Management Indicators: Assessing Product Reliability and Maintainability

Constance V. Rosson

TR 88-40

Technical Report SRC-88-011

MANAGEMENT INDICATORS: ASSESSING PRODUCT RELIABILITY AND MAINTAINABILITY

Constance V. Rosson

Systems Research Center
Virginia Tech
Blacksburg, VA 24061

August 1988

This research was supported in part by the Naval Sea Command under Contract No. N60921-83-G-A165-B031 through the Systems Research Center at Virginia Tech.

Abstract

This report discusses the role of Management Indicators in validating the predictive capability of the bottom-up evaluation process, which is defined by the Procedural Approach to the Evaluation of Software Development Methodologies. The bottom-up evaluation process provides a framework for determining the extent to which software engineering objectives, e.g., reliability and maintainability, are present in a software product from a design perspective of the code and supporting documentation. The bottom-up evaluation process is observed to be a predictor of the extent to which the objectives are realized in the post-developed product.

Employment of the bottom-up evaluation process to determine the extent to which the objectives are present in the product is accomplished by the utilization of Design Indicators. Management Indicators are proposed as a counterpart to Design Indicators and enable one to measure the extent to which the objectives are realized in a developed product. While Design Indicators focus on design structure characteristics of the product, Management Indicators focus on the acquisitional, behavioral, and maintenance characteristics. To accomplish the validation of the predictive capability, the correlation between the values obtained by utilizing Design Indicators and those obtained by utilizing Management Indicators must be investigated. The author has chosen to study and present the software engineering objectives of reliability and maintainability as they related to a future validation effort.

CR Categories and Subject Descriptions: D.2.9 [**Software Engineering**]: Software Quality Assurance; K.6.3 [**Software Management**] -- development, maintenance, selection; K.6.4 [**Software Quality Assurance**]

General Terms: Assessing software quality, design indicators, management indicators, assessing reliability and maintainability

Additional Key Words and Phrases: Faults, failures, defect, error, software acquisition, behavioral indicators, acquisitional indicator

Table of Contents

1. Introduction 1

1.1. Problem Statement 5

1.2. Solution Approach 5

2. Background Concepts and Terminologies 8

2.1. The Procedural Approach to the Evaluation of Software Development Methodologies . . . 8

2.1.1. Linkages 9

2.1.2. Assessment Capabilities 16

2.2. Current Evaluation of Software Quality 18

2.3. Concepts and Terminologies 21

2.3.1. Reliability 21

2.3.2. Maintainability 22

2.3.3. Software Life Cycle 23

2.3.4. Faults, Failures, Defects, and Errors 25

2.3.5. Review and Inspection Processes 26

2.3.6. Software Trouble Reports 29

3. A Comparison of Design and Management Indicators Relative to Software Quality Assessment 31

3.1. Design Indicators 32

3.2. Management Indicators 35

3.2.1. Acquisitional Indicators 38

3.2.2. Behavioral Indicators 42

3.2.2.1. Non-affective Behavioral Indicators 45

3.2.2.2. Affective Behavioral Indicators 49

3.2.3. Evolutionary View of Management Indicators 53

3.2.4. Multiple View of Management Indicators 53

3.3. Relationship between Design and Management Indicators 54

4. Concluding Remarks 59

4.1. Summary 59

4.2. Author's Contribution 62

4.3. Future Directions 63

Bibliography 68

Appendix A. Management Indicators of Software Reliability and Maintainability 72

Defect Removal Rate 74

Defect Age Profile 80

Defect Density 84

Defect Detection Efficiency 88

Extent of Modification 93

Defect Introduction 97

Defect Days 99

Defect Index 103

Defect Distributions 106
Maturity Index 109
Person-Hours per Major Defect Detected 113
Person-Hours per Source Statement Modified 116
Requirements Traceability 118
Test Coverage 121
Robustness 125
Mean Time to Failure 127
Mean Time to Repair 131
Input Index 135
Failure Rate 138
Availability 142
Appendix References 146

Vita 147

List of Illustrations

Figure 1. Objectives, Principles, and Attributes . . . 3
Figure 2. Linkages . . . 17
Figure 3. Motivation for Early Detection of Defects . . . 28
Figure 4. Management Indicators . . . 39
Figure 5. A Behavioral Indicator . . . 44
Figure 6. Management Indicators . . . 46
Figure 7. A Non-affective Indicator . . . 48
Figure 8. An Affective Indicator . . . 52
Figure 9. Design vs. Management Indicators . . . 57
Figure 10. Defect Removal Rate . . . 75
Figure 11. Defect Removal Rate . . . 78
Figure 12. Defect Age Profile . . . 81
Figure 13. Defect Density . . . 86
Figure 14. Defect Detection Efficiency . . . 90
Figure 15. Defect Detection Efficiency . . . 91
Figure 16. Extent of Modification . . . 96
Figure 17. Defect Distributions . . . 107
Figure 18. Maturity Index . . . 111
Figure 19. Test Coverage . . . 124
Figure 20. Mean Time to Failure (MTTF) . . . 129
Figure 21. Mean Time to Repair (MTTR) . . . 133

Figure 22. Input Index 137
Figure 23. Failure Rate 140
Figure 24. Availability 144
Figure 25. Availability 145

List of Tables

Table 1. Software Engineering Objectives 10
Table 2. Software Engineering Principles 11
Table 3. Software Engineering Objectives and Corresponding Principles 13
Table 4. Software Attributes 14
Table 5. Software Engineering Principles and the Effects on Attributes 15
Table 6. Management Indicators: Relationship to Objectives 64
Table 7. Management Indicators: Type Classification 65
Table 8. Defect Days 101

1. Introduction

The rapid growth and increasing importance of software systems during the past ten years has brought about the need to increase the quality of software. Software Engineering, as a discipline, has evolved as the result of an effort to increase the quality of software. As software systems continue to grow larger and more complex, high quality software products become more difficult to obtain. The Software Engineering community has only begun to identify fundamental concepts and can claim only a partial understanding of the software development process, which would permit the complex software systems of today and tomorrow to meet quality requirements.

The Software Engineering discipline first recognized that the development of better quality software is accomplished by the application of beneficial *principles* to the development process [ARTJ86], e.g. stepwise refinement [WIRN71] and information hiding [LISB72]. These principles are associated with the development of a software product, the code and the supporting documentation. Today, numerous *software development methodologies*, guided by these principles, seek to aid the developer in producing better quality software. Software development methodologies provide a systematic approach to direct the developer through the development process.

Recognizing the impact of a methodology on the development process and the subsequent quality of the software product, Arthur, Nance, and Henry [ARTJ86] have proposed an evaluation proce-

dure for assessing the *adequacy* and the *effectiveness* of a methodology. The procedure relies on two sets of linkages:

1. those among the *principles* that guide the development process and desirable software engineering *objectives*, and

2. those among the same set of *principles* and the *attributes* of a software product.

The linkages among objectives, principles, and attributes provide the foundation for *the Procedural Approach to the Evaluation of Software Development Methodologies*. Because the procedure relies on these linkages, the approach is also known as the Objective/Principle/Attribute approach, or the OPA approach. These linkages are illustrated in Figure 1 on page 3 [ARTJ86,p.11]. The basic tenet underlying the OPA approach is that objectives reflect an assessment of the needs and requirements, and these objectives are met when certain principles are utilized in the development process. Employment of these principles induces specific attributes in the software product, i.e. the documentation and code.

The OPA approach to evaluating software development methodologies involves a *top-down* and a *bottom-up* evaluation process. The approach proposes the following assessment processes:

1. The ***top-down*** evaluation process provides the framework for assessing the *adequacy* of a methodology. It allows one to determine which development methodology is most appropriate relative to achieving a software product possessing desirable objectives.

2. The ***bottom-up*** evaluation process supports an assessment of the *effectiveness* of a methodology. It allows one to determine the extent to which the objectives are actually present, by examining the software product.

Quality control, a means for measuring and assessing the quality of code and documentation, needs to be introduced during the development of a software product. This provides early warning of

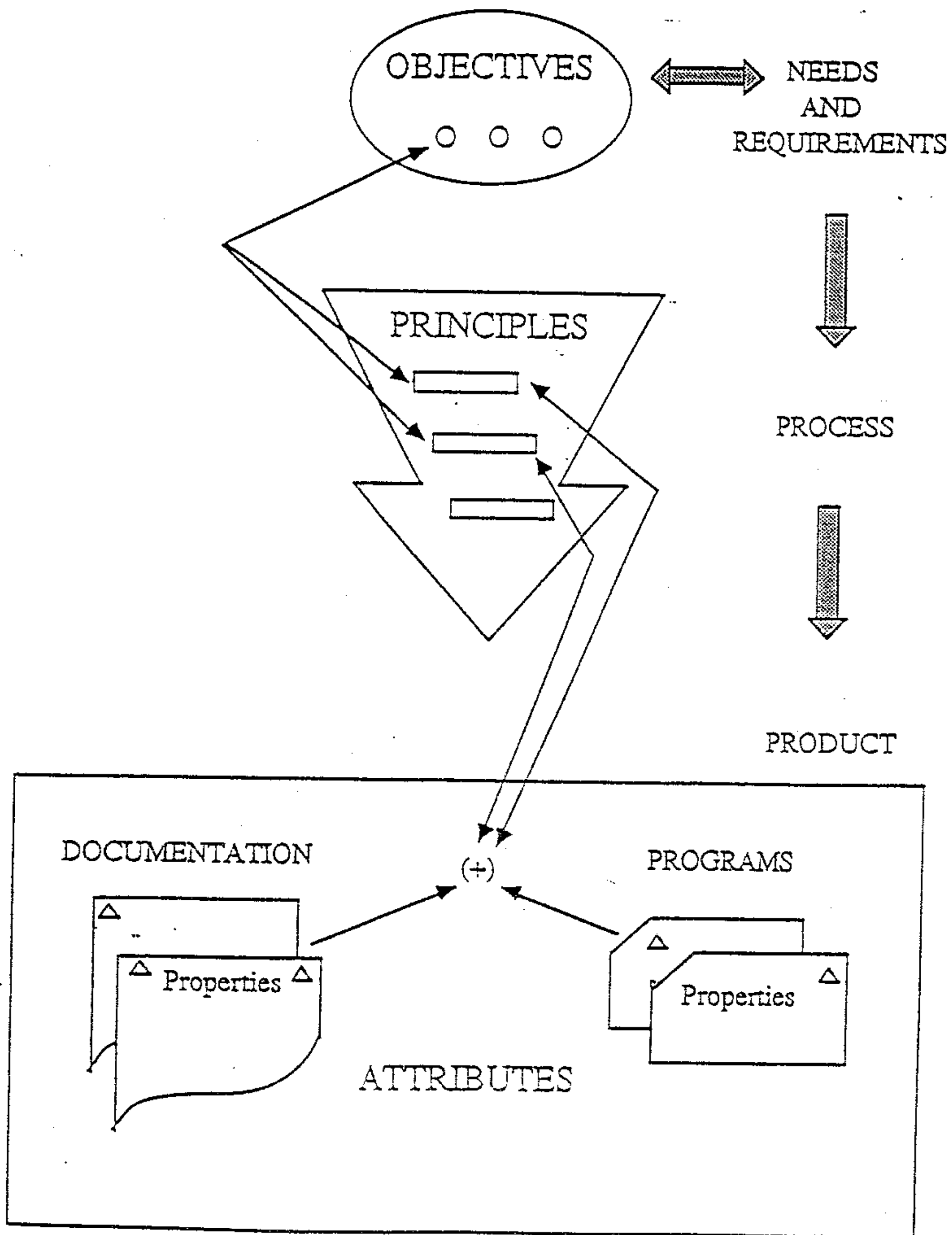

Figure 1. Objectives, Principles, and Attributes: Linkages

potential problems in the quality of the product. In some sense, software product quality can be envisioned as the extent to which a product meets its objectives. Effectively, the bottom-up evaluation process enables one to assess the extent to which objectives are present in a product. Arthur and Nance [ARTJ87a] reason that the evaluation procedure supports a basis for assessing product quality.

Software Quality Assurance as defined by Arthur, et. al. [ARTJ87b], necessitates a:

- *systematic* approach to
- *assessing* a product's (or process') conformance to
- *acceptance standards.*

According to Arthur, et. al. [ARTJ87b], and in concert with this definition, the evaluation procedure provides a basis for:

- *identifying* measureable software quality criteria,
- *defining* software quality acceptance levels based on those criteria, and
- *measuring* software quality utilizing a systematic, automated approach.

Effectively, as mentioned in [WHIR87], these capabilities provide a framework for the synthesis of a software quality assessment procedure.

1.1. Problem Statement

The bottom-up evaluation process defined by the OPA approach provides a basis for determining the extent to which software engineering objectives are present in a software product. If one bases quality assessment on the degree to which a software product achieves stated objectives, the bottom-up evaluation process provides a method for assessing quality relative to stipulated acceptance standards applied to the desired objectives. Moreover, for particular software engineering objectives, e.g. maintainability and reliability, the measurement of quality obtained through the use of the bottom-up evaluation process also provides a predictive indication of a product's post-development behavior. This observation motivates the research described in the report. In particular, the author addresses the following question: *How can the predictive capability of the bottom-up evaluation process be validated?*

Such validation, states Gilb [GILT77,p.65], is important in that "Any improvement in our ability to measure system quality is certainly an improvement in our ability to motivate people to design, construct and maintain systems of desired quality levels." Correspondingly, this validation should prove to be a large step in the measurement of software quality [KEAJ86].

1.2. Solution Approach

The values of the quality measures obtained from the bottom-up evaluation process are hypothesized to be a predictor of software product quality. In order to validate the predictive capability of the bottom-up evaluation process, the predicted values must be compared to values obtained from quality characteristics observed in the production version of the system. That is, the values

obtained from the bottom-up evaluation process must be reflective of values measuring corresponding characteristics of acquisition and execution behavior of a product.

The bottom-up evaluation process focuses on the *design characteristics* of the code and supporting documentation. The values obtained represent the extent to which the objectives are present in the code and documentation. These measurements are captured by utilizing the concept of *Design Indicators*. Design Indicators are discussed in greater detail in section 3.1.

How can the acquisitional and behavioral values be obtained for comparison to these predicted values? To obtain such values, a counterpart to Design Indicators is recognized, to measure the extent to which the objectives are present in the acquired product or during its operation in the intended environment. This view of the product focuses on acquisitional and deployment characteristics of the developed product rather than those related to a product's design structure.

The approach taken by this research is to view the product in terms familiar to a *manager* of the developed software product. The role a manager plays following software development is twofold:

1. acquiring a product that meets specified goals and objectives, and

2. confirming that the operational behavior of the product is consistent with the specified objectives.

Because the operational arena provides a natural setting for validation of the predictive framework defined by the OPA approach, this research effort entails an exploration of *Management Indicators* relative to the identification of acquisitional and behavioral product characteristics that can be used to confirm or refute the OPA predictive capability. Management Indicators are discussed in greater detail in Section 3.2.

As a basis for the validation effort, the software engineering objectives of reliability and maintainability have been selected. Reliability is chosen because of the extensive use and complex applica-

tions of software today. Software systems, increasing in size and responsibilities, are utilized in life-critical situations which dictate that the software must not fail. Maintainability is of primary concern because of the amount of time and the cost associated with the maintenance effort. It is estimated that over half of the effort invested in the life of a software product is spent on maintenance activities [BOEB84]. In concert with the above, this research effort focuses on identifying factors relating to reliability and maintainability that provide a structure for data collection, analysis, and validation of the OPA predictive capability.

In presenting the results of this research, Chapter 2 discusses the OPA approach to evaluating software methodologies and the current methods of assessment of software quality, and provides concepts and terminologies necessary to present Management Indicators. In Chapter 3, Design Indicators are reviewed and Management Indicators are defined (with examples of each given). The hypothesized relationship between these indicators is also discussed in greater detail. Chapter 4 concludes the report with a summary, the author's contribution to this research, and an outline of future work that is associated with the validation of the predictive capability of the bottom-up evaluation process.

2. Background Concepts and Terminologies

This chapter presents the background material necessary for discussing Management Indicators as a means of validating the predictive capability of the bottom-up evaluation process. The Procedural Approach to the Evaluation of Software Development Methodologies (the OPA approach), which defines the bottom-up evaluation process, is discussed in detail and current methods for assessing software quality are provided as motivation for the validation. Concepts and terminologies underlying Management Indicators conclude the chapter.

2.1. The Procedural Approach to the Evaluation of Software Development Methodologies

The Procedural Approach to the Evaluation of Software Development Methodologies (the OPA approach) is introduced in Chapter 1. As previously noted, this approach:

1. assists the developer in choosing an appropriate methodology to achieve the desired objectives in a software product, and

2. enables an assessment of the extent to which the desired objectives are achieved in a software product from a design perspective.

These assessments are accomplished via the linkages among objectives, principles, and attributes. These linkages are now discussed in detail.

2.1.1. Linkages

A software product is developed with an emphasis on particular software engineering ***objectives***, depending upon the type of application. Life-critical systems, for example, focus on reliability, while systems expected to have a long life-span focus on maintainability and adaptability. Seven commonly recognized software engineering objectives are listed in Table 1 on page 10 [ARTJ87a]. Expanded definitions of these objectives can be found in [IEEE85] and [ARTJ86, Appendix 1]. The requirements for meeting these objectives should be specified along with the functional requirements of a software system [ADRW82].

To achieve these objectives in the product, certain software engineering ***principles*** must be applied in the development process. A list of principles, which are beneficial if utilized in the software development process, is found in Table 2 on page 11 [ARTJ87a]. Expanded definitions of these principles can be found in [IEEE85] and [ARTJ86, Appendix 1]. These principles are associated with the development of a software product, the code and the supporting documentation.

Table 3 on page 13 [DANA87] specifies the principles which contribute to the achievement of the objectives in the product. These are the enumerated linkages between the principles and the objectives. For example, the principles of stepwise refinement, concurrent documentation, hierarchi-

Table 1. Software Engineering Objectives

Maintainability	the ease with which corrections can be made in responce to recognized inadequacies
Correctness	strict adherence to specified requirements
Reusability	the use of developed software in other applications
Testability	the ability to evaluate conformance with requirements
Reliability	the error-free use of software over time
Portability	the ease in transferring software to another environment
Adaptability	the ease with which software can accommodate to change

Table 2. Software Engineering Principles

Heirarchical Decomposition	components defined in a top-down manner
Functional Decomposition	components are partioned along functional boundaries
Information Hiding	insulating the internal details of component behavior
Stepwise Refinement	incorporation of progressively finer component details in successive steps
Structured Programming	using a restricted set of control constructs
Concurrent Documentation	management of supporting documents (system specifications, user manuals, ets.) throughout the life cycle
Life-cycle Verification	verification of requirements throughout the design, development, and maintenance phases of the life cycle

cal decomposition, functional decomposition, information hiding, and structured programming contribute to the achievement of the objective of maintainability. Thus, in order to obtain a software product possessing certain software engineering objectives, a software development methodology which encourages the use of the principles necessary for achieving these objectives must be applied during the development process.

Attributes are characteristics of a software product that are often difficult to measure from a purely objective perspective. Table 4 on page 14 [ARTJ87a] lists software attributes. Applying a software engineering principle to the development process has a predictable effect on certain attributes of the product. The effects of principles on the attributes are specified in Table 5 on page 15 [DANA87] through the enumerated linkages between the principles and the attributes. For example, the principle of stepwise refinement decreases coupling, enhances cohesion, and reduces complexity.

By employing a software development methodology utilizing principles which support the objectives desired in the software product, the product should evince the desired attributes. For example, utilizing a methodology which promotes maintainability by encouraging the use of functional decomposition, should reduce coupling among product modules.

In order to determine if the proposed objectives are being met in a software product, the attributes of the product must be measured. Attributes are identified by properties (also called factors), which are the observable, measurable surface qualities of the product. A measurement approach is necessary to assess the extent to which the attributes are present. For example, the readability of a program is affected by the number of GOTOs in a program, the consistency of explanatory comments, and a number of other properties. Figure 2 on page 17 [ARTJ88] lists the properties which affect the readability of the product and illustrates how readability, as well as other attributes, relates to the objectives through the linkages with principles. As can be seen in Figure 2, the attribute of readability is linked to the principles of structured programming and concurrent documentation. These principles, in turn, are linked to certain objectives. The principle of structured programming contributes to achieving the objectives of maintainability, correctness, testability, reliability, and

Table 3. Software Engineering Objectives and Corresponding Principles

Objective	Principles
Maintainability	Stepwise Refinement Concurrent Documentation Hierarchical Decomposition Functional Decomposition Information Hiding Structured Programming
Correctness	Hierarchical Decomposition Life-cycle Verification Stepwise Refinement Structured Programming
Reusability	Concurrent Documentation Hierarchical Decomposition Functional Decomposition Information Hiding
Testability	Life-cycle Verification Hierararchical Decomposition Functional Decomposition Information Hiding Stepwise Refinement Structured Programming
Reliability	Hierarchical Decomposition Information Hiding Stepwise Refinement Structured Programming
Portability	Functional Decomposition Concurrent Documentation
Adaptability	Stepwise Refinement Concurrent Documentation Hierarchical Decomposition Functional Decomposition Information Hiding Structured Programming

Table 4. **Software Attributes**

Cohesion	the binding of statements within a software component
Coupling	the interdependence among software components
Complexity	an abstract measure of work associated with a software component
Well-defined Interface	the definitional clarity and completeness of a shared boundary between a pair of software components
Readability	the difficulty in understanding a software component (related to complexity)
Ease of Change	the ease with which software accommodates enhancements or extensions
Traceability	the ease in retracting the complete history of a software component from its current status to its design inception
Visibility of Behavior	the provision of a review process for error checking
Early Error Detection	indication of faults in requirement's specification and design prior to implementation

Table 5. Software Engineering Principles and the Effects on Attributes

Principle	Effect on Attribute
Hierarchical Decomposition	Ease of change Coupling/cohesion enhanced Reduced complexity
Functional Decomposition	Ease of change Coupling/cohesion enhamced Reduced complexity
Information Hiding	Ease of change Coupling/cohesion enhanced Reduced Complexity Well-defined interfaces User-defined data types
Stepwise Refinement	Coupling/cohesion enhanced Reduced complexity
Structured Programming	Reduced complexity Readable code
Concurrent Documentation	Readable code Traceability Ease of change Reduced complexity
Life-cycle Verification	Visibility of behavior Early error detection

adaptability. The principle of concurrent documentation contributes to achieving the objectives of maintainability, reusability, portability, and adaptability. Thus, in determining if any of these objectives are met, the properties affecting readability must be examined.

As discussed by Arthur, et. al. [ARTJ87a], however, manual collection of these visual properties can introduce bias into the measurement of attributes. To free the measurement process of this subjectiveness, Design Indicators are employed to assess compliance with objectives through the accumulation of statistical evidence based on a static analysis of the software product. Design Indicators are discussed in greater detail in section 3.1.

To summarize, the OPA approach exploits the premise that software development methodologies provide a systematic approach to guide the developer through the development process. The direction provided by a methodology encourages the realization of certain objectives in the software product through the utilization of beneficial principles. Employment of these principles induces certain attributes in the product. The next section discusses how these linkages are used in the evaluation processes.

2.1.2. Assessment Capabilities

The linkages among objectives, principles, and attributes support a *top-down* and a *bottom-up* evaluation process. These processes provide for the assessment of software development methodologies.

The *top-down* evaluation process allows one to assess the *adequacy* of a methodology in the development of a software product with specific objectives in mind. In order to employ the top-down process, one must first examine the objective/principle linkages to see if the principles which contribute to the achievement of the desired objectives are encouraged by the methodology. The methodology should encourage the principles which contribute to the achievement of the stated

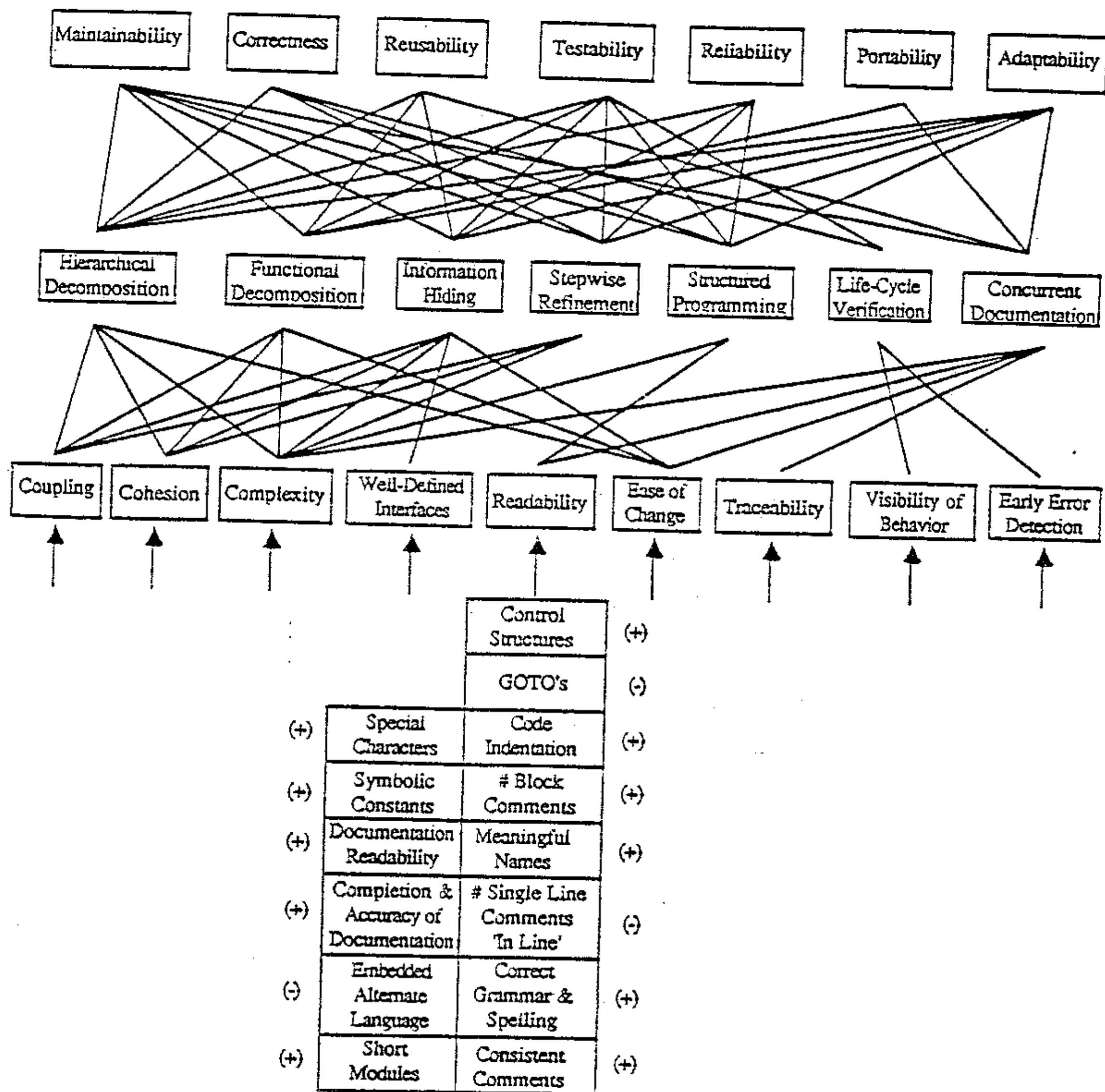

Figure 2. **Linkages:** The Readability Attribute

objectives. If the necessary principles are utilized, then the evaluation process can continue. By examining the causative principles, resulting attributes can be identified and compared to those attributes desired in the product. If the two sets match, then the methodology is adequate for use in the development of a software product emphasizing the specified objectives.

