Compare total estimated lines of code and the estimated code growth with other similar projects. Correlate size estimates over time with staffing profiles for the development team. There should be sufficient staff assigned during each time period to complete coding assignments, after taking into account rework, concurrent assignments, non-project time, and programmer productivity. |
| Performance Analysis | Figure 3-9a shows progress in actual code development. Code production is approaching the current size estimate. The graph also shows growth in the size estimate. Figure 3-9b shows that most of the estimated size increase was attributable to CSCI C. |
| Lessons Learned | It is not unusual for there to be moderate increases in total software size over the original estimates. Increases of up to 20% are common. Larger increases in estimates or actuals should be investigated. |

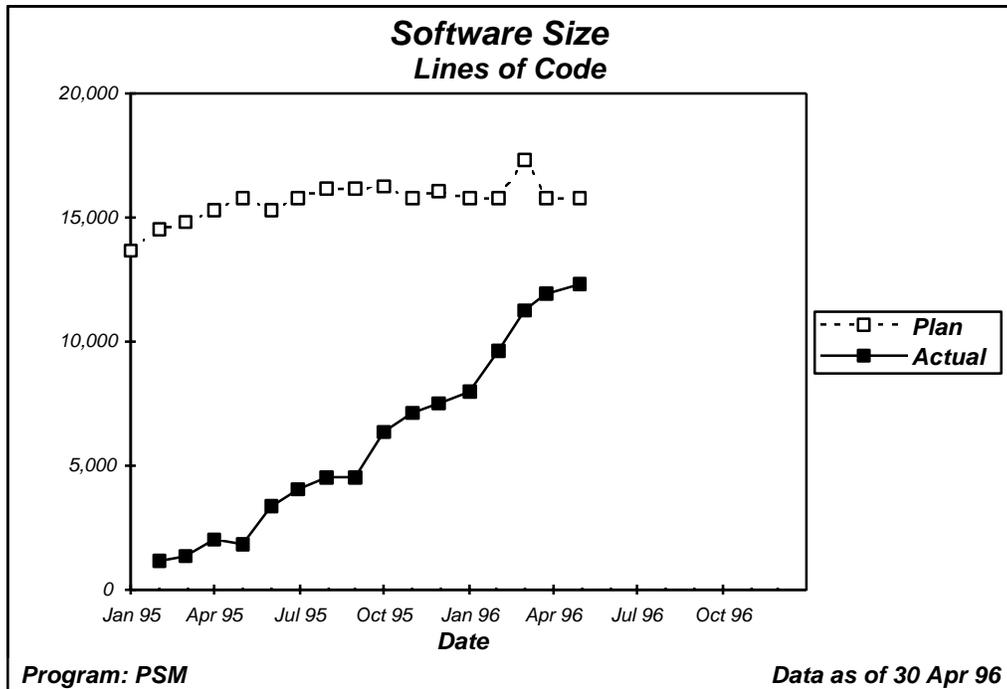


Figure 3-9a. Software Size (LOC)

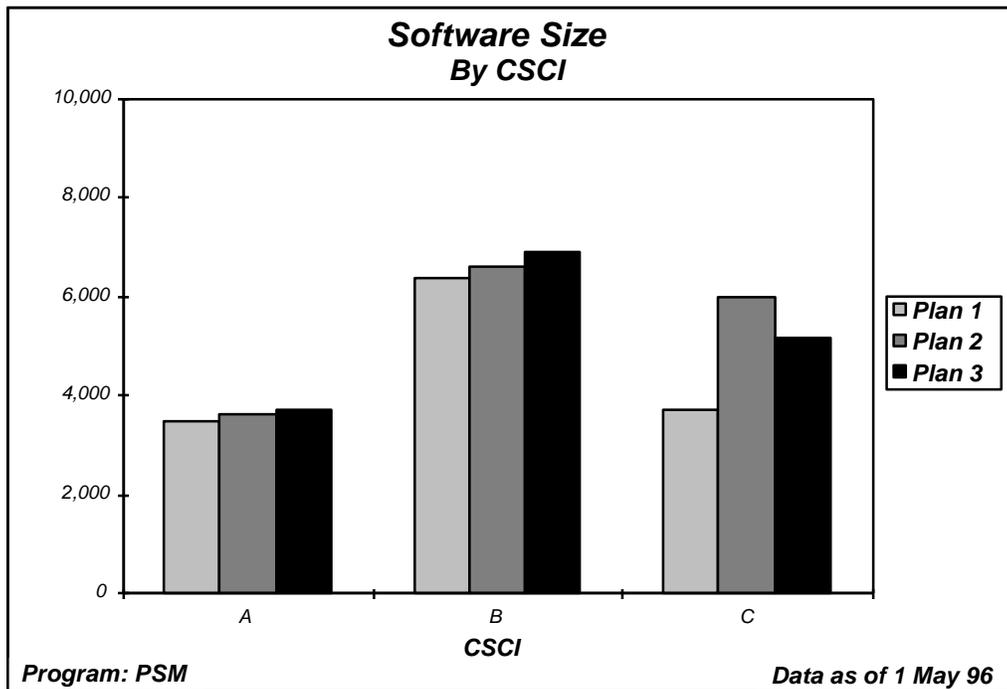


Figure 3-9b. Software Size (CSCI)

3.10 REQUIREMENTS STABILITY INDICATOR

| | |
|-----------------------------|---|
| Issue | Growth and Stability |
| Category | Functional Size and Stability |
| Selected Measure | Requirements |
| Description | Provides an early measure of software size. Used to monitor changes to requirements throughout a project, which can serve as a leading indicator of delays, rework, and cost increases. |
| Example Graph | A line chart (Figure 3-10a) was used to show two related pieces of information. The top line shows the trend in total number of actual requirements defined to date. Data points past the “as of” date reflect estimates. The bottom line is the total number of requirements either added, changed, or deleted during the reporting period. A bar chart (Figure 3-10b) was also produced to provide more detail about whether the changes made were requirements additions, modifications, or deletions. |
| Feasibility Analysis | If requirements growth has been estimated for the project, use other program knowledge to evaluate whether the amount of change expected is realistic. Consider things like the developer’s capability, the team’s understanding of the problem, and the number of customers involved. |
| Performance Analysis | Figure 3-10a shows an overall increase in requirements after the March SSR, which was expected, and another unexpected increase this month which can be traced to the PD. held in June. Figure 3-10b indicates that the changes were the result of additions and modifications to already defined requirements. The magnitude (approximately 20% of the total requirements were affected during this last period and total requirements increased by over 10%) and timing (the project is well into the design activity) of these requirements is a cause for concern. Resource allocations, effort estimates, budgets, and schedules may be in jeopardy and should be reevaluated. |
| Lessons Learned | Constantly changing requirements or a large number of additions after requirements reviews are leading indicators of schedule and budget problems later in the project. Requirements should be tracked at a lower level, such as by CSCI. |

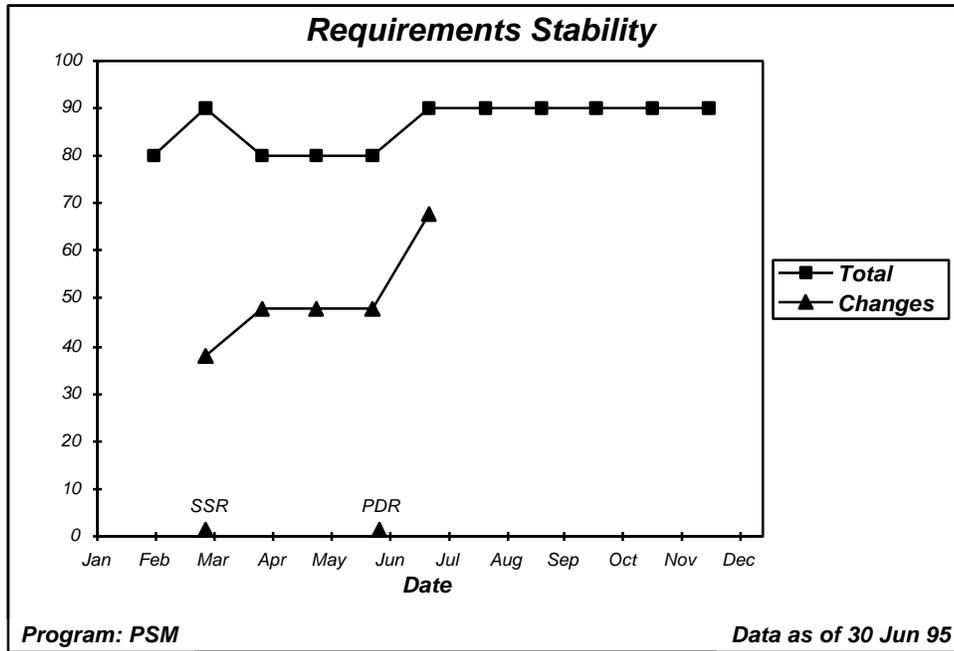


Figure 3-10a. Requirements Stability

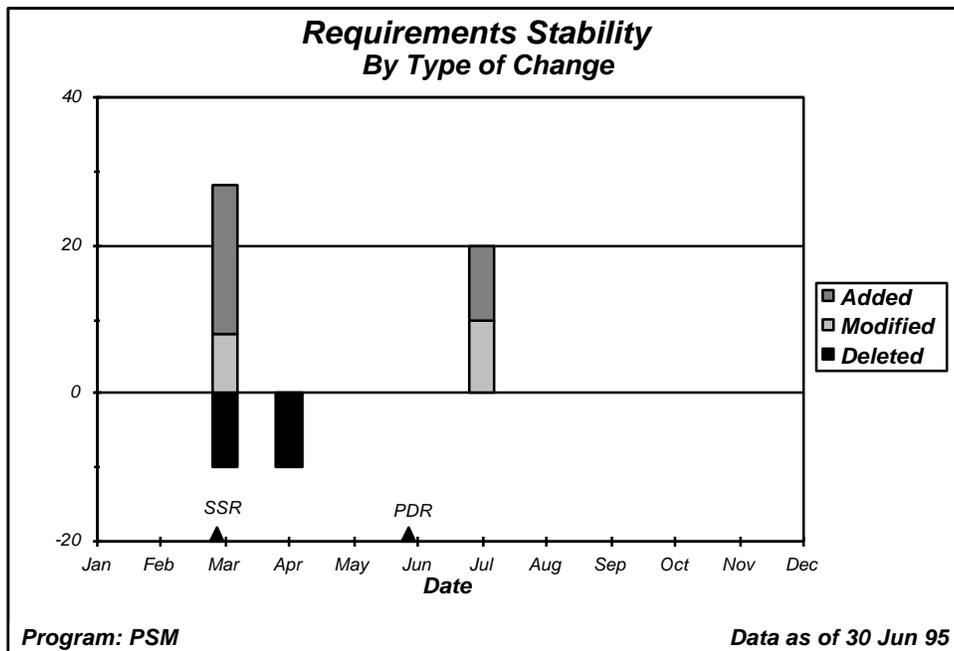


Figure 3-10b. Requirements Stability by Type Change

3.11 RESPONSE TIME INDICATOR

| | |
|-----------------------------|--|
| Issue | Growth and Stability |
| Category | Target Computer Resource Utilization |
| Selected Measure | Response Time |
| Description | Measures whether the system can perform standard on-line functions in a timely manner by comparing actual response times to the required response times. |
| Example Graph | A bar graph was used to compare the results of a series of response time tests against a contract-specified on-line response time requirement. A series of test runs were executed for selected sets of representative queries and update functions. For each function, response time measures were collected (using an automated Monitor tool). The collected data sets were then averaged. The sample graph shows a series of three test runs and indicates the acceptable average response time as a straight line. |
| Feasibility Analysis | Ensure that the response time requirement specified is feasible given published or observed statistics such as database and hardware benchmarks, performance models, or operational results from similar systems. |
| Performance Analysis | Figure 3-11 shows that query-type functions were initially exceeding response time requirements. These functions were subsequently modified to improve performance and are now within the acceptable range. Update functions initially performed well, but performance problems were noted in the second test. These were apparently resolved prior to the third test. When results are outside the acceptable range, a more detailed analysis by component or transaction can help pinpoint the problem code. |
| Lessons Learned | Define the criteria for choosing the functions whose response time will be measured (typical, importance/criticality, frequency of occurrence). Also determine what form of response time measures should be compared to the planned or target figure (an average, sample, worst case). Factors that may influence the validity of actual response time measures include: 1) not simulating sufficient load on the target machine during the tests, 2) not sampling representative functions, and 3) using a test database whose size is smaller than the operational version. |

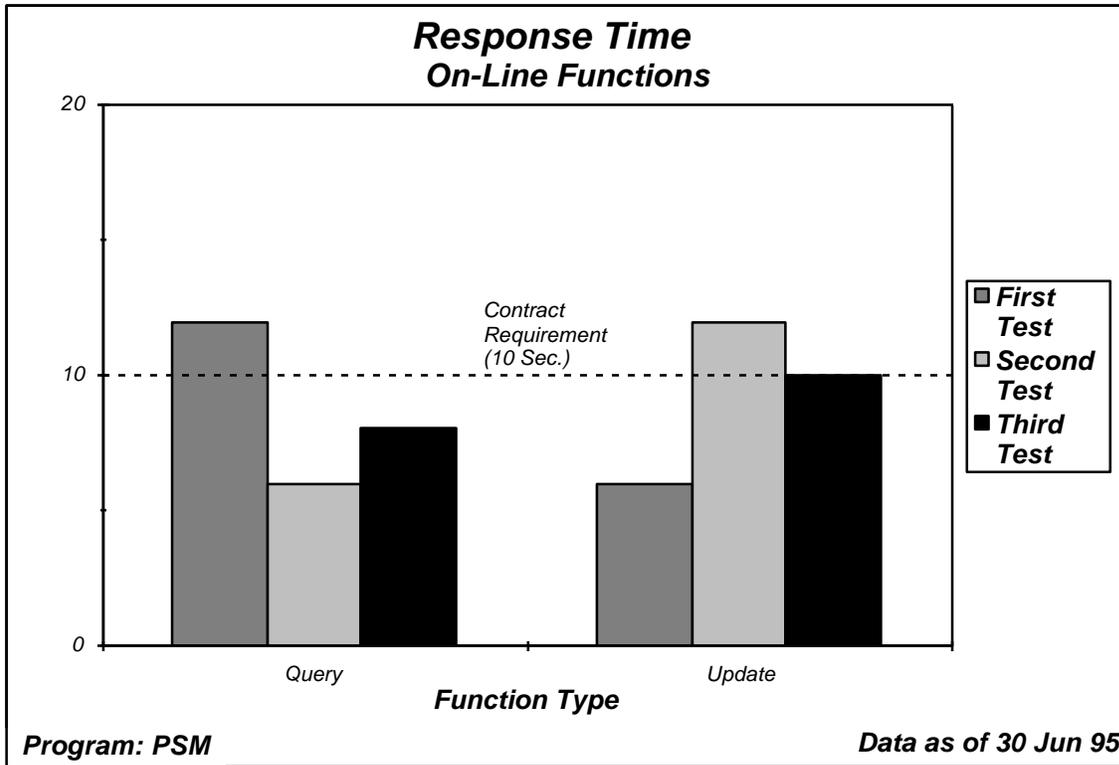


Figure 3-11. Response Time for On-Line Functions

3.12 PROBLEM REPORT STATUS INDICATOR

| | |
|-----------------------------|---|
| Issue | Product Quality |
| Category | Defect Profile |
| Selected Measure(s) | Problem Report Trends |
| Description | Problem Status provides information on the number of problem reports (PRs) found over time, and their status (open/closed). The quantity of PRs provides an indication of rework effort and overall product quality. Closure rates help assess progress by indicating the amount of work (rework) left to be done. |
| Example Graph | The top line of the line chart in Figure 3-12a shows the cumulative number of PRs detected to date. The bottom line shows the number of PRs that have been closed. Figure 3-12b shows the total number of PRs still open, by priority code. |
| Feasibility Analysis | Not applicable. |
| Performance Analysis | The top line in Figure 3-12a indicates that problems have been steadily discovered over the past year. However, in the past several months the discovery rate appears to have tapered off. If the reason for this is that testing is successfully completing and the project is nearing completion, this is a good sign. If the reason is that testing has prematurely slowed or halted, this may indicate a significant problem. The bottom line indicates that closure rate has kept pace with the discovery rate. Figure 3-12b shows that over half of the remaining open PRs are priority 1 and 2. These PRs should be reviewed to determine whether this is a cause for concern. |
| Lessons Learned | The closure rate should remain similar to the discovery rate. Large gaps between the two trend lines indicates that problem correction is being deferred, which could result in serious schedule, staffing, and budget problems later in the project. A flat PR discovery trend line during design, coding, or testing may indicate that reviews and tests are not being performed, and should be investigated. Monitor open PRs by priority to insure that high priority defects are being fixed quickly. |

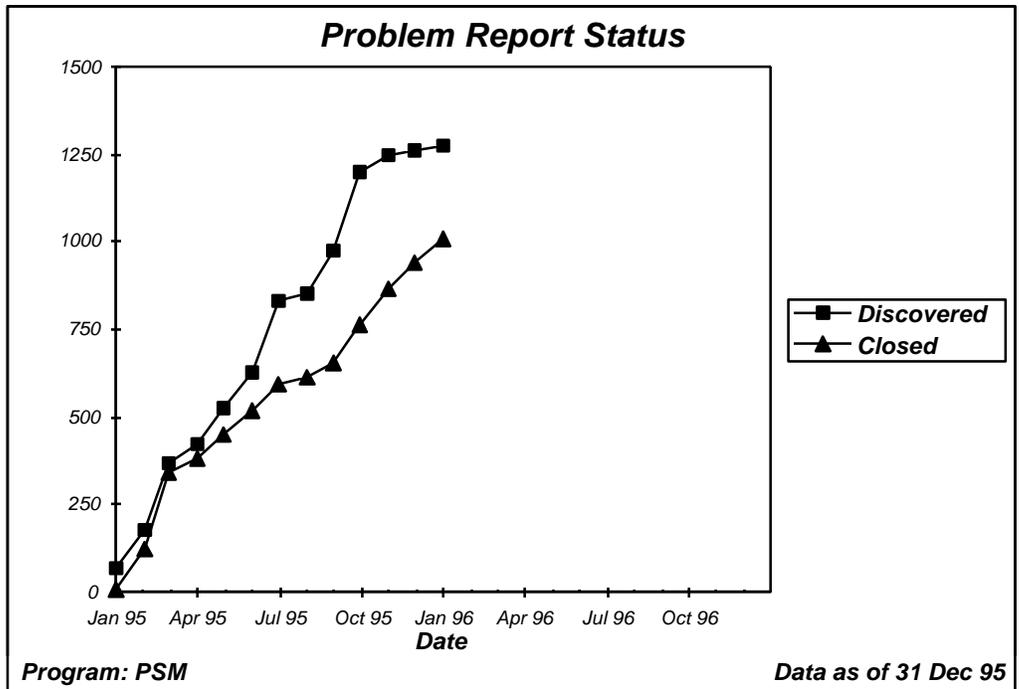


Figure 3-12a. Problem Report Status

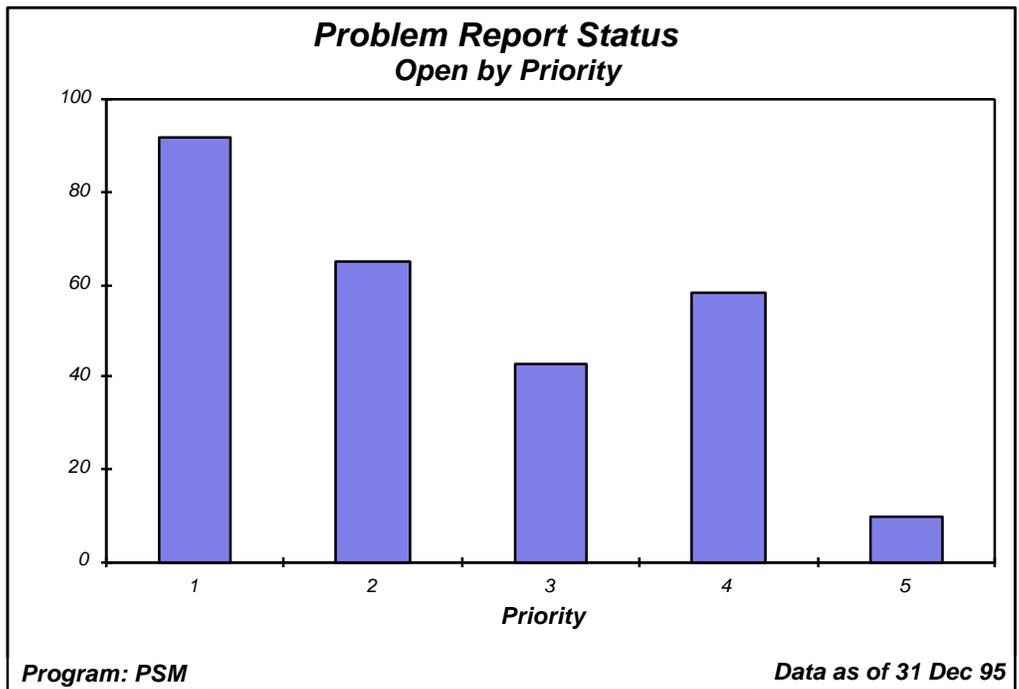


Figure 3-12b. Problem Report Status Open by Priority

3.13 PROBLEM REPORT AGING INDICATOR

| | |
|-----------------------------|--|
| Issue | Product Quality |
| Category | Defect Profile |
| Selected Measure(s) | Problem Report Aging |
| Description | Provides information on the number and age of open problem reports. The age distribution of problem reports help assess whether or not problems are being dealt with in a timely manner. |
| Example Graph | The bar chart includes all open PRs, divided into categories by age. This was done by first calculating, for each PR, the number of days that have elapsed since the PR was initially reported. PRs were then grouped by age categories and graphed. |
| Feasibility Analysis | Not applicable. |
| Performance Analysis | Figure 3-13 shows an average open age of 5.7 weeks for the 60 open problem reports. This is below the target of 8 weeks. Assessing whether the age of open PRs is a problem requires an understanding of the length of program, the program's current status, commitments to users, and the type and severity of the defects still open. |
| Lessons Learned | When measurement results indicate that problem correction is being deferred, it is likely that schedules, staffing levels, and budgets will be impacted later in the project. During testing, test progress is often significantly impacted by the deferment of problem correction. During maintenance, the age of problems reported by customers should be monitored to insure that customer problems are addressed in a timely manner. |

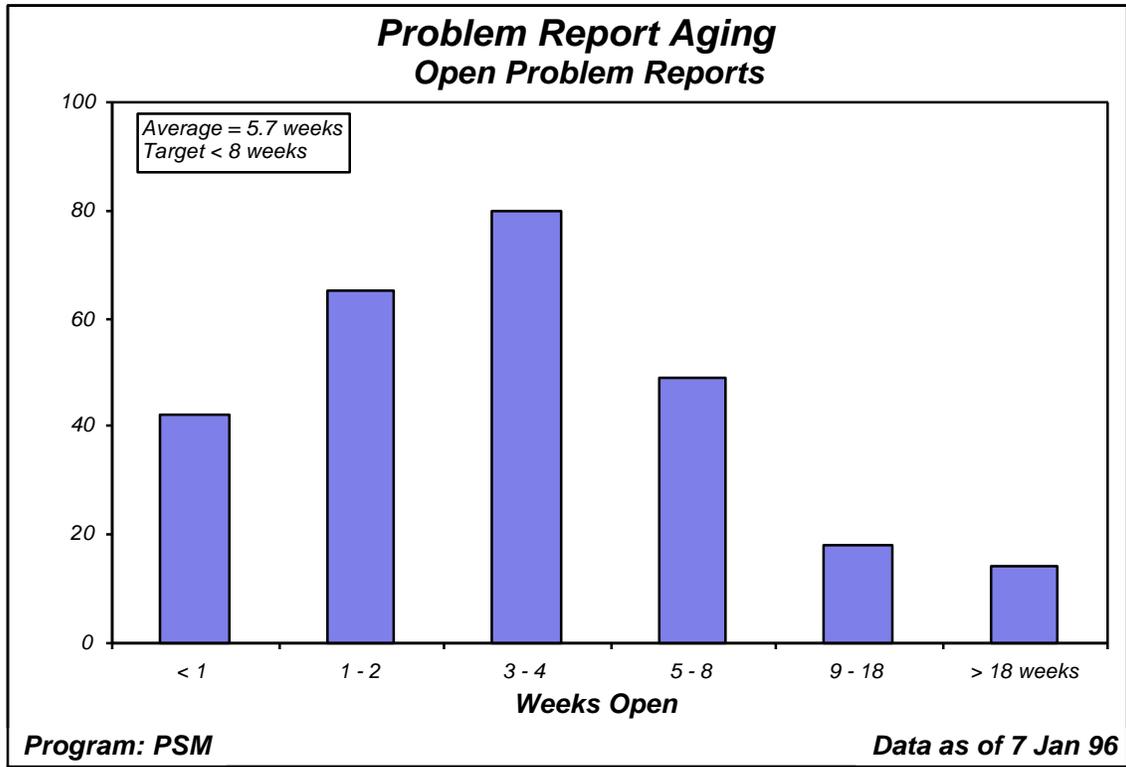


Figure 3-13 Problem Report Aging

3.14 DEFECT DENSITY INDICATOR

| | |
|-----------------------------|--|
| Issue | Product Quality |
| Category | Defect Profile |
| Selected Measure | Defect Density |
| Description | Used to assess product quality by normalizing the number of defects detected in a product by the product's size. Can be used to identify which components, subsystems, or CSCIs have the most quality-related problems. |
| Example Graph | A table was used to show CSCI level defect densities for the various development organizations that participated in a particular project. Defect densities were calculated by dividing the number of defects identified to date by CSCI size. |
| Feasibility Analysis | Not Applicable. |
| Performance Analysis | Figure 3-14 indicates that organization Z's defect densities are higher than average. This may mean that CSCI's F & G will need more attention, such as additional reviews or testing. Other project-related factors such as component complexity, defect distribution by classification, and organizational factors such as process maturity should also be reviewed to gain a better understanding of the reasons for these densities. |
| Lessons Learned | Defect densities can be generated at lower levels to identify specific components which should be subject to more quality control or should be targeted for redevelopment. The overall quality of a development project can often be evaluated by looking at the first 6-12 months of post-release defect densities. Large numbers of defects reported from the field may be the result of requirements not being met, inadequate testing, or poor code quality. |

| Defect Density | | | | |
|-----------------------|-------------|-------------------------|----------------|---|
| Org | CSCI | Size (KSLOC) | Defects | Defect Density (Defects/KSLOC) |
| X | A | 44 | 48 | 1.1 |
| X | B | 32 | 60 | 1.9 |
| Y | C | 36 | 36 | 1.0 |
| Y | D | 28 | 33 | 1.2 |
| Y | E | 34 | 42 | 1.2 |
| Z | F | 15 | 46 | 3.1 |
| Z | G | 9 | 30 | 3.3 |
| Total | | 198 | 295 | 1.5 |

Program: PSM **Data as of 30 Jun 95**

Figure 3-14. Defect Density

3.15 SOFTWARE COMPLEXITY INDICATOR

| | |
|-----------------------------|--|
| Issue | Product Quality |
| Category | Complexity |
| Selected Measure | Cyclomatic Complexity |
| Description | Measures the number of logic paths in a component. Can be used to assess the amount of testing required, predict component defect density, estimate future maintenance effort, identify the components that should be redesigned or reimplemented. Component complexity measures are typically compared to a standard or required threshold. |
| Example Graph | <p>A bar chart was used to identify the number of components whose complexity measures fall outside the threshold. Each component within CSCI A was measured using an automated Code Complexity Analysis Tool. Component complexity values were separated into six complexity range categories and then graphed. Then, the count of components in each range was divided by the total number of components, resulting in the percentage used to graph each bar on the chart. The threshold line divides the chart into acceptable and unacceptable ranges.</p> <p>A table was also produced by sorting the raw data by complexity, and showing only those components whose complexity was higher than the threshold.</p> |
| Feasibility Analysis | Evaluate whether the selected threshold can be met if one is set. The types of software being developed and the language used should be considered when evaluating the threshold selected. |
| Performance Analysis | Figure 3-15a indicates that about 80% of the components in CSCI A are less than or equal to the maximum threshold (10) for component complexity. The corresponding table (3.15b) identifies the specific components that exceed complexity limits. Further analysis of these components may identify one or more causes which are contributing to high complexity. A decision should be made about whether these components should be modified or rewritten. |
| Lessons Learned | This measure is not generally available until after a component has been coded. An automated code analysis tool is needed to accurately and efficiently produce the measure. |

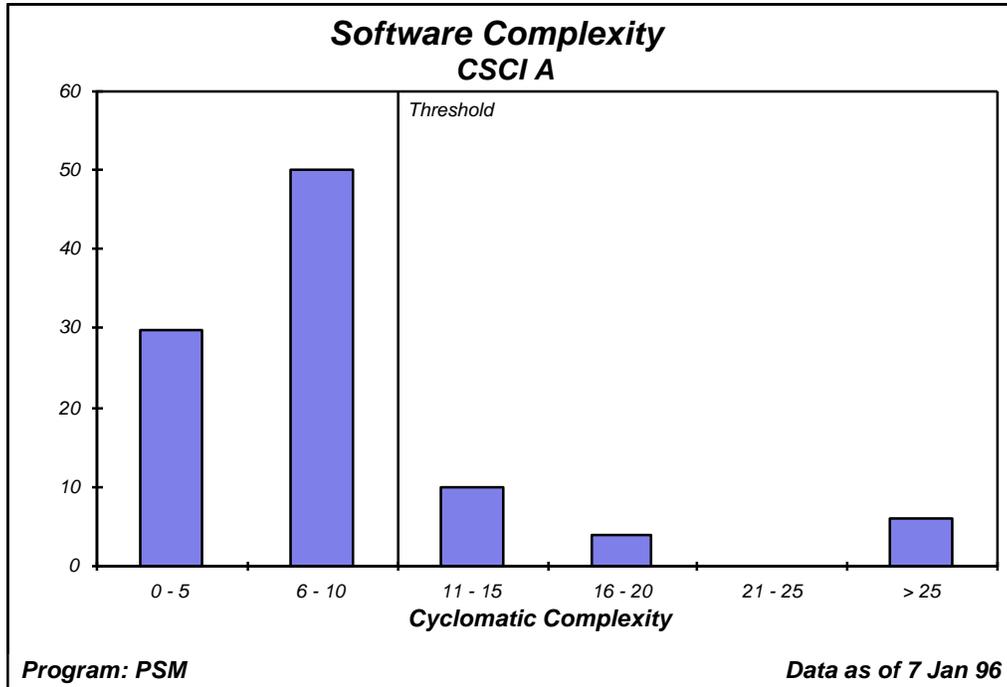


Figure 3-15a. Software Complexity CSCI A

Software Complexity CSCI A
Units with Complexity > 10

| Unit | Cyclomatic Complexity |
|------|-----------------------|
| A1 | 53 |
| A2 | 49 |
| A3 | 32 |
| A4 | 27 |
| A5 | 25 |
| A6 | 25 |
| A7 | 20 |

Program: PSM Data as of 7 Jan 96

Figure 3-15b. Software Complexity CSCI A
Units That Exceed Complexity Limits

3.16 SOFTWARE PROCESS MATURITY INDICATOR

| | |
|-----------------------------|--|
| Issue | Development Performance |
| Category | Process Maturity |
| Selected Measure | CMM (Capability Maturity Model) Level |
| Description | Used to gain an understanding of an organization's relative software development capability. The CMM Level measure results from a formal software capability evaluation (SCE) of an organization's software engineering and project management processes. Often used to set standards for selecting a software development contractor and to select among competing development organizations. |
| Example Graph | A table was used to display process maturity scores for three organizations. The score was produced using the formal SEI Capability Maturity Model-based SCE assessment procedures. Highlights (strengths and weaknesses) from the assessment findings were also noted in the table. |
| Feasibility Analysis | Not Applicable. |
| Performance Analysis | Figure 3-16 reflects the results of SCE assessments for three organizations. The rating scale for SCE assessments ranges from 1 to 5, where 5 indicates an organizations with a high level of software development capability (i.e., a very mature software engineering process); Company B has received a higher rating than companies A and C. However, all three organization have a score which either meets or exceeds the target level set as part of the contract requirements. When this analysis is being performed as part of the contractor selection process, detailed findings from the assessment should be reviewed with special attention given to the critical processes a contractor must possess for this contract. |
| Lessons Learned | The process maturity score is only as good as the assessment process that produced it. Also, consider how long ago the SCE was performed and recognize that a maturity score is given at an organization-level, based on a sampling of projects. |

| Software Process Maturity SEI Capability Maturity Model | | | |
|--|--------------|--|--|
| Organization | Level | Strengths | Weaknesses |
| <i>Target</i> | 3 | | |
| <i>Proposal 1 Company A (Prime)</i> | 3 | <i>Effective SEPG and task team structure, with many improvements implemented. Mature testing process</i> | <i>No defined measurement process/framework; measures not integrated into project management. Reviews are informal. Test automation is new and unproven.</i> |
| <i>Proposal 1 Company B (Subcontractor)</i> | 4 | <i>Measurement used in-process to make decisions. Historical measurements and lessons learned database used for project planning. Good subcontract management process.</i> | <i>Defect prevention/causal analysis just getting started. Few advanced tools used.</i> |
| <i>Proposal 2 Company C (Prime Only)</i> | 3 | <i>Good CM, testing, inspections with automation support</i> | <i>Planned measurement data not established for progress-related issues; measurements not used to make project decisions.</i> |

Program: PSM **Data as of 7 Jan 96**

Figure 3-16 Software Process Maturity
SEI Capability Maturity Model

3.17 SOFTWARE PRODUCTIVITY INDICATOR

| | |
|-----------------------------|---|
| Issue | Development Performance |
| Category | Productivity |
| Selected Measure | Product Size/Effort Ratio |
| Description | Indicates the amount of work produced relative to the effort expended. If an actual rate can be established early in a project or one can be predicted based on historical data, it can be used to estimate the remaining effort needed to complete the program. |
| Example Graph | A bar chart (Figure 3-17) was used to compare a project's planned productivity rates with an actual rate to date, and to proposed alternative replan rates. Each bar was produced by dividing work effort (reported in staff months) into the product size measure-Source Lines Of Code (SLOC). In this example, actual productivity is around 100 SLOC/staff month. |
| Feasibility Analysis | Compare planned productivity rates to past projects with similar characteristics (e.g., tools and methods used, staff skills, language used, etc.). Also, cross-check program data by calculating required productivity based on the present plan data, then compare those results to the planned productivity rates. Consider issues like learning curve, requirements volatility, expected turnover, and soft issues like staff morale and teamwork when evaluating the feasibility of a chosen rate. |
| Performance Analysis | Figure 3-17 shows that two productivity rates were used as the basis for developing project plans, about 170 for Build 1 and 110 for Build 2. However, with Build 1 well under way, actual productivity is only 100, significantly lower than planned. So, either productivity must be increased or substantially more effort will be needed to develop the complete product. Further analysis to determine the cause of lower-than-expected productivity should be performed before deciding on a course of corrective action. The third region of the bar chart shows two action plans. Plan 1 proposes increasing productivity by the end of Build 1 (slightly) and substantially increasing it for Build 2. Plan 2 assumes the rate throughout the remainder of the project will be similar to what has already been achieved, and adds a new Build 3 to complete production at this rate. Unless major changes could be immediately introduced, which is highly unlikely, option 2 appears to be a more realistic alternative. |

**Lessons
Learned**

If there is a significant change in productivity rates during a project, attempt to discover the underlying reasons for the change. Unplanned rework is a frequent cause of low productivity.

