A Procedural Approach to Evaluating Software Development Methodologies: The Foundation

James D. Arthur
Richard E. Nance
Sallie M. Henry

TR 86-24

A Procedural Approach to Evaluating Software Development Methodologies: The Foundation*

James D. Arthur, Richard E. Nance and Sallie M. Henry

Virginia Tech

Index Terms: Software Development Methodologies, Procedural Evaluation, Evaluating Methodologies, Software Engineering Objectives, Software Engineering Principles, Induced Attributes, Indicators.

1. Introduction

Within the past ten years, the scope of computer science has expanded to include the Software Engineering discipline [22]. Like many new disciplines, the growth of software engineering has been an evolutionary process, beginning with the recognition of beneficial techniques for developing desirable software, e.g., stepwise-refinement [33], modular decomposition [25], and information hiding [20]. Recognizing the fact, however, that the complex systems of today (and those proposed for the near future [23]) require more than just a loose collection of tools and techniques, the software engineering community has continued to investigate the fundamental concepts underlying the software development process. Today, a myriad of tools, techniques, and development methodologies address the challenging task of producing high quality software. For example, SCR [11,17] and DARTS [16] are development methodologies that emphasize specific software engineering goals (reducing software development costs and facilitating the design of real-time systems, respectively). SREM [1] and SADT [30,27] are methodology based *environments*. Both focus on particular phases of the software life cycle and are supported by unified sets of complementary tools.

* Work supported by the U.S. Navy through the Systems Research Center under Basic Ordering Agreement N60921-83-G-A165 B003-4.

Although software development methodologies have effected significant improvements in the quality of software, the steady proliferation, growth, and diversity of development approaches now foster new concerns. In particular, it is becoming increasingly difficult to

(1) choose an appropriate methodological approach,

(2) recognize what one should expect from a methodology as opposed to environments, tools and techniques, and

(3) understand the relationship and role of a methodology within a given environment and vice versa.

Adding to these concerns is the question of currently accepted methods and techniques as newer technologies challenge the "conventional wisdom" [10]. Although researchers like Weiss and Basili are investigating individual aspects of the software development process [5,6], research addressing the collective implications, interactions and general adequacy of software development methods and techniques remains rather narrow. In particular, studies that evaluate *methodologies* as a single, cohesive framework are severely lacking. Among those that do treat evaluation, see for example Bergland [8], Sawyer [28] or Kelly [19], the intent is to compare the benefits derived from a methodological approach in contrast with no methodology.

The observations and concerns noted above provide the basic motivation for our research. The primary objective is to formulate a *procedural approach to evaluating software development methodologies.* Because the resulting evaluation procedure is based on research spanning several software engineering subdisciplines, the authors choose to present an *overview* of the evaluation procedure rather than a detailed discussion of the procedure's synthesis. A more in-depth treatment of the research and corresponding results can be found in [2,21].

Because the term "methodology" has many connotations, the next section of this paper addresses the fundamental question – What *constitutes* a methodology? Current literature provides

a basis for the discussion which, in turn, serves as a basis for presenting a systematic approach to evaluating software development methodologies.

In Section 3, the discussion focuses on the major thrust of this paper: a basis for judging the adequacy of a particular methodology or the preferability of one methodology over another. Based on our understanding of what constitutes a methodology, we present a systematic approach to evaluating software development methodologies. This approach embodies three primary objectives:

(1) a *procedural* organization that guides the user through the evaluation process,

(2) *general* applicability (irrespective of paradigmatic focus), and

(3) the generation *useful* information to the user.

In presenting the evaluation process, Section 3 first describes a *natural* set of relationships that exist among commonly accepted software engineering objectives and principles. These relationships are then extended to include software engineering attributes, i.e. software qualities *induced* by the principle-based, software development process. Relationships among objectives, principles and attributes serve as a basis for assessing the adequacy and effectiveness of a software development methodology.

The final two sections of this paper describe the results of an application of the evaluation procedure to two software development methodologies currently used by the U.S. Navy. In particular, Section 4 presents an operational overview of the evaluation process, focusing on steps taken to minimize individual bias during the procedure definition and data collection phases. Section 5 addresses the evaluation results and offers a brief critique.

2. What is a Methodology?

The term "methodology", although widely used, connotes a variety of similar, yet distinct, meanings. Because methodologies permeate many scientific disciplines, definitions abound, largely influenced by specific application domains. Software engineering is no exception; "methodologies"

address various phases of the development process: needs analysis, requirements specification, system design, and program design. Although methodologies are based on a common underlying tenet, i.e. providing a systematic approach to attaining a desired goal, each is indelibly marked by individual elements tailored for application dependent end-products.