The ***bottom-up*** evaluation process allows one to assess the *effectiveness* of a methodology by examining a given software product. The effectiveness of a methodology is judged by how well the product reflects the espoused objectives. First, the attributes in the product are measured through the use of Design Indicators. The principles necessary to induce the measured attributes are determined by examining the principle/attribute linkages. The extent to which the objectives are present in the product is then determined by examining the principle/objective linkages. Propagating measures of the attributes through the linkages, up to the objectives, provides an indication of the extent each objective is realized in the given software product. To determine the effectiveness of the methodology, it must then be judged whether these objectives are the ones espoused by the methodology.

2.2. Current Evaluation of Software Quality

As mentioned in Chapter 1, the bottom-up evaluation process defined by the OPA evaluation procedure provides a framework for the synthesis of a software quality assessment procedure. That is, the approach provides a foundation for an assessment process that supports software quality assurance. The purpose of software quality assurance, as stated in [WHIR87], is to provide:

- a discipline [EVAM87] to systematically:
    - develop and monitor adherence to standards,

- evaluate processes and products, and
- perform acceptance tests [FAIR85].

Three main problems presently exist that limit the effectiveness of the current approaches to evaluating software quality [WHIR87]:

1. no single approach currently exists which adequately addresses both process and product concerns,
2. no approach covers the entire software life cycle, and
3. research in the development of evaluative approaches is fractionated and poorly supported.

In reference to the first point, a software quality assessment technique should be concerned with addressing both process and product concerns. One ***Process/Product Taxonomy*** [WHIR87] for assessing the quality of a software product consists of measuring characteristics of:

- the software development *process* to ensure that quality is built into the product, and
- the software *product* to assess the degree of quality actually possessed by the product.

Although it is noted that a software quality assessment technique should address both process and product concerns, no technique currently exists which adequately addresses both concerns.

In reference to the second point, software quality must be a continuing goal, because any slight deviation throughout the entire life cycle can cause a serious problem in the quality of the software product. For example, the quality cannot be judged simply by testing the product once the process is finished. Testing is often inadequate, and cannot be used as the only method to ensure the quality of the product.

Although increased importance has been placed on the entire life cycle, an integrated package for the continuous assessment of software quality throughout the life cycle phases does not exist. Currently, several tools must be acquired and many techniques learned in order to assess software quality over multiple phases of the software life cycle [ADRW82]. To remedy this situation, assessment techniques must be integrated in a mutually complementary, unified manner. Existing tools and techniques usually reflect myopic development concerns, addressing only one phase of the software life cycle.

In reference to the third point, numerous methodologies exist, and new techniques are being intensely researched, with the purpose of improving software quality. Nonetheless, no centrally focused, concerted research effort is apparent. The methods being developed capture only a piece of what is expressed in the term "software quality assurance," and often ignore any global objectives. This can be seen in the fact that there is no standard terminology for software quality assessment. For example, the terms "verification" and "validation" are frequently used throughout the literature as if they have the same meaning, when they are actually quite different.

In reviewing the current techniques and tools available for use in evaluating software quality, techniques are readily found that concentrate on product concerns, whereas few focus on process concerns [ARTJ87b]. Methods which focus on product concerns [ARTJ87a] include such automated software tools as Henry's Information Flow Analyzer [HENS87], DAVE [OSTL76], and SOFTDOC [SNEH85]. As previously mentioned, no current technique adequately addresses both process and product concerns. The OPA approach attempts to overcome this deficiency. While product concerns are addressed by the measurement of the extent to which objectives are attained in a software product, a step is taken in the right direction by recognizing the impact that methodologies have on the software development process.

2.3. Concepts and Terminologies

This section provides a number of discussions and working definitions fundamental to understanding the underlying concepts of Management Indicators.

2.3.1. Reliability

Management Indicators are precise, concise, and strongly indicative of the presence or absence of one or more objectives in a software product. Unlike the indicators described in [ARTJ87a], this research uses the term "indicator" loosely because the current investigation of Management Indicators is still in its preliminary stage. Reliability is one of the objectives chosen to begin the identification of factors which can be used in the measurement of the achievement of quality. As previously defined, ***reliability*** is the error-free behavior of software over time.

Error-free behavior means that the results of execution agree with specifications of the software product behavior. Although most situations are covered by the specifications, chances are that some situations may be omitted. Even if omitted, the users expect the software to perform in a reasonable manner [MYEG76]. For example, the user may enter an invalid input into the system which is omitted in the specifications. The system is expected to perform reasonably, such as returning an error message; an unexplainable failure is unacceptable. In order to be considered reliable, the execution behavior should match the users' expectations of the product if anomalous conditions are omitted in the specifications.

While error-free behavior is necessary for achieving reliability of a software product, errors which occur during execution have differing impacts on users. As a result, errors must be categorized in terms of their impact. Certain errors are more serious than others. For example, errors which cause

the system to fail are more serious than a misspelled word in the output. Thus, reliability must also be weighted by the cost to the users of each unexpected occurrence.

Realization of the factors which contribute to the reliability of a software product allows methods to be identified that can be used to measure reliability. The number and type of defects detected in the software product represent quantitative measures of reliability [MACJ86]. Motivation for these types of measurements lies in the fact that the probability of undetected defects extant in the product increases as the number of defects detected increases [MYEG76].

2.3.2. Maintainability

Maintainability is the second objective chosen to begin the Management Indicator identification process. As previously defined, ***maintainability*** is the ease with which corrections can be made in response to recognized inadequacies.

Maintainability is associated with the time and cost of revising a software product. These include corrections to recognized inadequacies in:

- *software* - the software product, i.e. the code or the documentation, contains defects which need to be removed or corrected,
- *system requirements* - features provided by the software product may need to be added or deleted,
- *hardware configuration* - the software product may need to be modified in order to be compatible with hardware changes, and
- *performance* - the software product may need to be modified to improve performance.

Thus, maintenance is performed in response to other inadequacies of the software product in addition to the correction of defects in the software.

Maintenance activities are classified into three types [SWAE76]:

- *Corrective Maintenance* - performed in response to defects in the software product,

- *Adaptive Maintenance* - performed in response to change in data or processing environments (this may be interpreted as a reflection of portability), and

- *Perfective Maintenance* - performed to eliminate processing inefficiencies, enhance performance, or improve maintainability of the product.

As mentioned, the time taken to perform the revisions to the software product is a crucial aspect of the maintainability of the product. Users expect a maintenance activity to be completed within a satisfactory time frame. Nonetheless, a software product should not be "fixed" haphazardly in order to expedite the maintenance activity. Such a "fix" may sacrifice the future reliability and/or maintainability of the product and can be more detrimental than missing a deadline.

Realization of the factors which contribute to the maintainability of a software product allows methods to be identified that can be used to measure maintainability. The amount of time necessary to detect and correct defects and the number of outstanding defects represent quantitative measures of maintainability of a software product.

2.3.3. Software Life Cycle

The ***life cycle of a software product*** typically consists of six phases. These typical phases are [BOEB84, GILP83]:

- *Requirements Analysis* - the understanding of the users' problem to be solved, i.e. what are the requirements of the software system,

- *Specification* - the formulation of complete, consistent, and precise specifications of the requirements, i.e. what the software product is to do,

- *Design* - the planning of the software product in accordance with the specifications, i.e. how the software product will perform its requirements,

- *Implementation* - the process of coding from the design,

- *Testing* - the validation of the performance of the software product according to specifications, and

- *Operations and Maintenance* - the execution of the software and modification of the software product.

These phases are not quite as fixed as they may appear from the discussion above. Phases can overlap, e.g. aspects of the requirements analysis and specification phases may be interleaved. A previous phase in the software life cycle may have to be revisited, e.g. after maintenance has been performed on a product, the product must again be tested to ensure, not only that the revisions perform correctly, but that features which were correct were not influenced as a result of the maintenance activity.

The phases are often subdivided into smaller components. For example, design usually consists of a preliminary design and a detailed design, and testing involves testing of the individual components (*unit testing*), testing of the integrated product (*system testing*), and testing of the operational capabilities of the software (*acceptance testing*). These subphases are sometimes considered and referred to as complete phases themselves.

The terms *acquisition* and *deployment* are usually associated with the software life cycle. ***Acquisition*** of a software product occurs after the testing phase is completed. At this point the product is contractually acquired; the sponsoring agent receives the product from the contractor. The phases up to the point of product acquisition are generally considered the development phases. Once acquired, the software is placed in the operational environment (although testing could continue). The Operations and Maintenance Phase is commonly known as ***deployment***. Management Indicators, as are discussed in section 3.2, are determined during *acquisition* and *deployment* of a software product.

2.3.4. Faults, Failures, Defects, and Errors

Numerous definitions exist for the terms fault, failure, defect, and error (such as those in [IEEE83]). These terms need to be precisely defined in order to discuss how they relate to the reliability and maintainability of a software product. This section attempts to provide useful definitions which confine the variance in meanings emanating from broad usage of the terms. These definitions distinguish how the terms are used throughout the remainder of this report.

Fault:

> The inability of the software to exhibit characteristics in accordance with requirements or users' expectations.

That is, a *fault* occurs when the exhibited characteristics of the software do not meet the system requirements or do not meet reasonable expectations of the users.

Failure:

> The abnormal termination of the functioning of the program.

That is, a *failure* occurs when the execution terminates abnormally due to the program. A *failure* is one type of *fault*.

Defect:

> The condition in the software product, either in the code or the documentation, that causes a *fault* to occur.

That is, a *defect* is a section of inadequately specified, designed, or implemented code or documentation. A *defect*, if encountered during execution, may cause a *fault*. A *defect* in the software product is synonymous with the term *bug*.

Error:

> Human action that results in the software product containing a *defect*.

That is, an *error* is made by a human producing the code or documentation in which there exists a *defect*. All software *defects* are a result of human *errors*.

2.3.5. Review and Inspection Processes

Reviews performed on a software product have the objective of detecting defects or omissions in the product as early as possible. Thus, these methods have a great effect on the reliability of the product. Review processes should be completed after each phase in the software life cycle, before advancing to the next phase, because in each phase there is a possibility of introducing defects in the software product. For example, a *requirements analysis review* is implemented at the end of the requirements analysis phase to detect defects introduced during that first phase of the software life cycle. A *specifications review* is implemented at the end of the specifications phase to detect defects

introduced during the specifications phase or those which may have carried over from the requirements analysis phase.

The reviews associated with the design and code phases are usually termed ***inspections***. Inspections are applied after the design phase (*design inspection*), preliminary and detailed, and after the implementation phase (*code inspection*). A good design results in the finding of fewer defects during the inspection procedures, if these procedures are implemented properly.

Reviews are utilized to detect defects in a software product as early as possible. There are two reasons why defects should be detected as early as possible. These are illustrated in Figure 3 on page 28 [MYEG76,p.44]:

- the cost of fixing a defect increases with time (Figure A), and
- the probability of fixing a defect correctly decreases with time (Figure B).

Detecting a defect in a software product does not imply that it is best to fix that defect. Sometimes a defect should be left in a product because the consequences associated with fixing the defect pose a greater risk than leaving the defect in the product [MYEG76]. In order to determine if a detected defect should be fixed, the probability of not being able to correct the defect, the probability of introducing a new defect into the product, and the severity of the defect should be weighed. For example, if the probability of not being able to correct the defect is high and the probability of introducing new defects into the system is high, but the defect is of low severity, it may be less detrimental if the defect is left in the product. A defect detected during a review or inspection that is determined to warrant fixing should be corrected before advancing to the next phase of the software life cycle.

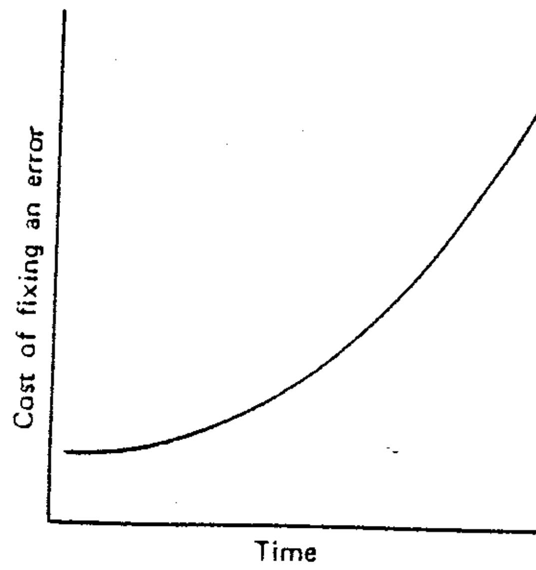

FIGURE A.

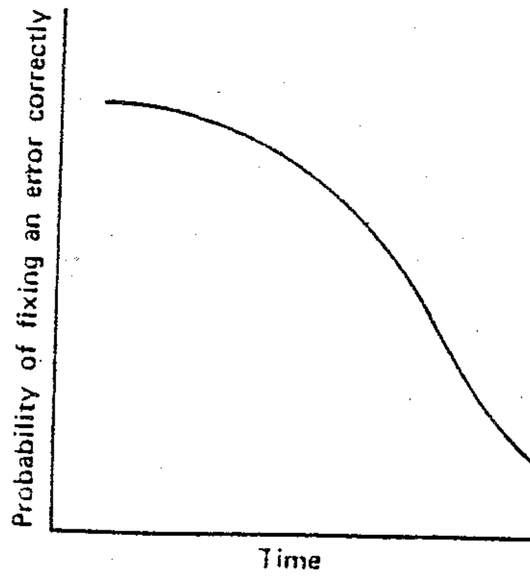

FIGURE B.

Figure 3. Motivation for Early Detection of Defects

2.3.6. Software Trouble Reports

Software Trouble Reports (STRs) are frequently used to keep track of defects found in the software product. *Software trouble* includes defects found in any phase of the software life cycle. Defects found during requirements and specifications reviews can often be corrected quickly. Thus, while it is helpful to know the number of defects found during these reviews, defects discovered in these phases are seldom written up in STRs. STRs are usually written and collected for defects detected during inspection processes, test phases, and during operation of the system.

Before an STR is compiled, a system fault must be investigated. When a fault occurs, it is first categorized as software, hardware, system software, operator, or unknown (something else). The proper category is sometimes difficult to determine. In considering STRs, information is collected on faults which have been determined as being caused by the software product in question. If a fault is determined as having been caused as a result of one or more defects in the software product, an STR should be compiled. An STR is not necessarily the result of one defect in the software product; it may result from a combination of defects, or a defect may contribute to more than one instance of software trouble.

In order to keep track of the defect data, an error tracking system or trouble reporting system should be automated. If defect reporting is automated, the data on each defect can be kept in a consistent format within a database. This information can be collected into a database for easier use at later points in time. This would allow for the tracking of specific defects and recognition of defect trends [GLAR79]. The information should include any data related to the reliability and maintainability of the software product. Information such as the following should be collected on each defect:

- the type of defect,

- the cause of the defect,

- the software unit(s) in which the defect was found,
- the phase and approximate date the defect was introduced,
- the phase and date the defect was detected,
- the phase and date the defect was removed, and
- the severity of the defect.

The severity of the defect is determined by the impact of the defect on the users. As previously mentioned, all defects do not result in the same consequences to the users. There is a great difference in the effect a system failure has on the users, as opposed to a misspelled word of output. Defects can be rated on different scales, as "high", "medium", or "low", or they can be classified on a point scale. Defects causing failures are generally of the highest severity, those having medium severity usually affect the performance of the operation but not to the extent of a failure, and defects classified as having low severity have little effect on the user, such as a misspelled word in the output. This discussion of severity levels is presented because these levels can be useful in the determination of discriminating categories for providing a more accurate picture of reliability and maintainability. The intent in this report, however, is not to address the use of these levels in particular, but only to suggest where they can be used.

3. A Comparison of Design and Management Indicators Relative to Software Quality Assessment

The concept of using *indicators* to measure software quality is presented by Arthur and Nance [ARTJ87a]. Indicators have been used with success in other disciplines, such as in the social sciences, to indirectly measure seemingly unmeasurable, qualitative concepts.

This chapter discusses two different types of indicators:

1. those which focus on the design characteristics of the software product, termed *Design Indicators*, and

2. those which focus on the acquisitional and behavioral characteristics of the developed software product, termed *Management Indicators*.

Design Indicators are utilized by the bottom-up evaluation process defined by the OPA approach to measure how well a software product conforms to the software engineering objectives espoused by the software development methodology utilized in the development process. If the extent to which objectives are present in a product can be obtained from the bottom-up evaluation process,

then the quality of a product can be assessed relative to stated acceptance standards for the desired objectives. Moreover, the extent to which the objectives are present in a product, as measured by the bottom-up evaluation process, is observed to be a *predictor* of the behavior of the software after development is completed. This report suggests Management Indicators as a means to validate this predictive capability by measuring the extent to which the objectives *are* realized by focusing on the acquisitional and behavioral characteristics of the developed product.

This chapter examines Design Indicators and their role in the bottom-up evaluation process and in the determination of the extent to which objectives are present in a software product. The Management Indicator concept is investigated to determine how objectives may be measured by exploiting post-development characteristics. The chapter concludes with a discussion of the relationship between the two types of indicators.

3.1. Design Indicators

Design Indicators are suggested for measuring the extent to which objectives are realized in a software product from a design perspective [ARTJ87a]. These indicators reduce the bias that is introduced by the evaluator via the metrics based on strictly observable properties. Design Indicators are employed by the bottom-up evaluation process to indirectly measure the presence of attributes in a software product.

The following definition is provided [ARTJ87a]:

> A ***Design Indicator (DI)*** is a variable whose value can be determined through direct analysis of product characteristics and whose evidential relationship to one or more attributes is undeniable.

Consequently, a DI *must* be

- measurable through analysis of programs and documentation, and
- indicative of the presence or absence of one or more attributes.

Further, a DI *can* be

- "raw" statistics extracted from code and documentation analysis, or
- variables computed from "raw" statistics.

Finally, a DI *should* be

- simple, understandable, easily related to attribute(s),
- targeted at design information (documentation) and at implementation (code and documentation), and
- as objective as possible.

Crucial to this definition are the aspects that [WHIR87]:

1. the value of a DI is *directly* measurable, and
2. a DI is *always* an *attribute/value* pair.

The bottom-up evaluation process defined by the OPA approach is initiated by utilizing the DIs to measure the attributes of a software product. By propagating the results up through the defined attribute/principle and principle/objective linkages, it can be determined if and to what extent the objectives are present in a software product. To implement this process, the measurement of the

attributes, principles, and objectives must be based on a common scale [DANA87]. The common scale allows comparison within and propagation among the attributes, principles, and objectives in a product. Previous research [DANA87] on the bottom-up evaluation process suggests the use of a 1 to 10 range as an appropriate measurement scale. On this scale, 1 is considered poor, 5 is a mid-range score, and 10 is the most desired.

The following example of a DI is provided [ARTJ87c, ARTJ88, WHIR87]:

Coupling relative to the use of Structured Data Types

if (# of calls > 0) and (# of parms > 0) *then*

CP/SDT := 5 - ((# SDT parms) / (# total parms)) * 4

else

CP/SDT := 5

In this measurement of the attribute of coupling relative to the use of structured data types, the value of the associated DI, CP/SDT, can be determined by direct analysis of the product characteristic of module parameters within the code. Structured data types have a negative effect on coupling, as a result of passing more information than necessary. A mid-range score of 5 is obtained if there are no module calls or no parameters utilized by the modules. Otherwise, a "poor" value between 1 and 5 is obtained, depending upon the percentage of parameters which constitute structured data types. Coupling is linked through the software engineering principles to all seven objectives, namely maintainability, correctness, reusability, testability, reliability, portability, and adaptability. Thus, this measurement is propagated through the linkages and used in the prediction of the extent to which each objective is realized.

This example illustrates a DI which is determined by analysis of the code. DIs also exist which are determined by analyzing the supporting documentation [STEK88].

3.2. Management Indicators

While DIs reflect design structure characteristics of a static software product to determine the presence of objectives, Management Indicators are suggested as a means of measuring the extent to which objectives are realized in a developed software product by considering acquisitional and behavioral characteristics of that product. If the extent to which objectives are present in a product during product acquisition and deployment can be determined by the use of Management Indicators, then the quality of a product can be assessed relative to stated acceptance standards of the desired objectives for these periods.

Management Indicators are used to indirectly measure the extent to which objectives are realized in a developed product. Management Indicators enable one to determine from a management perspective:

1. the quality of the product by measuring the extent to which the software is meeting desired objectives based on data available at acquisition, and

2. the quality of the product by measuring the extent to which the software is meeting desired objectives by considering behavioral and maintenance activity characteristics.

The first appraisal is the responsibility of the *acquisition manager.* It is the duty of the acquisition manager to measure the achievement of the desired objectives at product acquisition time. Product acquisition occurs after the testing phase is completed at the time the software product is acquired by the sponsoring agent from the contractor. At this time, data should be available to the acqui-

sition manager which has been collected during the development process (such as defects found during code synthesis and system testing), i.e. the acquisition manager relies on data supplied by the developer. This data can be analyzed to indicate the extent to which specific objectives are realized in the product. Correspondingly, product quality can be assessed at this time by comparing the results obtained by the use of Management Indicators to stated acceptance standards of the acquisitional product.

The second appraisal is the responsibility of the *operations manager*. It is the duty of the operations manager to measure the achievement of the desired objectives during operation of the software, and to ensure that maintenance does not degrade the achievement of the objectives. The behavior of the software observed during operation, in its intended environment, provides an indication of the actual quality of the software product. Data can be collected during execution and maintenance of the product which can be analyzed to indicate the extent to which specific objectives are realized. Again, product quality can be assessed at this time by comparing the results obtained by the use of Management Indicators to stated acceptance standards of the behavioral product.

The following definition of a Management Indicator has been adopted:

A ***Management Indicator (MI)*** is a variable whose value can be determined during

- Product *Acquisition* through direct analysis of results of software development reviews, inspections, and testing procedures, or

- *Execution* and *Maintenance* through direct analysis of software execution behavior or maintenance activity characteristics,

and whose evidential relationship to one or more software engineering objectives is undeniable.

Consequently, an MI *must* be

- measurable through analysis of the product acquisition, execution behavior, and maintenance activity characteristics, and

- indicative of the presence or absence of one or more objectives.

Similar to DIs, an MI *can* be

- "raw" statistics extracted from Acquisition/Execution/Maintenance analyses, or

- variables computed from "raw" statistics.

Finally, an MI *should* be

- oriented towards management,

- easy to

    - understand,

    - obtain,

    - apply, and

    - interpret,

- easily related to objective(s),

- targeted at Acquisition/Execution/Maintenance characteristics, and

- as objective as possible.

Crucial to this definition are the stipulations that:

1. the value of an MI is *directly* measurable, and

2. an MI is *always* an *objective/value* pair.

MIs should guide the activities of the acquisitional manager and the operations manager. That is, the manager should be concerned with the objectives, e.g. reliability and maintainability, of the software product that are measured by the MIs, and use these measurements in directing his future efforts. In order to measure and control the quality of the product, the manager must obtain the necessary data, apply the calculations dictated by the MIs, and interpret the results of the MIs.

As recognized, MIs are used during two different periods: during product acquisition and during deployment. During product acquisition, the product is assessed by reviewing the results of data obtained from the development process. During deployment, the product is assessed by examining data obtained from operational performance or maintenance activities. Figure 4 on page 39 illustrates these two types of MIs: Acquisitional and Behavioral.