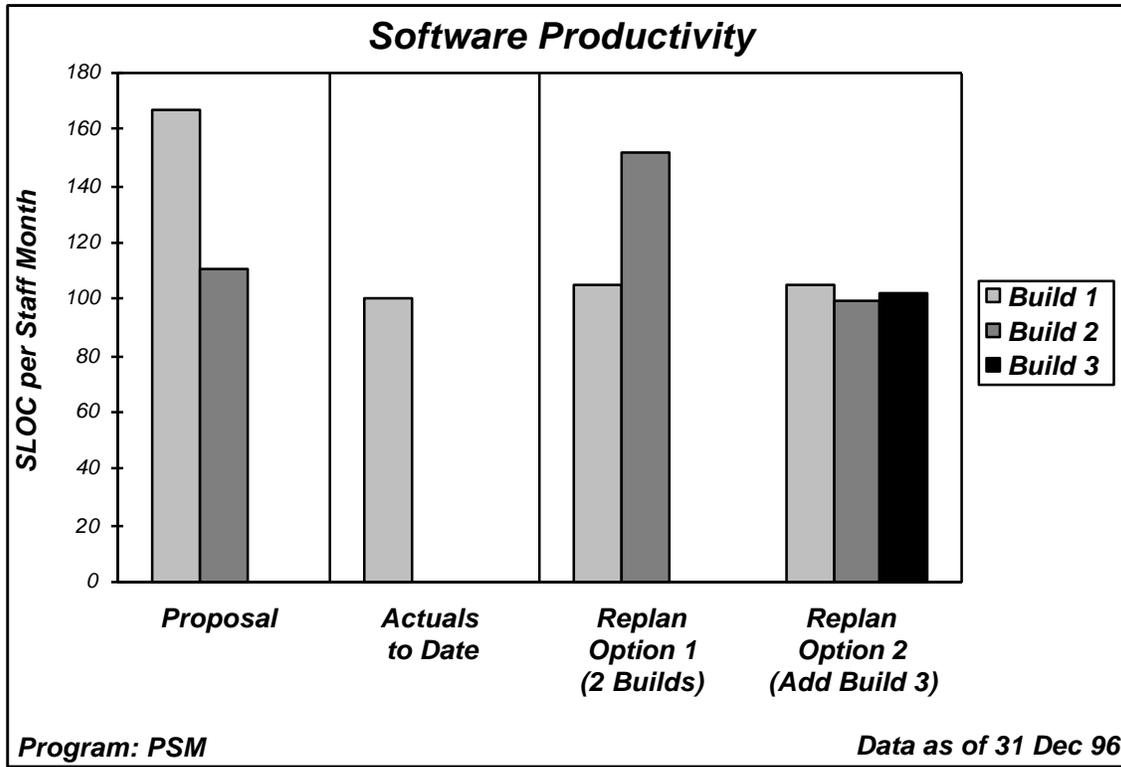


Figure 3-17 Software Productivity

3.18 REWORK EFFORT INDICATOR

| | |
|-----------------------------|---|
| Issue | Development Performance |
| Category | Rework |
| Selected Measure | Rework Effort |
| Description | Assesses the amount of effort expended to fix defects. Can be used to compare the amount of effort attributable to rework against the budget for rework. |
| Example Graph | Two bar charts were produced. The first chart (Figure 3-18a) reports rework as a separate major category of work effort and compares rework planned to the amount of rework actually performed to date. The second chart (Figure 3-19b) was produced by an organization whose time reporting system supports the collection of rework at the development activity level (i.e., requirements, design, etc.). For each chart, the accumulated number of planned and actual hours is used to produce the bars. |
| Feasibility Analysis | All projects will experience rework and it should be planned for. Analyze rework planned as a percentage of overall effort and look at the distribution of planned rework across project phases. Compare these percentages and distributions to the actual rework figures from recent, similar past projects. Achieving lower amounts of rework typically requires early defect control techniques such as reviews and inspections, and higher levels of process capability (see 3.16). |
| Performance Analysis | Figure 3-18a, reported during the integration and testing activity of the project, shows that planned rework has already been exceeded by over 100%. This chart cannot help identify the activities where the rework occurred, however. Figure 3-18b can be used when a more sophisticated rework reporting system is in place. In this chart, rework has been tracked at the software activity level and only the rework figures are graphed. This example shows that rework during both requirements and design was much greater than expected, but that rework during implementation was close to planned. |

**Lessons
Learned**

Rework occurs during all phases of a project. Taking the extra time up front to do things right the first time can reduce overall rework on a project. Few organizations do a good job of tracking rework. Most time accounting systems do not include separate rework tasks. In lieu of time reporting, rework can sometimes be tracked using review/inspection and problem report data.

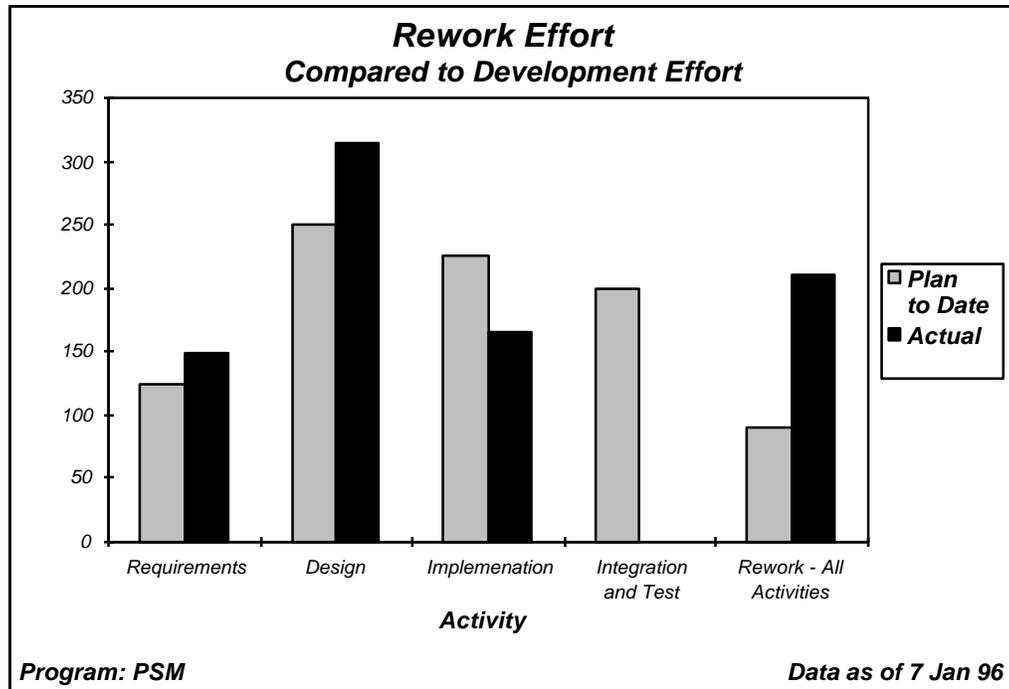


Figure 3-18a. Rework Activity Compared to Development Effort

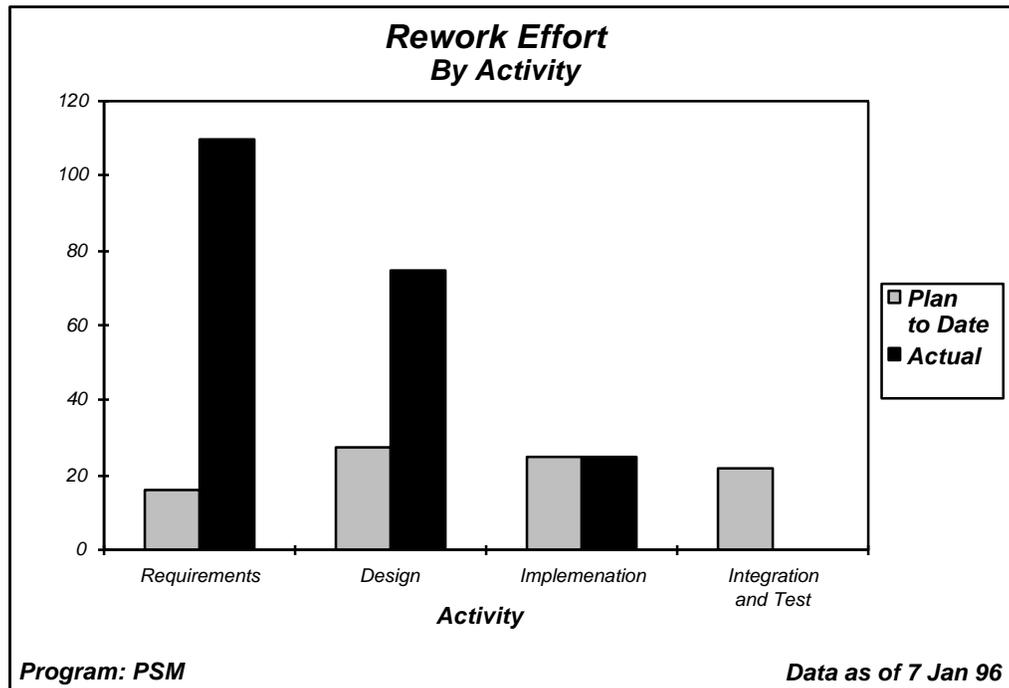


Figure 3-18b. Rework Activity By Activity

3.19 SOFTWARE ORIGIN INDICATOR

| | |
|-----------------------------|--|
| Issue | Technical Adequacy |
| Category | Technology Impacts |
| Selected Measure | Lines of Code |
| Description | Shows the amount of code by source (new, modified, retained, deleted, GOTS, COTS), which can serve as an indicator of the amount of work to be performed on a project. |
| Example Graph | A stacked bar chart was used to show the amount and distribution of developed and non-developed code. The non-developed portion of the bar is an estimate of the amount of code that would have to be developed if the COTS/GOTS software was not used; it is not an actual estimate of the COTS software itself (since this is not usually available and only a small portion of the COTS tool may be used). |
| Feasibility Analysis | The distribution of developed to non-developed code should be reviewed to assess whether expectations for the amount of code that will not be developed is realistic. The amount of new code needed to integrate COTS and non-COTS software should also be considered. |
| Performance Analysis | Figure 3-19 shows three planned and one actual size measure. Plan 1 shows an almost 50-50 split between non-developed and new code for the project. In plan 2, this ratio is revised; more new code development is planned. Plan 3 shows an additional increase in new code, resulting in an overall size increase. The actual size measures are close to plan 3 estimates, with only approximately 20% of the final product the result of non-developed code. This change most likely resulted in schedule delays and effort increases. |
| Lessons Learned | Changes in assumptions concerning the use of COTS/GOTS software or the amount of code that can be reused, can significantly impact project schedules and budgets. Plans should be re-evaluated when this occurs. |

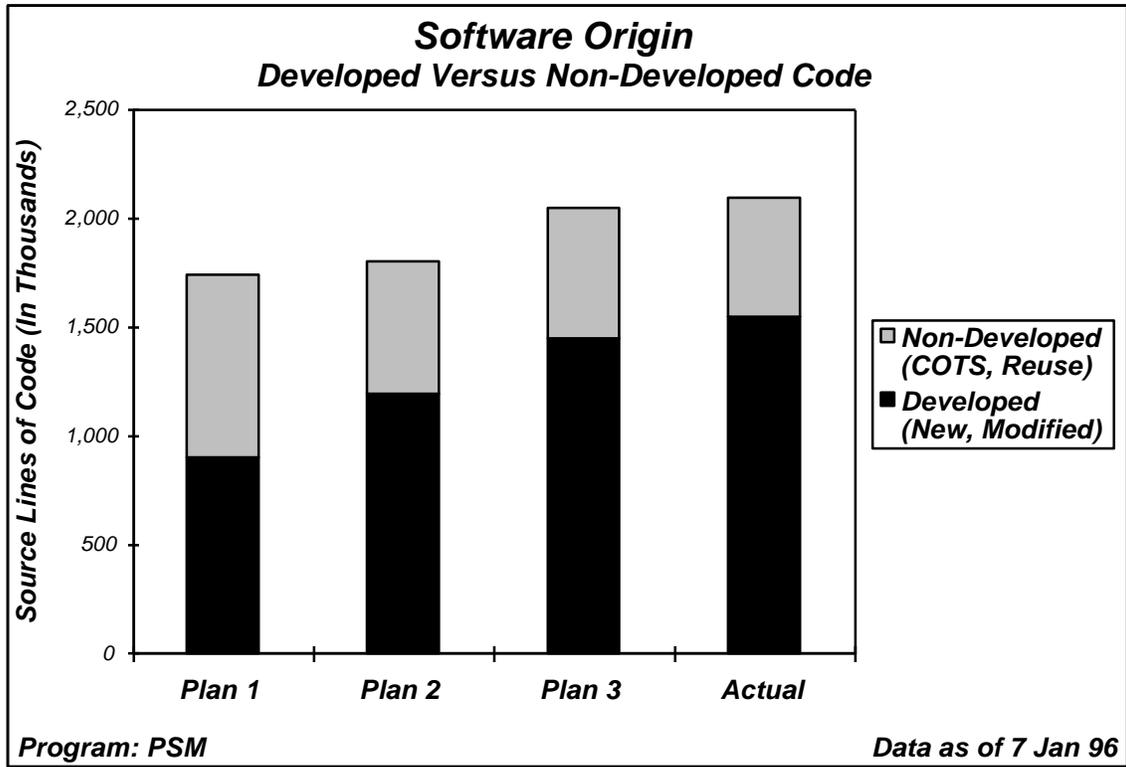


Figure 3-19 Software Origin Developed Versus Non-Developed Code

CHAPTER 4 – INTEGRATED INDICATOR EXAMPLES

This chapter provides examples of how sets of indicators can be used together to address the issues typically of concern during particular phases of the software life cycle. An integrated analysis approach which examines related indicators together has been found to be very effective for gaining insight into an issue. *These are examples only and do not represent a definitive set that should be applied to all programs.*

The first three examples in this chapter assume that the project being monitored is in the middle of a major development activity. Each example starts with the basic analysis of a progress-related issue and proceeds through a series of supplemental analyses, in an attempt to better understand project status or to uncover the underlying cause of a problem. The fourth example in this chapter shows how an organization might use a set of indicators to analyze maintenance issues during the post-development support phase.

The following examples are included:

| Identifier | Analysis Focus | Indicators |
|------------|------------------------|---|
| 4.1 | Design Completion | Design Progress Staff Level |
| 4.2 | Test Completion | Implementation Progress Test Progress Problem Report Status Staff Level |
| 4.3 | Readiness for Delivery | Test Progress Problem Report Status Software Reliability CPU Utilization |
| 4.4 | Maintenance | Requirements Stability Changes Implemented Software Reliability Milestone Progress |

4.1 DESIGN COMPLETION ANALYSIS

Description

As a project completes the system design phase and the focus shifts to implementation activities such as coding and unit testing, the staffing also shifts from primarily analysts to programmers. Therefore, it is important not only to monitor progress during this phase, but to also anticipate how changes in the schedule will impact staffing.

Basic Analysis

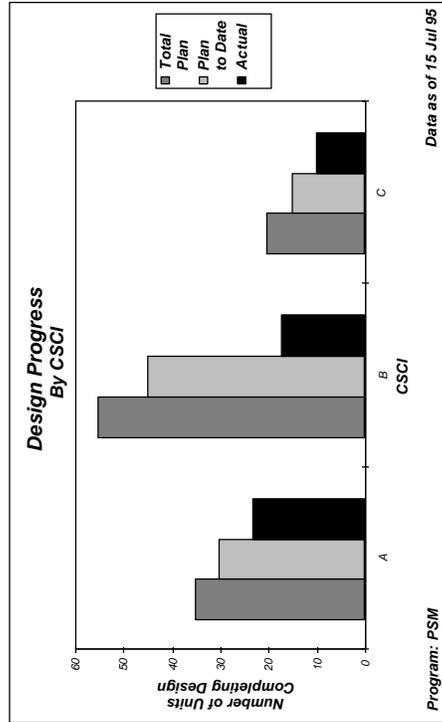
In Figure 4-1, the primary design progress indicator used is the work unit progress measures for units designed (a). This compares actual units completing design each week to the planned rate of completion. This indicator reveals that actual progress is significantly under plan as of July. The plan data line also indicates that all units should have completed design by the next month.

Supplemental Analyses

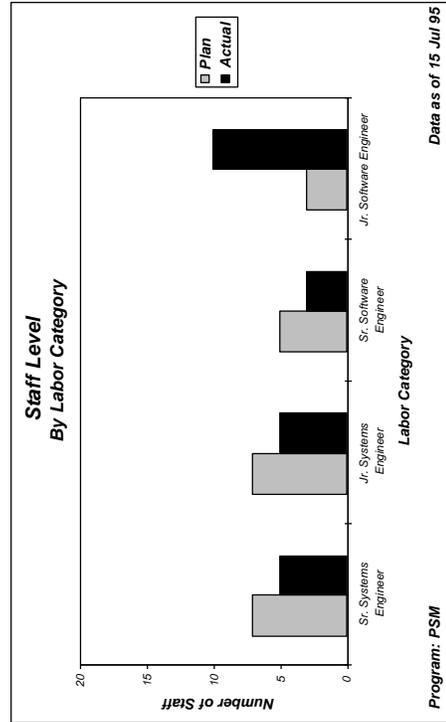
A closer look at the CSCI level indicator (b) reveals that, while all CSCIs are behind schedule, CSCI B is significantly below its completion plan.

The overall staff level (c) indicates that the project is currently staffed with approximately the right number of people, according to the monthly staffing plan. However, a drop in staffing occurred in May. It was during that timeframe that some staff turnover was experienced. Is the project behind schedule due primarily to the May dip in staffing? An analysis of the current month's actual staff level by labor category (d) shows that, while the original staff plan for design included mainly systems engineers and senior software engineers, the new design team composition is quite different than planned. The May changes in staffing resulted in the loss of several senior designers. Instead of bringing on new analysts to complete the design, the programmers assigned to join the project in July were brought onto the project early and assigned to design tasks. This had a negative impact. The programmers didn't have the experience to perform these tasks and the remaining designers were delayed bringing the new team members up to speed.

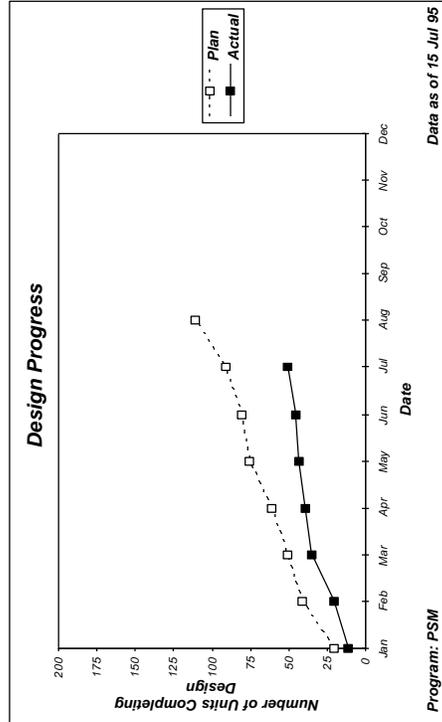
A revised plan for the remaining project activities is recommended.



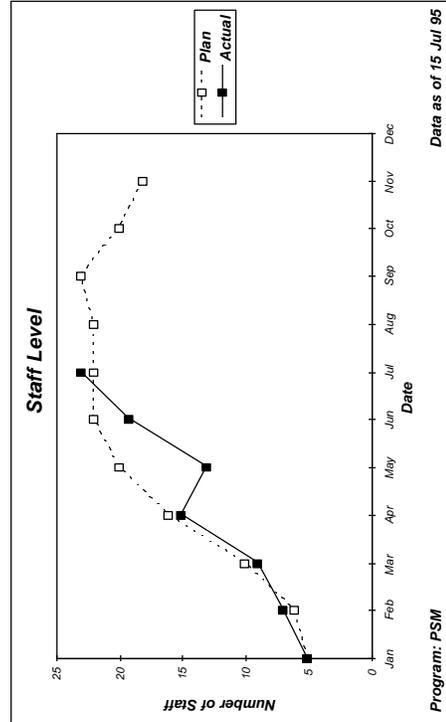
b)



d)



a)



c)

Figure 4-1. Design Progress

4.2 TEST COMPLETION ANALYSIS

Description

Two issues that impact test completion and often result in test schedule problems are: 1) not receiving components on schedule to test, and 2) waiting for fixed components to return to test after defects have been identified. This example shows how four indicators can be used to monitor test progress during the integration and test phase of a software development program.

Basic Analysis

The implementation progress line graph (a) indicates that implementation of components (and consequently, delivery of components to the testing group was late) and that, while all components have been delivered to date, they were finally delivered weeks behind the original schedule. Test progress is then shown (b). Three progress measures are compared: 1) the original plans for test component completion; 2) components upon which tests have been attempted; and 3) components that have passed testing. Not surprisingly, testing which was scheduled to start during week 4, did not **attempt** to test as many components as planned, and they also did not **pass** as many components as planned. While components attempted have remained fairly close to planned, components actually passed are well below plan.

Supplemental Analyses

An assessment of problem report status (c) indicates that testing has discovered a large number of problems to date. The closure rate, however, is not keeping pace with the discovery rate. Additionally, some high priority PRs are still open, which may also be impacting test progress. This may explain why the “components passed” trend line (b) has recently leveled off. It may be that a large number of components are actually being tested, but have not been “passed” due to problem reports found. Or, it may be that components originally delivered to test have been returned to development awaiting defect removal, meaning that testing cannot be completed for those components.

Test staffing (d) was scheduled to taper off, but the delays have prevented this. Based on developer input regarding new plans for fixing the outstanding PRs, test schedules and staffing plans must be revised.

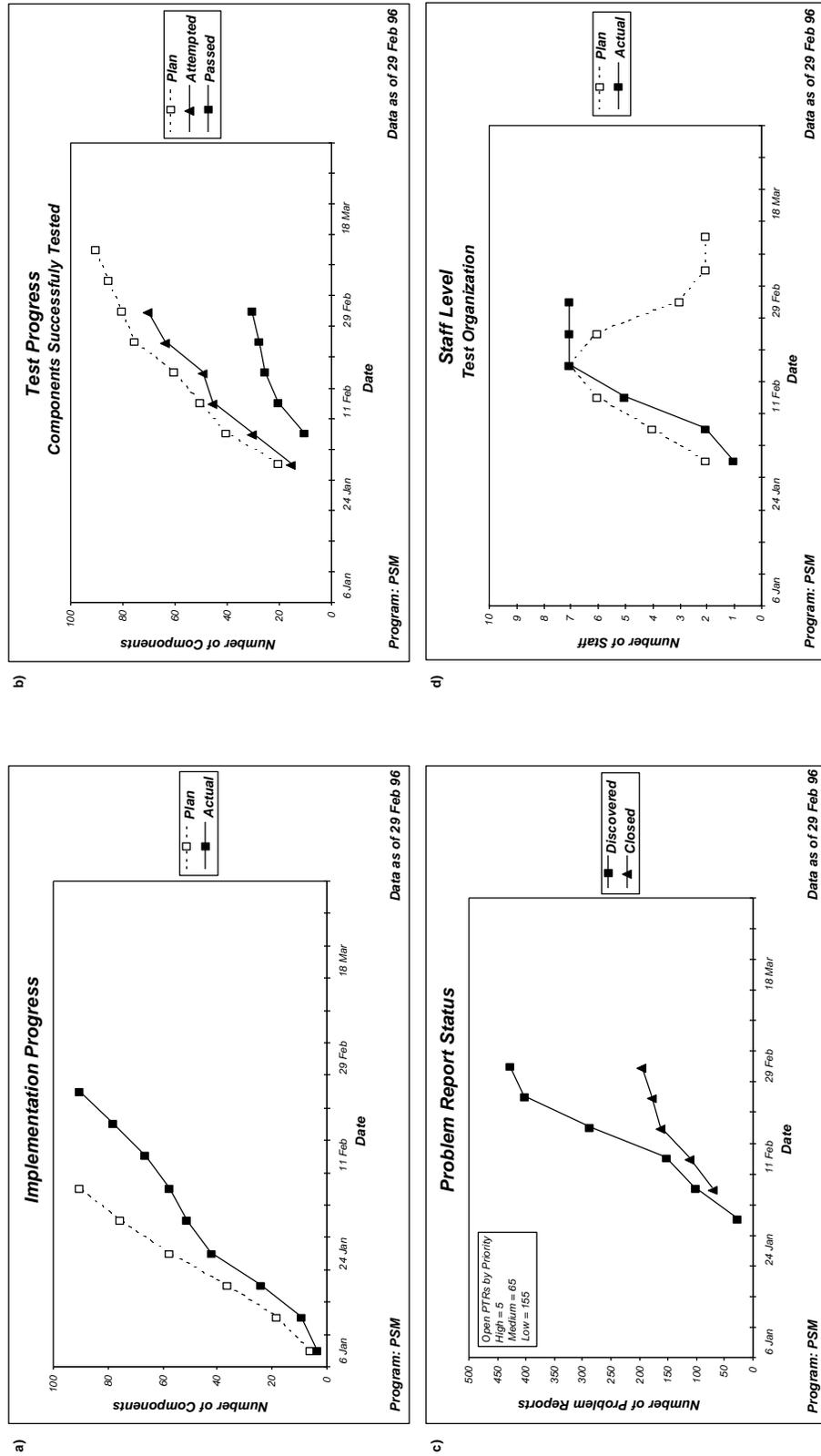


Figure 4-2. Test Completion

4.3 READINESS FOR DELIVERY ANALYSIS

Description

As a system approaches its delivery date, a number of issues may influence the decision to release the product. In addition to assuring that all testing has been completed, it is often necessary to demonstrate that certain contract requirements have been met. For example, there may be identified constraints that must be accommodated or specified thresholds that must be met. Figure 4-3 contains a set of diverse indicators which represent the specific concerns for this sample program prior to release.

Basic Analysis

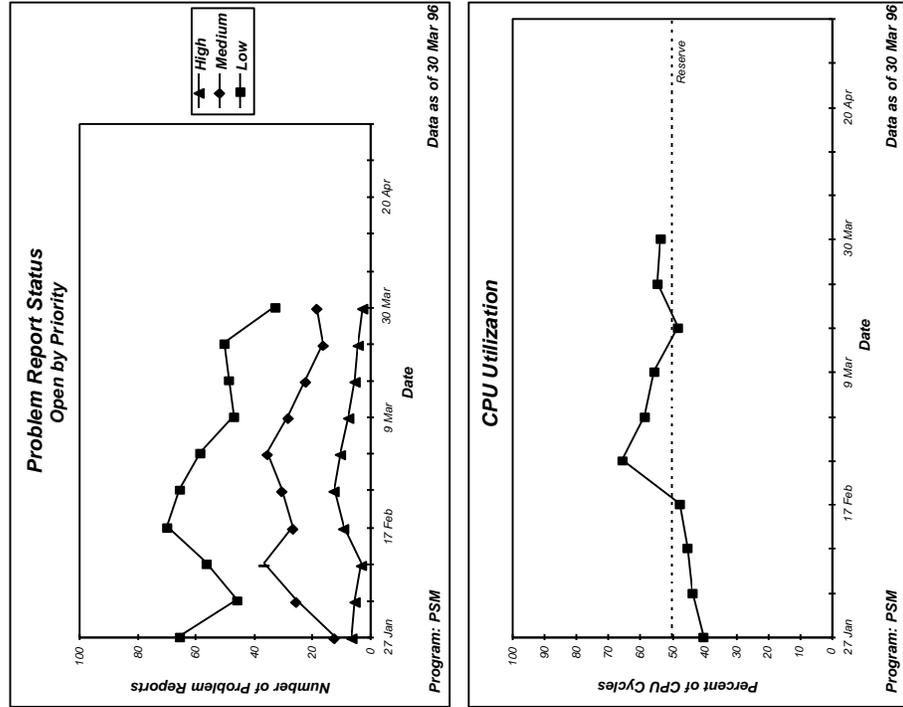
The test progress graph (a) reveals that requirements testing is proceeding close to plan, with almost 80% of requirements tested to date. With a release date scheduled at the beginning of week 15, it appears that testing can be completed as scheduled.

Supplemental Analyses

A look at the number and severity of open problem reports (b) indicates that, while a large number of PRs remain open, only 6 are high priority (priority 1 or 2). These will have to be fixed before the system can be released. The remaining PRs should probably be reviewed to ensure that deferment of those problems will not adversely affect usability or customer satisfaction.

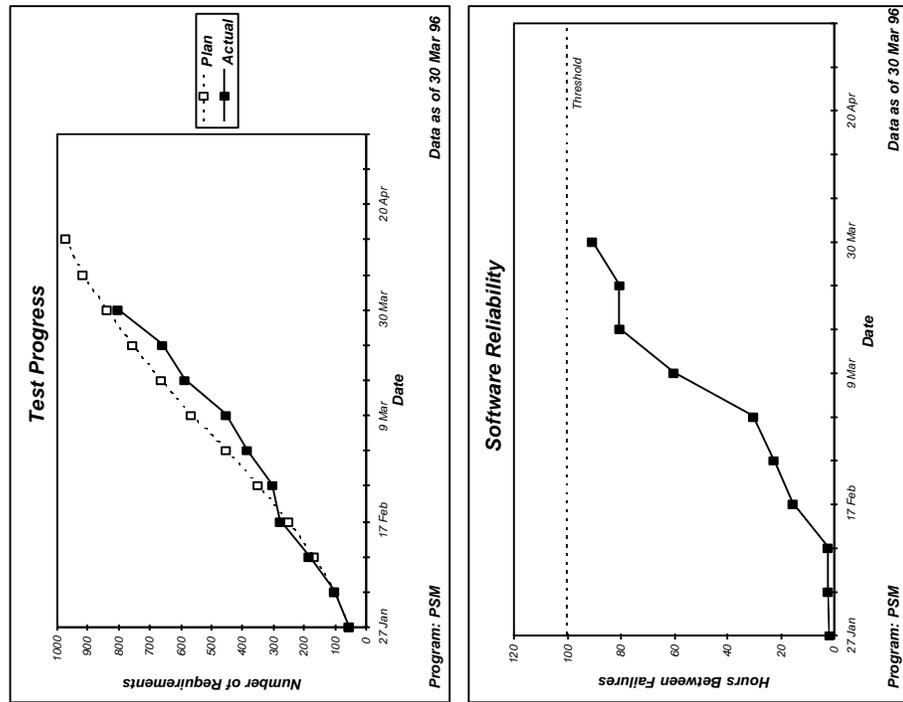
Software reliability (c) is calculated by logging the total number of usage hours that elapse between failures during acceptance test, is approaching the acceptable minimum of 100 hours between failures. The trend line continues to rise. The failure interval doesn't appear to be a cause for concern at this point.

The final issue being monitored is CPU utilization. Contract requirements call for 50% reserve capacity. Tests show that current utilization levels are above the 50% threshold, but only slightly. Reducing this rate would require additional changes to some programs that have otherwise been certified as working properly. This rework decision could delay delivery. The program manager may decide to make a trade off by accepting a system that exceeds the desired threshold, in order to allow the system to be delivered on time.



b)

d)



a)

c)

Figure 4-3. Readiness for Delivery

4.4 MAINTENANCE ANALYSIS

Description

Systems maintenance issues are often very different than issues related to new software development. Figure 4-4 provides a sample set of common indicators which might be monitored on a regular basis for a system which has recently entered into the software support phase. The sample system is currently on a three month release cycle. The system has undergone three releases so far this year. Work on a fourth release is currently in progress.