2.1 Methods and Methodologies

To describe the term "methodology", a distinction must first be made between it and the term "method". In [14], Peter Freeman provides an excellent discussion of concepts associated with both terms. Several of his major points are paraphrased and extended below.

Simply stated, a *method* describes the means of accomplishing a given task, e.g., developing software. In general, a method specifies three elements:

(1) what decisions are to be made,

(2) how to make them, and

(3) in what order they are to be made.

For example, "top-down" design is a method that describes one approach to designing software.

In contrast with the limited scope of a method, a methodology is a *collection* of complementary methods, and a set of rules for applying them. More specifically, a methodology

(1) organizes and structures the tasks comprising the effort to achieve a global objective, establishing the relationships among tasks,

(2) defines methods for accomplishing individual tasks (within the framework of the global objective), and

(3) prescribes an order in which certain classes of decisions are made, and ways of making those decisions that lead to the overall desired objective.

As methodologies are applied to specific application domains, the associated collection of methods and procedural guidelines change accordingly. In general, software development methodologies should be guided by accepted software engineering principles that, when applied to the defined process, achieve a desired goal.

2.2 Methodologies, Tools, and Environments

Within the software engineering discipline, a clear delineation among tools, environments and methodologies is rarely made. Our view is that an environment is comprised of an *integrated* set of tools, each selected to provide designated functionality. Collectively, the set of tools – the environment – engenders a psychological reaction (a "feeling") among users. Whether this reaction is positive or negative depends on many factors, not the least being: 1) the match between tool functionality and user needs and 2) the degree of tool integration. Both factors are determined to a great extent by the underlying methodology. In essence, an environment dictates *how* one accomplishes a task as well as *what* one can accomplish. Hence, it is natural to view tools as *elements* of an *environment* that *supports* a given methodology.

In summary, individual tools can support methodological approaches to accomplishing a given task. The methodology establishes tool requirements, influences functional design, and most importantly, forms a "blueprint" for achieving consistency and complementarity indicative of an environment.

3. A Procedure for Evaluating Software Development Methodologies

The development of large, complex software systems is considered a *project* activity, involving several analysts and programmers and at least one manager. What then is the role of a methodology in this setting and how does it relate to objectives, principles and attributes? Figure 1 assists in providing an answer to this question.

In general terms, an *objective* is "something aimed at or striven for." More specific to the software development context, an objective pertains to a *project* desirable – a characteristic that can

