

IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software

IEEE Standards Board

Approved September 27, 1988

Sponsor

**Software Engineering Technical Subcommittee
of the
IEEE Computer Society**

© Copyright 1989 by

The Institute of Electrical and Electronics Engineers, Inc 345 East 47th Street, New York, NY 10017

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

PDF: ISBN 0-7381-0398-5, SS12559

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE which have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least once every five years for revision or reaffirmation. When a document is more than five years old, and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board 345 East 47th Street New York, NY 10017 USA

IEEE Standards documents are adopted by the Institute of Electrical and Electronics Engineers without regard to whether their adoption may involve patents on articles, materials, or processes. Such adoption does not assume any liability to any patent owner, nor does it assume any obligation whatever to parties adopting the standards documents.

Foreword

(This Foreword is not a part of IEEE Std 982.2-1988, IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software.)

This guide provides the underlying concepts and motivation for establishing a measurement process for reliable software, utilizing IEEE Std 982.1-1988, IEEE Standard Dictionary of Measures to Produce Reliable Software. This guide contains information necessary for application of measures to a project. It includes guidance for the following:

- 1) Applying product and process measures throughout the software life cycle, providing the means for continual self-assessment and reliability improvement;
- 2) Optimizing the development of reliable software, beginning at the early development stages, with respect to constraints such as cost and schedule;
- 3) Maximizing the reliability of software in its actual use environment during the operation and maintenance phases;
- 4) Developing the means to manage reliability in the same manner that cost and schedule are managed.

The guide is intended for design, development, evaluation (e.g., auditing or procuring agency) and maintenance personnel; software quality and software reliability personnel; and operations and acquisition support managers. It is organized to provide input to the planning process for reliability management.

This document contains six sections and one appendix.

Section 1, “Scope and References,” establishes the goals and boundaries of the guide.

Section 2, “Definitions,” serves as a central location for key terms used throughout the body of the document.

Section 3, “Measures to Produce Reliable Software,” contains guidance for measure selection and application.

Section 4, “Measure Organization and Classification,” provides both functional and software life cycle taxonomies of the measures.

Section 5, “Framework for Measures,” refines the measurement process to nine stages for effective implementation.

Section 6, “Errors, Faults, and Failures Analysis for Reliability Improvement,” provides further definition of the interrelationship of these concepts.

Appendix A, “Measures for Reliable Software,” contains the standard set of measures.

The goal of IEEE Std 982.2-1988, IEEE Guide for the Use of Standard Dictionary of Measures to Produce Reliable Software, is to generate a dialogue on the use and effectiveness of selected measures. This dialogue should result in the evolution of existing measures and the development of new measures that evaluate and promote software reliability. In this way, the body of knowledge of and experience with software reliability can be improved and expanded.

This is one of a series of integrated IEEE Software Engineering Standards. This guide is consistent with ANSI/IEEE Std 729-1983, Glossary of Software Engineering Terminology; ANSI/IEEE Std 730-1984, IEEE Standard for Software Quality Assurance Plans; ANSI/IEEE Std 828-1983, IEEE Standard for Software Configuration Management Plans; and ANSI/IEEE Std 1012-1986, IEEE Standard for Software Verification and Validation Plans. This guide may be applied either independently or in conjunction with those standards.

The Software Reliability Measurement Working Group Steering Committee had the following membership:

James Dobbins, *Chair*

Ray Leber, *Chair*

Ted Workman, *Chair*

Peter Bright
Antonio Cicu
Carlo Cucciati
Walter Ellis
William Farr
Stuart Glickman

Laura Good
Nancy Hall
Philip Marriott
Roger Martin
Walter Murch
Patricia Powell

Frank Salvia
Frank Siefert
Raghu Singh
Vijaya Srivastava
Henry Trocheset
William Wilson

The following persons were on the balloting committee that approved this document for submission to the IEEE Standards Board:

F. Ackerman
W. Boll
J. Bowen
P. Bright
F. Buckley
H. Carney
J. Chung
F. Coallier
P. Denny
J. Dobbins
D. Doty
R. Dwyer
W. Ellis
W. Farr
D. Favor
J. Fendrich
G. Fields
J. Forster
D. Gelperin
S. Glickman
S. Gloss-Sloer

L. Good
D. Gustavson
V. Haas
C. Hay
H. Hecht
L. Kaleda
T. Kurihara
G. Larsen
F. C. Lim
D. Lindberg
M. Lipow
D. Livson
P. Marriott
R. Martin
J. McCall
J. Mersky
W. Murch
J. Navlakha
D. Nickle
D. Percy
W. Perry
P. Petersen

D. Pfeiffer
R. Poston
P. Powell
N. Schneidewind
W. Schnoege
R. Schueppert
R. Shillato
D. Siefert
D. Simkins
R. Singh
V. Srivastava
A. Sukert
K. Tai
R. Thibodeau
P. Thompson
D. Town
H. Trocheset
R. van Tilburg
P. Wolfgang
T. Workman
N. Yopconka

The Software Reliability Measurement Working Group had the following membership:

B. Andrews
S. Beason
R. Berlack
M. Berry
N. Beser
J. Bieman
J. Blackman
J. Bowen
M. Bruyere
F. Buckley
N. Chu
G. Chisholm
G. Clow
J. Corrinne
W. Covington
W. Daney
B. Daniels
F. DeKalb
I. Doshay
E. Dunaye
F. Fisk
A. Florence
D. Frevert
D. Gacke

W. Goss
S. Gray
F. Gray
V. Haas
C. Harrison
S. Hartman
L. Hearn
H. Hecht
R. Hood
J. Horch
D. Hurst
N. Johnson, Jr
R. Kenett
R. Kessler
R. Kumar
M. Landes
R. Lane
J. Latimer
L. Lebowitz
M. Lipow
C. McPherson
A. Moseley
S. Nemecek
K. Nidiffer

K. Oar
R. Panchal
R. Pikul
E. Presson
R. Prudhomme
H. Richter
M. Richter
L. Sanders
R. Sandborgh
H. Schock
D. Simkins
O. Smith
R. Spear
E. Soistman
L. Sprague
A. Stone
D. Sudbeck
R. Sulgrove
D. Town
R. Wamser
J. Wang
D. Winterkorn
J. Wujek
B. Zamastil

When the IEEE Standards Board approved this standard on September 27, 1988, it had the following membership:

Donald C. Fleckenstein, *Chair*

Marco Migliaro, *Vice Chair*

Andrew G. Salem, *Secretary*

Arthur A. Blaisdell
Fletcher J. Buckley
James M. Daly
Stephen R. Dillon
Eugene P. Fogarty
Jay Forster*
Thomas L. Hannan
Kenneth D. Hendrix
Theodore W. Hissey, Jr.

John W. Horch
Jack M. Kinn
Frank D. Kirschner
Frank C. Kitzantides
Joseph L. Koepfinger*
Irving Kolodny
Edward Lohse
John E. May, Jr.
Lawrence V. McCall

L. Bruce McClung
Donald T. Michael*
Richard E. Mosher
L. John Rankine
Gary S. Robinson
Frank L. Rose
Helen M. Wood
Karl H. Zaininger
Donald W. Zipse

*Member emeritus

Contents

CLAUSE	PAGE
1. Scope and References	1
1.1 Scope	1
1.2 References	1
2. Definitions.....	1
3. Measures to Produce Reliable Software	2
3.1 Constructive Approach to Reliable Software.....	3
3.2 Measurement Environment	12
3.3 Measurement Selection Criteria.....	13
4. Measure Organization and Classification	14
4.1 Functional Classification.....	15
5. Framework for Measures	29
5.1 Measurement Process.....	29
5.2 Stages of Measurement Process	29
6. Errors, Faults, and Failure Analysis for Reliability Improvement	34
6.1 Dynamics of Errors, Faults, and Failures.....	34
6.2 Analysis of Error, Fault, Failure Events	36
6.3 Minimizing Failure Events.....	37
6.4 Summary	37

APPENDIX

Appendix A (informative) Measures for Reliable Software.....	38
A1 . Fault Density	38
A2 . Defect Density.....	40
A3 Cumulative Failure Profile.....	42
A4 Fault-Days Number.....	44
A5. Functional or Modular Test Coverage	46
A6 Cause and Effect Graphing	49
A7. Requirements Traceability	53
A8. Defect Indices	55
A9. Error Distribution(s).....	55
A10. Software Maturity Index	59
A11. Manhours per Major Defect Detected.....	62
A12. Number of Conflicting Requirements.....	64
A13. Number of Entries and Exits per Module	67
A14 Software Science Measures	67
A15. Graph-Theoretic Complexity for Architecture	70
A16. Cyclomatic Complexity	74
A17. Minimal Unit Test Case Determination	76
A18. Run Reliability	78
A19 . Design Structure.....	82
A20. Mean Time to Discover the Next K Faults	84
A21 . Software Purity Level	87
A22. Estimated Number of Faults Remaining (by Seeding)	89
A23. Requirements Compliance	92
A24. Test Coverage	96
A25. Data or Information Flow Complexity.....	98
A26. Reliability Growth Function	101
A27. Residual Fault Count.....	103
A28. Failure Analysis Using Elapsed Time.....	105

A29. Testing Sufficiency	107
A30. Mean Time to Failure.....	109
A31. Failure Rate	111
A32. Software Documentation and Source Listings.....	115
A33. RELY-Required Software Reliability.....	118
A34. Software Release Readiness.....	121
A35. Completeness	125
A36. Test Accuracy	128
A37. System Performance Reliability.....	130
A38. Independent Process Reliability.....	134
A39. Combined Hardware and Software (System) Operational Availability.....	136

