Developing an Automated Procedure for Evaluating Software Development Methodologies and Associated Products

James D. Arthur
Richard E. Nance

TR 87-16

Developing an Automated Procedure For Evaluating Software Development Methodologies and Associated Products*

James D. Arthur and *Richard E. Nance*

PREFACE

On 31 December 1985 the Naval Surface Weapons Center at Dahlgren and the Systems Research Center at Virginia Tech formally concluded a joint research effort that addressed immediate software issues for embedded systems applications [NANR 85]. A major portion of that effort focused on the formulation and preliminary validation of a procedural approach to evaluating software development methodologies. Based on the validation results and potential significance of the evaluation procedure, the Systems Research Center was tasked on 16 June 1986 to continue its validation effort and further investigate the merits of the evaluation procedure. More specifically, that task statement focuses on

(1) assessing the perceived strengths and weaknesses of the current procedure for evaluating software development methodologies,

(2) basing the evaluation process on statistical indicators rather than "surface" properties of the product, and

(3) automating the evaluation process.

This report presents a brief overview of the evaluation procedure, a summary of the initial validation study, and individual discussions of the above three task issues.

CR Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/ Specifications; D.2.2 [**Software Engineering**]: Tools and Techniques

General Terms: Methodology Evaluation, Software Development Methodologies, Procedural Evaluation

Additional Keywords: Methodology, Evaluation, Linkages, Objectives, Principles, Attributes, Indicators, Properties

* Work supported by the U.S. Navy through the Systems Research Center under the Basic Ordering Agreement N60921-83-G-A165 B023

Developing an Automated Procedure For Evaluating Software Development Methodologies and Associated Products

1. Introduction.

Over the past decade the demand for increasingly complex software systems has risen dramatically [PARD 85]. Recognizing the fact that such systems cannot be developed effectively through *ad hoc* means, the software engineering community has continued to investigate the fundamental concepts, development methodologies, and automated tools to assist in the software development process. For example, SREM [ALFM 85] and SADT [SOFT 76, ROSD 77] are methodology based environments that focus on supporting particular phases of the software life cycle. SCR [CLEP 84, HENK 78] and DARTS [GOMH 84], on the other hand, are methodologies that emphasize specific goals, e.g., reducing software development costs and designing real-time systems, respectively. This steady proliferation of design methodologies, however, is not without its price. In particular, users find increasing difficulty in choosing an appropriate methodological approach and recognizing reasonable expectations of a design or development methodology. These concerns motivated an initial research effort that led to a procedural approach for evaluating software development methodologies [ARTJ 86].

Intuitively, the evaluation procedure is based on the observations that

(1) software development methodologies emphasize specific goals, and in particular, software engineering *objectives*,

(2) to achieve these objectives, however, software engineers must utilize the proper set of software engineering *principles* during the software development process, and

(3) the application of these principles induces software *attributes* considered desirable and beneficial. These observations imply a natural relationship among objectives, principles, and attributes that provides the basis for the evaluation procedure.

Because the above mentioned relationships stem from observations that appeal more to intuitive logic, part of the initial research effort mentioned above focuses on verifying the individual relationships and ascertaining the actual effects of their collective interaction [ARTJ 86]. Results of that effort, detailed in Section 2 of this paper, indicate that the evaluation procedure does support a well-defined process for *assessing* the adequacy of development methodologies. Moreover, the findings also suggest that the evaluation procedure may provide a basis for *measuring* how well a software product conforms to the underlying objectives of a designated methodology. As with most research efforts, however, new results (and their implications) foster additional concerns. In particular, the following set of questions have risen.

(1) **What are the strengths and weaknesses of the current evaluation procedure?** The evaluation procedure relies on a well-defined set of linkages (or relationships) among software engineering objectives, principles and attributes. The extent to which these linkages reflect the general opinion of software engineering experts is a measure of procedure's accuracy and adequacy.

(2) **How can subjectivity and personal bias be minimized during the evaluation process?** Product properties form a basis for determining the "usability" of a methodology. Unfortunately, the measurement of such properties is currently susceptible to the evaluator's opinions and biases. Basing the assessment on statistical criteria, or indicators, is suggested as an alternative approach.

(3) **Can the evaluation procedure be automated?** The effort required to collect evaluation statistics during the initial study mentioned above were substantial. Reduction of this effort is a prerequisite for a practical approach to evaluating software development methodologies.

Motivated by these issues, and desiring a better understand the software development process, the authors undertook a critically review the evaluation procedure. Results and implications of the review process are described in the remainder of this paper.

The organization of this paper reflects the major emphases of the review process. As background, the following section presents a brief outline of the evaluation procedure and the results of applying the evaluation procedure to two Navy software development methodologies. Next, Section 3 discusses a literature "verification" and the authors' reassessment of perceived linkages among the software engineering objectives, principles and attributes. Section 4 addresses the need for and logical derivation of *Software Quality Indicators*. Finally, Section 5 discusses automating the evaluation process. In particular, it describes why an automated evaluation process is necessary, how one might achieve such a goal, and what difficulties must be overcome. As implied by the brief descriptions of Sections 3 through 5, the topics discussed directly addressed the three issues (or questions) presented above.

2. Background and Overview of the Current Evaluation Procedure

A methodology can be viewed as a collection of methods (or procedures), chosen to complement each other, and a set of rules for applying them. Generally, a methodology applies to one or more phases of the software life cycle [BOEB 76]. In concert with the principal goal of software engineering [BAUF 72], a methodology should:

(1) embody sound engineering principles, that lead to

(2) the economic production of software, that is

(3) reliable and efficient on existing computers,

(4) assessed over the entire life of the software.

Although methodologies may differ in how this goal is to be reached, the goal should be apparent in the objectives set forth by any methodology.

2.1 Linking Objectives, Principles and Attributes

Reflecting the above view, the rationale supporting a *procedural* approach to evaluating software development methodologies is based on the argument that:

> A set of *objectives* can be defined that should be postulated within any software engineering methodology. Achieving these objectives requires adherence to certain *principles* that characterize the process by which software is created. Adherence to a process governed by those principles should result in a product (program and documentation) that possesses *attributes* considered desirable and beneficial.

Underlying this rationale is a *natural* relationship that links objectives to principles and principles to attributes. That is, as illustrated in Figure 1, one achieves the *objectives* of a software development methodology by applying fundamental *principles* that, in turn, induce particular code and documentation *attributes.* It is precisely this linkage that provides a basis for the evaluation procedure.

Consider for example the single objective, *reusability*. Formally, reusability can be defined as the extent to which a module can be used in multiple applications. Accepting this objective as a goal of a software development methodology, Figure 2 illustrates the major principles (hierarchical decomposition, functional decomposition, information hiding, and documentation) contributing to the realization of that objective. Expanding the single principle, information hiding, we note the five attributes (reduced coupling, enhanced cohesion, well-defined interface, ease of change, and low complexity) that should be evident in software developed using a process governed by the principle of information hiding. Narrowing our attention to one of these attributes, well-defined interface, we identify an additional set of characteristics related to the well-defined interface attribute. These characteristics form the set of observable properties which contribute to the claim that a piece of software exhibits a well-defined interface.

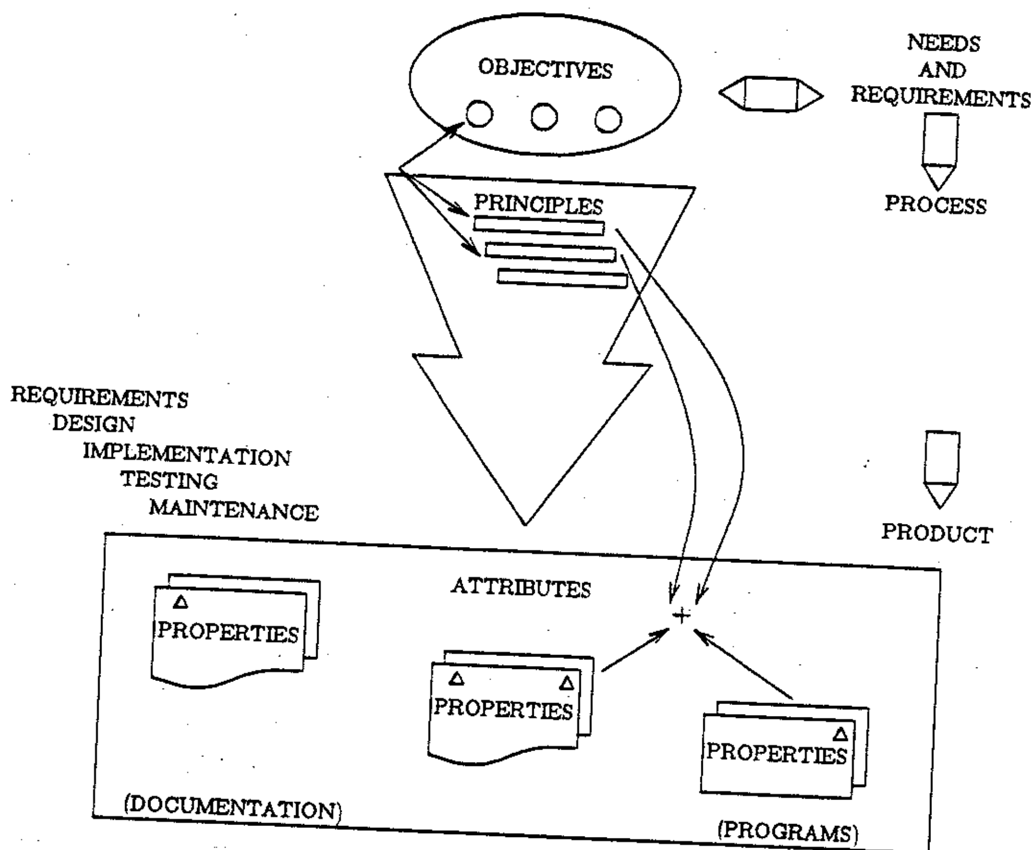

Figure 1

Illustration of the Relationships Among Objectives, Principles, Attributes, and Properties in the Software Development Process

The above description presents a simplified overview of the linkage relationship that exists among the software engineering objectives, principles, and attributes. It also introduces the concept of ***properties***, that is, characteristics of code and documentation that allow one to assess to what extent an attribute is present or absent. The exhaustive enumeration of linkages among objectives, principles, and attributes produces a graph with numerous edges, revealing many dependencies [NANR 85]. Additionally, a single property can be indicative of several attributes. Appendix 1 provides an outline of the linkage foundation; a detailed discussion relating the complete set of objectives, principles, and attributes can be found in [ARTJ 86].

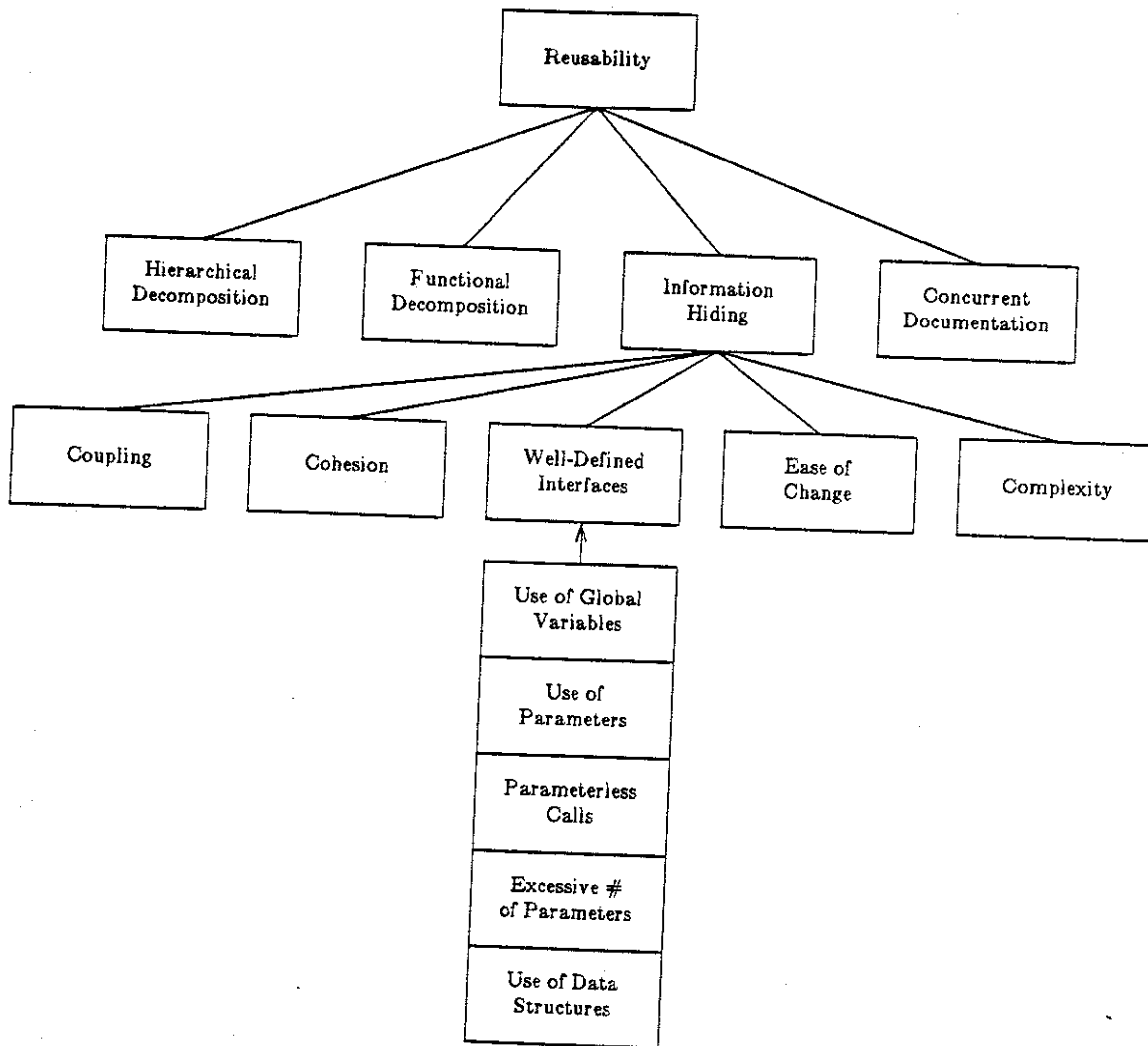

Figure 2

Illustration of the Evaluation Procedure

2.2 On Evaluating Methodologies and Products

Currently, the relationships (or linkages) among objectives, principles, and attributes support several evaluation methods, each of which provides various kinds of information. Prominent among these methods are the *top-down* and *bottom-up* evaluation processes. Together, they provide a basis for judging the adequacy of a software development methodology and to what extent a software product conforms to the objectives of a methodology.

The Top-Down Evaluation Process

The top-down evaluation process allows one to evaluate a methodology in terms of its objectives (goals). The first step in this process is to recognize the *objectives* emphasized by the methodology. Based on the linkages among objectives and principles, the second step is an investigation of the software development *process*. That is, given a stated set of methodological objectives, one asks: What *principles* are supported by the methodology to achieve those objectives? The presence of principles without corresponding objectives or vice versa should trigger an alarm. The third step in the top-down scheme, formulating the set of expected product *attributes*, is based on the fact that principles govern the process by which a software product is produced. That is, a given set of principles should induce a corresponding set of product attributes. Hence, using the linkages among principles and attributes, one determines the expected set of product attributes and asks – are these attributes desirable? If the answer to this question is "yes" then the selected methodology is deemed adequate. That is, it supports an underlying software development process for achieving the desired software engineering objectives and product attributes.

The Bottom-Up Evaluation Process

With respect to the stated objectives of a software development methodology, the *top-down* evaluation process amplifies deficiencies in the the software development process, the *bottom-up* evaluation process, however, reveals product anomalies. That is, the bottom-up approach enables one to determine to what extent a software product conforms to the stated objectives of a software development methodology.

Similar to the top-down approach, the bottom-up evaluation scheme also relies on the existence of linkages among software engineering objectives, principles, and attributes. Unlike the top-down process, however, the bottom-up scheme (1) starts with the *computed* set of attributes (based on properties found to exist in the software product), and (2) using the defined linkages among principles and attributes, infers the necessary set of principles needed to induce the attributes,

and then (3) using the objective/principle relationship determines the overall set of objectives that are stressed in constructing the software product. In effect, given the pronounced objectives and supporting principles of a software development methodology, one can apply the bottom-up evaluation process to a specified product and determine *if* and *to what extent* the attributes of that product reflects the goals and objectives stressed by the methodology.

2.3 Application of the Evaluation Procedure

A joint investigation of two comparable Navy software development methodologies and respective products is described in [NANR 85]. The investigation effort utilizes:

- four software development methodology documents for

  (1) identifying the pronounced software engineering objectives, principles, and attributes, and

  (2) assessing the adequacy of each methodology through the objective/principle/attribute linkages defined by the evaluation procedure, and

- eight software system documents and 118 routines, comprising 8300 source lines of code, for

  (1) determining the evident set of product attributes, and

  (2) via the attribute/principle/objective linkages, empirically assessing the principles and objectives emphasized during product development.

The following section provides a summary of those results and illustrates the utility and versatility of the procedural approach to evaluating software development methodologies. For simplicity, we will refer to the software systems as system A and system B (and methodology A, methodology B, respectively).