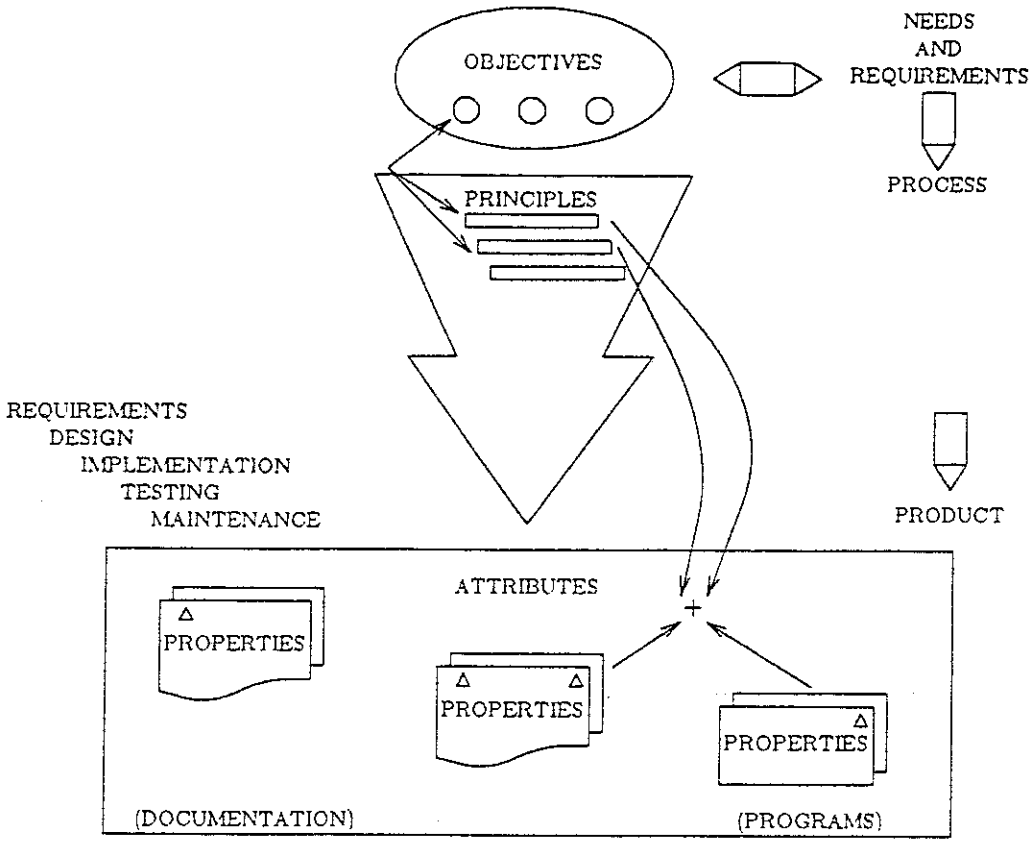

Figure 1

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

be judged as achieved only at the completion of the project. Achievement of a software engineering objective exacts a price – often in terms of other objectives. That is, tradeoffs among objectives are frequently encountered. For example, greater adaptability[1] may be achieved by taking steps that reduce reliability[1]. Even more significant in the development of embedded systems is the potential conflict between *systems engineering* and software engineering objectives. That conflict is underscored in the RADC study, which characterizes system performance as having a negative effect on *every* software engineering factor (to use their terminology) but correctness [26].

[1] Definitions of specific software engineering objectives, principles, and attributes, based on a consensus among several sources [4,5,7,8,11,14,15,18,20,24,25,29,32], are listed in Appendix 1.

A software engineering *principle* describes an aspect of *how* the process of software development should be done. The process of software development, if it is to achieve the stipulated objectives, must be governed by these "rules of right conduct." While the discovery of these rules may be incomplete at this time, a number of guiding principles have been established in the growing body of software engineering literature.

Attributes are the intangible characteristics of the product: the software produced by project personnel following the principles set forth by the methodology. Attributes can be exhibited by each unit of code and documentation although their intangible nature makes it difficult to establish their presence. Unlike objectives, which pertain only to the *total* project activity, attributes may be observed in one unit of the product and absent in another. The claim of presence or absence of an attribute is based on the recognition of *properties*, which contribute evidence supporting the claim. Properties are observable and can be subjective as well as objective. Observation of properties can lead to an accumulation of evidence of attribute presence or absence. Much like civil litigation, the truth of such a claim cannot be established; only through the preponderance of evidence can one reach the decision that a software unit possesses an attribute.

3.1 The Rationale

Influenced by Fritz Bauer's original definition of software engineering [7] and reflecting the above description of software engineering objectives, principles and attributes, the rationale for the evaluation procedure described in this paper is founded on the philosophical argument that:

> The *rasion d'etre* of any software development methodology is the achievement of one or more *objectives* through a *process* governed by defined *principles.* In turn, adherence to a process governed by those principles should result in a *product* (programs and documentation) that possesses *attributes* considered desirable and beneficial.

This philosophy, exemplified by Figure 1, is tempered by practical concerns:

(1) While a set of software engineering objectives can be identified, this set might not be complete, and additions should be permitted.

(2) Objectives can be given different emphasis within a methodology or in applications of a methodology.

(3) Attributes of a large software product might be evident in one component yet missing in another.

(4) The evaluation procedure should admit application to varying levels of detail, consonant with the perceived importance of the decisions motivating the study.

3.2 A Framework for Evaluation

A broad review of software engineering literature [11,15,18,20,24,25,29,32] 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 performance of software over time,

(6) Portability - the ease in transferring software from one host system to another, and

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

The authors note that several of these definitions, as well as others presented in this section, are abridged; they are primarily intended to reflect a *working* definition based on general literature usage. A list of more comprehensive definitions can be found in Appendix 1.

Achievement of these objectives comes through the application of principles supported (encouraged, enforced) by a methodology. The principles enumerated below are extracted from the

references cited above as mandatory in the the creative process producing high quality programs and documentation.

(1) Abstraction - defining each program segment at a given level of refinement.

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

   (*b*) Functional Decomposition – components partitioned along functional boundaries.

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

(3) Stepwise Refinement – utilizing a convergent design.

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

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

(6) 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 those objectives. The important correspondence between objectives and principles is shown in Table 1; a literature confirmation of these relationships is discussed in [3].