List of Symbols			
Symbol	Meaning	Measures Used	Primitive/Metric
A	(1) Number of arrivals during time period T (2) Reliability growth factor	37 26	P M
A_{existing}	Number of ambiguities in a program remaining to be eliminated	6	P
$A(t)$	System availability	39	M
A_{tot}	Total number of ambiguities identified	6	P
α	Test accuracy	36	M
AQ	Average queue length	37	M
AS	Average service time	37	P
B	Software science measure (number of errors)	14	M
B_1	Number of functions not satisfactorily defined	35	P
B_2	Number of functions	35	P
B_3	Number of data references not having an origin	35	P
B_4	Number of data references	35	P
B_5	Number of defined functions not used	35	P
B_6	Number of defined functions	35	P
B_7	Number of referenced functions not defined	35	P
B_8	Number of referenced functions	35	P
B_9	Number of decision points not using all conditions, options	35	P
B_{10}	Number of decision points	35	P
B_{11}	Number of condition options without processing	35	P
B_{12}	Number of condition options	35	P
B_{13}	Number of calling routines with parameters not agreeing with defined parameters	35	P
B_{14}	Number of calling routines	35	P
B_{15}	Number of condition options not set	35	P
B_{16}	Number of set conditions options having no processing	35	P
B_{17}	Number of set condition options	35	P
B_{18}	Number of data references having no destination	35	P

List of Symbols			
β	Observed hardware failure rate	39	P
C	Complexity metric (static, dynamic, cyclomatic)	15, 16	M
C_i	Complexity for program invocation and return along each edge e_i as determined by user	15	P
CE	Cause and effect measure	6	M
CM	Completeness measure	35	M
D	Software science measure (program difficulty)	14	M
d_k	Complexity for allocation of resource k as determined by user	15	P
D_i	Total number of unique defects detected during the i th design or code inspection process or the i th life cycle phase	2, 8	P
DD	Defect density	2	M
DEs	Decomposition elements	23	P
DI	Defect index	8	M
DSM	Design structure metric	19	M
datain	Number of data structures from which the procedure retrieves data	25	P
dataout	Number of data structures that the procedure updates	25	P
E	(1) Number of edges (2) Software science measure (effort)	15, 16, 17 14	P M
e_i	Number of entry points for the i th module	13	P
F or F_i	Total number of unique faults found in a given time interval resulting in failures of a specified severity level	1	P
f or f_i	Total number of failures of a given severity level in a given time interval	3, 20, 21, 27, 31	P
F_d	Fault density	1	M
F_{ij}	Frequency execution of intermodule transfer from module i to j	38	P
F_a	Number of software functions (modules) in the current delivery that are additions in the current delivery	10	P
F_c	Number of software functions (modules) in the current delivery that include internal changes from a previous delivery	10	P
FD	Fault days metric	4	M
FD_i	Fault days for the i th fault	4	P
F_{del}	Number of software functions (modules) in the previous delivery that are deleted in the current delivery	10	P

List of Symbols			
FE	Number of the software functional (modular) requirements for which all test cases have been satisfactorily completed	5	P
F_{it}	Total number of faults detected to date in software integration testing	29	P
F_{pit}	Number of faults detected prior to software integration testing	29	P
FT	Total number of software functional test requirements	5	P
Γ	Average rate of jobs	37	M
Υ	Observed hardware repair rate	39	P
I	Total number of inspections to date	2, 11	P
IFC	Information flow complexity	25	M
J	Total number of jobs completed	37	P
K	Number of resources	15	P
k	Number of runs for the specified period of time	18	P
$KSLOC$	Number of source lines of executable code and nonexecutable data declarations in thousands	1, 2, 8	P
$KSLOD$	Number of source lines of design statements in thousands that have been inspected to date	2, 8	P
L	Software science measure (observed program length)	14	M
l	Software science measure (program level)	14	M
LI	Software science measure (program level)	14	M
$\lambda(t)$	Failure rate function	31	M
λ	Observed software failure rate	39	P
length	Number of source statements in a procedure	25	P
lfi	Local flows into a procedure	25	P
lfo	Local flows from a procedure	25	P
M	Manhours per major defect detected	11	M
M_i	Number of medium defects found	8	P
m_i	Number of entry and exit points for module i	13	M
MD	Number of modules	38	P
$ME(i)$	Measure effectiveness for the i th issue	34	P
M_{it}	Number of modules integrated	29	M
M_T	Number of software functions (modules) in the current delivery	10	M

List of Symbols			
M_{tot}	Total number of modules in final configuration	29	M
MTTF	Mean-time-to-failure	30	M
μ_i	Observed software fault correction rate with i faults in the software	39	P
N	Number of nodes	15, 16, 17	P
nc	Number of correct runs in a given test sample	18, 26	P
NF	Total number of faults within a program	29, 39	M
NF_{rem}	Total number of faults remaining within a program	22, 29	M
$NF_{\text{rem}}(\%)$	Percentage of faults remaining within a program	36	M
n_F	Number of faults found that were not intentionally seeded	22	P
NR	Number of runs (or test cases) in a given test sample	18, 26	P
NS	Total number of stages	26	P
N_s	Number of seeded faults	22, 36	P
n_s	Number of seeded faults found	22	P
$N1$	Total number of occurrences of the operators in a program	14	P
$n1$	Number of distinct operators in a program	14	P
$N2$	Total meaning of occurrences of the operands in a program	14	P
$n2$	Number of distinct operands in a program	14	P
N_1	Number of errors detected using SVDs due to inconsistencies	23	P
N_2	Number of errors detected using SVDs due to incompleteness	23	P
N_3	Number of errors detected using SVDs due to misinterpretation	23	P
P	Probability measure over the sample space	18	P
p_n	Probability of repairing a hardware failure correctly	39	P
P_i	(1) Probability that the i th run is selected from the sample space	18, 38	P
P_s	Probability of correcting a software fault when detected	39	P
$P_{N, n}(t)$	System upstate probabilities	39	P
PI_i	Phase index metric for the i th life cycle phase	8	M
PL	Purity level	21	M
PS	Product size	8	P

List of Symbols			
$P1$	Total number of modules in the program	19	P
$P2$	Number of modules dependent on the input or output	19	P
$P3$	Number of modules dependent on prior processing	19	P
$P4$	Number of database elements	19	P
$P5$	Number of nonunique database elements	19	P
$P6$	Number of database segments (partition of the state)	19	P
$P7$	Number of modules not single entrance/single exit	19	P
Q	Queue length distribution	37	P
q_i	Probability that P_i will be generated in a user environment	38	P
R	(1) Total response time for each functional job (2) Reliability of the system	37 38	M M
R_i	Reliability of the i th module	38	P
R_k	Run reliability at a specified stage	18, 26	M
$R(t)$	Reliability function	31	M
r_i	Flag to denote if the i th process generates the correct software system output	38	P
r_{ki}	Resource status array indicator flags	15	P
RG	Number of regions	16, 17	P
$RK(i)$	Risk factor for the i th measure	34	M
RT	Response time distribution	37	P
$R1$	Number of requirements met by the architecture	7	P
$R2$	Number of original requirements	7	P
S	Sample space of possible input patterns and states	18	P
S_i	Point from the sample space of possible inputs	18	P
s_i	Number of serious defects found	8, 11	P
SB	Total amount of time server is busy	37	P
SE	Server's efficiency	37	M
SN	Number of splitting nodes	16, 17	P
SMI	Software maturity index	10	M
ST	Service time distribution	37	P
T	(1) Time period over which measurements are made (2) Time	37 14	P M
T_i	Number of trivial defects found	8	P

List of Symbols			
t_i	Observed times between failures of a given severity level	20, 21, 27, 28, 30, 31	P
TC	Test coverage	24	M
TM	Traceability measure	7	M
T_1	Time expended by the inspection team in preparation for design or code inspection meeting	11	P
T_2	Time expended by the inspection team in conduct of a design or code inspection meeting	11	P
U	Utilization	37	P
V	Software science measure (program volume)	14	M
VR	Number of requests per functional job for each server during time period T	37	P
W_i	Weighting distribution	8, 19, 34	P
WT	Waiting time distribution	37	P
X	Throughput	37	P
x_i	Number of exit points for the i th module	13	P
NOTE—P stands for primitive; M stands for derived metric			

IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software

1. Scope and References

1.1 Scope

This guide provides the conceptual insights, implementation considerations, and assessment suggestions for the application of IEEE Std 982.1-1988, IEEE Standard Dictionary of Measures to Produce Reliable Software [2].¹ The Dictionary, by its very nature, is limited in what it may contain. While this is necessary for standardization, abbreviation of detail might be detrimental to the most effective use of the Dictionary. This guide is written to provide a bridge between IEEE Std 982.1-1988 and the user, so that full benefit may be derived from the application of the standard to a software project.

1.2 References

This guide shall be used in conjunction with the following publications:

[1] ANSI/IEEE 729-1983, IEEE Standard Glossary of Software Engineering Terminology.²

[2] IEEE Std 982.1-1988, IEEE Standard Dictionary of Measures to Produce Reliable Software.³

2. Definitions

Most of the definitions listed below can be found in ANSI/IEEE Std 729-1983, IEEE Standard Glossary of Software Engineering Terminology [1]. The definitions of the terms “concept phase,” “defect,” “measure,” “primitive,” “software reliability management” are not present in ANSI/IEEE Std 729-1983, and establish meaning in the context of this guide. The acronym ISO (International Organization for Standardization) indicates that the definition is also accepted by ISO.

concept phase: The period of time in the software life cycle during which system concepts and objectives needed by the user are identified and documented. Precedes the requirements phase.

defect: A product anomaly. Examples include such things as

- 1) omissions and imperfections found during early life cycle phases and
- 2) faults contained in software sufficiently mature for test or operation. See also “fault.”

¹ The numbers in brackets correspond to those of the references in 1.2 ; numbers in brackets preceded by an “A” correspond to references in the Appendix .

² ANSI/IEEE publications are available from the Sales Department, American National Standards Institute, 1430 Broadway, New York, NY 10018, or from the IEEE Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08854-1331.