	Methodology A	Methodology B
Objectives		
Maintainability		Yes
Correctness	Yes	
Reusability		
Testability		
Reliability	Yes	Yes
Portability		
Adaptability		Yes
Principles		
Hierarchical Decomposition	Yes	
Functional Decomposition	Yes	
Information Hiding		
Stepwise Refinement		
Structured Programming	Yes	Yes
Documentation		Yes
Life Cycle Verification		
Attributes	None	None

Figure 3

Pronounced Objectives, Principles, and Attributes

The initial step in the evaluation process is to perform a "top-down" analysis of methodologies A and B. For each methodology, this reveals the set of *pronounced* software engineering objectives, principles, and attributes. Because both methodologies are a product of evolution, however, a *clear* statement of their respective methodological objectives is lacking. Nonetheless, as detailed in Figure 3, the documentation for methodology A does appear to stress the objectives of *reliability* and *correctness* supported by the principles of *structured programming, hierarchical decomposition,* and *functional decomposition.* Following the objective/principle relationships defined by the evaluation procedure, for each objective stressed in methodology A only three of the necessary four principles are emphasized. The implication is, that unless the principles of life-cycle verification and information hiding are implicitly assumed *and* utilized, correctness and reliability, respectively, are compromised.

Using metric values and properties, a corresponding "bottom-up" examination of product A

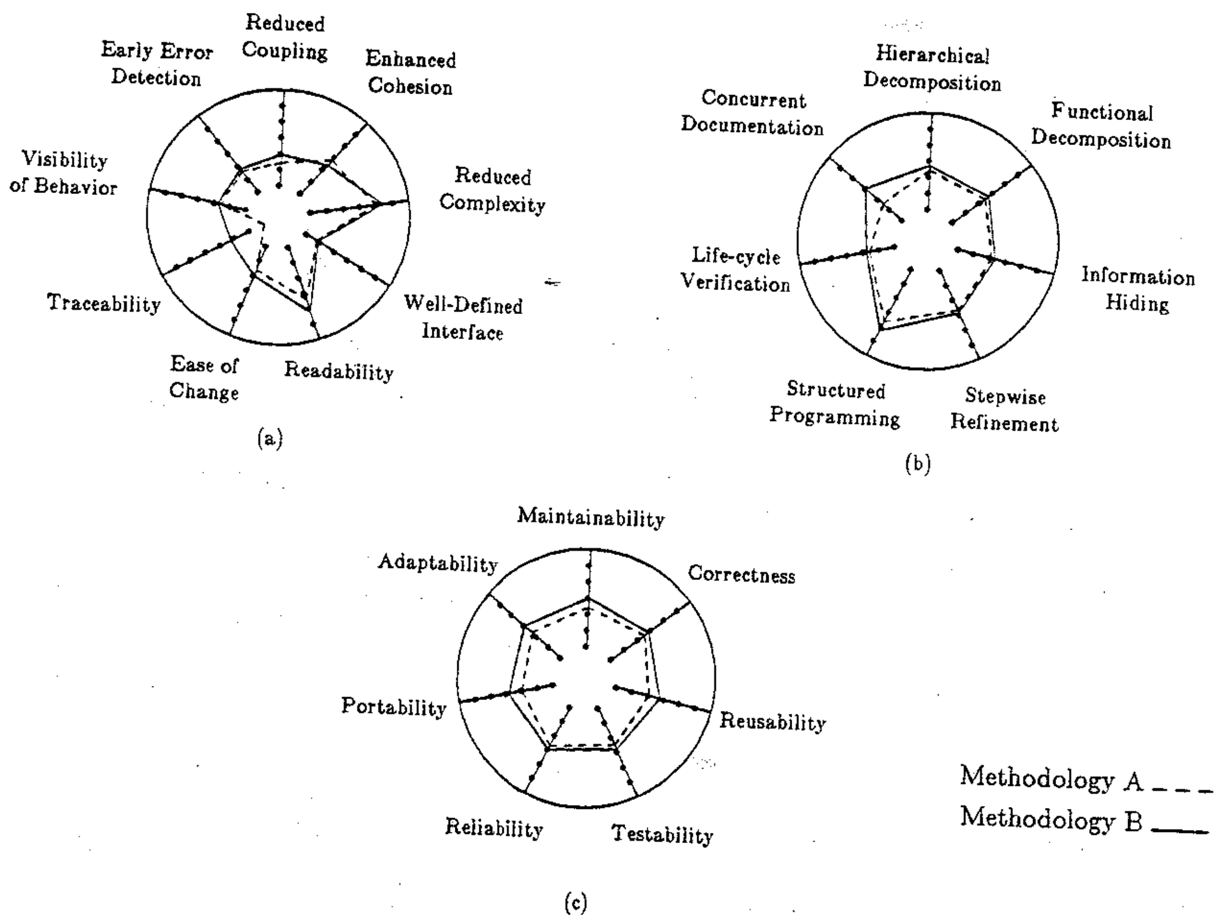

Figure 4

Detected Presence of Objectives, Principles, and Attributes

provides some interesting results. The Kiviat graph displayed in Figure 4a illustrates the extent to which each attribute is *assessed* as present in the product. Note that (reduced) complexity attains the highest rating (8.0), closely followed by readability (7.4) and cohesion (6.8). Based on the three principles stressed in methodology A, the evaluation procedure predicts that (reduced) complexity, readability, and cohesion *should*, in fact, be among the product attributes.

In concert with the stated objectives and principles for methodology A, Figure 4b reveals that structured programming (7.7) is the prominent principle used in developing system A, followed by stepwise refinement (6.7), hierarchical decomposition (6.4), and functional decomposition (6.4). Figure 4c depicts the results of emphasizing these principles in the software development process. In particular, reliability is rated as the major software development objective (6.7). Although

correctness is also stressed by methodology A, ascertaining correctness necessitates life-cycle verification; this principle is neither emphasized by methodology A, nor evident in the software product. As illustrated by Figures 4a, 4b and 4c, other objectives and principles are given some emphasis during the software development process for system A. It is the authors' opinions, however, that because they are not explicitly stressed in methodology A, the associated product suffers.

For methodology B, the objectives enunciated in the documentation are *maintainability, adaptability,* and *reliability*. *Structured programming* and *documentation* are the emphasized principles. Like methodology A, however, a complete set of supporting principles are not stated; *hierarchical decomposition, functional decomposition*, and to some extent *information hiding* are implicitly assumed as underlying principles of methodology B. According to the linkages among objectives and principles, all of the above principles (both stated and assumed) are required to achieve the objectives explicitly stated in methodology B.

Subsequent analysis of product B and a "bottom-up" propagation of the results through the linkages defined by the evaluation procedure reveals structured programming as the most prominent principle (8.3), closely followed by documentation (7.0). Moreover, the evaluation also indicates that the implicitly assumed principles of methodology B are highly utilized – stepwise refinement, hierarchical decomposition, functional decomposition, and information hiding rate 6.9, 6.7, 6.7, and 6.3, respectively. Finally, the results imply that during the development of product B the objectives of maintainability, adaptability, and reliability are most emphasized. The above assessments are illustrated in Figures 4a, 4b, and 4c.

To summarize, the evaluation procedure reveals that both methodologies lack a clear statement of goals and objectives, as well as sufficient principles for achieving the objectives that are emphasized. Moreover, glaring deficiencies are apparent in both software development methodologies. That is, both fail to actively support the principle of information hiding and also have difficulties in incorporating the desirable attributes of traceability and well-defined interfaces in respective system products. In general, the evaluation procedure does accurately assesses the software engineering objectives, principles, and attributes *espoused* by methodologies A and B. Of particular

significance, however, is that the objectives and principles *determined* to be "emphasized" during the product development process, yet not stated in the methodology documentation, are precisely those that are *implicitly assumed* important by the software engineers developing products A and B. A more detailed account of the evaluation can be found in [NANR 85].

The evaluation results described above expose inherent strengths and weaknesses of methodologies A and B. Although the appraised strengths are consistent with general perceptions, several implied deficiencies are unexpected. The discrepancy between the *perceived* and *assessed* methodological traits underly stated concerns and are addressed in the following sections. In particular, items discussed include (a) the adequacy of the sets of objectives, principles and attributes assumed to be desirable and beneficial, (b) the sufficiency and validity of the linkages that bind objectives to principles and principles to attributes, and (c) the subjectivity, assessment criteria and manpower requirements inherent to the current evaluation process.

3.0 Strengths and Weaknesses of the Evaluation Procedure

The current procedure for evaluating software development methodologies is based on a set of relationships (or linkages) that exist among software engineering objectives, principles and attributes. The operational approach to assessing the adequacy of a given methodology entails (a) an identification of the objectives, principles and attributes that a methodology *claims* to support, and (b) a product analysis to determine the extent to which measurable characteristics *substantiate* the claim. Results obtained from (a) and (b) are heavily influenced by the defined sets of objectives, principles and attributes, and the perceived linkages among them. Accordingly, any effort to substantiate or enhance the credibility of the evaluation findings must include:

(1) a *comparison* between the current evaluation procedure and others similar to it,

(2) a *verification* of elements fundamental to the evaluation process, i.e. the assumed sets of objectives, principles, attributes and measurement properties, as well as the linkages among these sets, and

(3) a *reassessment* of the the evaluation procedure based on the finding from (1) and (2).

As discussed below, current literature provides the material for the comparison and verification effort.

3.1 A Comparison Among Procedures for Evaluating Methodologies

To develop a software product one begins with a set of requirements, proceeds toward a satisfactory design and then translates the design into a system that satisfies those requirements. This seemingly natural approach to developing software systems, however, has not always been at the disposal of the software engineer. Early software development practices were guided by "seat of the pants" intuition and were not even questioned until the late 1960s [DIJE 68, BOHC 66]. Today, software development techniques, methods, and methodologies abound. Their diversity and expansiveness, however, has prompted a real need for evaluating their applicability and adequacy to specific problem domains. Unfortunately, a survey of current literature indicates that only tangential issues pertaining to the evaluation of software development techniques and methodologies are being addressed. For example, researchers like Basili [BASV 84], Weiss [WEID 85], and Shen [SHEV 85] discuss methods for detecting error-prone software and suggest techniques for minimizing error occurrence. Although their results and conclusions are impressive, error detection and error avoidance represent only one facet of the software development process. From a more global perspective, Bergland [BERG 81], Colter [COLM 82], and Peters[PETL 77] present comparisons *among* software development methodologies. Their criteria for selecting one methodology over another, however, is based on matching the software engineers' needs with espoused methodological capabilities. Unfortunately, this approach only emphasizes the benefits of employing a methodological approach and lacks a procedural basis for comparing alternative approaches.

In summary, current literature does not suggest any procedural basis for evaluating software development methodologies. Although published results do provide significant insights into the evaluation of individual software development techniques, the studies treat each technique as an isolated entity, divorced from any particular methodological framework.

3.2 Linkage Verification

As described in Section 2.2, assessing the *adequacy* of a software development methodology is achieved through a "top-down" evaluation process that utilizes a formal description of the methodology and a well-defined set of relationships (or linkages) that exist among objectives, principles, and attributes. On the other hand, determining the *usability* of a methodology requires a product evaluation that focuses on the identification of properties which imply the existence of product attributes, followed by a "bottom-up" analytical process employing the same set of linkages mentioned above. Implicit in both the top-down and bottom-up evaluation scenarios are the assumptions that:

(1) the sets of objectives, principles, and attributes emphasized by the evaluation procedure are consistent with those deemed most significant by the software engineering community, and

(2) the objective/principle, principle/attribute and attribute/property linkages do, in fact, reflect a natural relationship underlying the achievement of objectives and the inducement of product attributes, respectively.

For the interested reader, a literature confirmation for assumption (1) is outlined in Appendix 1 and detailed in [ARTJ 86]. Assumption (2) is discussed in the remainder of this sub-section.

The Objective/Principle Linkages

The relationships among objectives and principles imply that to achieve a particular software engineering objective certain fundamental principles must be employed during the software development process. For each objective, Table 1 of Appendix 1 enumerates the major principles that contribute to the achievement of that objective. For example, maintainability is achieved and/or enhanced through (a) a design process that utilizes decomposition and stepwise refinement, (b) an implementation phase that exploits (again) stepwise refinement, information hiding and structured

programming, and (c) concurrent documentation. Now, from a verification standpoint one should ask - to what extent do other software engineers agree with the espoused objective/principle linkages shown in Table 1? The answer to this question is found in Appendix 2. That is, for the objective/principle relationships defined by the evaluation procedure, literature references corroborating *all* 32 linkages exist and are cited.

Based on the literature survey outlined in Appendix 2, the relationships defined among software engineering objective and principles appear to be well-defined and reflect a compendium of opinions. This set of relationships, however, is but one of three sets. The next discussion addresses the verification of linkages defined among principles and attributes.

The Principle/Attribute Linkages

As stated above, a fundamental premise of the evaluation procedure is that to achieve specific software engineering objectives, the software development process must be guided by the appropriate software engineering principles. A direct ramification of this premise is that one can determine, *a priori*, an expected set of product attributes. That is, whenever specific software engineering principles are used during product development, particular product attributes are realized. For each individual principle, Table 2 of Appendix 1 outlines the corresponding set of induced attributes. For example, if one uses stepwise refinement during the software development process, one should expect to (a) reduce coupling among the software modules comprising the product, (b) enhance statement cohesion within each module, and (c) reduce overall product complexity. Again, the appropriate question to ask is - do the principle/attribute relationships assumed by the evaluation procedure agree with the published findings of software engineers? Referring to Appendix 3, *all* assumed relationships are, in fact, substantiated in the cited literature articles.

As described in Section 2, the objective/principle and principle/attribute linkages are fundamental to assessing the adequacy of a software development methodology. Determining the extent to which a methodology is followed during the development process, however, requires product measurement based on resident properties and a set of relationships linking those properties to

the software engineering attributes discussed above. Understanding that product *measurement* methods do not fall within the scope of this research effort, the following discussion focusses on the results of a literature search aimed at verifying assumed attribute/property linkages.

The Attribute/Property Linkages

Properties, also referred to as assessment factors in Appendices 4 and 5, are defined to be observable qualities of a product that contribute to the claim that a piece of code or code document exhibits particular attributes. Product properties and the set of attribute/property linkages outlined in Appendix 4 play an integral roles in determining extant product attributes. For example, the use of control structures and code indentation tends to lower code complexity. On the other hand, properties that contribute to code complexity include the use of GOTOs and multiple exit points from a module. The complete set of properties (of which there are 41) and their relationships to the 9 attributes is presented in Appendix 4. Of the 74 linkages among properties and attributes, only 9 remain unverified. The 65 substantiated attribute/property relationships are presented in Appendix 5 along with appropriate references and literature extractions.

With respect to the 9 attribute/property linkages yet to be verified, an interesting observation is that 6 of them are "comment" related and intuitive by nature. For example, the evaluation procedure assumes that the use of block comments enhances readability. Similarly, the 3 non-comment attribute/property linkages are also generally accepted, e.g. the use of parameterless calls increases coupling between modules. Because literature references could not be found to substantiated the 9 linkages, however, the authors did reconsider each individual linkage with respect to its contribution and appropriateness. The resulting consensus was to keep all 9 linkages.

To summarize, substantiating the above mentioned linkages from a literature perspective has proven quite fruitful. In particular, the authors' selection of objective/principle, principle/attribute and attribute/property linkages appear to be consistent with other researchers and software engineers. Moreover, the literature search also revealed particular linkages that needed to be reexamined. The literature verification and reassessment of *individual* linkages, however, is only one

part of the research effort described in this report. The next step in ascertaining the strengths and weaknesses of the current evaluation procedure is to individually consider the completeness and adequacy of the enunciated sets of objective, principles and attributes. The process and results of that assessment are described in the following section.

3.3 A Reassessment of Objectives, Principles and Attributes

The original identification and delineation of objectives, principles and attributes rely primarily on the collective knowledge and experience of the three investigators. While literature references served to crystallize definitions and to promote convergence toward terminology choices, the foundations of the evaluation procedure emerged as a joint realization of *what should be the role of a methodology.* The first task called for a thorough reexamination of the three component sets: objectives, principles and attributes. This reexamination focused on:

(1) completeness: should additions be made to any set?

(2) redundancy: should any set member be excluded because it overlaps excessively with another member?

(3) accuracy: is the term properly defined, designated in the appropriate component set (objective, principle, attribute), and its scope adequately represented?

3.3.1 Objectives

Reexamination of the original set of seven objectives led to no revisions. The early exclusion of performance, based on the contention that such criteria represent a constraint imposed at the systems level, has been reaffirmed. "Cost" in the form of development cost or the cost of using a particular methodology is excluded following the rationale that the entire evaluation procedure is to establish the "life cycle cost." How the two types of cost – that incurred today versus that deferred until later – should be compared is a task beyond the software engineering domain. Perhaps our attitude here is an expression of an earlier philosophical directive [DIJE 72, p. 6]:

My conclusion is that it is becoming most urgent to stop to consider programming primarily as the minimization of a cost/performance ratio. We should recognize that already now programming is much more an intellectual challenge: the art of programming is the art of organising complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible.

The review of terminology reintroduced some earlier discussion, e.g. is "extensibility" preferable to "adaptability" or is the definition of reliability consistent with that used in the literature? No convincing arguments proved persuasive to alter the earlier judgments.

3.3.2 Principles

3.3.2.1 Addition of Abstraction

Reexamination of the set of principles led to the addition of *abstraction,* which has been identified as a principle by a number of authors:

> Abstraction is the definition of each program segment at a given level of refinement, and in terms of its relation as a unit to other program segments [GILP 82, p.248].
>
> Abstraction, then, is the ability to see the whole problem, ignoring irrelevant detail; to manipulate designs while they are general, without having to become specific [INFO 79].
>
> The essence of abstraction is to extract essential properties while omitting inessential details [ROSD 75, p. 22].
>
> We should appreciate abstraction as our main mental technique to reduce the demands made upon enumerative reasoning [DIJE 72, p. 11].