Employment of well-recognized principles should result in software products possessing attributes considered to be desirable and beneficial. A short definition of those attributes is given 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 relative to human understanding and/or machine execution.

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

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

(6) Ease of Change - the ease with which software 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 requirements specification and design prior to implementation.

The relationships among attributes and principles are denoted in Table 2. A literature confirmation of each attribute/principle relationship is also presented in [3].

Software attributes represent a collective and subjective judgment of a characteristic. This judgment can be made more objective by the identification of product *properties* that reflect the presence (or absence) of an attribute. For example, the use of subordinate indentation to demarcate control statement ranges contributes to more readable code. As is evident in this example, however, such properties may be highly dependent on the programming language employed in implementation, documentation organization, and formatting conventions; therefore, the definition of properties cannot be generalized but remains specific to the instances of evaluation.

3.3 Defining the Evaluation Process

The software development process, illustrated in Figure 1, depicts a *natural* relationship that links objectives to principles and principles to attributes. That is, one achieves the *objectives* of a software development methodology by applying fundamental *principles* which, in turn, induce

Table 1: Software Engineering Objectives and Corresponding Principles

Objective	*Principles*
Maintainability	Stepwise Refinement
	Concurrent Documentation
	Hierarchical Decomposition
	Functional Decomposition
	Information Hiding
	Structured Programming
Adaptability	Stepwise Refinement
	Concurrent Documentation
	Hierarchical Decomposition
	Functional Decomposition
	Information Hiding
	Structured Programming
Reusability	Concurrent Documentation
	Hierarchical Decomposition
	Functional Decomposition
	Information Hiding
Portability	Functional Decomposition
	Concurrent 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

particular code and documentation *attributes*. From a more detailed perspective, Tables 1 and 2 define the *precise* set of linkages relating objectives to principles and principles to attributes. As described below, these linkages provide a framework for assessing both the *adequacy* of a methodology as well as its *effectiveness*.

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	Ease of change
	Coupling/cohesion enhanced
	Reduced complexity
	Well-defined interfaces
	User-defined data types
Concurrent Documentation	Readable code
	Traceability
	Ease of change
	Reduced complexity
Life-cycle Verification	Visibility of behavior
	Early error detection

The adequacy of a software development methodology can be defined as its ability to achieve the software engineering objectives corresponding to those dictated by system needs and requirements. Intuitively, the adequacy of a methodology is assessed through a top-down evaluation scheme starting with an examination of stated methodological objectives relative to system needs and requirements. This step is then followed by a comparison of stated methodological principles and attributes with those deemed most appropriate. An examination of of linkages defined by the evaluation procedure reveals the "most appropriate" set. With reference to Figure 1, an application of the evaluation procedure to the assessment of methodological *adequacy* entails:

(1) The identification of *objectives* and the relationships tying objectives to needs and requirements is usually accomplished by reading the descriptions of a software development

methodology. Evaluation at this level is quite subjective; however, the absence of a clear statement of objectives for a methodology should trigger an alarm: Is the "methodology" only a tool or an incomplete set of tools without coherent structure? A methodology should not be faulted, however, for emphasizing certain objectives at the expense of others; such prioritization is highly dependent on the application domain.

(2) Based on the objectives emphasized by the methodology and the predefined set of linkages among objectives and principles, the second step in applying the evaluation procedure is an investigation of the software development *process*. That is, given a stated set of methodological objectives, one asks: Are the *principles* supported by the methodology consistent with those deemed necessary (through linkage examination) to achieve the stated set of objectives? The presence of principles without corresponding objective(s) or vice versa should evoke concerns. Although this level of evaluation is inherently subjective, some analytical quality is introduced through the established objective/principle correspondence.

(3) The third step in the evaluation procedure, 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 consequent set of product attributes. Obviously, the expected set of product attributes should correspond to those desired by the software engineer, and to some extent, be implied or stated in the description of the software development methodology. More objectivity is introduced at this level because, although intangible, evidence of the attributes should be discernible in the product.