³ IEEE publications are available from the IEEE Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08854-1331.

error: Human action that results in software containing a fault. Examples include omission or misinterpretation of user requirements in a software specification, and incorrect translation or omission of a requirement in the design specification (see ANSI/IEEE Std 729-1983 [1]).

failure:

- 1) The termination of the ability of a functional unit to perform its required function. (ISO; ANSI/IEEE Std 729-1983 [1]).
- 2) An event in which a system or system component does not perform a required function within specified limits. A failure may be produced when a fault is encountered [1].

fault:

- 1) An accidental condition that causes a functional unit to fail to perform its required function. (ISO; ANSI/IEEE Std 729-1983 [1]).
- 2) A manifestation of an error in software. A fault, if encountered, may cause a failure. Synonymous with bug [1].

measure: A quantitative assessment of the degree to which a software product or process possesses a given attribute.

primitive: Data relating to the development or use of software that is used in developing measures or quantitative descriptions of software. Primitives are directly measurable or countable, or may be given a constant value or condition for a specific measure. Examples include error, failure, fault, time, time interval, date, number of noncommentary source code statements, edges, nodes.

software reliability: The probability that software will not cause the failure of a system for a specified time under specified conditions. The probability is a function of the inputs to, and use of, the system as well as a function of the existence of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered [1].

software reliability management: The process of optimizing the reliability of software through a program that emphasizes software error prevention, fault detection and removal, and the use of measurements to maximize reliability in light of project constraints such as resources (cost), schedule, and performance.

3. Measures to Produce Reliable Software

This guide addresses two major topics. The first is the rationale for the document IEEE Std 982.1-1988; IEEE Standard Dictionary of Measures to Produce Reliable Software. The guide presents the measure selection criteria. It address these questions: “To whom is the standard directed?” “When should the standard be applied?” and “Why was the standard created?”

The second topic is the application of these measures to product, project, and industry. For each measure the guide provides further definition, examples, interpretation, and references. The questions addressed include: “How can the standard be used effectively?” “How should results be interpreted?” and “How should objectives be set?”

The goal of the Dictionary is to support software developers, project managers, and system users in achieving optimum reliability levels in software products. It was designed to address the needs of software developers and customers who are confronted with a plethora of models, techniques, and measures in the literature, but who lack sufficient guidance to utilize them effectively. The standard addresses the need for a uniform interpretation of these and other indicators of reliability.

The Dictionary assumes an intimate relationship between the reliability of a product and the process used to develop that product. The reliable product provides confirmation of a successful process; the unreliable prod-

uct provides a lesson for process change. It is in this general spirit that the measures selected give visibility to both process and product so that the essential facts necessary for process evaluation and change are available.

The measures were selected to provide information throughout the life cycle of a product. They can be used to make optimum operational reliability levels and to provide a *post factum* measurement of reliability levels achieved. The basic goal is to provide the elements of a measurement program that support a constructive approach for achieving optimum reliability of the end product.

The selected measures are related to various product and process factors that may have an impact on software reliability and that are observable throughout the entire life cycle. The Dictionary provides, even in the early life cycle phases, a means for continual assessment of the evolving product and the process. Through early reliability assessment, the Dictionary supports constructive optimization of reliability within given performance, resource, and schedule constraints.

The Dictionary focuses on measures of the potential causes of failure throughout the software life cycle, rather than just on measures of the effect of poor reliability of nearly completed products. The intent is to produce software that is reliable, rather than just an estimate of the failure-freeness of a nearly completed and possibly unreliable product. Both the traditional approach of measuring reliability and the constructive approach of building in reliability are placed in context in this document.

The primitives to be used and the method of computation are provided for each measure. The standard calibrates the rulers, the measurement tools, through the use of common units and a common language. It promotes the use of a common database in the industry. Through commonality the standard provides the justification for building effective tools for the measurement process itself.

The standard and this guide are intended to serve as the foundation on which researchers and practitioners can build consistent methods. These documents are designed to assist management in directing product development and support toward specific reliability goals. The purpose is to provide a common set of definitions through which a meaningful exchange of data and evaluations can occur. Successful application of the measures is dependent upon their use as intended in the specified environments.

The future study of extensive sets of basic data collected through the application of this standard will make it feasible to understand causal relationships, if any, between reliability indicators obtained during the process and operational reliability of a completed product. It will also promote the refinement of existing models and the development of new models based on this consistent data.

3.1 Constructive Approach to Reliable Software

An analysis of the relationship between the software development and maintenance processes and software reliability is provided. The purpose is to clarify the criteria used in the selection of measures.

Software reliability measurement is the determination of the probability that software will not cause the failure of a system for a specified time under specified conditions. The probability is a function of the inputs to, and use of, the system as well as a function of the existence of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered [1].

3.1.1 Constructive Approach

Reliability is an estimation of system failure-freeness. System failures that are caused by software are a valid concern of the software community. A constructive approach to reliable software seeks to remove the root causes of this class of system failures (Fig 3.1.1-1) through software development and support processes that promote fault avoidance, early fault detection, appropriately prompt removal, and system-designed fault tolerance (Fig 3.1.1-2).

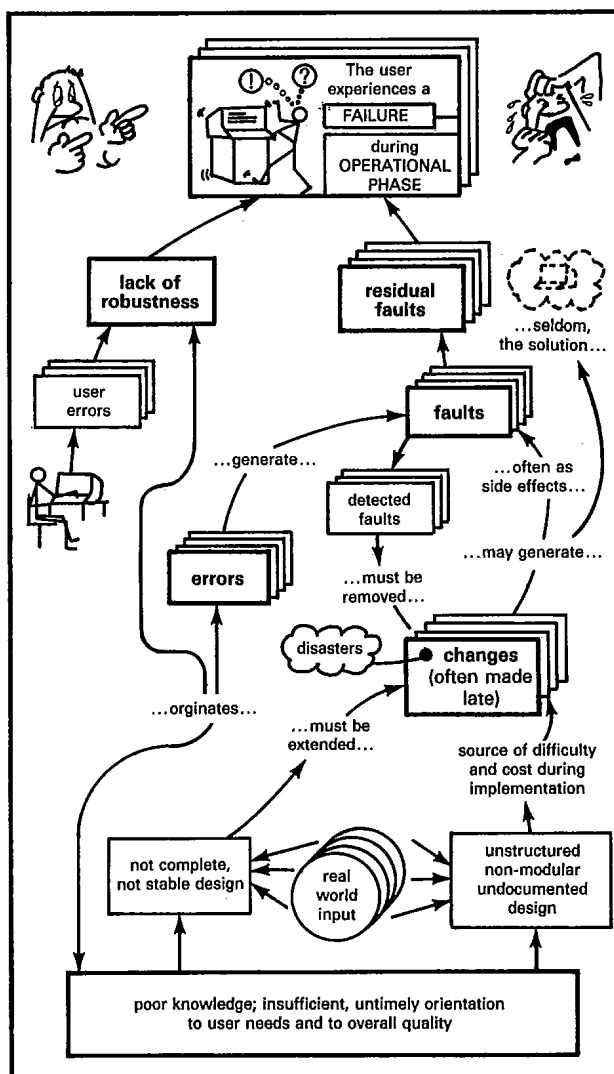


Figure 3.1.1-1 — Causes of Failures

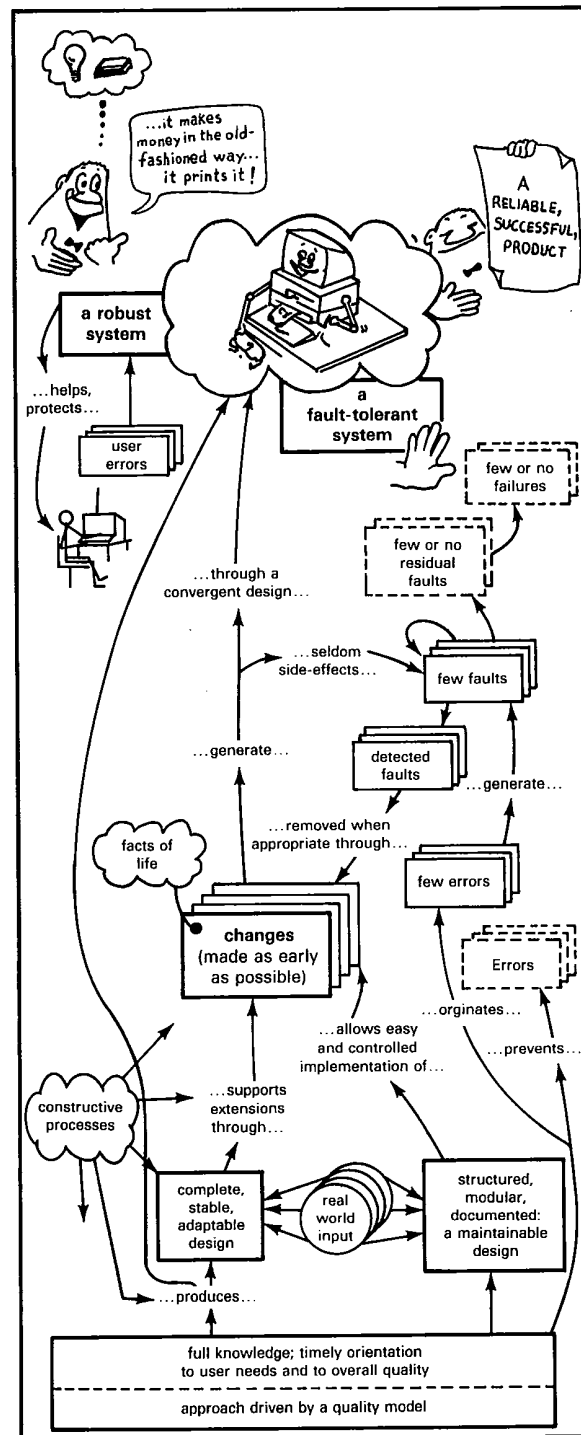


Figure 3.1.1-2 — Constructors of a Reliable Product

The analysis of the errors, faults, and failures from previous development and support processes can lead to improved future processes. While the exact functional relationships are not proven, it is through experience that the majority of failures are related to their origins. Examples include the following:

- 1) Incompletely defined user needs and requirements
- 2) Omissions in the design and coding process
- 3) Improper usage of the system
- 4) Excessive change activity

The error, fault, and failure analysis is detailed further in Section 6.