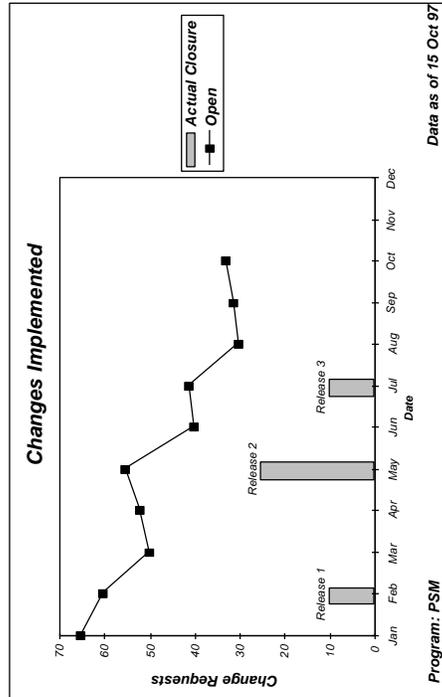
Basic Analysis

Requirements stability (a) provides an indication of how the planned content of each release was affected by changing requirements prior to installation. (Maintenance requirements are approved change requests.) Unplanned changes in release content can cause delays because work effort is often expended making changes to one set of requirements (i.e., approved change requests), then those requirements are set aside in order to work on higher priority requests in the release. This is what happened during the May release. It shows a large number of changed requirements were associated with Release 2.

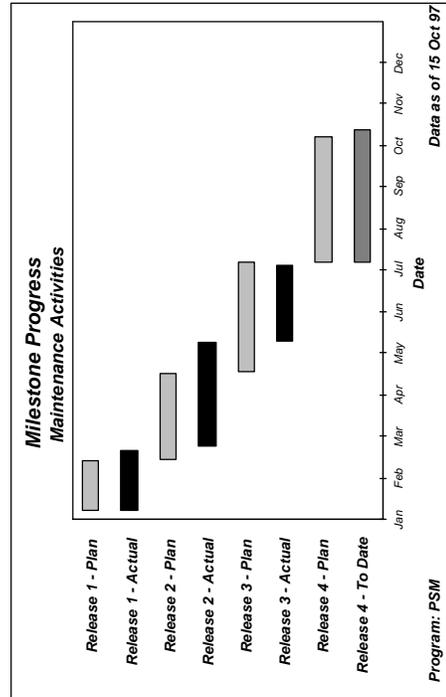
The size of the “backlog” of approved, pending change requests can be seen by looking at the descending line at the top of the changes implemented graph (b). Only a few additional change requests have been introduced in the last 10 months, and the backlog of changes is being gradually reduced. The chart indicates that, after each release has been implemented, the problem reports addressed in the release are closed and removed from the backlog.

A third chart (c) tracks software reliability. It is calculated by dividing the number of failures reported by the actual hours of usage between releases. The increase in the failure interval associated with Release 2 is probably related to the volatility of Release 2’s content. Releases 1 and 3 have exhibited a failure interval acceptably below the desired target rate.

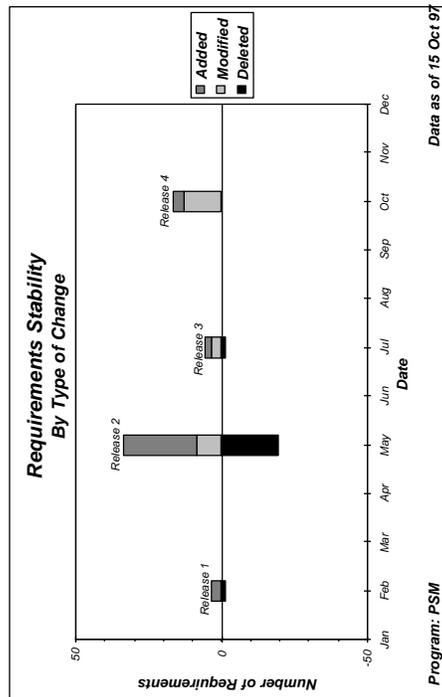
Finally, milestone progress (d) shows that Release 2 took longer than planned and that Release 4 is behind schedule. The delay for Release 2 was probably due to the large number of changes made in that release. The reason for Release 4’s delay is most likely due, again, to changes in release content (see chart a). If this trend continues and is determined to be a problem, the process for release content planning should be reviewed.



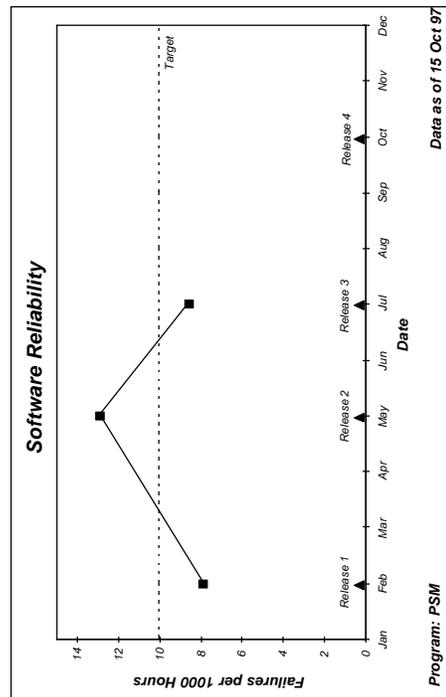
b)



d)



a)

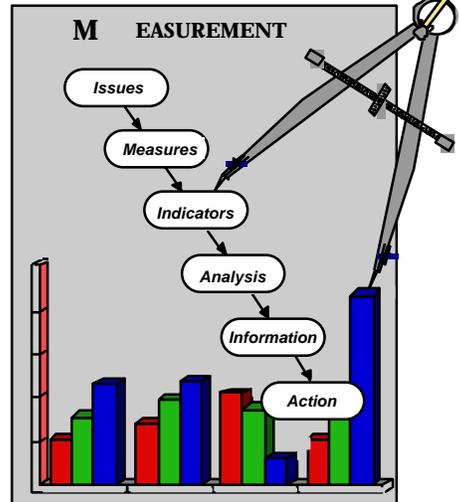


c)

Figure 4-4. Maintenance

PRACTICAL

SOFTWARE



ACQUISITION AND CONTRACT IMPLEMENTATION GUIDANCE

PART 4

ACQUISITION AND CONTRACT IMPLEMENTATION GUIDANCE

The measurement tailoring process described in Part 1 applies to all programs, whether the software is developed organically or acquired through an external agreement. The first two activities of the tailoring process result in a specification of the program manager's measurement requirements (see Part 2 for details). The last activity in the tailoring process integrates that specification into the developer's process.

For programs where the developer is organic, the measurement requirements can be conveyed and negotiated informally. However, when the software is being acquired from an external source, the interface between the program manager and the developer must be managed more formally. This part of the Guide explains how measurement is implemented via a contract between a government organization and a private contractor. These concepts also apply to situations in which software is acquired from another government organization via a Memorandum of Understanding/ Agreement (MOU/MOA) or Inter-Service Support Agreement (ISSA).

This part of the Guide is organized into three chapters:

- Chapter 1, Contract Implementation Guidance, describes the activities by which the program manager's measurement requirements are integrated with the developer's software process.*
- Chapter 2, Sample RFP Wording, contains sample wording that may be inserted into a Request For Proposal (RFP) or Contract along with the rationale for each contract requirement.*
- Chapter 3, Additional Sample Material, contains sample WBS structures for both Weapons and AIS systems in addition to a draft outline for a software measurement plan.*

TABLE OF CONTENTS

CHAPTER 1 – CONTRACT IMPLEMENTATION GUIDANCE..... 251

1.1 Contract Planning and Preparation..... 251

1.2 Proposal Evaluation..... 251

1.3 Negotiations..... 252

1.4 Contract Modifications..... 253

CHAPTER 2 – SAMPLE RFP WORDING..... 255

CHAPTER 3 – ADDITIONAL SAMPLE MATERIAL..... 261

CHAPTER 1 – CONTRACT IMPLEMENTATION GUIDANCE

Through the contracting process, the program management team ensures that the selected developer provides the measurement data necessary to manage the program effectively. This chapter identifies the measurement considerations to be observed in each of the four steps of the contracting process: contract planning and preparation, proposal evaluation, negotiations, and contract modifications.

This contracting process is applicable to programs in both the development and software support phases, although the issues and the selected measures may be different. When adding measures to an existing contract, the contract planning and preparation and proposal evaluation steps are generally not implemented.

1.1 CONTRACT PLANNING AND PREPARATION

During *Contract Planning and Preparation*, software measurement requirements are identified and documented. These requirements are defined using the process described in Part 2 of the *Guide*. The RFP provides a vehicle for communicating these requirements to potential contractors. Chapter 2 contains sample wording that may be inserted into a RFP for this purpose. In the RFP, the program management team also requests historical data necessary to substantiate the developer's proposal and to conduct an independent feasibility analysis of the proposed software plan. Chapter 2 also provides wording for this. In parallel with RFP development, the program management team will develop independent estimates of size, schedule, effort and cost to use in evaluating the contractors' proposals.

1.2 PROPOSAL EVALUATION

Contractors respond to the RFP with a proposal explaining how their measurement process will meet the program manager's information needs. Each prospective contractor's proposed measurement process must be evaluated as part of the overall

proposal evaluation process. This evaluation includes an assessment of the developer's understanding of the issues specified in the contract, as well as the effectiveness of the process and measures that the developer is planning to use to address the issues. The evaluation should assess the adequacy of proposed measurement data definitions and methodologies. An on-site evaluation at each developer's facility may be performed to validate the proposed measurement process identified in each proposal.

The proposal evaluation team also needs to assess the feasibility of each proposed developer's estimates with respect to size, schedule, effort, and cost. The team may use software development cost and schedule estimating models to compute performance parameters and look for inconsistencies that need to be reconciled. In addition, the developer's estimates should be compared to the independent estimates done by the program office. Feasibility is also evaluated with respect to the historical data provided.

1.3 NEGOTIATIONS

Once a developer has been selected, the negotiator begins the task of finalizing the measurement requirements in the contract. In the proposal, the developer should have identified any concerns with the specified issues and measures and proposed alternatives as appropriate. Alternative measures must adequately address the identified issues and be used internally to manage the software development.

The developer should identify any problems associated with the specification guidance including the data items to be collected, the collection and reporting levels, and the method for counting actuals. The developer should describe his proposed implementation of the measures, including definitions, estimation and actual measurement methodologies, and data reporting mechanisms. All of these items must be agreed upon during negotiations. The results of the negotiations should be documented in the contract or in an approved software measurement plan.

1.4 CONTRACT MODIFICATIONS

It is important to understand that the software issues will change during the program, and the measurement and contracting process has to be flexible enough to accommodate these changes. Different measures may be required to address new or modified issues. Changes may be required to specifications of existing measures such as data definitions, data elements, or reporting mechanisms.

Contract modifications may also be necessary to implement measurement requirements for existing programs that did not originally require measurement. Even in these situations, the program management team should still go through the process of defining program issues and measurement requirements. The team should work with the existing developer to determine if any measures are already available that address these issues. The developer may already be collecting some useful measures

CHAPTER 2—SAMPLE RFP WORDING

This chapter contains sample wording that may be inserted into a RFP, contract or other agreement between the program manager and developer. The first section, Contract Management, contains sample wording that may be used to request software measurement data, address questions about that data, and obtain a software measurement plan.

Each of the following sections contain a description of the rationale for each request, followed by sample wording that may be directly inserted into an agreement. The sample wording is in quotes in the shaded area.

Requirements for Software Measures

Contract wording to require the collection of measurement data should be specified. In the RFP, the program management team should detail the software issues identified and the measures required to address them. For each measure required, the program management team should identify the specification guidance including the data items to be collected, the collection and reporting level, and the method for counting actuals as complete. The following paragraph specifies monthly reporting, but this may be adjusted as appropriate for your program.

“The developer shall provide the software measures specified in Paragraph XXX on a monthly basis. For each measure, data shall be provided for each data item at the specified collection level. Data shall not be considered as actuals until the criteria for counting actuals has been successfully met.”

Requirements for most software measures should include both planned and actual performance data. Any changes to the planning data should be identified, quantified, and provided to the program manager. A few measures may not be accompanied by planning data (such as defect and requirements stability data).

“For all the measures specified in Paragraph XXX, the developer shall provide an initial plan and periodic actual data. Any time

that the planning data for any of the detailed measurement parameters changes, the developer shall provide an updated plan within 30 days of the change.”

For each measure, the developer should propose measurement definitions, methodologies, and data reporting mechanisms.

“For each measure specified in Paragraph XXX, the developer shall provide a measurement definition, an estimation methodology, the method used to measure actual data, and the data reporting format and associated mechanism. This information shall include a description of any tools utilized.”

Planned and actual data shall be based on the same measurement methodology. Any changes in definitions, estimation methodologies, or actual measurement approaches shall be documented within 30 days of the change and shall require approval of the program manager”

The data should be provided in a timely manner, as soon as possible after data collection occurs (the wording recommends within 30 days - you may want to modify this time period). The lag time between data collection and reporting should be minimized so that early warning indicators are available early.

“The required measures shall be delivered within 30 days after the data is collected.”

Developer Access

Throughout the development, the program management team should periodically review the measurement processes. In addition, the measurement analyst will have questions about some of the data. The measurement analyst needs to have access to the developer to answer these questions and to gather the subjective data that supports the proper interpretation of the quantitative data.

“The developer shall provide direct access to the program team to facilitate open communications with respect to the measurement process. The developer shall also provide explanations and rationale for changes, answer questions, and provide clarifications regarding the measurement process and associated data and information.”

Data Alternatives

The measures specified in the RFP represent the needs of the Program Manager. The developer may request the substitution of an alternate software measure, if the alternative measure provides similar insight into the associated software issue. The alternative measure should be readily available from the developer's development process and should be used internally by the developer.

“In the event that a specified measure is unavailable, the developer shall submit a request for substitution. This request shall identify an alternative measure with a data definition, rationale for the change, a description of how this measure addresses the identified issue, and a description of how this measure will be used internally. The alternative measure must be readily available from the software development process.”

Draft Measurement Plan

The developer should be required to develop a measurement plan which specifies which issues and measures will be addressed during the program. The plan should identify the software measurement process to be used and specify how the developer will use the measurement information. Chapter 3 contains a sample outline of a software measurement plan

“The developer shall submit a draft measurement plan which specifies the issues to be addressed, the measures to be utilized, and definitions of specified measures and measurement methodologies. This plan shall identify the measurement approach to be utilized including a description of how measurement information will be utilized in the developer's internal management of this program, how data will be collected and utilized, points of contact, responsibilities, and organization communications and interfaces.”

Proposal Evaluation Data

Proposal evaluation should include an assessment of the feasibility of the software development plan based on information provided in the proposal, historical data about the developer's performance, and

independent estimates prepared by the program management team. Information used for this assessment includes:

- **Required Productivity-** The developer should provide an assessment of the productivity required to successfully execute the proposal, based on the planning parameters provided in the proposal. The developer should include a definition of any tools or methodologies used.
- **Product Size, Effort Allocation, Milestone Dates** The developer should submit estimated data for each of these measures. This allows the proposal evaluation team to do an independent feasibility assessment on each bidder. The data should include a data definition and estimation methodology.
- **Historical Data-** The developer should submit actual data (product size, effort allocation, milestone dates, cost profile, and productivity) from completed programs. Data should be collected from programs that are similar in domain, size, and complexity to the proposed program.

The first two items usually are required parts of the proposal, whether or not the measurement approach described in this Guide is applied. The following RFP wording is suggested to collect historical data to substantiate the potential developer's proposal and to conduct the feasibility analysis:

“The developer shall provide historical data from at least three completed programs to support the proposal. The technical characteristics of the historical programs shall be similar to the proposed system with respect to domain, size, and complexity. If the developer does not have experience within these criteria, data from other completed programs shall be provided. The data shall include measures of size, schedule, effort, cost, and productivity by WBS element. Any models and methodologies used shall be documented for each historical program to a sufficient level of detail to allow replication by the evaluation team”.

CHAPTER 3 – ADDITIONAL SAMPLE MATERIAL

Two items that are important to implementing an effective measurement process are the WBS and the measurement plan. This chapter contains examples of a weapons system WBS, automated information system WBS, and software measurement plan outline.

Figure 3-1 contains a sample WBS that can be used in a weapons software development. Once the developer is selected, it is important to modify the WBS used during the selection process to map to the developer's negotiated WBS. The revised WBS provides a tie between the estimated measures and the actual measures. It also insures that the cost account elements map to the same WBS that is used for data collection. Figure 3-2 contains a sample WBS that can be used in a AIS software development.

Figure 3-3 contains a sample outline of a Software Measurement Plan. This plan should be modified as needed to accommodate different program information needs and developer processes. It may be included in the Software Development Plan (SDP), Software Maintenance Plan (SMP), Computer Resource Life Cycle Management Plan (CRLCMP), or similar planning document.

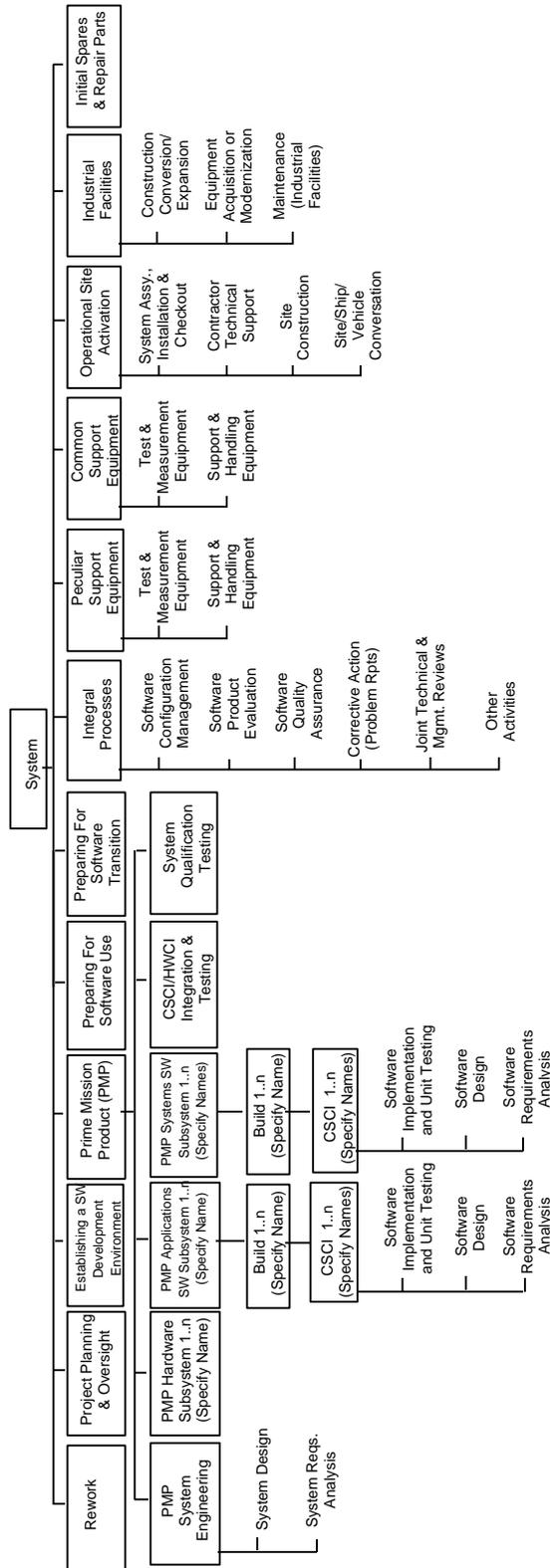


Figure 3-1. Sample WBS for Weapon System

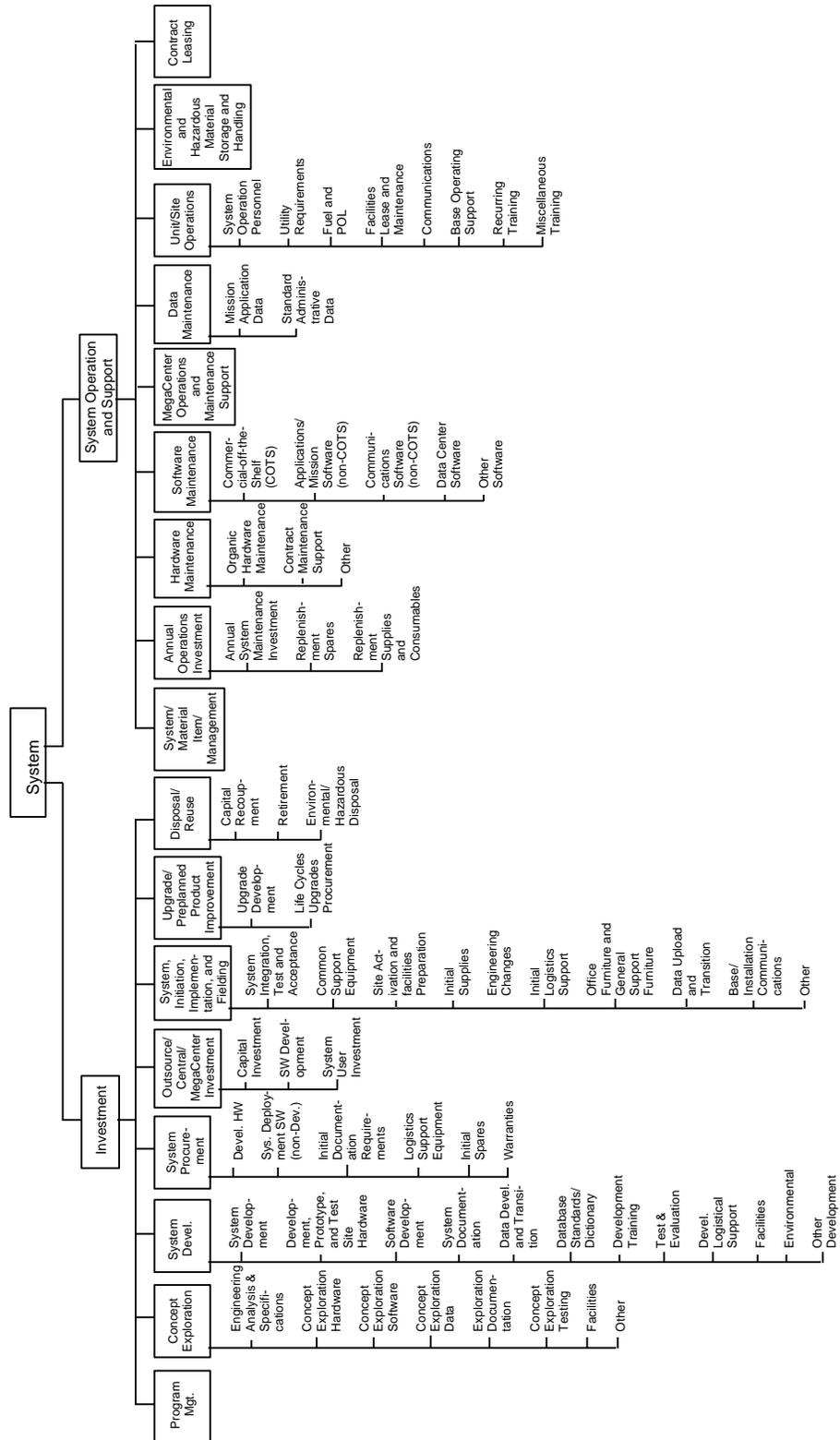


Figure 3-2. Sample WBS for AIS System

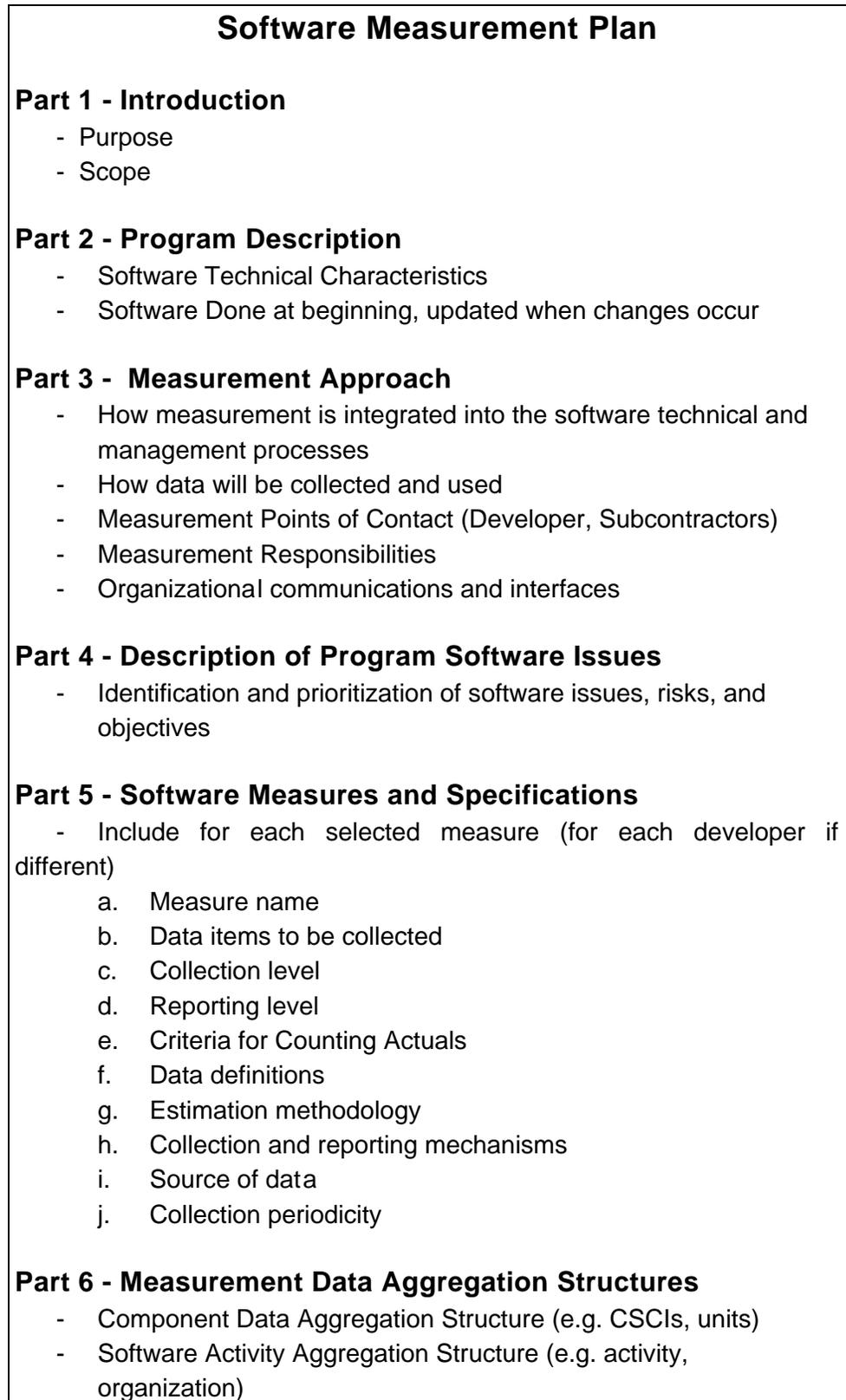
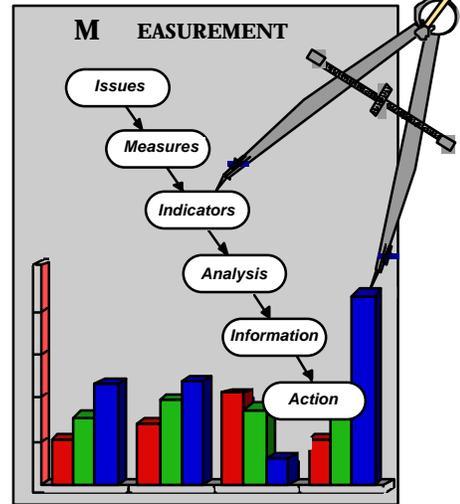


Figure 3-3. Sample Outline for Software Management Plan

PRACTICAL

SOFTWARE



SOFTWARE MEASUREMENT CASE STUDIES

PART 5

MEASUREMENT CASE STUDIES

Practical Software Measurement was developed to show how measurement can be used to address the software issues faced by today's DoD Program Manager. To better illustrate how the measurement process is implemented for different types of programs, this part of the Guide contains two software measurement case studies. The first (Part 5A) is based upon a major shipboard Weapons System development. The second (Part 5B) is based on the development of an Automated Information System designed to manage military personnel information. These case studies describe how the measurement process is tailored and applied to meet specific, and sometimes unique, program management requirements.

The PSM case studies address the issues and challenges that most DoD program managers face in planning and implementing software intensive development programs. The case studies concentrate on the issues the program manager must address with respect to managing the program within defined acquisition and technical constraints. They show how measurement is used to help make decisions concerning these issues. The case studies also illustrate how any DoD software development program can benefit from implementing a tailored set of software measures within an effective measurement process.

Although the Practical Software Measurement case study parameters are based on actual DoD program characteristics, the program scenarios, including the described system architectures, program names, and program organizations, are fictitious.

| | |
|---|------------|
| WEAPONS SYSTEM CASE STUDY..... | 271 |
| CHAPTER 1 - PROGRAM OVERVIEW..... | 275 |
| 1.1 Introduction..... | 275 |
| 1.2 Program Technical Approach..... | 277 |
| 1.2.1 System Requirements Definition and Design Analysis..... | 277 |
| 1.2.2 DDG 51 C ⁴ I Baseline System Description..... | 278 |
| 1.2.3 System Requirements and Design Recommendations..... | 280 |
| 1.3 Program Management Approach..... | 281 |
| CHAPTER 2 - PROGRAM PLANNING AND ACQUISITION..... | 284 |
| 2.1 Software Program Planning..... | 284 |
| 2.2 Software Acquisition..... | 287 |
| 2.2.1 Request for Proposal..... | 287 |
| 2.2.2 Proposal Evaluation..... | 288 |
| 2.2.3 Award..... | 290 |
| 2.2.4 Negotiations..... | 292 |
| CHAPTER 3 - DEVELOPMENT PHASE..... | 296 |
| 3.1 Tracking Development Performance..... | 296 |
| 3.1.1 Software Measurement Overview..... | 296 |
| 3.1.2 Software Issue Identification and Analysis..... | 297 |
| 3.2 Revising The Development Plan..... | 306 |
| 3.3 Software Delivery..... | 308 |
| 3.4 Epilogue..... | 309 |
| AUTOMATED INFORMATION SYSTEM CASE STUDY..... | 312 |
| CHAPTER 1 - PROGRAM OVERVIEW..... | 317 |
| 1.1 Introduction..... | 317 |
| 1.2 Air Force Business Process Modernization Initiative..... | 319 |
| 1.3 Program Description..... | 320 |
| 1.4 System Architecture and Functionality..... | 322 |
| 1.4.1 Current Personnel System..... | 322 |
| 1.4.2 Military Automated Personnel System (MAPS)..... | 323 |
| CHAPTER 2 - GETTING THE PROGRAM UNDER CONTROL..... | 327 |
| 2.1 Evaluating the Software Development Plan..... | 327 |
| 2.2 Revising the Software Development Plan..... | 330 |

| | |
|---|------------|
| 2.3 Tracking Performance Against the Revised Plan..... | 334 |
| CHAPTER 3 - EVALUATING READINESS FOR DELIVERY..... | 341 |
| 3.1 Increment 1..... | 341 |
| 3.2 Increment 2..... | 345 |
| CHAPTER 4 - INSTALLATION AND SOFTWARE SUPPORT..... | 349 |
| 4.1 Increment 1 Installation..... | 349 |
| 4.2 Software Support..... | 350 |
| 4.3 Epilogue..... | 353 |



WEAPONS SYSTEM CASE STUDY

PART 5A

WEAPONS SYSTEM CASE STUDY

The Weapons System case study is based on the development of a complex shipboard Weapons System designed to integrate multiple platform target engagement and weapons management functions into an existing system baseline. In this scenario, measurement is used to help plan and track the software development effort from the inception of the program through system deployment. The development approach is based on the upgrade of an existing system using Commercial Off the Shelf components and reused software in a revised architecture. The developer is a competitively selected contractor who works closely with the Navy program manager to identify and resolve issues typical in a large development program. These issues include software requirements and size growth, incremental schedule slips, and overall software development productivity shortfalls.

The Weapons System case study is organized into three chapters:

- Chapter 1, Program Overview, describes the technical and management aspects of the software development effort.*
- Chapter 2, Program Planning and Acquisition, shows how measurement can be used to define and evaluate a realistic software development plan.*
- Chapter 3, Software Development, illustrates how measurement helps to identify and track software issues, and how the program manager uses measurement information to evaluate development status and make informed program decisions.*

CHAPTER 1 - PROGRAM OVERVIEW

This chapter introduces the example Navy program and describes the technical and management aspects of the development effort. The program scenario is based on a major program upgrade to an existing Navy surface ship Command, Control, Communications, Computer, and Intelligence (C⁴I) system. The upgrade integrates multiple platform target engagement and weapons management functions into an existing software functional baseline. It includes the addition of new software functions to the system, as well as modifications to the existing software baseline.

Although the case study parameters are based on actual Navy surface ship characteristics, the program scenario—including the described system architecture, program names and organizations—is fictitious.

1.1 INTRODUCTION

In the early 1990's, the Navy began to recognize a growing need for its ships and aircraft to operate interactively in a multiple threat environment. This need was clearly demonstrated during the Gulf War where well coordinated engagements, which integrated the capabilities of a number of different platforms, provided significant tactical advantages.

To define its changing mission requirements, the Navy initiated a concept study to determine the feasibility and effectiveness of integrating a multiple platform target engagement capability into the fleet. The results of the study, completed in 1994, validated the need for the proposed engagement capabilities and recommended an implementation approach which built upon the Navy's existing C⁴I tactical systems on various platforms. The study recommended that the Navy initially focus on the upgrade of its existing surface combatants with new communications, engagement management, and weapons control functions. These new functions would be designed to allow two or more ships to engage the enemy as a single entity. With the new capabilities, one ship would be able to manage the overall sensor and target scenarios for the entire group

and assign, launch, and control the weapons on the other ships using advanced tactical communications links.

The Navy decided that the Arleigh Burke DDG 51 class of guided missile destroyers (DDG) would be the first ships to receive the capability upgrade, as it was the largest and most modern class of DDGs in the fleet. It named the program the **DDG 51 Surface Ship Concurrent Weapons Engagement Upgrade Program**, or **DDG 51 SCWE** for short. The objective of the DDG 51 SCWE program was to define, develop and integrate a new concurrent weapons engagement function into the existing C⁴I system on the Arleigh Burke DDGs. Most of the efforts were to be focused on the coordinated employment of long-range surface-launched weapons, with an emphasis on the Tomahawk Cruise Missile.