3.2.1. Acquisitional Indicators

The following working definition is provided:

> An *Acquisitional MI* is determined at Product Acquisition time and is used in assessing whether the product achieves the desired objectives.

The acquisition manager should have access to data from the development process. This includes the results of product reviews, design and code inspection processes, testing processes, and the defects detected in the product at these times. The acquisition of this data from the contractor should be placed in the contract. A review of this data via Acquisitional Indicators provides a means of assessing a software product relative to desired objectives.

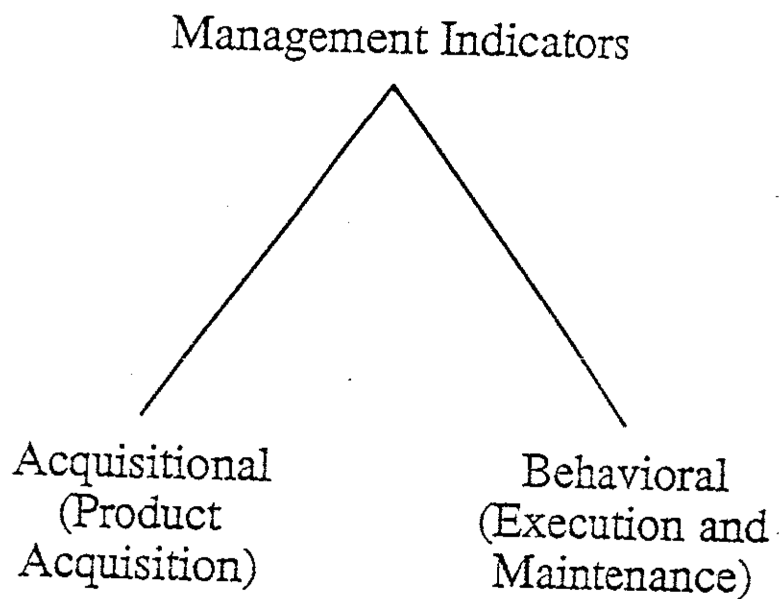

Figure 4. Management Indicators: Acquisitional vs. Behavioral

Many methods have been developed to predict the reliability of a software product by utilizing the acquisitional data. These methods usually involve statistics too complex for acquisition managers to use effectively. Other methods, which appear less complicated (such as error seeding), are impractical. Error seeding involves randomly inserting known defects in the software product. (Is anyone really going to do this?) The motivation behind the procedure is that seeded defects and inherent defects are equally likely to be detected. (Is this realistic?) The product is then tested. The number of inherent defects remaining in the product can be determined by the number of seeded defects remaining in the product. Unlike the above methods and as stated earlier, the author's intent in exploring MIs is to identify those indicators that are easy to understand, obtain, and interpret.

The following is an example of an Acquisitional MI:

Person-Hours per Major Defect Detected

The Person-Hours per Major Defect Detected Indicator [IEEE85] uses data gathered during design and code inspections which is available during product acquisition to determine if the inspection processes were effective in detecting defects in a software product. If used effectively these inspection processes provide a valuable method for the early detection of defects. By reviewing these acquisitional characteristics, the reliability and/or maintainability of the product may be evaluated.

The Person-Hours per Major Defect Detected Indicator is implemented by determining the amount of time that was necessary to detect defects during the design and code inspections of the development process. The value of the indicator is calculated as follows:

$$\text{Person-Hours per Major Defect Detected} = \frac{\text{total person-hours on design or code inspection}}{\text{number of major (non-trivial) defects detected}}$$

The result is the average length of time needed to detect a defect in the product during these inspections. The total person-hours on design or code inspection includes the time spent in the meetings for each person, as well as the preparation time for the meetings. Major defects should include all defects except those whose severity is determined as "low."

Empirical evidence suggests that values less than 3 or greater than 5 may indicate a problem in the reliability and/or maintainability of the software product. For example, excessive time to locate a defect in the design or code may imply that the inspection processes are poorly implemented. If the processes are poor, many defects which should have been detected during inspection may still remain in the software product, causing the reliability to be suspect. A large value may also suggest that maintainability is suspect. Excessive time to locate a defect may indicate the difficulty in understanding the design or code, resulting in a software product which is difficult to maintain. A small value may indicate a reliability problem. If the value is inordinately small, many defects are being detected in the software product at a high rate. Detecting a large number of defects rapidly can indicate the product is dense with defects, and many may still remain to be detected. Additional information may be gained by looking at the numerical trend provided by the computation of this indicator based on individual inspections.

The Person-Hours per Major Defect Detected Indicator is determined during acquisition of the product when the results of the inspection processes of the product during development should be readily available. While Acquisitional Indicators focus on data obtained from the development process, Behavioral Indicators focus on execution and maintenance characteristics.

3.2.2. Behavioral Indicators

The following working definition is provided:

> A *Behavioral MI* is determined during Execution and Maintenance and is used in assessing whether the operational product exhibits the desired objectives.

The operations manager should have access to data from the execution of the programs and from maintenance activities. This data includes information such as the defects detected in the product during operation and how long it takes to correct each defect. Reviewing this data through the use of Behavioral Indicators provides a means of assessing a product's conformance to desired objectives.

The following is an example of a Behavioral MI:

Mean Time to Failure (MTTF)

The Mean Time to Failure (MTTF) Indicator [KOPH79, ROSL83] uses data gathered on the number of failures which have occurred as a result of program defects during execution to determine the average length of time the program executes after execution begins until a failure occurs. Program failures have a tremendous impact on the reliability of the software product. By reviewing these behavioral characteristics the reliability of the product may be evaluated.

The state of a software product during the Operations and Maintenance phase follows a continuous cycle. Typically, the execution is initiated, then the program operates for a period of time until the execution terminates abnormally. This termination is sometimes the result of encountering a serious defect in the program. The system is then inoperable for a period of time, until it can be restarted. If the termination is caused by a program failure, a serious defect usually requires immediate correction. The program is usually restarted after the defect which

caused the failure has been repaired. The cycle then continues (i.e. after a period of time, the system will again fail and once again be restarted). The MTTF Indicator is determined by the average length of time the program executes until a program failure occurs. The value of the indicator is computed as follows:

$$\text{Mean Time to Failure (MTTF)} = \frac{\text{total execution time}}{\text{total number of failures}}$$

This indicator is calculated on a periodic basis (e.g. every month). A line chart of the values is plotted against time. See Figure 5 on page 44 for an example. Time is plotted along the horizontal axis. The MTTF value is plotted along the vertical axis. During the first month, 150 failures occurred which were the result of defects in the program. The system executed a total of 300 hours. These values result in a MTTF of 0.5 hours. Plotting continues in this manner throughout deployment.

A decreasing or relatively flat plot indicates a reliability problem may exist. The plotted line is expected to increase, as in Figure 5, indicating that a longer length of time is passing between failures for each successive period. Thus, defects which are serious enough to cause failures are presumably being reduced in the software product.

Relative to DIs, Behavioral MIs are of two types: *Non-affective* and *affective*. The recognition of these two types of Behavioral MIs is prompted by the realization that as software maintenance is performed on a product its design structure may change, which in turn may cause the values associated with the DIs to change. For example, parameters may be added to modules which may cause coupling among the modules to be increased or diminished. If this does occur, the values of the DIs which measure the presence of coupling in the product will be affected. Since coupling is linked through principles to all seven objectives, the extent to which each objective is present is also influenced. MIs which deal directly with modification are termed *affective* because modification may affect the values of DIs. DIs are used in the bottom-up evaluation process which is hypothesized to predict the quality of a software product relative to objectives. A change in the value of an

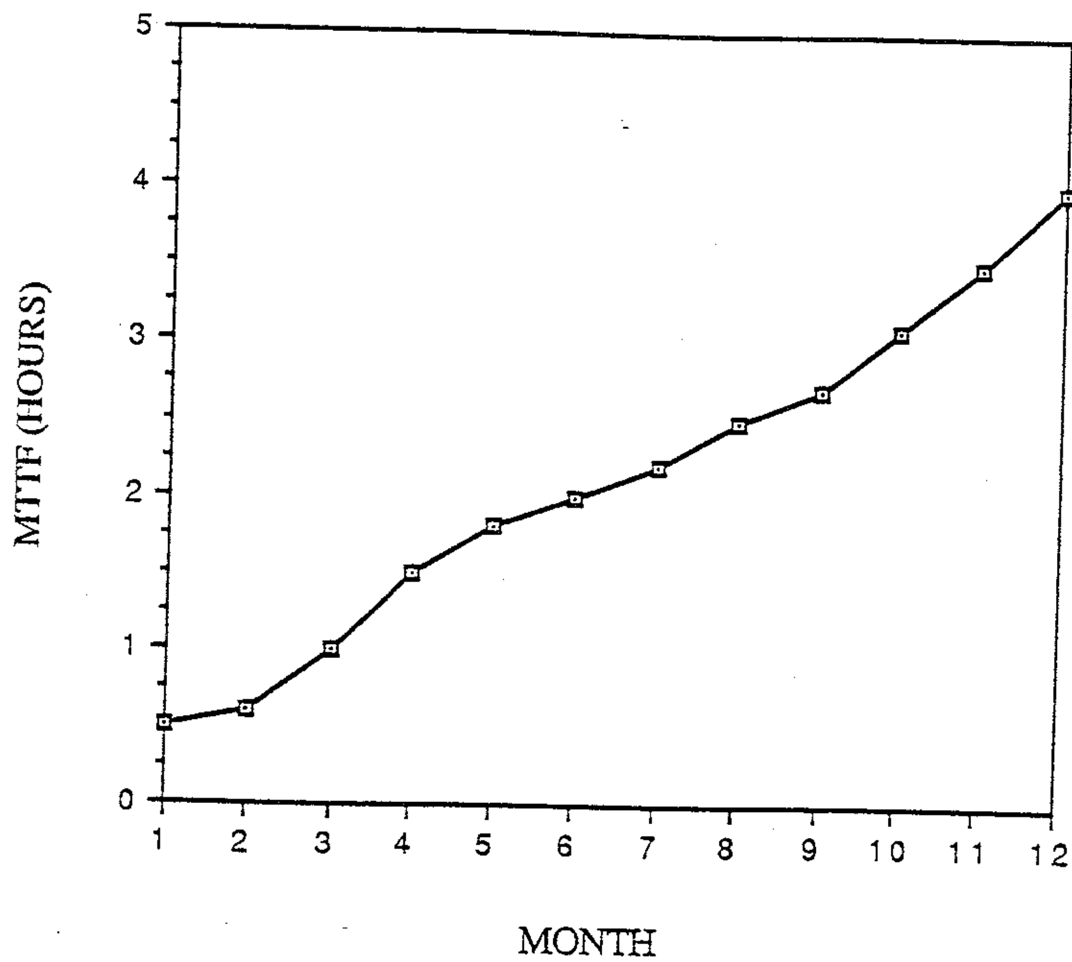

Figure 5. A Behavioral Indicator: Mean Time to Failure (MTTF)

Affective Indicator can signal that the values of the DIs may have changed and the predictions may no longer be valid. Conformance to objectives as assessed by each set of indicators, Design and Management, should remain consistent. Thus, in order to serve as a baseline for statistical comparison, the DIs must be recomputed. Figure 6 on page 46 illustrates the two types of Behavioral MIs: Non-affective and Affective.

3.2.2.1. Non-affective Behavioral Indicators

The following working definition is provided:

> A Behavioral MI is ***non-affective*** if a change in its value does not imply a change in values of DIs.

More intuitively, a Non-affective Indicator does not deal directly with maintenance activities. Thus, a change in the value of a Non-affective Indicator does not suggest that the presence of the objectives as measured by DIs should change in accordance.

The following is an example of a Non-affective MI:

Defect Density

The Defect Density Indicator [IEEE85, MACJ86] is used to identify the most defect-prone units of the software product. Identification of these units is useful because the majority of defects are usually concentrated in the same software units of a software product. By reviewing the defects detected in the product and the units in which they are located, the reliability of the product may be evaluated.

The Defect Density Indicator can be used during product acquisition and during deployment. This indicator is used during product acquisition by reviewing the defects found during the

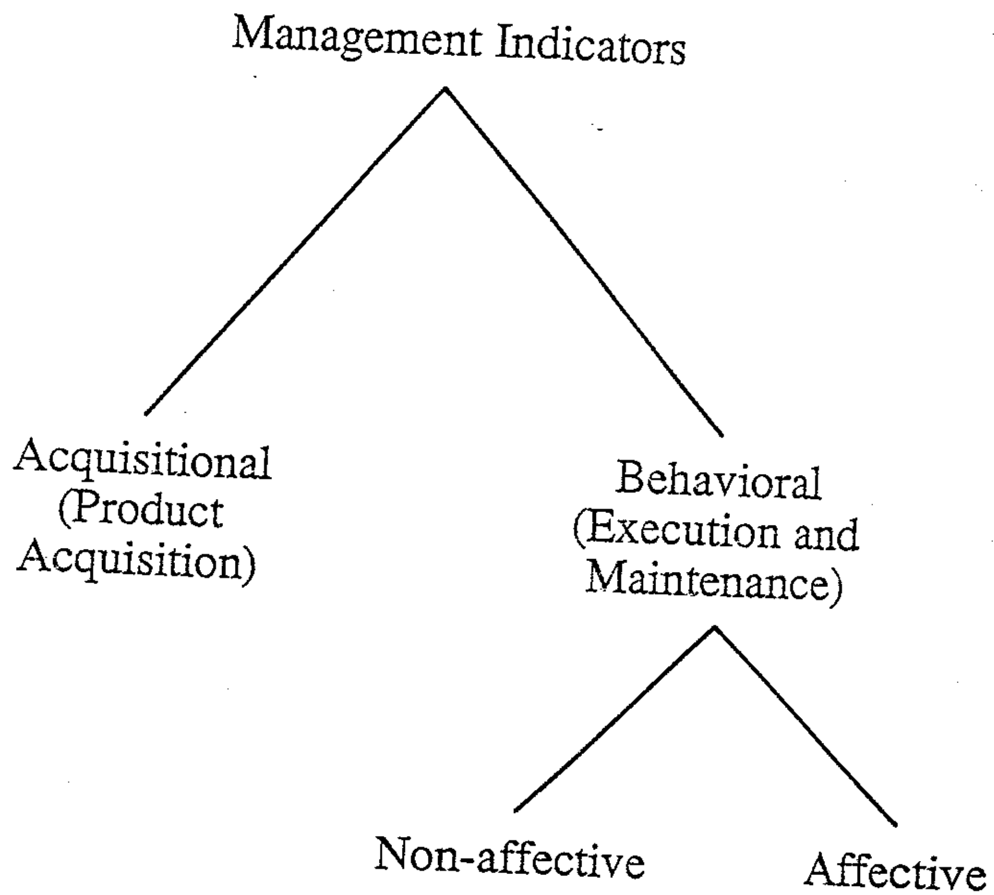

Figure 6. Management Indicators: Non-affective vs. Affective

design and code inspections and the testing phases of the development process. This indicator can also be used during deployment by examining the defects detected from operational use.

The Defect Density Indicator is a Non-affective Indicator because a change in its value does not imply a change in values of DIs. Although modification of the software product may have occurred during the time in which data for this indicator is collected, the results of modification are not being directly addressed by this indicator.

Defects are used in determining the reliability of software units, because the absence of the defects in the units cannot be shown. The number of defects in certain software units is by itself not appropriate as a means of identifying defect-prone units because software units are of different sizes. Thus, a normalization factor must be utilized. The normalization factor utilized by the Defect Density Indicator is implemented by determining the size of the unit in lines of source code (SLOC) or thousands of lines of source code (KSLOC). The value of the Defect Density of each unit can be calculated by the following:

$$\text{Defect Density} = \frac{\text{number of defects found in unit}}{\text{size of unit (SLOC or KSLOC)}}$$

The number of defects found in a unit is the number of times that modifications have been made to that unit because a defect was detected. Because the size of a unit is not known until coding is completed, the number of lines of source code must be approximated when implementing this indicator utilizing data from the design inspection processes.

To gain further insight into the Defect Density Indicator, a vertical bar chart is plotted of the Defect Densities of each unit. See Figure 7 on page 48. The software units are labeled across the horizontal axis, while the values of the Defect Densities are plotted along the vertical axis. The height of the bars indicates the relative Defect Densities of the individual units.

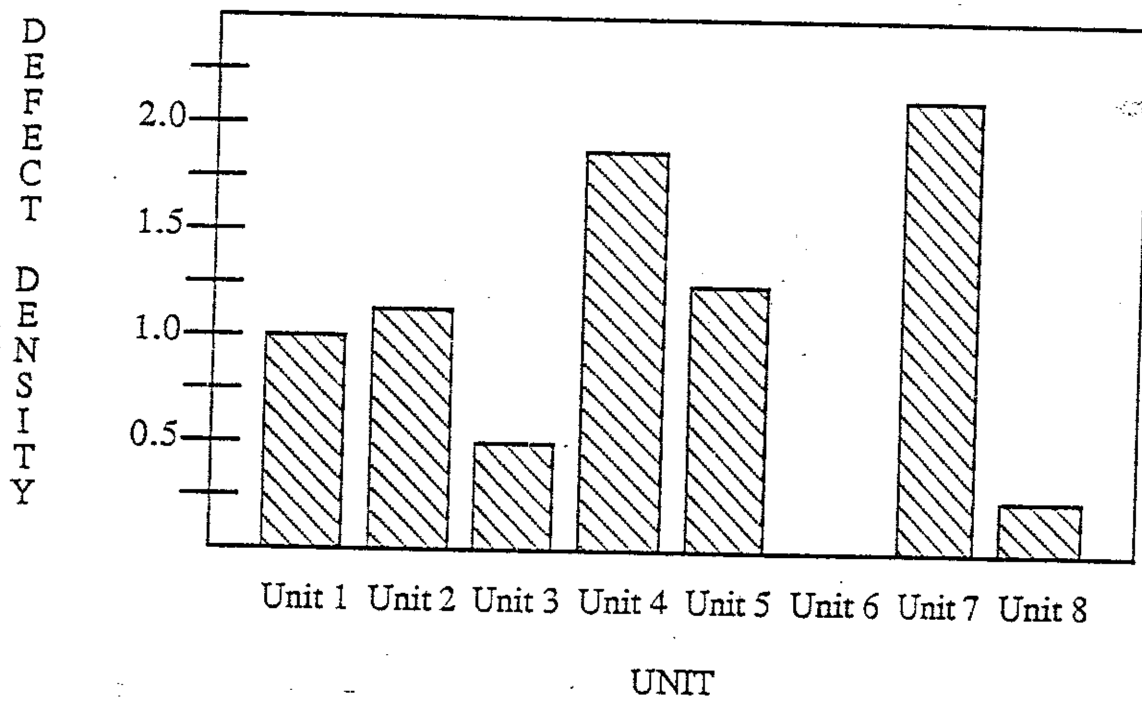

Figure 7. A Non-affective Indicator: Defect Density

By comparing the Defect Densities of the units, defect-prone units of the software product can be identified. The defect-prone units are likely to be the most unreliable units of the software in the future, because defects tend to occur in the same units of code. More modification within a unit increases the likelihood that more defects occur within that unit. Such units are also likely to involve a majority of the effort expended during the testing and maintenance phases. Early identification of these units allows corrective actions to be taken to ensure that defects which may appear in these units in the future are reduced, including a total redesign and recoding if necessary.

The Defect Density Indicator does not deal with maintenance activity characteristics which can affect the design structure of the software product. Thus, it is a Non-affective Indicator. On the other hand, Affective Indicators deal directly with modification characteristics. Modification of the product introduces the possibility of a change in the design structure.

3.2.2.2. Affective Behavioral Indicators

The following working definition is provided:

A Behavioral MI is ***affective*** if a change in its value implies a potential change in values of DIs.

More intuitively, an Affective Indicator deals directly with maintenance activities. Thus, a change in the value of an Affective Indicator suggests that the values of the objectives as determined by the bottom-up evaluation process should change in accordance.

The following is an example of an Affective MI:

Maturity Index

The Maturity Index Indicator [IEEE85] is used to determine the maturity of a software product at each delivery, or on a periodic basis. By taking into account the stability of the program, reliability of the product may be evaluated.

The Maturity Index Indicator can be used during deployment. It cannot be implemented during product acquisition. This indicator is implemented during deployment at the time of each delivery or major block build, or on a periodic basis (e.g. every year). The modifications made to the program for the release or during the time period must be examined.

It is an affective indicator because a change in its value implies a potential change in values of DIs. The modification of the software product is being directly addressed by this indicator.

The value of the Maturity Index Indicator is determined by the following formula:

$$\text{Maturity Index} = \frac{M_T - (F_a + F_c + F_{\text{del}})}{M_T}$$

where:

- M_T =number of functions (modules) in the current delivery,
- F_a =number of functions (modules) in the current delivery which were added since the previous delivery,
- F_c =number of functions (modules) in the current delivery which were changed from the previous delivery, and
- F_{del} =number of functions (modules) in the previous delivery which were deleted in the current delivery.

The factor $(F_a + F_c + F_{del})$ is the total of all functions (modules) that were added, changed, or deleted in the current delivery. All are treated the same because all are functions (modules) in which modifications have been made for the current delivery.

To determine if the software product is approaching a stable configuration, the Maturity Index is plotted for each release, or designated time period. See Figure 8 on page 52 for an example. The deliveries, or periods of time, are labeled along the horizontal axis. The values of the Maturity Indicator are plotted along the vertical axis. The first release consisted of 100 total functions, of which 10 were added, 35 were changed, and 5 were deleted for a total of 50 functions in which modifications had been performed. These values result in a Maturity Index of 0.5. Plotting continues in this manner for each subsequent release or time period.

By subtracting the total number of functions (modules) in which modifications have been made in this formula, a value of 1 or less is obtained for the value of the Maturity Index Indicator. A reliable software product should stabilize; that is, less of the program should have to be modified as a result of recognized inadequacies of the product for each successive release or time period, indicating that the plotted line should increase and flatten off towards a value of 1, as seen in Figure 8. A negative slope of the plotted line reveals that more of the program had to be modified for that release or time period than for the previous time frame. Thus, the recognized inadequacies of the system covered a greater portion of the program.

In summary, MIs are of two types: Acquisitional and Behavioral. Behavioral Indicators are either Non-affective or Affective. An example of each has been presented in this chapter. For a more extensive enumeration, see Appendix A.

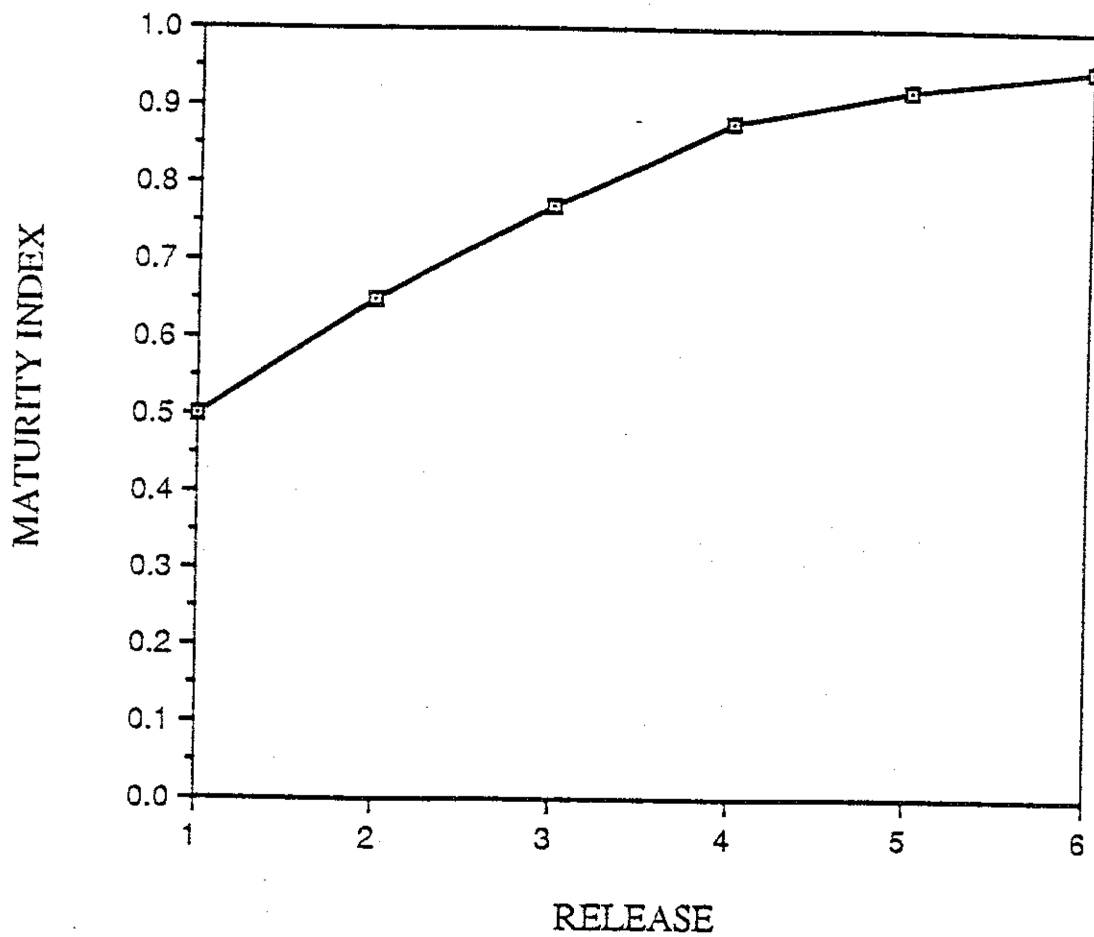

Figure 8. An Affective Indicator: Maturity Index

3.2.3. Evolutionary View of Management Indicators

As is the case with the MTTF Indicator and the Maturity Index Indicator, discussed in sections 3.2.2 and 3.2.2.2 respectively, many Management Indicators require implemention in an evolutionary fashion to enable temporal comparisons to be made. That is, their values must be calculated and examined over periods of time to enable trends to be detected in order that reliability and maintainability problems of a software product can be recognized.