A process strategy to achieve reliable software includes activities, methods, tools, and measurements. Figure 3.1.1-3 shows the process strategy and the integrated role of measurements as the control window through which the quality of the product and of the process are monitored.

While Fig 3.1.1-3 is in large part self-explanatory, it is worthwhile to emphasize the major drivers for the process strategy. The strategy does not itself contain new techniques or new methodologies. It is a collection of known concepts, activities, and techniques, that can be used to drive a project toward better reliability.

The driving criteria of this constructive process strategy are as follows:

3.1.1.1 Do It Right the First Time

This approach stresses the use of all means necessary to define and build a product with no faults. “Doing it right the first time” must stress error prevention and fault avoidance through the following practices:

- 1) Employment of skilled, competent personnel is of foremost importance in building reliable products. It is important in all phases of development, especially the identification of user requirements and needs in the concept phase. The strategy stresses the necessity of intensely studying the stated and unstated needs of the user application, and of having sufficient experience in solving similar problems to apply to the present problem. For this reason, the concept phase is considered a major life cycle component toward a complete and stable definition of the objectives of a project. Quality work requires extensive knowledge of the application, which is often forgotten in the intermediate design phases, and remembered only when the user realizes some important features are missing during operation and maintenance.
- 2) Early user involvement in the conception of the product through techniques such as prototyping must be stressed to ensure that usability matches failure-freeness in importance.
- 3) Modern methods, languages and automated tools should be selected to promote productivity and quality in the activities of specification, design, coding and configuration management. The selected languages and tools must also support the verification activities through automation of requirements tracing, completeness and consistency analysis, performance simulation, complexity measurements, and configuration definition and control. Automation significantly reduces the risk of human errors.

3.1.1.2 Detect It Early; Fix It as Soon as Practicable

Detection and repair should be properly planned activities. Verification [1] is the process of determining whether or not the products of a given phase of the software development cycle fulfill the requirements established during the previous phase. Validation [1] is the process of evaluating software at the end of the software development process to ensure compliance with software requirements.

Early detection of faults throughout the life cycle through reviews, inspections, prototyping, simulation, or proof of correctness of the product will improve reliability. During the operations and maintenance phase, faults do not necessarily have to be fixed immediately. The severity of the problem, the frequency of occurrence of the problem, and the opportunity to manage and test the new configuration should weigh in the decision of when to fix a fault.

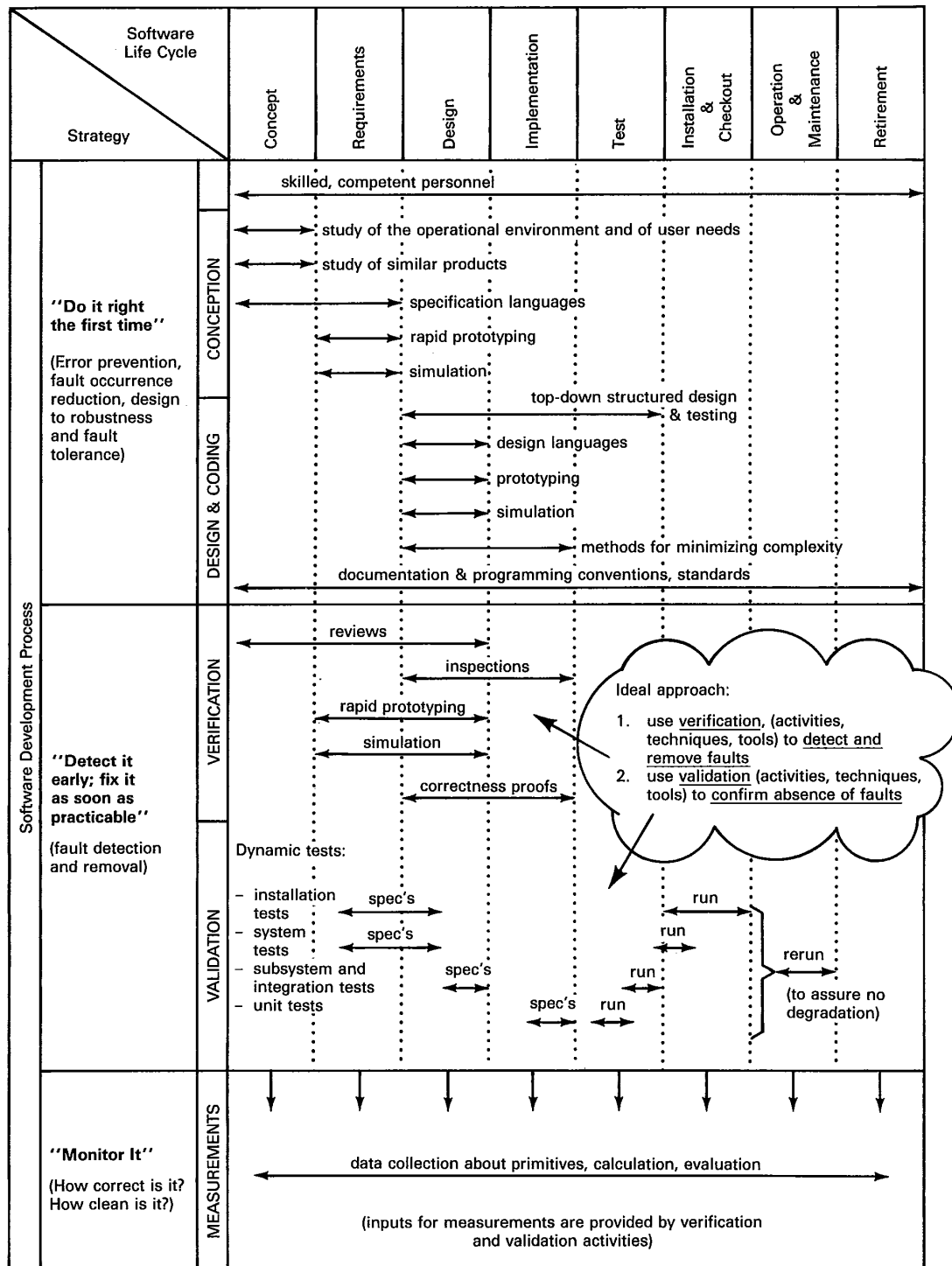


Figure 3.1.1-3 — Strategy of the Constructive Approach for Development of Reliable Software

Timely design of software tests, plans, and procedures is necessary to determine required resources and schedule. Early design of tests contributes to earlier detection of consistency and completeness faults, and ensures the testability of the requirements.

Reliability cannot be tested into a product. More testing does not necessarily mean better reliability. However, dynamic tests, the main tool for validation, have to be used cost-effectively with measures to provide the widest test coverage of a product. In the light of a constructive approach to reliable software, dynamic tests should be viewed as a tool for confirming the absence of faults rather than as the preferred tool for detecting them. When system test design is driven by user requirements, and when test samples are randomly selected based on defined probabilities of product inputs, testing contributes effectively to this constructive approach by providing an independent assessment of the software execution in representative operating environments. The costs of early verification efforts can be offset by savings from less repetition of dynamic tests, because most faults should be detected during verification.

3.1.1.3 Monitor It

Primitive data should be collected and evaluated throughout the life cycle.

Integration between measurements and verification and validation activities is of paramount importance. There cannot be good measurements without a well-organized verification and validation process that provides the input data for measurements.

Measurements evaluation is used not only for software quality evaluation, but also for fault detection. Each instance where the criterion used for evaluation indicates a “below threshold” value should be considered an instance of a software fault. For example, measures of processing times higher than a given value may be sufficient reason to reject the implemented code and to redesign it.

3.1.2 Reliability Optimization within Project Constraints

Software reliability should not be considered separately from other major factors: resources (cost), performance, and schedules. Software reliability should be optimized with respect to the project constraints. For example, if reliability and cost are considered as a ratio in which to attempt to increase the reliability and minimize the cost, then the project should seek the more efficient fault detection and removal activities. If reliability and schedule are jointly considered, early fault detection should be emphasized. Each project should assign priorities to these project factors and seek to optimize reliability within these constraints.

The cost of fixing faults rises significantly from the time they are generated to the time they are detected. It is more efficient to dedicate resources to fault detection and removal as soon after fault insertion as possible. Total cost comparisons for various levels of verification and validation efforts have shown that the savings of a verification and validation program are substantial. Figure 3.1.2-1 illustrates these considerations. Cases A and B have an equal cumulative amount of costs but different distribution of costs in time. In Case B the verification program puts greater emphasis, hence more resources, in early reviews, inspections, prototyping, and dynamic test design than in Case A.

3.1.3 Software Reliability in the Context of Product Quality

An analysis of the relationship of software reliability to other quality objectives is provided. The purpose of this discussion is to clarify the relationship among selected measures, software reliability and other quality objectives in general.