The DDG 51 SCWE program was projected to require significant changes in the architecture of the existing DDG 51 C⁴I system, especially with respect to the software. Existing software functions and interfaces required numerous changes, and the multiple platform communications, target management, and weapons management functions had to be developed and integrated. New acquisition policies made the use of an Open Systems Architecture (OSA) and Commercial-Off-The-Shelf (COTS) software components almost mandatory, and the overall business environment required that the program be well managed in terms of delivered functionality, and in meeting pre-defined cost and schedule objectives.

The Navy recognized the critical nature of the software development component of the DDG 51 Surface Ship Concurrent Weapons Engagement Upgrade Program and emphasized the need for effective software management as part of the overall program management approach. Understanding this need, the Naval Sea Systems Command (NAVSEA) assigned Captain Katherine McLain, USN, as the Program Manager. Captain McLain held an advanced degree in Electrical Engineering from Stanford University, and she had served as the software technical manager on a number of previously successful Navy development programs. After completing the Program Manager's course at the Defense Systems Management College (DSMC), Captain McLain assembled her program management team at NAVSEA. Her office was designated as PMO-551. The award date for DDG 51 SCWE Engineering and Manufacturing Development (E&MD) was

projected for mid 1996. To ensure a successful program, a considerable amount of work had to be completed before award.

1.2 PROGRAM TECHNICAL APPROACH

1.2.1 System Requirements Definition and Design Analysis

Based on her previous experience, Captain McLain was familiar with the software architecture and capabilities of the existing DDG 51 C⁴I system. Like most of the large Navy systems developed in the late 1980's, the system on the Arleigh Burke DDG class was built around the AN/UYK-43 Navy standard computer, which centrally handled the processing for most of the system's different functions. The original C⁴I systems on the DDG 51's had been incrementally upgraded since they were first deployed to integrate new sensor and weapons capabilities. Over time, the system design had proven to be effective and reliable.

The software for the DDG 51 C⁴I system was implemented largely in CMS-2, the Navy's pre-Ada standard high order programming language. The functions where real-time processing and timing considerations were critical were coded in assembly language. The original software had been developed using a modified DoD-STD-2167 software development process and was currently being maintained by the original developer under a separate maintenance contract.

The mission requirements driving the DDG 51 SCWE program provided some significant technical and program management challenges for PMO-551. Captain McLain felt that one of the keys to a successful development program was a well defined set of system requirements. As part of her acquisition strategy, PMO-551 awarded a series of competitive System Requirements Definition-Design Analysis Study Contracts. These design study contracts were specifically implemented to accomplish the following:

- Provide a definitive analysis and characterization of the existing DDG 51 C⁴I system hardware and software architectures.
- Develop an approved set of system level requirements for inclusion in the E&MD Request For Proposal (RFP).

- Develop innovative system design alternatives. These alternatives in particular were focused on the use of COTS hardware and software components, and on the integration of an OSA into the existing system to support future capability growth.

1.2.2 DDG 51 C⁴I Baseline System Description

The results of the System Requirements Definition–Design Analysis Study efforts provided a detailed characterization of the existing DDG 51 C⁴I software architecture. Figure 1-1, a simplified diagram, shows that the system consisted of six primary software functions, all resident in the AN/UYK-43 computer. Functional data interfaces to the External Communications subsystems, the Weapons subsystems, and to own-ship sensors such as Navigation, Radar, Sonar, and Electronic Support Measures (ESM), were through the System Control software function using a Navy Tactical Data System (NTDS) interface protocol. Two way data communications to the Command Display and Control consoles was also provided by the System Control software through an NTDS interface.

Each of the six primary software functions in the system was comprised of three to six Computer Software Configuration Items (CSCIs), as defined in DoD-STD-2167. In all, there were 24 CSCIs in the baseline system. The software architecture was well defined, and the original developer had done an excellent job of allocating and mapping the original software requirements to the CSCIs. There was a full set of software technical specifications available, but these had not been kept uniformly up to date, especially with respect to the incremental design changes.

The DDG 51 C⁴I system software was relatively large and somewhat complex. The various software functions worked together to integrate real-time data from a variety of distinct combat and ship control subsystems and processed the data into the information needed to effectively engage enemy targets.

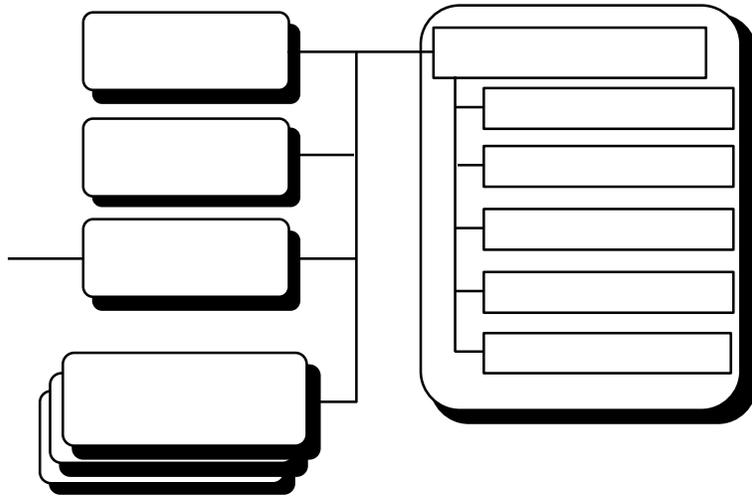


Figure 1-1. DDG 51 Weapons System Software Architecture Baseline System

Each of the six primary software functions addressed a unique set of functional requirements:

- **System Control**- The System Control function included the AN/UYK-43 operating system and provided the primary software services functions for the system. Its functions included system database management, initial program load, configuration and reconfiguration management, and display control.
- **Surface Control** -The Surface Control function addressed own-ship maneuvering and navigation requirements and calculated ship's heading, speed, and position on a real-time basis. It also included capabilities that helped position the ship with respect to other surface contacts.
- **Target Tracking** -The Target Tracking function integrated and correlated all sensor data, and calculated, evaluated, and tracked surface, subsurface, and air contacts on a real-time basis.
- **Threat Evaluation** -The Threat Evaluation function correlated all of the sensor data from all targets and through a series of complex threat algorithms calculated and prioritized each target within an overall threat profile.

- **Target Engagement** - The Target Engagement function included software that managed the overall enemy engagement and controlled all weapons allocations to individual targets. It also assigned weapons presets based on the calculated target parameters. This function was one of the most critical in the system.
- **External Communications** - The External Communications function provided interfaces between the C⁴I system and a number of tactical digital communications data links. These data links provided for the exchange of contact and targeting information with other off-ship platforms.

Together, the DDG 51 C⁴I system software functions included over one million logical lines of source code distributed among 24 CSCIs as shown in Figure 1-2.

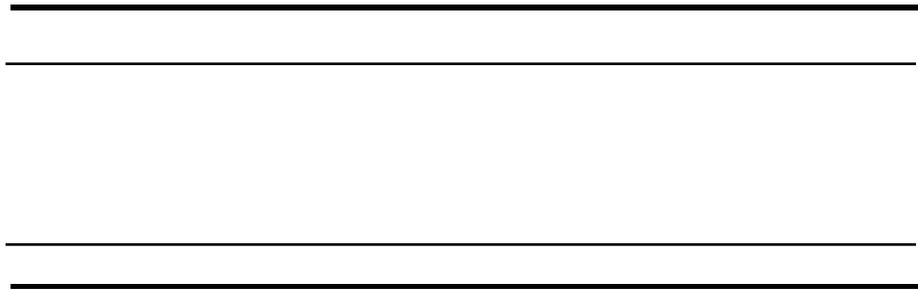


Figure 1-2. DDG 51 Baseline System Software Description

1.2.3 System Requirements and Design Recommendations

The System Requirements Definition-Design Analysis study effort provided a definitive set of system level requirements for the DDG 51 SCWE upgrade program. After reviewing the requirements with her staff, Captain McLain had a clear understanding of the magnitude of the changes required for the existing DDG 51 C⁴I system. She knew that the new multi-ship engagement functions would have a significant impact on the existing system and software architectures. She also estimated that the current software baseline would more than double in size.

In addition to the new multiple platform engagement management and weapons control functions, the system level requirements included the need for:

- New display processing capabilities.
- New assignable command and display workstations.
- Automatic reconfiguration of the engagement control functions in the event of a platform specific failure.
- Enhanced weapons safety provisions.
- Advanced multiple ship and aircraft contact correlation.
- Additional secure digital data links.
- An increase in the overall system processing capacity.

Even at this point in the program, Captain McLain knew that managing the requirements, especially those allocated to the software, would be important to the success of the upgrade program.

Given the large amount of functionality that was to be added to the baseline DDG 51 system, the System Requirements Definition-Design Analysis studies also proposed a number of system and software design alternatives that addressed the Navy's desire for development affordability and lifecycle cost savings. These alternatives were all based upon retaining a large part of the baseline system hardware and software and adding the new capabilities using COTS components integrated via an OSA local area network. In all cases, the alternatives addressed the addition of new processing and display capabilities using advanced display workstations.

The design alternatives outlined in the study recommendations maintained much of the integrity of the existing system hardware and software. In addition, it addressed the Navy's policy to embrace open commercial interface standards and COTS products in implementing the new functionality.

1.3 PROGRAMMANAGEMENTAPPROACH

With the system specifications and the design studies completed, Captain McLain began to concentrate on the program's acquisition requirements. With her own program office personnel, and support from the Naval Surface Warfare Center (NSWC) in Dahlgren,

Virginia, Captain McLain believed she had a very capable acquisition team, especially with respect to software.

With the changes in the DoD business environment over the past several years, Captain McLain knew that the DDG 51 SCWE program would be very visible within the Navy and DoD. It was one of the first major programs to fully address the DoD's acquisition reform requirements, which included the extensive use of commercial product standards, COTS hardware and software, software reuse, and the integration of a new OSA.

One of the key aspects of acquisition reform was its emphasis on less developer oversight by the acquisition organization. This requirement led to several very critical software decisions by Captain McLain:

- The developer had to have a mature software development process, and the developer's overall capability with respect to software process would be a key consideration in source selection.
- Insight into the software processes and products, across all activities and development phases, would be provided by a practical software measurement process. Both PMO-551 and the developer would use software measurement to identify and manage the software development issues.
- The government and developer organizations would function as an integrated project team and communicate on an objective basis.
- The software would be developed using a tailored MIL-STD-498 development process. Along with this, a detailed software Work Breakdown Structure (WBS) would be implemented to manage the program's development products and activities.

Captain McLain planned to award the development contract to a capable software developer with a proven history of success. She made it clear that she expected both her PMO-551 organization and the developer to address the software issues in an objective manner. Captain McLain knew that delivering the specified requirements to the fleet within the program's schedule and funding constraints would be a significant challenge.

CHAPTER 2 - PROGRAM PLANNING AND ACQUISITION

With the system requirements completed, PMO-551 began to focus on the detailed planning for the DDG 51 Surface Ship Concurrent Weapons Engagement Upgrade Program. Before awarding the development contract, Captain McLain and the Navy program team had to define a feasible software development plan, issue the Request for Proposal (RFP) and evaluate the submitted proposals during source selection. Even at this early planning stage, Captain McLain used information derived from the software measurement process to support her planning objectives.

This chapter of the case study shows how software measurement can help during the Program Planning and Acquisition phase of software development. **The activities that take place during this phase set the stage for project success or failure.** Their importance cannot be over-emphasized. It is during this time in the program that the Program Manager implements the measurement process as an integral part of the overall program management structure. Software measurement is used to ensure that the software development plan is realistic and the software developer has the capability to successfully complete the job.

2.1 SOFTWARE PROGRAM PLANNING

The most important software planning task for Captain McLain and her staff was to develop a realistic DDG 51 SCWE software implementation plan. Aware of the direct relationships between the overall size of the software and development cost and schedule, Captain McLain and her software engineering team generated independent estimates for the key parameters.

PMO-551 began with a preliminary allocation of the system requirements to a notional set of software components, keeping in mind that they would be retaining much of the existing software and using a significant amount of COTS software to implement the new functions. Based on these requirements, and the size of the existing code, the team estimated the size of the software to be developed. They generated estimates of development effort and

schedule using two techniques. First, they had their own engineering rules-of-thumb for development productivity (lines of code per staff month) and code-production rates (lines of code per calendar month). These rules-of-thumb were derived from their past experiences with similar C⁴I programs. In both cases, these engineering estimates encompassed the entire software development cycle (from software requirements analysis through system integration and test). Secondly, they used a commercially-available software cost estimating model.

From these estimates, Captain McLain concluded that the schedule required to realistically complete the software was between 74 and 78 months, starting with contract award and ending with certification testing and delivery. Unfortunately, this time was somewhat longer than the schedule the Navy defined. The ship deployment and shipyard availability schedules were driving the DDG 51 SCWE development schedule, and the software was the “long pole in the tent”. Captain McLain knew, based on her analysis, that the schedule was going to be a high risk area and took steps to address this issue in her plan.

Captain McLain understood that the program budget and functional requirements were essentially set, so she looked at several options for reducing the planned software development schedule.

Captain McLain updated her plan to include the following:

- More parallel implementation of the software development activities. This included an incremental development approach for the software with the functionality developed and integrated into multiple builds and the overlapping of specific software implementation, integration, and test activities.
- Maximize use of COTS and non-developed (NDI) software and reuse of as much of the existing code as possible.
- Assumption of relatively high software development productivity based on her plan to make the developer’s software process capability a key criterion for contract award.

After these modifications, the PMO-551 re-ran the software estimates. Specifically, the PMO-551 planning team assumed the

amount of code that had to be developed was smaller due to the use of additional COTS software and more reused software components from the baseline system. The resulting DDG 51 SCWE software development schedule showed that the full set of software requirements could be implemented in 66 months, within the original budget objective. This estimate was close to the delivery target date set by the Navy.

From a technical perspective, Captain McLain decided that any new software developed for the system should be developed in Ada, using the Ada 95 standard.

The results of the estimation process helped the PMO-551 planning team complete their tailoring of MIL-STD-498. The risk areas and the issues they identified in their software analysis helped to define the required MIL-STD-498 activities and products.

At the completion of the PMO-551 planning process, Captain McLain had a pretty good idea of what her software development risks were and where she would have to focus her attention during the development.

Although her development plan was not without some risk, it was realistic. Most importantly, she had a clear picture of the program's software development issues:

- The realism of the software schedule and the capability of the developer to meet the planned milestones.
- The real possibility for growth in the software requirements.
- The ability of the developer to adequately staff the software development effort.
- The overall impact of cost and schedule constraints on the ability of the developer to build quality into the software.
- The adequacy of the developer's software process capability.
- The adequacy and effectiveness of the software development technical approach.

2.2 SOFTWARE ACQUISITION

2.2.1 Request for Proposal

After the independent PMO-551 software development plan was complete, Captain McLain turned her attention to issuing the DDG 51 SCWE Request for Proposal (RFP) and to choosing a capable developer. The results of the design analysis studies were made available to all bidders. At the bidder's conference, Captain McLain made it clear the successful bidder would have to demonstrate an effective software development process capability. With the success of the program tied to the overall capability of the software developer, Captain McLain specifically addressed her software development requirements in the RFP. In their proposals, the bidders were required to provide the following:

- Their preliminary allocation of the system level requirements provided with the RFP to a proposed software architecture.
- Their approach for using the existing DDG 51 software as the baseline for DDG 51 SCWE software development.
- Their proposed use of COTS software components and an OSA in the redesigned system.
- A comprehensive set of software data describing the bidder's performance on similar development programs. This data included the sizing, schedule, effort, and problem report data, as well as the program descriptive data required to evaluate the developer's software development performance.
- A detailed software measurement plan that linked the program's risks and issues to defined measures and that explained how the software measurement data would be used to track software progress and quality and support objective communications between the Navy and developer teams.
- A detailed description of the proposed software development processes and activities, coupled to an overall DDG 51 SCWE software development plan. A detailed software WBS was also required, as well as quantified estimates of key software parameters related to the proposed software development approach.

Captain McLain also required that each bidder submit, as part of their proposal, a summary of defined issues resulting from their analysis of the technical program and planning parameters and the innovative approaches they would implement to address these issues.

2.2.2 Proposal Evaluation

The RFP was released in the fall of 1995, and a total of five proposals were submitted. Of these five, two were considered by the source selection team to be in the competitive range. Each of the two prime contractors on these two bids was teamed with several subcontractors. After a detailed evaluation of each proposal, a recommendation for award was forwarded to the Program Manager. There were many aspects about the winning proposal that impressed the source selection team:

- The successful bidder's historical data was credible. The proposal provided clear definitions for software size, schedule, effort, and problem report data, and indicated what was included and what was excluded in the numbers. The data supported the bidder's claim that it had an effective software process.
- The successful bidder's DDG 51 SCWE software development plan was based on achievable performance and productivity objectives, and the rationale for the projections was supported by objective estimates of the associated software parameters. Further, the proposed software development plan included a detailed software WBS mapped to the proposed architecture and development activities. The WBS related the proposed software development process to the bidder's recommendations for tailoring MIL-STD-498.
- The successful bidder's software development plan included an incremental software development approach with a relatively sequential set of development activities allocated between two major builds.
- The successful bidder's proposed measurement program met all of the requirements specified in the RFP and clearly reflected that the bidder had experience in using measurement to support successful development programs. In the proposal, the proposed measures were tied to an accurate assessment of the program issues.

From the systems design perspective, the successful bidder met the defined technical requirements for the DDG 51 SCWE program. The proposed system design, as shown in Figure 2-1, included the following:

- The modification of the existing system architecture to include open system interfaces. This change called specifically for the implementation of an open commercial standard Fiber Distributed Data Interface (FDDI) Local Area Network (LAN) to interface the existing sensors and the new functions to the AN/UYK-43 computer. This design change provided for minimal “breakage” to the existing system and supported an affordable development and future system expansion using cost-effective components.
- The development and integration of new display workstations with integrated processors to handle the new multiple ship engagement functions and associated display and control functions. The proposed workstation design made use of both COTS hardware and software. The workstations were to be interfaced to the baseline system through the FDDI LAN. This approach also addressed the need for an advanced human-machine interface required to implement the new target engagement and weapons management functions.
- The replacement of the existing flat-file data management software in the AN/UYK-43 with a COTS based relational database manager and the use of the same relational database structure for the new applications resident in the new workstations. This design change addressed the large increase in the amount of data that the new system would have to process.
- The reallocation of the revised software functionality between the AN/UYK-43 and the new processors in the display workstations. The proposal included the revision and reallocation of the critical engagement and weapons management functions to the workstation processor.

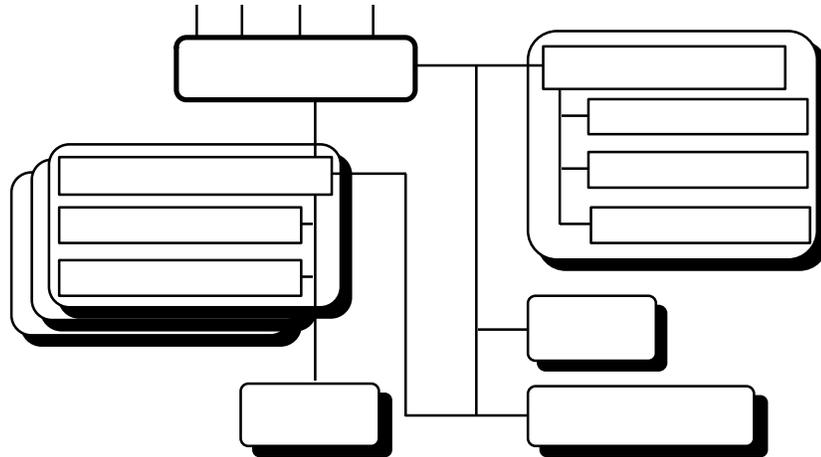


Figure 2-1. DDG 51 Weapons System Software Architecture Upgraded System

In all, the new software architecture added one function, Workstation Control, to the system. The AN/UYK-43 Threat Evaluation function, however, was materially revised and moved to the workstation processor. The AN/UYK-43 Target Engagement function was to be completely rewritten and also moved to the workstation. This work increased the number of CSCIs to 32. The overall amount of software change was significant, but it reflected the nature of the new concurrent weapons engagement mission requirements.

2.2.3 Award

The PMO-551 software measurement analyst, Gary Wilson, was a member of the source selection team. The results of his analysis of the submitted software measurement data were an important factor in selecting the winning bidder. Also significant was the quality of the data in the winning proposal which demonstrated that the developer could objectively identify and manage software issues using software measurement.

The source selection team developed a number of software measurement indicators to support analysis of the proposed software development plans. The critical question was the feasibility of the proposed software development schedule, given the bidder's estimated software size and proposed effort profile. This assessment was based on the calculated software development productivity required to meet the proposed objectives and the

relationship of this required productivity to the bidder’s performance history on previous programs. Did the proposal indicate, for example, that the bidder would have to improve his demonstrated software productivity significantly to meet his proposed schedule, and, if so, was his approach for doing this realistic?

Of equal importance was the relationship between the proposed DDG 51 SCWE software planning parameters. For example, did the scheduled software development activities peak while the development staff was being reduced? These were the types of questions the source selection team was asking.

Gary Wilson developed an indicator which showed the software productivity history of the two bidders in the competitive range. On the same indicator, he graphed the software productivity required for the DDG 51 SCWE, based on the measurement data submitted in each of the proposals (Figure 2-2). The software size estimates were normalized based on how the developer said the code was to be implemented (COTS, NDI, new, or modified), and the schedule and effort data was used as it was submitted.

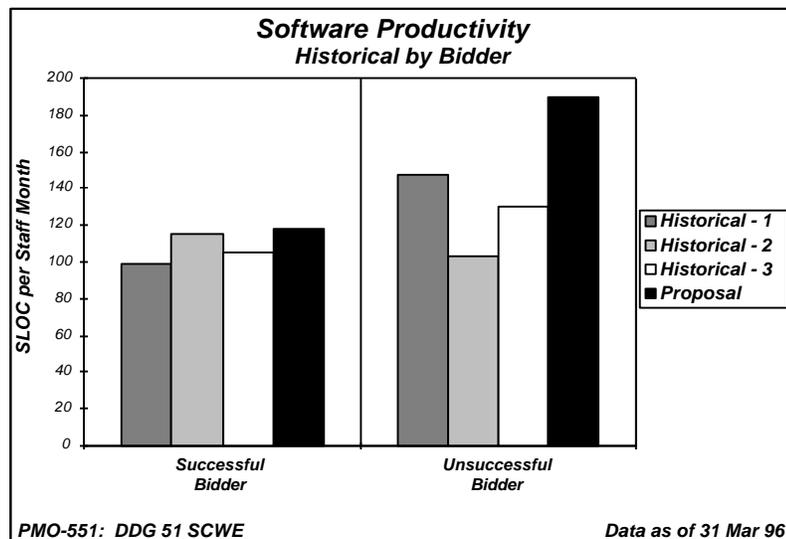


Figure 2-2. Historical Software Productivity by Bidder

The “Software Productivity” indicator clearly showed that the successful bidder had proposed a software productivity rate for the DDG 51 SCWE program that was consistent with his historical performance. The unsuccessful bidder had proposed a significant

increase over his demonstrated productivity rate, but there was no basis for his claim. In fact, when the source selection team investigated, it found that the high productivity rate, as proposed, was tied to an artificially low cost bid in terms of the number of software development staff that was planned for the development program. In addition, the historical data submitted by the unsuccessful bidder was inconsistent, with no clear definitions for how software lines of code, effort, or milestones were measured. The source selection team requested several clarifications from the bidder, but did not receive enough information to substantiate the data.

One concern with the successful bidder's proposal was a somewhat risky 60 month software development schedule. The source selection team, however, felt that the software process, as proposed, was effective enough to mitigate this risk.

When assessed with respect to the results of the cost and technical proposal evaluations, the software measurement results supported award to the higher priced, but more credible bidder. The successful bidder's software data was clearly representative of a development organization that had an established software measurement program embedded into a mature software development process. This bidder's measurement process could best address the software issues and risks associated with the DDG 51 SCWE program.

In May 1996, Captain McLain announced that CDX Systems, Inc. was awarded the development contract for the DDG 51 Surface Ship Concurrent Weapons Engagement Upgrade Program.

2.2.4 Negotiations

During contract negotiations, PMO-551 finalized the software development and measurement plans with the Project Manager from CDX Systems. There were several key objectives:

- Re-affirm the software development start date of 1 July 1996.
- Define the software development schedule, effort, and sizing plans.
- Define clearly which software measures would be applied, how CDX Systems would define each software

measure, and how software measurement data would be transferred between CDX Systems and the program office.

- Ensure the subcontractors were consistent in their use of measurement when reporting to the prime contractor.

The discussions with CDX Systems were extremely important. The developer was able to make sure that the program office software team had a clear understanding of the software data they would be receiving. They would understand what the data represented, how it was measured, and most important, how it related to the CDX Systems software development process. The actual contract wording is presented in Appendix A.

Captain McLain asked her staff to evaluate the software plans for feasibility and consistency. Gary Wilson graphed a set of indicators based on the current CDX Systems planning data. These indicators are shown in Figure 2-3, Figure 2-4, and Figure 2-5.

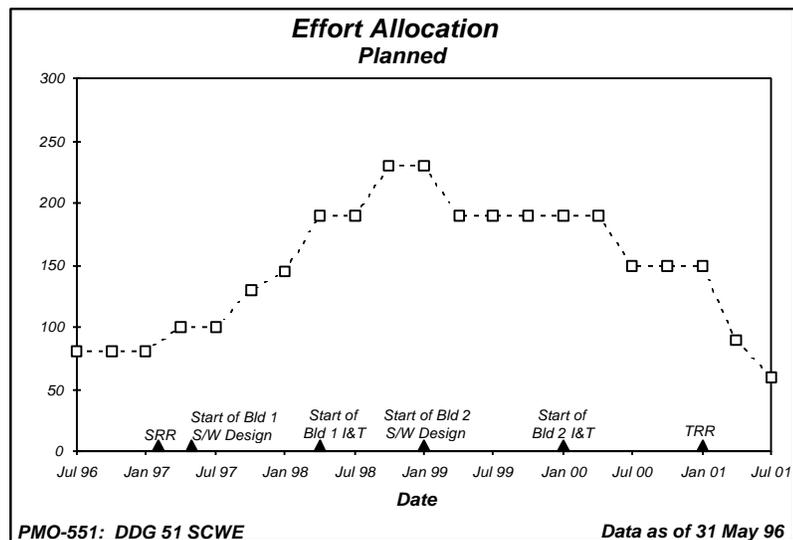


Figure 2-3. Planned Software Effort Allocation

The proposed changes to the existing system resulted in a large increase in the total size of the software. Almost 700K lines of existing software were retained from the baseline system. Even with this amount of software reuse, close to one million new lines of code would have to be written. With the addition of the COTS software components, the total estimated size of the new system was over 3 million logical lines of code. The software effort plan

showed a traditional staffing profile and was consistent with the overall development activities as scheduled. Overall, software planning data represented a well defined software development approach.

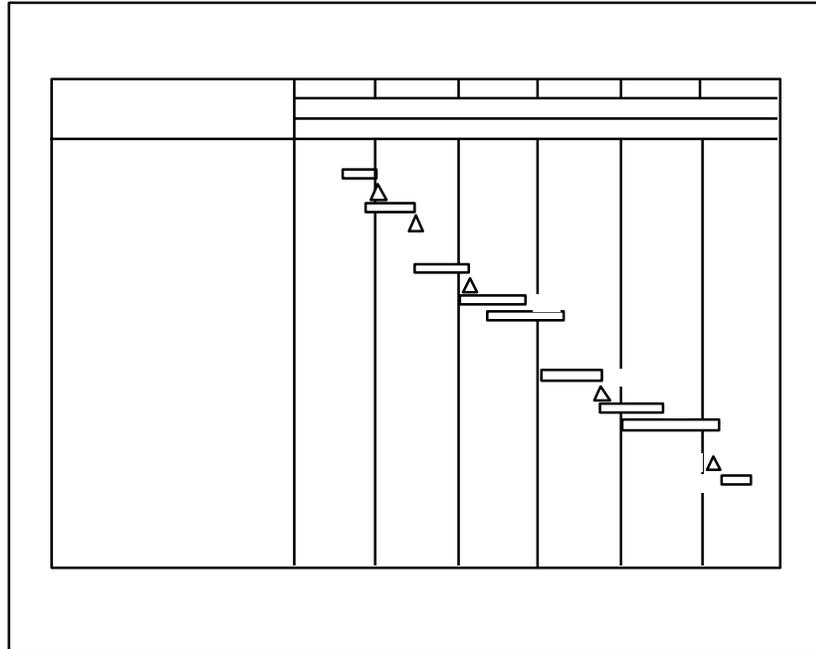


Figure 2-4. Master Software Development Schedule

| Software Size | | | | | | |
|---|----------------|-----------------|------------------|----------------|------------------|-----------------------|
| <i>Estimated Logical Source Lines of Code</i> | | | | | | |
| Build 1 | New | Modified | Existing | Deleted | COTS | Delivered SLOC |
| System Control | 20,000 | 0 | 305,000 | 45,000 | 325,000 | 605,000 |
| Surface Control | 0 | 0 | 175,000 | 0 | 0 | 175,000 |
| Target Tracking | 5,000 | 3,000 | 125,000 | 0 | 0 | 133,000 |
| External Communications | 0 | 0 | 0 | 0 | 0 | 0 |
| Threat Evaluation | 70,000 | 0 | 45,000 | 45,000 | 0 | 70,000 |
| Target Engagement | 190,000 | 0 | 95,000 | 95,000 | 0 | 190,000 |
| Workstation Control | 250,000 | 0 | 0 | 0 | 1,225,000 | 1,475,000 |
| Build 1 - Total | 535,000 | 3,000 | 745,000 | 185,000 | 1,550,000 | 2,648,000 |
| Build 2 | New | Modified | Existing | Deleted | COTS | Delivered SLOC |
| System Control | 0 | 0 | 0 | 0 | 0 | 0 |
| Surface Control | 0 | 0 | 0 | 0 | 0 | 0 |
| Target Tracking | 0 | 0 | 0 | 0 | 0 | 0 |
| External Communications | 30,000 | 0 | 110,000 | 0 | 0 | 140,000 |
| Threat Evaluation | 215,000 | 0 | 135,000 | 135,000 | 0 | 215,000 |
| Target Engagement | 210,000 | 0 | 125,000 | 125,000 | 0 | 210,000 |
| Workstation Control | 0 | 0 | 0 | 0 | 0 | 0 |
| Build 2 - Total | 455,000 | 0 | 370,000 | 260,000 | 0 | 565,000 |
| Total | 990,000 | 3,000 | 1,115,000 | 445,000 | 1,550,000 | 3,213,000 |

PMO-551: DDG 51 SCWE Data as of 31 May 96

Figure 2-5. Software Size Estimate

CHAPTER 3 - DEVELOPMENT PHASE

After the DDG 51 SCWE contract was awarded, Captain McLain began the complex task of managing the software development process. Software measurement activities shifted from evaluating the software plans to tracking performance against those plans. With her own Navy program organization and CDX Systems, Captain McLain believed she had a capable software development team—one that could effectively identify and resolve the expected software development issues and make the program a success.

This chapter explains how software measurement helps identify and objectively analyze the software issues and shows how the Program Manager uses the resulting information to make informed program decisions. For the DDG 51 SCWE program, software measurement has become an integral part of the Program Management process and provides PMO-551 with an effective tool for communicating with the developer.

3.1 TRACKING DEVELOPMENT PERFORMANCE

3.1.1 Software Measurement Overview

DDG 51 SCWE software development officially started with the kickoff meeting between CDX Systems and PMO-551 on 1 July 1996. At the kickoff meeting, Captain McLain explained it was important that her software engineering staff communicate effectively with the developers at CDX Systems. She also stated her expectation that they take an integrated team approach to resolving any technical and management issues. Captain McLain addressed the importance of an effective software measurement process and emphasized she would use the software data to help manage the program and to identify problems as early as possible.

CDX Systems presented an overview of their DDG 51 SCWE software development process and explained how they were going to use software measurement to manage the progress and quality of the software. The lead CDX software engineer on the program provided a description of the key characteristics of their measurement program:

- The overall measurement program was applied across all software development activities at the CSCI level. Low level software data was collected monthly and entered into their software project management database. For some measures (e.g., lines of code), data was collected down to the level of individual units. Per the development contract, PMO-551 would have direct electronic access to this data.
- For the project, the process for estimating and measuring each software parameter was defined and was consistent with the CDX approach used for past programs. In addition, CDX reported that all of the software development subcontractors agreed to use the same measurement definitions.
- CDX Systems reviewed the DDG 51 SCWE software WBS and showed how the overall measurement structure was aligned with the defined software activities and products. They also reviewed their MIL-STD-498 implementation.
- CDX Systems stressed that the measurement program began with the accurate definition and tracking of both the stated and derived software requirements. They showed how they were going to measure the total number of requirements and how they were going to track the allocation of the requirements to the software architecture.
- CDX completed the discussion by reviewing the overall set of measures they intended to use. The measures themselves were relatively basic, but were implemented within a well defined process at a meaningful level of detail.

3.1.2 Software Issue Identification and Analysis

During the first year, the project proceeded relatively smoothly. The software measurement process was running smoothly and Captain McLain received a monthly issue evaluation from Gary Wilson. The software measurement indicators showed some variance in the monthly actuals relative to the plans, but there were no major deviations. The Preliminary Software Design Review, which addressed the CSCI architectural design, was completed on 15 June 1997, six weeks behind schedule. Considering the DDG 51

SCWE was a six-year development program, this was only a small schedule slip and did not cause much concern.