While a top-down evaluation process reveals deficiencies of a software development methodology, the *effectiveness* of a methodology is assessed through a bottom-up process. As the term implies, the effectiveness of a methodology can be defined as the degree to which that methodology produces a desired result. In particular, the effectiveness of a methodology is reflected by a product's conformance to the software development process defined by that methodology. We

note, however, that elements *independent* of the methodology can influence its effectiveness, e.g. an inadequate understanding and/or use of the methodology. To measure effectiveness, one examines the product for properties that indicate the presence or absence of software engineering attributes, and then compares that computed set of attributes with those espoused by the software development methodology. Again, referring to Figure 1:

(4) Assessing effectiveness entails an examination of the software product (code and documentation) for the presence or absence of attributes. Because attributes are intangible, subjective qualities, the current evaluation is based on defined properties that provide evidence as to the presence or absence of attributes. More specifically, the assimilation and analysis of metric values imply the extent to which particular properties are observed. In turn, this information is used to synthesize the effective set of product attributes. Clearly, the set of attributes determined from a product evaluation should agree with those induced by the corresponding development methodology. Set mismatch can imply an inappropriate software development methodology, an inadequate understanding of the methodology, or perhaps, the failure of users to adhere to the principles advocated by the methodology.

In general, the three levels of examination defined by top-down evaluation establish relationships that convey the extent to which the methodology *supports* perceived needs, requirements, and the software development process. On the other hand, the bottom-up evaluation process reveals how well the methodology is applied in the software development process through the use of *quantitative* measures to support an objective, *qualitative* assessment.

To illustrate the evaluation scheme, consider the single objective, *reusability.* Formally, reusability can be defined as the extent to which a module can be used in multiple applications. Accepting resuability as an objective mandates the inclusion of four principles (hierarchical decomposition, functional decomposition, information hiding, and concurrent documentation) contributing to the realization of that objective. Expanding the single principle, information hiding,