An evolutionary view of Management Indicators and the collection of data over the development process and the entire Operations and Maintenance Phase provide for more accurate measurements by enabling isolated anomalies to be dampened and the norm to be reflected. Also, these two activities imply the necessity of a database to hold the information collected. The processes of data collection and interpretation of the Management Indicators should be automated to include features such as a procedure for recording STRs and performing the calculations of the indicator values. Automation reduces the effort and can assist in narrowing subjective variations [ARTJ87a].

3.2.4. Multiple View of Management Indicators

Reliability and maintainability are not affected by just a single factor, and therefore cannot be judged by one indicator. No single measurement can reveal all aspects necessary to determine the presence (or absence) of an objective in a software product. Multidimensional objectives, e.g. reliability and maintainability, cannot be judged from singular aspects.

For example, consider the Person-Hours per Major Defect Detected Indicator as discussed in section 3.2.1. Based solely on this indicator, a determination cannot be made as to whether a problem exists with the product's reliability, maintainability, both, or neither. While a high value *may indicate* a reliability and/or maintainability problem, a problem may not even exist at all. Locating

defects during the inspection processes may have taken a long time due to some other reason. Thus, Management Indicators which measure complementary and contrasting aspects of the objectives are needed in order to effectively determine the extent to which the objectives are realized in a product.

3.3. Relationship between Design and Management Indicators

Both Design and Management Indicators are used to determine the degree to which objectives are present (or absent) in a software product. Design Indicators focus on the design characteristics of the code and supporting documention to measure existence of objectives, while Management Indicators focus on the acquisitional, executional, and maintenance characteristics of the developed product.

The bottom-up evaluation process defined by the OPA provides the framework for determining the extent to which the objectives are present in a product from a design sense. This determination is accomplished by first utilizing the Design Indicators to measure the presence of the attributes. These measurements are propagated up through the linkages among the attributes, principles, and objectives. By propagating the presence of the attributes through the attribute/principle linkages, the extent to which each principle is employed in the development process is determined. Next, by propagating the extent to which each principle is utilized through the principle/objective linkages, the extent to which each objective is realized in the product can be assessed.

The capability of determining the presence of the objectives in a software product by implementation of the bottom-up evaluation process provides a basis for the synthesis of a software quality

assessment procedure [WHIR87]. As noted by Arthur, et. al. [ARTJ87b], a software quality assessment procedure must involve a systematic approach to assessing a product's (or process') conformance to acceptance standards. According to [ARTJ88], the evaluation procedure, and effectively the bottom-up evaluation process, provides a rigorous framework for:

- relating *acceptance criteria* based on attributes to software engineering principles and objectives, and

- defining *acceptance levels* based on measures reflecting the achievement of objectives, principles, and attributes.

More intuitively, acceptance standards can be set by stating desired levels at which objectives should be realized in a software product. After measuring such levels, conformance to desired levels can be assessed.

Measuring the presence or absence of objectives as determined through the use of Design Indicators and the bottom-up evaluation process should provide a basis for predicting the extent to which the objectives are achieved in the post-developed product. For example, if the objective of reliability is determined to be at a certain level in the product by the use of Design Indicators, then the product should exhibit a similar level of reliability as measured through acquisitional and behavioral characteristics. That is, the product at acquisition time and during execution should reflect the same level of reliability as predicted through Design Indicators.

Management Indicators, which focus on the acquisitional, behavioral, and maintenance characteristics of a product have been proposed as a means to validate the predictive capability of the bottom-up evaluation process. Both Design Indicators and Management Indicators are used to measure product quality relative to achieving desired objectives in a product. As can be seen in Figure 9 on page 57, both are tied to the determination of product quality through the same set of objectives, e.g. reliability and maintainability. They differ, however, in their perspective of a

product. As discussed by Arthur, et. al. [ARTJ88], Design Indicators employ the design characteristics found in the product (code and supporting documentation). On the other hand, Management Indicators exploit acquisitional, behavioral, and modification characteristics inherent to

1. the development process through

   a. *periodic* requirements analysis and specifications reviews, design and code inspections, and

   b. unit, system, and acceptance testing, and

2. the execution/maintenance activities associated with the *production* version of the product.

Figure 9 illustrates both the design perspective and the management perspective of the software product and how each reflects a measure of reliability and maintainability. It also illustrates the roles that managers, both the acquisition manager and the operations manager, play in obtaining data necessary to determine the presence of the objectives of reliability and maintainability in the developed product.

Whereas Design Indicators are used to measure the presence of attributes in the product and the measurements are propagated to determine the presence of objectives, Management Indicators (as far as can be determined at this time) seem to have a more obvious relationship with the objectives. Management Indicators also provide a framework for assessing a product's conformance to acceptance standards that are stated for product acquisition and deployment.

The validation of the predictive capability can be accomplished by measuring the presence of the objectives during acquisition and deployment of the product with the Management Indicators and then determining the correlation between these values and those obtained with the Design Indicators. If the measurements of the objectives by the utilization of Design Indicators can be validated

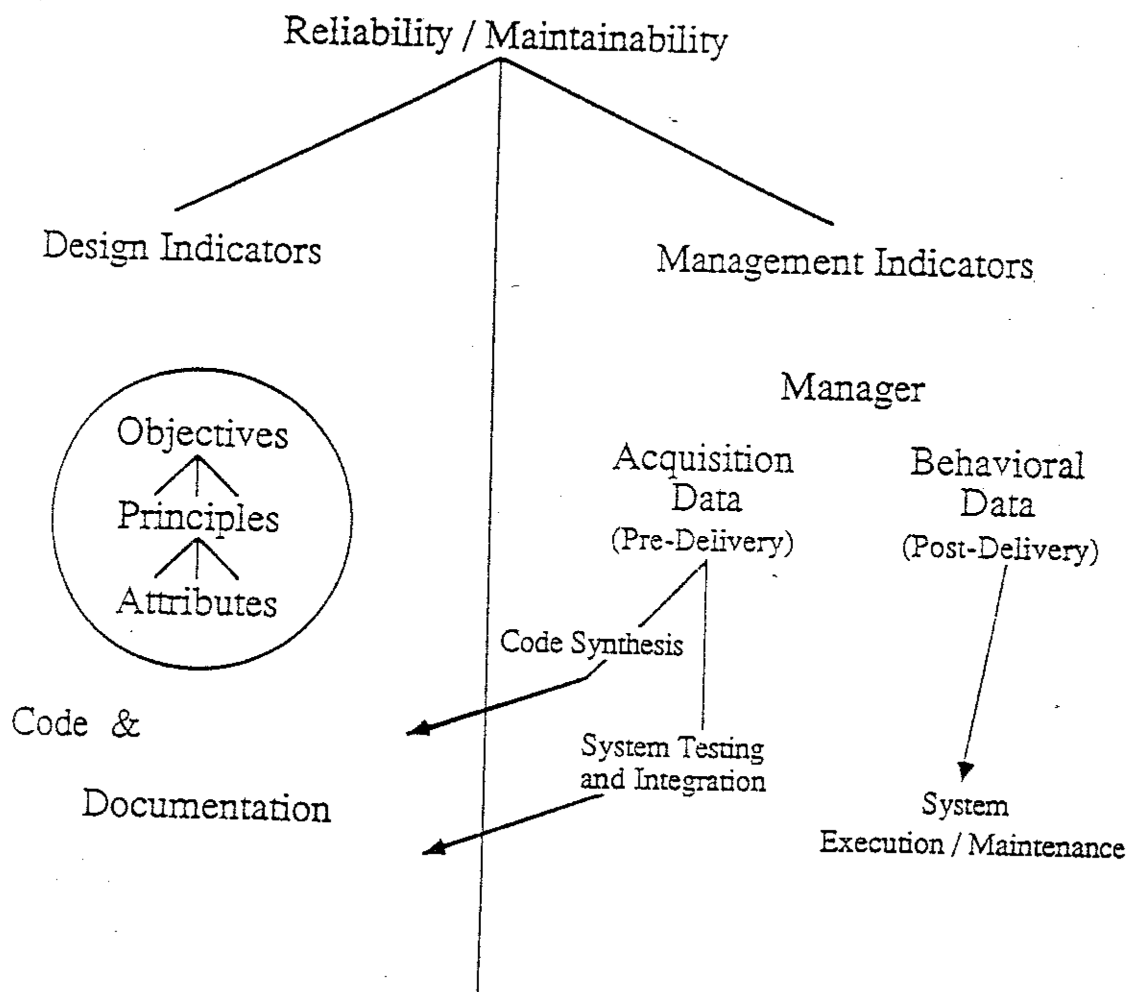

Figure 9. Design vs. Management Indicators

as predictive of post-development behavior, then a basis exists for recognizing undesirable product characteristics *early* in the development stages, and certainly *before* product acquisition.

4. Concluding Remarks

This report has focused on the investigation and identification of Management Indicators relative to the software engineering objectives of reliability and maintainability. Management Indicators are proposed by this research to validate the predictive capability of the bottom-up evaluation process defined by the Procedural Approach to the Evaluation of Software Development Methodologies (the OPA approach). This chapter summarizes the results of the investigation of Management Indicators and how Management Indicators relate to Design Indicators, the underlying measurements of the bottom-up evaluation process. Also presented in this chapter are the author's contributions to this research effort and future work associated with the validation.

4.1. Summary

The bottom-up evaluation process as defined by the OPA evaluation procedure enables one to determine the extent to which software engineering objectives, and in particular, reliability and maintainability, are present in a software product by examining its design structure. The presence of objectives, as determined by this process, is also hypothesized to be a predictor of the presence

of the same objectives in the post-developed product. That is, measurements based on the bottom-up evaluation process should be able to predict post-development behavior of a product with respect to the objectives.

The research effort described in this report suggests that Management Indicators can provide a framework for validating the above mentioned predictive capability by acting as a counterpart to Design Indicators, the underlying measurements of the bottom-up evaluation process. Design Indicators are variables that indicate the presence (or absence) of one or more software attributes as reflected by design structure characteristics of a product. The presence of the attributes can then be propagated through the linkages among attributes, principles, and objectives, to determine the extent to which the objectives are present in the product. Management Indicators are also variables that are related to one or more of the software engineering objectives, but which reflect acquisition, execution, or maintenance characteristics of a product. In other words, Design Indicators employ a design perspective in the measurement of the objectives, while Management Indicators take a post-development perspective as regarded from an acquisitional and behavioral point of view.

Key issues in validating the predictive capability of the bottom-up evaluation process include a recognition of the following:

- *The distinction between Acquisitional and Behavioral Management Indicators.* A developed product can be assessed for quality during two periods:

  - during Product Acquisition through analysis of data collected during the software development process, and

  - during Execution and Maintenance through analysis of data collected during execution and maintenance activities.

Acquisitional Indicators are Management Indicators whose values are determined during product acquisition, and Behavioral Indicators are those whose values are determined during execution and maintenance.

- *The distinction between Non-affective and Affective Behavioral Management Indicators.* As software maintenance is performed on a product its design structure may change, which can cause a change in the values of Design Indicators. A Management Indicator that deals directly with maintenance activities is termed Affective, because a change in its value implies a potential change in the values of Design Indicators in accordance with its change. A Non-affective Management Indicator does not deal directly with maintenance activity characteristics.

- *The necessity for an evolutionary view of Management Indicators.* Many Management Indicators require that their values be calculated and examined over periods of time to enable trends to be detected. Also, a large amount of data must be collected and analyzed during the development, execution, and maintenance of a product. These two activities contribute to more accurate measurements by dampening the effect of isolated anomalies and thereby reflecting the norm.

- *The necessity for multiple complementary and contrasting Management Indicators.* By itself, a single Management Indicator is not always reliable in determining the extent to which an objective is present in a product. Effectively, multiple indicators are needed to assess the presence or absence of an objective by measuring complementary and contrasting aspects of the objective.

4.2. Author's Contribution

The research effort described in this report has focused on the use of Management Indicators as a basis for validating the predictive capability of the bottom-up evaluation process defined by the OPA evaluation procedure. In particular, this effort has entailed an investigation of fundamental relationships linking Management Indicators to Design Indicators. Relative to the investigation, the author's major contributions include:

1. *The identification of 20 Management Indicators.* Based on an extensive literature search and an examination of numerous methods for assessing the quality of a software product during product acquisition, execution, and/or maintenance, the author has identified 20 measurement techniques appropriate to the validation effort. These techniques are judged for inclusion as Management Indicators according to their ability to represent software quality from a manager's perspective, their ease of use and application, and their relation to the objectives of reliability and maintainability. Of the 20 Management Indicators chosen:

   - 17 deal with reliability, and

   - 10 deal with maintainability.

   Moreover,

   - 11 of the 20 can be used at product acquisition time, and

   - 19 can be used during deployment.

   Table 6 on page 64 provides a list of the Management Indicators identified by this research as they relate to reliability and/or maintainability. These Management Indicators are classified

in Table 7 on page 65 according to type: Acquisitional and/or Behavioral. Behavioral Indicators are then distinguished as Affective or Non-affective.

2. *A recognition and investigation of the distinction between Acquisitional and Behavioral Management Indicators.* In exploring issues surrounding the quality of a software product relative to software engineering objectives, the author recognizes that post-development quality can be judged at two different time periods: product acquisition and product deployment. Data collected during each period can provide the manager with judgements of product quality.

3. *A recognition of the distinction between Non-affective and Affective Behavioral Management Indicators.* Further investigation of Behavioral Indicators led the author to recognize that a change in value of certain Behavioral Indicators implies a potential change in the values of Design Indicators. This potential change is the consequence of maintenance activities that may influence the design structure of a product. Management Indicators that deal directly with the maintenance of a product are termed affective, while those which do not are termed non-affective. Because Management Indicators must be correlated with Design Indicators, this recognition is crucial to validating the predictive capability of the bottom-up evaluation process. In particular, if values of Affective Indicators do change over the period of deployment, then a recalculation of the Design Indicators must be carried out.

4.3. Future Directions

Based on the identification of Management Indicators, the following set of steps is proposed [ARTJ88] as one approach to validating the predictive capabilities of the bottom-up process defined by the OPA approach and ultimately establishing a validated procedure for assessing software quality:

Table 6. Management Indicators: Relationship to Objectives

	Reliability	Maintainability
Defect Removal Rate	Yes	Yes
Defect Age Profile	No	Yes
Defect Density	Yes	No
Defect Detection Efficiency	Yes	No
Extent of Modification	Yes	Yes
Defect Introduction	Yes	Yes
Defect Days	Yes	Yes
Defect Index	Yes	No
Defect Distributions	Yes	No
Maturity Index	Yes	No
Person-Hours per Major Defect Detected	Yes	Yes
Person-Hours per Source Statement Modified	No	Yes
Requirements Traceability	Yes	Yes
Test Coverage	Yes	No
Robustness	Yes	No
Mean Time to Failure	Yes	No
Mean Time to Repair	No	Yes
Input Index	Yes	No
Failure Rate	Yes	No
Availability	Yes	Yes

Table 7. Management Indicators: Type Classification

	Acquisitional	Behavioral	Affective or Non-affective
Defect Removal Rate	Yes	Yes	Non-affective
Defect Age Profile	Yes	Yes	Non-affective
Defect Density	Yes	Yes	Non-affective
Defect Detection Efficiency	Yes	Yes	Non-affective
Extent of Modification	No	Yes	Affective
Defect Introduction	No	Yes	Non-affective
Defect Days	Yes	Yes	Non-affective
Defect Index	Yes	Yes	Non-affective
Defect Distributions	Yes	Yes	Non-affective
Maturity Index	No	Yes	Affective
Person-Hours per Major Defect Detected	Yes	No	N/A
Person-Hours per Source Statement Modified	No	Yes	Affective
Requirements Traceability	Yes	Yes	Affective
Test Coverage	Yes	Yes	Affective
Robustness	Yes	Yes	Non-affective
Mean Time to Failure	No	Yes	Non-affective
Mean Time to Repair	No	Yes	Non-affective
Input Index	No	Yes	Non-affective
Failure Rate	No	Yes	Non-affective
Availability	No	Yes	Non-affective

1. Complete the investigation of indicators, both Design Indicators and Management Indicators, relative to reliability and maintainability of a software product.

2. Identify a software development project suitable for and amenable to the collection of raw data for assessing both sets of indicators.

3. Develop an automatic capability for extraction and analysis of data relative to both sets of indicators.

4. Collect data throughout development and deployment activities. To serve as a basis for experimentation and refinement of both sets of indicators a control group must be established.

5. Perform validation of Design Indicators through statistical comparison with the control group.

6. Perform validation of the predictive capability of the bottom-up evaluation procedure by statistical comparison of the objectives determined present by each set of indicators.

In addition to, and to some extent part of the validation effort itself, the following subject matter warrants further investigation:

- *Exploration of the Management Indicator/Objective relationship.* Unlike the well-defined linkages which relate Design Indicators to objectives (DI/attribute/principle/objective linkages), the vertical relationships linking Management Indicators to objectives appear to be more obvious. The Management Indicator/Objective relationship must be investigated to determine if any intervening linkages exist, and resultant findings must be substantiated.

- *Exploration of the horizontal relationship between the design and management domains.* If vertical linkages are discovered in the management domain, these linkages should be examined to determine how they relate to the linkages in the design domain. For example, one might ask if counterparts to attributes and principles exist within the management domain.

- *Investigation and analysis of metrics defining Management Indicators.* Management Indicators relative to the objectives of reliability and maintainability have been identified by this research. Each Management Indicator is composed of an attribute/value pair. The value element is computed through metrics. These metrics defining the Management Indicators must be further analyzed before their appropriateness relative to each objective/value pair can be determined.

- *Refinement of Design Indicators and Management Indicators to compare and contrast their inherent qualities.* The two types of indicators must be based on a common scale to allow statistical comparison.

To summarize, this work focused on the investigation and identification of Management Indicators relative to the software engineering objectives of reliability and maintainability. Management Indicators are proposed to validate the predictive capability of the bottom-up evaluation process defined by the OPA procedure. This capability involves the prediction of the extent to which objectives are realized in the production version of a software product. This validation will provide for the following:

1. *Quality to be built-in a software product.* By employing the bottom-up evaluation process periodically throughout the development of a product, the quality can be controlled.

2. *A quality product to be acquired.* By employing the bottom-up evaluation process at product acquisition time, the quality can be compared to stated acceptance standards.

Both of these elements should prove valuable to ensuring the quality of software.

Bibliography

ADRW82 Adrion, W. R., Branstad, M. A., and Cherniavsky, J. C., " Validation, Verification, and Testing of Computer Software, " *ACM Computing Surveys,* Vol. 14, No. 2, 1982.

AFSC87 Air Force Systems Command Pamphlet 800-14, *Software Quality Indicators; Management Quality Insight,* Department of the Air Force, January 20, 1987.

ARTJ86 Arthur, J. D., Nance, R. E., and Henry S. M., *A Procedural Approach to Evaluating Software Development Methodologies: The Foundation,* Technical Report TR 86-24, Virginia Tech, Blacksburg, Virginia, 1986.

ARTJ87a Arthur, J. D. and Nance, R. E., *Developing an Automated Procedure for Evaluating Software Development Methodologies and Associated Products,* Technical Report TR 87-16, Virginia Tech, Blacksburg, Virginia, 1987.

ARTJ87b Arthur, J. D., Nance, R. E., Lavender, R. G., and Stevens, K. T., Presentation: *Fundamental Issues in Software Quality Assurance,* Virginia Tech and The Systems Research Center, Blacksburg, Virginia, November 13, 1987.

ARTJ87c Arthur, J. D. and Nance, R. E., Presentation: *The Evaluation of Software Development Methodologies: An Overview,* Virginia Tech and The Systems Research Center, Blacksburg, Virginia, April 16, 1987.

ARTJ88 Arthur, J. D., Nance, R. E., Rosson, C. V., and Stevens, K. T., Presentation: *Fundamental Issues in the Automation of Software Quality Assurance: Validation of Design Indicators Through Management Indicators,* April 1988, Virginia Tech and The Systems Research Center, Blacksburg, Virginia, April 8, 1988.

BOEB84 Boehm, B. W., " Software Life Cycle Factors, " *Software Engineering Handbook,* C. R. Vick and C. V. Ramamoorthy (editors), Van Nostrand Rheinhold Co., New York, New York, 1984.

DANA87 Dandekar A. V., *A Procedural Approach to the Evaluation of Software Development Methodologies,* M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, Virginia, September, 1987.

EVAM87 Evans, M. W. and Marciniak, J., *Software Quality Assurance and Management,* John Wiley and Sons, Inc., New York, New York, 1987.

FAIR85 Fairley, R., *Software Engineering Concepts,* McGraw-Hill, Inc., New York, New York, 1985.

GILT77 Gilb, T., *Software Metrics,* Winthrop Publishers, Inc., Cambridge, Massachusetts, 1977.

GILP83 Gilbert, P., *Software Design and Development,* Science Research Associates, Inc., Chicago, Illinois, 1983.

GLAR79 Glass, R. L., *Software Reliability Guidebook,* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1979.

HENS88 Henry, S. M., " A Technique for Hiding Proprietary Details While Providing Sufficient Information for Researchers, " *Journal of Systems and Software,* January, 1988.

IEEE83 IEEE Standard 729, *IEEE Standard Glossary of Software Engineering Terminology,* IEEE Computer Society, February 18, 1983.

IEEE85 IEEE Project P982, Draft Standard P982, *Software Reliability Measurement,* IEEE Computer Society, June 1, 1985.

KEAJ86 Kearney, J. K., Sedlmeyer, R. L., Thompson, W. B., Gray, M. A., and Adler, M. A., " Software Complexity Measurement, " *Communications of the ACM,* Vol. 29, No. 11, pp.1044-1050, November, 1986.

KOPH79 Kopetz, H., *Software Reliability,* MacMillian Press Ltd., London, England, 1979.

LISB72 Liskov, B., " A Design Methodology for Reliable Systems," *AFIPS Conference Proceedings,* Vol. 41, Part 1, pp.191-199, 1972.

MACJ86 MacMillian, J. and Vosburgh, J. R., *Software Quality Indicators,* Scientific Systems, Inc., Cambridge, Massachusetts, September 24, 1986.

MYEG76 Myers, G. J., *Software Reliability:Principles and Practices,* John Wiley and Sons, Inc., New York, New York, 1976.

OSTL76 Osterweil, L. and Fosdick, L., " DAVE - A Validation, Error Detection and Documentation System for FORTRAN Programmers, " *Software: Practice and Experience,* Vol. 6, No. 4, 1976.

ROSL83 Rosenthal, L. S., " Planning and Implementing System Reliability, " *Proceedings of Total Systems Reliability Symposium,* IEEE Computer Society, pp.112-118, 1983.

SNEH85 Sneed, H. and Merey, A., " Automated Software Quality Assurance, " *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 9, 1985.

STEK88 Stevens, K. T., " A Taxonomy for the Evaluation of Computer Documentation, " M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, Virginia, 1988.

SWAE76 Swanson, E. B., " The Dimensions of Maintenance, " *Proceedings of the 2nd International Conference on Software Engineering,* IEEE Computer Society, 1976.

WHIR87 Whitner, R. B., Arthur, J. D., and Nance, R. E., Presentation: *Fundamental Issues in the Automation of Software Quality Assurance,* Virginia Tech and The Systems Research Center, Blacksburg, Virginia, August 27, 1987.

WIRN71 Wirth, N., " Program Development by Stepwise Refinement, " *Communications of the ACM,* Vol. 14, No. 4, pp,221-227, April, 1971.

Appendix A. Management Indicators of Software Reliability and Maintainability

This appendix provides a detailed enumeration of the Management Indicators (factors) identified in this research effort. The format of each Management Indicator included in this appendix is as follows:

- **A. Description** - The first section of each MI introduces the MI by providing a brief description which includes the MI's purpose, its application, and if it is related to reliability and/or maintainability.

- **B. Application Phase** - This section notes the periods in which the MI can be applied. An MI can be applied during product acquisition (Acquisitional MI) and/or deployment (Behavioral MI). If the MI is Behavioral, it is either non-affective or affective. Affective MIs deal directly with maintenance activity while Non-affective MIs do not.

- **C. Implementation** - The implementation section provides a discussion of the techniques and metrics that are used to compute the MI.

- **D. Interpretation** - The interpretation section discusses how the MI relates to the objectives of reliability and/or maintainability, i.e. it discusses how the presence or absence of the objectives in the software product may be indicated.

- **E. Additional Comments** - This section provides any comments on the MI not covered in the other sections.

- **F. References** - This section lists the publications in which the measurement technique of the MI was described. These references may provide further information.

Defect Removal Rate

A. Description

The Defect Removal Rate Indicator is used to determine the rate at which defects in the software product are being detected and removed. It is also used to determine the readiness for advancement to the next phase of the software life cycle during the development process. Implementation of this indicator is accomplished by tracking and comparing the cumulative number of open and closed Software Trouble Reports (STRs) over time. This indicator can be used in determining the reliability and/or maintainability of the product.

B. Application Phase

The Defect Removal Rate Indicator can be used during product acquisition and during deployment. It is a Non-affective Indicator. Section 2.3.6. discusses how an STR is compiled for use during these periods and how STRs and defects are related.