In June of 1997, the Navy decided that the DDG 51 SCWE functional baseline had to be modified to incorporate a new variant of the surface-launched Tomahawk cruise missile. The functions required to implement this new missile were added to the Build 1 software requirements. Since the new missile was added at the beginning of CSCI detailed design, both PMO-551 and CDX Systems believed that there would not be any major schedule impact from the modification.

During the next two months, a team of software engineers from CDX Systems worked with PMO-551 to analyze and document the additional requirements and to prepare the technical inputs for the Engineering Change Proposal (ECP). The new Tomahawk variant added approximately 550 additional requirements and 62,000 Source Lines of Code (SLOC) to the planning baselines. The resultant changes were allocated to 450 new software units. The majority of these requirements were applicable to the Target Engagement function. The Workstation Control and System Control functions also had minor revisions due to the new missile. These new requirements increased the risk associated with the Target Engagement function, which had already been identified as high risk by both PMO-551 and CDX Systems.

In November of 1997, Gary showed Captain McLain a Build 1 software development progress indicator based on the number of software units completing detailed design (Figure 3.1). The first thing he pointed out was the lag in development progress. The number of units completing detailed design was significantly behind plan. CDX Systems had developed a revised plan that took into account the additional software units which were added because of the new missile functionality. The new plan called for a much higher unit design completion rate than originally projected or had been achieved to date.

Gary had discussed this indicator with CDX Systems. They believed they could meet the higher unit completion rate projected in their revised plan. They based this assumption on the 80 new people they added to the staff over the past few months. CDX Systems indicated they now had sufficient resources available to complete the software development within the projected schedule.

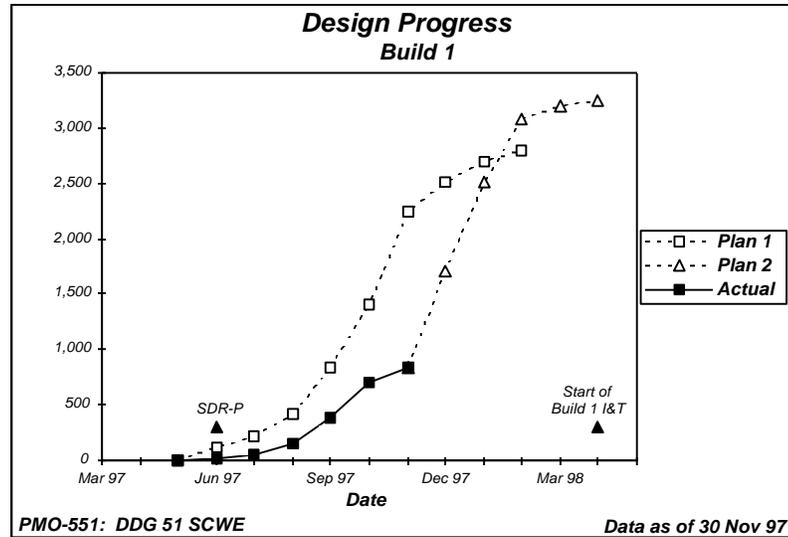


Figure 3-1. Unit Design Progress

In April of 1998, the developer experienced serious and unexpected problems trying to integrate the COTS products into the DDG 51 SCWE software baseline. Specifically, CDX Systems ran into the following difficulties:

- The task of integrating the COTS operating system was considerably more complicated than had been originally anticipated. Performance problems required the design and implementation of a functional software “shell” between the applications software and the COTS operating system. This meant that new requirements and code had to be added to the Workstation Control function.
- Performance problems were discovered while integrating the COTS relational databases in both the workstation and the AN/UYK-43. The critical ship-to-ship data items were not being processed quickly enough. The only solution was to revert back to flat file processing for the critical portions of this data.

With this new set of problems, it was clear that the schedule risk was increasing. In fact, the Build 1 Software Design Review covering CSCI detailed design was delayed for almost three months.

Captain McLain continued to review the summary level indicators on a monthly basis. In August 1998, she decided she wanted to see

some indicators that could localize the problem areas to specific software functions. She directed Gary to take a close look at the current set of indicators to assess project status.

First, Gary constructed a graph, shown in Figure 3.2, showing the growth in requirements over the past two years. The first point, July 1996, represents the number of stated requirements that were defined in the contract proposal. Between the beginning of the contract and June 1997, the number of requirements increased. The majority of this growth occurred during software requirements analysis, as the CDX system and software engineers achieved a better understanding of the system functionality and developed the derived software requirements. The number of requirements increased again between June and August 1997 due to the addition of the new Tomahawk missile functionality. Between August 1997 and August 1998, the number of requirements again increased with the addition of requirements resulting from the problems experienced while integrating the COTS software.

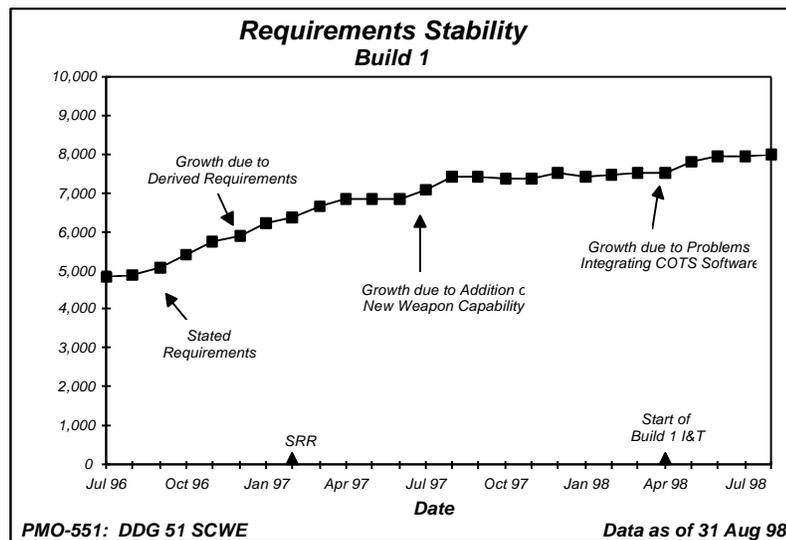


Figure 3-2. Requirements Stability

While it was obvious that the system as a whole experienced significant requirements growth, Gary also looked at the requirements growth for each of the major software functions in the system as shown in Figure 3.3. From this breakdown, it becomes clear that a large portion of the requirements growth was in the workstation functions. Most of the requirements growth related to the new missile occurred in the Target Engagement function. The

growth related to the COTS implementation problems increased the number of requirements in the Workstation Control and System Control functions.

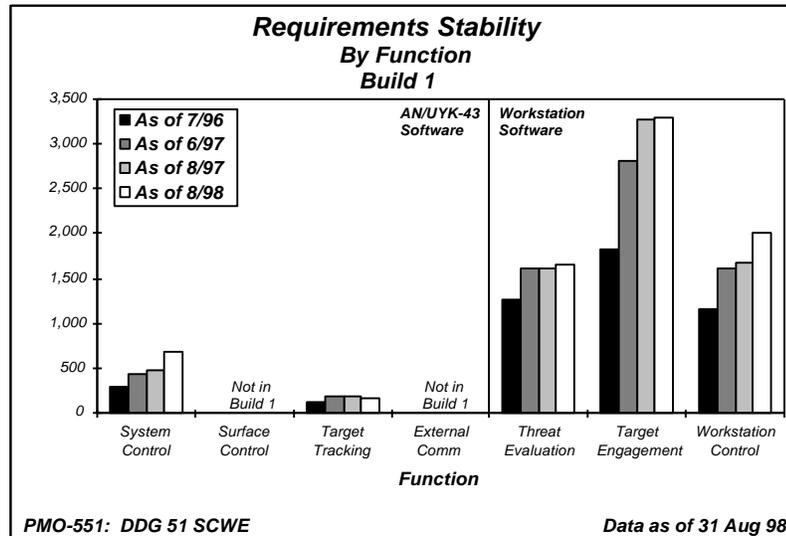


Figure 3-3. Requirements Stability by Software Function

Captain McLain also wanted more information about the growth of requirements and the impact of that growth on product size. She asked Gary to provide a breakdown of size by major function. From the low level data in the PMO-551 database, he constructed a software size estimate by software function indicator as shown in Figure 3.4. The data that was graphed corresponded to the same periods included in the requirements growth by function indicator. It was clear that size growth paralleled the growth in requirements, as was expected.

Captain McLain was also concerned about whether CDX's software development staffing levels were tracking to plan and if the amount of effort being applied to the project was adequate. The next graph Gary showed Captain McLain was the monthly effort data presented in Figure 3.5.

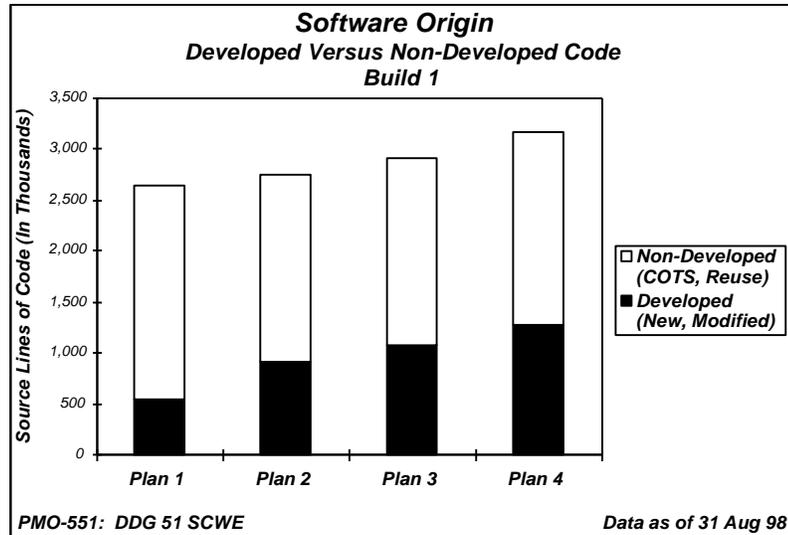


Figure 3-4. Software Size Estimates by Software Origin

This graph showed that although the development was initially understaffed, CDX Systems added additional people to make up for the early deficit. In a subsequent discussion with CDX Systems, Captain McLain was assured there were enough resources to complete the software development.

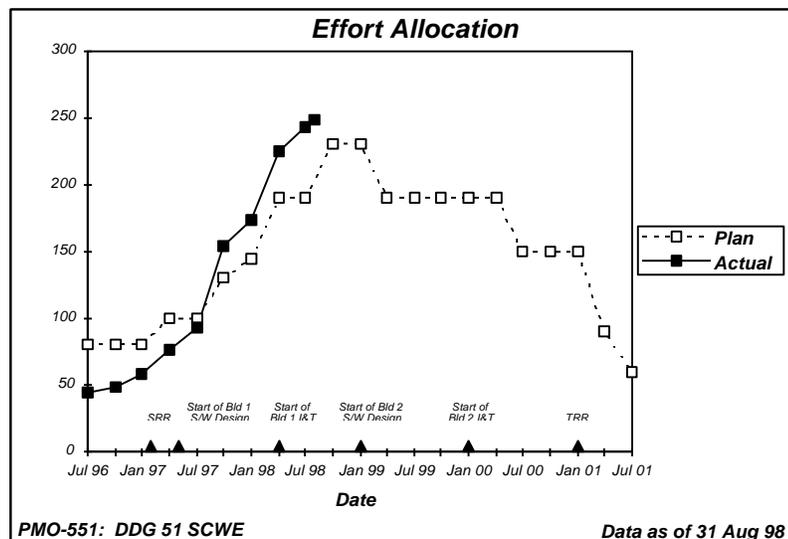


Figure 3-5. Software Effort Allocation

Gary then showed Captain McLain an earlier indicator of software development progress based on the number of units that have completed detailed design as shown in Figure 3.6. From this indicator, it appeared the rate of units completing detailed design

had increased significantly after the initial lag noticed in November of 1997. The data showed that all of the units had completed the detailed design milestone within one month of the revised plan.

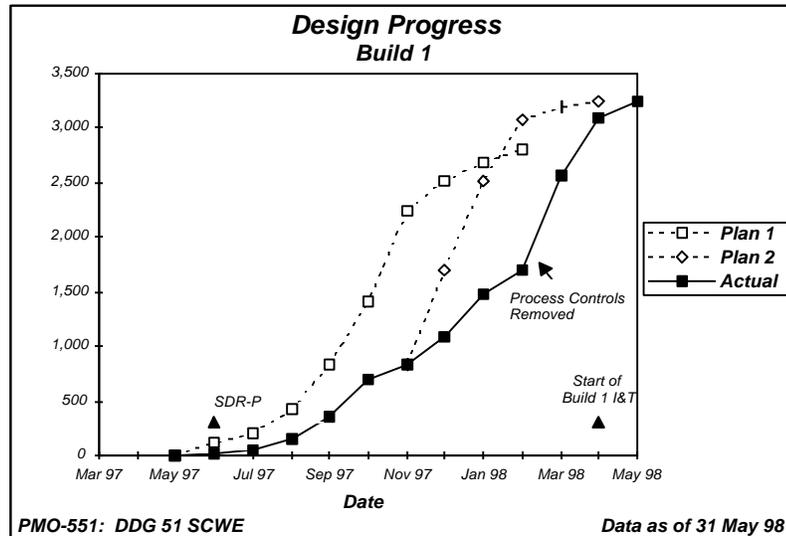


Figure 3-6. Unit Design Progress

While the progress indicator gave Captain McLain some reason for optimism, the problem report data told a different story. Gary showed Captain McLain a summary of the cumulative number of total and closed problem reports which had been collected during integration and test. These are shown in Figure 3.7.

Captain McLain noted that the problem report discovery rate increased rapidly during integration and test. She was disturbed by the fact that problem report discovery appeared to be occurring at a much higher rate than problem report closure. She then asked Gary to show her the problem report data for the individual functions.

Gary calculated problem density by dividing the number of unique valid problem reports by the new and modified source lines of code for each function as shown in Figure 3-8. It was clear that, even when normalized by size, the Target Engagement function was much more problem prone than any of the other functions. Captain McLain asked Gary to find out what was going on with this function.

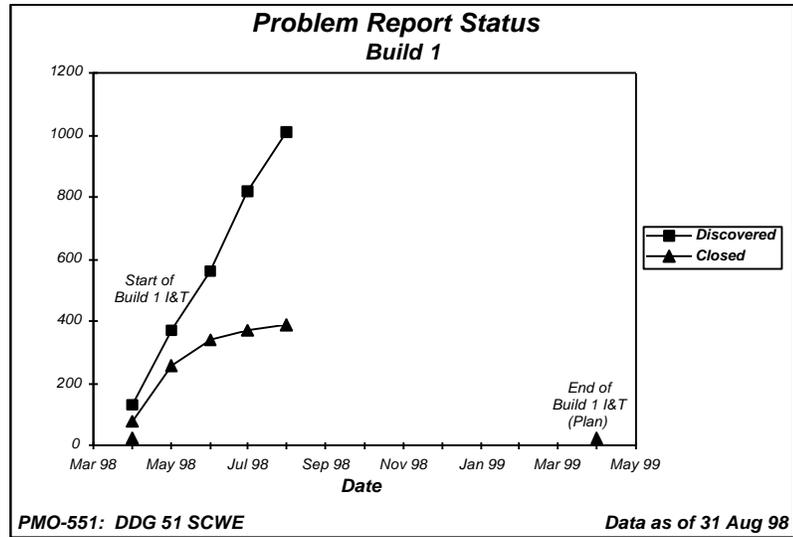


Figure 3-7. Problem Report Status

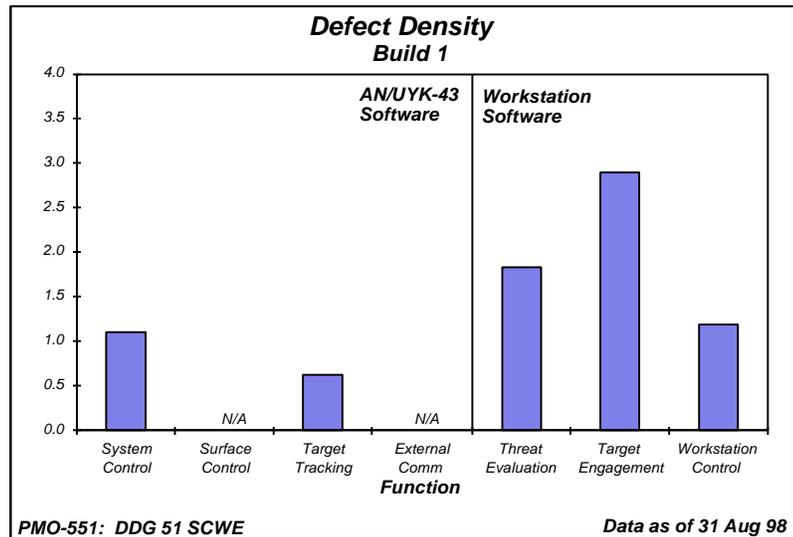


Figure 3-8. Defect Density – Build I

Gary visited CDX Systems and met with the software engineering manager to discuss the workstation CSCI problems. He discovered that as of January 1998, unit design and code inspections had been discontinued in an effort to complete the development activities as quickly as possible. The delays in software development progress had begun to impact the software testing process. Successful completion of the unit design and code inspections had been the primary exit criteria for measuring unit development progress. With this requirement relaxed, the software developers were not required to adhere to a key process activity which ensured that only complete, high quality units were delivered for integration. In

effect, the quality of the software became secondary to meeting the schedule and the measurement indicators had helped to identify the problem.

Most of the software units impacted by this process change belonged to the Target Engagement function. This explained the sudden increase in apparent software development progress based on the number of units completing detailed design. It also explained the large number of problem reports being discovered in integration and test. Defects that should have been found during those inspections were not being discovered until later. In his discussions with CDX Systems personnel, Gary also found out that the majority of the recently added personnel were working on problem corrections and rework. It was clear that, by removing the process controls, CDX had only made the situation worse.

By this time, Captain McLain had serious doubts about the likelihood of completing Build 1 on schedule. In examining the schedule revisions as shown in Figure 3.9, she noted that CDX Systems made a series of periodic minor revisions to the DDG 51 SCWE detailed milestone schedule. No changes to any intermediate milestone were made until it was obvious that the completion date for that milestone would not be met. Even when revisions were required, CDX Systems made only small incremental changes, rather than doing a comprehensive analysis to determine when the activities could realistically be completed. While completion estimates for the detailed milestones had slipped, the completion of integration and testing for Build 1 had not been adjusted. The result was an integration and test schedule that was becoming less and less feasible.

Captain McLain began to realize that she might have to reassess the current DDG 51 SCWE software development plan. Her suspicions were confirmed when she looked at the achieved productivity to date for Build 1. It did not appear that CDX Systems would be able to produce the planned amount of code for Build 1 within the current schedule. After reviewing the analysis results with the CDX Systems Program Manager, Captain McLain decided to replan the software development effort.

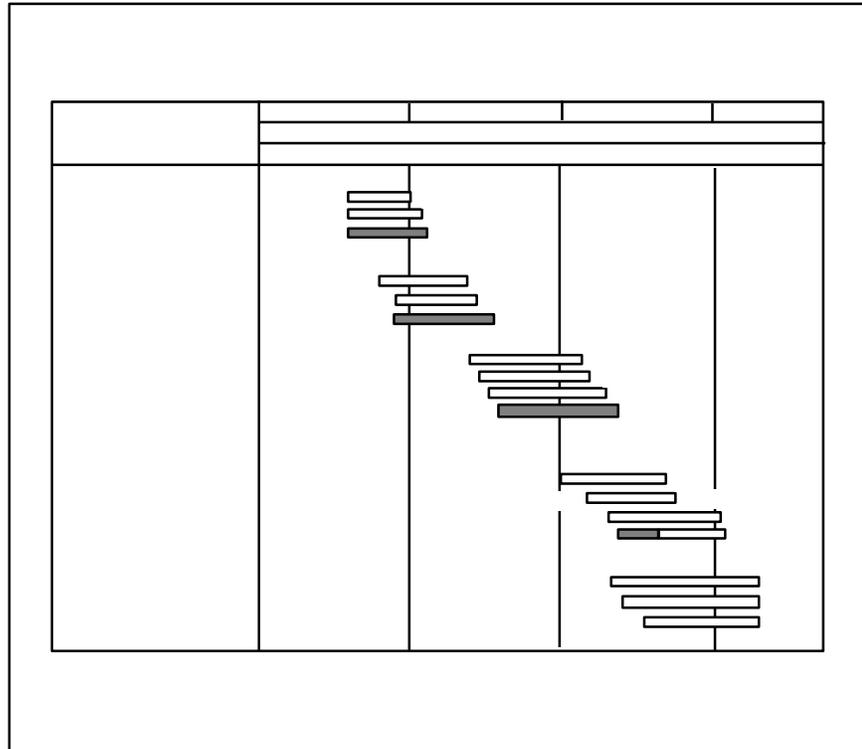


Figure 3-9. Revised Software Development Schedule

3.2 Revising The Development Plan

In October of 1998, Captain McLain met with the PMO-551 staff and with managers from CDX Systems to replan the remainder of the DDG 51 SCWE project. Two options were considered:

- Moving software functionality from Build 1 to Build 2.
- Adding another build, Build 3, and shifting software functionality between the builds. Under this option, Build 1 and Build 2 were revised so that each would contain an equal amount of code, while a smaller amount of code was integrated into Build 3. Most of the code that was shifted to a later build was from the Target Engagement function. This was the highest risk function and had the most problems with respect to development progress and quality. The Threat Evaluation and Workstation Control functions also had a small amount of code shifted between the builds.

Captain McLain asked Gary to evaluate the feasibility of each of these two options as shown in Figure 3-10.

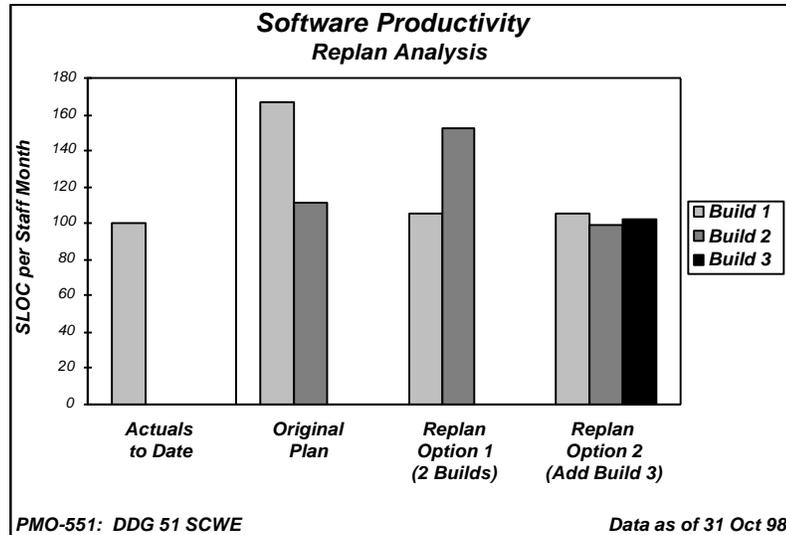


Figure 3-10. Replanned Software Productivity by Build

The first replan was rejected because of its high productivity requirement for Build 2. This option required the productivity of Build 2 to be higher than what had been achieved to date on Build 1. This requirement was assessed to be unrealistic. Implementing this option would most likely have resulted in a second replan later in the development cycle.

The second option was selected because of several favorable elements:

- The required productivity for each remaining build was based on CDX Systems' achieved software productivity to date on DDG 51 SCWE.
- This option supported the original delivery schedule of 1 July 2001, although with reduced functionality. An additional delivery was added for September 2002, 14 months after the original delivery. This additional delivery would include all of the required functionality.
- This option was based on the current staffing resources available to the DDG 51 SCWE program. No additional personnel would be required for this approach. It was believed that adding more people at this point in the development would only delay the delivery further.
- Although the schedule was extended by 14 months and additional funding had to be identified, the revised plan was realistic and contained no major risks.

When PMO-551 presented the replan to the Navy, the measurement analysis results helped to clarify the situation and showed that PMO-551 had an objective understanding of the software development constraints and issues.

As part of the replan, CDX Systems assured the program office that all process controls would be reinstated, including the unit design and code inspections.

3.3 SOFTWARE DELIVERY

After the replan, Captain McLain and Gary continued to monitor the DDG 51 SCWE project. Captain McLain believed that two issues needed to be monitored more closely. First, she wanted to ensure that the requirements were being verified at a sufficient rate to meet the delivery schedule. Secondly, Captain McLain wanted to assess the adequacy of CDX Systems' integration and testing process.

To address the requirements issue, an indicator depicting the number of software requirements that had been successfully verified during integration and test was developed as shown in Figure 3-11. Progress was steady, which told the program office that the planning revisions were effective. CDX Systems was producing the software in accordance with the revised schedule and was projected to meet all delivery requirements.

The quality of the software had also improved. The problem report discovery rate had begun to decrease, and even with the increased test activity, CDX was finding fewer serious problems as shown in Figure 3-12).

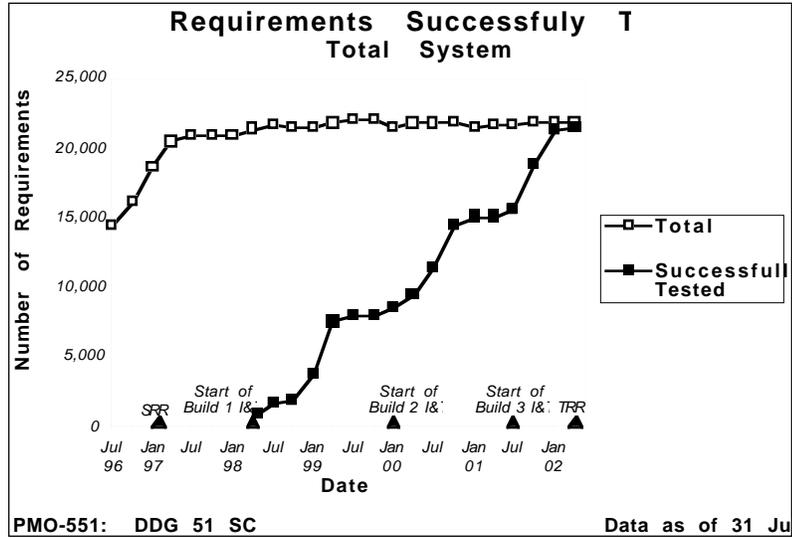


Figure 3-11. Software Requirements Successfully Tested

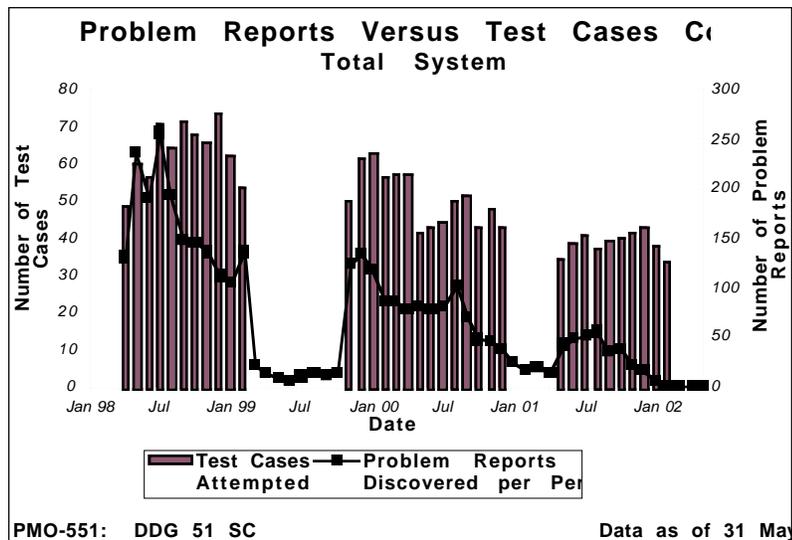


Figure 3-12. Software Problem Reports and Test Cases Completed

3.4 EPILOGUE

Buils 1 and 2 were delivered on schedule. Figure 3-13 shows the actual productivities achieved for those builds along with the productivity observed for Build 3 as of March 2002. Productivity increased across the three builds, contributing to the on-time delivery for each build.

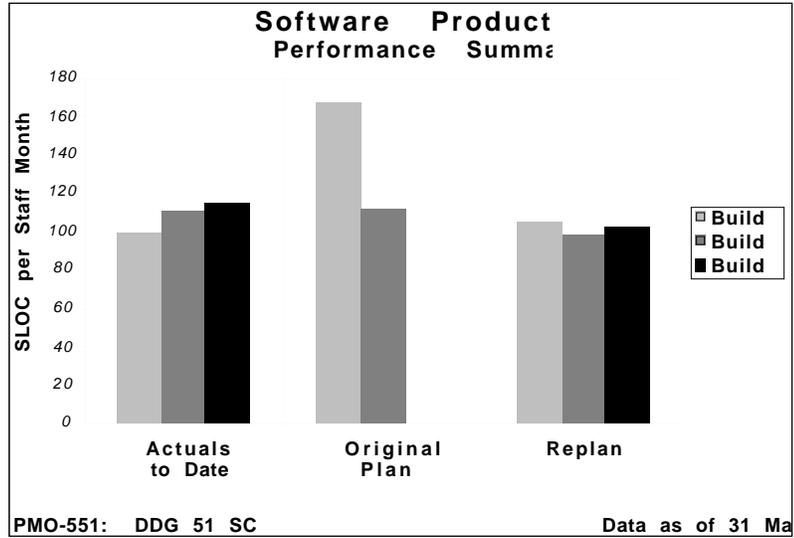
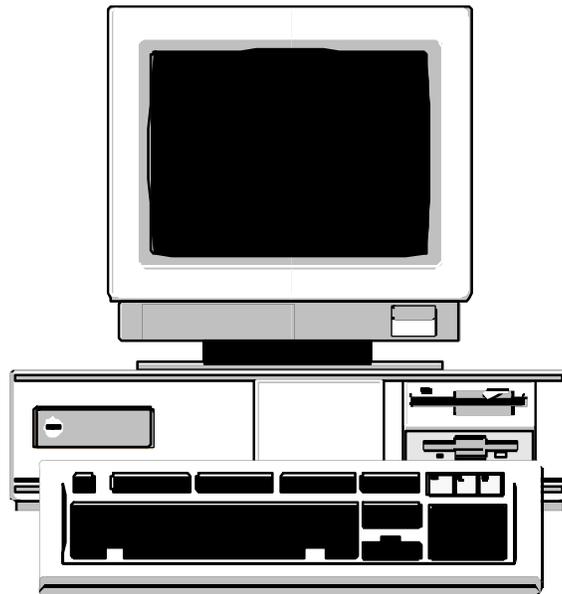


Figure 3-13. Software Productivity Performance Summary

In September of 2002, the PMO-551 / CDX Systems, Inc. team deployed the **DDG 51 Surface Ship Concurrent Weapons Engagement Upgrade** system on the USS John Paul Jones, DDG 53. Although the issues related to the software development were significant, use of software measurement had helped the Program Manager to make objective and informed decisions which led to the program’s ultimate success.



AUTOMATED INFORMATION SYSTEM CASE STUDY

PART 5B

AUTOMATED INFORMATION SYSTEM CASE STUDY

The Automated Information System case study is based on the development of a military personnel information system for the U.S. Air Force. It addresses the use of measurement on a program which has been under development for some time. The program has recently failed a major acquisition milestone review, and measurement is seen as a way to gain some control over the software development effort. The system is being developed by an organic Air Force activity working for a program manager within the same command. The development addresses current DoD initiatives to promote open systems, AIS interoperability, and the use of commercial-off-the-shelf (COTS) software packages. The technical approach includes the use of third and fourth generation languages and conversion of existing data structures. The critical issues are largely driven by external development dependencies. They include the need to meet aggressive development and deployment schedules, and the requirement that the overall readiness of the software for deployment be objectively determined.

The Automated Information System case study is organized into four chapters:

- Chapter 1, Program Overview, describes the technical and management aspects of the software development effort.*
- Chapter 2, Getting the Program Under Control, shows how measurement can be implemented on an existing program to define a realistic software development plan, and subsequently how to track the development against that plan.*
- Chapter 3, Evaluating Readiness for Test, illustrates how measurement helps to objectively determine if the software is ready for operational test and subsequent deployment.*

- *Chapter 4, Installation and Software Support, shows how measurement is used after the system is fielded to identify and correct user problems.*

CHAPTER 1 - PROGRAM OVERVIEW

This chapter introduces the AIS program scenario and describes the technical and management aspects of the development effort. The program scenario is based on the implementation of a measurement process on an existing program. As such, special consideration is given to using software measurement data that is readily available within the established software and program management processes. The example program is representative of a typical AIS system under development to meet DoD business process reengineering objectives.

1.1 INTRODUCTION

Over the past several years, Ridgway Air Force Base in Cheyenne, Wyoming has become established as a primary source for the development of Air Force business information systems. The software development group at Ridgway began as an organic software maintenance organization, and has successfully transitioned its business base from the support of Air Force logistics and maintenance systems to software system reengineering and development. Ridgway has benefited from the recent DoD emphasis on upgrading existing information systems into an integrated set of more manageable, cost-effective resources, and has become an important resource in the Air Force Material Command.

In 1994 the Air Force designated Ridgway Air Force Base as the lead development organization for the Military Automated Personnel System (MAPS). MAPS represented the Air Force's "next generation" military personnel information system. The program was part of a larger initiative to reengineer the Air Force's administrative business processes. The reengineering plan included a service-wide initiatives to modernize information system hardware, software, and communications interfaces at both the base and headquarters levels. Existing mainframes and terminals were to be replaced by client/server architectures, and new capabilities were to be implemented by adapting existing databases and integrating them with newly developed applications software. MAPS was an

important link in business system modernization effort, since it was the first part of the overall system to be developed and delivered. MAPS was scheduled to be deployed at a number of Air Force bases during 1997. Needless to say, MAPS was an important, and highly visible program.

In 1995, MAPS had been under development for three years. During that time, the Ridgway software development group tried to keep current with changing DoD acquisition policy and related software initiatives. These included the definition of open systems architectures, the integration of Commercial Off the Shelf (COTS) software components, the use of advanced third and fourth generation programming languages, and an overall restructuring of the development organization using Integrated Product Teams (IPT).

