Technical Report
CMU/SEI-92-TR-022
ESC-TR-92-022

Software Quality Measurement: A Framework for Counting Problems and Defects

William A. Florac

with the Quality Subgroup of the Software Metrics Definition Working Group and the Software Process Measurement Project Team

Technical Report

CMU/SEI-92-TR-022
ESC-TR-92-022
September 1992

Software Quality Measurement:
A Framework for Counting Problems and Defects

William A. Florac

with the
Quality Subgroup of the Software Metrics Definition Working Group
and the
Software Process Measurement Project Team

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

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

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

FOR THE COMMANDER

(signature on file)

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

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

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

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

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

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

Table of Contents

List of Figures

Preface

In 1989, the Software Engineering Institute (SEI) began an effort to promote the use of measurement in the engineering, management, and acquisition of software systems. We believed that this was something that required participation from many members of the software community to be successful. As part of the effort, a steering committee was formed to provide technical guidance and to increase public awareness of the benefits of process and product measurements. Based on advice from the steering committee, two working groups were formed: one for software acquisition metrics and the other for software metrics definition. The first of these working groups was asked to identify a basic set of measures for use by government agencies that acquire software through contracted development efforts. The second was asked to construct measurement definitions and guidelines for organizations that produce or support software systems, and to give specific attention to measures of size, quality, effort, and schedule.

Since 1989, more than 60 representatives from industry, academia, and government have participated in SEI working group activities, and three resident affiliates have joined the Measurement Project staff. The Defense Advanced Research Projects Agency (DARPA) has also supported this work by making it a principal task under the Department of Defense Software Action Plan (SWAP). The results of these various efforts are presented here and in the following SEI reports:

- *Software Effort & Schedule Measurement: A Framework for Counting Staff-Hours and Reporting Schedule Information* (CMU/SEI-92-TR-21)
- *Software Size Measurement: A Framework for Counting Source Statements* (CMU/SEI-92-TR-20)
- *Software Measures and the Capability Maturity Model* (CMU/SEI-92-TR-25)
- *Software Measurement Concepts for Acquisition Program Managers* (CMU/SEI-92-TR-11)
- *A Concept Study for a National Software Engineering Database* (CMU/SEI-92-TR-23)
- *Software Measurement for DoD Systems: Recommendations for Initial Core Measures* (CMU/SEI-92-TR-19)

This report and the methods in it are outgrowths of work initiated by the Quality Subgroup of the Software Metrics Definition Working Group. Like the reports listed above, this one contains guidelines and advice from software professionals. It is not a standard, and it should not be viewed as such. Nevertheless, the framework and methods it presents give a solid basis for constructing and communicating clear definitions for two important measures that can help us plan, manage, and improve our software projects and processes.

We hope that the materials we have assembled will give you a solid foundation for making your quality measures repeatable, internally consistent, and clearly understood

by others. We also hope that some of you will take the ideas illustrated in this report and apply them to other measures, for no single set of measures can ever encompass all that we need to know about software products and processes.

Our plans at the SEI are to continue our work in software process measurement. If, as you use this report, you discover ways to improve its contents, please let us know. We are especially interested in lessons learned from operational use that will help us improve the advice we offer to others. With sufficient feedback, we may be able to refine our work or publish additional useful materials on software quality measurement.

Our point of contact for comments is

Lori Race
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Acknowledgments

The SEI measurement efforts depend on the participation of many people. We would like to thank the members of the Quality Subgroup of the Software Metrics Definition Working Group who contributed to the preparation of this report. We are indebted to them and to the organizations who sponsored the efforts to improve measurement of problems and defects. Without the contributions of these professionals, we could not have completed this task:

Frank Ackerman
Institute for Zero Defects Software

John K. Alexiou
IBM Corporation

David Babuder
Allen-Bradley Company

Mark Baker
Modern Technologies Corporation

Faye C. Budlong
Charles Stark Draper Lab., Inc.

William S. Coyne
US Air Force

Michael K. Daskalantonakis
Motorola, Inc.

Deborah A. DeToma
GTE Government Systems

Robert L. Erickson
Bellcore

Tom Grabowski
Westinghouse Electric Corporation

Joel Heidelberg
US Army Communications-Electronics Command

Sallie Henry
Virginia Tech

Sujoe Joseph
Bull HN Information Systems, Inc.

Cheryl L. Jones
Naval Underwater Systems Center

Hsien Elsa Lin
AT&T Bell Laboratories

Stan Rifkin
Master Systems, Inc.

Cindy D. Roesler
Rockwell International

Kyle Y. Rone
IBM Corporation

Dileep R. Saxena
Bell Communications Research

Lee Shaw
Westinghouse Electric Corporation

Sylvester A. Vassalo
Unisys Defense Systems

Terry A. Wilcox
DPRO-General Dynamics

A first draft of this report was presented and distributed for review at the SEI Affiliates Symposium in August 1991. A second draft was distributed to approximately 400 reviewers in June 1992. Nearly 200 comments and suggestions for improvement were returned. All have received careful consideration, and most have been incorporated or addressed through the development of new materials. We are indebted to those who took the time and care to provide so many constructive recommendations:

Jim Bartlett
Allstate Insurance Company

Barry Boehm
Defense Advanced Research Projects Agency

John Bolland
ITT Avionics Division

Nancy Cheney
Hamilton Standard, UTC

Lyle Cocking
General Dynamics

Dean Dubofsky
The MITRE Corporation

R.L. Erickson
Bellcore

Betty Falato
Federal Aviation Administration

M. Hosein Fallah
AT&T Bell Laboratories

Liz Flanagan
AT&T Bell Laboratories

Harvey Hallman
Software Engineering Institute

John Harding
Bull HN Information Systems, Inc.

Jim Hart
Software Engineering Institute

Derek Hatley
Smiths Industries

Daniel Heur
Magnavox

George Huyler
Productivity Management Group

Chris Kemerer
Massachusetts Institute of Technology

Gary Kennedy
IBM Corporation

Ed Kettler
Electronic Data Systems

Harry T. Larson
Larbridge Enterprises

George Leach
AT&T Paradyne

Donna Lindskog
University of Regina and SaskTel

Everald Mills
Seattle University

Marc Meltzer
Pratt & Whitney

Kerux-David Lee Neal
Northrop Corporation

Donald Reifer
Reifer Consultants, Inc.

Michael Robinson
Boeing Computer Services

Paul Rook
S.E.P.M.

John Salasin
Software Engineering Institute

Hal Schwartz
Fujitsu Systems of America

Brian Sharpe
Hewlett-Packard

Marie Silverthorn
Texas Instruments

Al Snow
AT&T Bell Laboratories

S. Jack Sterling
Logicon Eagle Technology, Inc.

Irene Stone
AIL Systems, Inc.

Terry Wilcox
DPRO-General Dynamics

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

William Agresti
The MITRE Corporation

Henry Block
University of Pittsburgh

David Card
Computer Sciences Corporation

Andrew Chruscicki
USAF Rome Laboratory

Samuel Conte
Purdue University

Bill Curtis
Software Engineering Institute

Joseph Dean
Tecolote Research

Stewart Fenick
US Army Communications-Electronics Command

Charles Fuller
Air Force Materiel Command

Robert Grady
Hewlett-Packard

John Harding
Bull HN Information Systems, Inc.

Frank McGarry
NASA (Goddard Space Flight Center)

John McGarry
Naval Underwater Systems Center

Watts Humphrey
Software Engineering Institute

Richard Mitchell
Naval Air Development Center

John Musa
AT&T Bell Laboratories

Alfred Peschel
TRW

Marshall Potter
Department of the Navy

Samuel Redwine
Software Productivity Consortium

Kyle Rone
IBM Corporation

Norman Schneidewind
Naval Postgraduate School

Herman Schultz
The MITRE Corporation

Seward (Ed) Smith
IBM Corporation

Robert Sulgrove
NCR Corporation

Ray Wolverton
Hughes Aircraft

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

And finally, we could not have assembled this report without the active participation and contributions from the other members of the SEI Software Process Measurement Project and the SWAP team who helped us shape these materials into forms that could be used by both industry and government practitioners:

Anita Carleton
Software Engineering Institute

John Baumert
Computer Sciences Corporation

Mary Busby
IBM Corporation

Elizabeth Bailey
Institute for Defense Analyses

Andrew Chruscicki
USAF Rome Laboratory

Judith Clapp
The MITRE Corporation

Wolfhart Goethert
Software Engineering Institute

Donald McAndrews
Software Engineering Institute

Robert Park
Software Engineering Institute

Shari Lawrence Pfleeger
The MITRE Corporation

Lori Race
Software Engineering Institute

James Rozum
Software Engineering Institute

Timothy Shimeall
Naval Postgraduate School

Patricia Van Verth
Canisius College

Software Quality Measurement: A Framework for Counting Problems and Defects

Abstract. This report presents mechanisms for describing and specifying two software measures–software problems and defects–used to understand and predict software product quality and software process efficacy. We propose a framework that integrates and gives structure to the discovery, reporting, and measurement of software problems and defects found by the primary problem and defect finding activities. Based on the framework, we identify and organize measurable attributes common to these activities. We show how to use the attributes with checklists and supporting forms to communicate the definitions and specifications for problem and defect measurements. We illustrate how the checklist and supporting forms can be used to reduce the misunderstanding of measurement results and can be applied to address the information needs of different users.

1. Introduction

1.1. Scope

This report describes a framework and supporting mechanisms that may be used to describe or specify the measurement of software problems and defects associated with a software product. It includes the following:

- A framework relating the discovery, reporting, and measurement of problems and defects.
- A principal set of measurable, orthogonal attributes for making the measurement descriptions exact and unambiguous.
- Checklists for creating unambiguous and explicit definitions or specifications of software problem and defect measurements.
- Examples of how to use the checklists to construct measurement specifications.
- Examples of measurements using various attributes of software problem reports and defects.

1.2. Objective and Audience

Our objective in this report is to provide operational methods that will help us obtain clear and consistent measurements of quality based on a variety of software problem reports and data derived by analysis and corrective actions. The report is appropriate for any

organization that wants to use software problem report data to help manage and improve its processes for acquiring, building, or maintaining software systems. The attributes and checklists described in this report are intended specifically for software project managers, software planners and analysts, software engineers, and data management specialists.

The mechanisms may be used for defining and specifying software problem and defect counts found by static or non-operational processes (e.g., design reviews or code inspections) and for dynamic or operational processes (e.g., testing or customer operation). The measurements (counts), properly qualified and described, may be used in various formats along with other software data to measure the progress of the software development project, to provide data for prediction models, and to improve the software life cycle process.

Software problem and defect measurements have direct application to estimating, planning, and tracking the various software development processes. Users within an organization are likely to have different views and purposes for using and reporting this data. Our goal is to reduce ambiguities and misunderstandings in these measures by giving organizations a basis for specifying and communicating clear definitions of problem and defect measurements.

1.3. The Software Measurement Environment

We based our work on the notion that the reader is familiar with the basic elements of a software measurement environment that is structured along the following points:

- Goals and objectives are set relative to the software product and software management process.
- Measurements are defined and selected to ascertain the degree to which the goals and objectives are being met.
- A data collection process and recording mechanisms are defined and used.
- Measurements and reports are part of a closed loop system that provides current (operational) and historical information to technical staff and management.
- Data on post-software product life measurement is retained for analysis leading to improvements for future product and process management.

These points are prerequisites for all measurement environments, and are stated here to emphasize that an understanding of them is essential for the successful use of the framework and mechanisms described in this report Each of the points above are in themselves subjects and topics of a host of papers, reports, and books generated by members of the software engineering community over the past twenty years. If you are not familiar with the implications of these points, or are not familiar with the issues, activities, and effort associated with establishing and sustaining a software measurement

system, we encourage you to become familiar with the software engineering literature on the subject before attempting to use the mechanisms discussed in this report. Recent literature that provides a comprehensive overview of software measurements includes the following:

[Baumert 92]

[Conte 86]

[Fenton 91]

[Grady 92]

[Grady 87]

[Humphrey 89]

[Musa 87]

[IEEE 88a]

Each of the above include extensive listings of papers, reports, and books that provide additional information on the many aspects of software measurement.

2. Understanding the Framework for Counting Software Problems and Defects

The purpose of this chapter is to provide a structured view of the activities involved with data collection, analysis, recording, and reporting of software problems and defects, and thereby set the framework necessary to define a common set of attributes. These attributes are used in a checklist to communicate the description and specification of problem and defect measurements.

In providing a framework for this report, we have tried to be consistent with the software measurement environment outlined in Section 1.3. We provide a rationale (albeit brief) for measuring software problems and defects in Section 2.1. This is followed by a discussion in Section 2.2 of terminology pertinent to problems and defects leading to a definition of each used in this report. Section 2.3 addresses the issues of problem data collection and reporting. The topics of problem and defect attributes and measurement definition checklists are discussed in Sections 2.4 and 2.5 respectively. Figure 2-1 shows the relationship of the software problem measurement framework to that of the measurement environment framework previously outlined.

Software Measurement Environment	**Framework for Measuring Software Problems**
Goals	Why measure problems and defects
Measurement definition	What are problems and defects
Data collection/recording	Problem finding/reporting activities
Measurements/reports	Measurement attributes and checklists

Figure 2-1 Measurement Environment Framework Relationship

2.1. Why Measure Problems and Defects

The fundamental reason for measuring software and the software process is to obtain data that helps us to better control the schedule, cost, and quality of software products. It is important to be able to consistently count and measure basic entities that are directly measurable, such as size, defects, effort, and time (schedule). Consistent measurements provide data for doing the following:

- Quantitatively expressing requirements, goals, and acceptance criteria.
- Monitoring progress and anticipating problems.

- Quantifying tradeoffs used in allocating resources.
- Predicting the software attributes for schedule, cost, and quality.

To establish and maintain control over the development and maintenance of a software product, it is important that the software developer and maintainer measure software problems and software defects found in the software product to determine the status of corrective action, to measure and improve the software development process, and to the extent possible, predict remaining defects or failure rates [Boehm 73], [Murine 83]. By measuring problems and defects, we obtain data that may be used to control software products as outlined in the following sections.

2.1.1. Quality

While it is clear that determining what truly represents software quality in the customer's view can be elusive, it is equally clear that the number and frequency of problems and defects associated with a software product are inversely proportional to the quality of the software. Software problems and defects are among the few direct measurements of software processes and products. Such measurements allow us to quantitatively describe trends in defect or problem discovery, repairs, process and product imperfections, and responsiveness to customers. Problem and defect measurements also are the basis for quantifying several significant software quality attributes, factors, and criteria—reliability, correctness, completeness, efficiency, and usability among others [IEEE 90b].

2.1.2. Cost

The amount of rework is a significant cost factor in software development and maintenance. The number of problems and defects associated with the product are direct contributors to this cost. Measurement of the problems and defects can help us to understand where and how the problems and defects occur, provide insight to methods of detection, prevention, and prediction, and keep costs under control.

2.1.3. Schedule

Even though the primary drivers for schedule are workload, people and processes, we can use measurement of problems and defects in tracking project progress, identifying process inefficiencies, and forecasting obstacles that will jeopardize schedule commitments.

2.2. Measuring Software Problems and Defects

To measure with explicitness and exactness, it is of utmost importance to clearly define the entities being measured. Since we concentrate on problems and defects in this report, we shall define their meaning and discuss the relationship they have to other terms and entities.