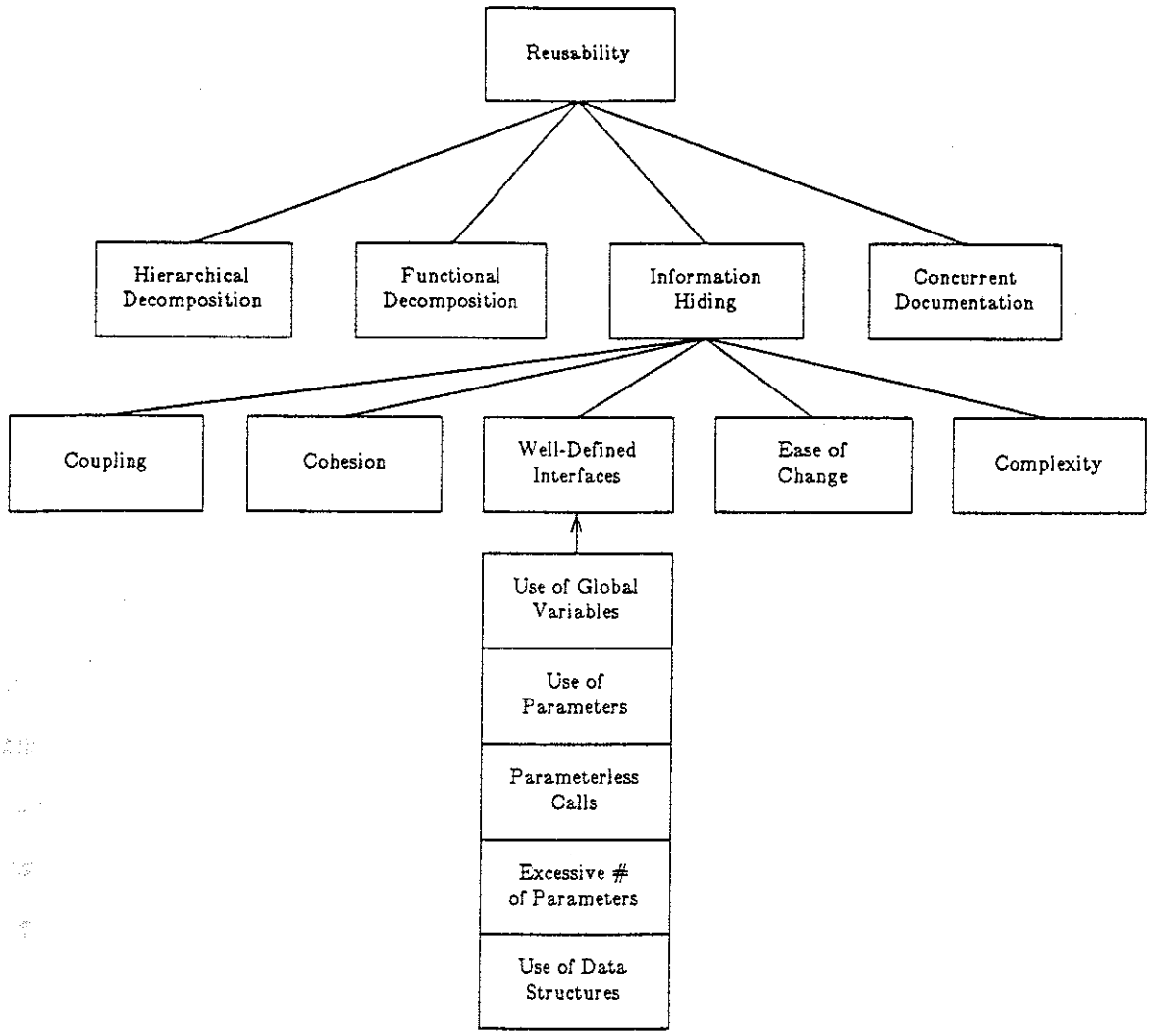

Figure 2

Illustration of the Evaluation Process

we note the five attributes (reduced coupling, enhanced cohesion, well-defined interfaces, 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 interfaces, we identify an additional set of characteristics related to the well-defined interfaces attribute. These characteristics form the set of observable properties which contribute to the claim that a piece of software exhibits a well-defined interface. The relationships described above are illustrated in Figure 2.

4. Operational Specifications

Based on the defined set of linkages among objectives, principles, and attributes, the operational specification of the evaluation procedure is guided by two fundamental axioms:

(1) the methodology description and project requirements provide standards, conventions, and guidelines that *describe how to produce* a product, and

(2) the project documentation, code, and code documentation *reflect how well* the development process prescribed by the methodology is followed.

Axioms 1 and 2 strongly suggest a logical and physical partitioning of the operational procedures that support the top-down and bottom-up evaluation processes, respectively. A major advantage of such a partitioning is that both the top-down and bottom-up evaluation processes can be applied *concurrently*. Basing an examination on the first axiom leads to the identification of the objectives that are stressed, the principles that should be employed to achieve those objectives, and the intended product attributes. Simultaneously, examination of documentation and code reveals the extent to which the set of defined attributes are actually present in the product. The description that follows reflects this partitioning and provides an overview of operational characteristics associated with the complete evaluation process.

4.1 Evaluating the "HOW TO"

System requirements define constraints on software development. A methodology should provide a necessary and sufficient collection of techniques, methods, and guidelines to produce a satisfactory product. Hopefully, these are embodied in an environment based on the methodology. The evaluation begins with the identification of the methodological objectives and relating them to needs and requirements. This task is accomplished by first examining the appropriate document(s) for a clear statement of objectives which are assumed to correspond with the needs and

requirements. Additional examination of the methodology documentation should also yield a set of espoused principles that characterize the process for achieving the stated objectives. Although determining this set of principles involves subjectivity, the process of determining their adequacy is made significantly more objective by examining the appropriate set of objective/principle linkages defined by the evaluation procedure. Using a similar set of linkages defined among principles and attributes, one can then formulate a set of *expected* product attributes.

4.2 Evaluating the "HOW WELL"

A software development methodology describes the process for producing a product; the intrinsic "value added" and the extent to which that process is followed, however, is reflected in the actual product attributes. Examination of the product yields a set of actual product attributes. The level of correspondence between this set and the set of expected attributes indicates, to a large extent, the *effectiveness* of the methodology. Moreover, as later illustrated in Section 5, the actual principles employed in the product development process and the objectives attained can also be determined from the set of measured product attributes and the defined linkages relating those attributes to principles and objectives. Measuring the product attributes, however, requires the definition of product properties, and a measurement technique to evaluate them. As discussed below, product metrics quantify product properties, providing a basis for evaluating attributes of the development process.

Currently, subjective and objective product metrics are computed from two types of elements – logical and numerical.

- *Logical elements* are the most subjective; "yes", "no", and "sometimes" are typical responses. Determination for these elements are obtained from questions such as:

  (1) Is there an alternate language embedded in the source code?

  (2) Are the variable names meaningful?

- *Numerical elements* are easier to verify and much less subjective. They are constructed from values such as:

  (1) the number of executable source lines of code, and

  (2) the number of global data items referenced.

Measurements of both types of elements support the collection of many different data points; results of those measurements contribute to the calculated metrics.

In general, metric computations can be divided into three major categories based on criteria that support the measurement process.

(1) *Established baselines.* These are well defined metrics based on empirical studies; they are generally accepted by the software engineering community. For example, 30 lines of executable code is considered an optimal module length [31].