In November of 1995, a new program manager was assigned to the MAPS program. Air Force Lt. Col. Barry Thompson was a 1978 graduate of the Air Force Academy. His background included four years with the Air Force's Operational Test & Evaluation Center and eight years in various Air Force system program offices. His last assignment was as the Deputy Program Manager for a major upgrade to an Air Force maintenance data system.

Lt. Col. Thompson's assignment to the MAPS program did not come under the best of circumstances. At the time of Lt. Col. Thompson's arrival, MAPS had just undergone an unsuccessful review by the DoD's oversight committee for major AIS systems, the Major Automated Information Systems Review Council (MAISRC). MAPS had failed to receive a Milestone III approval for system production and deployment from the MAISRC. This was largely a result of problems with the software, especially with respect to the amount of completed functionality and the overall quality of the existing code. The MAISRC report indicated that there was little confidence in the cost and schedule estimates presented by the previous program manager in an effort to substantiate his development plan. There was also a lack of available data which showed the MAISRC how the program manager was addressing the key MAPS software development issues.

Lt. Col. Thompson arrived at Ridgway with clear direction to get the project under control and to establish an objective, credible plan

for the remainder of the development. Lt. Col. Thompson's first task was to review the overall technical and management characteristics of the program. He wanted to identify the events and decisions which had helped to shape the program in order to identify the key software issues and problems that he needed to address.

1.2 AIR FORCE BUSINESS PROCESS MODERNIZATION INITIATIVE

In reviewing the MAPS program history with the Ridgway development team, Lt. Col. Thompson learned exactly how MAPS fit into the Air Force Business Process Modernization Initiative. The MAPS program was intended to reengineer the existing military personnel information system currently in use throughout the Air Force. MAPS was the first application to be developed. Subsequent applications which were to be integrated as part of the initiative included revised supply, finance and accounting, medical, payroll, and base-level maintenance functions. The scope of the initiative was significant. In addition to the upgrade of the base level business functions, the new applications were required to support a seamless interface at the headquarters level. As such, almost all of the key Air Force AIS systems would be impacted in one way or another.

Lt. Col. Thompson noted several key features of the Air Force Business Process Modernization Initiative:

- **Client/Server Architecture**- The existing mainframe computers and associated video terminals were to be replaced by client/server architectures at each base and at each command headquarters.
- **Open Systems**- The current dependence on vendor-specific, proprietary operating systems and database management systems was to be replaced by open system standards-based architectures. A POSIX compliant operating system had been selected as part of the software architecture for MAPS and the other Air Force AIS systems which were to be reengineered.
- **Standard Data Elements**- The efficient flow of data from one DoD information system to another was an important objective of the initiative. In order to achieve a high level of interoperability, the revised Air Force systems, including MAPS, had to adhere to a standard

set of data definitions. Control of the data standardization effort was the responsibility of the Defense Information Systems Agency (DISA).

- **Process Modeling**- All of the business processes which fell under the modernization initiative were required to be modeled using the CAM definition language (IDEF). This modeling effort was important to ensure the efficiency and interoperability of the various information systems which would be reengineered as part of the initiative.
- **Integrated Databases** - An important aspect of the modernization initiative was the intent to move away from “stove-piped” business applications, each with its own database and unique application characteristics. MAPS, therefore, had to include an integrated database which could be accessed by the various user applications using a common data interface. The intent was for any given data element to be entered only once at the point of origination. The data would then be made available to other applications. Development and control of the logical and physical data models rested with the Air Force, and again the MAPS design had to comply with higher level requirements.
- **Maximum use of COTS Software Components**- The use of commercial software packages was strongly encouraged. As part of the modernization initiative, special waivers had to be obtained to develop unique software applications if a commercial counterpart which met the defined requirements was available.
- **TAFIM** - All of the revised AIS systems which comprised the modernization initiative, including MAPS, were required to be designed and implemented in accordance with the DoD’s Technical Architecture Framework for Information Management (TAFIM).

1.3 PROGRAM DESCRIPTION

Lt. Col. Thompson's staff briefed him on the key project events and the technical and design characteristics of the MAPS program. MAPS began in the summer of 1994. It had been under development since that time by the Air Force's Administrative Systems Development Activity at Ridgway Air Force Base in Cheyenne, Wyoming. All of the personnel involved in the MAPS development effort were organic to the Activity. That is, they were

either civilian or military personnel directly employed by the Air Force. The system and software requirements, and high-level design were defined during the first year of the MAPS development. In November of 1995, a briefing was given to the DoD MAISRC oversight group to support a Milestone III decision. Serious concerns were voiced by the members of the group during the briefing. The major issues focused on the development of the MAPS software and included the following

- The original software development schedule had been slipping on an incremental basis. The revised “get well” schedule presented by the previous program manager appeared to be unrealistic, and could not be substantiated based upon the development performance to date.
- Similar to the schedule issue, there was no credible basis for the cost projections presented to the MAISRC. It appeared to the MAISRC that the cost of the software was being driven by the number of development personnel available, not by the size and capability of the software which had to be developed.

The original MAPS development plan called for two incremental deliveries of the required capability. When Lt. Col. Thompson arrived at Ridgway, the software for the first incremental release was under development.

MAPS began under a tailored MIL-STD-7935A software process and had begun to transition to MIL-STD-498. The software development languages included both Ada 95 and C. Development tools included a state of the art Ada programming support environment, a screen generator, and a report generator. A COTS relational database was also an integral part of the design

The MAPS software design included twenty-four functionally defined Computer Software Configuration Items (CSCIs). Thirteen of these were allocated to Increment 1 of the development and nine were allocated to the second increment. The remaining two CSCIs were data conversion software. For each of these CSCIs, access to the database was to be implemented using SQL. User access and interface was designed to be implemented using predefined, “user friendly” screens. Site operators had additional access using SQL. The user interface was to be developed using X-Windows and was designed to be MOTIF compliant.

1.4 SYSTEM ARCHITECTURE AND FUNCTIONALITY

The primary objective of the MAPS program was to reengineer the existing Air Force military personnel information system to add new functionality and to meet the overall integrated system requirements defined by the Business Process Modernization Initiative. To fully understand the technical implications of migrating the existing system to the new design, Lt. Col. Thompson compared the architecture and functionality of the current military personnel system with the MAPS requirements and specifications.

1.4.1 Current Personnel System

Figure 1-1 shows the hardware architecture for the current personnel system. The current system is, in reality, two separate AIS systems. One resides at the base level and the other at command headquarters. Both the base level and the headquarters implementations were based on the use of mainframe computers and video terminals. The applications for both system levels were written in COBOL, and included hierarchical databases. Both incorporated character-oriented non-graphical user interfaces.

The operating concept of the current system included periodic data transactions from the base-level systems to the headquarters level system. Selected data was uploaded to headquarters every 24 hours. As with many legacy information systems, the current military personnel implementation had experienced a significant number of problems with respect to inconsistent edits between the two systems. Part of this was attributable to the base level system requiring very loose edits, while the edits for the headquarters system were much more constrained. Consequently, there was a very large rejection rate for data which was uploaded to the headquarters system. As such, data was often lost in the transaction process.

To access data at the base level from the headquarters database, users had to log in and connect the systems over standard phone lines. This interface approach had proven to be unreliable and added to the problems associated with transferring data.

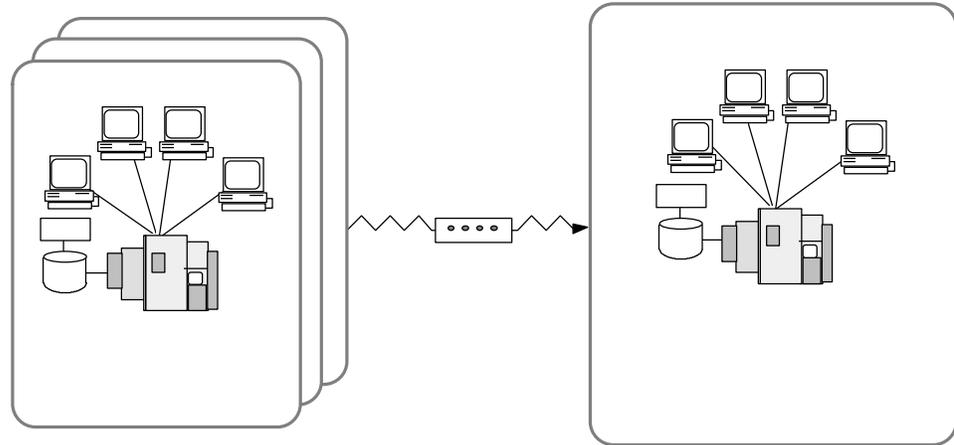


Figure 1-1. Current System Architecture

1.4.2 Military Automated Personnel System (MAPS)

The hardware architecture for MAPS is shown in Figure 1-2. MAPS is designed as a single integrated personnel system which incorporates real time data updates and access between the base and headquarters level system implementations. The headquarters portion of the system incorporates a mainframe computer which is used only for data storage. It is part of the headquarters local area network (LAN). MAPS incorporates a client/server design at both the base and headquarters levels. Data transfer between the levels is provided by a designated MILNET interface.

The MAPS client/server architecture integrates Graphical User Interface (GUI) and display functions on individual PCs, while the shared application functions reside on a UNIX based server. This design is applicable at both the base and headquarters levels.

When MAPS is initially fielded at each Air Force base, it will be required to interface with the existing base-level AIS systems. These systems will gradually disappear as the Business Process Modernization Initiative progresses. As each existing AIS system is reengineered and integrated into the overall information system structure, all base-level applications will transition to a common enterprise architecture with access to a common database. As with MAPS, all interaction between applications will then occur through the shared database.

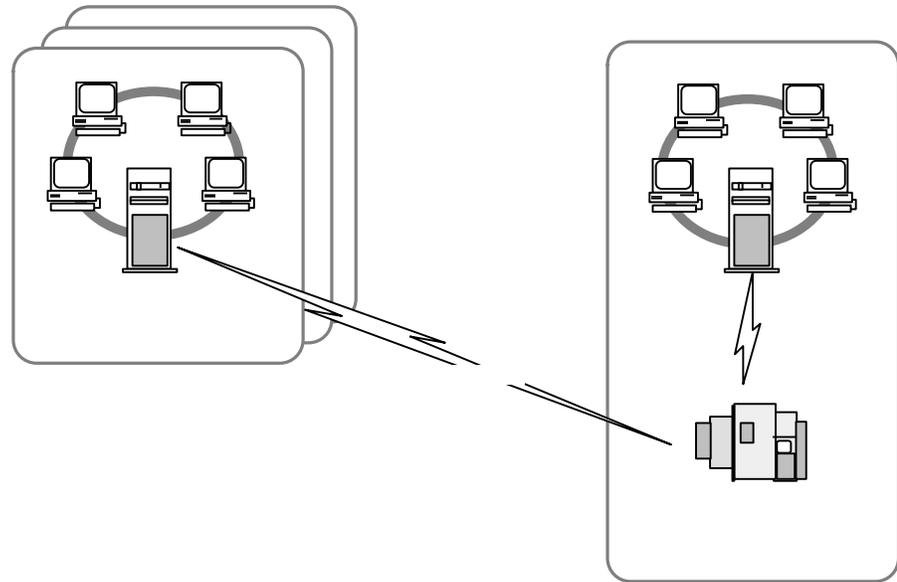


Figure 1-2. MAPS Architecture

The MAPS design incorporates two functional subsystems. As expected, these include the Base-level functional subsystem and the Headquarters functional subsystem. The Base-level subsystem includes those standard functions that support the military personnel assigned to individual bases, or to commands, such as individual aircraft squadrons, which are resident on base. The type of personnel data which must be available from MAPS at the base level includes individual information on each officer and enlisted person assigned at the base. This data includes age, rank, skill level, training history, individual personnel assignment and promotion history, and information pertinent to past performance evaluations. The Base-level MAPS subsystem also contains personnel information at the command level, such as squadron mobilization personnel requirements, casualty data, skill profiles, and personnel replacement priority information.

The MAPS Headquarters subsystem includes military personnel functions that generally support higher level information requirements than those needed at the base level. The Headquarters subsystem provides information which supports overall force mobilization, strategic planning, and analysis of force manpower requirements. For example, if a senior Air Force commander wants to deploy an offensive air superiority fighter such as the F15-E, the Headquarters subsystem can provide information about the location of each F15-E squadron, and the availability and

training history of the pilots, maintenance personnel and other support crew. If the Air Force needed to plan for night time air sorties into mountainous terrain, MAPS would help identify those squadrons with the appropriate qualifications

The overall MAPS development plan called for the subsystems to be developed and delivered in separate increments. The Base-level functions would comprise Increment 1 and the Headquarters functions Increment 2. In addition to development of the respective increment functionality, MAPS required that the data from the current military personnel information system be converted and entered into the redesigned MAPS data structures. As such, the MAPS software development effort included the development of data conversion software for both the base-level and the headquarters-level databases.

CHAPTER 2 - GETTING THE PROGRAM UNDER CONTROL

After his review of the MAPS development effort, Lt. Col. Thompson knew that he had a pretty big challenge in front of him. A detailed review of the software development and management processes revealed that the program was essentially being run with milestone schedules and viewgraphs. By mid-1995, the software development schedule milestones had begun to slip on a regular basis. Although this was evident in the milestone charts, there was no action being taken to identify and correct the underlying causes. An analysis of the problem report data in the configuration management database showed that many more software problem reports were being opened than were being closed. All of the available personnel, as it was explained to him, were assigned to implementing and testing the code to meet the defined schedule for Increment 1. There wasn't really enough time to keep up with the problem fixes at this stage of the development.

To gain control over the MAPS software, Lt. Col. Thompson had to address two key issues. The primary issue was software development Schedule and Progress. Lt. Col. Thompson had to assess the feasibility of the current schedule, and determine why performance against the schedule was lagging. Second, he had to address the overall Product Quality of the developed software products. Based upon past experience, Lt. Col. Thompson had a pretty good idea that the software defects represented in the open problem report backlog had a lot to do with the schedule issue. Given the increased visibility of the program after the results of the MAISRC review, Lt. Col. Thompson knew that the system had to work correctly when it was initially fielded.

2.1 EVALUATING THE SOFTWARE DEVELOPMENT PLAN

When Lt. Col. Thompson reviewed the MAPS development plan, he tried to identify how the original schedules and staffing requirements were established. The most detailed schedule information that was available was in the form of Gantt charts showing major project milestones and dates. There was little detail with respect to the low level MAPS software development activities

and associated CSCI development tasks. There was a project Work Breakdown Structure (WBS), but it seemed to apply only loosely to the current tasks. It appeared that the overall development schedule was driven by the required delivery date of the system. Key development activities were scheduled very optimistically to meet the delivery date.

There was no MAPS staffing plan that allocated personnel resources to specific software development tasks. A total of 40 software personnel were assigned full time to the MAPS program. All were available through the planned delivery date for Increment 2. The people were being applied to the program on a level of effort basis.

By this time it was clear to Lt. Col. Thompson that in order to manage the critical software issues he needed better and more detailed information. To help him get the information, he assigned one of the members of his program staff, Jennifer Cooper, as the MAPS software measurement analyst. Jennifer was familiar with implementing a measurement process from her experience on past programs, but this would be the first time she had to tailor and apply measurement for an existing program. Jennifer met with Lt. Col. Thompson to identify and prioritize the major software issues to be addressed by the measurement effort. From the discussion it was clear that Lt. Col. Thompson would give the measurement activities a high priority, and that he intended to use the measurement results to not only help to get the program back on track, but also to show senior management how the program was progressing.

Lt. Col. Thompson and Jennifer Cooper discussed the problems related to implementing measurement on an existing program, especially one which was in trouble. Although all of the measurement data that they wanted would not be immediately available, they felt that they had enough basic information to start to address the key issues. They both decided that it would be a good idea to review the software measurement results on a weekly basis.

The first question Lt. Col. Thompson had to answer was whether or not the original MAPS software schedule was realistic, given the projected level of staffing and the overall performance of the development team to date.

Lt. Col. Thompson asked Jennifer to generate an independent schedule estimate based upon the size of the software product and the expected software productivity. Although this sounded like a straightforward request, Jennifer understood that the characteristics of the program required two separate sets of analysis. There were two different “types” of software development taking place, each described by distinct development approaches. These included:

- Development of the application software for both incremental deliveries. This development effort was based on the use of advanced development techniques and 4th generation languages.
- Development of the data conversion software. This development effort could best be described as a “typical” support software development effort using a high order language with minimal process requirements.

When the system is ready for delivery there will actually be a third “type” of software development, the conversion of the databases and the installation of the system at each base (Increment 1) and at command headquarters (Increment 2). At this point, however, this was not a major concern.

Jennifer needed to estimate the size of the software to be developed in order to project the MAPS development schedule. She decided to use function points as the basic size measure for the Increment 1 and 2 application software. Function points, although somewhat harder to estimate and measure than lines of code, seemed to be a better choice due to the mix of third and fourth-generation languages (4GL) on this part of the development. Since the developers were also using a screen generator tool to develop the user interface screens, the used of lines of code or another product size measure would have been difficult. Jennifer used two methods to calculate the required productivity figures. In addition to a simple functional size to effort ratio, Jennifer used a software cost model that accepted function points as a data input. The model also took into account the productivity impact of language type and reused code.

For the data conversion software, Jennifer decided to use lines of code to estimate the size of the software. In this case, lines of code were a better choice because the software was being written entirely in C.

Jennifer spent several weeks with the development team to arrive at the function point counts and the lines of code estimates. The function point counts were based upon the methodology defined in the Function Point Counting Practices Manual, Release 4.0, from the International Function Point Users Group (IFPUG). The lines of code estimates were based on the number of logical statements and excluded comments. Jennifer summarized the sizing results for Lt. Col. Thompson on the table shown in Figure 2-1.

The information showed the size for each of the CSCIs in Increments 1 and 2. The table also showed the primary language and the projected number of low-level design components or units.

The relational database and the Ada to SQL bindings inherent in the MAPS design were relatively new COTS software products. Input screens and reports were being generated by 4GLs.

Jennifer's projections indicated the following:

- The minimum schedule to develop both functional increments is four months longer than the current development schedule.
- In order to meet even the extended schedule, the MAPS development staffing levels would have to be significantly increased.

Although these analysis results were expected, they indicated that Lt. Col. Thompson would have to replan the remainder of the MAPS program to define a more realistic development plan.

2.2 REVISING THE SOFTWARE DEVELOPMENT PLAN

Lt. Col. Thompson used the cost model estimates as the basis for a revised software development plan. He asked Jennifer to show the new schedule in the form of a Gantt chart. This revised schedule is shown in Figure 2-2.

The revised schedule began with the completed activities. The system requirements and high-level design activities were ongoing from July 1994 through May 1995.

| Software Size Estimates | | | | |
|--|--------------|-----------------|-----------------------------|-------------------------------|
| CSCI | Abbr. | Language | Number of Units | Size (Function Points) |
| Increment 1 - Base Level Functions | | | | |
| 1. Personnel Information | BPI | Ada | 58 | 429 |
| 2. Assignments | BAS | Ada | 36 | 227 |
| 3. Availability (TDY, etc.) | BAV | Ada | 12 | 71 |
| 4. Unit Training | BUT | Ada | 20 | 114 |
| 5. Unit Skills Inventory | BUS | Ada | 34 | 223 |
| 6. Security Clearances | BSC | Ada | 15 | 138 |
| 7. Performance Evaluations | BPE | Ada | 41 | 252 |
| 8. Promotions | BPR | Ada | 37 | 154 |
| 9. Unit Mobilization | BUM | Ada | 51 | 390 |
| 10. Unit Reenlistments | BUR | Ada | 17 | 92 |
| 11. Casualty Reporting | BCR | Ada | 23 | 109 |
| 12. Unit Replacement Priorities | BUP | Ada | 27 | 147 |
| 13. Personnel Database (Base level entities) | BPD | | | 450 |
| Increment 1 Total | | | 371 | 2,796 |
| Increment 2 - HQ Functions | | | | |
| 1. Organization Master | HOM | Ada | 33 | 189 |
| 2. Force Training | HFT | Ada | 28 | 141 |
| 3. Force Skills | HFS | Ada | 22 | 123 |
| 4. Manpower Requirements | HMP | Ada | 55 | 375 |
| 5. Manpower Authorization | HMA | Ada | 21 | 115 |
| 6. Force Replacement Priorities | HFP | Ada | 30 | 170 |
| 7. Strategic Planning | HSP | Ada | 47 | 320 |
| 8. Force Mobilization | HFM | Ada | 65 | 392 |
| 9. Personnel Database (HQ-level entities) | HPD | | | 210 |
| Increment 2 Total | | | 301 | 2,035 |
| CSCI | Abbr. | Language | Number of Units | Size (SLOC) |
| Data Conversion Programs | | | | |
| 1. Base-level | BDC | C | 10 | 9,500 |
| 2. HQ-level | HDC | C | 7 | 6,000 |
| Conversion Total | | | 17 | 15,500 |
| Ridgway AFB: MAPS | | | Data as of 31 Dec 95 | |

Figure 2-1. Software Size Estimates

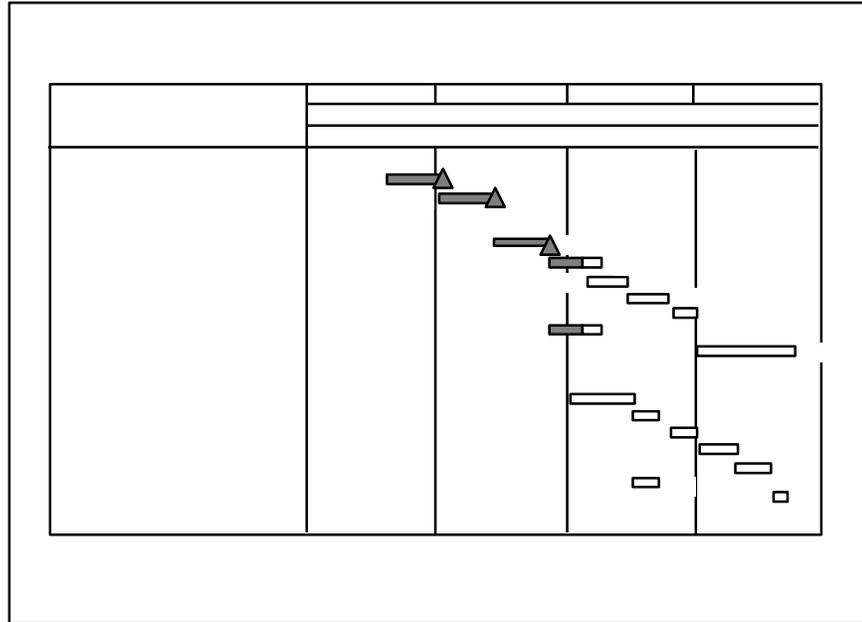


Figure 2-2. Software Development Schedule

Top level requirements and design were completed early in the development effort for the entire system. With these activities complete, the revised schedule called for the independent development of the application software in two parallel increments as previously defined. The development of each increment included detailed design, coding, and integration and test.

The detailed design for Increment 1 was completed in November of 1995. Increment 1 was to be fielded by the end of 1996. Detailed design for Increment 2 was scheduled to begin in early 1996. Increment 2 was scheduled for delivery in mid 1997. The data conversion software was scheduled to be developed in parallel with the respective functional increments. Data conversion and installation was scheduled to occur over a ten-month period for Increment 1 and a one-month period for Increment 2.

Lt. Col. Thompson identified two major development activities on the critical path. These were the "Personnel Information" CSCI for the Base-level subsystem and the data conversion software for both functional increments. The "Personnel Information" CSCI was critical because it has to be completed before the other CSCIs could be integrated and tested. The data conversion software was critical because it was needed to convert the databases at each base and at

headquarters. The data conversion software had to be completed, and had to work properly, before the MAPS increments could be fielded. Lt. Col. Thompson decided to track these critical-path items closely.

The results of the productivity analysis were also used as the basis for the revised MAPS staffing plan. The projected effort allocations for Increment 1 and Increment 2, were graphed as shown in Figure 2-3. When Lt. Col. Thompson reviewed the incremental effort allocation, he noted that the peak full time staffing requirement did not exceed 35 people. Since the schedule called for the MAPS increments to be developed in parallel, Lt. Col. Thompson asked Jennifer to generate a system level effort allocation graph. This graph is depicted in Figure 2-4.

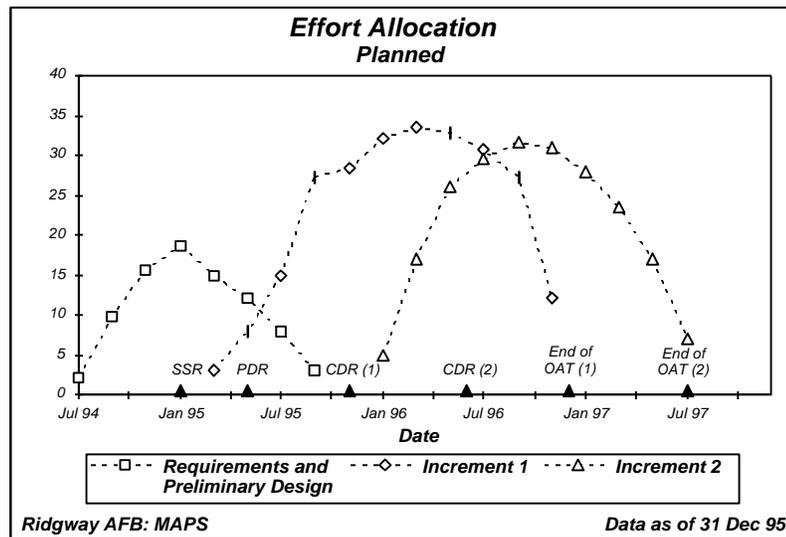


Figure 2-3. Planned Effort Allocation by Increment

When Lt. Col. Thompson looked at the total system effort profile, which aggregated the individual effort requirements, several things became apparent. It was clear that the number of people currently assigned to the development team was not adequate to meet the peak staffing requirements which would occur in 1996. Even more important, the level staffing profile of 40 people did not meet the needs of the program. The development had been inefficiently overstaffed through 1995, and was then projected to experience shortfalls as both Increments 1 and 2 were under development in 1996.

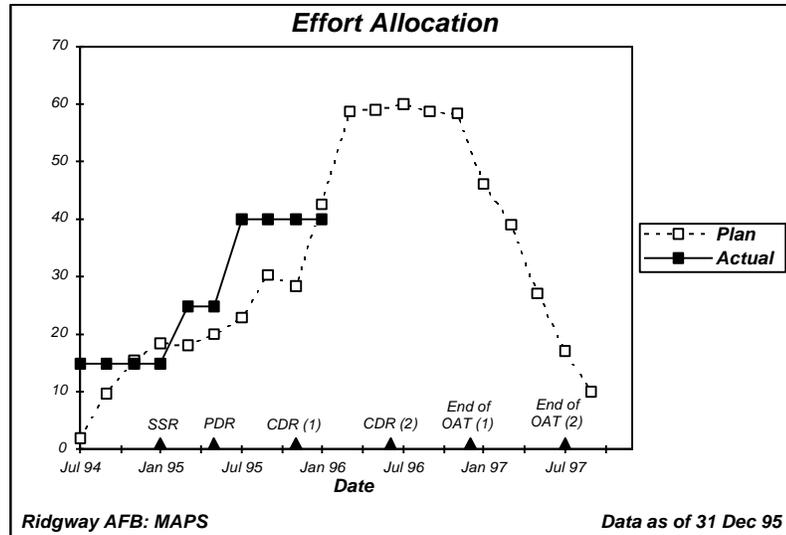


Figure 2-4. System Level Effort Allocation

Lt. Col. Thompson used the measurement results to brief senior management about some of the issues impacting the development of the MAPS software. They agreed with his overall assessment and agreed to add four months to the current development schedule. They also agreed to allocate additional funding to support the 1996 staffing requirements. The plan was to use qualified Air Force personnel from other projects, and to hire outside contractors to help with detailed design, coding, and software integration and test for the MAPS Increment 2 development

2.3 TRACKING PERFORMANCE AGAINST THE REVISED PLAN

Once the new schedule and staffing plans were in place, Lt. Col. Thompson's concerns shifted from evaluating the feasibility of the plans to assessing performance against the plans. Although the milestone data continued to be useful in addressing the schedule and progress issues, more detailed information was required to track the degree of completion of the key development activities and products. The need for this information was clear as Lt. Col. Thompson reviewed the information in the Gantt chart which represented the revised program schedule (Figure 2-2 refers). The milestone schedule indicated that detailed design for Increment 1 had been completed and software implementation was well underway. Based on the schedule, about two-thirds of the time allocated for coding had already elapsed. This didn't mean

however, that two-thirds of the Increment 1 software had been coded. To get the information about the degree of activity and product completion that they needed, Lt. Col. Thompson and Jennifer Cooper decided to implement several work unit progress measures.

Work unit progress measures compare the actual completion of associated work units for software products and activities against a pre-established plan. If objective completion criteria for each type of work unit are defined and adhered to, work unit progress measures provide for a clear determination of software development progress. For each of the MAPS CSCI's, Jennifer recommended that the program use counts of the number of design units completed as the work unit progress measure. The design units represented the lowest practical level of measurement, and the data could easily be collected from the configuration management system. In this case, a "completed" design unit was defined as passing unit test and being entered into the program library.

To generate the CSCI work unit progress indicators, Jennifer first defined the planned rate of unit completion. Without detailed planning data available, Jennifer generated a straight line completion plan beginning with CDR and ending with the scheduled completion of the Increment 1 coding activity. In Jennifer's previous experience with work unit progress measures, she had found that the more accurate plans often looked more like an S-shaped curve than a straight line. This was due to the fact that the first few units tended to be completed slowly, followed by a faster rate of completion rate as the activity progressed. Nearing the end of the software activity, the completion rates tended to slow again as the more difficult units tended to be completed last. For the MAPS work unit progress measures, the straight line plan was not perfect, but was seen as a useful approximation. Everyone understood that they would not be too alarmed if progress lagged behind the straight-line plan at the beginning of the development activity.

Once Jennifer had established the plan, she accessed the configuration management library to obtain a count of units completed to date. Specifically, she counted the number of units that had been entered into the library each week over the course of Increment 1 implementation. The resulting graph is shown in

Figure 2-5. The graph indicated that the CSCI implementation was progressing in accordance with the revised development plan

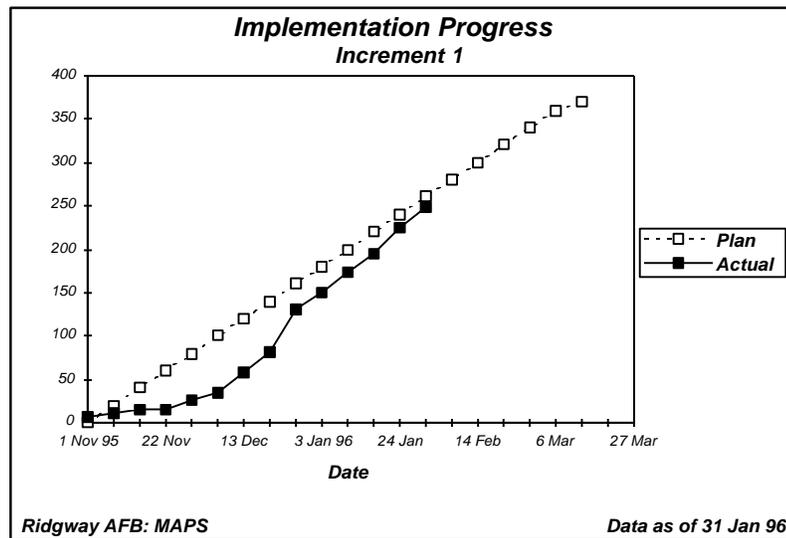


Figure 2-5. Unit Implementation Progress

Jennifer knew that Lt. Col. Thompson wanted to emphasize software measures related to the schedule and progress issue. As such, she decided to track progress for the two items on the critical path very closely. These were the development of the Personnel Information CSCI and development of the data conversion software. The Personnel Information CSCI was scheduled to be completed by March, 1996. Jennifer constructed a plan to track work unit progress for the single CSCI the same way she did it for the aggregate of the CSCIs in Increment 1. Again, the plan was derived by drawing a straight line between CDR and the scheduled end of the coding activity. The resulting indicator was graphed and is depicted in Figure 2-6. When the actual number of design units were compared to the plan, it became immediately clear that progress on this critical CSCI was lagging significantly.

Jennifer then decided to try and identify the source of the progress problem in the Personnel Information CSCI. She defined two new work unit progress indicators using a somewhat different perspective. She graphed the development progress data for the screens and reports separately from the units which performed internal processing. The screens and reports were being implemented using a 4GL while the internal processing code was being written in Ada.