We find none of these definitions sufficiently descriptive of the meaning that we associate with the *principle* that should be employed in developing software.

Hoare [HOAC 76, p.84] describes the *process* of abstraction as extending over four stages:

(1) Abstraction: the decision to concentrate on properties which are shared by many objects in the real world, and to ignore the differences between them.

(2) Representation: the choice of a set of symbols to communicate the abstraction.

(3) Manipulation: the rules for transformation of the symbolic representations as a means of predicting the effect of similar manipulation of the real world.

(4) Axiomatisation: the rigorous statement of those properties which have been abstracted from the real world, and which are shared by manipulations of the real world and of the symbols which represent it.

We find this description of the abstraction process instructive and at the same time indistinguishable from a characterization of "modeling." We conclude that the definition of the *principle of abstraction* must rely on the fundamental relationship with the modeling process.

> *Abstraction* is the use of models of real world behavior that conform with a paradigm (representation, manipulation, axiomatisation) to transform descriptions of behavior ultimately to realizations.

3.3.2.2 Revision for Concurrent Documentation

The principle of documentation as adopted in the early study proved troublesome in that no evaluative basis is implied by the term. Clearly, the principle should not advocate "documentation for the sake of documentation," but the use of "good" as a modifier seems unnecessary if not trivial. The text by Tausworthe eliminated the difficulty in its definition of the *principle of concurrent documentation*, which includes the properties of purpose, content, and clarity but also stipulates that "the definition, design, coding, and verification phases of development cannot be regarded as complete until the documentation is complete" [TAUR 77, p. 32]. The implication of the modifier "concurrent" is precisely on target in that the principle should emphasize the recording of decisions, assumptions, restrictions and numerous other items *as they are encountered* not later subject to the inaccuracy of human memory.

3.3.2.3 Primary and Consequent Principles

As the investigation of principles developed, a recognition of dependency among terms became more apparent. For example, both forms of decomposition (functional and hierarchical) emanate from the employment of abstraction. As the progression is made from describing WHAT behavior is to take place to HOW that behavior is achieved, the use of either decomposition form is needed to impose a "boundary of understanding" and control the difficulty of the development task. Recognizing this dependency led to the characterization of functional and hierarchical decomposition as *consequent principles* of the *primary principle* of abstraction.

3.3.2.4 Other Issues Related to Principles

Several "principles" have been advocated in the literature that on examination are found to be included among the set under another name or inappropriately tagged as applicable to the process when the term describes an objective or attribute. The Software Cost Reduction Project (SCR) endorses *separation of concerns* as a principle [NRL 84], but the original use of the term by Dijkstra [DIJE 76, p.56, p.203] clearly indicates that the originator is advocating that preoccupation with (engineering) efficiency be avoided lest the (mathematical) concerns with correctness be undermined. More generally, Dijksta is admonishing the software developer to avoid the temptation to permit *implementation* issues to unduly affect early *specification* decisions. The SCR use of the term appears to reflect the same desired qualities that are associated with application of abstraction; therefore, no separation of concerns is viewed as encompassed within the principle of abstraction.

Modularity continues to be cited as a principle by some authors; however, these sources generally appear to be older (e.g. [ROSD 75]). In fact, modularity is an attribute that can be ascribed to code units to indicate whether they exhibit *cohesion* and *coupling*. Thus, the property of "modularity" is subsumed by coupling and cohesion. This same argument extends to the "principle of localization" [GILP 82, pp.247-248].

3.3.3 Attributes

No additions or modifications resulted from the examination of the set of attributes. Modularity is particularized in coupling and cohesion. Localization is perceived as a consequence of the same principles that lead to modularity. However, one discovery emerging from the review is the potential for analysis of design documentation to reveal characteristics very different from those extracted through code or code documentation. Coupling is a prime example, but complexity and ease of change also exhibit characteristics more likely to be identified in design documentation.

The major result of the attribute review is the clarification of the attribute set as essentially *application independent software characteristics.* This distinction originally caused much consternation in that "software performance" was excluded as either a software development attribute or objective. Rather, performance was viewed as a systems engineering decision that became manifest in timing and memory constraints imposed on the software engineering task. Insistence on application independent characteristics leads to the exclusion of "functionality" and "usability" as attributes since neither can be assessed beyond a specific application (see [GOON 87] for an interesting discussion of the relationship between the two).

4. The Basis for Software Quality Indicators

"Software quality factors," "software quality metrics," and "software quality indicators" – all are terms used in the conviction that the quality of the software product should be measurable, at least in a relative sense. Justification for such an opinion is rarely provided. The capability for measuring software quality seems such a clearly pervasive need that *it must simply be done.* A recent paper [KEAJ 86] issues a rather compelling criticism of the inadequate basis for measuring software complexity and the shortcomings of experimental research intended to support such metrics.

The motivation for using statistical indicators of software quality stems from the qualified successes in applying them to unmeasurable social concepts. This motivation is described in

Section 4.1. Extension of the applicable theory and the derivation of software quality indicators is described in Section 4.2, and the identification of some unresolved issues follows in Section 4.3.

4.1 Measuring the Unmeasurable: Social Indicators

Both economic and social indicators are based on the thesis that intangible qualitative conditions can be indirectly assessed by *measurable* quantitative characteristics. The economic indicators of a "good economy" are routinely discussed in business news. Social indicators of "safe streets" or a "good neighborhood," while less popular with the media, are often cited in cases where policy considerations extend beyond economic boundaries. The concept of social indicators followed that of economic indicators, and their development represents a more recent investigation.

A social indicator is defined as a "statistic of direct normative interest which facilitates concise, comprehensive and balanced judgment about the condition of major aspects of society" [DHEW 81]. Meier and Brudney provide a more instructive definition [MEIK 81, pp. 95-96]:

> An *indicator* is a variable that can be measured directly and is linked to a concept through an operational definition. An *operational definition* is a statement that tells the analyst how a concept will be measured.

Slightly different connotations are ascribed to social indicators by Carlisle [CARE 1972, p. 25]:

> (a social indicator is) the operational definition or part of the operational definition of any one of the concepts central to the generation of an information system descriptive of the social system.

The term "information system" is used far more restrictively here than commonly accepted in computer science.

Two important characteristics of social indicators are stressed by Carley [CARM 81, p. 2]:

(1) Social indicators are *surrogates* that do not stand by themselves – a social indicator must always be related back to the unmeasurable concept of which it is a proxy.

(2) Social indicators are concerned with information which is conceptually quantifiable, and must avoid dealing with information which cannot be expressed on some ordered scale.

A nagging problem is the establishment of the correlation between *unmeasurable* phenomena and *quantifiable* surrogates [CARM 81, p. 13].

4.2 Derivation of Software Quality Indicators (SQIs)

Cruickshank and Gaffney describe the use of quality indicators for assessment of the "goodness" of software structure and prediction of the cost to complete a design. Software design indicators belong to one of three classes [CRUR 80, pp. 1-2]:

(1) Structural Composition – abstract characteristics of the design independent of application or efficiency concerns.

(2) Degree of Design Detail – measurement of completeness of design and its satisfaction of application requirements.

(3) Resource Utilization – degree of utilization of memory and time.

The emphasis on design *prediction* and the concerns for resource requirements introduce major differences from the intended use of SQIs for methodology evaluation. Neither the second or third class defined above applies in methodology evaluation.

A concern for social applications that is rendered moot for SQIs is whether indicators should measure phenomena amenable to policy manipulation [CARM 77, p. 24]. Certainly, SQIs are applied to the result of a process that is totally subject to policy and operational control. Another concern is not so easily dismissed: In the role of value judgment, whose value of quality prevails in the absence of agreement [PALT 73, p. 7]?

4.2.1 The Concept of a Software Quality Indicator

Based on the review of social indicators summarized above and the prior work in quality factors and SQIs, we have adopted the following definition:

> A *software quality indicator* (SQI) 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 SQI *must* be

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

Further, a SQI *can* be

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

Finally, a SQI *should* be

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

4.2.2 Implications of Software Quality Measurement

The terms "measurement" and "quality" seem almost inconsonant. This perception stems from an inherent view of measurement as possible only in a *quantitative* sense which is to be distinguished from a *qualitative* sense. A few reflections on common uses of the terms "quality control" and "quality assessment" serve to dispel this notion. Product quality is commonly measured in quantitative terms and the search for quantitative means of assessing product quality continues.

Software development technology currently suffers from an indiscriminant use of terms describing software quality (attributes) confounded with terms describing project goals (objectives), which are interlaced with descriptives that apply to the development process (principles). The discrimination of objectives, principles, and attributes provides a long-needed clarification and organization of the terminology. More importantly, the existence of valid SQIs can provide the basis for specification of *quality levels for deliverable products.* Assuming that valid quality indicators can be formed from quantifiable characteristics of the code and documentation, then the potential exists for a totally automatic, procedural assessment of software quality. That possibility is extremely exciting and warrants further research.

4.3 Unresolved Issues in the Development of Software Quality Indicators

Recognizing that a SQI is a defined to be a statistical concept, then the usual problems with error are encountered. That is, an indicator value is comprised of concept plus error. Consequently, statistical accuracy requires that a determined effort be made to define indicators with minimal error components.

$$\text{indicator} = \text{concept} + \text{error}$$

The challenge lies in the identification of indicators that apply to the concept and can be measured with relatively small error such that

$$\lim (\text{error} \mid n \to \infty) = 0$$

where n is the sample size. The validation of social indicators has proved difficult, and the validity of SQIs appears to be an issue that has received little attention from the software engineering community. Such an investigation must form a prominent role in the use of SQIs in methodology evaluation or in quality assurance of software products. Review of the different approaches to validating *social indicators* would appear to be a logical point of departure.

A second major issue is the inherent temporal nature of most "quality indicators" – their values can change and sometimes rapidly. Sampling so as to account for temporal variations, induced or potential autocorrelation, and relative versus absolute scaling in time are subissues that must be addressed for SQIs.

The elimination of subjectivity is likely to be impossible, but the reduction of dependency on human judgment, and the concommitant variability, is a paramount concern. Basic statistical data drawn directly from the documentation and code provide objective but often inadequate candidates for indicators. Composites formed from "raw" data, constructed from intuitive perceptions, are difficult to confirm by literature references. As noted above, little work in SQI validation is to be found. The most appealing strategy perhaps lies in the strategy of redundancy: seek to have at least two indicators that reflect either the absence or presence of each attribute. More than two are preferable, and antithetic relationships (one indicator measures on an inverse scale to another) are especially desirable.

A final issue is the degree of independence that a SQI should exhibit from the semantic and syntactic constructs of specification or implementation languages. Ideally, language independence is the goal, but a more realistic expectation is to restrict the dependency to a manageable level that requires "customized interfaces."

5. Automating the Evaluation Process Through Product Evaluation

The literature verification and reassessment process discussed in Section 3 and the encouraging results coming from the SQI research imply that the procedural *approach* to evaluating software development methodologies is fundamentally sound. From an *operational* perspective, however, additional concerns have risen. In particular, critical operational issues include:

(1) the manpower requirements for analyzing a methodology and associated product(s), and

(2) the potential for introducing bias during the data collection phase.

The significance of these two issues is magnified when one considers that, although a methodology needs only be examined once for adequacy, product assessment is performed multiple times. Not only is this aspect of the evaluation process manpower intensive, it is also the most susceptible to human bias. One possible approach to dealing with the the manpower requirements and data bias is to automate the data collection phase of the evaluation process. Crucial to the automation based approach, however, is that the entity being analyzed must possess a detectable structure with which a meaning can be associated. The remainder of this section expands on the above statement, describes current approaches to automating product assessment, and discusses several difficulties surrounding the automated process.

In developing a software system two major components are always stressed as deliverables: the software code and the supporting documentation. It is natural to assume, therefore, that a complete product analysis must include an examination of both of these components. Similar views, although motivated from a system validation and verification perspective, are also expressed by Taylor and Osterweil [TAYR 78] and Hammond [HAML 78]. A closer look at code and documentation characteristics, however, reveals that product evaluation is facilitated by a further partitioning code and documentation components. The refined partitioning consists of (1) the source code language constructs (excluding comments), (2) documentation embedded in the source code via commenting facilities, and (3) documentation external to the source code, e.g. document

manuals. With respect to automated analysis, each of these components are individually discusses below.

5.1 Analyzing Source Code Language Constructs

From an automated perspective source code analysis is probably the most well understood. For example, compilers exploit formal parsing and translation techniques in their systematic analysis of programs [AHOA 86]. Additionally, software systems like SOFTDOC [SNEH 85], DAVE [OSTL 76] and Henry's Information Flow Analyzer [HENS 87] synthesize source code properties based on underlying language structures. In turn, these properties are used in detecting data flow anomalies, computing information flow metrics and so forth.

Although current technology does support automated techniques for extracting data elements based on language constructs, the validity of computational formulas utilizing those data elements, and the interpretation of ensuing results, are still open issues. For example, excessive nesting of control structure is perceived to adversely affect program complexity [CONS 86, ARTL 84]. The point at which excessive nesting occurs, however, remains unclear.

To summarize, automated techniques for analyzing source code language structures are well known. Moreover, the rapid prototyping of such analyzers can be achieved through support facilities like LEX and YACC [JOHS 78]. Hence, with respect to the evaluation procedure, an automated process supporting the acquisition of data pertinent to source code analysis is both feasible and practical with today's technology.

5.2 Document Generation: Two Approaches

Documentation is recognized as being inextricably bound to all facets of project development, ranging form conception, to design, to coding, testing, and so fort. As described by Tausworthe [TAUR 77]:

> "The goal of documentation is *communication*. During the project, documentation serves as a working vehicle to prevent distortion of ideas, promote project control, record design-

phase decisions, permit orderly sub-system development, and make the system visible, both in its capabilities as well as its limitations. When the project is complete, it records the history of development, serves as a tutorial guide to system operation, demonstrates that the system works, and provides a means for maintenance and evaluation of obsolete or amendable portions of the system."

Clearly, the importance of documentation cannot be understated. Yet, the literature abounds with articles describing the the need for better documentation techniques and facilities [HESS 81, SNEH 85, HOWE 86].

One particular aspect of documentation that is currently receiving significant attention focuses on the need for documentation which can be analyzed and restructured through automated processes. According to Osterweil, Brown and Taylor [OSTL 79] past efforts in automating the evaluation of documentation have failed because (1) written documents lack specification *rigor* and design formats, and (2) they are not written with machine readability as an objective.

Today, the two more prominent approaches to developing documentation that admits to automated analysis are where

(1) the user specifies documentation through an *automated* process tailored for a particular application domain and specification level, e.g. SADT [ROSD 77] and SREM [ALFM 85], and

(2) the user writes documentation based on a predefined set of constraints and guidelines.

Both approaches are considered below.

Assuming an automated document specification system (approach 1), the system enforces document specification rigor by guiding the user through a predefined set of paths that lead to a (hopefully) fully specified document. Moreover, because the document specification system maintains control of how and when information units are collected, the document can easily be

stored in a format that is machine readable and which facilitates automated analysis. Although this approach does provide a mechanism for producing software that amenable to automated analysis, because the user is forced to present document material in a predefined (and perhaps unnatural) order, *human* readability of the document is often sacrificed. That is, if the user is unable to deviate from a standard, predefined progression of information solicitations, information related to tangential but pertinent considerations can be lost. Moreover, documentation formats as well as content play an important in the human readability of a document; in general, document specification systems lack support for storing and disseminating non-textual information and often textual information in non-prose formats.

The second approach to producing documentation amenable to automated analysis is based on the documenter adhering to a predefined set of guidelines that embraces the necessary specification rigor. This approach provides the *flexibility* to produce a complex document that is both human readable and human understandable. Moreover, adherence to the specification guidelines insures a basis for machine readability and automatic analysis. For this approach to be effective, however, three elements must be present. First, the set of stipulated guidelines for developing and maintaining a document must be unambiguous and well-defined. The documenter must know *how* to present the information as well as *what* to present. Second, the guidelines and constraints must be designed with the human in mind. For example, imposing a complex traceability scheme is not appropriate; let a computer algorithm infer global traceability based on simple, local references. Finally, each documenter must strictly adhere to the documentation guidelines. Unfortunately, it is this third element that is the most difficult to insure and enforce.

5.3 Automating Documentation Analysis

The two approaches to documentation described in the above section lie at different ends of a spectrum: in one case, an automated process dictates and enforces documentation guidelines; in the other, a set guidelines are defined but it is up to the documenter as to the extent they are followed. Hybrid approaches reflecting the middle ground also exist. Although we recognize that

techniques will *always* exist for subverting documentation system dictates, we shall assume in the remainder of this section that a well-defined, effective documentation procedure used. With this assumption in mind, the appropriate questions now are:

(1) What elements are crucial in defining general documentation guidelines?

(2) With respect to the two major forms of documentation, i.e. source code comments and document manuals, to what extent are these elements known *a priori*?

(3) How do (1) and (2) affect automatic document analysis?

In response to the first question, three major elements must guide documentation synthesis:

(1) **The User of the Document.** Knowing who is going to read the document enables the documenter to determine the appropriate level of documentation detail, use the correct document vocabulary set and stress system aspects pertinent to reader.

(2) **Documentation Focus.** The focus of a document is defined to be the topic or subject of the document contents. In general, each document should have a *single* subject emphasis. A subject can, of course, be described as a sequence of related subtopics.

(3) **Documentation Purpose.** The purpose of a document is viewed as the framework that surrounds and guides a description of the document focus. For example, a document might focus on a particular subject while its purpose is to provide an overview of that subject.

Intuitively, *a priori* knowledge about the user of a document, the document's focus and its purpose should lead to documentation that clearly and consistently reflect collective characteristics stressed by *all* three elements. Based on this assertion and in response question (2) posed above, the following two subsections discuss source code comments and document manuals relative to a document's user, focus and purpose.