2.2.1. Defects

A defect is any unintended characteristic that impairs the utility or worth of an item, or any kind of shortcoming, imperfection, or deficiency. Left as is, this definition of a defect, while correct, needs to be more definitive to be of help in the software measurement sense. If we further define a defect to be an inconsistency with its specification [IEEE 88a], the implication is that the reference specification cannot have any defects. Since we know this is not the case, we will try another approach.

We will define a *software defect* to be any flaw or imperfection in a software work product or software process.

A *software work product* is any artifact created as part of the software process including computer programs, plans procedures, and associated documentation and data [CMU/SEI 91]. A *software process* is a set of activities, methods, practices, and transformations that people use to develop and maintain software work products [CMU/SEI 91].

This definition of a software defect covers a wide span of possibilities and does not eliminate software artifacts that we know from experience to contain defects. It does suggest the need for standards, rules, or conventions that establish the type and criticality of the defect (Grady proposes a model for defect nomenclature and classification in [Grady 92]).

A software defect is a manifestation of a human (software producer) *mistake;* however, not all human mistakes are defects, nor are all defects the result of human mistakes.

When found in executable code, a defect is frequently referred to as a *fault* or a bug. A fault is an incorrect program step, process, or data definition in a computer program. Faults are defects that have persisted in software until the software is executable. In this report, we will use the term *defect* to include faults, and only use the term fault when it is significant to refer to a defect in executable code.

2.2.2. Problems

The definition of a software problem has typically been associated with that of a customer identifying a malfunction of the program in some way. However, while this may be correct as far as it goes, the notion of a software problem goes beyond that of an

unhappy customer. Many terms are used for problem reports throughout the software community: incident report, customer service request, trouble report, inspection report, error report, defect report, failure report, test incident, etc. In a generic sense, they all stem from a unsatisfactory encounter with the software by people. Software problems are human events. The encounter may be with an operational system *(dynamic)*, or it may be an encounter with a program listing (a *static* encounter.) Given the range of possibilities, we will define software problem as follows:

A *software problem* is a human encounter with software that causes difficulty, doubt, or uncertainty in the use or examination of the software.

In a dynamic (operational) environment, some problems may be caused by *failures.* A failure is the departure of software operation from requirements [Musa 87]. A software failure must occur during execution of a program. Software failures are caused by faults, that is, defects found in executable code (the same kind of faults discussed in the previous section as persistent defects).

In a static (non-operational) environment, such as a code inspection, some problems may be caused by defects. In both dynamic and static environments, problems also may be caused by misunderstanding, misuse, or a number of other factors that are not related to the software product being used or examined.

2.3. Problem and Defect Finding Activities

To establish a software measurement environment, the software organization must define a data collection process and recording media. Software problem reports typically are the vehicles used to collect data about problems and defects. It is worthwhile to note that the data that is assembled as part of the problem analysis and correction process is precisely the same data that characterizes or gives attribute values to the problems and defects we wish to measure. Although this process facilitates the data collection aspects of software problem and defects measurement, the variety of finding activities and related problem reports make it difficult to communicate clearly and precisely when we define or specify problem and defect measurements.

The primary points of origin for problem reports are activities whose function is to find problems using a wide variety of problem discovery or detection methodologies, including using the software product (see Figure 2-2). During software development, these activities would include design and code inspections, various formal reviews, and all testing activities. In addition, activities such as planning, designing, technical writing, and coding are also sources of problems reports. Technical staff engaged in these activities frequently will encounter what appears to be an defect in a software artifact on which they are dependent to complete their work and will generate a problem report. Following product development, the software product customer is another source of problem reports.

ACTIVITIES	Find Problems In:
Product synthesis	Requirement specs Design specs Source code User publications Test procedures
Inspections	Requirement specs Design specs Source code User publications Test procedures
Formal reviews	Requirement specs Design specs Implementation Installation
Testing	Modules Components Products Systems User publications Installation procedures
Customer service	Installation procedures Operating procedures Maintenance updates Support documents

Figure 2-2 Problem and Defect Finding Activities

To facilitate the communication aspect, we have identified five major finding activities:

- Software product synthesis[1]
- Inspections
- Formal reviews
- Testing
- Customer service

This classification retains the functional identity of the finding activities without relying on a specific development process model, and therefore becomes a communicative attribute for problem or defect measurement.

Problem reports give rise to additional measurement communication issues. Problem reports generated by the finding activities are typically tuned to the needs of the activity and vary in content and format. For example, inspection reports, requirement review reports, test reports, and customer service reports carry data not required by or available

[1] By *product synthesis* we mean the activity of planning creating and documenting the requirements, design, code, user publications, and other software artifacts that constitute a software product. This would exclude all types of peer reviews.

to the others. The problems are recorded and reported at different points in time (e.g., before and after configuration management control), in batches or continuously, by different organizations, by people with varying degrees of understanding of the software product. Often, the data is captured in separate databases or record-keeping mechanisms. The problem and defect attributes and attribute values described in Sections 2.4 and 3.1 bridge these differences and provide a consistent basis for communicating (Figure 2-3).

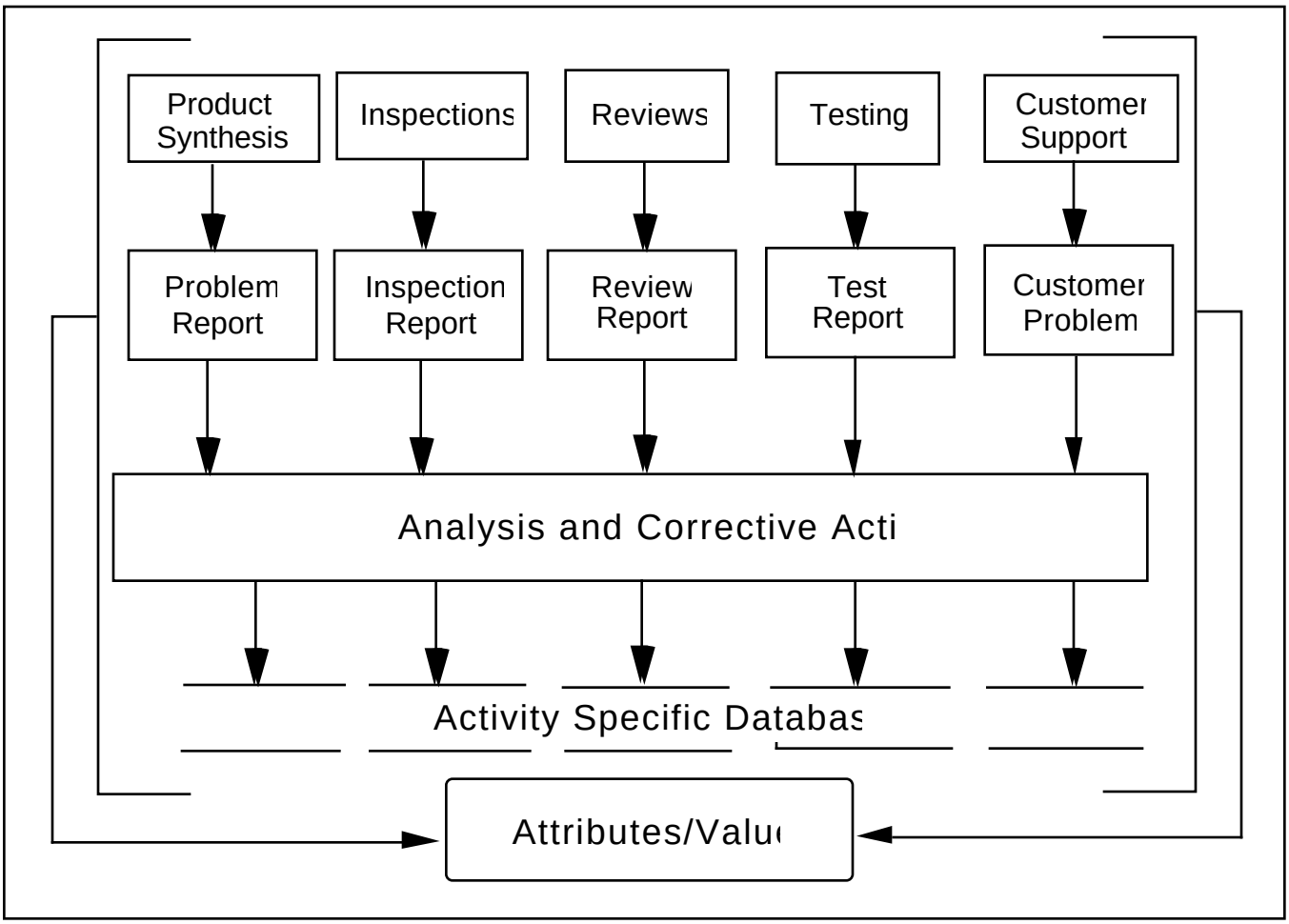

Figure 2-3 Problem and Defect Data Collection and Recording

2.4. Problem and Defect Attributes

In spite of the variances in the way software problems may be reported and recorded, there are remarkable similarities among reports, particularly if the organization or activity-specific data is removed. There are several ways of categorizing this similarity. The approach we use to arrive at a set of attributes and attribute values that encompass the

various problem reports is to apply the "who, what, why, when, where, and how" questions in the context of a software measurement framework [Fenton 91].

In developing the attribute values, we are careful to ensure they are mutually exclusive; that is, any given problem may have one, and only one, value for each attribute. We also work to ensure that the values for each attribute are exhaustive so that inconsistent or erroneous counts do not occur.

We have identified the following attributes with the intention of transcending the variations in problem reports generated by the finding activities (Section 2.3). Each of these attributes and its values are fully described in Chapter 3. Several of the attributes have values that require special consideration when used to specify a problem count. This is discussed later in the report (Section 2.6, Chapter 3, and Chapter 4).

These attributes provide a basis for communicating, descriptively or prescriptively, the meaning of problem and defect measurements:

- **Identification:** What software product or software work product is involved?
- **Finding Activity:** What activity discovered the problem or defect?
- **Finding Mode:** How was the problem or defect found?
- **Criticality:** How critical or severe is the problem or defect?
- **Problem Status:** What work needs to be done to dispose of the problem?
- **Problem Type:** What is the nature of the problem? If a defect, what kind?
- **Uniqueness:** What is the similarity to previous problems or defects?
- **Urgency:** What urgency or priority has been assigned?
- **Environment:** Where was the problem discovered?
- **Timing:** When was the problem reported? When was it discovered? When was it corrected?
- **Originator:** Who reported the problem?
- **Defects Found In:** What software artifacts caused or contain the defect?
- **Changes Made To:** What software artifacts were changed to correct the defect?
- **Related Changes:** What are the prerequisite changes?
- **Projected Availability:** When are changes expected?
- **Released/Shipped:** What configuration level contains the changes?
- **Applied:** When was the change made to the baseline configuration?

- **Approved By:** Who approved the resolution of the problem?
- **Accepted By:** Who accepted the problem resolution?

While the attributes used in the checklists are extensive, they are not exhaustive. For example, there is no attribute that identifies the different types of errors (mistakes) that have caused defects. Examples of software error classifications are found in [Grady 92] and [Schneidewind 79] among others. Your organization may develop its own error classifications based on its own particular situation and need.

You may find that you wish to identify additional attributes. If you exercise this option, take care to rephrase existing descriptions so overlapping does not occur. If values are not mutually exclusive, then observed results can be assigned to more than one category by different observers. If this happens, it may result in inconsistent and erroneous counts, including double counting.

2.5. Measurement Communication with Checklists

Our primary goal is to provide a basis for clearly communicating the definition and specification of problem and defect measurements. Two criteria guide us:

- *Communication*: If someone generates problem or defect counts with our methods, will others know precisely what has been measured and what has been included and excluded?
- *Repeatability*: Would someone else be able to repeat the measurement and get the same results?

These criteria have led us to the use of checklists as the mechanism to construct problem and defect measurement definitions and specifications. By using a checklist to organize the attributes and attribute values, the process for constructing problem and defects definitions is methodical and straightforward.

The checklist can also be used to request and specify the attribute value counts and data arrays of defect counts that we would like to have reported to us. We also design and use other *supporting forms* (Section 2.6) to record special rules and clarification or to report request instructions that are not amenable to checklist treatment.

After listing the principal attributes and arranging them in a checklist, the process for constructing a definition becomes a matter of checking off the attribute values we wish to include and excluding all others.

Figure 2-4 is an example of how the first page of the Problem Count Definition Checklist might look for one particular definition of a problem measurement. The attributes are shown as bold-faced section headings, and these headings are followed by the lists of values that the attributes take on. For example, **Problem Type** is an attribute, and the

values for this attribute are requirement defect, design defect, code defect, operational document defect, test case defect, other work product defect, hardware problems, etc. All the software defect values have been collected have been grouped under a generic class called Software Defect. Those that are not software defects are grouped under Other Problems, or Undetermined. The attributes and values used in Figure 2-4 are discussed in detail in Chapter 3.

Immediately next to the Attribute/Values column are the two columns used to include and exclude attribute values. The two columns on the far right, Value Count and Array Count, are used to specify individual attribute counts.

The checklist in Figure 2-4 helps provide a structured approach for dealing with the details that we must resolve to reduce misunderstandings when collecting and communicating measures of problems and defects. With such a checklist, we can address issues one at a time by designating the elements that people want included in measurement results. At the same time, designating the elements to be excluded directs attention to actions that must be taken to avoid contaminating measurement results with unwanted elements. A side benefit is that the checklist can be used also to specify detailed data elements for which individual reports are wanted.

Problem Count Definition Checklist-1

Software Product ID [Example V1 R1]

Definition Identifier: [Problem Count A] **Definition Date [01/02 /92]**

Attributes/Values	**Definition [ ]**		**Specification [X]**	
Problem Status	**Include**	**Exclude**	**Value Count**	**Array Count**
Open	✔		✔	
Recognized				
Evaluated				✔
Resolved				✔
Closed	✔		✔	
Problem Type	**Include**	**Exclude**	**Value Count**	**Array Count**
Software defect				
Requirements defect	✔		✔	
Design defect	✔		✔	
Code defect	✔		✔	
Operational document defect	✔		✔	
Test case defect		✔		
Other work product defect		✔		
Other problems				
Hardware problem		✔		
Operating system problem		✔		
User mistake		✔		
Operations mistake		✔		
New requirement/enhancement		✔		
Undetermined				
Not repeatable/Cause unknown		✔		
Value not identified		✔		
Uniqueness	**Include**	**Exclude**	**Value Count**	**Array Count**
Original	✔			
Duplicate		✔	✔	
Value not identifed		✔		
Criticality	**Include**	**Exclude**	**Value Count**	**Array Count**
1st level (most critical)	✔			✔
2nd level	✔			✔
3rd level	✔			✔
4th level	✔			✔
5th level	✔			✔
Value not identifie		✔		
Urgency	**Include**	**Exclude**	**Value Count**	**Array Count**
1st (most urgent)	✔			
2nd	✔			
3rd	✔			
4th	✔			
Value not identified		✔		

Figure 2-4 Problem Count Definition Checklist

Later (in Chapters 4 and 5), we will explain how to use the checklist and supporting forms and will include examples of definitions we have constructed with this checklist. Readers should keep in mind that there are no universal best choices when completing a definition checklist. Instead, each choice should be made so as to serve an organization's overall measurement needs. This may involve tradeoffs between how the measurement results will be used and the difficulties associated with applying the definition to collect data from real software projects.

In practice, a checklist turns out to be a very flexible tool. For example, an organization may want to merge results from several values into a new value. Moreover, some measures exist that some organizations may want to record and aggregate (duplicate problems, for example) but not include in a total size measure. All these options can be addressed with the checklist.