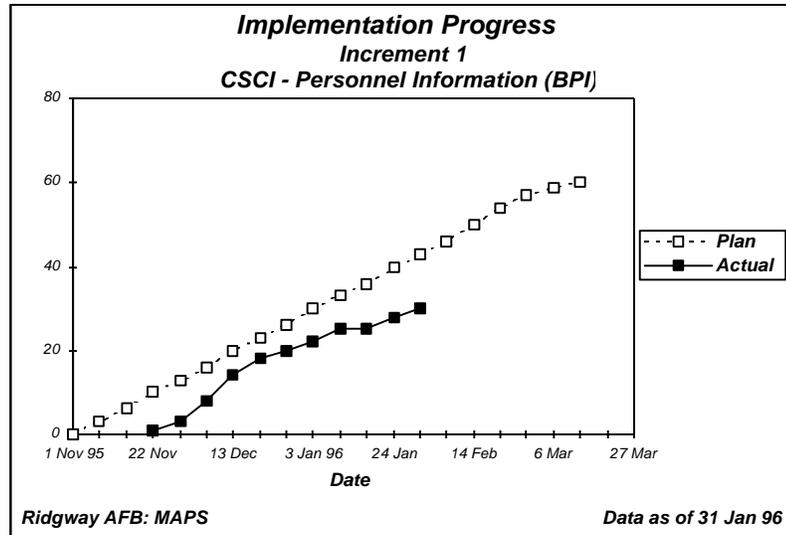


Figure 2-6. BP1 Unit Implementation Progress

The results are shown respectively in Figures 2-7 and 2-8. The measurement data showed that the screen and report development was on track and indicated that the problem was confined to the Ada code. When Lt. Col. Thompson investigated, he found out that the Ada developers were having difficulty with interfacing their respective CSCIs to the COTS relational database. The problem was not critical from a technical perspective, but the workarounds were taking quite a bit of time to implement using SQL. Lt. Col. Thompson did several things to correct the interface problems. The first thing that he did was to bring in representatives from the COTS vendors to work on-site with the Ada developers to provide real-time support in resolving interface problems. Secondly, he had the development team conduct a one-time in-depth inspection of the CSCI's design and completed coded. This inspection identified some design structures which were inefficient, but which could be corrected. Col. Thompson also assigned several of his most experienced Ada programmers to work on the Personnel Information CSCI in an attempt to correct the problem

The other portion of the Increment-1 work that was on the critical path was the data conversion software for the base-level databases. In tracking work unit progress for this software, Jennifer decided to count of lines of code that had been entered into the configuration management library rather than counting the number of completed units.

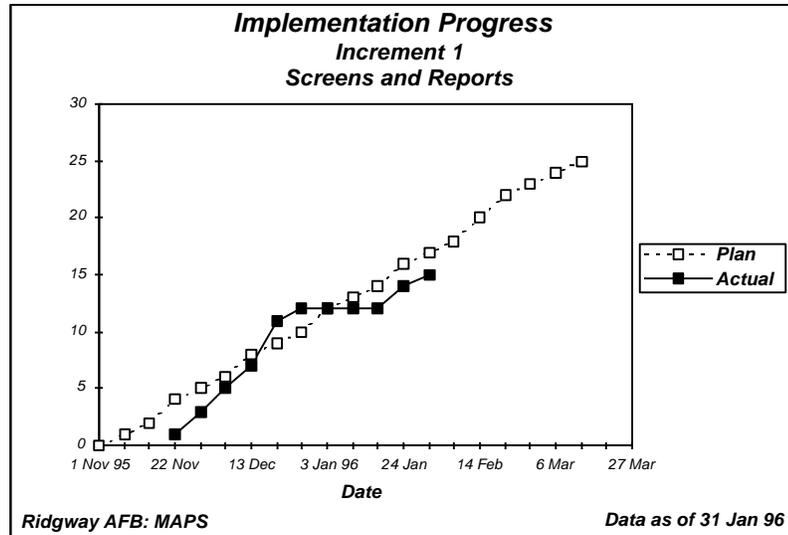


Figure 2-7. Screens and Reports Implemented

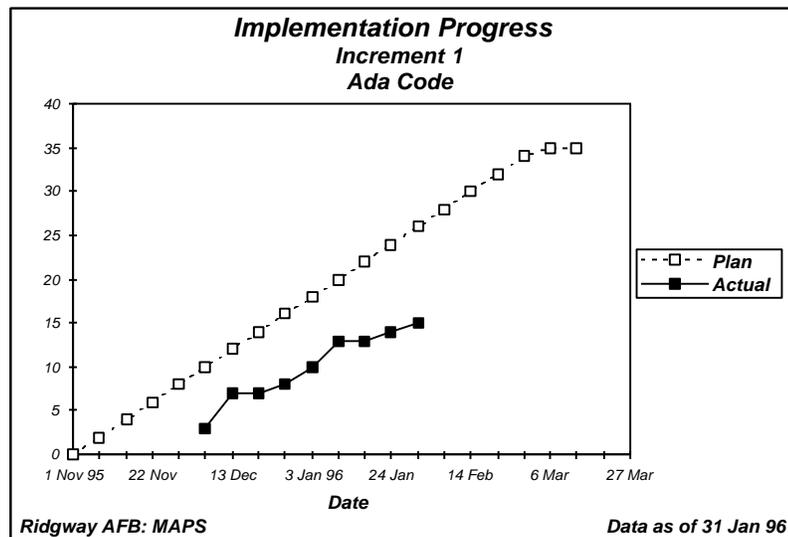


Figure 2-8. Ada Unit Implementation Progress

She decided that completed lines of code was a better measure of progress than a count of units because the data conversion software was divided up into relatively few units and they varied drastically in size. The units were not equivalent and using them to track progress would have been misleading. Jennifer generated the plan and actuals for the data conversion software and graphed the indicator as shown in Figure 2-9.

The results showed that the data conversion software development progress was reasonably on track.

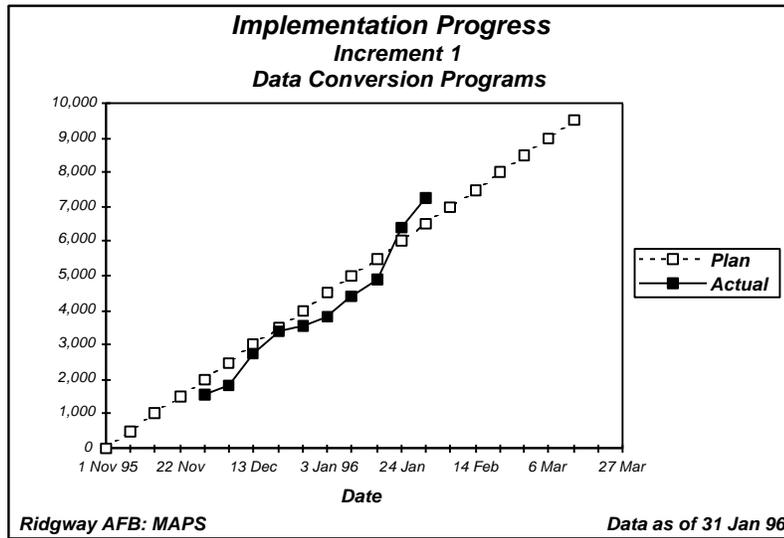


Figure 2-9. Data Conversion Implementation Progress

CHAPTER 3 - EVALUATING READINESS FOR DELIVERY

During 1996, the MAPS measurement process was effective in helping to manage software development effort. Progress against the revised plan was sufficient enough to allow for the resolution of the problem reports that were previously backlogged. Additional personnel which were earlier added to the development team allowed for the concurrent development of both the Base and Headquarters level MAPS increments. The progress measures showed that Increment 1 was nearing the completion of integration and test, and some system level testing had already been conducted. The primary issue had shifted from schedule and progress to the quality of the software. The key question was the readiness of the software for Operational Acceptance Testing.

3.1 INCREMENT 1

As the initial 1997 delivery dates grew closer, Lt. Col. Thompson wanted to know if Increment 1 was ready to begin the Operational Acceptance Test. To help answer this question Jennifer defined a set of related indicators and graphed them as shown in Figure 3-1.

When Jennifer first joined the MAPS program, the program had not been collecting effort data at the level of detail required to show how much effort was being applied to software rework. As an organic development activity it was difficult to get the staff to record on their timecards how they actually applied their effort during the week. Since the emphasis had been on generating new code to meet the existing schedule, the development team didn't see a need for the information anyway. As such, only development effort was collected as part of the time-reporting system. To get the data that she needed, Jennifer asked one of the programmers to modify the problem reporting system to collect the "re-development" and "retesting" effort data related to software rework on a problem by problem basis.

Software Size Estimates
Increment 1

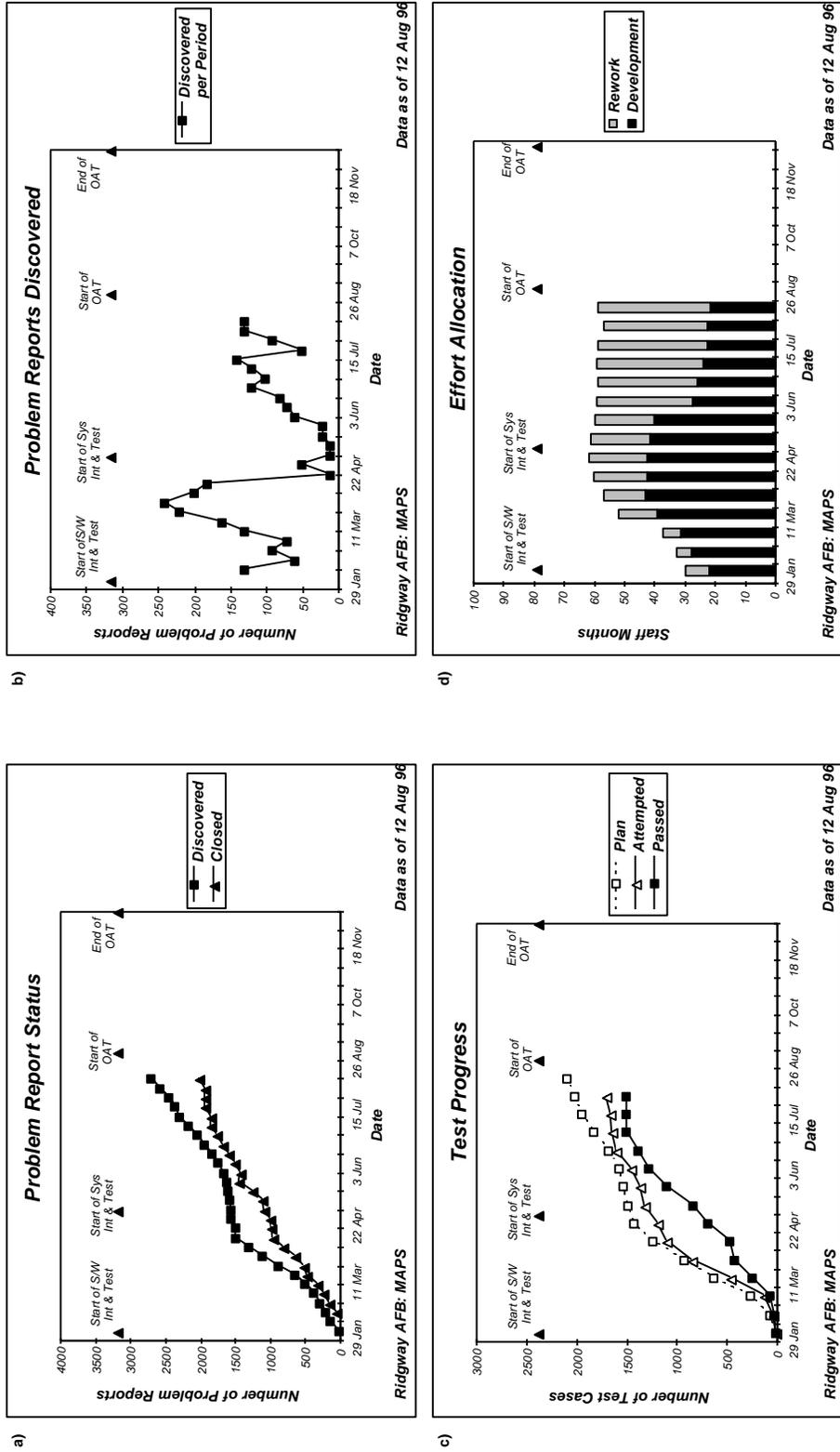


Figure 3-1. Increment 1 Readiness for Test

The change in the process was briefed to the developers, and Jennifer began to collect the data she needed to compare the amount of effort spent in rework vs. new development. The data was graphed and is presented in the lower-right hand quadrant of Figure 3-1.

Jennifer combined the rework effort data with a work unit progress graphs for cumulative problem reports (Figure 3-1a), and a graph of the number of problem reports being opened on a weekly basis (Figure 3-1b). She also included a graph of test case progress (Figure 3-1c). This combination of measurement results suggested that Increment 1 was not yet ready to begin the Operational Acceptance Test. Lt. Col. Thompson wanted to see the open and closed problem report trends converging, the number of new problems being discovered declining, the number of test cases passed equal to the number planned, and the amount effort being applied for rework decreasing. The results indicated that the development staff was increasingly spending time correcting new Increment 1 problems. This was of concern because they should have been transitioning to the development of the code for Increment 2. He met with Jennifer and asked her for more information in order to identify what needed to be done to improve the situation. Specifically, he wanted information about the types of problems that were being reported. He was hoping that there was a common type of problem that could be effectively dealt with.

Jennifer spent the better part of a week with several of the testing personnel reviewing the problem reports and classifying them as being related to performance, logic, interfaces, or other. She decided to implement this classification scheme as a permanent part of the problem reporting system so that the information would be readily available to support future analysis. The results of the classification effort were graphed and are depicted in Figure 3-2. By far, the greatest source of the Increment 1 problems were related to performance deficiencies

Jennifer further classified the performance problems according to their source. The results are shown in Figure 3-3. The most common type of performance problem was due to the incorrect use of SQL by the developers

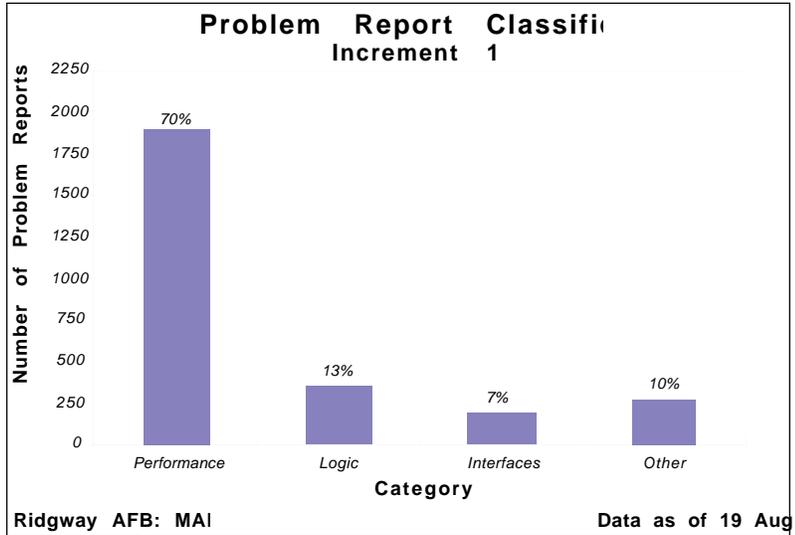


Figure 3-2. Increment 1 Problem Report Classification

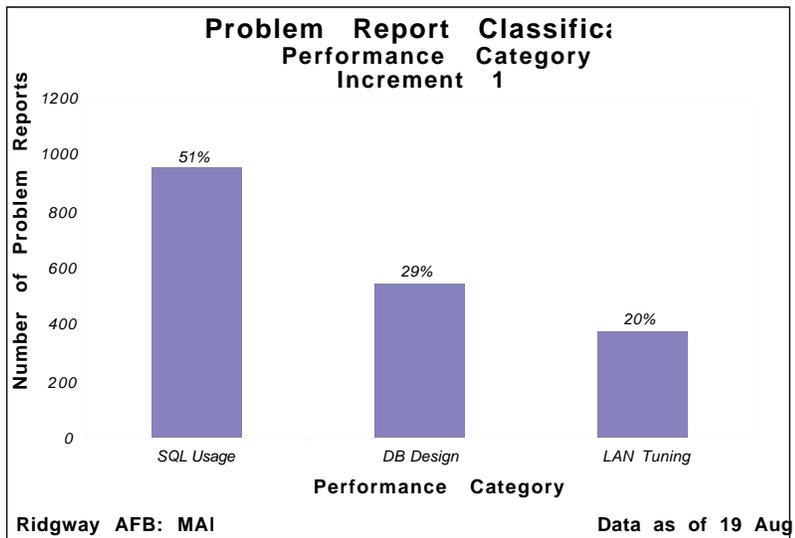


Figure 3-3. Source of Performance Problems

Jennifer discussed the results of her analysis with Lt. Col. Thompson and pointed out that the MAPS development represented the first time that many of the people on the development team had used a relational database and SQL. The staff's previous experience had been with hierarchical databases and COBOL. This probably should not have been a surprise since the SQL issue was part of the reason for the previous Personnel Information CSCI development problems. Lt. Col. Thompson again decided to bring in some additional expertise to address the SQL issue. Although it wasn't

the best approach this late in the program, the problems needed to be fixed quickly.

3.2 INCREMENT 2

Increment 2 was scheduled for delivery early in 1997. According to the development schedule, Increment 2 should have been nearing the completion of System Test by the end of February, 1997. To assess the Increment 2 readiness for test status, Jennifer generated the same combination of graphs using the same indicators as she had done for Increment 1. The results are shown in Figure 3-4.

This time the situation was much more encouraging. The trends for open and closed problem reports were converging, the discovery rate for new problems was declining rapidly, and the amount of rework was relatively low and stable. In addition, a comparison between the number of test cases planned, executed, and passed provided further evidence that testing was being completed in accordance with the schedule. Jennifer wondered why the number of newly discovered problems was declining so rapidly. Was the software that much better? Were discovered problems not being reported? Had the testing stopped? The test progress results helped Jennifer answer part of her question. Since testing was proceeding as scheduled, the lower number of new problem reports were not a result of reduced testing efforts. Jennifer looked into the reporting process and found that the identified problems were still being consistently documented.

Jennifer had continued to track the classes of problems being reported as shown in Figure 3-5. In contrast to the results for Increment 1, which had a high proportion of problems related to performance, the problems for Increment 2 were much more evenly distributed. In all the measurement data for Increment 2 indicated that the issues and problems that were experienced in Increment 1 had been successfully addressed. Lt. Col. Thompson's decisions with respect to focusing the right resources where they were needed had helped.

Readiness for Test Increment 2

Software Size Estimates

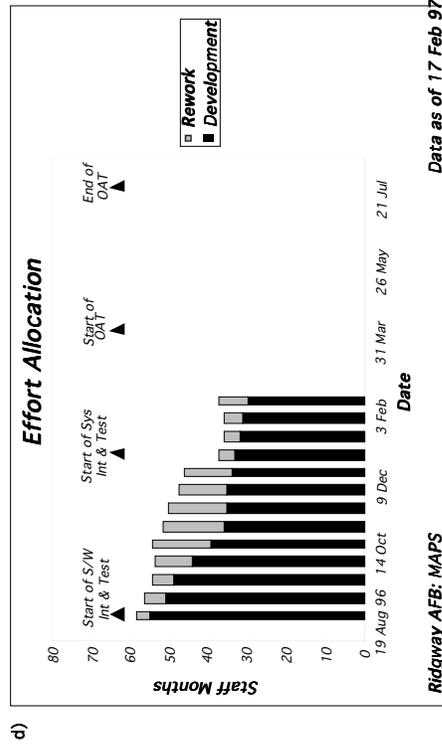
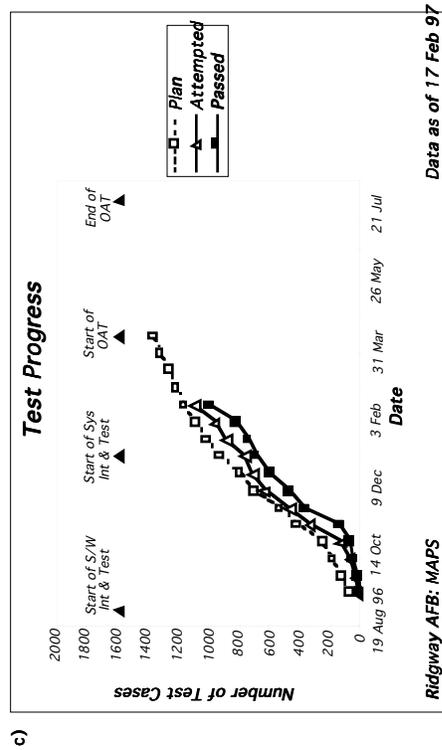
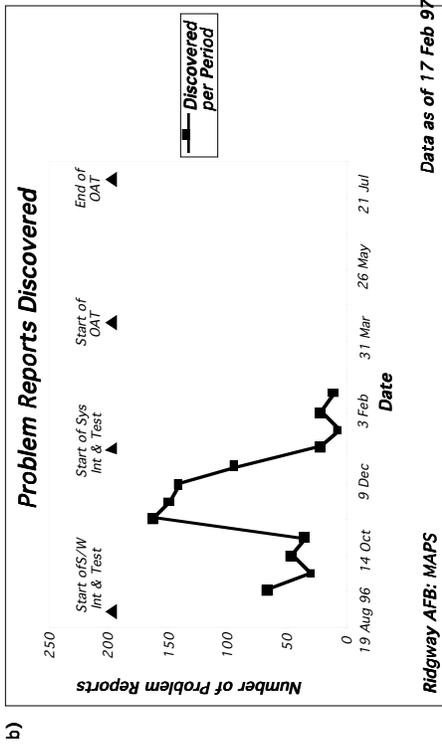
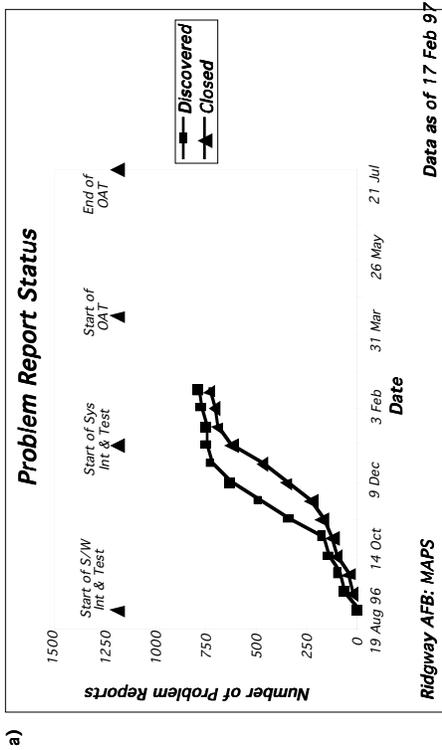


Figure 3-4. Increment 2 Readiness for Test

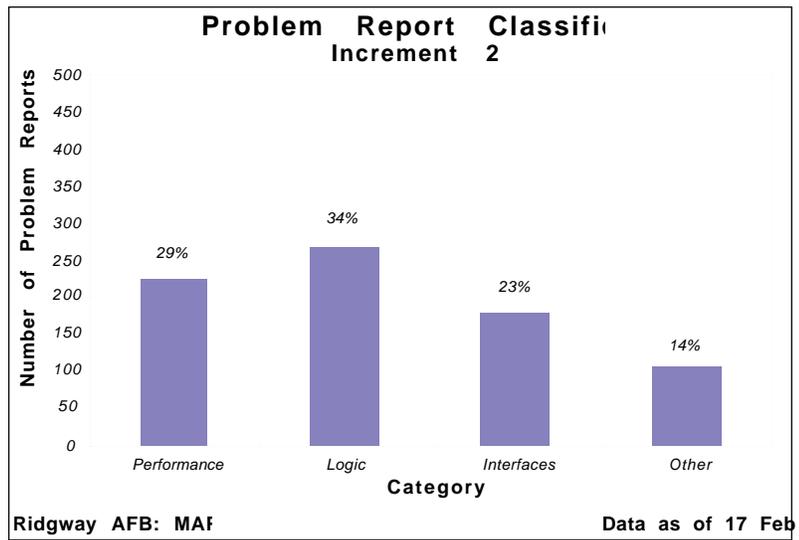


Figure 3-5. Increment Problem Report Classification

CHAPTER 4 - INSTALLATION AND SOFTWARE SUPPORT

With the development of the MAPS software proceeding according to plan, Lt. Col. Thompson asked Jennifer to extend the measurement process to track the progress of the fielding of the Increment 1 Base-level systems at the various bases. This was scheduled to occur throughout 1997, from January through October, with delivery of the systems occurring at a relatively constant rate.

To support the installation process, a total of ten people were assigned and divided into five teams. Each team was scheduled to spend two weeks installing MAPS at each of the 100 base-level sites. The work during the two week installation period included data conversion, software installation, user training and user support. After installation the MAPS development team would provide support via a 24-hour help line. The planned called for each site to concurrently run the existing military personnel system with the newly installed MAPS for one week before shutting down the old system completely. The 100 base-level sites included all Air Force bases in the United States and overseas, Air Force Reserve commands, and selected Air National Guard units.

4.1 INCREMENT 1 INSTALLATION

To address the installation progress question, Jennifer defined and graphed a simple work unit progress indicator as depicted in Figure 4-1.

It's clear from the graph that the installations were behind schedule almost from the start. Jennifer investigated and contacted each of the installation teams to try and identify the causes for the delays. She heard a consistent story. The old base-level system that MAPS was replacing had very loose edit requirements.

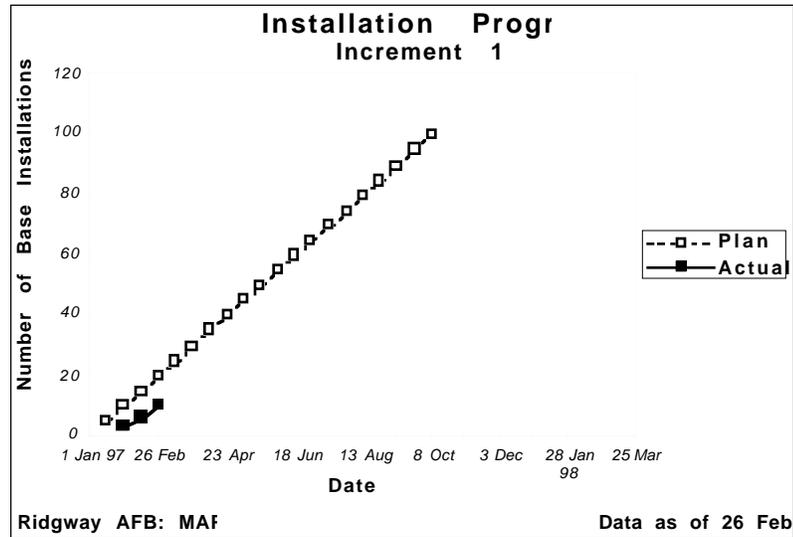


Figure 4-1. Increment 1 Installation Progress

It would accept almost any personnel data that was entered. The result was that the data conversion software, which was written to the MAPS data specifications, kept rejecting data that was in a different format from what was expected. This was not an easy problem to fix because each of the existing base-level databases was different from the others. Jennifer showed Lt. Col. Thompson a linear extrapolation of the actual installation data points. This is shown in Figure 4-2. Based on the actual rate of progress, a total of fifteen months was required to complete the installations, not ten months as originally planned. The rate of base installation was limited by the availability of teams. Based on the projection, Lt. Col. Thompson decided to extend the installation schedule. He also asked Jennifer to provide an update to the projection as more data became available.

4.2 SOFTWARE SUPPORT

By November of 1997, sixty-eight of the 100 base-level sites had been installed. As part of the measurement process, Jennifer had been tracking and categorizing problem reports from the field. Given the previous problems on the program, it was important to Lt. Col. Thompson to address the user's concerns.

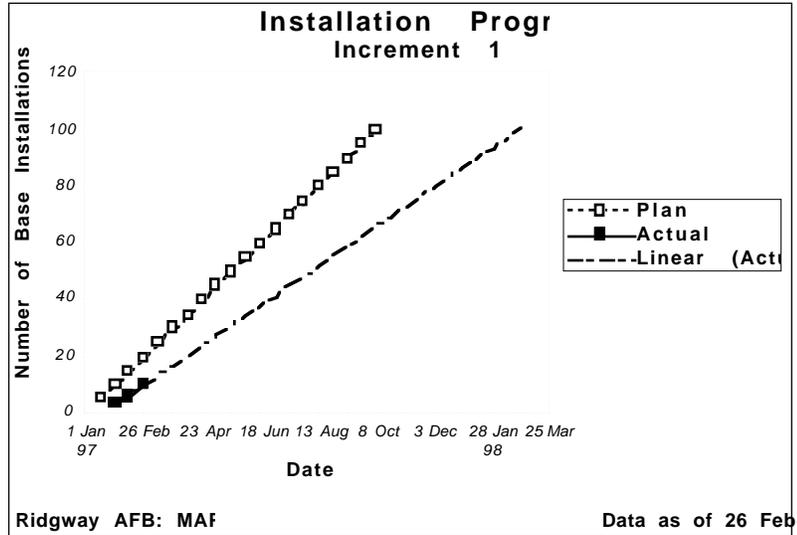


Figure 4-2. Increment 1 Projected Installation Completion

At the highest level, Jennifer classified the problem reports as being related to hardware, software, or user error. She analyzed the software-related problem reports in more detail by focusing on those that were the result of defects in the design or the code. She classified the problems as related to performance, logic, interfaces with other systems, and other. The data coming in from the field showed that the most frequent type of problem was related to logic defects. This is shown in Figure 4-3.

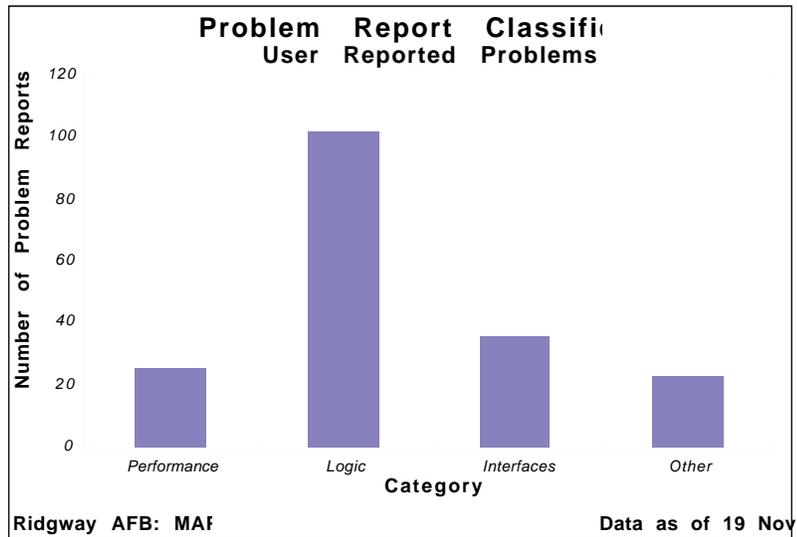


Figure 4-3. User Reported Problems

Jennifer also decided to classify the problems according to their source by identifying the CSCI which had to be changed in order to correct the problem. She graphed the ratio of problem reports to function points for each CSCI. The results were graphed as shown in Figure 4-4. Jennifer found that the Unit Mobilization CSCI accounted for a disproportionate number of the logic defects. There was clearly a problem with this particular CSCI.

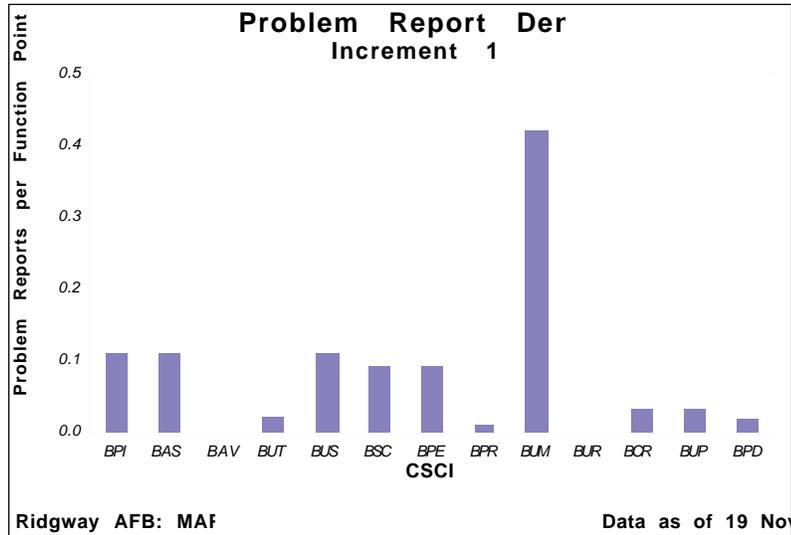


Figure 4-4. Problem Report Density by CSCI

Lt. Col. Thompson asked Jennifer to compare how much effort was being applied to correcting the problems, with what it would cost to redesign and redevelop the Unit Mobilization CSCI. Jennifer generated the graph shown in Figure 4-5, which reflected the effort that was applied over a two-month period.

Jennifer noted that the Unit Mobilization CSCI required the equivalent of three full-time staff members to support problem resolution. She was surprised that there continued to be such a high rate of newly discovered problems, particularly considering that the Unit Mobilization CSCI had been in operational use for almost a year. In talking with the lead programmer responsible for maintaining the CSCI, she found that as existing problems were corrected, new ones were being introduced. She decided to compare the cost of continuing to maintain the CSCI as currently implemented over a projected ten year period with the cost of reengineering and maintaining a more reliable version of the CSCI. (The screen and report generation functions did not need to be changed).

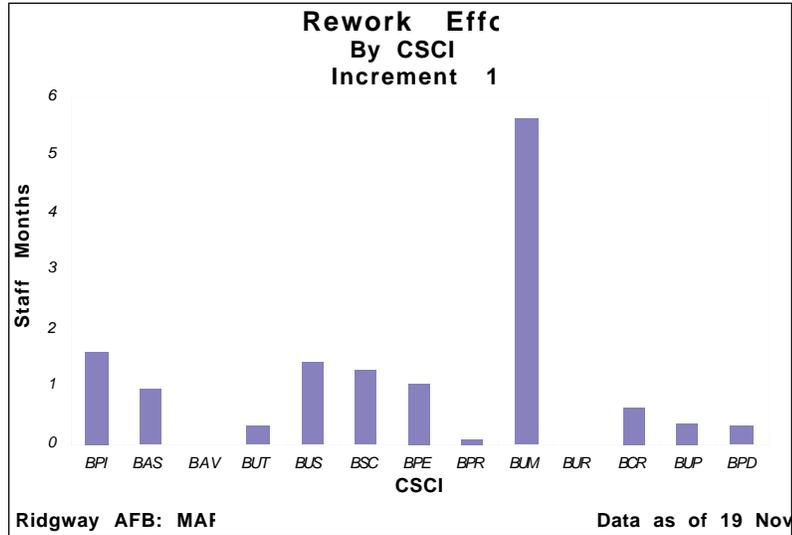


Figure 4-5. Rework Effort by CSCI

Jennifer estimated that the cost of reengineering would be \$1.2 million over a 10-month period, with estimated software support costs of \$800 thousand over the remaining nine year period. This was compared to an estimated \$3.0 million cost to maintain the existing CSCI over the same ten year time frame. This comparison was based on an average \$100K cost per person year.