C. Implementation

The Defect Removal Rate Indicator is implemented as a line chart of the cumulative number of STRs, both open and closed, against time. Refer to Figure 10 on page 75 for an example.

Time is plotted on the horizontal axis. The time unit is dependent upon the software product under analysis. The time unit is usually chosen as the benchmark set for resolving STRs. Choosing this as the time unit provides a better framework for analyzing the maintainability of the product, as is discussed in the following section. For example, if one month is determined to be sufficient for

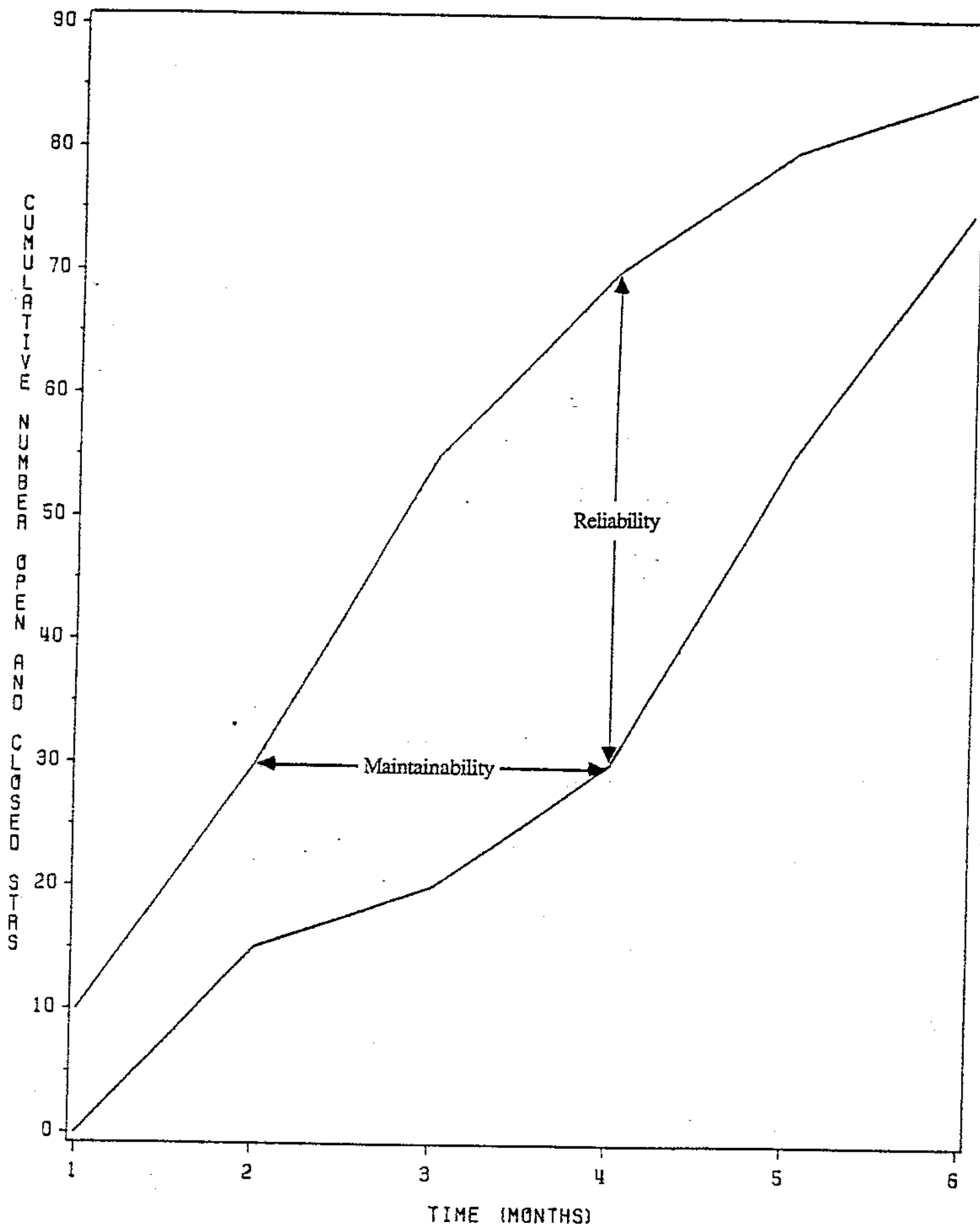

Figure 10. Defect Removal Rate

resolving STRs then the time unit is plotted as months, as in Figure 10. Other time unit examples are one week, two weeks, two months, etc. Figure 11 on page 78 provides examples where the time unit is one week.

The total cumulative number of both open and closed STRs at the end of each time period is plotted on the vertical axis. In Figure 10, it can be seen that 10 STRs were opened the first month and none were closed. During the second month 20 more were opened, for a cumulative total of 30 open STRs. No STRs were closed in the first month, but 15 of the STRs were closed in the second month, closing the cumulative number to 15. The third month saw 25 more STRs being opened and 5 STRs closed, resulting in 55 cumulative opened and 20 cumulative closed. Plotting continues in this manner until the phase is completed.

D. Interpretation

A large or increasing horizontal or vertical distance between the two plotted lines indicates a problem in the removal of defects. The problem could be attributable to the reliability and/or maintainability of the software product, if it is not attributable to factors such as insufficient resources available to resolve the defects.

Reliability:

The vertical distance between the two plotted lines at any point in time, as shown in Figure 10 for month 4, represents the number of open STRs at that time. This indicates the current level of reliability of the product, i.e. the number of STRs that have not been closed at that point in time.

If there is a large or increasing distance between the two plotted lines reliability may be suspect (i.e. more STRs are being opened than can be closed).

Maintainability:

The horizontal distance between between the two plotted lines represents the elapsed time between the date when the same number of STRs were opened and closed. This is shown in Figure 10 for month 4. This distance, which visually represents the rate of defect removal, indicates the level of maintenance being performed on the product at that point in time. If the time units are selected in conjunction with the benchmark, it is easily determined if the level of maintainability is meeting the benchmark, the horizontal distance is staying within one time unit. This distance can be used to estimate the amount of effort (time) needed to eliminate the open STRs.

If there is a large or increasing distance between the two plotted lines maintainability may be suspect (i.e. STRs are not being closed in a timely fashion).

E. Additional Comments

Near the end of each phase of the software life cycle the slope of the line of open STRs should flatten, displaying stabilization (i.e. fewer defects are being found). Also at the end of each phase the two plotted lines should meet, displaying that all STRs have been closed. These two observations should be prerequisites for advancing to the next phase in the software life cycle. Figure 11 on page 78 shows the combinations of meeting these prerequisites. The form of Figure A is desired. The cumulative number open line flattens revealing fewer defects are being detected and the two lines meet indicating that all open STRs have been closed. Figure B shows neither prerequisite met. In Figure C, the number of open STRs is stabilizing at the end of the phase, but all of the STRs which have been opened have not been closed. Although all STRs have been closed in Figure D, the number opened is not stabilizing, indicating STRs can probably continue to be opened at a fairly rapid rate.

To obtain a better indication of the reliability and maintainability of the software, the severity level of the STRs can be compiled. Only those STRs revealing defects above a specific severity level can be plotted. By separating serious STRs from trivial STRs which are easily corrected, a more accurate interpretation may result.

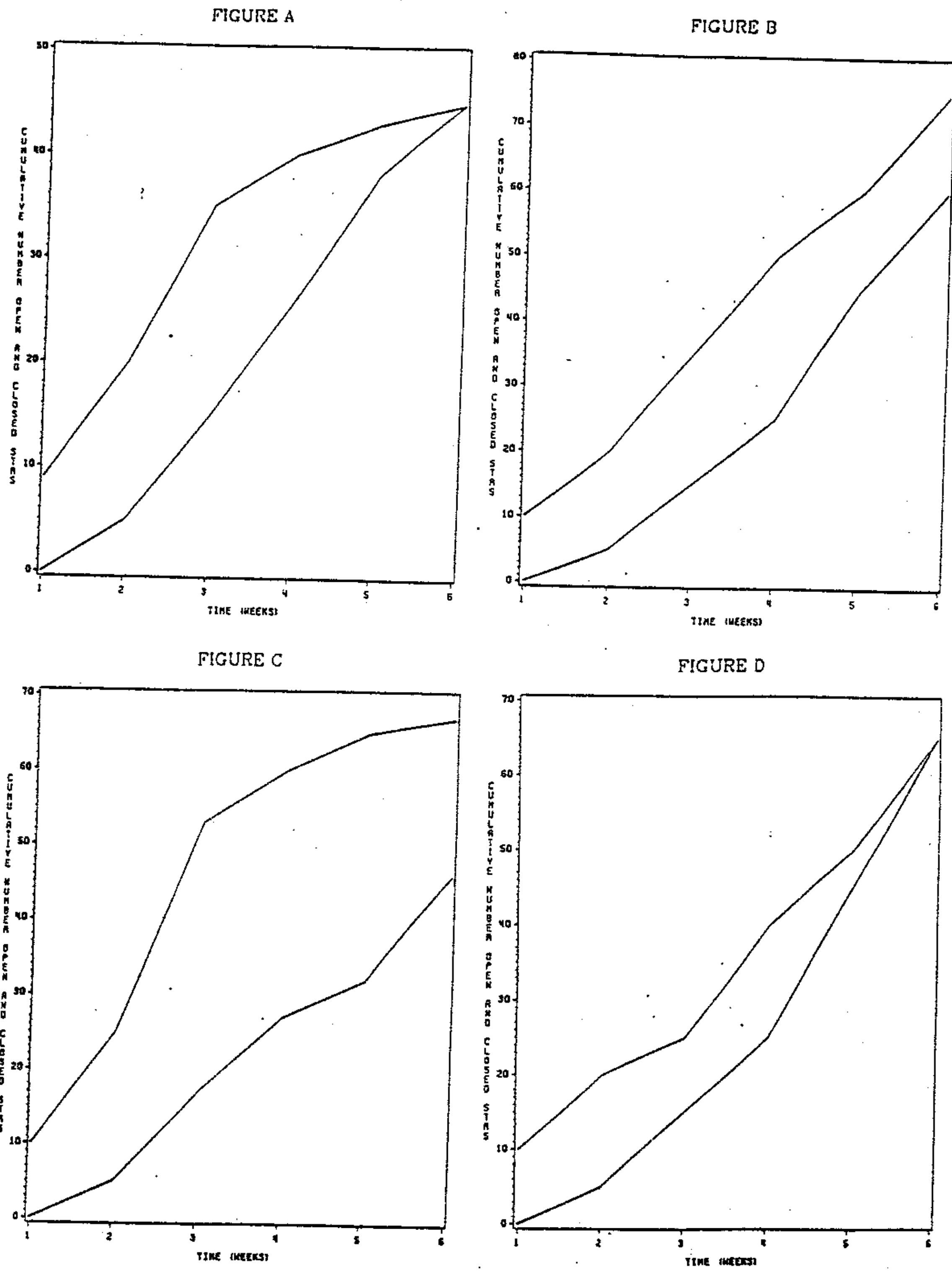

Figure 11. Defect Removal Rate: Examples

STRs can be grouped by structural level, i.e. the Defect Removal Rate Indicator can be implemented for each unit (or other structural level) with only those STRs requiring changes to the unit being plotted. This may give a greater idea of the reliability and maintainability of the specific units. Lower level analyses of this type may help identify why the software product may not be meeting its objectives because particular problems may be identified and analysed.

F. References

[AFSC87]/pp.9-12

[MACJ86]/pp.51-55

Defect Age Profile

A. Description

The Defect Age Profile Indicator is used to determine if the STRs are being closed in a timely fashion. Implementation of this indicator is accomplished by tracking and examining the number of open Software Trouble Reports (STRs) which have been open longer than a specified length of time. This indicator can be used in determining the maintainability of the software product.

B. Application Phase

The Defect Age Profile Indicator can be used during product acquisition and during deployment. It is a Non-affective Indicator. Section 2.3.6. discusses how an STR is compiled for use during these periods and how STRs and defects are related.

C. Implementation

The Defect Age Profile Indicator is implemented as a line chart of the current number of open STRs which are older than a specified length of time against time. Age of an STR is the length of time elapsed from the time when the STR is opened until the time it is closed. Refer to Figure 12 on page 81 for an example.

Time is plotted on the horizontal axis in the appropriate units of time for the software under analysis, as in the Defect Removal Rate Indicator. At the end of every time unit the open STRs whose age is older than the specified time unit are plotted on the vertical axis, i.e. STRs that are currently open and have been open at least the specified time unit are plotted. For example, as in

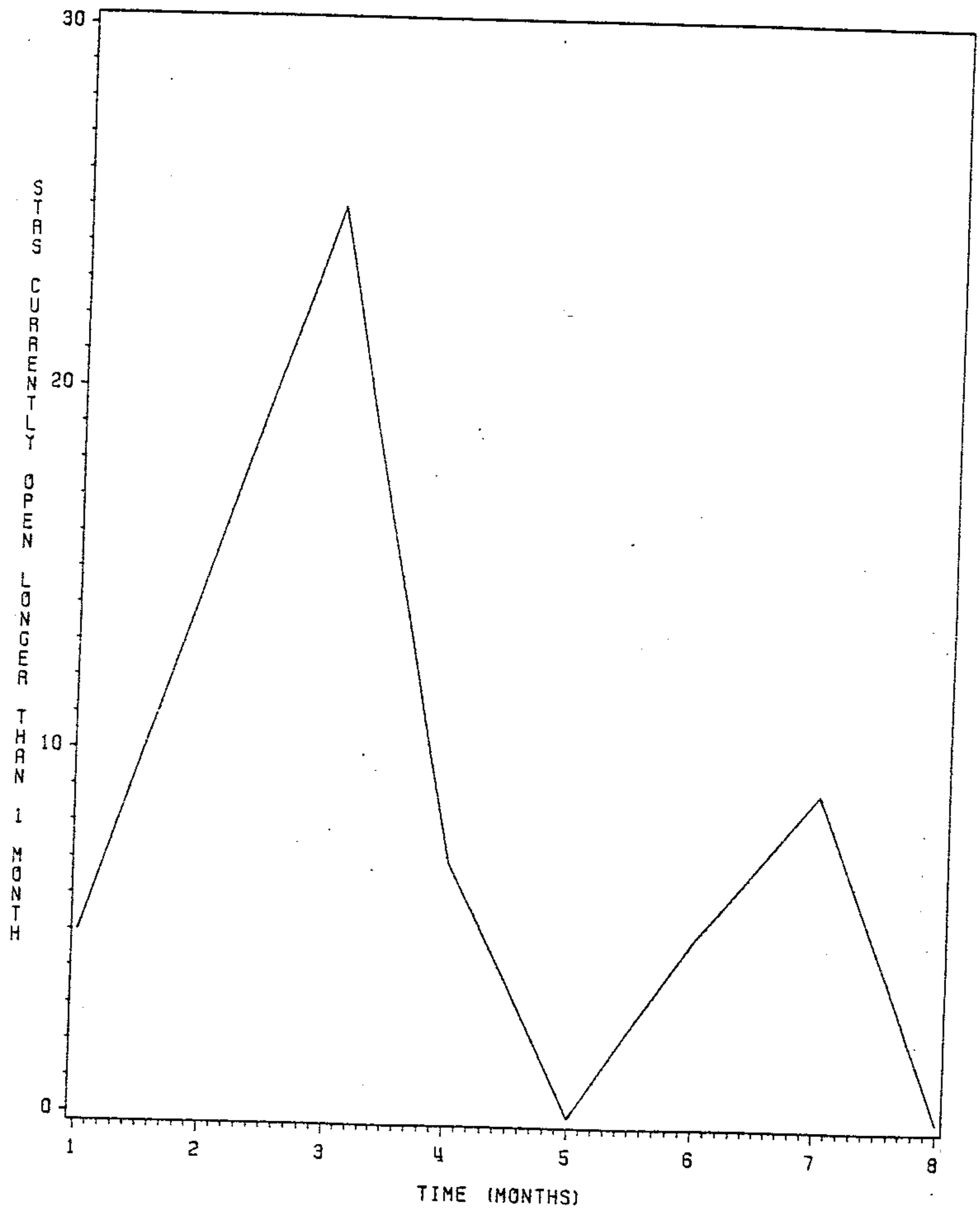

Figure 12. Defect Age Profile

Figure 12, at the end of the first month 5 STRs have been open for at least a month (which means that they must have been opened on the first day). At the end of the second month, 15 STRs had been open one month or longer. Plotting continues in this manner until the phase is completed.

D. Interpretation

Although the Defect Removal Rate Indicator can give an idea of the average time it takes to resolve STRs by the horizontal distance between the two plotted lines, it is not effective in determining how many of the STRs are actually exceeding the benchmark. Specifically, even if the average seems reasonable there may still exist a maintainability problem if a number of STRs have been open for a long period of time. The Defect Age Profile Indicator is used in identifying the number of STRs that are exceeding the benchmark.

Maintainability:

A maintainability problem may exist if there is a steadily increasing slope over time or if there is a sharp increase in the slope, indicating the difficulty in resolving STRs. When this occurs, the same STRs are remaining open for longer than two time units and/or a rapidly increasing number of STRs are exceeding the benchmark. This can indicate a more serious problem near the end of a phase, when there is less time to resolve these open STRs.

E. Additional Comments

The plotted line should approach the horizontal axis, i.e. zero trouble reports open, and meet it at the end of a phase, as shown in Figure 12. This indicates that all STRs have been closed as a prerequisite for advancing to the next phase in the software life cycle.

STRs can also be grouped by severity and/or structural level as described in the Defect Removal Rate Indicator, if necessary, for a more accurate representation.

F. References

[MACJ86]/pp.55-57

Defect Density

A. Description

The Defect Density Indicator is used to identify the most defect-prone units of the software product. Identification of these units is useful because the majority of defects are usually concentrated in the same software units of a software product. Implementation of this indicator is accomplished by keeping track of the defects detected in the software product and the units in which they are found. This indicator can be used in determining the reliability of the product.

B. Application Phase

The Defect Density Indicator can be used during product acquisition and during deployment. It is a Non-affective Indicator.

C. Implementation

Defects are used in determining the reliability of software units, because the absence of the defects in the units cannot be shown. The number of defects in certain software units is by itself not appropriate as a means of identifying defect-prone units because software units are of different sizes. Thus, a normalization factor must be utilized. The normalization factor utilized by the Defect Density Indicator is usually the size of the unit in lines of source code (SLOC) or thousands of lines of source code (KSLOC). The Defect Density of each unit can be calculated by the following:

$$\text{Defect Density} = \frac{\text{number of defects found in unit}}{\text{size of unit (SLOC or KSLOC)}}$$

The number of defects found in a unit is the number of times that modifications have been made to that unit because a defect was detected. Because the size of a unit is not known until coding is completed, the number of lines of source code must be approximated when using this indicator during the design inspection processes.

To gain further insight into the Defect Density Indicator, a vertical bar chart is plotted of the Defect Densities of each unit. See Figure 13 on page 86. The software units are labeled across the horizontal axis, while the values of the Defect Densities are plotted along the vertical axis. The height of the bars indicates the relative Defect Densities of the individual units.

D. Interpretation

By comparing the Defect Densities of the units, defect-prone units of the software product can be identified.

Reliability:

The defect-prone units are the most unreliable units of the software and are likely to be the most unreliable units in the future, because defects tend to occur in the same units of code. More modification within a unit increases the likelihood that more defects occur within that unit. Such units are also likely to involve a majority of the effort expended during the testing and maintenance phases. Early identification of these units allows corrective actions to be taken to ensure that defects which may appear in these units in the future are reduced, including a total redesign and recoding if necessary.

E. Additional Comments

The Defect Densities are usually recalculated periodically to identify the defect-prone units at that point in time. Although usually not calculated in an evolutionary fashion, this approach can reveal,

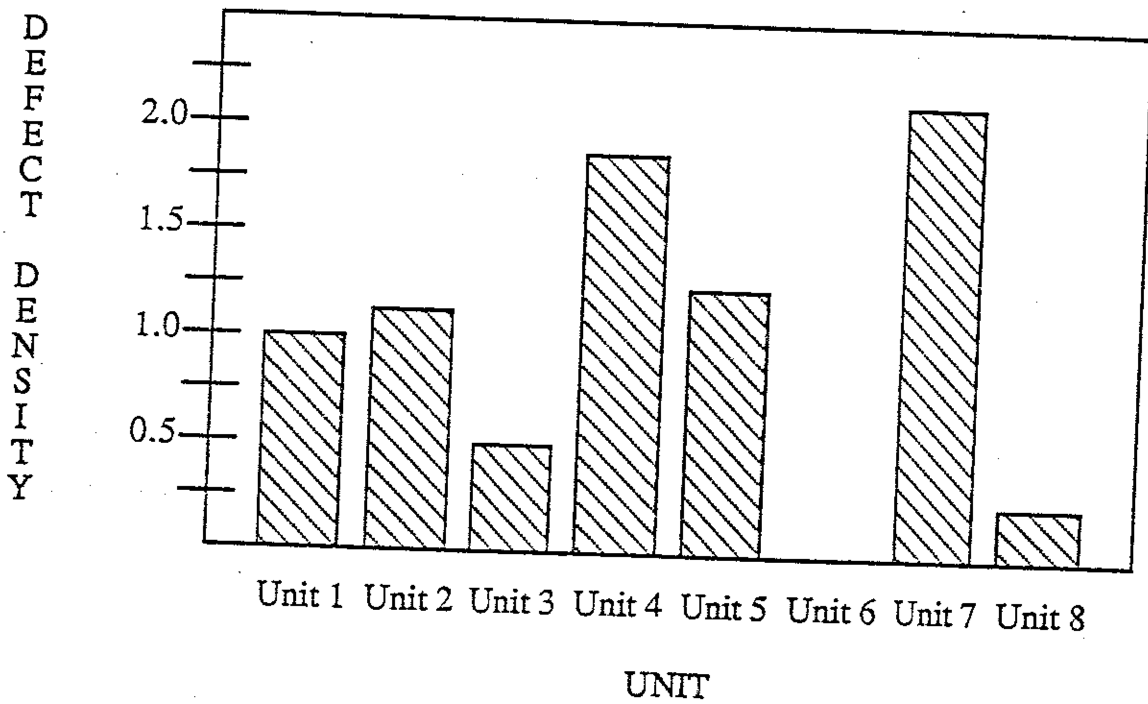

Figure 13. Defect Density

by rapidly increasing Defect Densities, those units within a software system which are soon to become defect-prone, but which had not been identified earlier.

Defect Densities are not generally compared across software systems because of different system characteristics. However, empirical data exists on defect densities for different types of software systems. By comparison with empirical data of similar size and type systems, defects remaining can be predicted and reliability ratings may be estimated.

Calculations for this indicator can also be taken by severity, if necessary for a more accurate representation. For example, only those defects over a certain level of severity can be used in the calculation of the Defect Densities of the individual units.

F. References

[IEEE85]/Appendix B,pp.1-8

[MACJ86]/pp.57-59

Defect Detection Efficiency

A. Description

The Defect Detection Efficiency Indicator is used to determine how efficiently defects in the software product are being detected. Implementation of this indicator is accomplished by tracking the percentage of defects detected in a phase of the software development life cycle that actually could have been detected in that phase. A defect is first detectable in the phase in which it is introduced. This indicator can be used in determining the reliability of the product.

B. Application Phase

The Defect Detection Efficiency Indicator can be used during product acquisition and during deployment. It is a Non-affective Indicator.

C. Implementation

The following example is provided to define this indicator. One hundred defects were detected during the requirements analysis review of a software product. Because no other defects were known at that time, Defect Detection Efficiency of the requirements analysis review was 100 percent. During the specifications review, 20 defects were found which were introduced during requirements analysis, and thus, could have been detected during the requirements analysis review. The Defect Detection Efficiency of the requirements analysis review decreases to 100 out of 120 total detected, or 83% of the defects detected during the requirements analysis review which could have been detected at that time. Calculation continues in this manner throughout the life of the product.

A line chart is plotted to view the results. See Figure 14 on page 90 for an example. The reviews, inspections, and testing phases are plotted along the horizontal axis. The Defect Detection Efficiency values are plotted on the vertical axis. The reviews, inspections, testing phases, and the Operations and Maintenance phase of the software development life cycle are labeled along the horizontal axis, up to the current stage in the life cycle. The percentages calculated are plotted against the vertical axis. A different graph of Defect Detection Efficiency is plotted for the defects which could have been detected during each review, inspection, or testing phase.

D. Interpretation

Defects which remain in a software product across more phases of the life cycle are likely to manifest themselves by causing more defects in the system. This fact provides the motivation for the Defect Detection Efficiency Indicator. The implementation of this indicator deals with the determination of the phases in which defects are introduced and detected.

Reliability:

A reliability problem is evidenced by a steady decrease over two or more reviews or by a sharp decrease in any period. The sharper the decrease the more serious the problem because undetected defects are remaining in the software for longer lengths of time. For instance, if in the previous example, 40 defects instead of 20 had been found in the detailed design inspection, the Defect Detection Efficiency decreases to 100/140 or 71%. A greater percentage of the defects have remained in the system longer. The longer a defect remains the greater impact it could have on the system. While Figure 14 shows a satisfactory Defect Detection Efficiency, Figure 15 on page 91 shows a potentially more serious reliability problem because more defects are being detected in subsequent phases.