With this in mind, we have found it useful to provide blank lines in checklists so that you can add other attribute values to meet local needs. When you exercise this flexibility to list additional values for inclusion or exclusion, you should take care to rephrase the labels for existing values so that overlaps do not occur.

The importance of ensuring that values within attributes are non-overlapping cannot be overstated. If values are not mutually exclusive, then observed results can get assigned to more than one category. If this happens, and if totals (such as total defects) are computed by adding across all values within an attribute that are designated for inclusion, double counting can occur. Reported results will then be larger than they really are. In the same vein, if overlaps exist between two values, and if one of the values is included within a definition while other is not, those who collect the data will not know what to do with observations that fall into both classes.

2.6. Supporting Forms

Sometimes definition checklists cannot record all the information that must be conveyed to avoid ambiguities and misunderstandings. In these instances, we need specialized forms to describe and communicate the additional measurement rules or description. In this report, we provide two such forms, the Problem Status Definition Form and the Problem Count Request Form.

Problem status is an important attribute by which problems are measured. The criteria for establishing problem status is determined by the existence of data that reflects the progress made toward resolving the problem. We can use the Problem Status Definition Form to define the meaning of the several problem states by defining the criteria for reaching each state in terms of the problem data, i.e., the problem attribute values. A description of the Problem Status attribute is in Chapter 3 and an explanation of the Problem Status Definition Form is in Chapter 5.

The Problem Count Request Form is used in conjunction with the Problem Count Definition form to supplement the measurement definition. The Problem Count Request Form is a somewhat modified checklist used to describe or specify attribute values that are literals (dates, customer IDs, etc.). This form is used to further describe or specify a problem or defect measurement in terms of dates, specific software artifacts, or specific hardware or software configuration. An example of the form is shown in Figure 2-5. Use of the form is discussed in Chapter 4.

The combination of a completed checklist and its supporting forms becomes a vehicle for communicating the meaning of measurement results to others, both within and outside the originating organization.

We make no attempt to describe a report form or format for the various problem counts for several reasons. The report format will vary significantly depending on the problem count definition. The report could be a single number or it could be an array of several thousand data elements. Additionally, much of the data will be more useful if graphed over a time scale or distribution form, especially if the count definition yields several hundred or more numbers. There are many computer based programs available to perform the transformation from numerical form to graphical form, each with their own data input format needs. Such a program will be a key factor in determining report format.

Problem Count Request Form

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

Date of Request: [6-15-92] **Requester's Name or ID: [I. M. Able]**

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

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

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

Report Count By:	Attribute Sort Order	Select Value, Sort Order	Special Instructions or Comments
Originator			
Site ID			
Customer ID			
User ID			
Contractor ID			
Specific ID(s) list			
Environment		Sort Order	
Hardware config ID			
Software config ID			
System config ID			
Test proc ID			
Specific ID(s) list			

Defects Found In:				
Select a configuration	Type of Artifact			
component level:	Requirement	Design	Code	User Document
Product (CSCI)				
Component (CSC)	✔	✔	✔	✔
Module (CSU)				
Specific (list)				

Changes Made To:				
Select a configuration	Type of Artifact			
component level	Requirement	Design	Code	User Document
Product (CSCI)				
Component (CSC)				
Module (CSU)				
Specific (list)				

Figure 2-5 Problem Count Request Form

2.8. Framework Summary

We have developed a frame of reference for communicating descriptions or specifications of problem and defect measurements. We defined software problems to include virtually any unsatisfactory human encounter with software, and defined software defects to be any imperfection of the software.

The issue of collection and recording problem and defect data is addressed by using data created by existing problem management processes (i.e., finding activities, and analysis and corrective action activities) in the course of performing their duties (Figure 2-6).

Problem and defect attributes that span the various finding activities are identified and are used as the basis for measurements. Checklist and supporting forms incorporate and organize the attributes to communicate problem and defect descriptions and specifications.

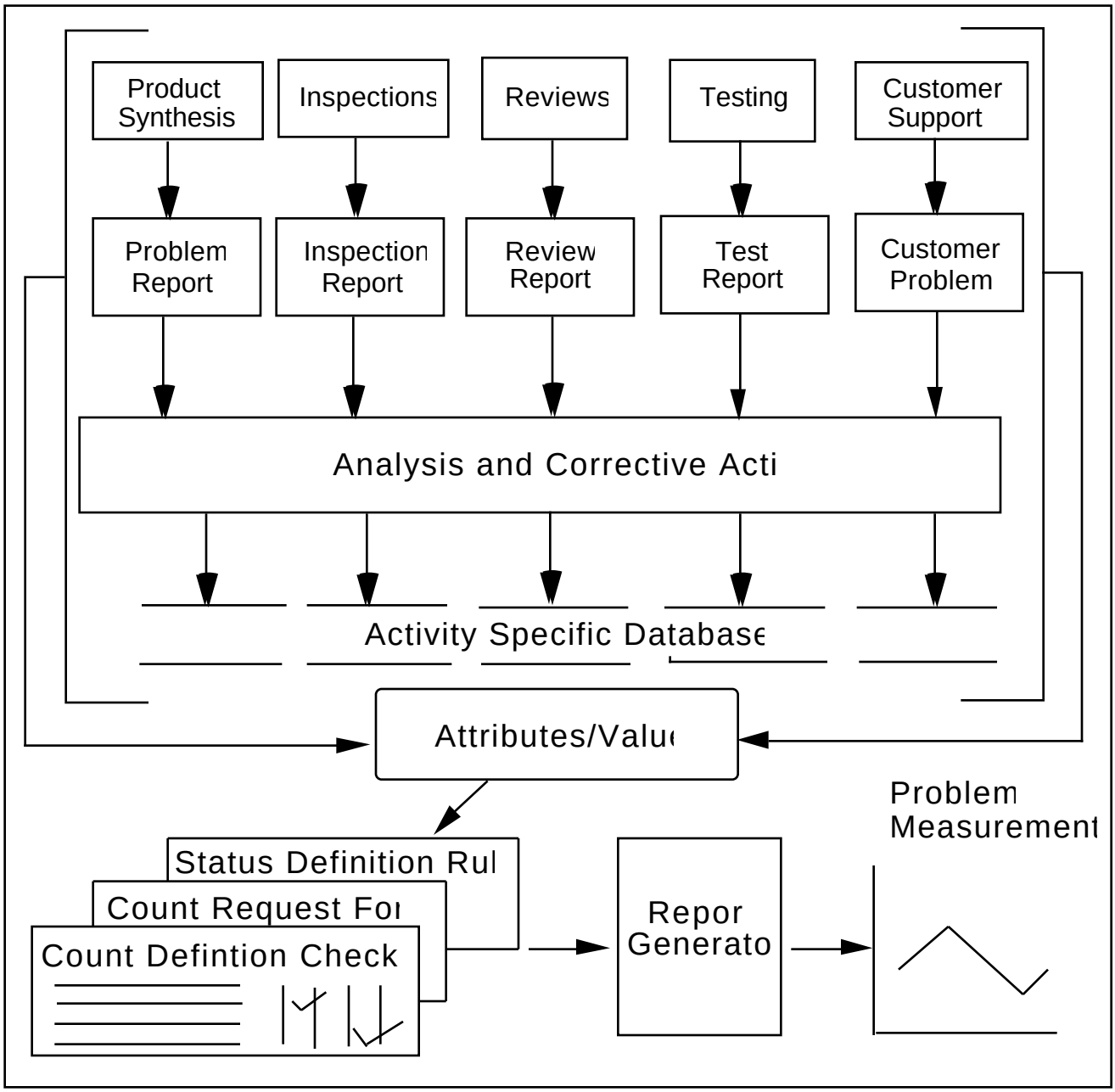

Figure 2-6 Framework Overview

3. Using the Problem and Defect Attributes

In this section, we define and illustrate the attributes and attribute values used in the checklist and supporting forms. The discussion sequence follows that of the Problem Count Definition Checklist and the Problem Count Request Form so that information about the attributes and their values may be readily located. (Appendix C contains copies of our forms, which you may use as reproduction masters.) Bold type is used to highlight the attributes. More specific instructions for using the checklist and other forms appear in Chapters 4 and 5.

Not all software organizations will use the same terms as are presented in the definition of the attributes. We have tried to use terms and names that are recognized by the *IEEE Standard Glossary of Software Engineering Terminology* [IEEE 90a], or describe them in sufficient detail so that their meaning is clear. The sequence of attributes is:

- **Identification**
- **Problem Status**
- **Problem Type**
- **Uniqueness**
- **Criticality**
- **Urgency**
- **Finding Activity**
- **Finding Mode**
- **Date/Time of Occurrence**
- **Problem Status Date**
- **Originator**
- **Environment**
- **Defects Found In**
- **Changes Made To**
- **Related Changes**
- **Projected Availability**
- **Released/Shipped**
- **Applied**
- **Approved By**
- **Accepted By**

3.1. Identification Attributes

Problem ID: This attribute serves to uniquely identify each problem for reference purposes.

Product ID: This attribute identifies the software product to which the problem refers. It should include the version and release ID for released products, or the build ID for products under development. This value also may be used to identify the product subsets or non-product units (tools or prototypes) used to produce the product.

3.2. Problem Status

The status attribute refers to a point in the problem analysis and corrective action process that has been defined to have met some criteria. See Chapter 5 for an explanation of the Problem Status Definition Form. The problem status is of prime interest to the software developer since it reveals information about the progress to resolve and dispose of the reported problems. Coupled with information about testing or inspections, problem status data gives the software manager a basis for projecting time and effort needed to complete the project on schedule. Because problem status is dependent on the amount of information known about the problem, it is important that we define and understand the possible states and the criteria used to change from one state to another (See Figure 3-1).

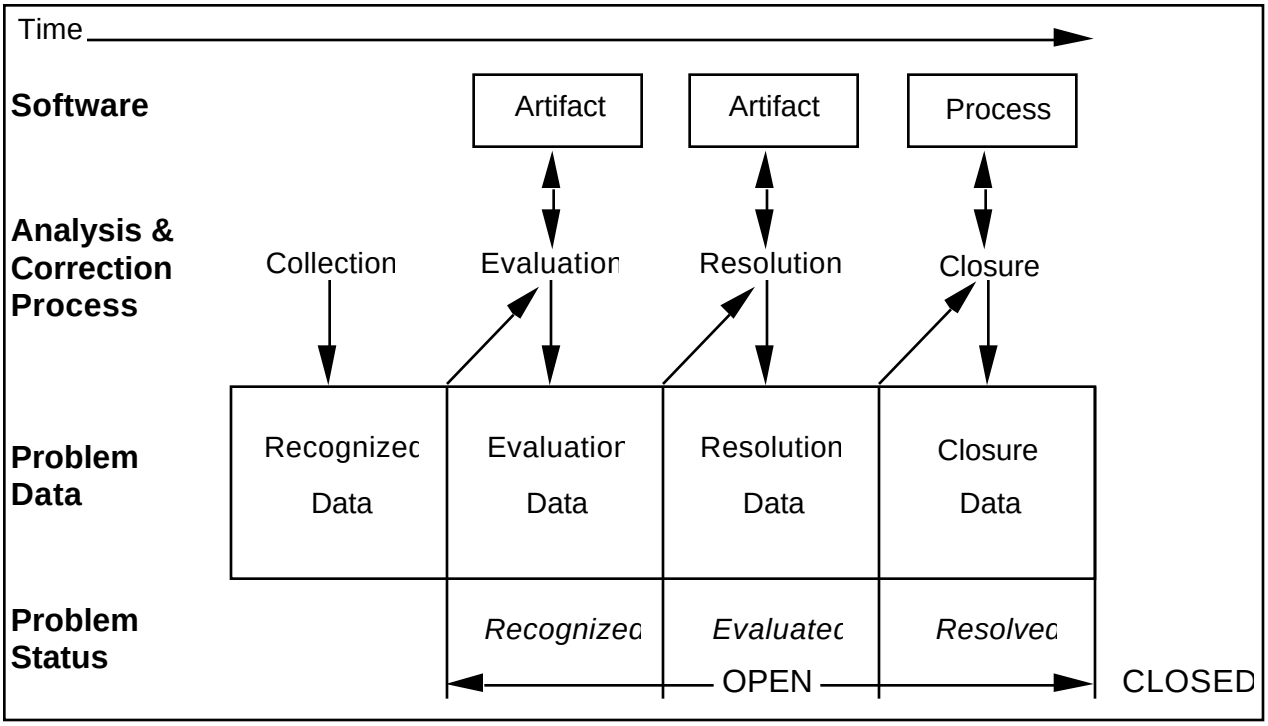

Figure 3-1 A Generic Problem Management System

The status of the problems is a matter of interest to the software developer and maintainer because it helps them determine the progress being made to resolve or dispose of the totality of problems reported. The recognition or opening of a problem is based on the existence of data describing the event. As the investigative work proceeds, more data is collected about the problem, including the information that is required to satisfy the issues raised by the problem. It is the existence of this data that is used as a set of criteria to satisfy moving from status = Open to status = Closed.

We decompose the analysis and corrective action process into subset states to give a more detailed understanding of the problem solving process. Measurements of the numbers of problem reports in the various problem states, shown with respect to time, will inform the manager of the rate the project is progressing towards a goal. The rate of problem arrival and the time it takes to process a problem report through closure addresses the efficacy of the problem management process.

In practice, many projects may want to greatly expand the number of subset states in both the Open and Closed status. The specifics of the problem management process will determine the extent of the subset states. The number of distinct activities, reviews, and organizations are a key factor in determining the number of subset states. Since this varies greatly from project to project and organization to organization, we have included three generic subset states for illustrative purposes.

Below are the attribute values for **Problem Status**:

Open: This term means that the problem is recognized and some level of investigation and action will be undertaken to resolve it. The Open value is often decomposed into substates to obtain a detailed understanding of the work remaining in the analysis and correction process to close the problem. The status categories used in this document were selected based on a generic description of the problem management tasks (Figure 3-1). The status categories are **Recognized**, **Evaluated**, and **Resolved** and are described as follows:

Recognized: Sufficient data has been collected to permit an evaluation of the problem to be made. This implies that the data has been verified as correct as well. This generally means the set of data that one can reasonably expect the problem originator to provide. Note that in the general case, the problem originator can vary from a skilled software engineer to an end-user who is unfamiliar with the software except, for example, as a data entry screen. These two extremes point out the need to be able to define the status criteria for Recognized (and hence Open) in terms of the problem data. The minimum criteria might be the Originator ID and the Product Name. This would give an analyst enough information to request additional data if it was not included in the initial problem report.

Evaluated: Sufficient data has been collected by investigation of the reported problem and the various software artifacts to at least determine the problem type.

Again, depending on the software organization, the amount of data required to satisfy the criteria for this state may vary significantly.

Resolved: The problem has been reported and evaluated, and sufficient information is available to satisfy the rules for resolution. Each organization will have its own prescribed set of processes or activities relative to resolution. This may include the proposed change, a designed and tested change, change verification with the originator, change control approval, or a causal analysis of the problem. This process and the resultant data are a function of the organization's software management process.

Closed: This term means the investigation is complete and the action required to resolve the problem has been proposed, accepted, and completed to the satisfaction of all concerned. In some cases, a problem report will be recognized as invalid as part of the recognition process and be closed immediately.