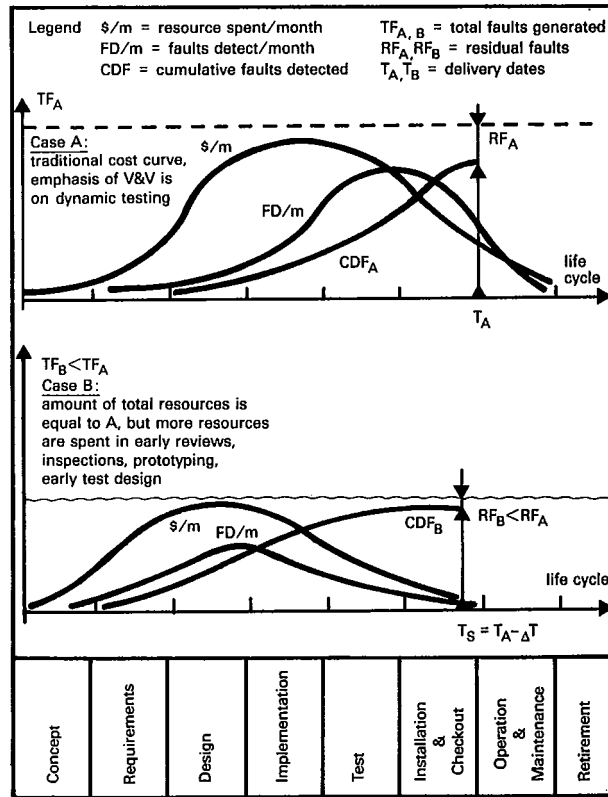


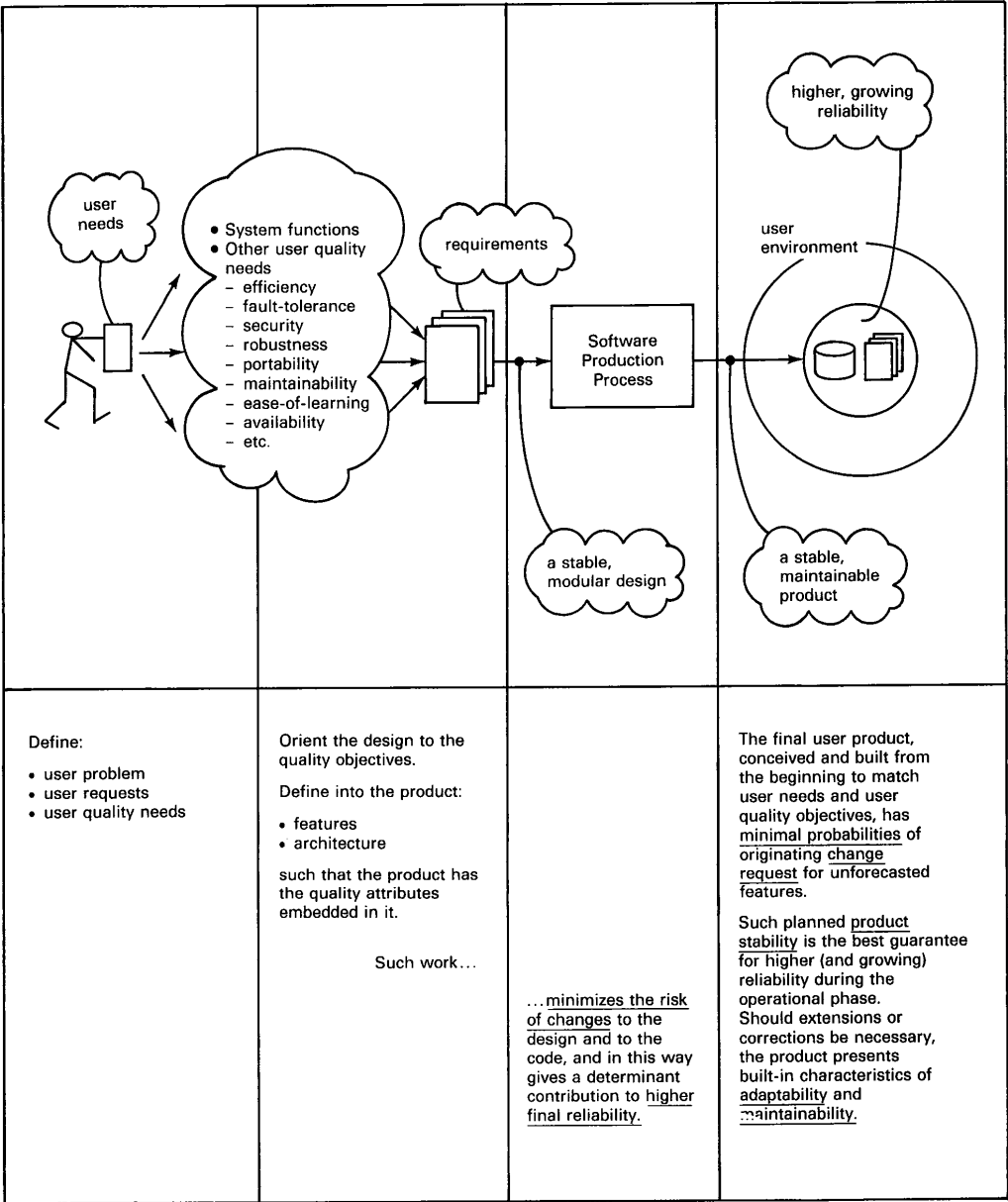
Figure 3.1.2-1 — Different Approaches in Verification and Validation Programs

3.1.3.1 Quality Models for Adaptability and Reliability

Change, a fact of life for system developers and maintainers, is one of the major indirect causes of faults in a software system. Change can impact not only reliability but the other major project drivers: schedule and cost.

A program to construct reliable software must consider not only the initial correctness of the product, but also the plan for accommodating change. It is important to strive to minimize reasons for change during the life cycle. Design stability and product stability are extremely important. But it is a reality that unforeseen changes in requirements, market forces, and other factors may introduce change into the process.

Change most often occurs in response to unmet quality objectives (such as, ease-of-use, maintainability, or portability) that were not given proper attention during design. Therefore, there is a need for a quality model that defines user-oriented needs. This model must be evolved during the same time period as function and performance objectives (Fig 3.1.3.1-1). Such a quality model is the premise for both a complete and an adaptable design that can easily absorb changes due to unforeseen new requirements.



*During the operational phase, the quality model for software reliability is a function of product stability and adaptability.

Figure 3.1.3.1-1 — Quality Model for Software Reliability

Software reliability is a major quality objective. Rarely is medium or low reliability acceptable. Software reliability is both an individual quality attribute and one that is interdependent on other user quality attributes. It is because of this duality that we must consider software reliability in context with other user quality objectives. It is for this reason that user quality objectives are within the scope of this guide.

3.1.3.2 User Quality Objectives and Reliability

User quality objectives should influence the system design and serve as criteria both for intermediate verification and for system validation and acceptance. The defined user quality objectives give the most direct means for fault search and detection. Examples of user quality objectives are ease-of-use for a word processor; ultra-high reliability for a telephone exchange plant; reliability and security for a banking application; safety and robustness for a nuclear power plant system.

User quality objectives are reached through intermediate software engineering objectives, which are achieved in the earlier phases of system development. Examples of intermediate software engineering objectives include complexity, traceability, modularity, topological coverage, defect index, and fault-days. These intermediate software engineering objectives are rarely of direct interest to the user. They constitute the tactics through which the software engineer achieves user quality objectives. Figure 3.1.3.2-1 gives a global, but not exhaustive, view of the mix of both user and software engineering objectives that have an impact on software reliability. The figure presents a schema for organizing quality objectives in each life cycle phase. Specific attention should be given to those intermediate quality objectives that directly affect project selected user quality objectives.

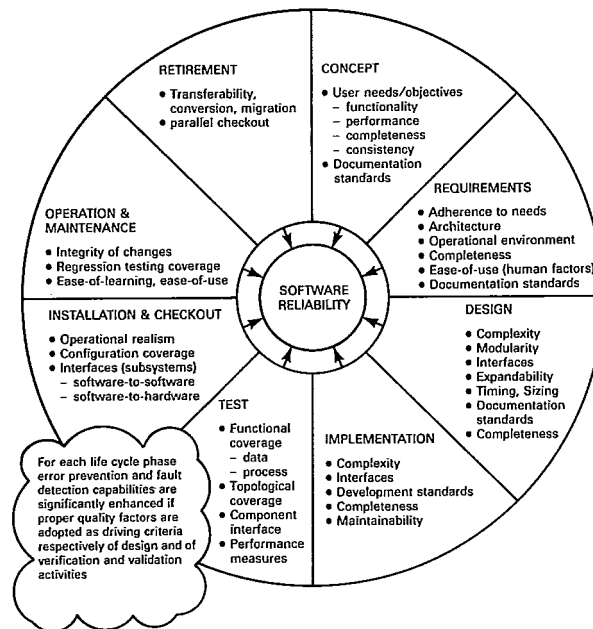


Figure 3.1.3.2-1 — Quality Factors Impacting Software Reliability

A software reliability management program requires the establishment of a balanced set of user quality objectives, and identification of intermediate quality objectives that will assist in achieving the user quality objectives. Resources should be spent commensurate with priority of objectives. Methods for measuring intermediate quality objectives related to reliability are identified in the Appendix of this guide.

A similar approach is applicable for the larger class of quality objectives in general and the maintenance activity specifically. Once software is delivered, the maintenance activity applied to that software has many

of the same attributes that are associated with a new development effort. Thus, many of the same quality and reliability objectives are applicable.

A balanced quality program is both necessary and instrumental for a constructive approach to reliable software.

3.2 Measurement Environment

Measurement is the comparison of a property of an object to a similar property of a standard reference. Measurements are effective when they are used either to orient decisions, to define corrective actions, or to get a better understanding of causal relationships between intended expectations and observed facts. If there is no pre-stated objective on how the results will be used, there should be no measurements.

When measurements are used as process diagnostics, there must be a proper environment for data collection, analysis, and corrective action in a closed loop system. Measurement information must flow throughout the process, supporting integration of both technical and managerial activities (Fig 3.2-1).

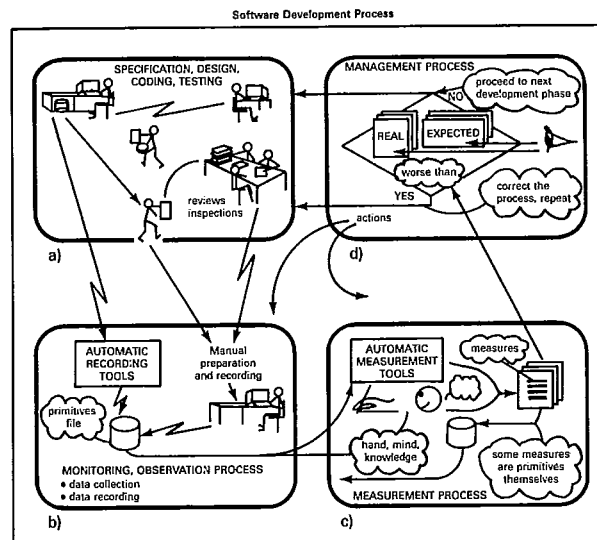


Figure 3.2-1 — Measurement Information Flow: An Integral Part of Development Process

The building blocks for deriving more complex measures from collected data are called primitives. Some primitives are derived directly from the product (specifications, source code, object code). Examples of these are statements, edges, nodes, branches, changes, capabilities, networks, and goals. Other primitives, produced by verification and validation activities, include errors, faults and failures.