E. Additional Comments

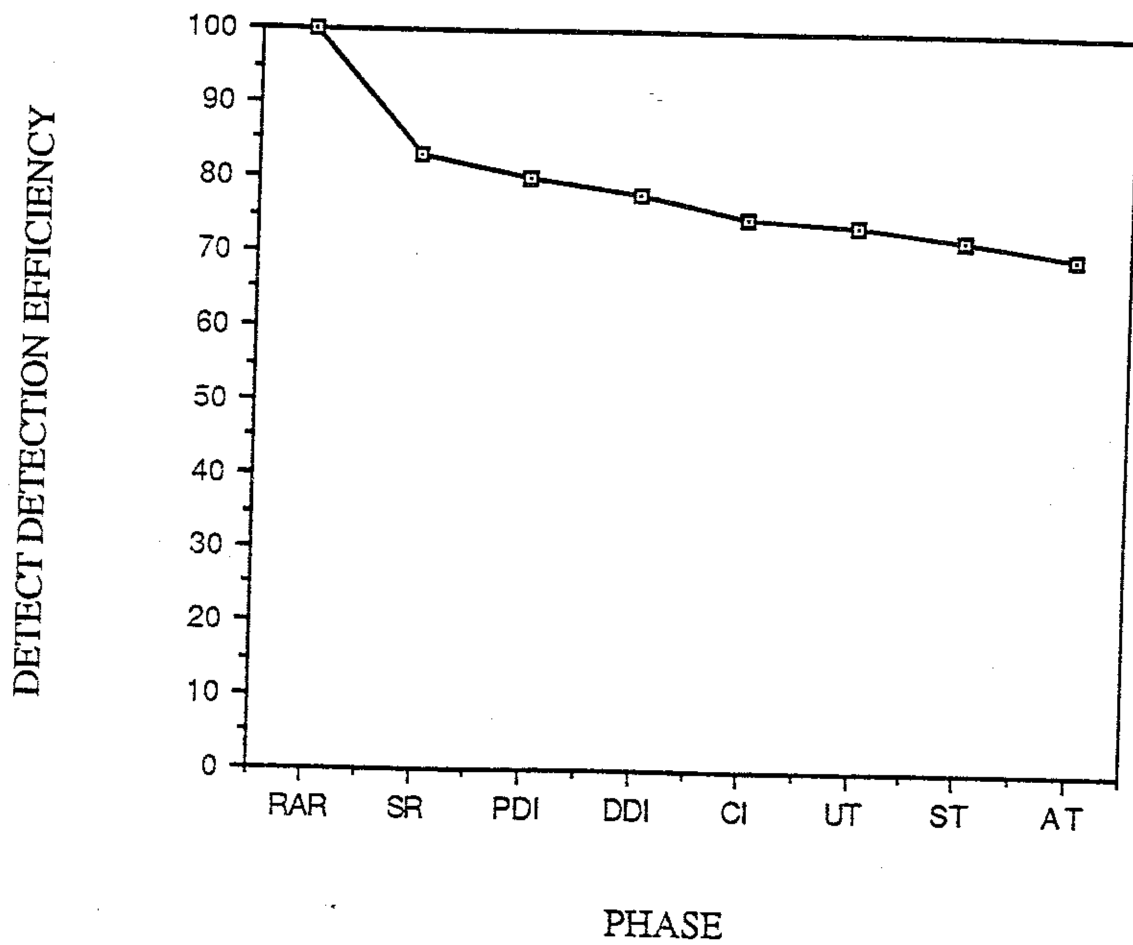

Figure 14. Defect Detection Efficiency

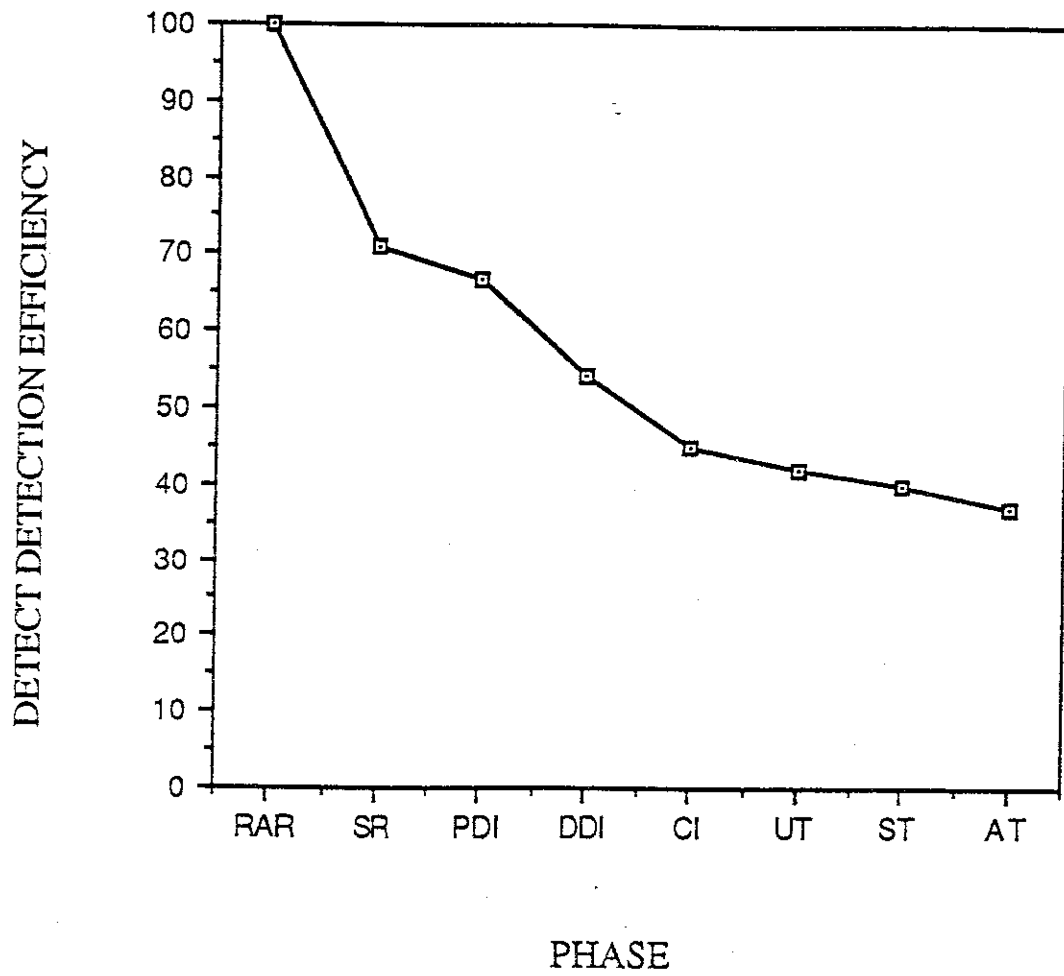

Figure 15. Defect Detection Efficiency: Unsatisfactory

Calculations for the Defect Detection Efficiency Indicator can also be taken by severity and/or structural level, if necessary for a more accurate representation.

This indicator could possibly be extended by determining the area under the Defect Detection Efficiency curve to provide a reliability rating.

F. References

[MACJ86]/pp.44-47

Extent of Modification

A. Description

The Extent of Modification Indicator is used to determine the extent of the software that is modified as a result of resolving an STR. Implementation of this indicator is accomplished by determining the percentage of units modified during the maintenance activity necessary to close an STR. This indicator can be used in determining the reliability and/or maintainability of the product.

B. Application Phase

The Extent of Modification Indicator can be used during deployment. It is not used during product acquisition. It is an Affective Indicator. Section 2.3.6. discusses how an STR is compiled for use during deployment and how STRs and defects are related.

C. Implementation

The Extent of Modification Indicator is calculated after each maintenance activity, i.e. the maintenance required to close one open STR. The calculation utilizes units within the software structure. The following formula is used:

$$\text{Extent of Modification} = \frac{\text{number of units modified}}{\text{total units in system}}$$

The number of software units modified is the sum of the units added, modified, or deleted during the maintenance activity. The total number of software units is taken to be the number of units before the maintenance activity.

D. Interpretation

Reviewing the Extent of Modification Indicator can give an indication of the modularity of the system, or the independence of the software units. A large value may indicate a problem in the reliability and/or maintainability of the software product.

Reliability:

If the value is large, reliability may be suspect. A large percentage of the software units modified may indicate that the software product is structured inadequately. A large number implies that more units are undergoing modification than if the units were designed with greater independence. By modifying a large number of units, the possibility of causing more defects to a greater portion of the product arises, because modification of a software product typically increases the risk of defects in the units modified.

Maintainability:

If the value is large, maintainability may also be suspect. A large percentage of the software units modified may indicate that the software product is inadequately structured, i.e. the software units are interdependent upon each other, thus, the modularity of the software is poor. Modularity which is inadequate causes difficulty in the understandability of the software product, decreasing the maintainability of the product.

E. Additional Comments

The Extent of Modification Indicator can also be utilized if maintenance to a software product is performed in a fashion where defects are collected and modification is conducted on a periodic basis (e.g. every six months or year). The indicator is calculated each time period (or, if a separate maintenance activity is performed for every open STR, the calculation can be delayed until the end

of the time period). The indicator can then be extended by implementing a line chart of the results. Time is plotted along the horizontal axis. The values of the indicator are plotted along the vertical axis. See Figure 16 on page 96 an example. During the first year of the Operations and Maintenance Phase, 80 out of 100 total units were modified for a value of 0.8. During the second year, 75 out of 100 units were modified, resulting in a value of 0.75. Plotting continues in this manner throughout the life of the product. The plotted line should decrease with time, as in Figure 16, indicating that the software product is stabilizing, i.e. less of the product has to be modified. An increase would indicate that defects are being found in a larger portion of the product. Thus, the defects are becoming more dispersed causing the reliability and maintainability to decrease.

F. References

[AFSC87]/p.9

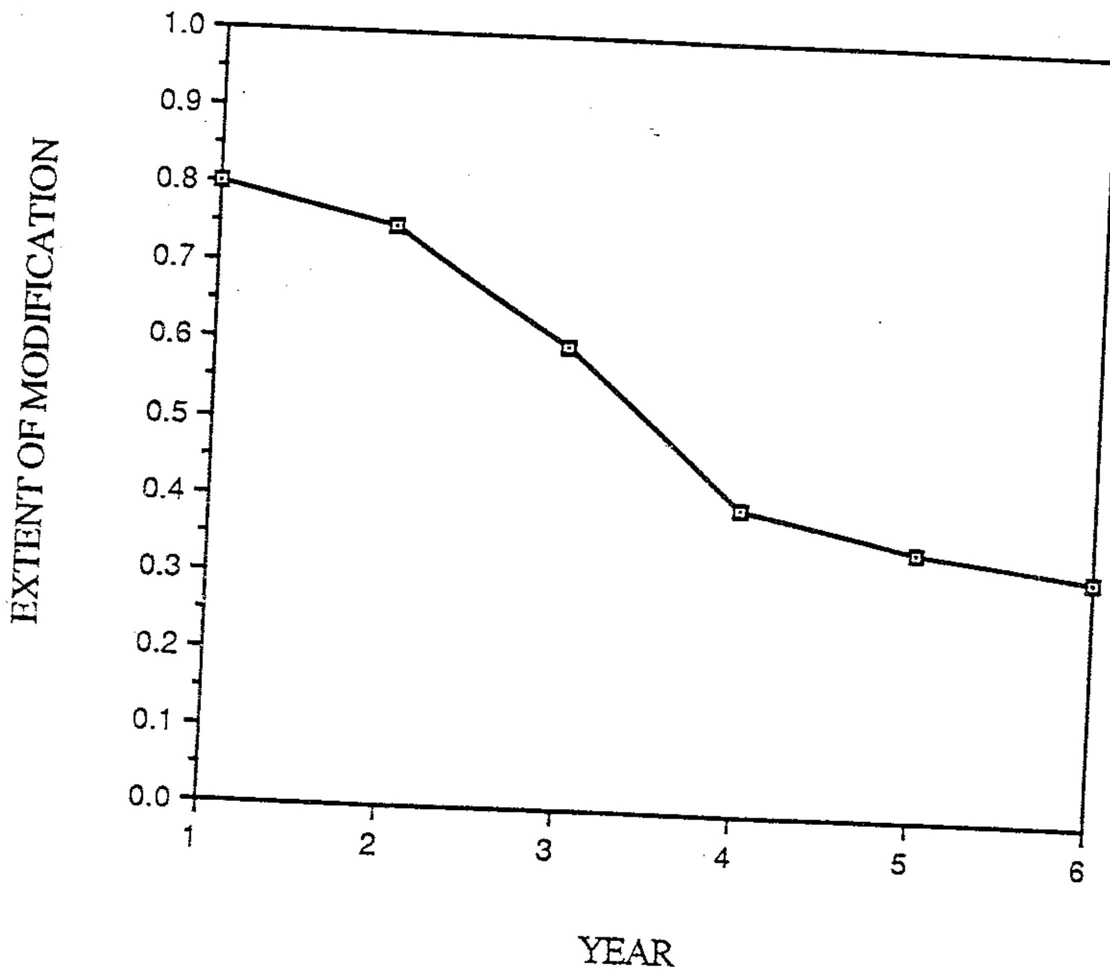

Figure 16. Extent of Modification

Defect Introduction

A. Description

The Defect Introduction Indicator is used to determine the amount of time the defects discovered during execution have remained in the software product. Implementation of this indicator is accomplished by determining the percentage of defects detected during the operation of the product that were introduced in each phase of the software life cycle. This indicator can be used in determining the reliability and/or maintainability of the product.

B. Application Phase

The Defect Introduction Indicator can be used during deployment. It is not used during product acquisition. It is a Non-affective Indicator.

C. Implementation

To implement the Defect Introduction Indicator, for every defect detected during execution of the software product, the phase the defect was introduced is determined. A percentage for each phase is calculated as follows:

$$\text{Percent Introduced in Phase} = \frac{\text{number of defects introduced in phase}}{\text{total defects detected}}$$

This indicator must be implemented after significant of modification has been performed on the product for an accurate representation.

D. Interpretation

Depending upon the distribution of percentages, the Defect Introduction Indicator can indicate the reliability and/or maintainability of the software product.

Reliability:

Higher percentages for earlier phases indicate defects have been present in the software product across more phases of the life cycle. The more phases the defects have survived in the product, the greater the opportunity to manifest themselves and cause other defects in the product.

Maintainability:

If the majority of the defects detected during operation are introduced during the Operations and Maintenance phase, maintainability may be a problem. This may indicate that more and more defects are being introduced into the software product as a result of performing modification to the product.

E. Additional Comments

By only using those defects above a certain severity, a more accurate representation of the Defect Introduction Indicator may be produced.

This indicator can be calculated by structural level (by units) for an indication of the reliability and maintainability of the individual units.

F. References

[AFSC87]/p.11

Defect Days

A. Description

The Defect Days Indicator is used to determine the length of time defects are remaining in the software product. Implementation of this indicator is accomplished by determining the number of days that each defect remains in the product, that is, the date the defect is introduced until the date it is removed. This indicator can be used in determining the reliability and/or maintainability of the product.

B. Application Phase

The Defect Days Indicator can be utilized during product acquisition and during deployment. It is a Non-affective Indicator.

C. Implementation

It is possible for two software products that were developed for the same set of specifications to be tested to the same acceptance conditions, that is, pass the required test cases. It is highly unlikely, however, for them to have the same defect histories.

The defect days for each defect removed from the software is determined by the number of days between the introduction of that defect and its removal. Unknown defect introduction dates are taken as the middle of the date of the phase of introduction. The Defect Days Indicator is calculated by the summation of the defect days of each individual defect.

D. Interpretation

A high value of the Defect Days Indicator indicates:

1. the presence of many defects in the product, i.e. a poor design (product), and/or

2. defects which have remained in the product a long time, i.e. a poor development effort (process).

This can be determined by examining the individual defect days more closely.

Suppose that the two software products in Table 8 on page 101 [MILH76,p.272] are produced from the same specifications and pass the same acceptance test cases. No known defects exist at the time of product acquisition. As can be seen, the defect histories of the two products are very different. System B has a higher total Defect Days value. There is a greater chance of more defects being detected in the future. Consequently during acceptance testing more defects were detected. Most likely, more defects will be detected during execution of system B.

Reliability:

The Defect Days Indicator may indicate the reliability of the software. Defects which have remained in the software a long time have more time to manifest themselves by causing other defects to be incorporated in the software product or by contributing to a number of failures. A product with a history of known defects, especially having many which have remained in the system for a long period of time, is more likely to experience a greater number of future defects.

Maintainability:

If many defects are detected in a software product or a product has a history of long-lived defects then the maintainability of that product is questionable. Defects may be constantly being intro-

Table 8. Defect Days

	System A		System B	
Before Acceptance Testing		Defect-Days		Defect-Days
Lines of Code	50,000		50,000	
Defects fixed				
day old	100	100	500	500
week old	10	50	50	250
month old	5	100	50	1,000
year old	5	1,250	20	5,000
Known Defects	0		0	
Defect-Days (Total)		1,500		6,750
During Acceptance Testing				
Defects Fixed	10		50	
Known Defects	0		0	

duced or are remaining a long period of time because of the difficulty to understand the system or the poor system design.

E. Additional Comments

By only using those defects above a certain severity, a more accurate representation of the Defect Days Indicator may be produced.

This indicator can be calculated by structural level (by units) for an indication of the reliability and maintainability of the individual units.

F. References

[MILH76]/pp.271-272

[IEEE85]/Appendix B,pp.13-16

Defect Index

A. Description

The Defect Index Indicator provides a continuing measure of reliability as the software product advances through the software life cycle. Implementation of this indicator is accomplished by a calculation weighted in terms of the severity and the phase in which defects of the software product are detected. This indicator can be used in determining the reliability of the product.

B. Application Phase

The Defect Index Indicator can be applied during product acquisition and during deployment. It is a Non-affective Indicator.

C. Implementation

The Defect Index Indicator is continually calculated throughout the Software Life Cycle. It can be calculated at any point during the product's life. Severities are weighted as to the impact on the user. The following weighting factors are suggested:

- W_L - low severity weighting (default 1),
- W_M - medium severity weighting (default 3),
- W_H - high severity weighting (default 10),

The following reliability factor, R_i is associated with each phase:

$$R_i = W_L \times \frac{L_i}{N_i} + W_M \times \frac{M_i}{N_i} + W_H \times \frac{H_i}{N_i}$$

$$= \frac{W_L \times L_i + W_M \times M_i + W_H \times H_i}{N_i}$$

where, for each phase i:

- L_i = number of low severity defects detected in phase,
- M_i = number of medium severity defects detected in phase,
- H_i = number of high severity defects detected in phase, and
- N_i = total defects detected in phase.

The Defect Index Indicator is calculated at each phase, n, of the life cycle by the following formula:

$$\text{Defect Index} = \sum_{i=1}^{n} i \times \frac{R_i}{\text{KSLOC}}$$

$$= \frac{R_1 + 2 \times R_2 + 3 \times R_3 + \cdots}{\text{KSLOC}}$$

where:

- n is the current software life cycle phase,
- each phase further into the life cycle has a heavier weighting (i.e. 1, 2, 3, 4, ..., n), because the later a defect is found the more costly is could be to fix and the more likely it is to cause other defects, and

- KSLOC is a normalization factor (see Defect Density); SLOC could also be used depending on the size of the software product.

D. Interpretation

The value of the Defect Index Indicator should be compared against Defect Indexes of other software products of similar application and size.

Reliability:

As a general rule, the smaller the value the more reliable the software product is likely to be, with certain exceptions. A poor inspection process may be indicated by few defects being detected during a review. If the reviews processes are found to be insufficient, reliability may be a problem because defects which should have been detected during these inspection processes have most likely remained in the product.

E. Additional Comments

Severities can also be weighted in other ways. A scale of 1 to 10 can be used, with 1 being very trivial and 10 being a failure causing loss of system performance.

Phases can also be weighted in other ways. The phases can also be divided to include subphases. For example, testing can be broken down into unit, system, and acceptance testing, and be given different weightings.

F. References

[IEEE85]/Appendix B,pp.33-37

Defect Distributions

A. Description

The Defect Distributions Indicator attempts to assist management in performing preventive actions to the software product. Implementation of this indicator is accomplished by calculating distributions of defects in different categories such as the cause of the defect. This indicator can be used in determining the reliability of the product.

B. Application Phase

The Defect Distributions Indicator can be used during product acquisition and during deployment. It is a Non-affective Indicator.

C. Implementation

To implement the Defect Distributions Indicator, defects are classified as to the cause of the defect and other categories such as cause for detection deferral. The defects of each class are counted and plotted in a vertical bar chart, as in Figure 17 on page 107. The classifications are labeled along the horizontal axis, while the number belonging to each classification is plotted along the vertical axis.

D. Interpretation

By examining the plots of the Defect Distributions Indicator the most frequent causes of defects, reasons for defect deferral, and other problems can be identified.

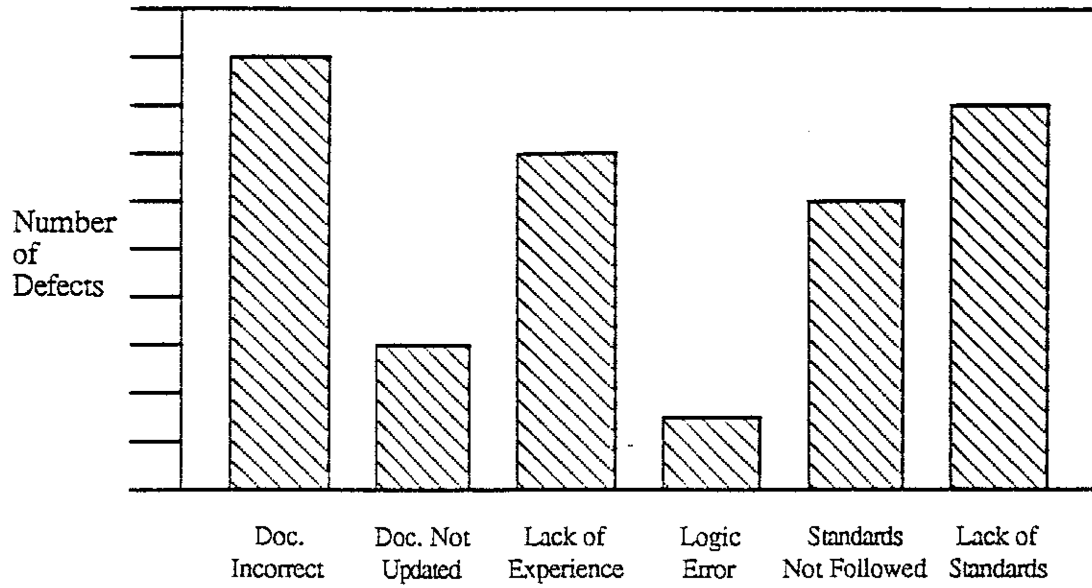

FIGURE A. Defect Distribution by Defect Cause

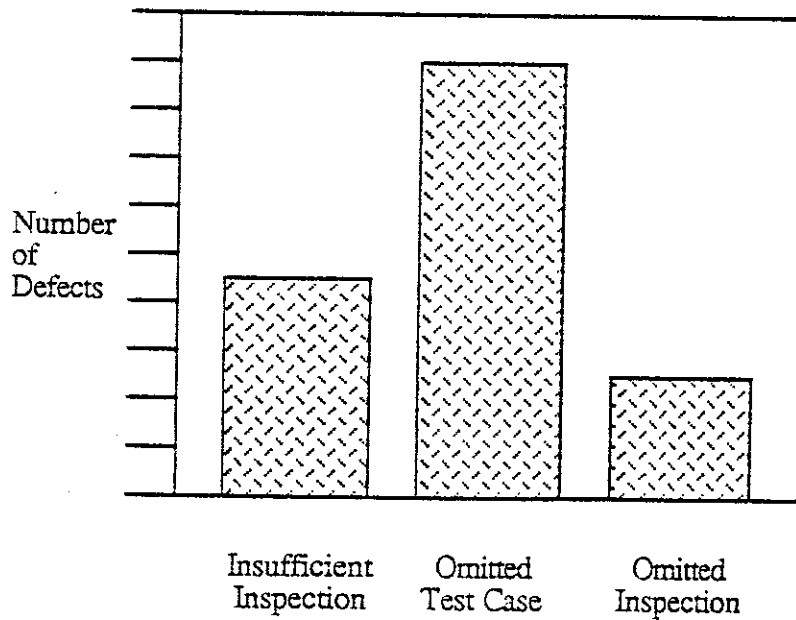

FIGURE B. Distribution of Defects by Cause for Detection Deferral

Figure 17. Defect Distributions

Reliability:

The results of this examination may provide insight into the most unreliable aspects of the software product. The Defect Distributions Indicator provides a means for accounting for the defects. Thus, it may signify corrective measures which need to be taken in order to improve the product and review procedures, or ways of avoiding reoccurrence of the same types of defects in the future.

E. Additional Comments

The use of the Defect Distributions Indicator during deployment can be limited to the defects detected during execution if amends have been made for the results discovered during product acquisition.

F. References

[IEEE85]/Appendix B,pp.38-41

Maturity Index

A. Description

The Maturity Index Indicator is used to determine the maturity of a software product at each delivery, or on a periodic basis. Implementation of this indicator is accomplished by taking into account a stability measure of the program. This indicator can be used in determining the reliability of the product.

B. Application Phase

The Maturity Index Indicator can be used during deployment. It cannot be implemented during product acquisition. It is an Affective Indicator.

C. Implementation

The following formula is used to calculate the Maturity Index Indicator:

$$\text{Maturity Index} = \frac{M_T - (F_a + F_c + F_{del})}{M_T}$$

where:

- M_T =number of functions (modules) in the current delivery,
- F_a =number of functions (modules) in the current delivery which were added since the previous delivery,

- F_c =number of functions (modules) in the current delivery which were changed from the previous delivery, and

- F_{del} =number of functions (modules) in the previous delivery which were deleted in the current delivery.