The software manager may also be interested in the number of problems that have been recognized but have not yet been evaluated. These problems are somewhere in transition between the Recognized state and the Evaluated state. We can refer to them as unevaluated. Since the problems change from the Recognized to the Evaluated state once evaluated, problems in the Recognized state are equivalent to being unevaluated.

A problem cannot be in more than one state at any point in time. However, since the problem state will change over time, one must *always* specify a time when measuring the problem status.

Problem Status	**Include**	**Exclude**	**Value Count**	**Array Count**
Open				
Recognized				
Evaluated				
Resolved				
Closed				

Figure 3-2 Problem Status Attribute

3.3. Problem Type

The **Problem Type** attribute is used to assign a value to the problem that will facilitate the evaluation and resolution of the problems reported. The problems may be encountered over a relatively long period of time. Problems are unsatisfactory encounters with the software; consequently, some of the problems reported are not software failures or software defects, and may not even be software product problems. The **Problem Type** attribute values are used to classify the problems into one of several categories to facilitate the problem resolution.

There are many ways to structure problem types. In this report, we use a structure that is oriented toward a software development and software service organization. The problem

type structure does not address hardware problem types, networking error types, human factor interface types, or any other problem types from other disciplines. We have arbitrarily divided the problems in to three subtypes–Software Defects, Other Problems, and Undetermined. This is only for the sake of convenience. The values used with the Problem Type attribute exist with or without this division. In practice, each problem data record will contain one, and only one, of the problem type values.

Software defect: This subtype includes all software defects that have been encountered or discovered by examination or operation of the software product. Possible values in this subtype are these:

- **Requirements defect:** A mistake made in the definition or specification of the customer needs for a software product. This includes defects found in functional specifications; interface, design, and test requirements; and specified standards.
- **Design defect:** A mistake made in the design of a software product. This includes defects found in functional descriptions, interfaces, control logic, data structures, error checking, and standards.
- **Code defect:** A mistake made in the implementation or coding of a program. This includes defects found in program logic, interface handling, data definitions, computation, and standards.
- **Document defect:** A mistake made in a software product publication. This does not include mistakes made to requirements, design, or coding documents.
- **Test case defect:** A mistake in the test case causes the software product to give an unexpected result.
- **Other work product defect:** Defects found in software artifacts that are used to support the development or maintenance of a software product. This includes test tools, compilers, configuration libraries, and other computer-aided software engineering tools.

Other problems: This subtype includes those problems that contain either no evidence that a software defect exists or contain evidence that some other factor or reason is responsible for the problem. It would not be atypical for many software organizations to consider problems that fall into this category as Closed almost immediately as evaluated with an “out of scope,” or similar closing code. Possible values in this subtype are:

Hardware problem: A problem due to a hardware malfunction that the software does not, or cannot, provide fault tolerant support.

Operating system problem: A problem that the operating system in use has responsibility for creating or managing. (If the software product is an operating system, this value should move to the Software defect category.)

User error: A problem due to a user misunderstanding or incorrect use of the software.

Operations error: A problem caused by an error made by the computer system operational staff.

New requirement/enhancement: A problem that describes a new requirement or functional enhancement that is outside the scope of the software product baseline requirements.

Undetermined problem: The status of the problem has not been determined. Values for this subtype are:

Not repeatable/Cause unknown: The information provided with the problem description or available to the evaluator is not sufficient to assign a problem type to the problem.

Value not identified: The problem has not been evaluated.

Problem Type	Include	Exclude	Value Count	Array Count
Software defect				
Requirements defect				
Design defect				
Code defect				
Operational document defect				
Test case defect				
Other work product defect				
Other problems				
Hardware problem				
Operating system problem				
User mistake				
Operations mistake				
New requirement/enhancement				
Undetermined				
Not repeatable/cause unknown				
Value not identified				

Figure 3-3 Problem Type Attribute and Values

3.4. Uniqueness

This attribute differentiates between a unique problem or defect and a duplicate. The possible values are:

Duplicate: The problem or defect has been previously discovered.

Original: The problem or defect has not been previously reported or discovered.

Value not identified: An evaluation has not been made.

We must take care to relate the uniqueness of a problem or defect to a particular configuration level or product release level. This attribute is critical for measuring reliability growth, or for verifying that a previously corrected defect was completely corrected or only partially corrected. (See the **Defects Found In** attribute)

Uniqueness	Include	Exclude	Value Count	Array Count
Original				
Duplicate				
Value not identifed				

Figure 3-4 Uniqueness Attribute and Values

3.5. Criticality

Criticality is a measure of the disruption a problem gives users when they encounter it. The value given to criticality is normally provided by the problem originator or originating organization. Criticality is customarily measured with several levels, the most critical being a catastrophic disruption, and the least critical being at the annoyance level.

Criticality	Include	Exclude	Value Count	Array Count
1st level (most critical)				
2nd level				
3rd level				
4th level				
5th level				
Value not identified				

Figure 3-5 Criticality Attribute

3.6. Urgency

Urgency is the degree of importance that the evaluation, resolution, and closure of a problem is given by the organization charged with executing the problem management process. The value is assigned by the supplier or developer, who should or must consider the criticality of the problem as expressed by the problem originator. Urgency determines the order in which problems are evaluated, resolved, and closed.

Urgency	Include	Exclude	Value Count	Array Count
1st (most urgent)				
2nd				
3rd				
4th				
Value not identified				

Figure 3-6 Urgency Attribute

3.7. Finding Activity

This attribute refers to the activity, process, or operation taking place when the problem was encountered. Rather than use the program development phases or stages to describe the activity, we have chosen to use the activities implicit in software development regardless of the development process model in use. We have identified five categories of activities, each unique in terms of the process used to detect problems, or defects. This attribute is used to distinguish between defects detected by the various activities. Several of the software reliability measures require such distinction. The efficiency of defect detection in each activity may be measured and used to focus on areas that appear to require process management attention.

Finding Activity	Include	Exclude	Value Count	Array Count
Synthesis of				
Design				
Code				
Test procedure				
User publications				
Inspections of				
Requirements				
Preliminary design				
Detailed design				
Operational documentation				
Software module (CSU)				
Test procedures				
Formal reviews of				
Plans				
Requirements				
Preliminary design				
Critical design				
Test readiness				
Formal qualification				
Testing				
Planning				
Module (CSU)				
Component (CSC)				
Configuration item (CSCI)				
Integrate and test				
Independent verif. and valid.				
System				
Test and evaluate				
Acceptance				
Customer Support				
Production/Deployment				
Installation				
Operation				
Value not identified				

Figure 3-7 Finding Activity Attribute and Values

3.8. Finding Mode

This attribute is used to identify whether the problem or defect was discovered in an operational environment or in a non-operational environment. The values for this attribute are the following:

Dynamic: This value identifies a problem or defect that is found during operation or execution of the computer program. If the problem type is a software defect, it is a fault by definition and the problem is due to a software failure.

Static: This value identifies a problem or defect that is found in a non-operational environment. Problems or defects found in this environment cannot be due to a failure or fault. Problems or defects found in this mode would typically be found by formal reviews, software inspections, or other activities that do not involve executing the software.

Value not identified: An evaluation has not been made.

This attribute is essential to defining problem counts that are due to software failures.

Finding Mode				
Static (non-operational)				
Dynamic (operational)				
Value not identified				

Figure 3-8 Finding Mode Attribute and Values

3.9. Date/Time of Occurrence

The Problem Count Request Form is used to identify the range and environment constraints desired for the measurements. The Date/Time of Occurrence attribute first appears on this form and describes the date and time of day or relative time at which a failure occurred. This data is of primary importance in establishing and predicting software reliability, failure rates, and numerous other time-related or dynamic measurements. We also need this data to reproduce or analyze problems that are keyed to the time of day (for example, power failures and workloads).

3.10. Problem Status Dates

These attributes refer to the date on which the problem report was received or logged in the problem database or when the problem changed status. These attributes are used to determine status, problem age, and problem arrival rate. This information is also of primary importance in product readiness models to determine when the product is ready for acceptance testing or delivery.

Date Opened: Date the problem was reported and recognized.

Date Closed: Date the problem met the criteria established for closing the problem.

Date Assigned for Evaluation

Date Assigned for Resolution

The Problem Count Request Form should be used to define how the selected values should be treated.

3.11. Originator

This attribute provides the information needed by the problem analyst to determine the originating person, organization, or site. This is useful in determining if a type of problem is unique to a particular source or in eliminating a problem type based on the source environment. Occasionally, sites will use software in a manner not envisioned by the original developers. This attribute allows analysts to identify such sites and further investigate how the software is being used. This is a prerequisite if the analyst is required to respond to the originator regarding the problem status, or if the analyst finds it necessary to obtain more information about the problem from the originator. This attribute has the values listed in Figure 3.9.

The Problem Count Request Form should be used to define how the selected values should be treated.

Report Count By:	Attribute Sort Order	Select Value, Sort Order
Originator		
Site ID		
Customer ID		
User ID		
Contractor ID		
Specific ID(s) list		

Figure 3-9 Originator Attribute and Values

The attribute is generally used by identifying a range or limit that is meaningful in terms of the count. For example, if we wished to count the number of failures encountered at each site, we would count the failures for each Site ID. This would be expressed in the Problem Count Definition Checklist by including a defect value under the **Problem Type**, the dynamic value under **Finding Mode**, and the customer operation value under the **Finding Activity**. In this case, the range or limit of the count is the Site ID, all of them. If the count pertained to a single site, the range would be the specific Site ID. The Problem Count Request Form may be used to specify the way in which the counts are aggregated or used to qualify the count.

3.12. Environment

This attribute provides information needed by the problem analyst to determine if a problem is uniquely related to the computer system, operating system, or operational environment, or if a particular environment tends to generate an abnormally large number of problems compared to other environments. This information is essential if the problem must be reproduced by the evaluator to determine failure or fault information. The environment attribute also can be used to identify the test case or test evaluation procedure in use when the problem occurred. The attribute has the values shown in Figure 3-10.

The Problem Count Request Form should be used to define how the selected values should be treated.

Report Count By:	Attribute Sort Order	Select Value, Sort Order
Environment		
Hardware config. ID		
Software config. ID		
System config. ID		
Test proc. ID		
Specific ID(s) list		

Figure 3-10 The Environment Attribute and Values

3.13. Defects Found In

This attribute enables us to identify the software unit(s) containing defects causing a problem. This information is particularly useful to identify software units prone to defects, or to demonstrate reliability growth from one configuration level or release to another. In addition, the attribute helps quantify the tightness or looseness of logical binding among various software units at the same configuration level. The Problem Count Request Form is used to identify the type of software unit and the configuration level to be counted.

Defects Found In: Select a configuration component level	Type of Artifact			
	Requirement	Design	Code	User Document
Product (CSCI)				
Component (CSC)				
Module (CSU)				
Specific (list)				

Figure 3-11 Defects Found In Attribute

3.14. Changes Made To

We use this attribute to identify the software unit(s) changed to resolve the discovered problem or defect. This information is particularly useful to identify software units prone to changes due to defects. The Problem Count Request Form is used to identify the type of software unit and the configuration level to be counted.

Changes Made To: Select a configuratior component level	Type of Artifact			
	Requiremen	Design	Code	User Document
Product (CSCI)				
Component (CSC)				
Module (CSU)				
Specific (list)				

Figure 3-12 Changes Made To Attribute

3.15. Related Changes

This attribute and those found in Sections 3.16–3.20 appear in the Problem Status Definition form. Related changes is a list of changed software artifacts required to be applied to the product before or at the same time as changes resolving the problem in question.

3.16. Projected Availability

Date Available: The date the product fix is committed to be made available.

Release/Build #: The Release or Build ID of the product fix is committed to be available.

3.17. Released/Shipped

Date Released/Shipped: The date the product fix is released or shipped.

Release/Build #: The Release or Build ID in which the product fix is included.

3.18. Applied

Date Applied: The date the product fix was applied to the problem-originating site or installation.

Release/Build #: The Release or Build ID in which the product fix was applied.

3.19. Approved By

Software Project Management: Indication that the product fix has been approved by appropriate project management.

Other: We have allowed space for you to add other required approvals.

3.20. Accepted By

Software Project Management: Indication that the product fix has been accepted by appropriate project management.

Problem Originator: Indication that the product fix has been accepted by the originating site or installation.

4. Using the Problem Count Definition Checklist

The Problem Count Definition Checklist is used to define and select the attributes and attribute values that must be counted to implement the problem and defect measurements selected by the software organization. Figure 4-1 is an example of how a completed Problem Count Definition Checklist might look for one particular definition of problem measurement.

We enter the software Product ID and Problem Definition ID in the spaces provided in the checklist header, along with the date the definition was constructed.

The far left column, titled Attribute/Values, lists seven attributes (on two pages) and the attribute values (defined and explained in Chapter 3). You may use the Definition box or the Specification box to indicate whether the purpose is to describe an existing measurement or specify a required measurement.

We use the column entitled Include, immediately adjacent to the Attribute/Value column, to indicate that only the problems having the attribute values checked were counted or are to be counted. The column titled Exclude is used to indicate that problems having the attribute values checked were not or are not to be counted. A problem or defect count is completely defined (within the range of the attributes) by entering a check mark in either the Include or Exclude column for each attribute value. The definition of the count is defined by the *union* of the checked Include attribute values for a given attribute and their *intersection* with the remaining checked attributes.

The checks in the Exclude column serve to indicate that if a value is not included, it must be excluded, leaving no ambiguity about the definition of the count. (Note that the **Problem Status** value Open has three substates listed. These substates cannot be selected in the Include or Exclude columns. They may be selected in the Value Count and Array Count columns described below.)

We use the Value Count and the Array Count columns to identify or specify counts of a specific attribute value, or a multidimensional array of attribute value counts. For example, a check in the Value Count column for the Requirement defect value means that a count is defined or required for the number of occurrences of that value. The value count is further defined by the existence of a check in either the Include or Exclude column of the same row. If the check is in either the Include column or the Exclude column, the value count will be the number of problems that have the attribute value within the intersection constraints defined by the Include checks for all the attributes. Since each attribute value must have a check in either the Include or Exclude column but not both, there is no ambiguity of the meaning of the Value Count if selected.

The Array Count column allows the definition or specification of multidimensional arrays of attribute value counts. A check mark in the Array Count column defines or specifies a count of the number of problems having that attribute value *and* any other attribute values with the Array Count checked as long as all the checked values are in the Include

column or in the Exclude column (one or the other). As with the Value Count column, the nature of the array counts is determined by the selection of either the Include or Exclude column for each value.

4.1. Example Problem Count Definition Checklist

To illustrate the use of the checklist as described above, we will invent a requirement for a problem count using the Problem Count Definition Checklist in Figure 4-1. Let us suppose we wish to count all the problems found over the course of the development cycle to date that are unique software product defects, i.e., not duplicates. We would like to know the number of such defects that are duplicate, how many of these problems are open and how many are closed; and finally, we wish to know the number of the unique problems that are open-evaluated and open-resolved by criticality level.

Our first task is to select all the values in the Include column that we wish to use as criteria for counting. Since we want to consider all problems, we will check both values, Open and Closed, for the **Problem Status** attribute. We want only software product defect problems, so we will check the first four values under **Problem Type** and exclude all others. Under **Uniqueness** we will include the Original value and exclude the Duplicate value. We want to include all those problems whose criticality has been established (evaluated) and exclude the rest, therefore we include the defined levels of criticality and exclude those problems which do not have a **Criticality** attribute. The **Urgency** values are selected similarly. Our interest is limited to problems found during development, so we include all **Finding Activities** except those in the Customer support category, which are excluded. Again we want only those problems that have been validated (evaluated) with a finding activity, so we exclude all those with no valid value. Lastly, we include all problems identified as being found either statically or dynamically and exclude those that have no value for the **Finding Mode** attribute.