Source Code Comments

Consider the role of code comments in the documentation process. According to Munson [MUNJ 78] source code comments should describe the functionality of a module, it limitations, basic assumptions, algorithmic/implementation logic and so forth. With respect to the three elements that guide documentation synthesis the following observations are made.

(1) Comments are embedded in the source code. Implicit in this observation is that the reader (or *user*) of the source code documentation possesses a certain level of knowledge, i.e. that of a software engineer.

(2) Comments describe external module relationships and internal source code characteristics. That is, comments *focus* on documenting code and module characteristics.

(3) Documentation comments appear in three formats: header comments, block comments and single line comments. Additionally, the *purpose* of each format is well-defined. For example, header comments describe the computer program name, input/output information, assumptions, and limitations and/or restrictions; block comments are used to describe sections of code.

From the above observations one can surmise that, when documenting via code comments, the reader (or user), focus and purpose of the documentation are know before the actual documentation content is specified.

Documentation Manuals

Unlike source code comments, when one considers general documentation manuals, the user, focus and purpose are not so well defined. In general, when one writes a document one first determines the user group and then appropriately constrains the subject matter (or focus) of the document and its presentation framework (or document purpose). Hence, with respect to both the source code documentation and documentation manuals, three crucial ingredients must

be identified before product documentation can be synthesized: knowledge about the document user, the focus of the document and its purpose. Only source code comments, however, possess characteristics that assume knowledge about these three ingredients; for documentation manuals, the focus and purpose are predicated on first defining the user group. Based on these observations a more appropriate wording of the third question posed above is:

> How does (a) knowing that documentation must be written with a focus and purpose that reflects the intended user community and (b) knowing that only source code comments presuppose such knowledge affect the automatic the analysis of documentation?

In addition to answering this question, the following discussion also addresses a more general question – why is an automated approach to documentation analysis so difficult?

As alluded to earlier in this report and substantiated by Lamb [LAMS 78] and Taylor [TAYR 78], problems with the automatic analysis of documentation stem from the lack of well defined rules and procedures for developing documentation. In particular, if structure rules are lacking then implied as well as explicit relationships among documentation elements will be difficult if not impossible to detect. We note, however, that general guidelines designed to support the automatic analysis of source code documentation have been defined, see for example Munson [MUNJ 78]. Unfortunately however, general guidelines exist primarily for source code documentation and not for external documentation, i.e. text manuals. To understand why this situation prevails we must examine the relationship between documentation and *a priori* knowledge about the document's intended user, focus and purpose. In the case of source code documentation, (a) the user community is well defined and assumed to be static, i.e., programmers, analysts and software engineers, (b) the purpose of each documentation unit is well defined and limited to three major categories, i.e. header, block and single-line comments, and (c) the focus of each documentation unit is a strict function of a program's source code units relative to the three categories described in (b). In the case of a documentation manual, however, none of this information is known *apriori*... the user community must be determined first, followed by the documents purpose

and focus. Although documentation guidelines have been devised based on selected user groups, documentation focus and purpose, e.g. SADT's SA language and SREM's REVS, guidelines supporting the structured synthesis of *general* documents suitable for automated analysis do not exist. It is the authors' opinion that this situation prevails because (a) defining one set of general purpose documentation guidelines tailored for a variety of user groups is extremely difficult, and (b) until now, the motivation for automatic document preparation and analysis has been lacking. In particular, the complexity of today's software systems ***demand*** up-to-date documents supporting system overviews, functional capabilities, usage and maintenance aspects. Pioneering efforts by researcher like Hester and Parnas [HESS 81], Sneed [SNEH 85] and Horowitz [HOWE 86], however, are providing significant insights into the functional partitioning of documentation as well as experimental approaches for systematically storing, retrieving and automatically analyzing documents.

In summary, past efforts to produce documentation that is machine readable and which contains sufficient structural information for local and global characteristic analysis have largely failed. The results of a literature survey indicate that this trend is expected to continue until proper documentation guidelines are defined and adherence to them are enforced. Although enforcement can be controlled, the authors believe that the definition of general purpose documentation guidelines is inextricably bound to knowing the intended user of a document, the documents focus and it purpose. Because these elements can vary widely, defining a universal set of documentation guidelines that supports automated analysis is difficult. Nonetheless, documentation guidelines assuming a homogeneous user community are being proposed. Documentation systems that enforce these guidelines have met with limited success. As outlined in the next section, however, general documentation guidelines founded on "relational objects" may provide a basis for defining general documentation standards and guidelines.

6. Tangential Research Issues

In the process of investigating the principle issues defined in this research effort, additional topics have surfaced and deserve discussion. This section briefly describes those topics and outlines associated research issues.

Applying the Evaluation Procedure to Embedded Systems

The current evaluation procedure provides a method for assessing the adequacy and usability of general software development methodologies. Embedded systems, however, mandate real-time constraints that stress specialized product structures and characteristics. For example, in embedded systems applications software/software and software/hardware communications is often accomplished through tailored synchronization constructs. These specialized communication constructs are crucial to communicating processes and provide a mechanism for specifying and implementing well-defined interfaces. Although a well-defined interface is a highly desirable product attribute and an integral part of the evaluation procedure, determining the extent to which an embedded system reflects this attribute is difficult within the scope of the existing evaluation procedure. The present difficulty lies with the fact that well-defined interface assessment is based on "standard" communication mechanisms like procedure calls and global variables; synchronization constructs like ADA's entry/accept statements are not incorporated. Similarly, consider correctness, testability and decomposition; related embedded systems characteristics that dictate special consideration include timing (temporal) constraints and concurrency. Recognizing and Assessing such characteristics require the identification of additional product properties (or indicators) specific to embedded systems and the definition of linkages that reflect appropriate relationships.

With respect to the evaluation procedure, the above paragraph outlines three instances where embedded systems dictate special considerations; certainly more exist. Additional research is needed, however, to identify these special considerations and incorporate them into an evaluation procedure tailored to the embedded systems domain.

Design Versus Code Level Factors

Fundamental to the concept of SQIs is the attribute/factor binding. That is, a factor by itself has no real meaning; it must be always be viewed with respect to a particular attribute. For example, the number of structure data types is a factor; attached to the attribute, well-defined

interfaces, one has a meaningful SQI that imparts what is being measured relative to what one is attempting to assess. While investigating two particular attributes, coupling and cohesion, the associated factors appeared to partition along design/code boundaries. That is, factors bound to coupling can be traced back to design decisions; factors bound to cohesion, however, appear to have their origin tied to the coding process. For example, the use of call statements with respect to coupling reflects decisions made during design phase; the use of structured code as it relates to cohesion is a decision made at implementation time.

Although no significance has yet been attached to the above observations, the partitioning does indicate a possible SQI classification scheme based on life cycle phases. Moreover, if other such partitionings exist, and they fit within some natural framework, then perhaps the partition classes will provide insights into the identification of additional factors and SQIs.

Automatic Documentation Analysis Based on Relational Objects

As discussed in Section 5, automated documentation analysis requires adherence to a rigid structure/content format. Although it is natural to view such constraints with respect to the user of document, the document's focus and purpose, a compromise approach based on a predefined relational network is envisioned. The approach does not consider the user or document content, but simply a predefined set of relations that can be imposed on documentation objects (i.e. documentation segments). Two such relations might be *subsumes* and *preceded*. Object B is said to be subsumed by object A if object B describes an element of A in more detail. The "subsumption" relationship is one-to-many and would reflect a hierarchical relationship where documentation object A is an overview for more detailed documentation objects B, C, and D. Hence, A subsumes B, A subsumes C, and A subsumes D. The "subsumption" relationship denotes a natural relationship that captures the hierarchical nature of documentation segments among and within documents. The "precedes" relationship, on the other hand, reflects a horizontal linkage between documentation objects. For example, describing an Input/Process/Output relationship might be expressed as Input precedes Process and Process precedes Ouput. In general, object A precedes object B. Similar to

subsumption, the "precedes" relationship denotes a natural relationship one would expect to find between documentation segments.

The power of this relational approach is that relationships can be defined knowing only *how* documentation is structured. For appropriately defined documents information retrieval, update and analysis can be supported by a conventional relational database system. As initially envisioned, only documentation *segments* are relationally bound. Word and phrase match algorithms would be used to verify defined relationships among documentation objects. As mentioned in the opening paragraph, however, this approach offers only a compromise solution to a difficult problem. The compromise approach does not offer the complete analytical capabilities that a more structure and content oriented system might. On the positive side, however, one is free to structure a document as desired - realizing of course, that the amount of imposed structure and defined relationships are directly related to the the analytical qualities of the document.

Automatic Hypothesis Synthesis and Verification

The increasing complexity of today's software systems is forcing a hard look at current documentation practices. Documentation trends are moving toward more structured formats that support automated analysis. Based on the assumption that automatic analysis is feasible, an additional research issue stemming from the current task involves automatic hypothesis synthesis and verification. That is, we believe that it should be possible to formulate an hypothesis based on source code analysis and then follow the hypothesis up the documentation chain to a point to where the hypothesis can either be accepted or rejected. For example, based on source code comments and file primitives, one can determine external files accessible to a given module. In turn, this accessibility implies specific interface capabilities and characteristics. The validity of such privileges can then be verified against interface requirements documents.

Once an hypothesis has been formulated a "bottom-up" analysis of code and documentation leads to its acceptance or rejection. If a hypothesis is rejected, then a related "top-down" document-to-source code analysis is deemed appropriate. The top-down approach provides a mechanism for

ascertaining *where* document specifications begin to deviate from what is actually present in the source code product. It must be stated, however, that this approach is highly dependent not only on the capability to automate source code and documentation perusal, but also assessment through precise defined and implied relationships.

7. Summary and Conclusions

Three major issues are addressed in this research effort:

Issue 1: Assessing the strengths and weaknesses of the current evaluation procedure,

Issue 2: Basing the evaluation procedure on software quality indicator, and

Issue 3: Automating the evaluation process.

The overall thrust of this research has been to examine the validity of the assumptions intrinsic to the evaluation procedure, reassess fundamental relationships and characteristics, refine the evaluation procedure based on perceived inadequacies and reassessment results, and explore appropriate measures for automating the evaluating process. The major results stemming from this research effort are described below.

For Issue 1, assessing the strengths and weaknesses of the evaluation procedure, the authors initiated a literature search to (a) identify other approaches to evaluating software development methodologies and contrast them with the procedural approach, and (b) validate and reassess assumptions concerning the selected sets of objective, principles and attributes as well as the linkages defined among them. The literature survey revealed that the selected set of methodological objective, principles and attributes are consistent with those perceived most important by other researchers in the software engineering domain. Even more significant, however, is that substantiating references to *every* objective/principle and principle/attribute linkage was found. Moreover, references supporting all but a few of the property/attribute were also discovered. From the reassessment perspective, the principle of abstraction has been added to the evaluation procedure and

issues concerning "separation of concerns" have been resolved or satisfactorily clarified. The authors were unable to find, however, articles that described techniques for evaluating single software development methodologies – all related papers primarily focussed on comparing one methodology to another.

In regard to Issue 2, basing the evaluation procedure on statistical indicators, significant strides have been made toward the formal definition and integration of software quality indicators (SQI). Based on statistical indicators, the definition of SQIs has been formulated within a well-defined, defensible framework. SQIs are expected to play an integral role in automating the evaluation process.

Finally, addressing Issue 3 has resulted in a better understanding of where the major problems lie in automating the evaluation procedure. In addition to integrating SQIs into the evaluation procedure and formalizing code analysis semantics, it is obvious that we must also address the under-publicized, yet substantially difficult task of automating documentation analysis. Future research efforts, however, can build on the newly acquired results that link the document user, focus and purpose to document synthesis as well as exploit the possibility of defining relations among documentation objects.

References

[ALFM 85] Alford, M., "SREM at the Age of Eight; The Distributed Computing Design System," *IEEE Computer*, Vol. 18, No. 4, April 1985, pp. 36-54.

[AHOA 86] Aho, A., Sethi, R. and Ullman, J., *Compilers: Principles, Techniques, and Tools,* Addison-Wesley Publishing Co., Reading, MA, 1986.

[ARTJ 76] Arthur, J., Nance, R. and Henry, S., "A Procedural Approach to Evaluating Software Development Methodologies: A Foundation," Technical Report TR-86-24, Virginia Tech, 1986.

[ARTL 84] Arthur, L., *Measuring Programmer Productivity and Software Quality,* pp. 153, John Wiley and Sons, Inc., New York, NY, 1984.

[BASV 84] Basili, V. and Perricone, B., "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM,* Vol. 27, No. 1, pp. 42-52, January 1984.

[BAUF 72] Bauer, F., "Software Engineering," *Information Processing 71*, North Holland Publishing Company, 1972.

[BERG 81] Bergland, G., "A Guided Tour of Program Design Methodologies," *IEEE Computer,* pp. 13-36, October, 1981.

[BOEB 76] Boehm, B., "Software Engineering," *IEEE Transactions on Computers*, Vol. C-25, No. 12, December, 1976, pp.1226-1241.

[BOHC 66] Bohm, C., and Jacopini, G., "Flow Diagrams, Turing Machines, and Languages with only Two Formation Rules," *Communications of the ACM,* Vol. 19, No. 5, May 1966.

[CARM 77] Carley, M. "Needs and Outputs," in *Fundamentals of Social Administration,* Weisler, E. (ed.), MacMillan, 1977.

[CARM 81] Carley, M. *Social Measurement and Social Indicators,* George Allen & Unwin, 1981.

[CARE 72] Carlisle, E. "The Conceptual Structure of Social Indicators," in *Social Indicators and Social Policy*, Shonfield, A. and Shaw, S. (eds.), Hinemann Educational Books, 1972.

[CLEP 84] Clements, P., "Function Specifications for the A-7E Function Driver Module," NRL Memorandum Report 4658, Naval Research Laboratory, Washington, D. C., November, 27, 1984.

[CONS 86] Conte, S., Dunsmore, H. and Shen, V., *Software Engineering Metrics and Models,* pp. 74, Benjamin Cummings, Reading MA, 1986.

[COLM 82] Colter, M., "Evolution of the Structured Methodologies," *Advanced Systems Development / Feasibility Techniques,* pp. 74-95, John Wiley and Sons Inc., New York, NY, 1982.

[CRUR 80] Cruickshank, R. and Gaffney, J. "Indicators for Software Design Assessment," IBM FSD Final Report on Creative Development Task 91, January 23, 1980.

[DHEW 81] U. S. Department of Health, Education and Welfare, *Toward a Social Report,* U. S. Government Printing Office, 1969.

[DIJE 68] Dijkstra, E., "Go To Statement Considered Harmful," *Communications of the ACM,* Vol. 11, No. 3, pp. 149-150, March 1968.

[DIJE 72] Dijkstra, E., "Notes on Structured Programming," *Structured Programming,* Dahl, O-J, Dijkstra, E., Hoare, C. A. R., Academic Press, 1972.

[DIJE 76] Dijkstra, E., *A Discipline of Programming,* Prentice-Hall, 1976.

[GILP 82] Gilbert, P. *Software Design and Development*, SRA, 1982.

[GOMH 84] Gomaa, H. "A Software Design Method for Real-Time Systems," *Communications of the ACM*, Vol. 27, No. 9, September 1984, pp. 938-949.

[GOON 87] Goodwin, N. "Functionality and Usability," *Communications of the ACM*, Vol. 30, No. 3, March 1987, pp. 229-233.

[HAML 78] Hammond, L., Murphey, D. and Smith, M., "A System for Analysis and Verification of Software Design," *Proceedings of COMPSAC 78,* Chicago, IL, pp. 42-47, 1978.

[HENK 78] Heninger, K., Kallander, J., Shore J. and Parnas, D., "Software Requirements for the A-7E Aircraft," NRL Memorandum Report 3876, Naval Research Laboratory, Washington, D. C., November, 1978.

[HENS 85] Henry, S., Arthur, J. and Nance, R., "A Procedural Approach to Evaluating Software Development Methodologies," Technical Report TR-85-20, The Department of Computer Sciences, Virginia Tech, March 1985.

[HENS 87] Henry, S., "A Technique for Hiding Proprietary Details While Providing Sufficient Information for Researchers," *Journal of Systems and Software,* to appear.

[HESS 81] Hester, S., Parnas, D. and Utter, D., "Using Documentation as a Software Design Medium," *The Bell System Technical Journal,* Vol. 60, No. 8, pp. 1941-1977, October, 1981.

[HOAC 72] Hoare, C. A. R. "Notes on Data Structuring," *Structured Programming,* Dahl, O-J., Dijkstra, E., and Hoare, C. A. R. Academic Press, 1972.

[HOWE 86] Howoritz, E. and Williamson, R., "SODOS: A Software Documentation Support Environment – Its Definition," *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 8, pp. 849-859, August 1986.

[INFO 79] Infotech, "Software Engineering Techniques," *Infotech State of the Art Report,* Infotech International, Maidenhead, England, 1979.

[JOHS 78] Johnson, S. and Lesk, M., "Language Development Tools," *Bell System Technical Journal,* Vol. 57, No. 6., pp. 2155-2175, 1978.

[KEAJ 86] Kearney, J., Sedlmeyer, R., Thompson, W., Gray, M., and Adler, M., "Software Complexity Measurement," *Communications of the ACM*, Vol. 29, No. 11, November 1986, pp. 1044-0150.

[LAMS 78] Lamb, S., Leck, V., Peters, L. and Smith, G., "SAMM: A Modelling Tools for Requirements and Design Specification," *Proceedings of COMPSAC 78,* pp. 48-53, Chicago IL, 1978.