Lt. Col. Thompson decided to redesign the Unit Mobilization CSCI and planned to release it in the next MAPS update scheduled for late 1998.

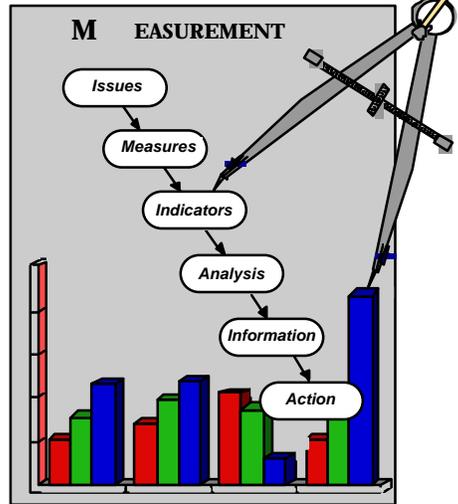
4.3 EPILOGUE

The MAPS development turned out to be a good example of implementing a measurement process on an existing program. As the program progressed, the data required to manage the key issues was identified, collected, and analyzed. The measurement activity was focused on the primary software issues. As such, a smaller number of measures were actually implemented, to address progress, cost, and quality concerns.

The measurement process was adapted to the specific characteristics of the MAPS program. Measures better suited to AIS software, such as Function Points, were implemented. By the end of the MAPS development, the entire program team had seen how measurement was useful in identifying and resolving both management and technical problems.

PRACTICAL

SOFTWARE



SUPPLEMENTAL INFORMATION

PART 6

SUPPLEMENTAL INFORMATION

This part of the PSM Guide provides supplemental information that will assist you in locating specific topics within the Guide. It provides more information about Practical Software Measurement, including its applicability to DoD policies.

The topics covered in this part of the Guide include:

- *Glossary - This section provides definitions of terms used throughout the Guide.*
- *Acronyms - Acronyms used throughout the Guide are defined in this section.*
- *Bibliography - Reference information and their respective sources are provided in this section.*
- *PSM Relationship to Specific DoD Policies - This section describes how PSM relates to key DoD measurement related policies and initiatives.*
- *PSM Project Information Summary - A brief discussion of the PSM project is provided in this section.*
- *Version Description Summary - This section provides the version history of the PSM Guide and the changes that have been made in each release of the document.*
- *Comment Form - Comments and suggestions concerning the PSM Guide may be submitted using this form.*

TABLE OF CONTENTS

GLOSSARY..... 357

ACRONYMS..... 365

BIBLIOGRAPHY..... 367

 Software Measurement References.....367

 Government Agency-Specific Software Measurement References.....372

PSM RELATIONSHIP TO SPECIFIC DOD POLICIES..... 373

PSM PROJECT INFORMATION SUMMARY..... 377

 Use of Practical Software Measurement.....378

 Project Contact Information.....378

VERSION DESCRIPTION SUMMARY..... 381

GLOSSARY

actual measure - See *measured value*

aggregation level - The level at which measurement data is combined for data reporting and analysis purposes. For example, to analyze the amount of work effort expended over time, work effort hours might be summed at the CSCI, build, or system levels. In general, aggregations are related to either the software design structure or to software activities. Aggregating data requires the definition of the hierarchical relationships among the measured items, such as is provided by a WBS.

application - In *PSM*, this term is used to refer to one of the two basic measurement activities which comprise the software measurement process. The *application* activity involves collecting, analyzing, and acting upon the measurement data. See also *tailoring* and *implementation*.

application software - Software specifically produced for the functional use of a computer system, as opposed to system software; for example, software for navigation, fire control, payroll, or general ledger.

attribute - A characteristic of a product or process.

Automated Information System (AIS) - A combination of computer hardware and software, data, or telecommunications, that performs functions such as collecting, processing, transmitting, and displaying information. Excluded are computer resources, both hardware and software, that are physically part of, dedicated to, or essential in real time to the mission performance of weapon systems.

Commercial Off-The-Shelf (COTS) - Commercial items that require no unique government modifications or maintenance over the life-cycle of the product to meet the needs of the procuring agency.

common issue - An issue that is basic or common to almost all programs. *PSM* defines six common issues. See *issue*.

Computer Software Configuration Item (CSCI) - An aggregation of software that satisfies an end use function and is designated for

separate configuration management by the acquirer. CSCIs are selected based on tradeoffs among software function, size, host or target computers, support concept, plans for reuse, criticality, interface considerations, need to be separately documented and controlled, and other factors.

Cost/Schedule Control System Criteria (C/SCSC) - A set of DoD requirements which defines what a contractor's management control system must have to qualify for bidding on selected military program acquisitions. The criteria include: requirements for integrating cost, schedule and technical performance measurements using the Work Breakdown Structure (WBS); use of accrual accounting methods to facilitate the analysis of variances from planned activities; and having a means to estimate the cost of the contract at completion.

customer - The organization that procures software products for itself or another organization. *PSM* generally considers the customer to be the DoD program manager/program office.

cyclomatic complexity - A measure of the logical complexity of a program module, based on the number of linearly independent paths in the module. Used to evaluate code quality and to predict testing effort.

defect - A product's nonconformance with its specification; any error in documentation, requirements design, code, test plans, or any other work product. Defects are uncovered during reviews, testing, and during operation.

developer - An organization that develops software products ("develop" may include new development, modification, reuse, reengineering, maintenance, or any other activity that results in software products). The developer may be a contractor or a Government agency.

development - The set of activities that result in software products, including requirements analysis, design, implementation, and integration and testing. This term is used throughout *PSM* to describe the second of three phases in the software life cycle.

expected (or planned) value - Planned or historical measurement data such as planned milestone dates, target level of reliability or required productivity. See also *measured value*

failure - 1) Termination of the ability of a functional unit to perform its required function; 2) an event in which the system or system component does not perform a required function within specified limits.

function points - Function Points are a software size measure. They measure the amount of information processing functionality contained within a software product. They are derived early in the software life cycle from requirements or design specifications, and are applied across diverse application domains and technology platforms.

implementation - In *PSM*, this term is used to refer to the activities required to establish a measurement process within an organization. See also *tailoring* and *application*.

indicator - A measure or combination of measures that provides insight into a software issue or concept. *PSM* makes frequent use of indicators that are comparisons, such as planned versus actual measures. These are generally presented as graphs or tables.

Indicators are used in two ways:

current indicator - An indicator that describes the current situation with respect to an issue. For example, staffing level reflects the variance between the number of personnel currently assigned to the program and the number of personnel allocated in the plan. Contrast with *leading indicator*.

leading indicator - An indicator that predicts the future situation with respect to an issue. For example, requirement changes may be a leading indicator for developer effort. Changes in requirements usually result in a need for increased effort. Contrast with *current indicator*.

There are two types of indicators:

limit-based indicator - An indicator whose expected or planned value remains relatively constant. For example, actual software size may be collected throughout a program and compared to a planned limit (initial estimate plus acceptable variation). Contrast with *trend-based indicator*.

trend-based indicator - An indicator whose expected or planned value changes over time. For example, progress might be tracked using work units completed. A different goal for the number of units completed would be set for each week and compared to actual units completed over time. Contrast with *limit-based indicator*.

instrumentation - Instructions inserted into software to monitor the operation of a system or component or to collect measurement data.

issue - A risk, constraint, objective or concern, often associated with resources, progress, productivity, or performance. Issues represent current or potential problem areas that should be monitored.

low level data - Measures collected at the lowest component and software activity levels as defined by the software architecture and work breakdown structure.

masking - When a problem that should show up in one issue area is disguised by an accommodation made in another issue area. For example, an increase in applied effort is masked by concurrent schedule slips so that increased effort does not result in a detectable increase in staff level. Masking makes it harder for management to recognize a problem.

measure - The result of counting or otherwise quantifying an attribute of a process or product. Measures are numerical values assigned to software attributes according to defined criteria.

measured (or actual) value - Actual, current measurement data such as hours of effort expended or lines of code produced. See also *expected value*.

measurement - The process of assigning numerical values to software attributes according to defined criteria. This process can be based on estimation or direct measurement. Estimation results in planned or expected measures. Direct measurement results in actual measures.

measurement analysis - The use of measurement data to identify, assess, project, and evaluate software issues.

feasibility analysis - Analysis conducted to determine whether plans and targets are realistic or achievable; should be conducted during the initial planning activity and at all subsequent replans. Contrast with *performance analysis*

performance analysis - Analysis conducted to determine whether software development is meeting the plans, assumptions, and targets; should be conducted continuously once a program has committed to a plan. Contrast with *feasibility analysis*

measurement analyst - The person(s) responsible for defining, collecting, analyzing, and reporting software measures in a given organization.

measurement category - A set of related measures. Each common issue defined in *PSM* has one or more corresponding measurement categories. Software measures which provide the same type of information are grouped under each measurement category. Each category answers different types of questions.

measurement data- A collection of measures.

measurement information - Knowledge derived from the analysis of measurement data and indicators.

metric - See *indicator*.

milestone - A scheduled event for which some project member or manager is held accountable and that is used to measure progress.

normalization - Combining or comparing measures from different activities, different programs, or with different units of production. For example, to compare the quality of work produced in two programs, it would be necessary look at defect counts in relation to the amount or size of the work produced. This often requires the definition and validation of conversion rules and/or models.

problem report - A documented description of a defect, unusual occurrence, observation, or failure that requires investigation and may involve software modifications.

program manager - The government official responsible for acquiring or supporting a system that meets technical, cost, schedule,

and quality requirements. Acquisition and support includes both internal and contracted tasks.

program planning - The set of activities involving the assessment and selection of software develop(s) and the development of program plans. This term is used throughout *PSM* to describe the first of three phases in the software lifecycle.

repeatability - A ability of two analysts, performing the same measurement analysis, to arrive at the same set of conclusions and recommendations.

rework - Any effort (Also, any size changes necessary to accomplish rework.) invested in reaccomplishing work already deemed complete. Rework effort begins once a defect is found and continues until all of the work required to obtain acceptance of the rework is complete.

rippling - When a problem that arises in one issue area may have a ripple effect on another issue. For example, software size growth may cause effort overruns. Rippling multiplies the effect of an issue.

risk - A subjective assessment made regarding the likelihood or probability of not achieving a specific objective by the time established with the resources provided or requested.

software activity - In *PSM*, this term is used to refer to the four key subprocesses of the overall software process; requirements analysis, design, implementation, and integration and test. Individual software activities can take place at any point in the software life cycle in any phase.

software component - A general term used to refer to a software system or an element, such as unit, CSCI, object, or screen.

Software Engineering Process Group (SEPG) - The group of specialists who facilitate the definition, maintenance, and improvement of the software process used by the organization.

software manager - The person responsible for making the decisions relating to the software issues. This could be the DoD Program Manager or the developer's program or technical manager

software program - The people, processes, and organizations responsible for developing or supporting a software product as a stand-alone item or as part of a larger system.

software support - The set of activities that takes place to ensure that software installed for operational use continues to perform as intended and fulfill its intended role in system operations. This term is used throughout *PSM* to describe the third of three phases in the software life cycle. Software development can take place during the software support phase.

tailoring - In *PSM*, this term is used to refer to one of the three basic measurement activities. The *tailoring* activity is part of the measurement process and involves selecting an effective and economical set of measures for a program. See also *application* and *implementation*.

traceability - The ability to link conclusions and recommendations to a defined sequence of steps.

weapon system - Items that can be used directly or indirectly by the armed forces to carry out combat missions.

work breakdown structure (WBS) - A work breakdown structure for software defines the software-related elements associated with program activities. *PSM* refers to cost and effort measures which are aggregated and analyzed at various WBS levels.

ACRONYMS

| | |
|------------------|--|
| A&T | Acquisition and Technology |
| AIS | Automated Information System |
| C/SCSC | Cost/Schedule Control System Criteria |
| C/SSR | Cost/Schedule Status Reports |
| C ⁴ I | Command, Control, Communications, Computer, and Intelligence |
| CCDR | Contractor Cost Data Report |
| CFSR | Contract Funds Status Report |
| CMM | Capability Maturity Model |
| COCOMO | Constructive Cost Model |
| COTS | Commercial Off the Shelf |
| CPR | Cost Performance Report |
| CSCI | Computer Software Configuration Item |
| DAB | Defense Acquisition Board |
| DSMC | Defense Systems Management College |
| DT&E | Development, Test, and Evaluation |
| E&MD | Engineering and Manufacturing Development |
| GAO | General Accounting Office |
| GOTS | Government Off the Shelf |
| IFPUG | International Function Point Users Group |
| IPPD | Integrated Product and Process Development |
| IPT | Integrated Product Team |
| ISSA | Inter Service Support Agreement |
| JGSE | Joint Group on Systems Engineering |

| | |
|--------|--|
| JLC | Joint Logistics Commanders |
| LAN | Local Area Network |
| MAISAP | Major Automated Information System Acquisition Program |
| MAISRC | Major Automated Information System Review Council |
| MDAP | Major Defense Acquisition Program |
| MIS | Management Information System |
| MOA | Memorandum of Agreement |
| MOU | Memorandum of Understanding |
| NDI | Non-Developed Item |
| OSA | Open Systems Architecture |
| OSD | Office of the Secretary of Defense |
| OT&E | Operational Test and Evaluation |
| OUSD | Office of the Under Secretary of Defense |
| PSM | Practical Software Measurement |
| RFP | Request for Proposal |
| SEI | Software Engineering Institute |
| SEPG | Software Engineering Process Group |
| SISMA | Streamlined Integrated Software Metrics Approach |
| SPC | Software Productivity Consortium |
| STEP | Software Test and Evaluation Panel |
| WBS | Work Breakdown Structure |

BIBLIOGRAPHY

This bibliography lists measurement references which augment or support the guidance included in Practical Software Measurement. Readers may wish to consult these resources for additional information. The first section includes published measurement-related books and reference manuals. Brief annotations are provided that describe each reference. The second section includes government agency-specific directives, instructions, reports, and standards which address software measurement. The books are generally available through most technical publishers and bookstores. Government documents are available through the National Technical Information Service, Springfield, VA 22161.

SOFTWARE MEASUREMENT REFERENCES

Baumert, John H., and Mark S. McWhinney, September 1992, *Software Measures and the Capability Maturity Model*, CMU/SEI-92-TR-25, ESC-TR-92-025, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

A reference which shows which software measures can reasonably be expected at the various levels of SEI software process maturity. It includes example graphs and advice on how to report specific measures.

Boehm, Barry W., 1981, *Software Engineering Economics*, Englewood Cliffs, NJ: Prentice-Hall.

A primary reference in the field of software estimation and measurement. This book provides detailed information on the COCOMO software estimation model.

Brooks, Frederick O., Jr., 1975, *The Mythical Man Month: Essays on Software Engineering*, Reading, MA: Addison-Wesley Publishing Company.

A primary reference in the field of software engineering. This book relates key lessons learned in managing a large software program, and provides an overall perspective for the DoD program manager.

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

September 1992, ***Software Measurement for DoD Systems: Recommendations for Initial Core Measures***, CMU/SEI-92-TR-19, ESC-TR-92-019, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

This reference provides recommendations and a rationale for the SEI defined "Core Measures". The Core Measures include size, effort, schedule, and quality (measured in terms of defects and problem reports) and address issues common to almost all software programs.

DeMarco, Tom, 1982, ***Controlling Software Projects: Management, Measurement, and Estimation***, New York: Yourdon Press.

This reference provides practical guidance for collecting and analyzing software measures.

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

This book describes the quality crisis across a number of industries and relates effective strategies for dealing with them. It focuses on the use of Statistical Process Control techniques.

Dumke, Reiner R., 1993, ***Software Metrics: A Subdivided Bibliography***, Magdeburg, Germany: Technical University "Otto von Guericke" of Magdeburg.

This bibliography provides a comprehensive guide to both research and practical publications in software measurement. Entries are grouped by topic.

Fenton, Norman E., 1991, ***Software Metrics: A Rigorous Approach***, London: Chapman & Hall.

This book advocates a rigorous approach to software measurement that is based on fundamental measurement theory. It argues that much of modern software measurement is flawed because it ignores measurement fundamentals. This book gives the reader specific tools to overcome these deficiencies and put a measurement program on solid theoretical ground. This book is for the reader who desires a more theoretical treatment of software measurement than is found in PSM.

Florac, William A., with the Quality Subgroup of the Software Metrics Definition Working Group and the Software Process Measurement Project Team, September 1992, ***Software Quality Measurement: A Framework for Counting Problems and Defects***,

CMU/SEI-92-TR-22, ESC-TR-92-022, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

This reference provides a framework for counting problems and defects in software and using them to assess quality, which is one of the SEI Core Measures. It includes checklists that allow the reader to define how defects are actively defined and counted.

Goethert, Wolfhart B., Elizabeth K. Bailey, Mary B. Busby, with the Effort and Schedule Subgroup of the Software Metrics Definition Working Group and the Software Process Measurement Project Team, September 1992, ***Software Effort and Schedule Measurement: A Framework for Counting Staff-Hours and Reporting Schedule Information***, CMU/SEI-92-TR-21, ESC-TR-92-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

This reference provides frameworks for counting software staff-hours and schedule, both of which are SEI Core Measures. It includes checklists that allow the reader to define how staff-hours and schedule data are actively defined and counted.

Grady, Robert B., and Deborah L. Caswell, 1987, ***Software Metrics: Establishing a Company-Wide Program***, Englewood Cliffs, NJ: Prentice-Hall.

This book describes how Hewlett-Packard's corporate measurement program was implemented. It includes information on topics which range from how to compute specific measures to how to sell a measurement program to senior management.

Grady, Robert B., 1992, ***Practical Software Metrics for Project Management and Process Improvement***, Englewood Cliffs, NJ: Prentice-Hall, Inc.

This book examines more detailed issues with respect to software measurement and more specifically relates it to software process improvement. It builds on information from the previous reference.

Hetzl, Bill, 1993, ***Making Software Measurement Work: Building an Effective Measurement Program***, Boston, MA: QED Publishing Group.

This book addresses how to get measurement implemented in an organization. It emphasizes fundamentals, explains how to begin, and includes a list of measurement tools and services available at the time of publication.

Humphrey, Watts, 1989, ***Managing the Software Process***, Addison Wesley, New York.

This book describes the software process maturity levels developed by the Software Engineering Institute at Carnegie Mellon University. It defines each maturity level (e.g., Ad Hoc, Repeatable, Defined, etc.), and outlines the criteria for distinguishing each one. This book contains the basis for, and is a precursor to, SEI's Capability Maturity Model.

The Institute of Electrical and Electronics Engineers, Inc., 1992, ***IEEE Standard for Software Productivity Metrics***, IEEE Std 1045-1992, New York, NY.

This IEEE standard describes a variety of software measures that can be used to consistently define software productivity. Detailed information is provided for each of the more than thirty measures it contains.

International Function Points Users Group, 1994, ***Function Points Counting Practices Manual*** Westerville, OH.

This industry-established standard defines the rules for counting function points. The document is updated periodically to account for advances in function point technology and usage.

International Function Points Users Group, 1994, ***Guidelines to Software Measurement*** Westerville, OH.

This guidebook introduces the basic concepts of software measurement. It describes how the measurement process fits into other software activities, and provides guidance on implementing a measurement program. It reviews product and process measures, discusses indicators, and examines ways to use measurement results.

Jones, T. Capers, 1991, ***Applied Software Measurement***, McGraw Hill, New York.

This book describes various methods for measuring schedule, cost, and quality of software projects. It discusses the major functional size metrics, including DeMarco's "Bang" metrics, function points, and feature points, but focuses primarily on the use of function points. Productivity and quality averages from Jones' historical data base are included.

Musa, John D., Anthony Iannino, and Kazuhira Okumoto, 1987, ***Software Reliability: Measurement, Prediction, Application***, New York, NY: McGraw-Hill Book Company.

This book discusses the theoretical and practical applications of software reliability measurement. It defines software reliability, reviews and compares the various reliability models, and describes how reliability measurement can be used in systems engineering, project management, and in the management of the operational phase of the software life cycle.

Park, Robert E., with the Size Subgroup of the Software Metrics Definition Working Group and the Software Process Measurement Project Team, September 1992, ***Software Size Measurement: A Framework for Counting Source Statements***, CMU/SEI-92-TR-20, ESC-TR-92-020, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

This reference provides a framework for counting source lines of code (SLOC) and using it to assess software size, which is one of the SEI Core Measures. It includes checklists that allow the reader to define how SLOC are defined and counted.

Paulk, Mark C., Bill Curtis, Mary Beth Chrissis, and Charles V. Weber, 1993, ***Capability Maturity Model for Software, Version 1.1***, CMU/SEI-93-TR-24, ESC-TR-93-177, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

This reference describes a software process maturity framework which forms the basis for assessing the capability of a software organization. Five maturity levels and the key practices within each level are described.

Putnam, Lawrence H. and Ware Myers, 1992, ***Measures for Excellence: Reliable Software on Time, within Budget***, Englewood Cliffs, NJ: Prentice-Hall.

This book focuses primarily on using tools for automated size estimation and project tracking, and also discusses life cycle models, life cycle management, and productivity analysis. It includes observations about patterns of software behavior based on Putnam's historical data base of software projects.

Schultz, Herman P., May 1988, ***Software Management Metrics***, M88-1, ESD-TR-88-001, The MITRE Corporation, Bedford, MA.

This reference is a relatively early work from a military software measurement viewpoint. It provides an initial overview of selected software management indicators.

Software Productivity Consortium, ***Software Measurement Guidebook***, SPC-91060-CMC, Version 02.01.00, August 1994, Software Productivity Consortium, Herndon, VA.

This reference provides detailed information which helps to define and interpret a software measurement process. It contains detailed guidance on a number of software measures.

GOVERNMENT AGENCY-SPECIFIC SOFTWARE MEASUREMENT REFERENCES

Department of the Air Force. *Acquisition Management: Software Management Indicators* AFP800-48, Washington, DC. June 1992.

Department of the Air Force. Software Technology Support Center. *Guidelines for Successful Acquisition and Management of Software Intensive Systems: Weapon Systems, Command and Control Systems, Management Information Systems*, U.S. Air Force STSC, Hill AFB, UT. February 1995.

Department of the Army. Communications-Electronics Command, Research, Development and Engineering Center, Software Engineering Directorate. *Streamlined Integrated Software Metrics Approach (SISMA) Guidebook: Application of STEP Metrics*, U.S. Army CECOM SED, Ft. Monmouth, NJ. July 1993.

Department of the Army. “*Software Test and Evaluation Guideline*” *Test and Evaluation Procedures and Guidelines*, DA Pamphlet 73-1, Washington, DC. June 1992.

Department of the Navy. Naval Aviation Systems Team. *Software Metrics Program Handbook*, AVDEP-HDBK-7, U.S. Navy, Naval Aviation Systems Team, Arlington, VA. November 1995.

Department of the Navy. Cruise Missile Project and Unmanned Aerial Vehicles Joint Project. *Software Metric Utilization Guidance*, PEO(CU)P 5234/2, June 1994.

Federal Aviation Administration. Software Engineering Specialty Group. *National Airspace System (NAS) Software Management Indicators Handbook*, FAA-SESG-HDBK-93.01, FAA SESG, Washington, DC. November 1993.

PSM RELATIONSHIP TO SPECIFIC DoD POLICIES

Practical Software Measurement was developed to help DoD Program Managers address key software issues related to planning and implementing a software intensive program. *PSM* is based on the measurement experience of many DoD and industry organizations, many of which are responsible for measurement policy and related programs. The participation of these organizations in the development of *PSM* helped to shape the guidance so that it complements existing DoD measurement policies and initiatives.

The guidance contained in *PSM* focuses on defining and implementing a practical measurement process. It explains how to tailor software measures to address the specific issues of each program, and how to apply the measures to support informed software decision making. One of the primary goals in the development of the *PSM* guidance was to ensure that it supported the objectives and intent of current DoD policy. Since the *PSM* guidance was developed using current measurement best practices, it is important that it support the measurement initiatives which have become accepted within the DoD software engineering community.

PSM's relationship to key DoD software measurement related policies and initiatives is described as follows:

DoD Directive 5000.1, “Defense Acquisition” and DoD Instruction 5000.2, “Mandatory Procedures for Major Defense Acquisition Programs (MDAPs) and Major Automated Information System Acquisition Programs (MAISAPs)”, 1996

These DoD policies establish the requirement for DoD program managers to provide periodic reports on the cost, schedule and performance of their systems throughout the lifecycle. *PSM* helps the program manager to implement a process for collecting and reporting this information for software intensive systems.

Secretary of Defense Memorandum of 10 May 1995, “Use of Integrated Product and Process Development and Integrated Product Teams in DoD Acquisition”

This memorandum from the Secretary of Defense requires the use of Integrated Product and Process Development (IPPD) and Integrated Product Teams (IPTs) throughout the acquisition process. The IPPD/IPT management approach emphasizes the use of objective information and associated measures within a cooperative framework which involves all program participants. It also stresses open communications within the program team to identify and resolve problems. Within the IPT structure, *PSM* helps the Program Manager to objectively identify and resolve program issues using objective software information. The overall characteristics of the *PSM* measurement process, including independent analysis, continuous feedback, communications within the program team, and the integration of measurement requirements into the developer’s process, directly support the IPPD/IPT requirements.

OUSD/A&T Memorandum of 23 May 1994, “Development Test and Evaluation (DT&E) Policy Guidance for Software-Intensive Systems in Support of Recommendations from the General Accounting Office (GAO)” and OSD/OT&E Memorandum of 31 May 1994, “Software Maturity Criteria for Dedicated Operational Test and Evaluation of Software-Intensive Systems”

These policy memoranda require that every program define and implement a set of software measures early in the program to help determine when the system is ready for Operational Test and Evaluation (OT&E). Commonly referred to as “maturity” measures, those addressed in the policy include fault profile, cost, schedule, requirements traceability, requirements stability, deficiency tracking, and breadth and depth of testing. The *PSM* guidance defines how to implement a measurement process which will provide the required maturity information. The *PSM* measurement selection guidance includes measures which satisfy the maturity assessment requirements.

Department of the Army Pamphlet 73-1, Part Seven (Draft), “Software Test and Evaluation Guidelines”, 30 September 1992

This Army policy memorandum defines a set of software measures which must be applied to all Army software-intensive systems. This policy resulted from the recommendations of the Army Software Test and Evaluation Panel (STEP), which required the implementation of twelve specific measures. These measures were not intended to be tailored for each program. The original Army policy was revised in 1994. Although the 12 measures were still required, the revised policy allowed the data elements to be tailored. The revised policy is implemented in the current Army Metrics System. *PSM* is complimentary with current Army measurement policy. The *PSM* guidance helps Army Program Managers to implement a measurement process to support the use of the twelve required measures. *PSM* helps to select additional measures that the Program Manager may require, and addresses associated analysis techniques. The twelve Army measures are included in those listed in the *PSM* measurement selection guidance

Department of the Air Force Software Metrics Implementation Policy (93M-017) of 17 February 1994

The Air Force measurement policy addresses the implementation of a software measurement process to support overall program management requirements. The policy also identifies five “core metrics” which are aligned with issues common to all Air Force programs. The five metrics include size, effort, schedule, quality, and rework. The policy requires that each program measure all five, but does not prescribe the actual measures to be used or associated measurement methodologies. These five measures are addressed in the *PSM* guidance. *PSM* also addresses the definition and implementation of the software measurement process required by Air Force policy.

Software Engineering Institute “Software Measurement for DoD Systems: Recommendations for Initial Core Measures Core Measures, 1992

In 1992 the Software Engineering Institute (SEI) published a series of documents focused on the definition and use of four core software measures. These measures include effort/staff hours, problems/defects, size, and schedule. The SEI documents provide a detailed

framework for defining and describing each measure, as well as implementation and interpretation guidance. The core measures have been widely adopted within DoD and industry. The SEI documents can be used in conjunction with the *PSM* guidance, especially with respect to integrating the core measures into the software process. The SEI documents help to explicitly define how each of the measures will be implemented.

Table 6-1 provides a summary of how *Practical Software Measurements* supports the listed policies and initiatives.

Table 6-1. PSM Relationship to DoD Measurement Policies and Initiatives

| PSM Guidance | 5000.1 5000.2 | IPPD IPT | OSD | Army | Air Force | SEI |
|---|--------------------------|---------------------|------------|-------------|----------------------|------------|
| Identify Program Specific Issues | A | A | N | N | N | A |
| Define Measures to be Collected | P | P | A | N | A | N |
| Integrate Measurement into Software Process | P | P | P | P | P | P |
| Put Measures on Contract | P | P | P | A | P | P |
| Provide Systematic Analysis Process | P | P | P | A | P | A |
| Provide Implementation Guidance | P | P | P | P | P | A |

Established by specific policy = N
 PSM fills a requirement established by policy = P
 PSM augments policy guidance = A

PSM PROJECT INFORMATION SUMMARY

The development of *Practical Software Measurement: A Guide to Objective Program Insight (PSM)* was sponsored by the Joint Logistics Commanders (JLC) Joint Group on Systems Engineering (JGSE). *PSM* was developed by a technical working group that includes representatives from across the DoD and industry. *PSM* represents the consensus of that community on the best practices for software measurement.

The authors of *PSM* have benefited from the published research and practical experience of many people outside of the technical working group. However, the *Guide* is a “user’s manual” for software measurement, not a survey of the research literature. Consult the bibliography for more background on the theory and origin of the basic measurement concepts integrated into *PSM*.

The *Guide* is the centerpiece of a family of products intended to help transition software measurement into practice within the DoD. *PSM* products currently available, or scheduled for delivery this year, include the following:

- **PSM Management Summary**- a short synopsis of the *PSM* approach intended to motivate managers to adopt software measurement.
- **PSM Guide (Version 2.0)**- the *Guide* explains the basic concepts of the software measurement process, offers detailed implementation guidance, and provides realistic case studies of software measurement use.
- **PSM Insight**- a Microsoft Access-based tool for tailoring and applying software measures for a specific program. The automated functions follow the *PSM* defined measurement process.
- **PSM Overview Course**- a one-day course that introduces the *PSM* principles and approach, and explains how to use the *PSM Guide*.
- **PSM Train-the-Trainers Course** a three-day course that explains how to teach the **PSM Overview Course** and how to use **PSM Insight**

- **PSM Orientation** -a two-hour overview of *PSM* principles and concepts.

In addition to these products, the *PSM* development team is also available to assist in applying *PSM* to actual programs and projects.

The *PSM* Technical Working Group (TWG) is developing guidance that will address the application of software measurement to process improvement and product engineering. Participation in the TWG is open to all. We encourage you to join and share your experience.

USE OF PRACTICAL SOFTWARE MEASUREMENT

One of the primary purposes of *Practical Software Measurement: A Guide to Objective Program Insight*, is to encourage the widespread implementation of software measurement throughout the DoD and industry. The information included in the *Guide* was developed by a group of measurement professionals who gave much of their own time and effort to help meet this objective.

We encourage you to make direct use of the material contained in *Practical Software Measurement*. We ask that you acknowledge the source of the information as:

Practical Software Measurement: A Guide to Objective Program Insight, Version 2.0, January 26, 1996.

Additional copies of this *Guide* are available in hard cover and electronic formats. Reproducible master copies are also available.

PROJECT CONTACT INFORMATION

Practical Software Measurement: A Guide to Objective Program Insight, is intended for those DoD acquisition and development organizations who need to more objectively plan, implement, control, and evaluate their software programs.

If you would like more information on using *Practical Software Measurement* for your program, please contact:

John McGarry
Naval Undersea Warfare Center
Code 2252
1176 Howell Street
Newport, RI 02841

(401) 841-4581 (Voice)
(401) 84-12130 (FAX)
DSN Prefix 948
mcgarry@ada.npt.nuwc.navy.mil

VERSION DESCRIPTION SUMMARY

| Date | Version | Change Description or Comments |
|------------------|----------------|---|
| April 12, 1995 | 1.0 | Initial coordination draft. |
| June 30, 1995 | 1.1 | Editorial updates |
| January 26, 1996 | 2.0 | Additional detail added throughout the <i>Guide</i> ; AIS Case Study added to Part 5. |

Practical Software Measurement Guide Evaluation and Comment Form

We welcome any comments that will help us improve Practical Software Measurement. Please provide your inputs via hardcopy or email using the information format provided below.

Holly G. Mills
Software Productivity Solutions, Inc.
122 4th Avenue
Indialantic, FL 32903

Phone: (407) 984-3370
FAX: (407) 728-3957
email: hgm@sps.com

| | |
|----------------------|------------|
| Name _____ | Date _____ |
| Organization _____ | |
| Street Address _____ | |
| _____ | |
| _____ | |
| E-mail Address _____ | |
| Telephone _____ | Fax _____ |
| _____ | |

Version of PSM Reviewed 2.0

Part Commented On:

- | | |
|--|--|
| <input type="checkbox"/> The Software Measurement Process | <input type="checkbox"/> Acquisition and Contract Implementation |
| <input type="checkbox"/> Guidance | |
| <input type="checkbox"/> Selecting and Specifying Program Measures | <input type="checkbox"/> Software Measurement Case Studies |
| <input type="checkbox"/> Analysis Techniques and Examples | <input type="checkbox"/> Supplemental Information |

Check here if you want to receive updates to the *Guide*

Overall value:

Excellent

Explanation:

Good

Fair

Not Useful

General Comments: _____

Specific Comments on Sections:

Section:

Page #

Comments:

Part 6 - Supplemental Information

Use additional sheets if more space is needed.