Checking the Include and Exclude columns as we have done above defines a set of counting rules and will result in one number representing the combination of all the values selected above. We also need to specify additional counts. We will obtain a count of the Open and Closed problems by checking the Value Count column for each of these attribute values. We will obtain a count of each type of defect by placing a check in the Value Count column for each of the defects we included. We obtain the number of duplicate problems meeting all the other criteria by checking Value Count on the Duplicate value row. We obtain a two-dimensional array (two values by five values) by checking the Array Count columns for the Evaluated and Resolved values in **Problem Status** and the five **Criticality** levels.

A problem count report following the above specification might appear as shown below:

Problem Count Report for Product ID = [Example Ver 1 Rel 1]

Problem Count Definition Name [Problem Count A]

Date of report [/ /] Date of Measurement [/ /]

Total Number of Problems = 200

Open = 100

Closed = 200

Duplicate problems (with same criteria) = 300

Requirement defects = 35

Design defects = 50

Coding defects = 95

User document defects =20

Array Counts:

Number of open-evaluated and open-resolved problems by critical value =

Criticality	Evaluated	Resolved
1st level	5	5
2nd level	8	12
3rd level	15	15
4th level	8	12
5th level	12	8

There are several additional factors that need specification or definition such as date of measurement and time interval of measurement. We address these and other factors with the Problem Count Request Form discussed in Chapter 5.

Problem Count Definition Checklist-1

Software Product ID [Example V1 R1]

Definition Identifier: [Problem Count A] **Definition Date [01/02 /92]**

Attributes/Values	**Definition [ ]**		**Specification [X]**	
Problem Status	**Include**	**Exclude**	**Value Count**	**Array Count**
Open	✔		✔	
Recognized				
Evaluated				✔
Resolved				✔
Closed	✔		✔	
Problem Type	**Include**	**Exclude**	**Value Count**	**Array Count**
Software defect				
Requirements defect	✔		✔	
Design defect	✔		✔	
Code defect	✔		✔	
Operational document defect	✔		✔	
Test case defect		✔		
Other work product defect		✔		
Other problems				
Hardware problem		✔		
Operating system problem		✔		
User mistake		✔		
Operations mistake		✔		
New requirement/enhancement		✔		
Undetermined				
Not repeatable/Cause unknown		✔		
Value not identified		✔		
Uniqueness	**Include**	**Exclude**	**Value Count**	**Array Count**
Original	✔			
Duplicate		✔	✔	
Value not identifed		✔		
Criticality	**Include**	**Exclude**	**Value Count**	**Array Count**
1st level (most critical)	✔			✔
2nd level	✔			✔
3rd level	✔			✔
4th level	✔			✔
5th level	✔			✔
Value not identifie		✔		
Urgency	**Include**	**Exclude**	**Value Count**	**Array Count**
1st (most urgent)	✔			
2nd	✔			
3rd	✔			
4th	✔			
Value not identified		✔		

Figure 4-1 Example Problem Count Definition Checklist

Problem Count Definition Checklist–2				
Software Product ID [Example V1 R1]				
Definition Identifier: [Problem Count A]		**Definition Date [01/02/92]**		
Attributes/Values	**Definition [ ]**		**Specification [X]**	
Finding Activity	**Include**	**Exclude**	**Value Count**	**Array Count**
Synthesis of				
Design	✔			
Code	✔			
Test procedure	✔			
User publications	✔			
Inspections of				
Requirements	✔			
Preliminary design	✔			
Detailed design	✔			
Code	✔			
Operational documentation	✔			
Test procedures	✔			
Formal reviews of				
Plans	✔			
Requirements	✔			
Preliminary design	✔			
Critical design	✔			
Test readiness	✔			
Formal qualification	✔			
Testing				
Planning	✔			
Module (CSU)	✔			
Component (CSC)	✔			
Configuration item (CSCI)	✔			
Integrate and test	✔			
Independent verif. and valid.	✔			
System	✔			
Test and evaluate	✔			
Acceptance	✔			
Customer support				
Production/deployment		✔		
Installation		✔		
Operation		✔		
Undetermined				
Value not identified		✔		
Finding Mode	**Include**	**Exclude**	**Value Count**	**Array Count**
Static (non-operational)	✔			
Dynamic (operational)	✔			
Value not identified		✔		

Figure 4-1 Example Problem Count Definition Checklist

5. Using the Problem Count Request Form

The Problem Count Request Form complements the Problems Count Definition Form by providing a supporting form that allows the user to use literal attribute values to qualify or specify conditions of a definition or specification.

Attributes such as dates, **Originator**, **Environment**, and **Changes Made To** have values not amenable to checklist use unless there is a convenient method for specifying exactly what values should be used to qualify or set conditions on a count. For example, if you wished to count the number of problems opened by date for a three-month period, you should be able to specify the range of time that is of interest and how the count ought to be aggregated (daily, weekly, or monthly). In other cases, you may define a problem count and wish to have the data separated by environment or originator. The Problem Count Request Form (shown in Figure 5-2) provides the capability to specify these needs for attributes with literal values.

The header data includes the Product ID, the Problem Count Definition ID, and the Requester Name ID. The form header is used to relate the Problem Count Request Form to a specific Problem Count Definition Form by using the same Product ID and Problem Count Definition ID.

By using the Date of Request field, you may use the Problem Count Definition Form repetitively with a new Problem Count Request Form when dealing with a count to be made periodically or with variations in the way the count data is reported.

The remainder of the form is used to select attribute values that will sort or aggregate the count defined in the Problem Count Definition Checklist. The Time Interval for Count field is used to delineate the time period for which the count is intended to cover. This is tied to the **Problem Status** values selected on the Problem Count Definition checklist, e.g., all the problems opened over a specific three-month period. The form allows us to identify the time unit to be used to sort or separate the count using the Aggregate Time By fields.

We use the **Originator** and **Environment** attributes to sort or organize the count defined in the Problem Count Definition Checklist. When we select an attribute and value, we are specifying or describing a list of all the literal values for that selection with a corresponding problem or defect count defined by the Problem Count Definition Checklist. If we select both attributes, we must indicate the sort order or sequence in the Attribute Sort Order box. If we select more than one value for any single attribute, we must also provide a sort order or sequence in the Select Value box. If we wish to restrict the count to a limited number of originators or environments, then the Specific ID value is checked along with a list of appropriate IDs attached to the form.

We use the **Defects Found In** and **Changes Made To** attributes to limit or organize the count when the Problem Count Definition Checklist defines or specifies a defect count. The requester may chose among four types of software artifacts–requirements, design,

code, or user document, and the configuration level of the artifact–product, component, or module. Assuming the Problem Count Definition Checklist specified a count for design defects, selection of the design artifact and the component configuration level would result in a listing of all the design artifacts at the component level and the number of defects found in each artifact. The Specific ID value in each attribute may be used to identify configuration level artifacts by name for which a count is desired.

Note: It is possible that a defect may encompass or span more that one software artifact. This is particularly true if the software design is poorly partitioned. Defects can and do span more that one type of artifact as well. Consequently, it is difficult to assign a defect to one and only one artifact in many cases. We must realize that we are counting the number of artifacts affected by each defect, then indexing the list by the configuration level to obtain a defect count by artifact. This is not entirely accurate defect count; however, if the artifact's configuration level is large enough (e.g., a product), the defect count will be a good approximation of a number that is very difficult to obtain absolutely.

5.1. Example Problem Count Request Form

To illustrate the use of the Problem Count Request Form, we will build on the example set up in Section 4.1 and use the Problem Count Definition Checklist in Figure 4-1, which is identified in its header as the definition of Problem Count A.

We will use the Problem Count Request Form to specify that we want to sort the defect count by product component (as specified in the Problem Count Checklist).

The date requested is June 15, 1992, and the count is to include the time interval January 1, 1992, to June 30, 1992. We have no need to request a sort by time or by originator or environment. There are six product components, each with its own specification, design, code, and user document configuration baseline.

The Problem Count Request Form in Figure 5-2 reflects this specification; the resulting measurement would include the information shown in Section 4.1 plus a listing of all the component-level artifacts and the number of defects found in each. Figure 5-1 presents the data in a 4 x 6 dimensional matrix.

Component ID	Requirement Specification	Design Specification	Code	User Document
Comp A	5	3	8	4
Comp B	5	4	9	2
Comp C	4	5	10	2
Comp D	7	8	12	8
Comp E	4	10	26	3
Comp F	10	20	30	1

Figure 5-1 Example Result of Problem Count Request Form Specification

Problem Count Request Form

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

Date of Request: [6-15-92] **Requester's Name or ID: [I. M. Able]**

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

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

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

Report Count By:	Attribute Sort Order	Select Value, Sort Order	Special Instructions or Comments
Originator			
Site ID			
Customer ID			
User ID			
Contractor ID			
Specific ID(s) list			
Environment		Sort Order	
Hardware config ID			
Software config ID			
System config ID			
Test proc ID			
Specific ID(s) list			

Defects Found In:				
Select a configuration component level:	Type of Artifact			
	Requirement	Design	Code	User Document
Product (CSCI)				
Component (CSC)	✔	✔	✔	✔
Module (CSU)				
Specific (list)				

Changes Made To:				
Select a configuration component level	Type of Artifact			
	Requirement	Design	Code	User Document
Product (CSCI)				
Component (CSC)				
Module (CSU)				
Specific (list)				

Figure 5-2 Problem Count Request Form

Summary of Instructions for Using Problem Count Request Form

Product Name: Provide the name or ID of the product.

Problem Count Definition ID: Provide the name of the Problem Count Definition that is to be used as specification for the problem count.

Date of Request: Enter the date the request was submitted.

Date Count is to be Made: Provide the date that the count is to be taken.

Requester's Name: Provide the name of requester.

Time Interval for Count: Enter the *from* and *to* dates (m/d/y) to which the count is limited.

The following section may be used to organize (sort) or constrain the problem count as defined in the Problem Count Definition ID.

Date Opened, Date Closed, Date Evaluated, Date Resolved (same rules apply for all):

Selection of these attributes will constrain and aggregate the count according to the value selected. Selection of one of the time units will result in the counts being aggregated within each of the time units according to the Problem Count Definition Checklist.

Originator and Environment (same rules apply for both):

Selection of either of these attributes will constrain and/or sort the count according to the attribute value selected. Selection of an identified attribute value will result in sorting the count by the value selected for all values found in the problem data. If more than more value is selected, indicate the sort order (1, 2, 3, etc.). If the count is to be limited to a specific originator or environment ID, select the specific list value and provide a list of the IDs of interest. The count will be limited to the specific IDs listed.

If more than one of the above four attributes are selected, then each must be assigned an attribute sort order in the column provided.

Defects Found In and Changes Made To

A list of software artifacts with attributed defects, or with the number of changes due to defects, may be obtained by selecting one of the configuration levels in either or both of these attributes and identifying which artifacts should be listed.

Figure 5-3 Instructions for Problem Count Request Form

6. Using the Problem Status Definition Form

Since a number of the problem measurements are keyed on problem status (e.g., open or closed), it is necessary to define the criteria for any given problem status. The Problem Status Definition Form is used for defining problem status based on the existence of certain project-selected data entities that signify completion of one of the problem analysis or corrective action tasks. (See Section 3.2 for a full discussion of problem status).

An example of the Problem Status Definition Form is given in Figures 6-1 and 6-2. At the top of the checklist we have made provision to identify the completed checklist with the name of the product or project, the finding activity which is reporting the problems, and the date the checklist was defined. The activity block allows each problem finding activity to define problem status in accordance with the respective activity problem management process.

Following the header, there are four sections which we use to identify the status criteria. We use Section I to identify the attributes that must have valid values for a problem to be in the Open or in the Closed state. By checking the appropriate boxes, we establish the criteria required for Open and Closed status.

The remaining sections are relevant only if the Open status is decomposed into substates. We decompose the analysis and corrective action process into subset states to give a more detailed understanding of the problem solving process.

We use Section II to list the substates and identify each substate with the preprinted reference or line number at the left of the section. Each substate should be listed in time or process sequence order insofar as possible.

Section III asks us to identify the attributes that are required to have values as criteria for each substate. We indicate the attribute required by checking the appropriate square under the applicable substate number.

In certain situations, the attribute values will determine what attributes are necessary to move to the next state. For example, if a problem's **Uniqueness** value is Duplicate, the problem may need only to have valid values for the ID of **Original Problem** attribute. On the other hand, if the **Uniqueness** value is Original, there may well be a number of additional attributes requiring valid values before the problem can be closed. In this situation, we use Section IV to identify the substate number and the conditional attribute value (in the left-hand columns). The affected substates and the required attributes are listed by reference number in the right-hand columns.

6.1. Example Problem Status Definition Form

We will use Figures 6-1 and 6-2 to illustrate the use of the Problem Status Definition Form. We enter the header data as shown. We must pay particular attention to the process being used by the problem finding activity, since the process will greatly influence the selection of criteria for each status or substatus. In the example, we have selected the Customer Support activity.

We check three attributes as criteria for Open status and four attributes as criteria for Closed status, as shown. The criteria may be more, or less, stringent, depending on the finding activity and the problem management process. In the example, we have three substates that are identified in Section II (these are the same substates defined in Section 3.2).

We use the next page of the form (Figure 6-2) to identify the attributes required to have valid values for each substate. For Substate 1 (Recognized), the first three attributes and the fifth attribute must have valid values for a problem to be in the Recognized state. There are two attributes that have values that determine the required values for the next state. In our example, we note that both **Uniqueness** and **Problem Type** are required attributes for the Evaluated state. However, if the evaluation of **Uniqueness** is Duplicate, the required attributes for the Resolved state are quite different than if the value of **Uniqueness** were Original. These conditional attribute values are identified in Section IV.

We list Substate 2 as having conditional attribute values; and in the adjacent column, we list the attribute and the attribute value providing the condition (Uniqueness = Duplicate). The third column identifies the substate affected (Substate 3 or Resolved in this case) and the fourth column lists the required attributes as shown (attribute 18 or ID of Original Problem)

Note that attribute 11 (Problem Type) has been selected as an Evaluation criteria. This implies that the problem management process includes initial screening and evaluation of the reported problems. If, for example, the screening and initial evaluation were conducted by one organization and the resolution was executed by a second, the second organization might have checked off attribute 11 as a criterion for problems being Recognized. This case might apply in the situation where the two organizations are subcontractors to a prime contractor. It would be particularly important for the prime contractor, as well as the two subcontractors, to clearly understand this difference.

The role of the Problem Status Definition Form is to communicate, either in a descriptive or prescriptive sense, the meaning of the **Problem Status** attribute discussed in Chapter 3. The need for status information, i.e., the count of problems in each of the problem analysis and correction process states, has been briefly discussed above. The problem status definition also plays an important role in the definition of problem counts as we illustrate in Chapter 4, which discusses the use of the Problem Count Definition Checklist.