[MEIK 81] Meier, K. and Brudney, J. *Applied Statistics for Public Administration,* Duxbury Press, 1981.

[MUNJ 78] Munson, J., "Software Maintainability: A Practical Concern for Life Cycle Costs," *Proceedings of COMPSAC 78,* pp. 54-59, Chicago, IL, 1978.

[NANR 85] Nance, R., Arthur, J. and Dandekar, A., "Evaluation of Software Development Methodologies," A Final Report of the Immediate Software Development Project, The Department of Computer Sciences, Virginia Tech, December 1985.

[NRL 84] Naval Research Laboratory, "Software Cost Reduction Methodology," Brief 2, Information Technology Review, 4 October, 1984.

[OSTL 76] Osterweil, L. and Fosdick, L., "DAVE – A Validation, Error Detection and Documentation System for FORTRAN Programmers," *Software: Practice and Experience,* Vol. 6, No. 4, pp. 473-486, October/December 1976.

[OSTL 79] Osterweil, L., Brown, J. and Stucki L., "Asset: A Life Cycle Verification and Visibility System," *Journal of System and Software,* Vol. 1, No. 1, pp. 77-86, 1979.

[PALT 73] Palys, T. "Social Indicators of Quality of Life in Canada: A Practical Theoretical Report," Department of Urban Affairs, Manitoba, Winnipeg, 1973.

[PARD 85] Parnas, D., "Software Aspects of Strategic Defense Systems," *Communications of the ACM*, Vol. 28, No. 12, December 1985, pp. 1326-1335.

[PETL 77] Peters, L. and Tripp, L., "Comparing Software Development Methodologies," *Datamation,* Vol. 23, No. 11, pp. 89-94, October 1977.

[ROSD 75] Ross, D., Goodenough, J. and Irvine, C., "Software Engineering: Process, Principles, and Goals," *IEEE Computer*, Vol. 8, No. 5, pp. 17-27, May 1975.

[ROSD 77] Ross, D., "Structured Analysis: A Language for Communicating Ideas," *IEEE Transactions on Software Engineering*, January, 1977.

[SHEV 85] Shen, V., Yu, J. and Thebaut, S., "Identifying Error Prone Software – An Empirical Study," *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 4, pp. 317-323, April 1985.

[SNEH 85] Sneed, H. and Merey, A., "Automated Software Quality Assurance," *IEEE Transactions on SOftware Engineering,* Vol. SE-11, No. 9, pp. 909-916, September, 1985.

[SOFT 76] *An Introduction to SADT: Structured Analysis an Design Technique* SoftTech Inc., Document No. BCS-10167, 1976.

[TAYR 78] Taylor, R. and Osterweil, L., "A Facility for Verification, Testing, and Documentation of Concurrent Software," *Proceedings of COMPSAC 78,* Chicago, IL, pp. 36-41, 1978.

[TAUR 77] Tausworthe, R., *Standardized Development of Computer Software,* Prentice-Hall, Englewood Cliffs, NJ, 1977.

[WEID 85] Wiess, D. and Basili, V., "Evaluating Software Development by Analysis of Change: Some Data form the Software Engineering Laboratory," *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 2, pp. 157-168, February 1985.

Appendix 1

An Overview of the Linkage Foundation

As discussed in [ARTJ 86] a broad review of software engineering literature leads to the identification of seven objectives commonly recognized in the numerous methodologies:

(1) Maintainability – the ease with which corrections can be made to respond to recognized inadequacies,

(2) Correctness – strict adherence to specified requirements,

(3) Reusability – the use of developed software in other applications,

(4) Testability – the ability to evaluate conformance with requirements,

(5) Reliability – the error-free use of software performance over time,

(6) Portability - the ease in transferring software to another environment, and

(7) Adaptability – the ease with which software can accommodate to change.

Achievement of these objectives comes through the application of principles supported (encouraged, enforced) by a methodology. The principles enumerated below are associated with the creative process by which programs and documentation are produced:

(1) Hierarchical Decomposition – components defined in a top-down manner.

(2) Functional Decomposition – components are partitioned along functional boundaries.

(3) Information Hiding – insulating the internal details of component behavior.

(4) Stepwise Refinement – utilizing a convergent design.

(5) Structured Programming – using a restricted set of control constructs.

(6) Documentation – management of supporting documents (system specifications, user manual, etc.) throughout the life cycle.

(7) Life Cycle Verification – verification of requirements throughout the design, development, and maintenance phases of the life cycle.

The enunciation of objectives should be the first step in the definition of a software development methodology. Closely following is the statement of principles that, employed correctly, lead to the attainment of these objectives. The important correspondence between objectives and principles is shown in Table 1.

Employment of well-recognized principles should result in software products possessing attributes considered to be desirable and beneficial. A brief definition of each attribute is provided below.

(1) Cohesion - the binding of statements within a software component.

(2) Coupling - the interdependence among software components.

(3) Complexity - an abstract measure of work associated with a software component.

(4) Well-defined Interface - the definitional clarity and completeness of a shared boundary between a pair of software components.

(5) Readability - the difficulty in understanding a software component (related to complexity).

(6) Ease of Change - software that accommodates enhancements or extensions.

(7) Traceability - the ease in retracing the complete history of a software component from its current status to its design inception.

(8) Visibility of Behavior - the provision of a review process for error checking.

(9) Early Error Detection - indication of faults in requirement's specification and design prior to implementation.

The relationships among attributes and principles are denoted in Table 2.

Software attributes represent a collective and subjective judgment of a characteristic. This judgment can be made more objective and quantifiable by the definition of *properties* that reflect the

Table 1: Software Engineering Objectives and Corresponding Principles

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

presence or absence of the attribute. For example, the use of subordinate indentation to demarcate control statement ranges contributes to more readable code. The utilization of block structuring and the establishment of conventions for procedure invocations improve the coupling/cohesion among components. However, as is evident in these two examples, such indicators are highly dependent on the programming language employed in implementation, documentation organization, and

Table 2: Software Engineering Principles and the Effects on Derived Attributes

Principle	*Effect on Attribute*
Stepwise Refinement	Coupling/cohesion enhanced Reduced complexity
Hierarchical Decomposition	Ease of Change Coupling/cohesion enhanced Reduced complexity
Functional Decomposition	Ease of Change Coupling/cohesion enhanced Reduced complexity
Structured Programming	Reduced Complexity Readable code
Information Hiding	Extensible software Coupling/cohesion enhanced Reduced complexity Well-defined interfaces
Documentation	Readable code Traceability Ease of Change Reduced complexity
Life-cycle Verification	Visibility of behavior Early error detection

formating conventions; therefore, the definition of indicators cannot be generalized but remains specific to the instances of evaluation.

Appendix 2

LINKAGES BETWEEN OBJECTIVES & PRINCIPLES

Objectives and principles and how they are related

In the following document (*) against a principle indicates identification of reference for the linkage.

ADAPTABILITY

Achieved by:

(a) Hierarchical Decomposition (*)

(b) Functional Decomposition (*)

(c) Information Hiding (*)

(d) Stepwise Refinement (*)

(e) Structured Programming (*)

(f) Documentation (*)

References:

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

(ii) Arthur, J.L. (1984), *Measuring Programmer Productivity and Software Quality,* John Wiley, New York, NY.

(iii) Ross, D.T., J.B. Goodenough and C.A. Irvine (1975), "Software Engineering: Process, Principles and Goals," *IEEE Computer,* Vol. 8, No. 5, pp 17-27.

(iv) Gilbert, P. (1983), *Software Design and Development,* Science Research Associates, Chicago, IL.

(v) Bergland, G.D. and R.D. Gordon (1981), "Software Design Strategies," *IEEE Compsac,* pp 79-92.

(vi) Bullen, R.H. (1976), "Program Modularization," *Structured Programming,* Infotech State of the Art Report, England.

NOTES

1. PP21/(iii): In general, hierarchical decomposition is cited as helping to improve software reliability, helping to allow multiple use of common designs and programs, and helping to make it easier to modify programs. {Hierarchical Decomposition}

2. PP82/(vi): Functional decomposition increases inherent modifiability of a system. {Functional Decomposition}

3. PP272/(iv): The capability of hiding details of data elements from other modules is a principal reason for the popularity of DBMS. Once the data elements have been have been isolated, details of the structure of files and tables can be changed as necessary without affecting the collection of programs. The data base management modules can also change the format of requested data items to meet the needs of the requesting program. Thus isolation of data promotes flexibility.{Information Hiding}

5. PP226/(i): The degree of modularity obtained by using stepwise refinement determines the ease or difficulty with which a program can be adapted to changes or extensions of the purpose or changes in the environment (language, computer) in which it is expected. {Stepwise Refinement}

6. PP416/(iv): Stepwise refinement is a design strategy that seeks to implant ease of understanding, and ease of maintenance and modification into the program text. {Stepwise Refinement}

7. PP403/(iv): Structured programming (SP) uses only three forms (sequence, choice and iteration). Any programming calculation can be programmed using these forms. And when only these forms are used, the resulting program is easily understandable to programmers seeking it for the first time. There are fewer intricacies and "clevernesses" to be puzzled through. Thus maintenance personnel can discover quickly which portions of the program require maintenance or modification. {Structured Programming}

8. PP193/(ii): Flexibility is a function of modularity, self-documentation etc. {Documentation} Modularity provides a structure of highly independent modules, having sharply defined interfaces that are tolerant to external changes. A module possessing func-

tional strength and data coupling has little to fear from the outside and its internal logic is often so easily understood that enhancing the module becomes simple. Self-documentation identifies the software attributes that explain the function of the software. Without a clear understanding of the code, the programmer will have a hard time identifying where to insert new functions or to change or delete old ones.

MAINTAINABILITY

Achieved by:

(a) Hierarchical Decomposition (*)

(b) Functional Decomposition (*)

(c) Information Hiding (*)

(d) Stepwise Refinement (*)

(e) Structured Programming (*)

(f) Documentation (*)

References:

(i) Wasserman, A.I. (1976), "On the Meaning of Discipline in Software Design and Development," In: P. Freeman and A.I. Wasserman (eds.), *Tutorial on Software Design Techniques,* IEEE Computer Society Press, Long Beach, CA.

(ii) Hosier, J. (1978), *Structured Analysis and Design,* Infotech international Ltd., England.

(iii) Ross, D.T., J.B. Goodenough and C.A. Irvine (1975), "Software Engineering: Process, Principles and Goals," *IEEE Computer,* Vol. 8, No. 5, pp 17-27.

(iv) Pressman, R.S. (1982), *Software Engineering: A Practitioner's Approach,* McGraw-Hill, New York, NY.

(v) Gilbert, P. (1983), *Software Design and Development,* Science Research Associates, Chicago, IL.

NOTES

1. PP21/(iii): In general, hierarchical decomposition is cited as helping to improve software reliability, helping to allow multiple use of common designs and programs, and helping to make it easier to modify programs. {Hierarchical Decomposition}

2. PP37/(ii): Another advantage to be achieved through the reduction of complexity by functional decomposition is that maintainability is enhanced. However, this benefit arises when decomposition is carried out in the appropriate way. {Functional Decomposition}

3. PP157/(iv): The use of information hiding as a design criteria provides greatest benefits when modifications are required during testing and later, during software maintenance. {Information Hiding}

4. PP416/(v): Stepwise refinement is a design strategy that seeks to implant ease of understanding, and ease of maintenance and modification into the program text. {Stepwise Refinement}

5. PP403/(v): Structured programming (SP) uses only three forms (sequence, choice and iteration). Any programming calculation can be programmed using these forms. And when only these forms are used, the resulting program is easily understandable to programmers seeking it for the first time. There are fewer intricacies and "clevernesses" to be puzzled through. Thus maintenance personnel can discover quickly which portions of the program require maintenance or modification. {Structured Programming}

6. PP77/(i): A good design document is not only essential for writing the program, but also for the maintenance of the program over time. {Documentation}

CORRECTNESS

Achieved by:

(a) Hierarchical Decomposition (*)

(b) Stepwise Refinement (*)

(c) Structured Programming (*)

(d) Life-cycle verification (*)

References:

(i) Freeman, P. (1976), "Software Reliability and Design: A Survey," In: P. Freeman and A. I. Wasserman (eds.), *Tutorial on Software Design Techniques,* IEEE Computer Society Press, Long Beach, CA.

(ii) Bergland, G.D. and R.D. Gordon (1981), "Software Design Strategies," *IEEE Compsac,* pp 79-92.

(iii) Fairley, R.P. (1985), *Software Engineering Concepts,* McGraw-Hill, New York, NY.

NOTES

1. PP105/(i): Construct programming is a term applied to any programming method that attempts to produce correct programs without the usual testing and debugging phases. Structured programming, top-down programming and stepwise refinement are all constructive approaches that result in programs that are substantially more correct than ones produced in less organized ways. {Structured Programming, Hierarchical Decomposition, Stepwise Refinement}

2. PP80/(ii): The three basic constructs (sequence, iteration and selection) form the basis for writing more understandable, correct programs. {Structured Programming}

3. PP267/(iii): Life-cycle verification activity helps assess and improve the quality of the work products generated during development and modification of software. Quality attributes of interest include correctness, reliability, usefulness, usability, effi-

ciency, conformance to standards and overall cost effectiveness. {Life-cycle Verification}

REUSABILITY

Achieved by:

(a) Hierarchical Decomposition (*)

(b) Functional Decomposition (*)

(c) Information Hiding (*)

(d) Documentation (*)

References:

(i) Stevens, W.P. (1981), *Using Structured Design,* John Wiley, New York.

(ii) Ross, D.T., J.B. Goodenough and C.A. Irvine (1975), "Software Engineering: Process, Principles and Goals," *IEEE Computer,* Vol. 8, No. 5, pp 17-27.

(iii) Lanergrn, R.G. and D.K. Dugan (1981), "Software Engineering with Reusable Designs and Code," *IEEE Compcon Digest of Papers,* Fall Conference, pp. 296-303.

(iv) Arthur, L.J. (1985), *Measuring Programmer Productivity and Software Quality,* John Wiley, New York, NY.

(v) Peters. L.J. (1981), *Software Design: Methods & Techniques,* Yourdon Press, New York, NY.

(vi) Freeman, P. (1983), "Reusable Software Engineering: Concepts and Research Directions," In: P. Freeman and A.I. Wasserman (eds), *Tutorial on Software Design Techniques,* IEEE Computer Society Press, Silver Spring, MD.

(vii) Ramamoorthy, C.V., A. Prakash, W. Tsai and Y. Usuda (1984), "Software Engineering: Problems and Perspectives," *IEEE Computer,* Vol. 17, No. 10, pp. 191-210.

NOTES

1. PP21/(iii): In general, hierarchical decomposition is cited as helping to improve software reliability, helping to allow multiple use of common designs and programs, and helping to make it easier to modify programs. {Hierarchical Decomposition}

2. PP199/(i): Documentation can benefit from increased flexibility, (re)usability and easier maintenance. {Documentation}

3. PP296/(iii): Techniques of functional modularity are employed to prepare modules for use in multiple applications. {Functional Decomposition}

4. PP138/(iv): Reusability is a function of modularity, self-documentation. Modularity provides a structure of highly independent modules. Typically single-function, well-structured modules possessing functional strength and data coupling are reusable. {Documentation, Functional Decomposition, Hierarchical Decomposition}

   Self-documentation helps explain the function of the software. For any software to be reusable, it must be documented. {Documentation}

5. PP180/(v): The resulting modules (using information hiding) would be simple, reusable, and easier to test, integrate, and maintain. {Information Hiding}

6. PP71/(vi): Reusability is enhanced by documenting the system adequately. As the range and extent of computer applications rapidly increases, a simple way to facilitate their construction is by providing some well-documented models of existing generic systems. {Documentation}

7. PP204/(vii): Standardization of software resources is necessary to permit engineers to design the target system for reusability. It is also more important, however, to standardize the interface than the programming styles or codes. Once the interface is fixed we can ignore all the details inside each module, according to Parnas' principle of information hiding. {Information Hiding}

TESTABILITY

Achieved by:

(a) Hierarchical Decomposition (*)

(b) Functional Decomposition (*)

(c) Information Hiding (*)

(d) Stepwise Refinement (*)

(e) Structured Programming (*)

(f) Life-cycle Verification (*)

References:

(i) De Marco, Tom (1979), *Concise Notes on Software Engineering,* Yourdon Press, New York, NY.

(ii) Pressman, R.S. (1982), *Software Engineering: A Practitioner's Approach,* McGraw-Hill, New York, NY.

(iii) Fairley, R.P. (1985), *Software Engineering Concepts,* McGraw-Hill, New York, NY.

(iv) Lanergan, R.G. and D.K. Dugan (1981), "Software Engineering with Reusable Designs and Code," *IEEE Compcon Digest of Papers,* Fall Conference, pp. 296-303.

(v) Basili V.R. and A.J. Turner (1975), "Iterative Enhancement: A Practical Technique for Software Development," *IEEE Transactions on Software Engineering,* V01. SE-1, No. 4, pp. 390-396.

NOTES

1. PP157/(ii): The use of information hiding as a design criteria provides greatest benefits when modifications are required during testing and later, during maintenance. {Information Hiding}

2. PP39/(i): Boehm, Jacopini, Dijkstra and Warnier introduced structured coding to describe the idea of building programs using limited control structures in order to enhance readability and ease of testing. {Structured Programming}

3. PP144/(iii): A hierarchical structure isolates software components and promotes ease of understanding, implementation, debugging, testing, integration and modification of a system. {Hierarchical Decomposition}

4. PP267/(iii): Life-cycle verification activity helps assess and improve the quality of the work products generated during development and modification of software. Quality attributes of interest include correctness, reliability, usefulness, usability, efficiency, conformance to standards and overall cost effectiveness. {Life-cycle Verification}

5. PP300/(iv): Functional modularity facilitates walk-throughs, testing, debugging and maintenance activities. {Functional Decomposition}

6. PP390/(v): Iterative enhancement is a practical means of applying stepwise refinement. This technique as a methodology for software development facilitates the achievement of modifiability and reliability.