The factor $(F_a + F_c + F_{del})$ is the total of all functions (modules) that were added, changed, or deleted in the current delivery. All are treated the same because all are functions (modules) in which modifications have been made for the current delivery.

To determine if the software product is approaching a stable configuration, the Maturity Index is plotted for each release, or designated time period. See Figure 18 on page 111 for an example. The deliveries, or periods of time, are labeled along the horizontal axis. The values of the Maturity Indicator are plotted along the vertical axis. The first release consisted of 100 total functions, of which 10 were added, 35 were changed, and 5 were deleted for a total of 50 functions in which modifications had been performed. These values result in a Maturity Index of 0.5. Plotting continues in this manner for each subsequent release, or time period.

D. Interpretation

By subtracting the total number of functions (modules) in which modifications have been made in this formula, a value of 1 or less is obtained for the value of the Maturity Index Indicator.

Reliability:

A reliable software product should stabilize, that is, less of the program should have to be modified as a result of recognized inadequacies of the product for each successive release or time period. The plotted line should increase and flatten off towards a value of 1, as seen in Figure 18.

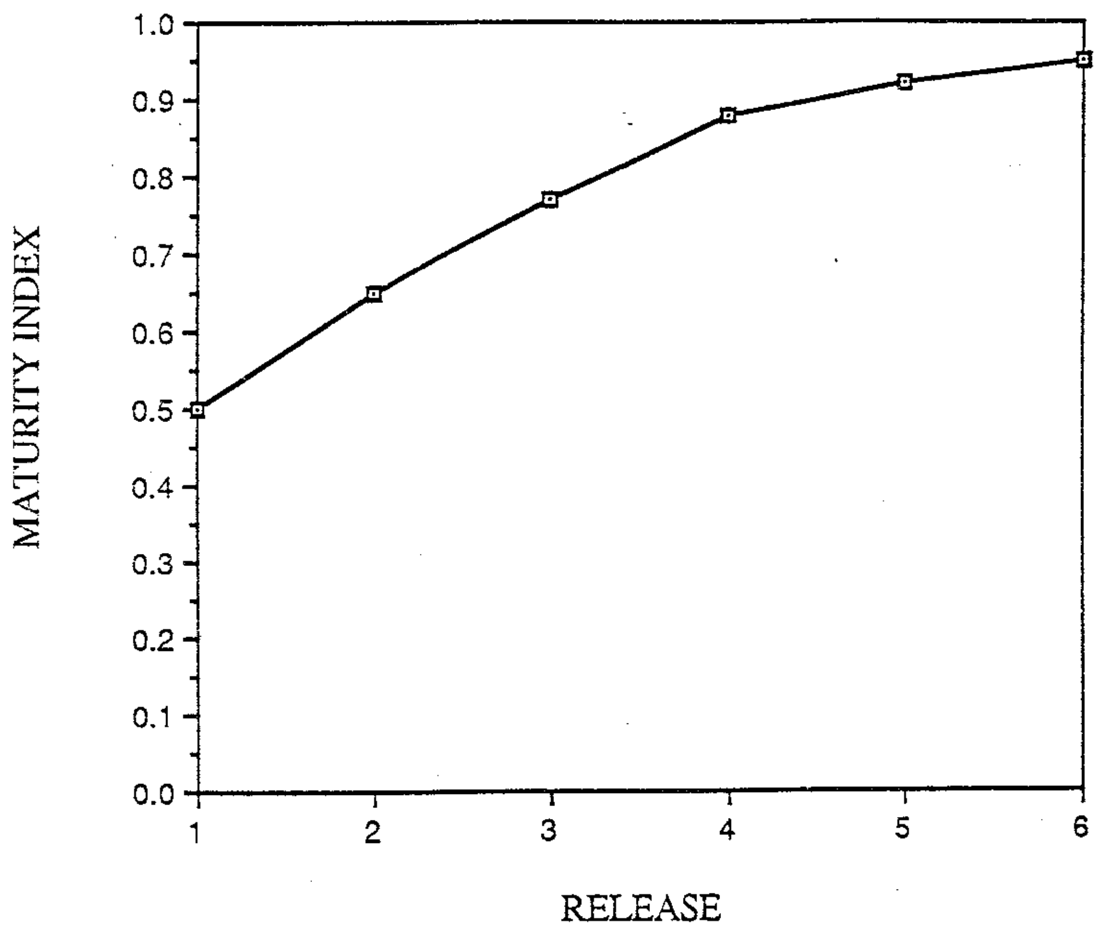

Figure 18. Maturity Index

A negative slope of the plotted line reveals that more of the program had to be modified for that release or time period than for the previous time frame. Thus, the recognized inadequacies of the system covered a greater portion of the program.

E. Additional Comments

A number of the modifications may have been the result of enhancements to the product, changing of the environment, etc. For a more accurate representation, the values of the Maturity Index Indicator may be calculated for those modifications which were the result of defects in the product.

F. References

[IEEE85]/Appendix B,pp.42-46

Person-Hours per Major Defect Detected

A. Description

The Person-Hours per Major Defect Detected Indicator is used to determine if the design and code inspections were effective in detecting defects in the software product. Implementation of this indicator is accomplished by determining the length of time required to detect defects in the product during the inspections. This indicator can be used in determining the reliability and/or maintainability of the product.

B. Application Phase

The Person-Hours per Major Defect Detected Indicator can be used during product acquisition. It is not used during deployment.

C. Implementation

The Person-Hours per Major Defect Detected Indicator determines the length of time that was necessary to detect defects during design and code inspections. The indicator is calculated as follows:

$$\text{Person-Hours per Major Defect Detected} = \frac{\text{total person-hours on design or code inspection}}{\text{number of major (non-trivial) defects detected}}$$

The result is the average length of time needed to detect a defect in the product during these inspections. The total person-hours on design or code inspection includes the time spent in the

meetings for each person, as well as the preparation time for the meetings. Major defects should include all defects except those whose severity is determined as low.

D. Interpretation

The design and code inspection processes provide a valuable method for the early detection of defects. Reviewing the results of these processes with the use of the Person-Hours per Major Defect Detected Indicator, can give an indication of the effectiveness of the processes. Empirical evidence suggests that values less than 3 or greater than 5 may indicate a problem in the reliability and/or the maintainability of the software product.

Reliability:

If the value is large, reliability may be suspect. Excessive time to locate a defect in the design or code may indicate the inspection processes are poorly implemented. If the processes are poor, many defects which should have been detected during inspection may still remain in the software product.

A small value may also indicate a reliability problem. If the value is inordinately small, many defects are being detected in the software product at a high rate. Detecting a large number of defects rapidly can indicate the product is dense with defects, and many may still remain to be detected.

Maintainability:

If the value is large, maintainability may be suspect. Excessive time to locate a defect may indicate the difficulty in understanding the design or code, resulting in a software product which is difficult to maintain.

E. Additional Comments

Calculations for the Person-Hours per Major Defect Detected Indicator can also be compiled for each unit, to gain insight into the reliability and maintainability of the individual units.

Additional information may be gained by looking at the numerical trend provided by the computation of this indicator based on individual inspections.

F. References

[IEEE85]/Appendix B,pp.47-50

Person-Hours per Source Statement Modified

A. Description

The Person-Hours per Source Statement Modified Indicator is used to determine the amount of effort required to maintain the software product. Implementation of this indicator is accomplished by determining the length of time which has been necessary to modify a source statement of the product. This indicator can be used in determining the maintainability of the product.

B. Application Phase

The Person-Hours per Source Statement Modified Indicator can be used during deployment. It is not used during product acquisition. It is an Affective Indicator.

C. Implementation

The Person-Hours per Source Statement Modified Indicator determines the average length of time necessary to modify a line of source code. The indicator is calculated after each maintenance activity as follows:

$$\text{Person-hours per Source Statement Modified} = \frac{\text{person-hours spent on modification}}{\text{lines of code modified}}$$

The lines of code modified is the sum of the lines added, changed, and deleted during the maintainability activity. This modification can include all types of maintenance, not just the corrective maintenance which is in response to defects detected in the code.

D. Interpretation

Reviewing the results of the maintenance activities with the use of the Person-Hours per Source Statement Modified Indicator, can give an indication of the effort (i.e. the length of time) necessary to maintain a relative amount of code An inordinately large value may indicate a problem in the maintainability of the software product.

Maintainability:

If the value is inordinately large, maintainability may be suspect. Excessive time to modify a line of source code may indicate the difficulty in understanding the code, resulting in a software product which is difficult to maintain. The product may be difficult to understand and to modify because of its structure. Modifying a difficult to maintain software product usually decreases its maintainability even more because many times a product is patched with the intent to resolve defects with no concern for the future Maintainability of the product.

E. Additional Comments

Calculations for the Person-Hours per Source Statement Modified Indicator can also be determined for each unit, to gain insight into the maintainability of the individual units.

The time necessary to perform a maintenance activity can be estimated by using the Person-Hours per Source Statement Modified Indicator if the number of source statement necessary to correct the inadequacy can be estimated.

F. References

[SWAE76]/p.495

Requirements Traceability

A. Description

The Requirements Traceability Indicator assists in determining if there are missing or additional requirements met by the software product. Implementation of this indicator is accomplished by the tracing of the requirements through the product. This indicator can be used in determining the reliability and/or maintainability of the product.

B. Application Phase

The Requirements Traceability Indicator can be used during product acquisition and during deployment. It is an Affective Indicator.

C. Implementation

The original requirements are traced to see if they are met by requirements found within the software product. Stepwise refinement of the requirements during development or maintenance provides the mappings necessary to trace the requirements. Requirements met by the architecture, R1, and original requirements, R2, are counted. The following formula is applied:

$$\text{Requirements Traceability} = \frac{R1}{R2} \times 100\%$$

D. Interpretation

As a general rule the following occurs:

- If there are no missing or additional requirements, the value will be 100%.

- If all original requirements are not included within the software, the value will be less than 100%.

- If the software contains requirements beyond those of the original requirements, i.e. an original requirement was taken out of the original requirements but was not deleted from the software product, the value will be greater than 100%.

The exception occurs when the second and third conditions occur, i.e. there are missing and additional requirements. The results of tracing the requirements should be examined carefully for this reason.

Reliability:

Obviously, if requirements are missing from the product the reliability of the product will suffer. The reliability may also be suspect if there are additional requirements. If this is the case, there should be concern as to how well the process was carried out.

Maintainability:

Missing or additional requirements, if not fixed, can cause the software product to be very difficult to understand. A product which is difficult to understand is hard to maintain.

E. Additional Comments

When deviations from the requirements are found, they should be corrected before proceeding.

F. References

[AFSC87]/pp.5-8

[IEEE85]/Appendix B,pp.30-32

Test Coverage

A. Description

The Test Coverage Indicator is used to measure the completeness, or thoroughness, of the testing performed on the software product. It can also be used to determine the readiness for advancement to the next level of testing. Implementation of this indicator is accomplished by assessing the thoroughness of the testing by considering both a user and a developer perspective of the testing. This indicator can be used in determining the reliability of the product.

B. Application Phase

The Test Coverage Indicator can be used during product acquisition and during deployment. It is an Affective Indicator.

C. Implementation

The Test Coverage Indicator determines the amount of software tested by taking an intersection of two testing categories which provides both a user and a developer perspective of the testing.

The first category takes a user perspective by determining the percentage of capabilities (requirements or functions) tested. These are system level capabilities required to perform to the users' expectations. Defined test cases are usually set up for testing these capabilities.

The second category takes a developer perspective by determining the percentage of software structures tested. Software structures may be taken as statements, branches, paths, segments, units, or modules. The more refined the software structure the more complete the testing. For example,

100% of statements tested is more complete than 100% of units tested. The software structure is chosen depending on the test level as depth of testing required. Determining the percentage of software structure tested is aided by instrumenting the code, inserting probes, to monitor the software structure coverage.

The indicator is calculated as follows:

$$\text{Test Coverage} = \frac{\text{number of implemented capabilities tested}}{\text{total required capabilities}} \times \frac{\text{software structure tested}}{\text{total software structure}} \times 100\%$$

D. Interpretation

The Test Coverage Indicator provides the level of thoroughness of exercising the software. The higher the value of this indicator, i.e.the closer it is to 100%, along with a high level of software structure included in the calculation, the more likely the defects in the software product have been detected the testing process.

Reliability:

If the value is low, reliability may be suspect. Although a high value does not ensure the defects were located, it indicates more of the defects have been detected in the software product than if the value were lower.

E. Additional Comments

Goals for the Test Coverage Indicator can be set for the conclusion of each level of testing (unit, system, and acceptance) which are agreed upon before the software product is developed. Achievement of these values must be met before advancing to the next level of testing or concluding testing. This can be tracked by implementing a line chart for each level of testing. See Figure 19 on page 124 for an example. Time is plotted on the horizontal axis. The value of the Test Cov-

erage Indicator is plotted on the vertical axis. A horizontal reference line is plotted in accordance with the set goal. Before that level of testing is complete, the plotted line must meet or surpass the reference line. The goal for the value of the Test Coverage Indicator in Figure 19 is 0.9. This requirement is met after the twelfth month of testing. The presence of a decreasing slope in the plotted line indicates that defects have been found causing retest of certain capabilities and software structure. This condition is more serious the steeper the decrease and the later it occurs, when there is less time to perform the retest.

The value of this indicator may be low because of a restricted test environment. The test environment may not provide all the necessary conditions to fully test the system. In such cases the value of this indicator may need to be compared against values calculated from systems of a similar application and size.

F. References

[AFSC87]/pp.12-14

[IEEE85]/Appendix B,pp.107-111

[MACJ86]/pp.41-44

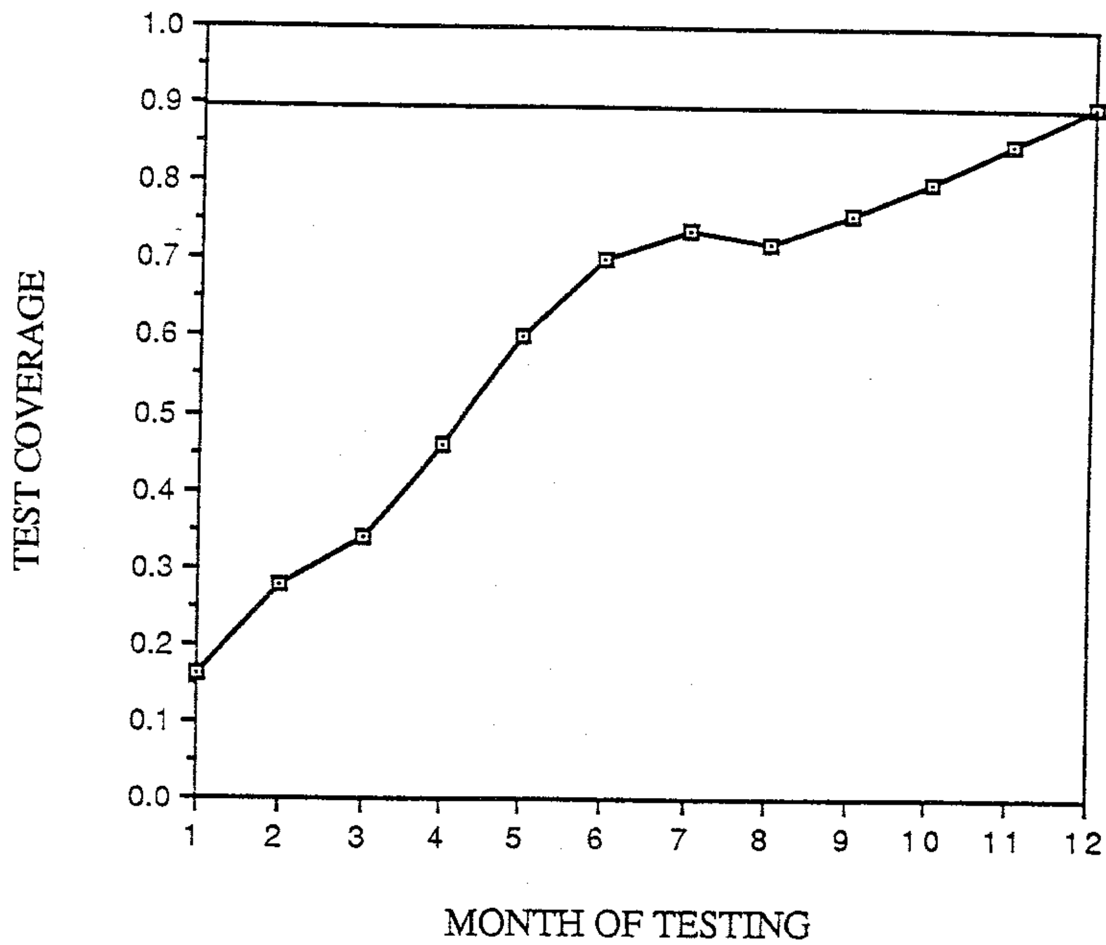

Figure 19. Test Coverage

Robustness

A. Description

The Robustness Indicator is used to determine if severe faults are occurring too frequently. Implementation of this indicator is accomplished by examining the number of faults at each severity level that have occurred. This indicator can be used in determining the reliability of the product.

B. Application Phase

The Robustness Indicator can be used during product acquisition and during deployment. It is a Non-affective Indicator.

C. Implementation

A software product is said to be robust if the following relationship holds:

$$C \propto \frac{1}{p}$$

where:

- C is the cost (severity) to the user of the fault occurring, and
- p is the probability of that type of fault occurring.

To determine if this relationship holds, data must be collected on the faults, their frequency of occurrence and their severity.

D. Interpretation

In order for the software product to be considered robust, faults occurring frequently should be the least severe faults, and the most severe faults should occur infrequently.

Reliability:

The software product is not considered to be reliable if severe faults are occurring too frequently as compared to the number of less severe faults which are occurring. Thus, the number of faults which have occurred in each severity level must be examined. The more severe the fault level, the less frequently the faults at this level should have occurred.

E. Additional Comments

The Robustness Indicator can be implemented on any scale in which the severities are relative. That is, the scale could be expressed as a high, medium, low scale, or as a point scale.

F. References

[KOPH79]/pp.9-10

Mean Time to Failure

A. Description

The Mean Time to Failure (MTTF) Indicator is used to determine the length of time the program executes after execution begins until a failure caused by a program defect occurs. Implementation of this indicator is accomplished by keeping track of the number of times that the system fails as a result of a program defect relative to its total execution time. This indicator can be used in determining the reliability of the product.

B. Application Phase

The Mean Time to Failure Indicator can be used during deployment. It is not used during product acquisition. It is a Non-affective Indicator.

C. Implementation

The state of a software product during the Operations and Maintenance phase follows a continuous cycle. Typically, the execution is initiated, then the program operates for a period of time until execution terminates abnormally. This termination is sometimes the result of encountering a serious defect in the program. The system is then inoperable for a period of time, until it can be restarted. If the termination is caused by a program failure, a defect usually requires immediate correction. The program is usually restarted after the serious defect which caused the failure has been repaired. The cycle then continues (i.e. after a period of time, execution will again halt and once again be restarted). The Mean Time to Failure, or the average time the program executes until a failure caused by a defect in the program occurs, is computed as follows:

$$\text{Mean Time to Failure (MTTF)} = \frac{\text{total execution time}}{\text{total number of failures}}$$

This indicator is calculated on a periodic basis (e.g. every month). A line chart is plotted against time. See Figure 20 on page 129 for an example. Time is plotted along the horizontal axis. The MTTF value is plotted along the vertical axis. During the first month, 150 failures occurred which were the result of defects in the program. The system executes a total of 300 hours. These values result in a MTTF of 0.5 hours. Plotting continues in this manner throughout deployment.

D. Interpretation

The MTTF Indicator determines the average length of time the program will execute before it abnormally terminates as a result of a defect in the program. Program failures have a tremendous impact on the reliability of the system.

Reliability:

A decreasing or steady plotted line indicates that there may exist a reliability problem. The plotted line is expected to increase, as in Figure 20, indicating that a longer length of time is passing between failures for each successive period. Thus, defects which are serious enough to cause failures are presumably being reduced in the system.

E. Additional Comments

A more accurate representation of reliability may be obtained by calculating Mean Time to Fault, that is, performing the calculation utilizing faults instead of failures. Faults, especially those that are more severe, also have an impact on the reliability of the software.

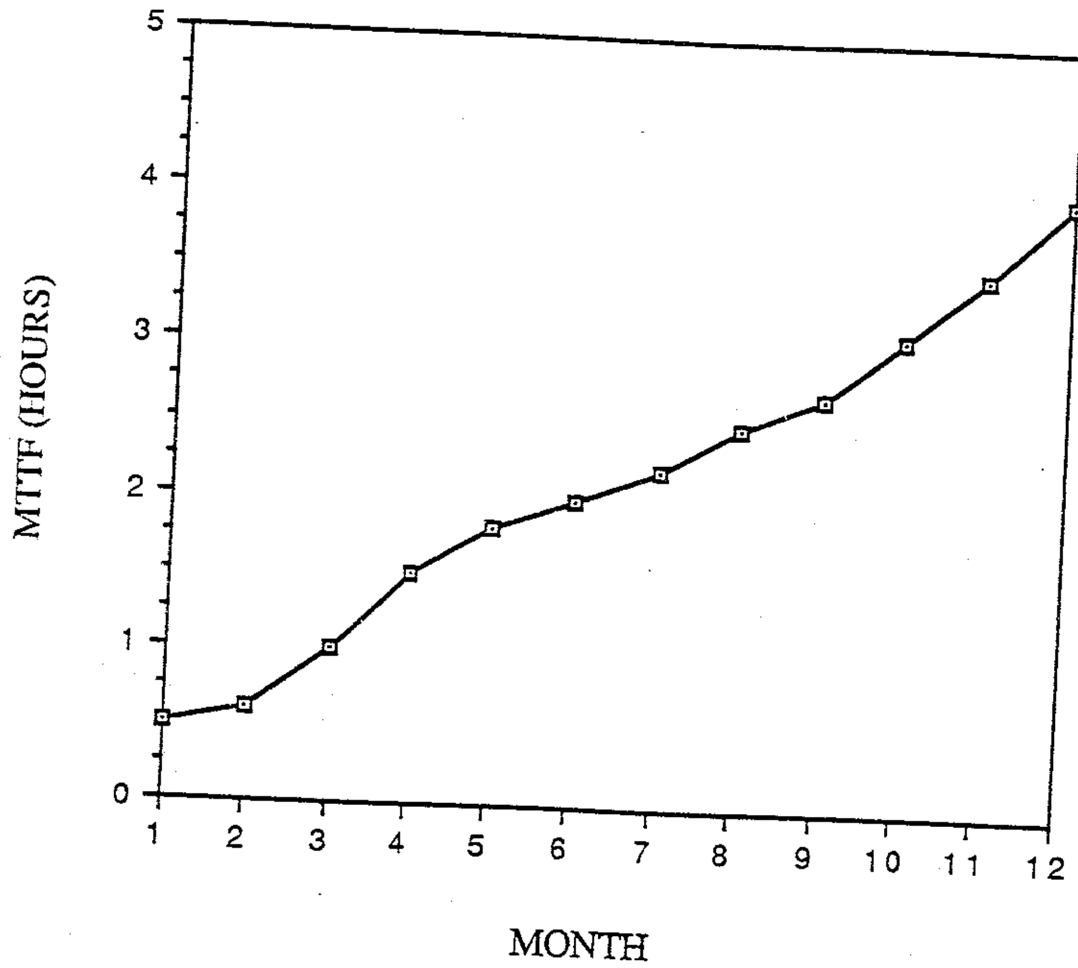

Figure 20. Mean Time to Failure (MTTF)

Distributions of the Time to Failure have been proposed as more helpful than an average as given by MTTF [LITB78].

This indicator has also been calculated using calendar time instead of execution time. The use of calendar time provides no allowance if the system abnormally terminates as a result of hardware failures or other problems. Thus, utilizing calendar time may not provide an accurate representation of the reliability of the software product, but it may be sufficient if abnormal termination of the system occurs infrequently due to other reasons beside program failures.

Also of interest may be the Mean Time Between Failure (MTBF), which is used to determine the length of time between program failures. The Mean Time Between Failure, or the average time (downtime and uptime) between program failures is computed as follows:

$$\text{Mean Time Between Failure (MTBF)} =$$

$$\text{Mean Time to Failure (MTTF)} + \text{Mean Time to Repair (MTTR)}$$

The MTTR Indicator is discussed next. MTBF includes both the MTTF and the MTTR Indicators, and therefore addresses both reliability and maintainability factors. Because it cannot indicate either the reliability or maintainability without further looking at either MTTF or MTTR, it is not considered as an indicator.

F. References

[KOPH79]/p.4

[ROSL83]/p.114

Mean Time to Repair

A. Description

The Mean Time to Repair (MTTR) Indicator is used to determine the length of time the system is down after execution halts as a result of a defect in the program. Implementation of this indicator is accomplished by keeping track of the length of time that the system is down as a result of a program failure. This indicator can be used in determining the maintainability of the product.

B. Application Phase

The Mean Time to Repair Indicator can be used during deployment. It is not used during product acquisition. It is a Non-affective Indicator.

C. Implementation

The program operates for a period of time, before a defect in the program causes the program to be inoperable for a period of time. The execution is restarted, after repair of the serious defect causing the failure. The cycle continues. The Mean Time to Repair, or the average time to repair the software after a program failure occurs is computed as follows:

$$\text{Mean Time to Repair (MTTR)} = \frac{\text{total downtime}}{\text{total number of failures}}$$

The total downtime is taken as the total downtime necessary to repair the defects causing the failures and not downtime as a result of hardware failures or other problems.