Problem Status Definition Rules

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

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

Section I

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

Section II

What Substates are used for Oper

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

Figure 6-1 Example Problem Status Definition Form-1

Problem Status Definition Form-2

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

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

Section III

What attributes are unconditionally required to have values as criteria for each substa

Attribute		Substate Numbe									
#	Name	1	2	3	4	5	6	7	8	9	10
1	Problem ID	✔	✔	✔							
2	Software Product Name or ID	✔	✔	✔							
3	Date/Time of Receipt	✔	✔	✔							
4	Date/Time of Problem Occurenc										
5	Originator ID	✔	✔	✔							
6	Environment ID		✔	✔							
7	Problem Description (text)		✔	✔							
8	Finding Activity		✔	✔							
9	Finding Mode		✔	✔							
10	Criticality		✔	✔							
11	Problem Type		✔	✔							
12	Uniqueness		✔	✔							
13	Urgency										
14	Date Evaluation Completed		✔	✔							
15	Evaluation Completed By										
16	Date Resolution Completed			✔							
17	Resolution Completed By										
18	ID of Original Problem										
19	Changes Made To										
20	Related Changes										
21	Defect Found In										
22	Defects Caused By										
23	Projected Availability			✔							
24	Released/Shipped										
25	Applied										
26	Approved By:										
27	Accepted By:										

Section IV

List the substates with conditional attribute values

Substate #	Conditional Attribute/Value
2	Uniqueness = Duplicat
2	Problem Type = Software defe
	and Uniqueness = Original

Substates affected

Substate #	Attribute Numbers
3	18
3	19,20,21,22

Figure 6-2 Problem Status Definition Form-2

7. Summary

The framework discussed in this report provides a structure that is the basis for deriving and describing measurable attributes for software problems and defects. A Problem Count Definition Checklist and supporting forms have been used to organize the attributes to allow a methodical and straightforward description and specification of software problem and defect measurements. The checklist and supporting forms may be used for defining and specifying a wide variety of software problem and defect counts, including those found by static or non-operational processes (e.g., design reviews or code inspections) and by dynamic or operational processes (e.g., testing or customer operation). Use of these checklists has the potential of reducing ambiguities and misunderstandings in these measures by giving organizations a basis for specifying and communicating clear definitions of problem and defect measurements. While it is not the intention of this report to comprehensively enlighten the reader about all the ways in which measurement of software problems and defects may be used, we hope to have encouraged interest in pursuing problem and defect measurement among the software project mangers, software engineers, planners, analysts, and researchers for which this report is intended.

Steps for Using Checklists to Define Problem and Defect Measurements

1. List the reasons why your organization wants measurements of problems and defects. Who will use your measurement results? How? What decisions will be based on these reports?
2. Use the Problem Status Checklist to describe the problem states used to measure progress of analysis and corrective action for each of the finding activities.
3. Organize the measurement needs according to product, purpose, software process, and time domains.
4. Use the Problem Count Definition Checklist to create descriptions or specifications of measurements that are made periodically or indicate degree of goal attainment at a point in time. Typically there will be a need for several Count Definition Checklists to accomplish this.
5. Use the Problem Count Request Form for each Problem Count Checklist to define the time the counts are to be made (or were made) and to identify any special organization or view of the measurements.
6. Then use checklists to specify to the database administrator(s) the measurement needs or to describe (verify) the results of measurements.

8. Recommendations

Since the framework, attributes, and checklists discussed in this report are focused on the objective of clear and precise communication of problem and defect measurements, it follows that our primary recommendation is to use attributes and the checklist and supporting forms to communicate—that is, describe or specify—the meaning of the measurements. Numerous opportunities arise for using the attributes and checklists to advantage, both in the establishment and use of these measurements. We outline several below.

8.1. Ongoing Projects

For those projects that are currently in the midst of development and are measuring problem and defects, we recommend using the Problem Count Definition Checklist and supporting forms to verify that the data that they are collecting and using in their measurements conforms to their requirements and needs. This may reveal two things about the measurements: (1) the measurements being taken do not "measure up", that is, the measurements are less than clear and precise in their meaning, and (2) the existing measurements fall short of what is needed to control the development or maintenance activity. If the measurements in use are verified by using the Problem Count Definition checklist, we urge you to use the checklist to describe your measurements to those who wish or need to understand.

The combination of a completed checklist and its supporting forms becomes a vehicle for communicating the meaning of measurement results to others, both within and outside the originating organization. They can be used for this purpose whether or not definitions have been agreed to in advance. They can also be used at the start of a project to negotiate and establish standards for collecting and reporting measures of software size. The benefits become even greater when the standards that are established are applied uniformly across multiple projects (and organizations).

8.2. New and Expanding Projects

For those projects that wish to establish or expand a measurement system, an initial task is to define problem and defect measurements required to determine and assess project progress, process stability, and attainment of product requirements or goals. We recommend using the Problem Count Definition Checklist and supporting forms as the primary mechanisms to specify the software problem and defect measurement part of the system. The checklists can be of help in addressing many of the issues to be resolved in developing or expanding a measurement system. Using the checklists for precise definitions of the measurements helps to crystallize several significant questions–

what data is required, when is it required, who collects and how is it collected, where and how is it kept, when it is reported, who has access, and how are the measurements to be used?

8.3. Serving the Needs of Many

Software problem and defect measurements have direct application to estimating, planning, and tracking the various software development processes. Users within organizations are likely to have different views and purposes for using and reporting this data. The Problem Count Definition Checklist may be used to negotiate and resolve issues that arise because of these differences, if only to serve as a vehicle to clearly express the various needs of each of the users.

8.4. Repository Starting Point

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

8.5. Conclusion

The power of clear definitions is not that they require action but that they set goals and facilitate communication and consistent interpretation. With this report, we seek only to bring clarity to definitions. Implementation and enforcement, on the other hand, are different issues. These are action-oriented endeavors, best left to agreements and practices to be worked out within individual organizations or between developers and their customers. We hope that the materials in this report give you the foundation, framework, and operational methods to make these endeavors possible.

References

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

[Boehm 73] Boehm, B. W., and others, "Characteristics of Software Quality," *TRW Software Series*, December 22, 1973.

[CMU/SEI] Paulk, Mark C.; Curtis, Bill; & Chrissis, Mary Beth; *Capability Maturity Model for Software* (CMU/SEI-91-TR-24, ADA 240603). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1991.

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

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

[Fenton 91] Fenton, Norman E. *Software Metrics: A Rigorous Approach*. New York, N.Y.: Van Nostrand Reinhold, 1991.

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

[Grady 92] Grady, Robert B. *Practical Software Metrics for Project Management and Process Improvement.* Englewood Cliffs, N.J.: Prentice-Hall, 1992.

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

[IEEE 90a] *IEEE Standard Glossary of Software Engineering Terminology* (IEEE Std 610.12-1990). New York, N.Y.: Institute of Electrical and Electronic Engineers, Inc., 1990.

[IEEE 90b] IEEE *Standard for a Software Quality Metrics Methodology* (IEEE Standard P-1061/D21). New York, N.Y.: Institute of Electrical and Electronic Engineers, Inc., 1990.

[IEEE 88a] *IEEE Standard Dictionary of Measures to Produce Reliable Software* (IEEE Std 982.1, 982.2-1988). New York, N.Y.: Institute of Electrical and Electronic Engineers, Inc., 1989.

[IEEE 88b] *IEEE Standard for Software Reviews and Audits* (IEEE Std 1028-1988). New York, N.Y.: Institute of Electrical and Electronic Engineers, Inc., 1988.

[IEEE 86] *IEEE Standard for Software Verification and Validation Plans* (IEEE Std 1012-1986). New York, N.Y.: Institute of Electrical and Electronic Engineers, Inc., 1986.

[Murine 83] Murine, G. E., "Improving Management Visibility Through the Use of Software Quality Metrics," *Proceedings from IEEE Computer Society's Seventh International Computer Software & Application Conference*. New York, N.Y.: Institute of Electrical and Electronic Engineers, Inc., 1983.

[Musa 87] Musa, John D.; Iannino, Anthony; & Okumuto, Kazihra. *Software Reliability Measurement, Prediction, Application.* New York, N.Y.: McGraw-Hill, 1987.

[Schneidewind 79] Schneidewind, N.F.; & Hoffmann, Heinz-Michael. "An Experiment in Software Error Data Collection and Analysis." *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 3, May 1979.

Appendix A: Glossary

A.1. Acronyms

CMU	Carnegie Mellon University
CSC	computer software component
CSCI	computer software configuration item
CSU	computer software unit
IEEE	The Institute of Electrical and Electronics Engineers, Inc.
KLOC	thousands of lines of code
KSLOC	thousands of source lines of code
LOC	lines of code
PDL	program design language
SEI	Software Engineering Institute
SLOC	source lines of code
S/W	software

A.2. Terms

Activity: Any step taken or function performed, both mental and physical, toward achieving some objective (CMU/SEI-91-TR-25).

Anomaly: Anything observed in the documentation or operation of software that deviates from expectations based on previously verified software products or reference documents. (IEEE 601.12-1990)

Artifact: An object made by human beings with a view to subsequent use.

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

Baseline: A specification or product that has been formally reviewed and agreed upon, which thereafter serves as the basis for future development, and which can be changed only through formal change control procedures (CMU/SEI-91-TR-25).

Causal analysis: The analysis of defects to determine their underlying root cause.

Commitment: A pact that is freely assumed, visible, and expected to be kept by all parties.

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

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

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

Customer specifications: Customer requirements for a software product that are explicitly listed in the requirements document.

Criticality: The degree of impact that a requirement, module, error, fault, or other item has on the development or operation of a system (IEEE 610.12-1990). See also **severity.**

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

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

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

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

Formal Review: A formal meeting at which a product is presented to the end-user, a customer, or other interested parties for comment and approval. It can also be a review of the management and technical activities and progress of the hardware/software development project (CMU/SEI-91-TR-25).

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

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

Metric: a quantified measure of the degree to which a system, component, or process possesses a given attribute. See **measurement** (IEEE 610.12-1990).

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

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

Peer review: A review of a software product, following defined procedures, by peers of the producers(s) of the product for the purpose of identifying defects and improvements. See also **software inspections** (CMU/SEI-91-TR-25).

Priority: The level of importance assigned to an item (IEEE 610.12-1990). See also **urgency.**

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

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

Project stage: A major activity within the software development and maintenance life cycle. Each project stage has a set of software work products to be produced. A stage is entered when the stage entry criteria are satisfied, and it is completed when the stage exit criteria are satisfied. The stages of the life cycle depend on the software process model used. Examples of stages are: system requirements analysis/design, software requirements analysis, preliminary design, detailed design, coding and CSU testing, CSC integration and testing, CSCI testing, and system integration and testing.

Product synthesis (software): The use of software tools to aid in the transformation of a program specification into a program that realizes that specification (IEEE 610.12-1990).

Repair: A set of changes made to a software product or software work product such that a fault or defect no longer occurs or exists with the application of an input set that previously resulted in a failure or defect.

Severity: The level of potential impact of a problem. This is also referred to as **criticality** (IEEE 610.12-1990).

Software inspection: A rigorous, formal**,** detailed technical peer review of the software design or implementation (code) (IEEE 1028-1988).

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

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

Software work product: Any artifact created as part of the software process, including computer programs, plans, procedures, and associated documentation and data, that may not be intended for delivery to a customer or end-user (CMU/SEI-91-TR-25).

Urgency: The degree of importance that the evaluation, resolution, and closure of a problem is given by the organization charged with executing a problem management process. The value is assigned by the supplier or developer, who should or must consider the severity of the problem as expressed by the problem originator. Urgency determines the order in which problems are evaluated, resolved, and closed. See also **priority.**

Appendix B: Using Measurement Results Illustrations and Examples

In this appendix, we illustrate a few of the ways in which counts of problems and defects are used to help plan, manage, and improve software projects and processes. Our purpose is not to be exhaustive, but rather to highlight some interesting uses that you may find worth trying in your own organization. We also want to encourage you to seek other new and productive ways to put quality measures to work.

Like most good ideas, the ones we show here have been borrowed from someone else. In fact, that was our principal criterion: the examples in this appendix—or ideas much like them—have all been used in practice. Several of the illustrations are adaptations of illustrations we found in Bob Grady's and Deborah Caswell's excellent book on experiences with software metrics at Hewlett-Packard [Grady 87]. Others are based on illustrations found in Watts Humphrey's book on software process management [Humphrey 89], in Bob Grady's latest book on practical software metrics [Grady 92], and in an SEI technical report [Baumert 92].

B.1. Project Tracking—System Test

The first use of counting problems that we illustrate (Figure B-1) will be familiar to almost every software development professional. It is a display of the cumulative, week-to-week status history of the problems encountered during integration and system test. This chart is useful because it identifies the number of problems encountered to date, the rate of occurrence, and the time taken to correct the problems for a pre-determined set of test cases or test procedures. It also provides a number of tips or indicators relative to the progress of the integration and test activity. This type of chart is typically updated at least once each week for use by the project manager. Monthly updates are sometimes used in formal reviews to indicate project status. Figure B-1 shows the total cumulative problems reported to date, and the number of open, and closed problems for each week of the test period. The dashed line shows the number of unevaluated open problems. Ideally, the number of closed problems should track the rate at which problems are received, with the number of open problems remaining flat or with a slight increase over the test period. When the number of new problems diminishes, the number of open problems will develop a negative slope and fall to zero at the end of test.

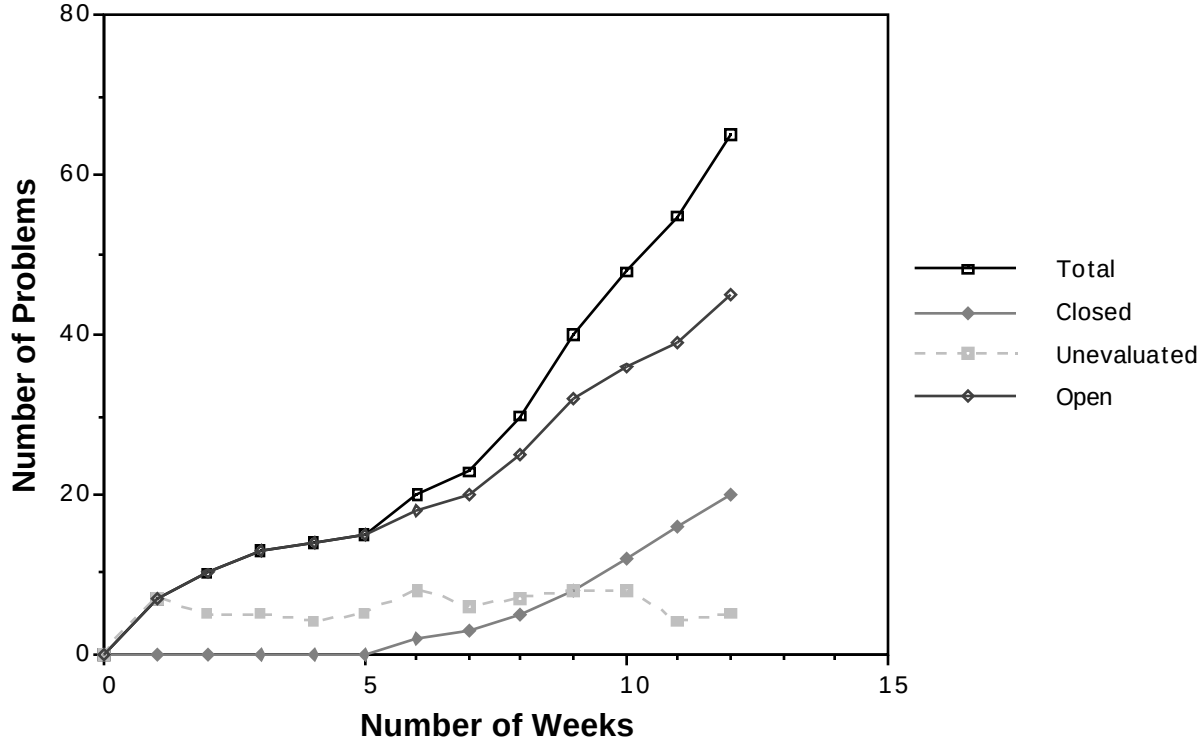

Figure B-1 Example of System Test Problem Status

The example data in Figure B-1 tells the project manager that the incoming problem rate is about eight problems per week, while the problem closing rate is about four problems per week, or about half what it should be to keep pace with the discovery rate. The time from discovery to closure is about seven weeks, which could have an impact on the

ability to execute test cases if the test case execution is dependent on the solution to the open problems.