   A major component of the iterative process is the analysis phase that is performed on each successive implementation. Specific topics of analysis include such items as the structure, modularity, modifiability, usability, reliability and efficiency of the current implementation as well as an assessment of the achievement of the goals of the project. {Stepwise Refinement}

RELIABILITY

Achieved by:

(a) Hierarchical Decomposition (*)

(b) Information Hiding (*)

(c) Stepwise Refinement (*)

(d) Structured Programming (*)

References:

(i) Liskov, B.H. (1976), "A Design Methodology for Reliable Software Systems," In: P. Freeman and A.I. Wasserman (eds.), *Tutorial on Software Design Techniques,* IEEE Computer Society Press, Long Beach, CA.

(ii) Baker, F.T. (1978), "Structured Programming in a Production Programming environment," In: C.V. Ramamoorthy and R.T. Yeh (eds.), *Tutorial: Software Methodology,* IEEE Computer Society Press, Long Beach, CA.

(iii) Ross, D.T., J.B. Goodenough and C.A. Irvine (1975), "Software Engineering: Process, Principles and Goals," *IEEE Computer,* Vol. 8, No. 5, pp 17-27.

(iv) Basili V.R. and A.J. Turner (1975), "Iterative Enhancement: A Practical Technique for Software Development," *IEEE Transactions on Software Engineering,* V0l. SE-1, No. 4, pp. 390-396.

(v) Dietel, H.M. (1984), *An Introduction to Operating Systems,* Addison-Wesley, Reading, MA.

NOTES

1. PP393/(ii): When carried to its fullest extent, top-down development has greater (better) effects on reliability than any other component of the methodology. {Hierarchical Decomposition}

2. PP21/(iii): In general, hierarchical decomposition is cited as helping to improve software reliability, helping to allow multiple use of common designs and programs, and helping to make it easier to modify programs. {Hierarchical Decomposition}

3. PP82/(i): Structured Programming is a programming discipline which was introduced with

reliability in mind. {Structured Programming}

4. PP390/(iv): Iterative enhancement is a practical means of applying stepwise refinement. This technique as a methodology for software development facilitates the achievement of modifiability and reliability. {Stepwise Refinement}

5. PP103/(v): Information hiding is a system structuring technique that greatly facilitates the development of more reliable software systems. {Reliability}

PORTABILITY

Achieved by:

(a) Functional Decomposition (*)

(b) Documentation (*)

References:

(i) Arthur, L.J. (1985), *Measuring Programmer Productivity and Software Quality,* John Wiley, New York, NY.

(ii) Stern, M. (1978), "Some Experience in Building Portable Software," *Proc. 3rd International Conference on Software Engineering,* Atlanta, GA, pp. 327-332.

NOTES

1. PP123/(i): Portability is a function of five underlying metrics: generality, hardware independence, modularity, self-documentation, and software system independence.

Modularity provides a structure of highly independent modules. Independent modules are less likely to have input/output statements that are one of the banes of portability.

A program with good informative comments will further improve the portability. The person porting the code probably has little knowledge of the original application development, but does know the new machine. Clear, self-documenting code and good comments will speed the transfer of software from one machine to another. {Functional Decomposition, Documentation}

2. PP332/(ii): One class of portability questions relates to the installation, maintenance, and portation of the product, rather than to its internal functions. One such problem relates to the object module format, another question relates to the actual installation procedure. The final system dependent component of the program is its operating documentation. {Documentation}

Appendix 3

LINKAGES BETWEEN PRINCIPLES & ATTRIBUTES

Principles and attributes and how they are related

In the following document an (*) against an attribute indicates identification of reference for the linkage.

HIERARCHICAL DECOMPOSITION

Induces:

(a) Coupling (*)

(b) Cohesion (*)

(c) Complexity (*)

(d) Ease of Change (*)

References:

(i) Pressman, R.S. (1982), *Software Engineering: A Practitioner's Approach,* McGraw-Hill, New York, NY.

(ii) Fairley, R.P. (1985) *Software Engineering Concepts,* McGraw-Hill, New York, NY.

(iii) Ross, D.T., J.B. Goodenough and C.A. Irvine (1975), "Software Engineering: Process, Principles and Goals," *IEEE Computer,* Vol. 8, No. 5, pp 17-27.

(iv) Yourdon, E. and L.L. Constantine (1979), *Structured Design,* Prentice-Hall, Englewood Cliffs, NJ.

NOTES

1. PP158/(i): The concept of module independence is the direct outgrowth of modularity and the concepts of abstraction and information hiding.

   Independence is measured using two qualitative criteria: cohesion and coupling. {Coupling, Cohesion}

2. PP144/(ii): Hierarchical decomposition promotes ease of understanding, implementation, modification. {Ease of Change}

3. PP21/(iii): In general, hierarchical decomposition is cited as helping to improve software reliability, helping to allow multiple use of common designs and programs, and helping to make it easier to modify programs. {Ease of Change}

4. PP89/(iv): The second dimension of coupling is complexity. The more complex a single connection is, the higher the coupling. {Coupling, Complexity}

FUNCTIONAL DECOMPOSITION

Induces:

(a) Coupling (*)

(b) Cohesion (*)

(c) Complexity (*)

(d) Ease of Change (*)

References:

(i) Pressman, R.S. (1982), *Software Engineering: A Practitioner's Approach,* McGraw-Hill, New York, NY.

(ii) Hosier, J. (1978), *Structured Analysis and Design,* Infotech International Limited, England.

(iii) Bergland, G.D. (1981), "Structured Design Methodologies," *Proc. 15th Annual Design Automation Conference,* June 1978, pp. 475-493.

(iv) Parnas, D.L. (1972), "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM,* Vol. 15 No. 5, May 1972, pp. 1053-1058.

NOTES

1. PP37/(ii): Another advantage to be achieved through the reduction of complexity by functional decomposition is that maintainability is enhanced. However, this benefit arises when decomposition is carried out in the appropriate way. {Complexity}

2. PP158/(i): The concept of module independence is the direct outgrowth of modularity and the concepts of abstraction and information hiding.

   Independence is measured using two qualitative criteria: cohesion and coupling.

   Cohesion is a natural extension of the information hiding concept. A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program. {Coupling, Cohesion}

3. PP479/(iii): Functional Decomposition is simply the divide and conquer technique applied to programming. The concept of divide and conquer as an answer to complexity is very important provided it is done correctly. When a program can be divided into two independent parts, complexity is reduced dramatically. {Complexity}

4. PP1053/(iv): The benefits expected of modular programming (functional decomposition) are: (1) managerial- development time should be shortened because separate groups would work on each module with little need for communication; (2) product flexibility- it should be possible to make drastic changes to one module without a need to change others. {Ease of change}

INFORMATION HIDING

Induces:

(a) Coupling (*)

(b) Cohesion (*)

(c) Complexity (*)

(d) Well-defined Interface (*)

(e) Ease of Change (*)

References:

(i) Pressman, R.S. (1982), *Software Engineering: A Practitioner's Approach,* McGraw-Hill, New York, NY.

(ii) Gilbert, P. (1983), *Software Design and Development,* Science Research Associates, Chicago, IL.

(iii) Peters, L.J. (1981), *Software Design: Methods and Techniques,* Yourdon Press, New York, NY.

(iv) Fairley, R.P. (1985), *Software Engineering Concepts,* McGraw-Hill, New York, NY.

(v) Parnas, D.L. (1979), "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering,* Vol. SE-5, No. 2, pp. 128-138.

(vi) Peters, L.J. (1981), *Software Design: Methods & Techniques,* Yourdon Press, New York, NY.

(vii) Heninger-Britton, K.A., R.A. Parker and D.L. Parnas (1981), "A Procedure for Designing Abstract Interfaces for Device Interface Modules," *Proc. 5th IEEE International Conference on Software Engineering,* pp. 195-204.

NOTES

1. PP158/(i): The concept of module independence is the direct outgrowth of modularity and the concepts of abstraction and information hiding.

   Independence is measured using two qualitative criteria: cohesion and coupling.

   Cohesion is a natural extension of the information hiding concept. A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program. {Coupling, Cohesion}

2. PP157/(ii): Use of information hiding provides greatest benefits when modifications are required. {Ease of Change}

3. PP180/(iii): This hiding is akin to eliminating the least desirable types of coupling. {Coupling} This method (information hiding) addresses real issues such as how to design for ease of change. {Ease of Change}

4. PP141/(iv): When a software system is designed using the information hiding approach, each module in the system hides the internal details of its processing activities and modules communicate only through well-defined interfaces. {Well-defined Interface}

5. PP131/(v): The crucial steps are:

   - Identification of the items that are likely to change. These items are termed "secrets".

   - Location of the specialized components in separate modules.

   - Designing intermodule interfaces that are insensitive to the anticipated changes. The changeable aspects or "secrets" of the modules are not revealed by the interface.

   It is exactly this that the concept of information hiding, encapsulation, or abstraction is intended to do for software. {Well-defined Interface}

6. PP180/(vi): This "hiding" is akin to eliminating the least desirable type of coupling, and treating each module as a black box, as is done in structured design. {Coupling}

7. PP195/(vii): Much of the complexity of embedded real-time software is associated with controlling special-purpose hardware devices. Many designers seek to reduce this

complexity by isolating device characteristics in software device interface modules, thereby allowing most of the software to be programmed without knowledge of device details. {Complexity}

STEPWISE REFINEMENT

Induces:

(a) Coupling (*)

(b) Cohesion (*)

(c) Complexity (*)

References:

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

(ii) Gilbert, P.(1983), *Software Design and Development,* Science Research Associates, Chicago, IL.

(iii) Pressman, R.S. (1982), *Software Engineering: A Practitioner's Approach,* McGraw-Hill, New York, NY.

NOTES

4. PP416/(ii): Stepwise refinement is a design strategy that seeks to implant ease of understanding, and ease of maintenance and modification into the program text. {Complexity}

2. PP226/(i): The degree of modularity obtained using stepwise refinement will determine the ease or difficulty with which a program can be adapted to changes or extensions of the purpose.

PP158/(iii): The concept of module independence is the direct outgrowth of modularity and the concepts of abstraction and information hiding.

Independence is measured using two qualitative criteria: cohesion and coupling. {Coupling, Cohesion}

STRUCTURED PROGRAMMING

Effect on derived attributes:

(a) Complexity (*)

(b) Readability (*)

References:

(i) Gilbert, P. (1983), *Software Design and Development,* Science Research Associates, Chicago, IL.

(ii) De Marco, Tom (1979), *Concise Notes on software Engineering,* Yourdon Press, New York, NY.

(iii) Bates, D. (ed.) (1976), *Structured Programming,* Infotech International Limited, England.

NOTES

5. PP403/(i): Structured programming (SP) uses only three forms (sequence, choice and iteration). Any programming calculation can be programmed using these forms. And when only these forms are used, the resulting program is easily understandable to programmers seeking it for the first time. There are fewer intricacies and "clevernesses" to be puzzled through. Thus maintenance personnel can discover quickly which portions of the program require maintenance or modification. {Complexity, Readability}

2. PP39/(ii): Boehm, Jacopini, Dijkstra and Warnier introduced structured coding to describe the idea of building programs using limited control structures in order to enhance readability and ease of testing. {Readability}

3. PP24/(iii): Most of the objectives of structured programming can be summarized by two principles: reduction of complexity and concern with correctness. {Complexity}

LIFE_CYCLE VERIFICATION

Induces:

(a) Visibility of Behavior (*)

(b) Early Error Detection (*)

References:

(i) Osterweil, L.J., J.R. Brown and L.G. Stucki (1979), "ASSET: A Life Cycle Verification and Visibility System," *The Journal of Systems and Software,* Vol. 1, No. 1, pp. 77-86.

(ii) Ramamoorthy, C.V. (1978), "Introduction," In: C.V. Ramamoorthy and Y.T. Yeh (eds.), *Tutorial: Software Methodology,* IEEE Computer Society, Long Beach, CA.

(iii) Ross, D.T., J.B. Goodenough and C.A. Irvine (1975), "Software Engineering: Process, Principles and Goals," *IEEE Computer,* Vol. 8, No. 5, pp 17-27.

(iv) Hammond, L.S., D.L. Murphy and M.K. Smith (1978), "A System for Analysis and Verification of Software Design," *Proc. Compsac 1978,* Chicago, IL.

(v) Hoffnagle, G.F. and W.E. Beregi (1985), "Automating the Software Development Process," *IBM Systems Journal,* Vol. 24, No. 2, pp. 102-120.

NOTES

1. PP77/(i): Visibility is greatly enhanced by applying verification techniques throughout life cycle. {Visibility of Behavior}

2. PP3/(ii): Because a methodology demands a critical analysis at each phase of the development, it reduces the amount of effort needed for testing and validation, and reduces the errors in problem definition which provoke a multiplicity of errors in the implementation. {Early Error Detection}

3. PP23/(iii): Confirmability (Life-Cycle Verification)- Confirmability is a principle that directs attention to methods for finding out whether stated goals have been achieved. Applied to design issues, confirmability refers to the structuring of a system so it is readily tested. It must be possible to stimulate the constructed system in a

controlled manner so its response can be evaluated for correctness. Applied to notational matters, confirmability means that a notation should require explicit specification of constraints that affect the correctness of a design or implementation (e.g., data declarations that specify range of values and units of value as well as mode of representation). Applied to the practice of software engineering, confirmability refers to the use of such methods as structured walk-throughs of designs, egoless programming, and other methods that help to ensure that nothing has been overlooked.

4. PP42/(iv): Verification and testing should not be viewed as a development phase, but rather as control activities occuring during each development phase.

5. PP110/(v): Continuous verification ensures that errors are found early and at least cost. {Early Error Detection}

DOCUMENTATION

Achieves:

(a) Reduced Complexity (*)

(b) Improved Readability (*)

(c) Improves Ease of Change (*)

(d) Traceability (*)

References:

(i) Bates, D. (ed.) (1977), *Software Engineering Techniques,* Infotech International Limited, England.

(ii) Gilbert, P.(1983), *Software Design and Development,* Science Research Associates, Chicago, IL.

(iii) Cave, W.C. and G.W. Maymon (1984), *Software Life-cycle Management,* Macmillan, New York, NY.

(iv) Horowitz, E. and R.C. Williamson (1986), "SODOS: A Software Documentation Support Environment - Its Definition", *IEEE Transactions on Software Engineering,* Vol. SE-12, No. 8, pp. 849-859.

NOTES

1. PP18/(i): The whole system must be fully recorded on paper, so that each decision taken in the development can be traced to its reasons, and every statement in the final code can be traced to a corresponding element in the problem specification. {Traceability}

2. PP18/(ii): Documentation should include features to aid debugging and planned changes or extensions. {Ease of Change}

3. PP68/(iii): Documentation provides a clear understanding between user and developer about what the system will do; and between designers and programmers about what the program modules will do. {Complexity, Readability}

4. PP849/(iv): The advantage of documentation is that it permits traceability through all phases of the software life cycle. {Traceability}.

Appendix 4

Software Engineering Attributes

and

Their Related Assessment Factors

COUPLING.

Influenced by:

(a) Use of global variables (-)

(b) Use of structured data types as parameters (-)

(c) Use of switches as parameters (-)

(d) Use of parameters (-)

(e) Use of parameterless procedure calls (-)

(f) Types of parameters- control vs data

(1) Control (-)

(2) Data (+)

(g) Multiple entry points in a routine (-)

(h) Fan-in to a routine (-)

(i) #Goto's & use of Goto's (-)

COHESION.

Influenced by:

(a) Use of switches as parameters (-)

(b) Use of control structures (+)

(c) Multiple entry points in a routine (-)

(d) Fan-in to a routine (+)

(e) Fan out (-)

(f) Modularization (+)

COMPLEXITY.

Influenced by:

(a) Use of structured data types as variables (+)

(b) Use of control structures (+)

(c) Use of "excessive" nesting of control structures (-)

(d) "Excessive" use of control structures (-)

(e) Use of dynamic structures (-)

(f) Use of meaningful names for routines, variables (+)

(g) Multiple exit points from loops (-)

(h) Multiple exit points from routines (-)

(i) Fan out (-)

(j) # Goto's (-)

(k) Use of recursive code (-)

(l) Use of negative/compound boolean expressions (-)

(m) Use of embedded alternate language (-)

(n) Use of code indentation (+)

(o) Use of "excessive" code indentation (-)

(p) Length of routine/module (-)

(q) Total program length (-)

(r) Modularization (+)

(s) Use of "excessive" number of routines (-)

(t) Use of block comments (+)

(u) Use of comments (+)

(v) Use of "excessive" # single line comments in line (-)

(w) Use of comments consistent with code functions (+)

(x) Completeness/Accuracy of documentation (+)

WELL-DEFINED INTERFACE.

Influenced by:

(a) Use of global variables (-)

(b) Use of structured data types as parameters (+)

(c) Use of "excessive" # parameters (-)

(d) Use of parameterless procedure calls (-)

READABILITY.