(2) *Known beneficial or detrimental features.* These metrics, although having no quantitative baseline values, do have a known qualitative effect on the product. For example, the use of a "goto" is known to reduce code readability [12].

(3) *Relative usage.* The extent to which alternate language features are used to accomplish a similar task provides a basis for the third approach to metric computation. Consider for example, inter-module communication. The comparatively high use of global variables rather than other communication forms (e.g., parameter calls, semaphores, etc.) has an adverse effect on module coupling.

The calculation of metrics within the above framework is used to determine the extent to which properties are evident in a product. As illustrated in Figure 3, assessing the presence or absence of product attributes is based on the accumulation of properties. The measured value is currently restricted to an arbitrarily chosen 1-10 scale. Moreover, because product attributes reflect principle

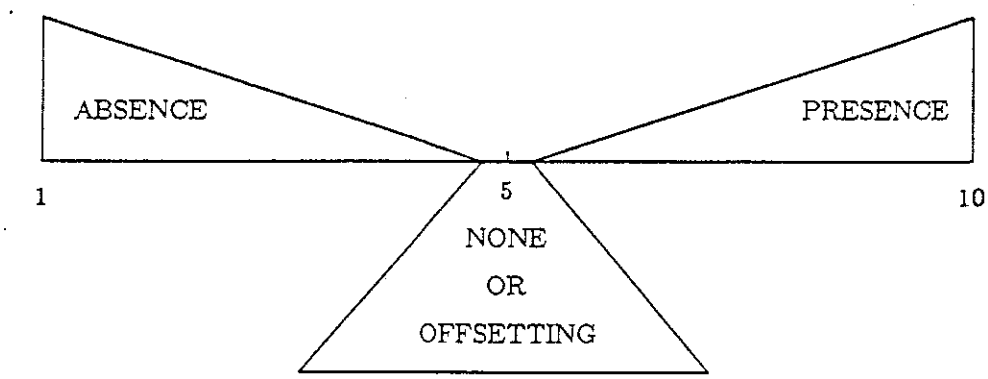

Figure 3

Assessing the Presence or Absence of Attributes

utilization and the achievement of objectives, the extent to which they are evident in the product is also measured most naturally on the same 1-10 scale. Hence, regardless of whether one is considering a desirable attribute, principle, or objective, an assessed value at the lower (higher) end of the scale quantitatively indicates its perceived absence (presence) in the product.

A disadvantage of using properties in attribute assessment, however, is that a visually based, manual process for collecting the metrics is sensitive to observer bias. *A priori* recognition of this fact, however, stimulates the design of data collection techniques aimed at minimizing bias; such is the case for the study presented in the next section. Additionally, a current investigation that focuses on the use of *statistical indicators* in lieu of properties appears to offer a more definitive assessment with less bias.

The operational aspects of applying the evaluation procedure are more involved than conveyed here. Length restrictions, however, prevent a more thorough discussion. For additional information the authors refer the interested reader to [21]. A formal paper that focuses more on the operational experience in using the defined evaluation procedure to analyze a software development methodology is forthcoming.

5. Application of the Evaluation Procedure

The following two subsections illustrate the utility and intrinsic prower of the evaluation procedure in assessing the adequacy and effectiveness of a methodology. Provided in this illustration is a characterization of the components used in the evaluation process, an individual assessment of two methodology descriptions, an analysis of associated products, and a summary of the results.

5.1 Data Sources

A joint investigation of two comparable Navy software development methodologies and respective products is described in [21]. 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 the results and illustrates the utility and versatility of the procedural approach to evaluating software development methodologies. For simplicity, we 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
Concurrent Documentation		Yes
Life Cycle Verification		
Attributes	None	None

Figure 4

Pronounced Objectives, Principles, and Attributes

5.2 Analyzing the Methodological Description and Associated Product

The initial step in the evaluation process is to perform a "top-down" analysis of methodologies A and B, to reveal the set of *claimed* software engineering objectives, principles, and attributes. Because both methodologies have experienced evolutionary development, a *clear* statement of their respective methodological objectives is lacking. Nonetheless, as detailed in Figure 4, 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.