A two-step process is recommended to prepare for data collection:

- 1) For each potential measure selected from IEEE Std 982.1-1988 [2], determine the data needed to perform the measurement from the list of primitives.

- 2) Develop project practices for organizing data so information related to events during the development effort can be stored in a data base and retained for historical purposes and analysis. Automated tools integrated with the development environment should be considered.

Patient, accurate data collection and recording is the key to producing accurate measurements. If not well-organized, data collection can become very expensive. To contain these costs, the information gathering should be integrated with verification and validation activities. Appropriate mechanisms, such as forms or direct terminal entry, should be used to record the results of reviews. Use of automated tools for data retrieval and data entry can also reduce costs. Properly augmented configuration change control systems may furnish an ideal support environment for recording and evaluating most of the data for reliability assessment.

An important outgrowth of implementing a measurement effort is the ability to evaluate the results of collected data, develop measures, and apply analysis of the results to the end product. By tracking progress and evaluating the results, the measurement effort can be used to assess product reliability growth as well as the effectiveness of the development and support processes.

3.3 Measurement Selection Criteria

Current technology in software reliability is limited largely to the use of models that rely exclusively on test data applied to software at the end of development. The measures selected for the standard were extracted from available literature and from the collective experience of practitioners in the field. The primary emphasis was to identify a varied set of measures that cover the many aspects of the concept of reliability. This led to the inclusion of faults and failures as well as models for the traditional reliability calculations.

The constructive approach expanded the search to measures that promote reliability by minimizing residual faults through error prevention and early fault detection and removal. Therefore, the standard includes product measures to determine how carefully a product was verified (e.g., completeness and traceability) and complexity measures to accommodate the possibility of change to the system.

Careful consideration was given to the selection of process measures that monitor the quality of the development process and identify the need to make changes early in the process. Some process measures monitor the accuracy and completeness of development and test activities through examination of product properties (such as completeness and consistency). Other process measures monitor the efficiency of the activities (e.g., error distribution). The process measures have the potential to stimulate adoption of good engineering practices at the proper project phase, specifically encouraging early fault detection.

A key consideration in the selection of the measures was to address the full range of the life cycle, a varied audience of both users and software and systems engineers, and a range of audience skills in reliability measurement.

The user class, which may include the owner, the operator and the recipient of the system capability, is more interested in measures related to system availability and operability, such as mean time to failure. The software engineer is more interested in software faults that cause excessive failure rates.

Reliability measures take on both aspects - system failures and software faults. The user can focus on system availability and capability; whereas the engineer can focus on the fault in the software product. Other measures, such as complexity, are of interest to managers and developers, assisting them to improve processes and control costs, schedules, personal productivity, and professional skills.

From the point of view of skill required, the measures are arranged in increasing order of difficulty. This allows individuals with little knowledge of computational methodologies to start making meaningful measurements. Subsequent measures depend upon increasing experience and sophistication with computational methodologies and automated assists.

4. Measure Organization and Classification

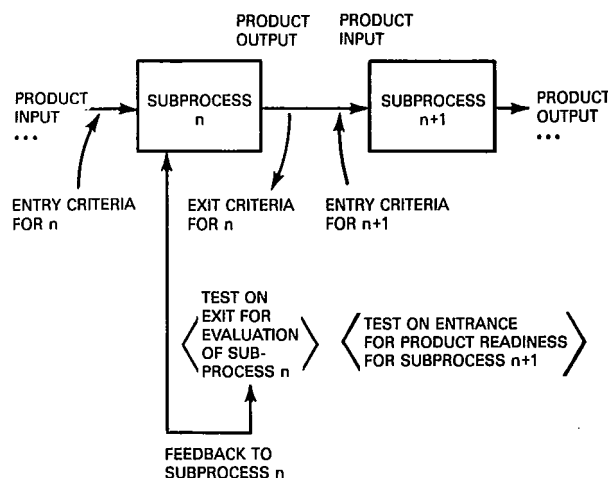
The measures in this guide are organized by the two basic objectives of reliability. The first objective is product maturity assessment - a certification of product readiness for operation (including an estimation of support necessary for corrective maintenance). The second objective of reliability is process maturity assessment - an ongoing evaluation of a software factory's capability to produce products of sufficient quality for user needs. Process maturity assessment includes product maturity assessment with the objective of process repair when necessary.

With hardware factories, process maturity assessment is usually performed through the technique of statistical process control. In software, this technique has been hampered by the lack of clearly defined measurables to relate product quality deficiencies to process cause. The measures in this guide provide the means to apply statistical process control to software.

The software life cycle can be modeled and refined into a sequence of subprocesses. Each subprocess has inputs that should meet well-defined entry criteria and outputs that should meet well-defined exit criteria.

An assessment of the product at the completion of a subprocess (the output against the exit criteria) not only evaluates the readiness of that individual product to enter the next subprocess, it serves as an indicator of the quality of the sub-process itself. It should be noted that failure of a product to pass the established exit criteria is a signal for further analysis. Either the subprocess could be defective or the criteria could be too stringent. Corrective action should be taken only after sufficient analysis (Fig 4-1).

For each measure the following topics are detailed in the Appendix : implementation, interpretation, considerations, training, an example, benefits, experience, and references.



NOTE: Entry criteria n+1 might differ from exit criteria n since owners of subprocess n, entry criteria n, and exit criteria n could be separate organizations from those of n+1.

Figure 4-1 — Generalized Process Control

These measures can be divided into two functional categories: product and process. Product measures are applied to the software objects produced and are divided into six subcategories. Process measures are applied to the life cycle activities of development, test and maintenance and are divided into three subcategories.

The functional categories are as follows:

Product Measures

Errors, faults, and failures
Mean time to failure; failure rate
Reliability growth and projection
Remaining product faults
Completeness and consistency
Complexity

Process Measures

Management control
Coverage
Risk, benefit, cost evaluation

In addition to the functional classification, the measures are also classified by the software life cycle phases in which their use is appropriate. Typical software life cycle phases based upon IEEE Std 729-1983, IEEE Standard Glossary of Software Engineering Terminology [1] are used to illustrate when each of the measures may be applied. The life cycle phases are as follows:

Concepts
Requirements
Design Implementation
Test Installation and Checkout
Operation and Maintenance
Retirement

Finally, the measures can be classified as indicators or predictors.

4.1 Functional Classification

The measure classification matrix (Table 4.1-1) is a cross index of measures and functional categories. It can be used to select measures applicable to each category. For each functional category, a matrix is presented that cross-references the measures with the phases of the software life cycle in which they are applicable (Tables 4.2-1 through 4.2-9).

Table 4.1-1 includes an experience rating of 0, 1, 2, or 3. A “0” indicates that the measure has been formalized but has not been sufficiently validated to have operational experience. A “1” means that the measure has had limited use in the software community. This could be because the measure has been used only within an organization and has not had industry-wide exposure, or because the measure is not sufficiently well known to have been used on many projects. A “2” means that the measure has had moderate experience and a “3” means that it has had extensive experience. In no way does the experience rating imply that one measure is better than another.

Table 4-1 — Category: Errors, Faults, Failures Counting

Measures	Life Cycle Phase							
	Concept	Requirements	Design	Implementation	Test	Installation & Checkout Installation	Operation & Maintenance	Retirement
1. Fault density	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
2. Defect density	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
3. Cumulative failure profile	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
4. Fault-days number	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
7. Requirements traceability		Δ	Δ				Δ	
8. Defect indices	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
12. Number of conflicting requirements		Δ					Δ	
23. Requirements compliance		Δ					Δ	

Table 4.1-1 — Measure Classification Matrix

Measures (Experience)	Product Measures					Process Measures		
	Errors, Faults, Failures	Mean Time to Failure; Failure Rate	Reliability Growth & Projection	Remaining Product Faults	Completeness & Consistency	Complexity	Management Control	Coverage
1. Fault density (2)	X							
2. Defect density (3)	X							
3. Cumulative failure profile (1)	X							
4. Fault-days number (0)	X						X	
5. Functional or modular test coverage (1)					X			X
6. Cause and effect graphing (2)					X			X
7. Requirements traceability (3)	X				X			X
8. Defect indices (1)	X						X	
9. Error distribution(s) (1)							X	
10. Software maturity index (1)			X					X
11. Man hours per major defect detected (2)							X	
12. Number of conflicting requirements (2)	X				X			X
13. Number of entries/exists per module (1)				X	X			X
14. Software science measures (3)					X			X
15. Graph-theoretic complexity for architecture (1)					X			X
16. Cyclomatic complexity (3)					X			X
17. Minimal unit test case determination (2)					X			X
18. Run reliability (2)			X					
19. Design structure (1)						X		
20. Mean time to discover the next K faults (3)								X
21. Software purity level (1)			X					
22. Estimated number of faults remaining (seeding) (2)				X				
23. Requirements compliance (1)	X				X			X
24. Test coverage (2)					X			X
25. Data or information flow complexity (1)						X		
26. Reliability growth function (2)			X					
27. Residual fault count (1)			X	X				
28. Failure analysis using elapsed time (3)			X	X				X
29. Testing sufficiency (0)			X					
30. Mean-time-to-failure (3)		X						
31. Failure rate (3)		X						
32. Software documentation & source listings (2)					X			
33. RELY - (Required Software Reliability) (1)								X
34. Software release readiness (0)								X
35. Completeness (2)				X	X			
36. Test accuracy (1)				X	X			X
37. System performance reliability (2)			X					
38. Independent process reliability (0)			X					
39. Combined HW/SW system operational availability (0)			X					

4.1.1 Product Measures

The product measures address cause and effect of the static dynamic aspects of both projected reliability

prior to operation, and operational reliability. As an example, reliability may change radically during the maintenance effort, due to the complexity of the system design. The six product measure sub-categories address these dimensions of reliability.