Influenced by:

(a) Use of control structures (+)

(b) Use of symbolic constants (+)

(c) Use of special characters (+)

(d) Use of meaningful names for routines, variables (+)

(e) Multiple exit points from loops (-)

(f) Multiple exit points from routines (-)

(g) # Goto's & use of goto's (-)

(h) Use of embedded alternate language (-)

(i) Use of code indentation (+)

(j) Use of "excessive" code indentation (-)

(k) TLOC > ELOC (+)

(l) Length of routine (-)

(m) Use of block comments (+)

(n) Use of comments (+)

(o) Use of "excessive" # single line comments in line (-)

(p) Use of comments consistent with code functions (+)

(q) Grammatically correct comments/spellings (+)

(r) Completeness/Accuracy of documentation (+)

(s) Use of "excessive" nesting of control structures (-)

(t) Use of parentheses around conditions (+)

EASE OF CHANGE.

Influenced by:

(a) Use of global variables (-)

(b) Use of parameters (-)

(c) Use of dynamic structures (+)

(d) Use of symbolic constants (+)

(e) Fan out (-)

(f) Modularization (+)

(g) Use of "sandwiching" (+)

(h) Completeness/Accuracy of documentation (+)

TRACEABILITY.

Influenced by:

(a) Use of comments referencing project documentation (+)

(b) Use of comments referencing "who called me" (+)

(c) Consistency in use of variable names in code & documentation (+)

(d) Organizational consistency between code and documentation (+)

VISIBILITY OF BEHAVIOR.

Influenced by:

(a) Certification levels in "comments" (+)

(b) Awareness of validation & verification (+)

(c) Procedures of validation & verification (+)

(d) Enforcement of validation & verification (+)

(e) Accessibility to results of validation & verification (+)

EARLY ERROR DETECTION.

Influenced by:

(a) Certification levels in "comments" (+)

(b) Awareness of validation & verification (+)

(c) Procedures of validation & verification (+)

(d) Enforcement of validation & verification (+)

(e) Accessibility to results of validation & verification (+)

Appendix 5

ASSESSMENT FACTORS

The product attributes to which they are related and why

USE OF GLOBAL VARIABLES

Affect:

(a) Coupling (-)

(b) Well-defined Interface (-)

(c) Ease of Change (-)

References:

(i) Yourdon, E. and L. L. Constantine (1979), *Structured Design,* Prentice Hall, New Jersey.

(ii) Troy, D.A. and S.H. Zweben (1981), "Measuring the Quality of Structured Design," *The Journal of Systems and Software,* Vol. 2, pp. 113-120.

(iii) Dunsmore, H.E. and J.D. Gannon (1978), "Programming Factors - Language Features that Help Explain Programming Complexity," *Communications of the ACM,* Vol. 21, pp 554-560.

(iv) Page-Jones, M. (1980), *The Practical Guide to Structured Design,* Yourdon Press, New York.

NOTES

1. PP98/(i): Whenever two or more modules interact with a common data environment, those modules are said to be common-environment coupled. Each pair of modules which interacts with the common environment is coupled- regardless of the direction of communication or the form of reference. {Coupling}

2. PP115/(ii): Common environments increase level of coupling in the design. {Coupling}

3. PP556/(iii): Data Environment Ratio: The number of global variables divided by the total number of variables (including parameters). {Well-defined Interface}

4. PP111/(iv): Common Coupling

   a) A bug in any module using a global area may show up in any other module using that global area because global data is not protected. {Coupling}

b) It is difficult to find what modules must be changed if a piece of (global) data is changed, e.g. if a record in global area is changed from 96 bytes to 112 bytes, several modules will be affected. But which? You must check every module in the system. {Ease of Change}

USE OF STRUCTURED DATA TYPES AS PARAMETERS

Affects:

(a) Coupling (-)

(b) Well-defined Interface (+)

References:

(i) Troy, D.A. and S.H. Zweben (1981), "Measuring the Quality of Structured Design," *The Journal of Systems and Software,* Vol. 2, pp. 113-120.

(ii) Lohse, J.B. and S.H. Zweben (1984), "Experimental Evaluation of Software Design Principles: An Investigation into the Effect of Module Coupling on the system and Modifiability," *Journal of Systems and Software,* Vol. 4, pp 301-308.

NOTES

1. PP115/(i): Coupling also increases as the extraneous information irrelevant to a module's task is present in the interface. {Coupling, Well-defined Interface}

2. PP302/(ii): A dimension of module coupling is whether simple data items or entire structures are passed. (When a structured data type is passed, it induces extra coupling as a result of passing some data items which are not necessary for computation) {Coupling}

USE OF STRUCTURED DATA TYPES AS VARIABLES

Affects:

(a) Complexity (+)

References:

(i) Conway, R., D. Gries and E.C. Zimmerman (1976), *A Primer on Pascal,* Winthrop Computer Systems Series, Cambridge, MA.

NOTES

1. PP215/(i): It takes a little longer to write such (data structures as variables) but they are much clearer to the reader. Both the names and the grouping emphasize the relationships between variables. {Complexity}

USE OF SWITCHES AS PARAMETERS

Affects:

(a) Coupling (-)

(b) Cohesion (-)

References:

(i) De Marco, T. (1978), *Structured Analysis and System Specification,* Yourdon Press, New York.

NOTES

1. PP309/(i): Direction of some couples is relevant, e.g. downward passing switches have a stronger linking effect than upward passing switches. This is because downward passing switches tend to ruin the integrity of receiving module, by driving it from above to do things whose significance it cannot fully understand. {Coupling}

2. PP312/(i): The downward passing switch is probably the simplest test of poor cohesion. {Cohesion}

   This is because when a switch is passed downwards the controlling module is not aware as to how and how many modules are going to be affected by this switch. On the contrary when a switch is passed upward the passing module knows exactly which (one) module will be affected. when a switch is passed upward the passing module has complete visibility of its effects.

USE OF PARAMETERS

Affect:

(a) Coupling (-)

(b) Ease of Change (-)

References:

(i) Yourdon, E. and L. L. Constantine (1979), *Structured Design,* Prentice Hall, New Jersey.

(ii) Lohse, J.B. and S.H. Zweben (1984), "Experimental Evaluation of Software Design Principles: An Investigation into the Effect of Module Coupling on the system and Modifiability," *Journal of Systems and Software,* Vol. 4, pp 301-308.

(iii) Page-Jones, M. (1980), *The Practical Guide to Structured Design,* Yourdon Press, New York.

(iv) Stevens, W.P. (1981), *Using Structured Design,* John Wiley & Sons, New York.

NOTES

1. PP86/(i): Inter modular coupling is influenced by complexity of the interface. This is approximately equal to the number of different items being passed (not the amount of data)- the more the items, the higher the coupling. {Coupling}

2. PP302/(ii): The size of the interface (the number of items passed) affects coupling. {Coupling}

3. PP103/(iii): The fewer the connections there are between two modules, the less chance is for the ripple effect (a bug in one module appearing as a symptom in another). Coupling through parameters (data coupling) is necessary for communication between two modules. It is harmless so long as its kept to a minimum. {Coupling}

4. PP85/(iv): It is usual for the usability of a module to increase as the number of parameters decreases. The decreased coupling makes the module more functional and more independent of its environment and thus more usable elsewhere. Flexibility increases as the number of parameters is decreased. {Coupling, Ease of Change}

5. PP102/(iv): Most modules should have three or fewer parameters, with ERROR and EOF parameters included in the count.

6. PP62/(iv): The primary goal is high module independence. Parameters are the most direct measure of the amount of independence achieved. The amount of communication needed between modules can usually be seen only through the process of specifying the parameters necessary to make the structure meet the specifications. {Coupling}

USE OF "EXCESSIVE" # OF PARAMETERS

Affect:

(a) Well-defined Interface (-)

References:

(i) Stevens, W.P. (1981), *Using Structured Design,* John Wiley & Sons, New York.

(ii) Fairley, R.F. (1985), *Software Engineering Concepts* McGraw-Hill, New York.

NOTES

1. PP102/(i): Most modules should have three or fewer parameters, with ERROR and EOF parameters included in the count. Too many parameters spoil the well-defined interface. {Well-defined Interface}

2. PP217/(ii): Parameters bind different arguments to a routine different invocations of the routine. Parameters should be few in number. Long, involved parameter lists result in excessive complex routines that are difficult to understand and difficult to use; they result from inadequate decomposition of a software system.

   A routine should not have more than five formal parameters. Selection of number five as the suggested upper bound is not an entirely arbitrary choice. It is well known that human beings can deal with approximately seven items or concepts at one time.

   Fewer parameters and fewer global variables improve the clarity and simplicity of subprograms. In this regard, five is a very lenient upper bound. In practice, we prefer no more than three or four formal parameters. {Well-defined Interface}

USE OF PARAMETERLESS PROCEDURE CALLS

Affect:

(a) Well-defined Interface (-)

(b) Coupling (-)

References:

(i) Pratt, T.W. (1984), *Programming Languages- Design and Implementation,* Prentice Hall, Inc., Englewood Cliffs, NJ.

NOTES

1. PP47/(i): Implicit arguments: An operation in a program ordinarily is invoked with a set of explicit arguments. However, the operation may access other implicit arguments through the use of global variables or other nonlocal identifier references. Complete determination of all the data that may affect the result of an operation is often obscured by such implicit arguments. {Well-defined Interface}

TYPES OF PARAMETERS- CONTROL vs DATA

Affect:

(a) Coupling

Control (-)

Data (+)

References:

(i) Yourdon, E. and L. L. Constantine (1979), *Structured Design,* Prentice Hall, New Jersey.

(ii) Troy, D.A. and S.H. Zweben (1981), "Measuring the Quality of Structured Design," *The Journal of Systems and Software,* Vol. 2, pp. 113-120.

NOTES

1. PP86/(i): Data-coupled systems have lower coupling than control coupled systems. {Coupling}

2. PP90/(i): The communication of data alone is necessary and sufficient for functioning systems of modules. Control communication represents a dispensable addition. {Coupling}

3. PP115/(ii): Coupling increases with increasing complexity of the interface between two modules and increases as the type of interconnection varies from data to control. {Coupling}

USE OF CONTROL STRUCTURES

Affect:

(a) Cohesion (+)

(b) Complexity (+)

(c) Readability (+)

References:

(i) De Marco, T. (1979), *Concise Notes on Software Engineering,* Yourdon Press, New York.

(ii) Yourdon, E. and L. L. Constantine (1979), *Structured Design,* Prentice Hall, New Jersey.

(iii) Troy, D.A. and S.H. Zweben (1981), "Measuring the Quality of Structured Design," *The Journal of Systems and Software,* Vol. 2, pp. 113-120.

(iv) Conte, S.D., H.E. Dunsmore and V.Y Shen (1986), *Software Engineering Metrics and Modules,* Benjamin Cummins.

(v) Shneiderman, Ben (1980), *Software Psychology,* Winthrop Publishers, Cambridge, MA.

NOTES

1. PP44/(i): Readability is enhanced by reformulating the logic to make it more nearly one-dimensional. (When a control structure is used to represent some complex logic, that piece of code is more like a unit with all relevant code encompassed in it). {Readability}

2. PP73/(ii): Span of control flow affects complexity. It is the number of lexically contiguous statements one must examine before one finds a black-box section of code that has one entry point and one exit point. A means of reducing this span to an almost minimal length is by organizing the logic into combinations of 'IF-THEN-ELSE', 'DO-WHILE' operations. {Complexity}

3. PP115/(iii): Cohesion - The goal here is to strive for modules whose elements (statements and

functions) are highly related. {Cohesion}

4. PP74/(iv): Nesting allows the programmer to avoid excessive compound conditionals in any one IF or WHILE statement by taking advantage of conditions in effect. {Complexity, Readability}

5. PP79/(v): One component of structured programming is the use of higher-level control structures such as IF-THEN-ELSE or the DO-WHILE statements to replace lower-level GOTO statements. The higher-level control structures have the advantageous 'one-in, one-out' property which restricts entry and exit, facilitating composition and comprehension by limiting complexity. {Complexity, Readability}

USE OF "EXCESSIVE" NESTING OF CONTROL STRUCTURES

Affects:

(a) Complexity (-)

(b) Readability (-)

References:

(i) Dunsmore, H.E. and J.D. Gannon (1978), "Programming Factors- Language Features that Help Explain Programming Complexity," *Communications of the ACM,* pp 554-560.

(ii) McCall, J.A., P.K. Richards and G.F. Walters (1977), "Factors in Software Quality," RADC TR-77-369, Vol. 1.

(iii) Conte, S.D., H.E. Dunsmore and V.Y. Shen (1986), *Software Engineering Metrics and Modules,* Benjamin Cummins.

(iv) Zolnowski, J.C. and D.B. Simmons (1981), "Taking the Measure of Programming Complexity," *Proc. AFIPS National Computer Conference,* pp 329- 336.

(v) Weissman, L. (1974), "Psychological Complexity of Computer Programs: An Experimental Methodology," *ACM SIGPLAN Notices,* No. 6, pp. 25-36.

(vi) Arthur, L.J. (1984), *Measuring Programmer Productivity and Software Quality,* John Wiley, New York, NY.

(vii) Perry, E. (1985), *Systems Analysis, Design and Development,* HRW Publishers, New York, NY.

NOTES

1. PP556/(i): Nesting depth affects complexity. {Complexity}

2. PP6-45/(ii): The greater the nesting level of decisions or loops within a module, the greater the complexity. {Complexity}

3. PP74/(iii): Excessive nesting can lead to circumstances in which it is difficult for programmers to comprehend what must be true for a particular statement to be reached (Especially w.r.t. conditionals and knowing what condition is applicable). {Com-

plexity}

4. PP75/(i): The higher the nesting depth, the more difficult it is to assess the entrance conditions. {Complexity}

5. PP333/(iv): Depth of nesting is a measure of program complexity. {Complexity}

6. PP28/(v): If the nesting is too deep, the program may become unintelligible. {Complexity}

7. PP153/(vi): Decision and loop nesting complexity are two more code level metrics that help indicate the difficulty involved in testing a piece of code. Nesting levels of three or more are difficult to understand and, therefore, to test. The maximum nesting level should be three or perhaps four. {Complexity}

8. PP399/(vii): Avoid over five levels of nested IFs; they are too hard to read. {Readability}

"EXCESSIVE" USE OF CONTROL STRUCTURES

Affects:

(a) Complexity (-)

References:

(i) Arthur, L.J. (1984), *Measuring Programmer Productivity and Software Quality,* John Wiley, New York.

NOTES

1. PP63/(i): Each decision (IF-THEN-ELSE, DOWHILE, DOUNTIL, REPEAT UNTIL etc.) has at least two program paths that must be tested. Each path adds complexity to the program. Decision Density (DD) is defined as follows:

   DD = (Total # Decisions)/ ELOC

   Higher the DD, higher the complexity. {Complexity}

USE OF DYNAMIC STRUCTURES

Affects:

(a) Ease of Change (+)

(b) Complexity (-)

References:

(i) Weissman, L. (1974), "Psychological Complexity of Computer Programs: An Experimental Methodology," *ACM SIGPLAN Notices,* No. 6, pp. 25-36.

(ii) Dale, N. and D. Orshalick (1983), *Introduction to Pascal and Structured Design,* D.C. Heath and Co., Lexington, MA.

(iii) Grogono, P. (1983), *Programming in Pascal,* Addison-Wesley, Reading, MA.

NOTES

1. PP257/(iii): Dynamic data structures, on the other hand, change in size during the execution of the program. {Complexity}

2. PP29/(i): Programs containing pointers are felt to be more complex than those without. {Complexity}

3. PP426/(ii): Using dynamic variables it is possible to overcome problems of insertion and deletion of components. {Ease of Change}

USE OF SYMBOLIC CONSTANTS

Affects:

(a) Readability (+)

(b) Ease of Change (+)

References:

(i) Kernighan, B.W. and Plauger P.J. (1981), *Software Tools in Pascal,* Addison-Wesley, Reading, MA.

(ii) Hansen, P.B. (1977), *The architecture of Concurrent Programs,* Prentice-Hall, Inc., Englewood Cliffs, NJ.

NOTES

1. PP11/(i): Symbolic constants contribute a great deal to the readability of the code. {Readability}

2. pp25/(i): The constant declaration will be easy to find and change. {Ease of Change}

3. PP36/(i): The purpose of symbolic constants is to retain mnemonic information as much as possible. {Readability}

4. PP265/(i): "Symbolic constants" like ENDFILE tell you what a number signifies in a way that the number itself could never do: if we had written some magic value like -1 you would not know what it meant without understanding the surrounding context. {Readability}

5. PP31/(ii): If a constant is used several times in a program, it is useful to define its value once and refer to it elsewhere by an identifier. This makes it easy to change the value later if necessary. {Ease of Change}

USE OF SPECIAL CHARACTERS

Affects:

(a) Readability (+)

References:

(i) Pratt, T.W. (1984), *Programming Languages- Design and Implementation,* Prentice Hall, Inc., Englewood Cliffs, NJ.

NOTES

1. PP310/(i): Variations among languages are mainly in the optional inclusion of special characters such as '.' or '-' to improve readability and in length restrictions. {Readability}

USE OF PARENTHESES AROUND CONDITIONS

Affects:

(a) Readability (+)

References:

(i) Perry, Edward (1985), *Systems Analysis, Design and Development,* HRW Publishers, New York, NY.

NOTES

1. PP399/(i): To improve readability, place parentheses around conditions being tested. {Readability}

USE OF MEANINGFUL NAMES FOR ROUTINES, VARIABLES

Affect:

(a) Readability (+)

(b) Complexity (+)

References:

(i) Kernighan, B.W. and P.J. Plauger (1978), *The Elements of Programming Style,* McGraw-Hill, New York.

(ii) Weissman, L. (1974), "Psychological Complexity of Computer Programs: An Experimental Methodology," *ACM SIGPLAN Notices,* N0. 6, pp. 25-36.