This indicator is calculated on a periodic basis (e.g. every month). A line chart is plotted against time. See Figure 21 on page 133 for an example. Time is plotted along the horizontal axis. The MTTR value is plotted along the vertical axis. During the first month, the system was down 400 hours as a result of 80 program failures. Thus, the MTTR for the first month was 50 hours. Plotting continues in this manner throughout the Operations and Maintenance Phase.

D. Interpretation

The MTTR Indicator determines the average amount of time necessary to repair the software before the execution can be restarted. Program failures are usually serious enough to require that the defect causing the failure be corrected as soon as possible.

Maintainability:

An increasing or steady plotted line indicates that a maintainability problem may exist. The plotted line is expected to decrease, as in Figure 21, indicating the defects which are serious enough to cause failures are being repaired at at a more rapid rate. If the defects are being corrected faster, the maintainability of the product should not be declining.

E. Additional Comments

A more accurate representation of maintainability may be obtained by performing the calculation utilizing faults which are severe enough to necessitate repair immediately. Faults may occur, which do not lead to failure, that are still considered as very severe.

Distributions of the Time to Repair have been proposed as more helpful than an average as given by MTTR [LITB78].

F. References

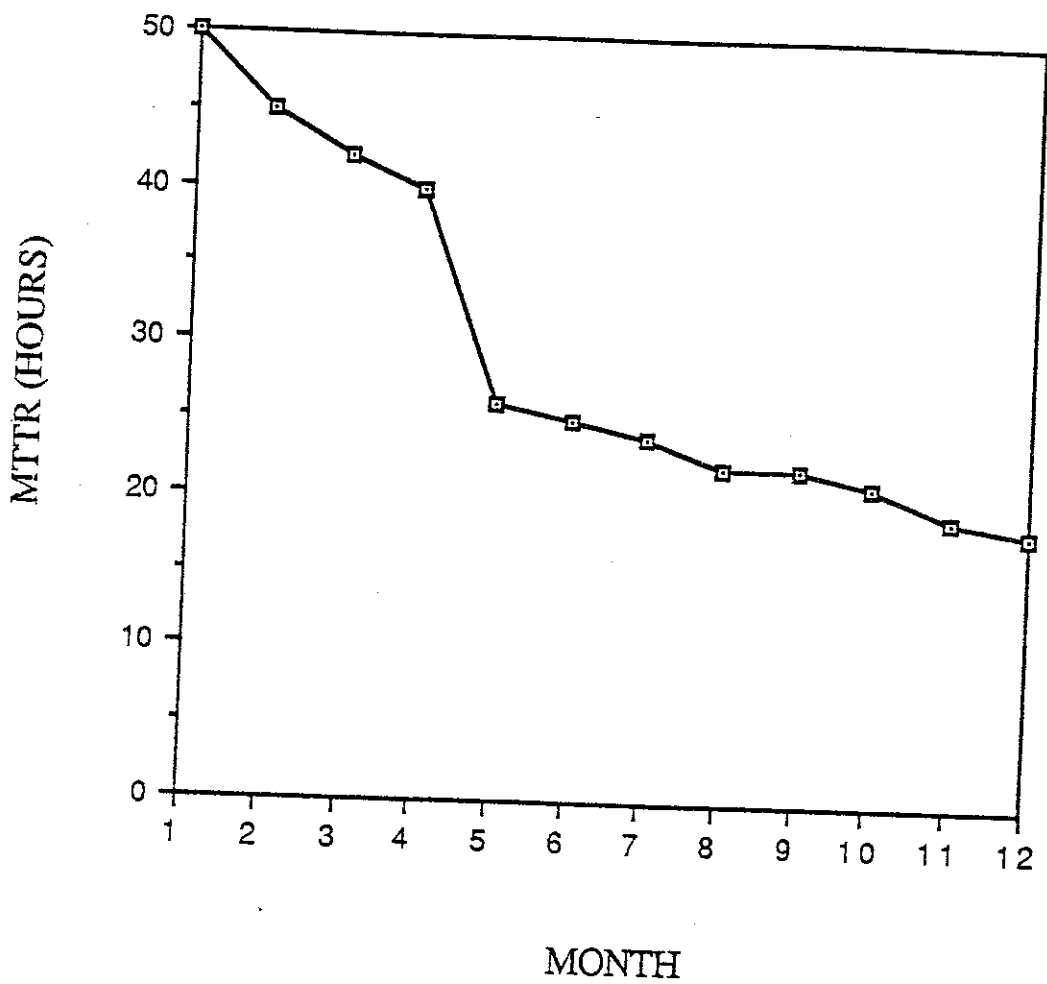

Figure 21. Mean Time to Repair (MTTR)

[KOPH79]/p.4

[ROSL83]/p.114

Input Index

A. Description

The Input Index Indicator is used to determine the percentage of inputs into the software system during operation which resulted in execution failure as a result of a defect in the program. Implementation of this indicator is accomplished by keeping track of the percentage of inputs that resulted in failure due to a defect in the program. This indicator can be used in determining the reliability of the product.

B. Application Phase

The Input Index Indicator can be used during deployment. It is not used during product acquisition. It is a Non-affective Indicator.

C. Implementation

The program operates for a period of time, before a program failure causes the program to be inoperable for a period of time. The failure is a result of a certain input or combination of inputs to the system. The Input Index, or the percentage of inputs upon which the execution does not fail, is computed as follows:

$$\text{Input Index} = 1 - \frac{\text{number of inputs with execution failures}}{\text{total number of inputs}}$$

This indicator is calculated on a periodic basis (e.g. every month). A line chart is plotted against time. See Figure 22 on page 137 for an example. Time is plotted along the horizontal axis. The

Input Index value is plotted along the vertical axis. During the first month of operation, there were 10,000 total inputs to the system, of which 2,300 resulted in a program failure. These values result in an Input Index of 0.77. Plotting continues in this manner for each time period of operation.

D. Interpretation

The higher the value of the Input Index Indicator, i.e. the closer to 1, the greater the number of inputs that can be entered into the system before a failure occurs as a result of a defect in the software product.

Reliability:

The greater the value, the more reliable the software product is likely to be. The plotted line is expected to increase, as in Figure 22, indicating that failures are occurring less frequently relative to the amount of information input to the system, i.e. more information is being input into the system before the execution fails.

E. Additional Comments

The Input Index Indicator can also be implemented by determining the number of inputs between software faults. Faults may occur, which do not lead to failure, that are still considered as very severe. This may provide a more accurate representation of reliability.

F. References

[GILT77]/p.146

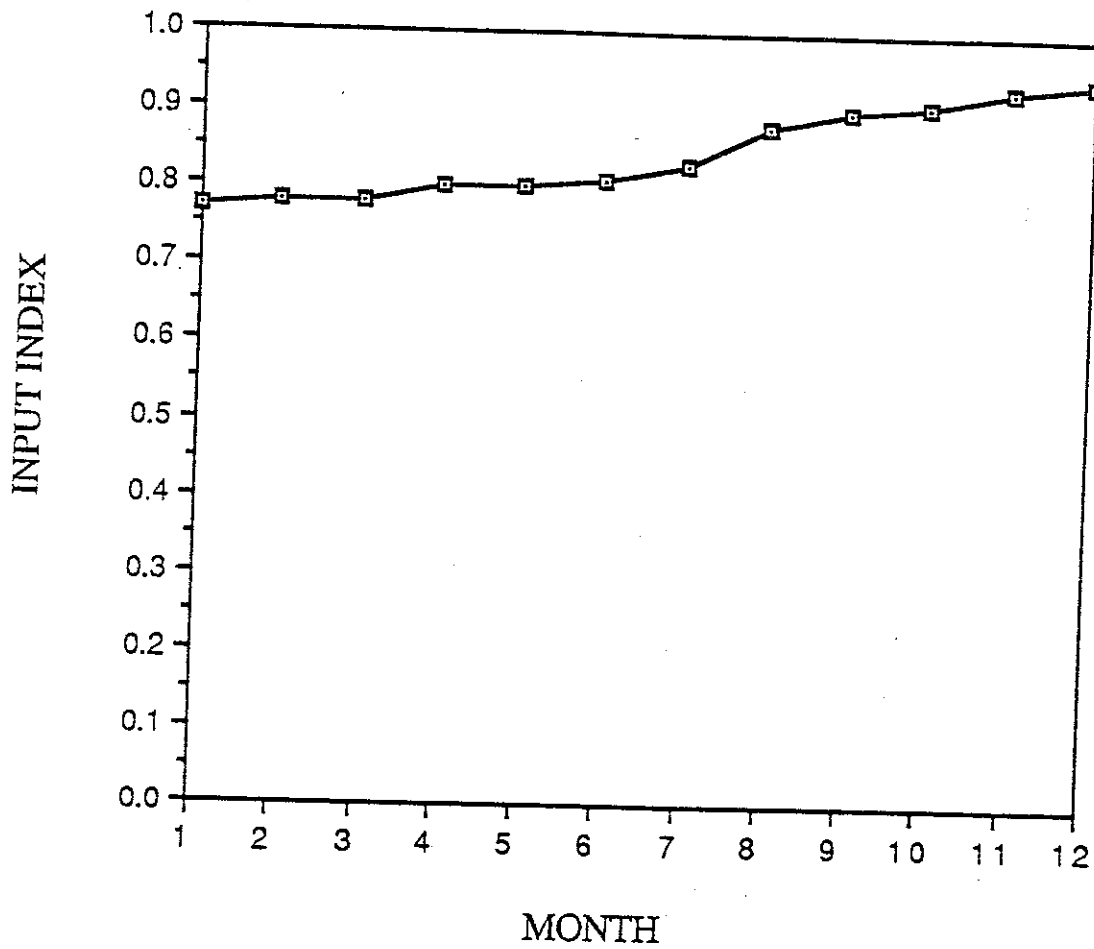

Figure 22. Input Index

Failure Rate

A. Description

The Failure Rate Indicator is used to determine how often program failures are occurring. Implementation of this indicator is accomplished by keeping track of the number of times that execution is terminated as a result of a defect in the program. This indicator can be used in determining the reliability of the product.

B. Application Phase

The Failure Rate Indicator can be used during deployment. It is not used during product acquisition. It is a Non-affective Indicator.

C. Implementation

The program operates for a period of time, and a defect in the program causes a failure which results in the program being inoperable for a period of time. Execution is restarted, after repair of the serious defect causing the failure. The cycle of execution and downtime continues. The Failure Rate, or the frequency of program failures, is computed as follows:

$$\text{Failure Rate} = \frac{\text{total number of failures}}{\text{observed time period}}$$

This indicator is calculated on a periodic basis (e.g. every month). See Figure 23 on page 140 for an example. A line chart is plotted against time. Time is plotted along the horizontal axis. The Failure Rate value is plotted along the vertical axis. During the first month, 600 total failures oc-

curred during 300 hours of execution which were a result of a defect in the program. Plotting continues in this manner throughout the life of the software product.

D. Interpretation

The Failure Rate Indicator determines the average rate at which program failures are occurring. These failures have a tremendous impact on the reliability of the system.

Reliability:

An increasing or steady plotted line indicates there may exist a reliability problem. The plotted line is expected to decrease, as in Figure 23, indicating the defects which are serious enough to cause failures are occurring less frequently. If failures are occurring less frequently then defects in the program are probably being reduced.

E. Additional Comments

A more accurate representation of reliability may be obtained by performing the calculation utilizing faults instead of only failures. Faults, particularly severe faults, also have an impact on the reliability of the software.

This indicator has also been calculated using both calendar time and execution time. The use of calendar time provides no allowance if the system is down as a result of hardware failures or other problems. Thus, utilizing calendar time may not provide an accurate representation of the reliability of the software product, but it may be sufficient if abnormal termination of the system occurs infrequently due to other reasons beside program failures.

F. References

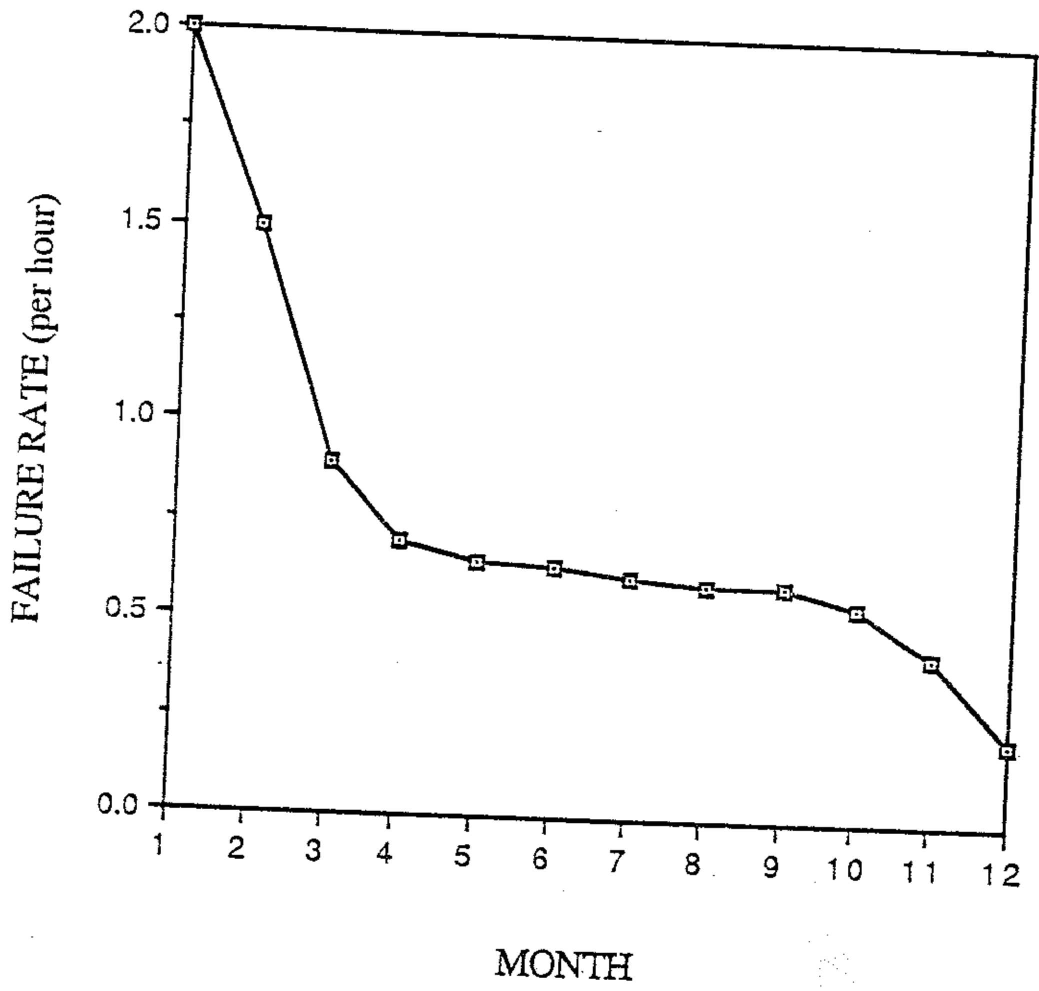

Figure 23. Failure Rate

[ROSL83]/pp.113-114

Availability

A. Description

The Availability Indicator is used to determine the percentage of time that the software product is available. Implementation of this indicator is accomplished by keeping track of the number of program failures and the total downtime and uptime of the system. This indicator can be used in determining the reliability and/or maintainability of the product.

B. Application Phase

The Availability Indicator can be used during deployment. It is not used during product acquisition. It is a Non-affective Indicator.

C. Implementation

The program operates for a period of time, and a program failure causes the program to be inoperable for a period of time. Execution is restarted, after repair of the serious defect causing the failure. The cycle of execution and downtime continues. Failures are serious enough to cause loss of availability of the software. The Availability of the software, or the percentage of uptime is computed as follows:

$$\text{Availability} = \frac{\text{Mean Time to Failure (MTTF)}}{\text{(Mean Time to Failure (MTTF) + Mean Time to Repair (MTTR))}} \times 100\%$$

D. Interpretation

A small value of the Availability Indicator may indicate a problem in the reliability and/or maintainability of the software product. Figure 24 on page 144 [KOPH79,p.5] illustrates how MTTF and MTTR must relate in in order to achieve a high reliability.

Reliability:

If the value is not very large, reliability may be suspect. A small Availability may be attributable to a low MTTF, that is, failures may be occurring very rapidly. Even if the average repair time is low, the lower the MTTF, the lower the Availability.

Maintainability:

If the value is not very large, maintainability may also be suspect. A small Availability may be attributable to a high MTTR, that is, defects causing the failures may be difficult to correct. Even if the average time to failure is high, the higher the MTTR, the lower the Availability.

E. Additional Comments

The Availability Indicator can be extended by performing the calculation on a periodic basis (e.g. every month). A line chart is plotted against time. See Figure 25 on page 145 for an example. Time is plotted along the horizontal axis. The Availability value is plotted along the vertical axis. During the first month, the MTTF is 45 hours and the MTTR is 55 hours, resulting in an Availability of 45%. Plotting continues in this manner throughout deployment. The Availability of the software is expected to increase with time. A steady or decreasing value may indicate a reliability and/or a maintainability problem.

F. References

[KOPH79]/p.4-5

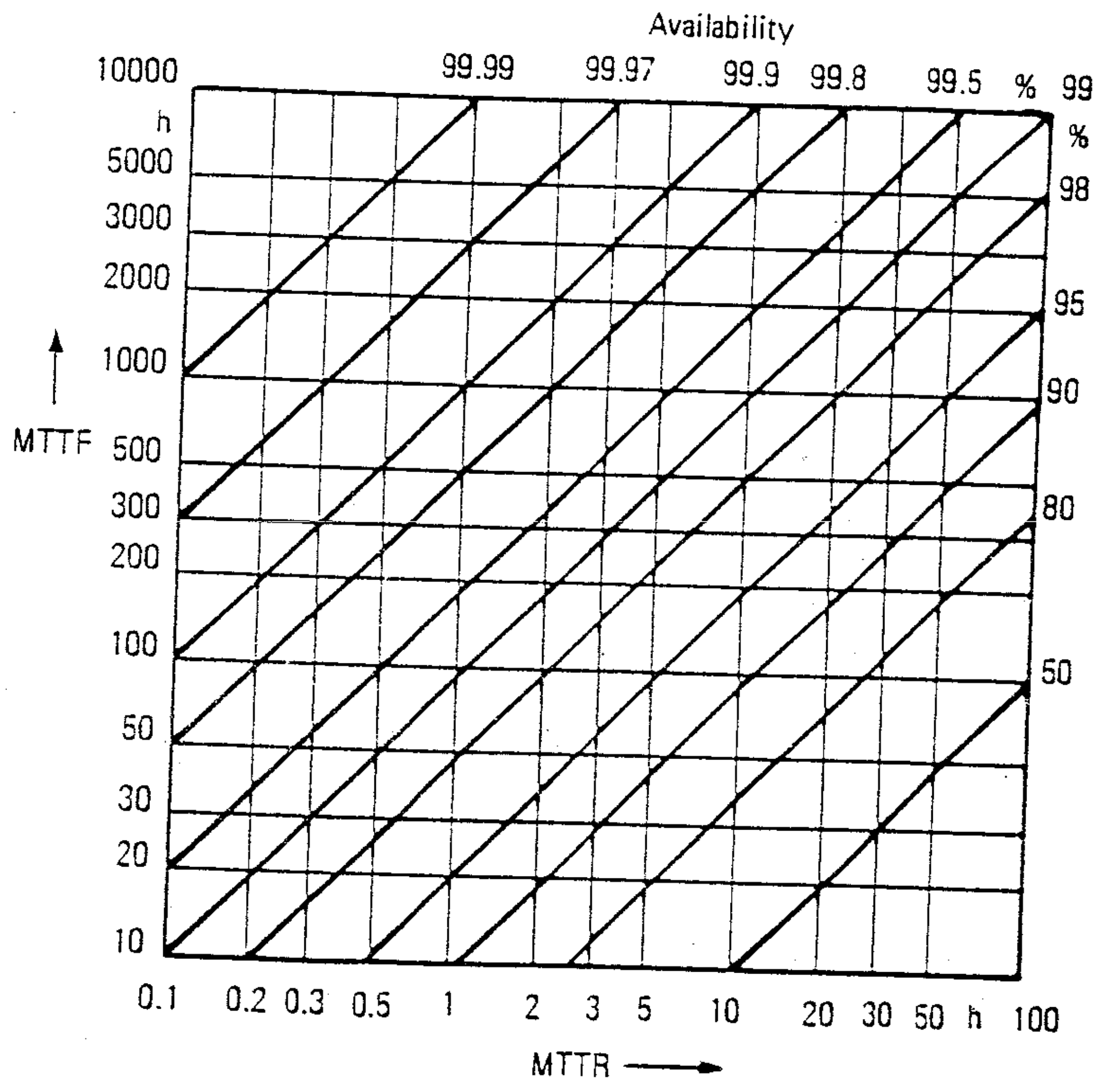

Figure 24. Availability: Relationship between MTTF, MTTF, and Availability

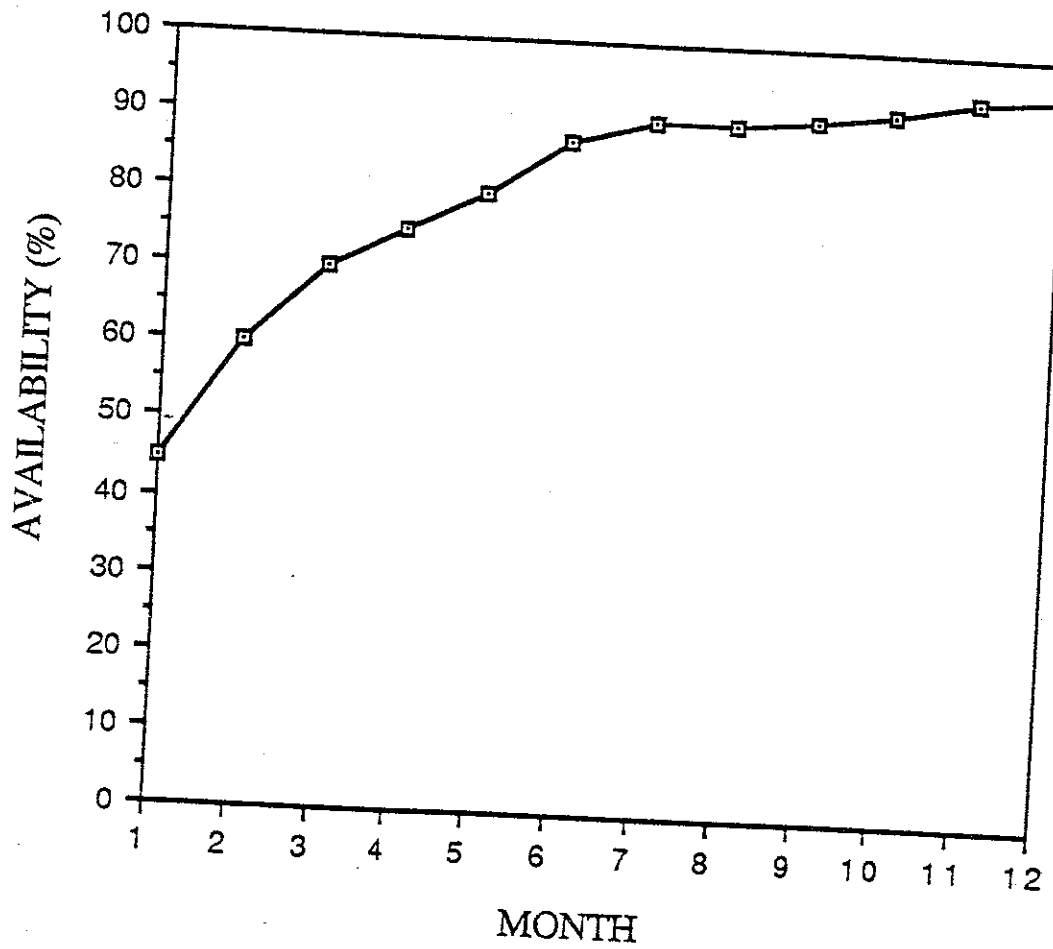

Figure 25. Availability

Appendix References

AFSC87 Air Force Systems Command Pamphlet 800-14, *Software Quality Indicators; Management Quality Insight,* Department of the Air Force, January 20, 1987.

GILT77 Gilb, T, *Software Metrics,* Winthrop Publishers, Inc., Cambridge, Massachusetts, 1977.

IEEE85 IEEE Project P982, Draft Standard P982, *Software Reliability Measurement,* IEEE Computer Society, June 1, 1985.

KOPH79 Kopetz, H., *Software Reliability,* MacMillian Press Ltd., London, England, 1979.

MACJ86 MacMillian, J. and Vosburgh, J. R., *Software Quality Indicators,* Scientific Systems, Inc., Cambridge, Massachusetts, September 24, 1986.

LITB78 Littlewood, B., " How to Measure Software Reliability, and How Not to...., " *Proceedings of the 3rd International Conference on Software Engineering,* IEEE Computer Society, 1978, pp.37-45.

MILH76 Mills, H. D., " Software Development, " *IEEE Transactions on Software Engineering,* Vol. SE-2, No. 4, December, 1976, pp.265-273.

ROSL83 Rosenthal, L. S., " Planning and Implementing System Reliability, " *Proceedings of Total Systems Reliability Symposium,* IEEE Computer Society, pp.112-118, 1983.

SWAE76 Swanson, E. B., " The Dimensions of Maintenance, " *Proceedings of the 2nd International Conference on Software Engineering,* IEEE Computer Society, 1976, pp.492-497.