- 1) Errors, faults, failures—Count of defects with respect to human cause, program bugs, and observed system malfunctions.
- 2) Mean-time-to-failure; failure rate—Derivative measures of defect occurrence and time.
- 3) Reliability growth and projection—The assessment of change in failure-freeness of the product under testing and in operation.
- 4) Remaining products faults—The assessment of fault-freeness of the product in development, test, or maintenance.
- 5) Completeness and consistency—The assessment of the presence and agreement of all necessary software system parts.
- 6) Complexity—The assessment of complicating factors in a system.

4.1.2 Process Measures

The process measures address cause and effect of both the static and dynamic aspects of the development and support management processes necessary for maximizing quality and productivity. The three process measure subcategories address the process dimension of reliability.

- 1) Management control measures address the quantity and distribution of error and faults and the trend of cost necessary for defect removal.
- 2) Coverage measures allow one to monitor the ability of developers and managers to guarantee the required completeness in all the activities of the life-cycle and support the definition of corrective actions to carry the coverage indicators to the desired levels.
- 3) Risks, benefits, and cost evaluation measures support delivery decisions based both on technical and cost criteria. Risk can be assessed based on residual faults present in the product at delivery and the cost associated with the resulting support activity.

Table 4.2-1—Life Cycle Classification Matrix: Errors, Faults, Failures Counting

Measures	Life Cycle Phase							
	Concept	Requirements	Design	Implementation	Test	Installation & Checkout Installation	Operation & Maintenance	Retirement
1. Fault density	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
2. Defect density	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
3. Cumulative failure profile	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
4. Fault-days number	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
7. Requirements traceability		Δ	Δ				Δ	
8. Defect indices	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
12. Number of conflicting requirements		Δ					Δ	
23. Requirements compliance		Δ					Δ	

Table 4.2-2—Life Cycle Classification Matrix: Mean-Time-to-Failure; Failure Rate

Measures	Life Cycle Phase							
	Concept	Requirements	Design	Implementation	Test	Installation & Checkout Installation	Operation & Maintenance	Retirement
30. Mean time to failure					Δ	Δ	Δ	
31. Failure rate					Δ	Δ	Δ	

Table 4.2-3—Life Cycle Classification Matrix: Reliability Growth and Projection

Measures	Life Cycle Phase							
	Concept	Requirements	Design	Implementation	Test	Installation & Checkout Installation	Operation & Maintenance	Retirement
10. Software maturity index	Δ	Δ	Δ			Δ	Δ	
18. Run reliability					Δ	Δ	Δ	
21. Software purity level					Δ	Δ	Δ	
26. Reliability growth function					Δ	Δ	Δ	
28. Failure analysis using elapsed time					Δ	Δ	Δ	
29. Testing sufficiency					Δ	Δ		
30. Mean time to failure					Δ	Δ	Δ	
37. System performance reliability		Δ	Δ	Δ	Δ	Δ	Δ	
38. Independent process reliability						Δ	Δ	
39. Combined HW/SW operational availability					Δ	Δ	Δ	

Table 4.2-4—Life Cycle Classification Matrix: Remaining Faults Estimation

Measures	Life Cycle Phase							
	Concept	Requirements	Design	Implementation	Test	Installation & Checkout Installation	Operation & Maintenance	Retirement
14. Software science measures				Δ			Δ^*	
22. Estimated number of faults remaining (seeding)					Δ	Δ	Δ	
27. Residual fault count					Δ	Δ	Δ	
28. Failure analysis using elapsed time					Δ	Δ	Δ	
36. Test accuracy					Δ			
NOTE— * If source code is changed								

Table 4.2-5—Life Cycle Classification Matrix: Completeness, Consistency

Measures	Life Cycle Phase							
	Concept	Requirements	Design	Implementation	Test	Installation & Checkout Installation	Operation & Maintenance	Retirement
5. Functional or modular test coverage					Δ	Δ	Δ	
6. Cause and effect graphing		Δ	Δ	Δ	Δ		Δ	
7. Requirements traceability		Δ	Δ				Δ	
12. Number of conflicting requirements		Δ					Δ	
16. Cyclomatic complexity (testing phase)				Δ	Δ		Δ	
23. Requirements compliance		Δ					Δ	
24. Test coverage		Δ	Δ		Δ	Δ	Δ	
32. Software documentation and source listings				Δ	Δ	Δ	Δ	
35. Completeness		Δ	Δ				Δ	
36. Test accuracy					Δ			

Table 4.2-6—Life Cycle Classification Matrix: Management Control

Measures	Life Cycle Phase							
	Concept	Requirements	Design	Implementation	Test	Installation & Checkout Installation	Operation & Maintenance	Retirement
13. Number of entries/exits per module			Δ	Δ			Δ	
14. Software science measures				Δ			Δ	
15. Graph-theoretic complexity for architecture		Δ	Δ				Δ	
16. Cyclomatic complexity			Δ	Δ			Δ	
17. Minimal test case determination				Δ	Δ		Δ	
19. Design structure			Δ				Δ	
25. Data or information flow complexity			Δ	Δ			Δ	

Table 4.2-7—Life Cycle Classification Matrix: Management Control

Measures	Life Cycle Phase							
	Concept	Requirements	Design	Implementation	Test	Installation & Checkout Installation	Operation & Maintenance	Retirement
4. Fault-days number	Δ	Δ	Δ	Δ	Δ	Δ	Δ	
8. Defect indices		Δ	Δ	Δ	Δ	Δ	Δ	
9. Error distribution(s)		Δ	Δ	Δ	Δ	Δ	Δ	
11. Man-hours per major defect detected		Δ	Δ	Δ	Δ	Δ	Δ	
20. Mean time to discover next K faults					Δ	Δ	Δ	

Table 4.2-8—Life Cycle Classification Matrix: Coverage

Measures	Life Cycle Phase							
	Concept	Requirements	Design	Implementation	Test	Installation & Checkout Installation	Operation & Maintenance	Retirement
5. Functional or modular test coverage					Δ	Δ	Δ	
6. Cause and effect graphing		Δ	Δ	Δ	Δ		Δ	
7. Requirements traceability		Δ	Δ				Δ	
12. Number of conflicting requirements		Δ					Δ	
23. Requirements compliance		Δ					Δ	
24. Test coverage		Δ	Δ	Δ	Δ	Δ	Δ	
29. Testing sufficiency					Δ		Δ	
32. Software documentation & source listings				Δ	Δ	Δ	Δ	
33. RELY - Required Software Reliability	Δ	Δ	Δ	Δ	Δ	Δ	Δ	
35. Completeness		Δ	Δ				Δ	
36. Test accuracy					Δ			

Table 4.2-9—Life Cycle Classification Matrix: Risk, Benefit, and Cost Evaluation

Measures	Life Cycle Phase							
	Concept	Requirements	Design	Implementation	Test	Installation & Checkout Installation	Operation & Maintenance	Retirement
5. Functional or modular test coverage					Δ	Δ	Δ	
10. Software maturity index	Δ	Δ	Δ			Δ	Δ	
11. Man-hours per major defect detected		Δ	Δ	Δ	Δ	Δ	Δ	
14. Software science measures				Δ			Δ	
18. Run reliability					Δ	Δ	Δ	
19. Design structure			Δ					
20. Mean time to discover the next <i>K</i> faults					Δ	Δ	Δ	
21. Software purity level					Δ	Δ	Δ	
27. Residual fault count					Δ	Δ	Δ	
31. Failure rate					Δ	Δ	Δ	
33. RELY - Required Software Reliability	Δ	Δ	Δ	Δ	Δ	Δ	Δ	
34. Software release readiness					Δ	Δ		

4.2 Life Cycle Classification

To assist the user in more practical considerations, such as “When are the measures applied?” the life cycle classification (Fig 4.2-1) has been developed. The life cycle has been divided into early, middle, and late segments. In the early segment, the measures relate to the potential causes of system reliability. The middle segment measures relate to the reduction of process errors that can improve the efficiency of software development. The late segment measures relate to actual system performance reliability.

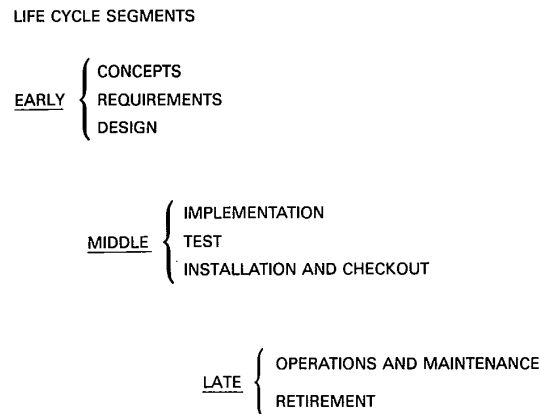


Figure 4.2-1 — Life Cycle Classification

For each life cycle phase a set of possible measures that might be applied during the development effort should be identified. Selection need not be limited to one measure per development phase, but on the other hand, it is expected that overzealous selection of measures would not benefit a project.

4.2.1 Early Life Cycle Segment

The early segment - concepts, requirements, and design - is the period when many of the characteristics of the system, including potential reliability, are determined. Problems in these early phases tend to result from premature constraints on the system design that limit system solution alternatives and affect the real system objectives, which include reliability. Emphasis and focus of the measures in the early stages are on the goals and objectives of the system. The concrete feedback they provide aids in achieving completeness and consistency in the definition of user requirements as well as the design.

4.2.2 Middle Life Cycle Segment

The middle segment of the software life cycle includes implementation, test, installation and checkout. The software engineer's goal in this segment is to develop fault-free software efficiently. The objectives of reliability measurement in this segment include not only reliability projection, but also guidance of the process so that fault-free software is developed efficiently. For reliability projection, faults found in this segment can relate directly to operational failures, especially when development activity is halted prematurely. These faults, in fact, can be used to determine software readiness. For the process evolution objective, these faults can be considered process errors and can be directly related to the effectiveness and efficiency of the previously applied processes. Cause-analysis can relate the fault to that point in the process where the error was made. Proper process-control techniques applied to these measures can indicate if and when action should be taken to change the process.