(iii) Conway, R., D. Gries and E.C. Zimmerman (1976), *A Primer on Pascal,* Winthrop Computer Systems Series, Cambridge, MA.

NOTES

1. PP145/(i): Meaningful names serve to aid memory of the person reading the code. {Readability}

2. PP27/(ii): Mnemonic variable names should make programs more understandable than non-mnemonic variables. {Readability, Complexity}

3. PP19/(iii): You should choose variable names that suggest the role the variables play in the program. {Readability, Complexity}

MULTIPLE ENTRY POINTS IN A ROUTINE

Affect:

(a) Coupling (-)

(b) Cohesion (-)

References:

(i) Yourdon, E. and L. L. Constantine (1979), *Structured Design,* Prentice Hall, New Jersey.

(ii) Perry, E. (1985), *Systems Analysis, Design and Development,* HRW Publishers, New York, NY.

NOTES

1. PP88/(i): Use of multiple entry points guarantees that there is more than the minimum number of interconnections for the system. {Coupling}

   If a module has multiple entry points, it is implied that there are pieces of code in the module that are performing single specific functions which in turn implies that the module is not functionally cohesive. {Cohesion}

2. PP110/(ii): Program modules must remain single purpose, with single entry and exit points. A program cannot jump into the middle of a module, nor can it leave the module except via its sole exit. (Such) Modules that perform a single logical function are called cohesive. {Cohesion}

MULTIPLE EXIT POINTS FROM LOOPS

Affect:

(a) Complexity (-)

(b) Readability (-)

(c) Cohesion (-)

References:

(i) Kernighan, B.W. and P.J. Plauger (1978), *The Elements of Programming Style,* McGraw-Hill, New York.

NOTES

1. Avoid multiple exits from loops. Multiple exits from a loop have an adverse effect on complexity and readability because they hamper the continuity of code from the reader's point of view. {Complexity, Readability}

MULTIPLE EXIT POINTS FROM ROUTINES

Affect:

(a) Complexity (-)

(b) Readability (-)

(c) Cohesion (-)

References:

(i) Arthur, L.J. (1984), *Measuring Programmer Productivity and Software Quality,* John Wiley, New York, NY.

(ii) Perry, E. (1985), *Systems Analysis, Design and Development,* HRW Publishers, New York, NY.

NOTES

1. PP224/(i): Each paragraph may be analyzed for multiple exits in the form of GOTO's, GOBACK, STOP RUN, and so on. Each of these exit points reduces the flexibility and maintainability of the module. In a well structured program, essential complexity will go to zero. By definition, a module with a single exit and no GOTOs will have an essential complexity of zero; it can be reduced to improve flexibility and maintainability. {Complexity, Readability}

2. PP110/(ii): Program modules must remain single purpose, with single entry and exit points. A program cannot jump into the middle of a module, nor can it leave the module except via its sole exit. (Such) Modules that perform a single logical function are called cohesive. {Cohesion}

FAN-IN TO A ROUTINE

Affect:

(a) Cohesion (+)

(b) Coupling (-)

References:

(i) Troy, D.A. and S.H. Zweben (1981), "Measuring the Quality of Structured Design," *The Journal of Systems and Software*, Vol. 2, pp. 113-120.

NOTES

1. PP115/(i): The following design features are measures of cohesion: {Cohesion}

   a) The maximum fan-in to any box in the structure chart.

   b) The average fan-in in the structure chart.

   The rationale behind this probably is that more the fan-in to a routine more specific, well-defined (single) function it is performing. In cohesion we are striving for this attribute.

2. More who call more the coupling. {Coupling}

FAN-OUT

Affects:

(a) Ease of Change (-)

(b) Complexity (-)

(c) Cohesion (-)

References:

(i) Stevens, W.P. (1981), *Using Structured Design,* John Wiley, New York.

(ii) Card, D.N., V.E. Church and W.W. Agresti (1986), "An Empirical Study of Software Design Practices," *IEEE Transactions of Software Engineering,* Vol. 12, No. 2, pp 115-139.

(iii) Miller, G.A. (1956), "The Magical Number Seven, Plus or Minus Two: some Limits on our Capacity for Processing Information," *Psychological Review,* Vol. 63, pp 81-97.

(iv) Perry, Edward (1985), *Systems Analysis and Design,* HRW Publishers, New York, NY.

NOTES

1. PP97/(i): Multiple calls from a module indicate a tendency toward too much control within a single module. The problem may be a missing level. The function and control contained within the original module may need to be divided. {Ease of Change, Complexity, Cohesion}

2. PP268/(ii): No module should call more than 7 other modules. The formulation of this concept is an adaptation of the "7 plus or minus two" rule (iii). {Ease of Change, Complexity, Cohesion}

3. Since modules are parents, just how many child modules should each control? Span of control refers to the number of subservient modules controlled by a parent module. Spans of five to nine are considered ideal. Spans of one or two are too few, and spans of 12 to 15 are too many. {Ease of Change, Complexity}

GOTO's & USE OF GOTO's

Affect:

(a) Complexity (-)

(b) Readability (-)

(c) Coupling (-)

References:

(i) De Marco, T. (1978), *Structured Analysis and System Specification,* Yourdon Press, NY.

(ii) Dijkstra, E.W. (1968), "Go To Statement Considered Harmful," *Communications of the ACM,* Vol 11, No. 3, pp149-150.

(iii) Arthur, L.J. (1983), *Programmer Productivity,* John Wiley, New York.

NOTES

1. PP149/(ii): The 'GOTO' statement as it stands is just too much an invitation to make a mess of one's program. {Complexity, Readability}

2. PP309/(i): Use of GOTO to transfer control between modules couples them almost hopelessly; it makes a mockery of any attempt to deal with modules one at a time, since we cannot even tell under what circumstances any piece of code is entered. {Coupling}

3. PP171/(iii): GOTOs are an unconditional branch to somewhere in the program. They violate every law of structure ever written and make testing the module more difficult. {Complexity, Readability}

USE OF RECURSIVE CODE

Affects:

(a) Complexity (-)

References:

(i) Boehm, B.W. (1984), "Software Life Cycle Factors," In: C.R. Vick and C.V. Ramamoorthy (eds.), *Handbook of Software Engineering,* Van Nostrand Rheinhold Co. NY, pp 494-518.

NOTES

1. PP498/(i): Recursive code increases complexity. {Complexity}

USE OF NEGATIVE/COMPOUND BOOLEAN EXPRESSIONS

Affects:

(a) Complexity (-)

References:

(ii) McCall, J.A., P.K. Richards and G.F. Walters (1977), "Factors in Software Quality," RADC TR-77-369, Vol. 1.

NOTES

1. PP6-43/(i): Compound expressions involving two or more boolean operators and negation can be avoided. These types of expressions add to the complexity of the module. {Complexity}

USE OF EMBEDDED ALTERNATE LANGUAGE

Affect:

(a) Readability (-)

(b) Complexity (-)

References:

Justification:

Use of embedded alternate language forces programmer to change "logic of thought". {Readability}

USE OF CODE INDENTATION

Affect:

(a) Readability (+)

(b) Complexity (+)

References:

(i) Arthur, L.J. (1984), *Measuring Programmer Productivity and Software Quality,* John Wiley, New York, NY.

(ii) Kernighan, B.W. and P.J. Plauger (1978), *The Elements of Programming Style,* McGraw-Hill, New York.

NOTES

1. PP194/(i): Always indent nested code under an IF-THEN-ELSE statement to make the code more readable. {Readability}

2. PP147/(ii): Indent to show the logical structure of the program. {Complexity, Readability}

"EXCESSIVE" USE OF CODE INDENTATION

Affect:

(a) Readability (+)

(b) Complexity (-)

References:

Justification:

1. This can hamper readability in a deeply nested programs, as the code is severely shifted to the right and may have to be split to accommodate margins. This would result in a hinderance rather than aid. {Readability, Complexity}

TLOC vs ELOC

Affects:

(a) Readability (+)

References:

(i) Arthur, L.J. (1984), *Measuring Programmer Productivity and Software Quality,* John Wiley, New York.

NOTES

1. PP /(i): It is possible to write an executable statement in one or more lines. In this case a program with more lines of code is more readable than one where TLOC = ELOC.

(TLOC => Total Lines Of Code)

(ELOC => Executable Lines Of Code) {Readability}

LENGTH OF ROUTINE/MODULE

Affect:

(a) Complexity (-)

(b) Readability (-)

References:

(i) Stevens, W.P., G.J. Myers and L.L. Constantine (1974), "Structured Design," *IBM Systems Journal,* Vol. 13, No. 2, pp 115-139.

(ii) Card, D.N., V.E. Church and W.W. Agresti (1986), "An Empirical Study of Software Design Practices," *IEEE Transactions of Software Engineering,* Vol. 12, No. 2, pp 115-139.

(iii) Stevens, W.P. (1981), *Using Structured Design,* John Wiley & Sons, NY.

NOTES

1. PP120/(i): A problem with large modules is understandability and readability. There is evidence to the fact that a group of about 30 statements is the upper limit of what can be mastered on the first reading listing. {Readability, Complexity}

2. PP266/(ii): Many programming standards limit module size to one page (or 50-60 SLOC). {Readability, Complexity}

3. PP94/(iii): The objective of structured design is to divide programs into pieces that can be handled easily and independently. Psychologists have found that the standard sheet of paper (8.5X11 in.) contains about the amount of information people can deal with comfortably at one time. In other words, one listing page of executable code is a size that can usually be handled easily as a unit. {Complexity, Readability}

TOTAL PROGRAM LENGTH

Affects:

(a) Complexity (-)

References:

(i) Dunsmore, H.E. and J.D. Gannon (1978), "Programming Factors- Language Features that Help Explain Programming Complexity," *Communications of the ACM,* pp 554-560.

(ii) McTap, J.L. (1980), "The Complexity of an Individual Program," *Proc. AFIPS National Computer Conference,* pp 767-771.

NOTES

1. PP558/(i): Program length is important in determining the programming complexity that will be experienced in constructing a program. {Complexity}

2. The features of program that can qualify for use in the measurement of program complexity are:

a) the total # imperative statements
b) the average depth of 'IF' nesting
c) the total # lines of source code
d) the total # entry points
e) the total # boolean variables declared
f) the average # parameters passed
g) the average # SLOC jumped by a forward transfer of control
h) the total # lines of annotation in the source code. {Complexity}

MODULARIZATION

Affects:

(a) Cohesion (+)

(b) Ease of Change (+)

(c) Complexity (+)

References:

(i) Troy, D.A. and S.H. Zweben (1981), "Measuring the Quality of Structured Design," *The Journal of Systems and Software,* Vol. 2, pp. 113-120.

(ii) Tausworthe, R.C. (1977), *Standardized Development of Computer Software,* Prentice-Hall, Englewood Cliffs, NY.

NOTES

1. PP116/(i): The main goal of modularizing a design is to divide the software into pieces that are functionally cohesive and independently modifiable. {Cohesion, Ease of Change}

2. PP76/(ii): I have alluded to the need for modularity in program design as a means toward organizing the program into subdivisions (which can be considered separately) to cope with complexity during the development phase, and to cope with side effects when later changes or corrections are made. {Complexity}

USE OF "EXCESSIVE" NUMBER OF ROUTINES

Affects:

(a) Complexity (-)

References:

(i) Yourdon, E. and L. L. Constantine (1979), *Structured Design,* Prentice Hall, New Jersey.

NOTES

1. PP73/(i): Complexity can be decreased by breaking the problem into modules, so long as they are relatively independent. Eventually, the process of breaking pieces of the system in smaller pieces will create more complexity than it eliminates, because of inter-module dependencies. {Complexity}

USE OF "SANDWICHING"

Affects:

(a) Ease of Change (+)

References:

(i) Parnas, D.L. (1979), "Designing Software for Ease of Extension and Contraction," *IEEE Transactions of Software Engineering*, N0. 3, pp 184-196.

NOTES

1.PP /(i): "Sandwiching" resolves the conflict resulted when two programs want to use each other. {Ease of Change}

USE OF BLOCK COMMENTS

Affects:

(a) Complexity (+)

(b) Readability (+)

References:

(i) McCracken, D.D. (1976), *A Simplified Guide To Structured COBOL Programming,* John Wiley, NY.

NOTES

1. PP149/(i): We recommend sparing use of comments, since a well-written program ideally ought to be understandable without them, but a brief description of the program is a good idea. {Complexity}

USE OF COMMENTS

Affects:

(a) Readability (+)

(b) Complexity (+)

References:

(i) Arthur, L.J. (1984), *Measuring Programmer Productivity and Software Quality,* John Wiley, New York.

(ii) Weissman, L. (1974), "Psychological Complexity of Computer Programs: An Experimental Methodology," *ACM SIGPLAN Notices,* No. 6, pp. 25-36.

(iii) Conway, R., D. Gries and E.C. Zimmerman (1976), *A Primer on Pascal,* Winthrop Computer Systems Series, Cambridge, MA.

NOTES

1. PP100/(i): Comment Density (CD) is defined as follows:

   CD = (# of Comments)/TLOC

Higher the CD, better the readability and less the complexity. {Readability, Complexity}

2. PP27/(ii): It is felt that comments can increase the ability to understand and maintain programs. {Complexity}

3. PP18/(iii): Clarity and precision in defining the role (through comments) of each variable in a program is of vital importance in producing a correct and understandable program. Many programming difficulties can be traced to fuzziness in the meaning of key variables. This approach is aided by following a consistent practice of supplementing the declarations of each variable with comments. {Complexity}

USE OF "EXCESSIVE" # SINGLE LINE COMMENTS IN LINE

Affect:

(a) Readability (-)

(b) Complexity (-)

References:

(i) Arthur, L.J. (1984), *Measuring Programmer Productivity and Software Quality,* John Wiley, New York, NY.

NOTES

1. PP194/(i): Use comments to clarify what the code is doing, never to restate what is already obvious. Too many comments can obscure the executable code, making maintenance difficult. {Readability, Complexity}

USE OF COMMENTS CONSISTENT WITH CODE FUNCTIONS

Affect:

(a) Readability (+)

(b) Complexity (+)

References:

(i) Kernighan, B.W. and P.J. Plauger (1978), *The Elements of Programming Style,* McGraw-Hill, New York.

NOTES

1. PP141/(i): The best documentation for a computer program is a clean structure. It also helps if the code is well formatted, with good mnemonic identifiers and labels (if any are needed), and a smattering of enlightening comments. {Complexity}

2. PP142/(i): The trouble with comments that do not accurately reflect the code is that they may well be believed subconsciously, so the code itself is not examined critically. {Complexity}

GRAMMATICALLY CORRECT COMMENTS/SPELLINGS

Affect:

(a) Readability (+)

References:

Justification:

1. Incorrect comments/spellings can lead to misunderstanding or non-understanding. {Readability}

CERTIFICATION LEVELS IN "COMMENTS"

Affect:

(a) Visibility of Behavior (+)

(b) Early Error Detection (+)

References:

Justification:

1. It tells one what has been tested and what has not been tested. It also provides information about the location of error, thus pinpointing the segment of complex code.

USE OF COMMENTS REFERENCING PROJECT DOCUMENTATION

Affect:

(a) Traceability (+)

References:

Justification:

1. Traceability implies overall relationship to other routines which is enhanced by the comments referencing project documentation.

USE OF COMMENTS REFERENCING "who called me"

Affect:

(a) Traceability (+)

References:

Justification:

1. "Who called me" is otherwise not visible from code alone.

COMPLETENESS/ACCURACY OF DOCUMENTATION

Affect:

(a) Readability (+)

(b) Complexity (+)

(c) Ease of Change (+)

References:

(i) Kernighan, B.W. and P.J. Plauger (1978), *The Elements of Programming Style,* McGraw-Hill, New York.

(ii) Tausworthe, R.C. (1977), *Standardized Development of Computer Software,* Prentice-Hall, Englewood Cliffs, NY.

NOTES

1. PP141/(i): A comment is of zero (or negative) value if its wrong. {Readability, Complexity}

2. PP32/(ii): The documentation must describe the program elements not only so that the design analysis and programming functions are exhibited clearly, but also so that management has visibility into the technical, budgetary, and schedule implications of system changes. It must contain a system description that a user can understand- function of the system, rules for use, domain of input, algorithms and procedures that turn input into output, etc. It must tell how the program is to be operated- the system environment, how much storage is used, how fast the program runs, how to load and start after failure, how to keep the program maintained, etc. {Complexity, Ease of Change}

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: SRC-87-007 (CS TR-87-16)
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): Developing an Automated Procedure for Evaluating Software Development Methodologies and Associated Products
5. TYPE OF REPORT & PERIOD COVERED: Final Report 06/19/87 - /9/18/86
6. PERFORMING ORG. REPORT NUMBER: SRC 87-007 (CS TR-87-16)

7. AUTHOR(s): James D. Arthur, Richard E. nance

8. CONTRACT OR GRANT NUMBER(s): N60921-83-G-A165

9. PERFORMING ORGANIZATION NAME AND ADDRESS: Systems Research Center and Department of Computer Science, Virginia Tech, Blacksburg, VA 24061

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Sea Systems Command, SEA61E, Washington, DC 20362
12. REPORT DATE: 15 April 1987
13. NUMBER OF PAGES: 130
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Naval Surface Weapons Center, Dahlgren, VA 22448
15. SECURITY CLASS. (of this report): Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

For internal distribution to the sponsors and to the project personnel.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Methodology Evaluation, Software Engineering Objectives, Software Engineering Principles, Software Engineering Attributes, Linkages.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This research focuses on (1) assessing the perceived strengths and weaknesses of the current procedure for evaluating software development methodologies, (2) basing the evaluation process on statistical indicators rather than "surface" properties of the product, and (3) automating the evaluation process.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)