Figure B-2 shows the number of unique open problems by age and criticality. The example data in this chart tells the project manager that there are 13 highly critical (Severity 1 and Severity 2) open problems, 9 of which are more than a week old. This should be an incentive for the project manager to look into the reasons for this and take action to correct the situation. The project manager should consider this information in light of other data (such as planned and actual tests run, the test plan and schedule, the number of people assigned to various tasks, and the problem analysis and correction process) to determine what action may be required to put the project back on track.

Other information the project manager might wish to review is the ratio of defects to problems, the ratio of problems to test procedures, the distribution of defects by component or module, the defect density (defects per KSLOC) as found by system test to date versus previous release system testing. Each of these will give the project manager a view of the current system test compared to previous versions or releases of the product, or the previous development process if no product data is available.

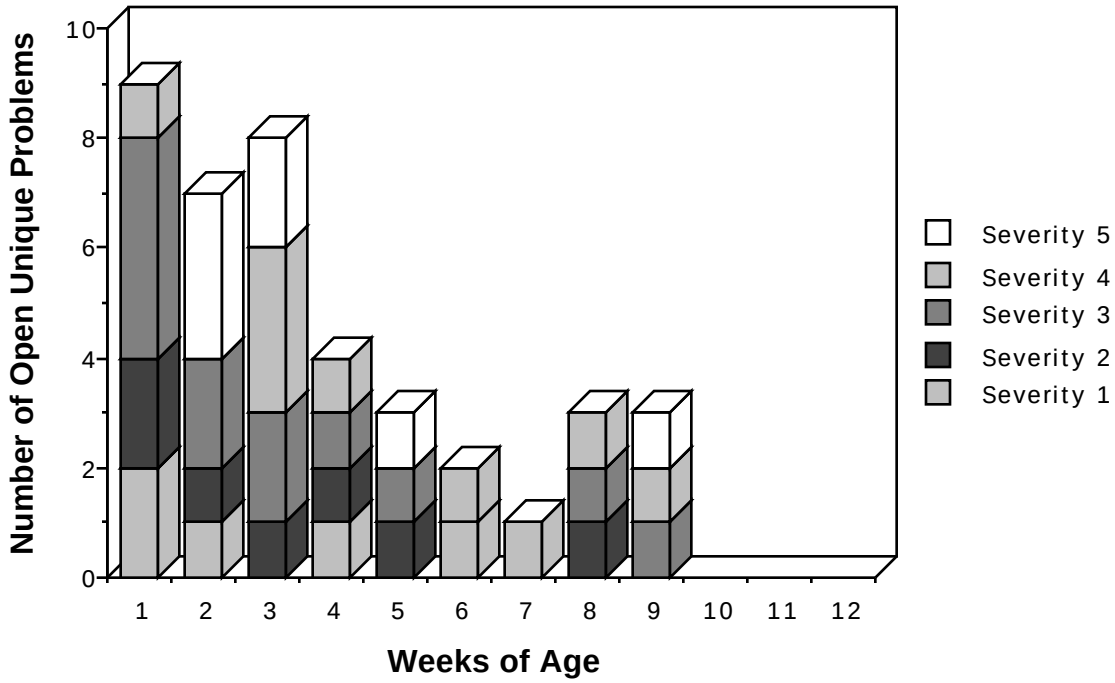

Figure B-2 Example of Open Problem Age by Criticality

B.2. Tracking Customer Experience

After the release of a product to the customer, we learn how well we met the target level of quality. The graphics in Figures 3-3 through 3-6 help to provide a quantifiable, objective track of the customer experience in using the product.

Figure B-3 shows the number of unique customer reported problems of all types over time. This chart indicates areas of customer support that may need to be addressed. For example, the consistently large number of "cause unknown" problems might indicate the need for better diagnostic tools or improvements in facilities for capturing data. The relatively large "enhancement request" type of problem report may be an indication of functional shortcoming in the product. The number of duplicate problems is not shown on this chart; however, it is useful to use the duplicate counts to determine the volatility of a problem or the effectiveness of the problem corrective action process. High ratios of duplicate problems not only require inordinate amounts of analysis effort but are indicative of shortcomings in the corrective action and distribution process.

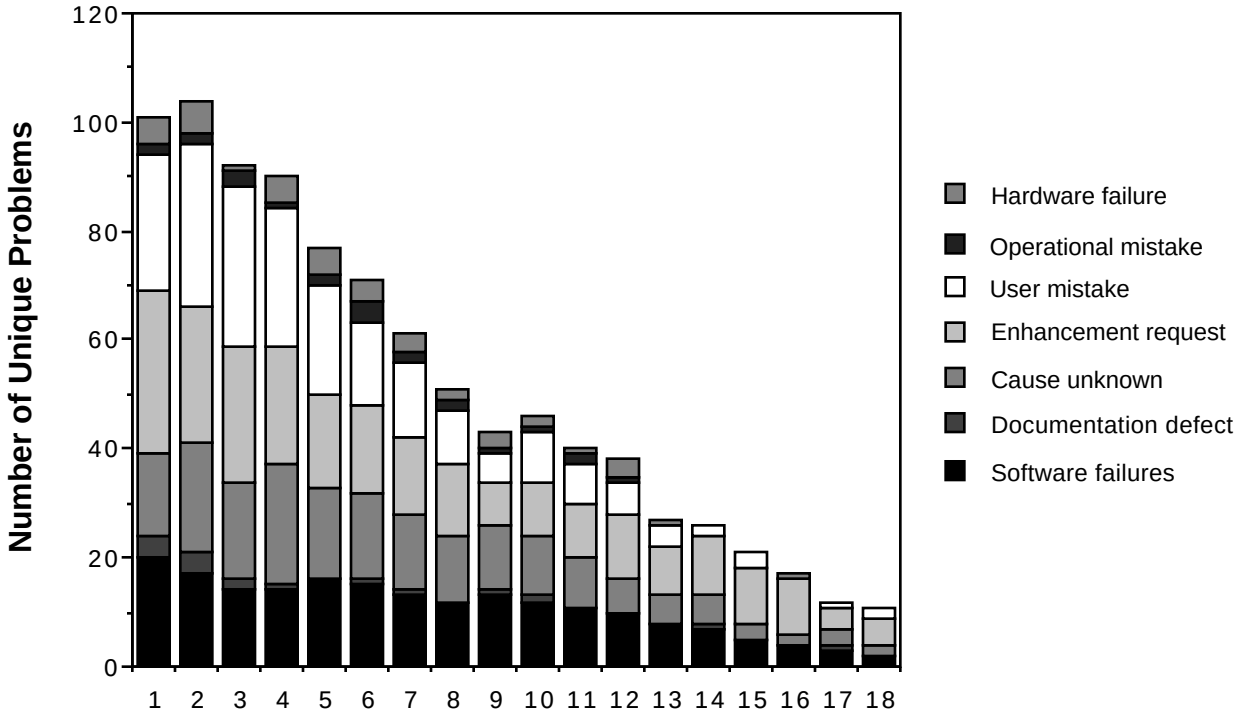

Figure B-3 Customer-Reported Problems

Figure B-4 plots the number of unique software failures each month over time and provides a software developer's view of the product reliability growth. The plot shows how the reliability of the product improves after the faults causing the failures have been corrected. This does not reflect the customers' operational experience since the data does not include duplicate or repeat failures caused by the same fault. However, each customer's experience will show the same trend assuming corrective changes are available and applied.

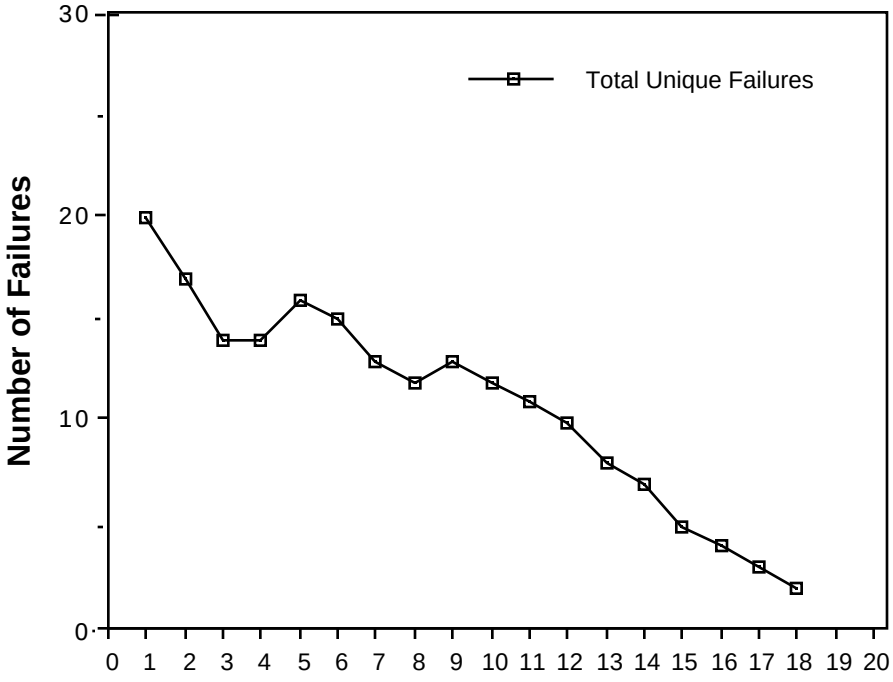

Figure B-4 Product Reliability Growth

It is sometimes important to understand how the failures are distributed by customer. Figure B-5 illustrates a failure distribution by customer for the first three months after release. It is apparent that two customers are experiencing significantly more unique failures than the others. It might be advisable to assign dedicated technical personnel to these customers so that they can quickly correct the faults causing the failures and prevent other sites from incurring the failures. Additionally, analysis of the customer's usage, system configuration, and so on, may reveal defects in the software development process that need correcting.

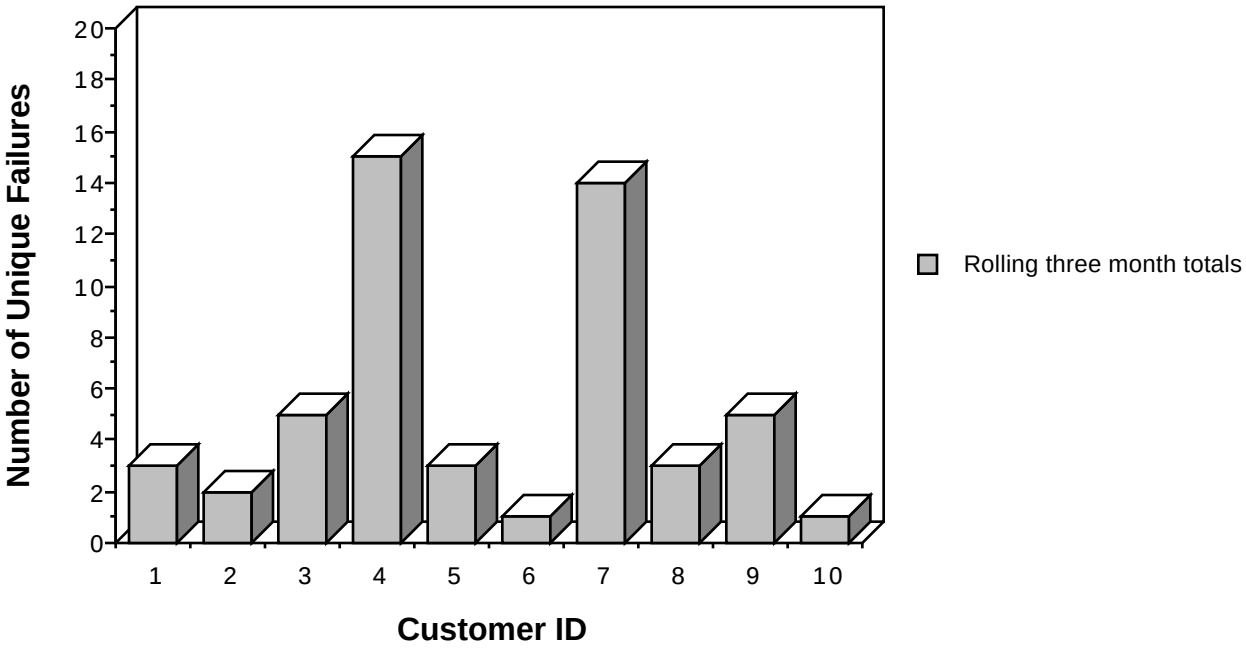

Figure B-5 Fault Distribution by Customer ID

Figure B-6 illustrates the reduction in product defect density over three releases of the product. This chart shows the results of improved defect prevention and detection activities during successive development cycles. This conclusion may not be valid if the number of customers has decreased or product usage has changed significantly over the prior release.

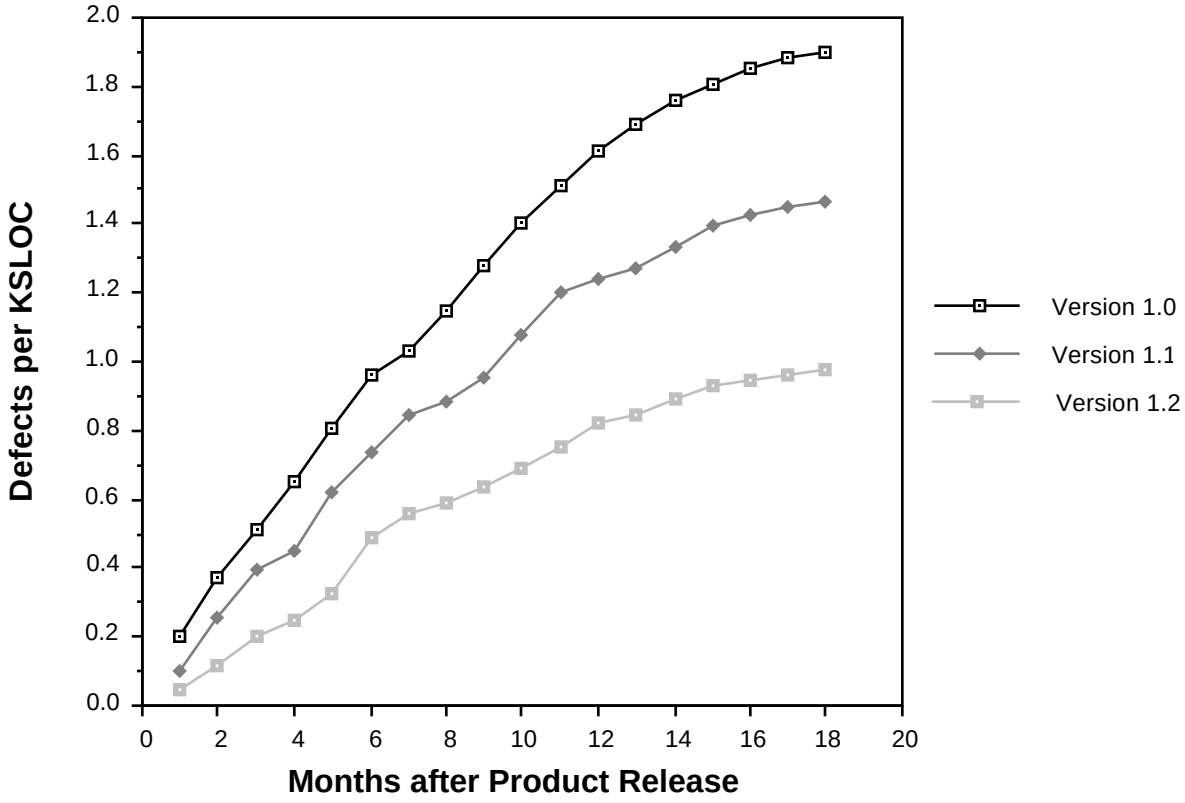

Figure B-6 Release-to-Release Improvement in Defect Density

B.3. Improving the Software Product and Process

When undertaking the development of a new product or the next release of an existing product, it is extremely important to analyze the previous experience of a project or project team to look for ways to improve the product and process. The graphs in this section illustrate two uses of defect data to identify product units and process steps with the most leverage for product and process improvement.