4.2.3 Late Life Cycle Segment

The late segment of the software life cycle - operation, maintenance, and retirement - is where system performance, as an expression of reliability and availability, can be directly measured using the traditional software reliability measures. Current software product performance can be used to project future performance. In addition, future development can be improved by relating the problems in the current software product to the steps in the development process that produced them.

In this segment there are two audiences for software reliability measures - the user and the software engineer. The users are the owner, the operator, and the recipient of the end product of the system capability. The software engineers include both the original developer and the subsequent maintainer. In most cases, the user is more interested in system availability and operability; the software engineer is more interested in the origins of software faults that cause failures. The user focuses on the capability of the system; the software engineer focuses on the faults in the restricted software domain.

Reliability measures should address both system failures and software faults to provide useful information to both user and software engineer. There are traditional measures of software performance that provide both a user and a software engineer view of system reliability. Many of these measures - for example, mean-time-to-failure (user) and software fault discovery rate (software engineer) - serve as primitives for reliability projections based on modeling techniques.

4.3 Indicators and Predictors

Measures can also be organized with respect to their primary function as indicators or predictors. Indicator measures provide an assessment of reliability at the point in time the measurement is taken. Predictor measures project forecasted behavior as suggested by past behavior. Historical data based upon consistently applied indicator measures are the basic input for predictor measures. The higher the integrity and quantity of the historical record, the better results can be expected from the predictor measures. Note that some measures are both predictor and indicator in nature.

Indicator measures can be used to determine completion of activities throughout the life cycle. In this context, established goals are considered successfully achieved based on the value of the indicator measure. Using the history of indicator measurements, future behavior can be determined based upon predictor measures.

Table 4.3-1 provides a cross reference of measures - first, to the object or events that are subjects of measurement, and second, to the primary indicator or predictor nature of the measure.

Table 4.3-1 — Indicator/Predictor Measure Classification

Object/Event	Measure
Requirements	
Indicator	4.6, 4.7, 4.12, 4.23, 4.33, 4.35
Predictor	—
Design	
Indicator	4.13, 4.15, 4.19, 4.25
Predictor	—
Source code	
Indicator	4.16, 4.17
Predictor	4.14
Test set	
Indicator	4.5, 4.24
Predictor	—
Any product	
Indicator	4.32
Predictor	—
Software faults	
Indicator	4.1, , 4.4, 4.8, 4.9, 4.11
Predictor	4.22, 4.29, 4.36
Software failures	
Indicator	4.3
Predictor	4.18, 4.20, 4.21, 4.26, 4.28, 4.30, 4.31, 4.39
Release package	
Indicator	4.10, 4.34
Predictor	—
Performing object code	
Indicator	4.37
Predictor	—
Tested object modules	
Indicator	4.38
Predictor	—

5. Framework for Measures

Software reliability measurements take place in an environment that includes user needs and requirements, a process for developing products meeting those needs, and user environment within which the delivered software satisfies those needs. This measurement environment establishes a framework for determining and interpreting indicators of software reliability.

5.1 Measurement Process

In this section, a measurement process is detailed that can constructively affect the delivered reliability of software through the selection and interpretation of indicators of software reliability. This process formalizes the data collection practices in both development and support. It provides for product evaluation at major milestones in the life cycle. Furthermore, it relates measures from one life cycle phase to another. In summary, this process provides the basis for reliability measurement of a product.

5.2 Stages of a Measurement Process

The software reliability measurement process can be described in nine stages (Fig 5.2-1). These stages may overlap or occur in different sequences depending on organization needs. These stages provide a useful model for describing measurement process issues. Each of these stages in the measurement process influences the production of a delivered product with the potential for high reliability. Other factors influencing the measurement process include the following: a firm management commitment to continually assess product and process maturity, or stability, or both during the project; use of trained personnel in applying the measures to the project in a useful way; software support tools; and a clear understanding of the distinctions between errors, faults, and failures.

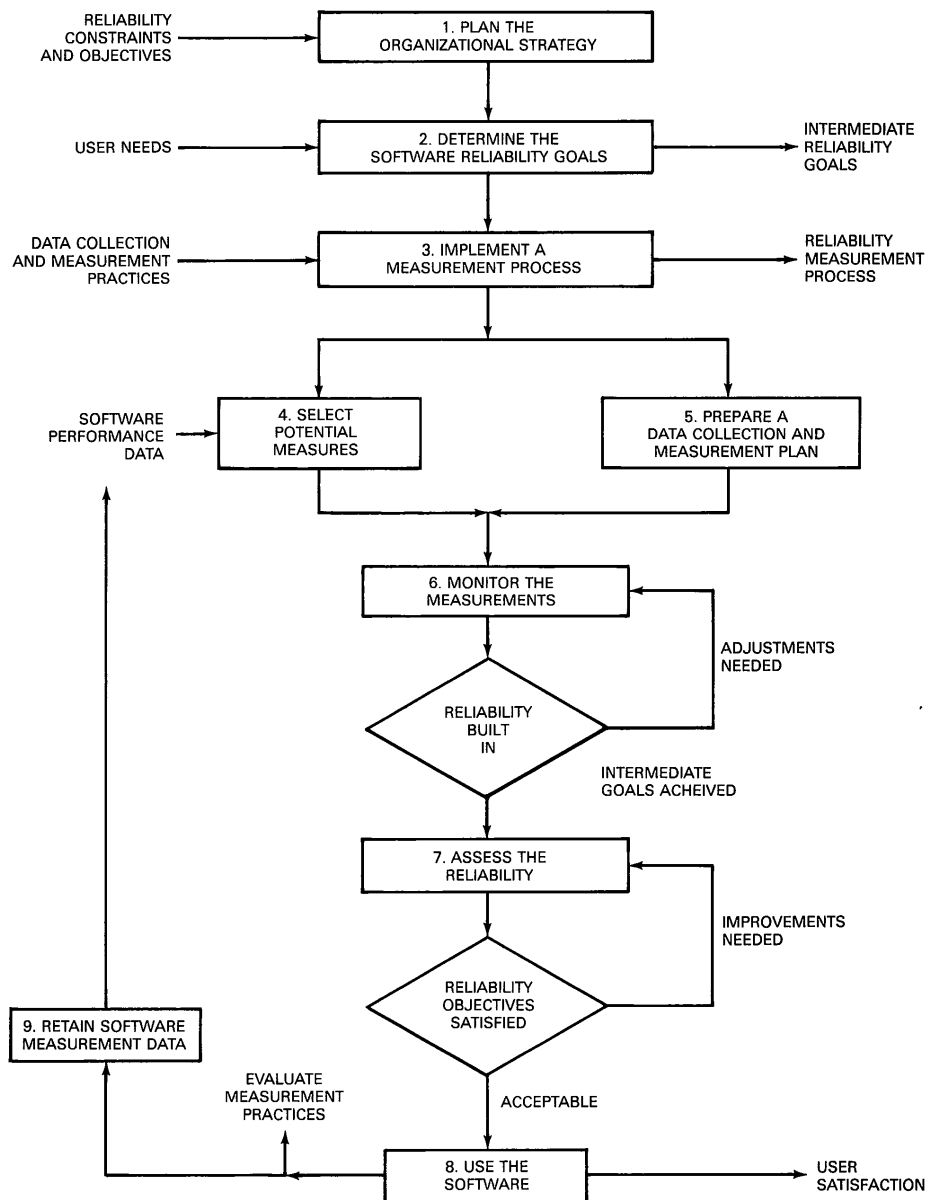


Figure 5.2-1 — Reliability Measurement Process

5.2.1 Stage 1: Plan Organizational Strategy

Initiate a planning process. Form a planning group and review reliability constraints and objectives, giving consideration to user needs and requirements. Identify the reliability characteristics of a software product necessary to achieve these objectives. Establish a strategy for measuring and managing software reliability. Document practices for conducting measurements.

5.2.2 Stage 2: Determine Software Reliability Goals

Define the reliability goals for the software being developed in order to optimize reliability in light of realistic assessments of project constraints, including size, scope, cost, and schedule.

Review the requirements for the specific development effort, in order to determine the desired characteristics of the delivered software. For each characteristic, identify specific reliability goals that can be demonstrated by the software or measured against a particular value or condition. Establish an acceptable range of values. Consideration should be given to user needs and requirements, including user perception of software reliability and the system environment in which the software must interact.

Establish intermediate reliability goals at various points in the development effort to assist in achieving the reliability goals.

5.2.3 Stage 3: Implement Measurement Process

Establish a software reliability measurement process that best fits an organization's needs.

Review the process described in stages 4 through 8 and select those stages that best lead to optimum reliability. Add to or enhance these stages as needed to reflect the specific work environment. Consider the following suggestions when establishing the measurement process:

- 1) Select appropriate data collection and measurement practices designed to optimize software reliability.
- 2) Document the measures required, the intermediate and final milestones when measurements are taken, the data collection requirements, and the acceptable values for each measure.
- 3) Assign responsibilities for performing and monitoring measurements, and provide necessary support for these activities from across the internal organization.
- 4) Initiate a measure selection and evaluation process.
- 5) Prepare educational materials for training personnel in concepts, principles, and practices of software reliability and reliability measures.

After selecting and establishing the steps of the measurement process, reduce it to an organization operation procedure. Maintain complete records of the measurement process to provide a historical perspective so that future reliability measurement programs may benefit.

5.2.4 Stage 4: Select Potential Measures

Identify potential measures that would be helpful in achieving the reliability goals established in Stage 2 .

Once the organization software reliability goals are established, select the measures best suited to the development and support environments.

Use the classification scheme described in Section 4 to determine which category or categories of reliability measurements would be most useful to the project. When possible, measures should be chosen within a particular category to show a continuous relationship between early indicators and the final software reliability achieved at delivery.

Measures should be selected to span categories and development phases. Selection need not be limited to one measure per category or phase. However, overzealous selection of measures should be avoided.

The classification scheme assists in identifying and applying related measures across a development project (commonality) or in applying the same measure(s) to several projects (comparison).