Figure B-7 accounts for all defects found during the software life cycle and identifies the activity in which they were injected. The difference between defects injected and those found within each activity reflect the defects that escaped the detection process (inspections, reviews, testing, etc.) and will affect the next activity. The purpose of the chart is to identify those activities which are the primary contributors of defects and those which have inadequate detection processes. The objective is to reduce the total number of defects injected and to improve the detection process so that the number of escaping defects is reduced. The example data in Figure B-7 tells us that the detection process in the early activities found 53% of the defects injected, leaving considerable room for improvement. It would appear that more disciplined and stringent inspections and reviews are needed.

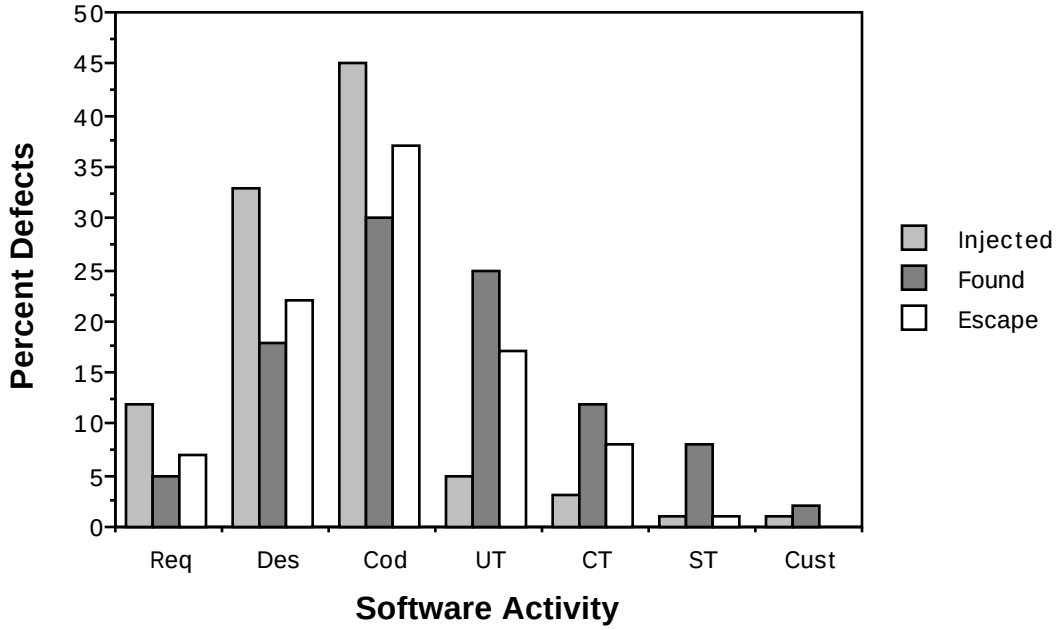

Figure B-7 Defect Analysis by Development Activity

Figure B-8 shows the same data as Figure B-7 except that it is expressed in terms of defect density. This view tells us that the software process being used to create the product needs to be scrutinized to reduce the total number of defects injected during development.

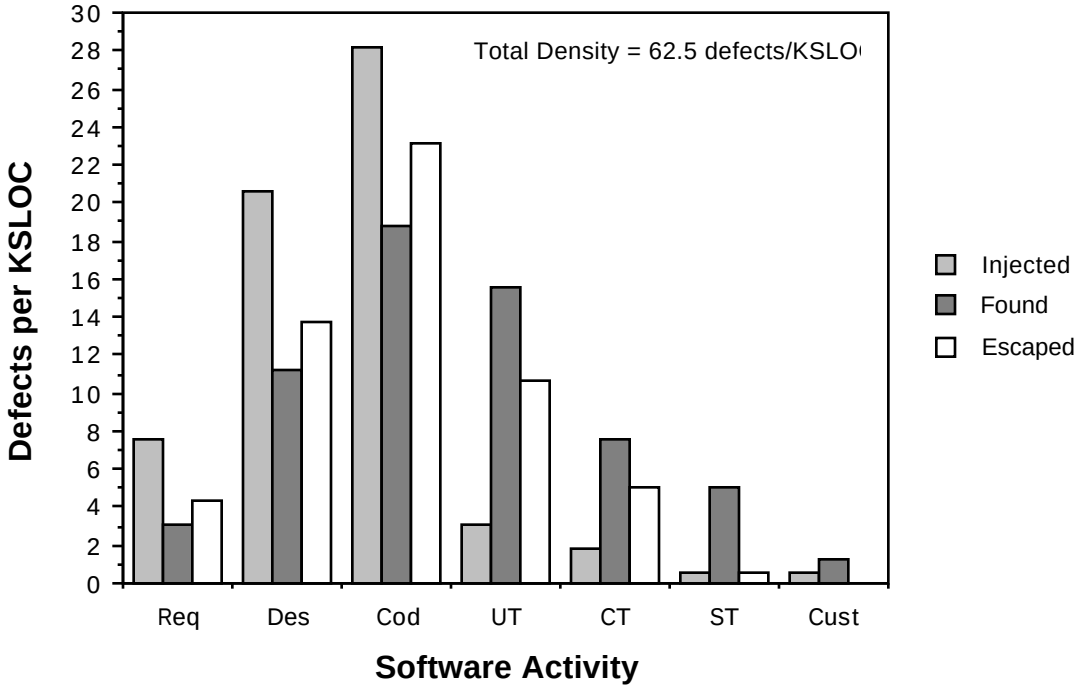

Figure B-8 Defect Density Analysis by Development Activity

Figures B-9 and B-10 identify the software modules which are “defect prone”; that is, they have been found to contain a high number of defects relative to the other product modules. The data shown in Figure B-9 identifies the modules with the highest percentage of defects, while the data in Figure in B-10 identifies the modules with the highest defect density. Both charts plot the modules in order of decreasing size (KSLOC). By comparing the module IDs on both charts, those modules with the largest defect contribution would be selected for examination, inspection, restructuring, or redesign as part of the development activity for the next release.

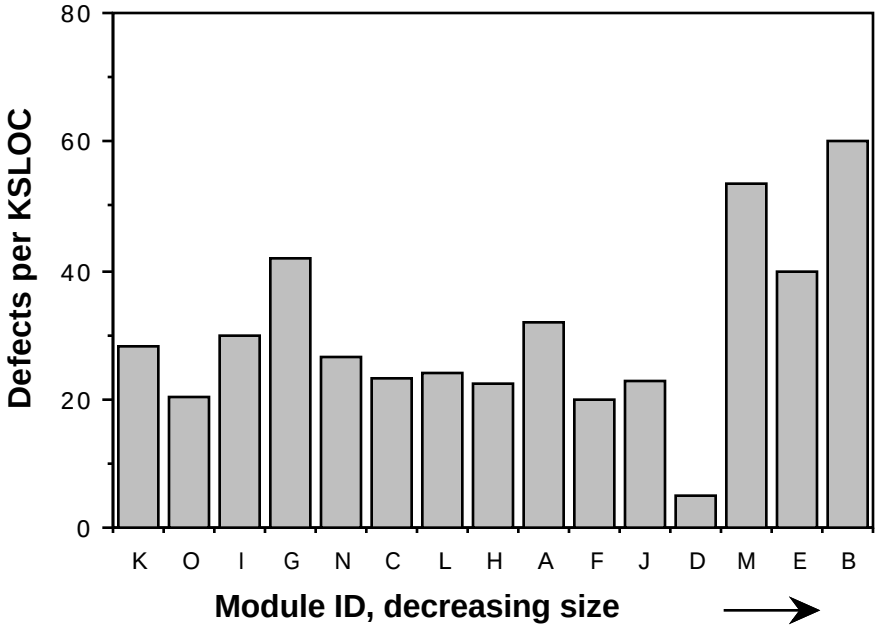

Figure B-9 Defect-Prone Modules by Defect Density

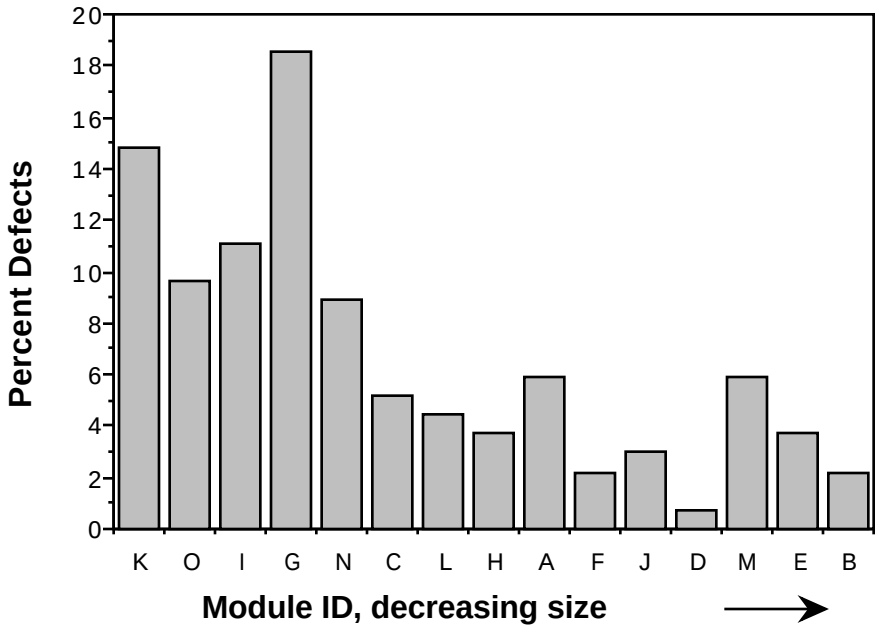

Figure B-10 Defect-Prone Modules by Percent Contribution

Appendix C: Checklists and Forms for Reproduction

Problem Count Definition Checklist–1

Software Product ID [ ]

Definition Identifier: [ ] **Definition Date [ ]**

Attributes/Values	**Definition [ ]**		**Specification [ ]**	
Problem Status	**Include**	**Exclude**	**Value Count**	**Array Count**
Open				
Recognized				
Evaluated				
Resolved				
Closed				
Problem Type	**Include**	**Exclude**	**Value Count**	**Array Count**
Software defect				
Requirements defect				
Design defect				
Code defect				
Operational document defect				
Test case defect				
Other work product defect				
Other problems				
Hardware problem				
Operating system problem				
User mistake				
Operations mistake				
New requirement/enhancement				
Undetermined				
Not repeatable/Cause unknown				
Value not identified				
Uniqueness	**Include**	**Exclude**	**Value Count**	**Array Count**
Original				
Duplicate				
Value not identifed				
Criticality	**Include**	**Exclude**	**Value Count**	**Array Count**
1st level (most critical)				
2nd level				
3rd level				
4th level				
5th level				
Value not identifie				
Urgency	**Include**	**Exclude**	**Value Count**	**Array Count**
1st (most urgent)				
2nd				
3rd				
4th				
Value not identified				

Problem Count Definition Checklist–2

Software Product ID [ ]

Definition Identifier: [ ] **Definition Date [ ]**

Attributes/Values	**Definition [ ]**		**Specification [ ]**	
Finding Activity	**Include**	**Exclude**	**Value Count**	**Array Count**
Synthesis of				
Design				
Code				
Test procedure				
User publications				
Inspections of				
Requirements				
Preliminary design				
Detailed design				
Code				
Operational documentation				
Test procedures				
Formal reviews of				
Plans				
Requirements				
Preliminary design				
Critical design				
Test readiness				
Formal qualification				
Testing				
Planning				
Module (CSU)				
Component (CSC)				
Configuration item (CSCI)				
Integrate and test				
Independent verif. and valid.				
System				
Test and evaluate				
Acceptance				
Customer support				
Production/deployment				
Installation				
Operation				
Undetermined				
Value not identified				
Finding Mode	**Include**	**Exclude**	**Value Count**	**Array Count**
Static (non-operational)				
Dynamic (operational)				
Value not identified				

Problem Count Request Form

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

Date of Request: [ ] **Requester's Name or ID: [ ]**

Date Count to be made: [ ]

Time Interval for Count: From [ ] To [ ]

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

Report Count By:	Attribute Sort Order	Select Value, Sort Order	Special Instructions or Comments
Originator			
Site ID			
Customer ID			
User ID			
Contractor ID			
Specific ID(s) list			
Environment		Sort Order	
Hardware config. ID			
Software config. ID			
System config. ID			
Test proc. ID			
Specific ID(s) list			

Defects Found In:				
Select a configuration	Type of Artifact			
component level:	Requirement	Design	Code	User Document
Product (CSCI)				
Component (CSC)				
Module (CSU)				
Specific (list)				

Changes Made To:				
Select a configuration	Type of Artifact			
component level:	Requirement	Design	Code	User Document
Product (CSCI)				
Component (CSC)				
Module (CSU)				
Specific (list)				

Problem Status Definition Rules

Product ID: **Status Definition ID:**

Finding Activity ID: **Definition Date:**

Section I

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

Section II

What Substates are used for Open?

#	Name	#	Name
1		6	
2		7	
3		8	
4		9	
5		10	

Problem Status Definition Form-2

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

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

Section III

What attributes are unconditionally required to have values as criteria for each substa

Attribute		Substate Number									
#	Name	1	2	3	4	5	6	7	8	9	10
1	Problem ID										
2	Software Product Name or ID										
3	Date/Time of Receipt										
4	Date/Time of Problem Occurence										
5	Originator ID										
6	Environment ID										
7	Problem Description (text)										
8	Finding Activity										
9	Finding Mode										
10	Criticality										
11	Problem Type										
12	Uniqueness										
13	Urgency										
14	Date Evaluation Completed										
15	Evaluation Completed By										
16	Date Resolution Completed										
17	Resolution Completed By										
18	ID of Original Problem										
19	Changes Made To										
20	Related Changes										
21	Defects Found In										
22	Defects Caused By										
23	Projected Availability										
24	Released/Shipped										
25	Applied										
26	Approved By:										
27	Accepted By:										

Section IV

List the substates with conditional attribute values		Substates affected	
Substate #	Conditional Attribute/Value	Substate #	Attribute Numbers