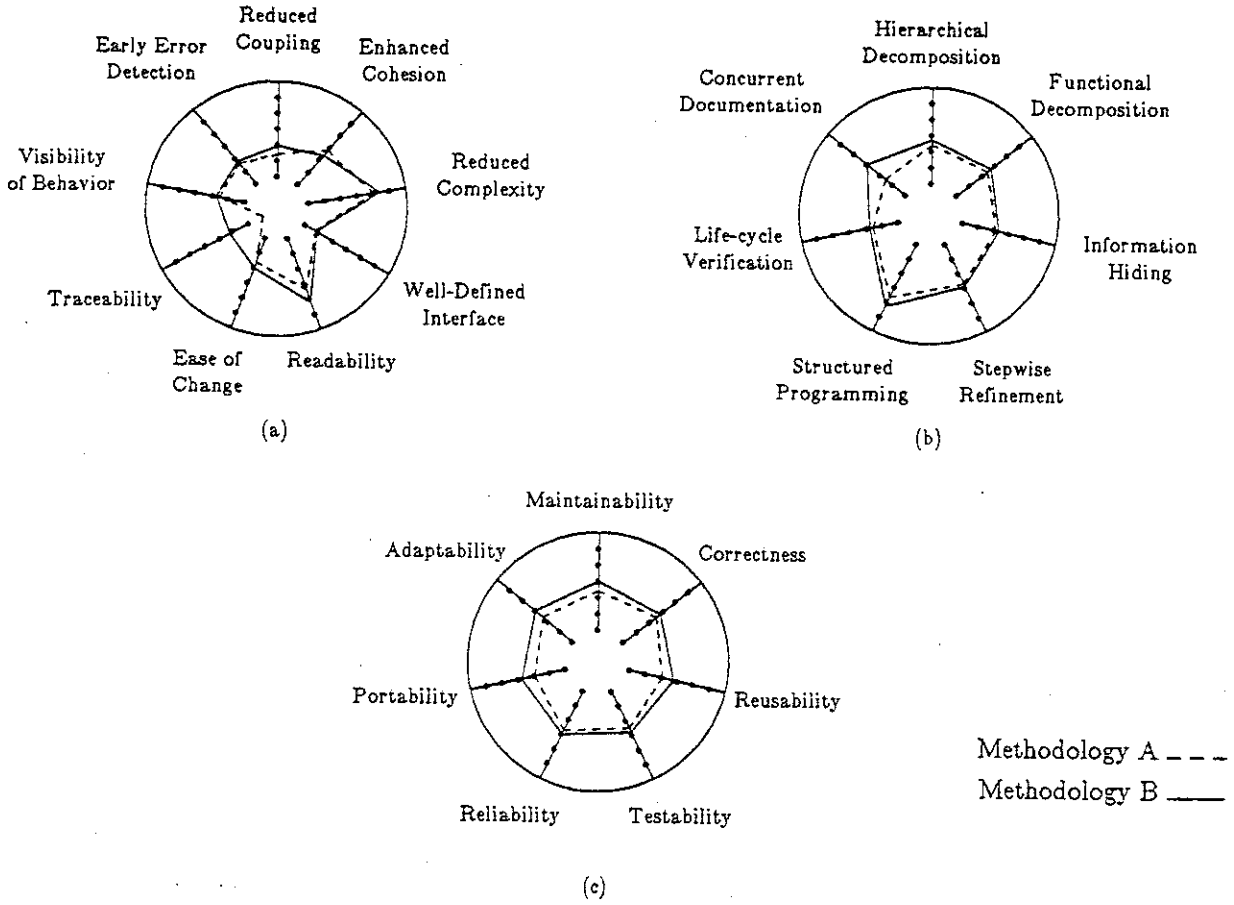

Figure 5

Detected Presence of Objectives, Principles, and Attributes

Using metric values and properties, a corresponding "bottom-up" examination of product A provides some interesting results. The Kiviat graph displayed in Figure 5a 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 5b 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 5c depicts the results of emphasizing these principles in terms of methodology objectives. 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 5a, 5b and 5c, 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 *concurrent 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 concurrent documentation (7.0). Moreover, the evaluation also indicates that the implicitly assumed principles of methodology B are 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 5a, 5b, and 5c.

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 [21].

6. Conclusion

Tools, techniques, environments, and methodologies dominate the software engineering literature, but relatively little research in the evaluation of methodologies is evident. This work reports an initial attempt to develop a procedural approach to evaluating software development methodologies. Prominent in this approach are:

(1) the definition of "methodology", in contrast with tools, techniques, and environments,

(2) the development of a procedure based on the linkage of objectives, principles, and attributes,

(3) the structuring of levels of evaluation in recognition of the practical constraints of time and money that can vary widely depending on evaluation needs, and

(4) establishment of a basis for reduction of the subjective nature of the evaluation through the introduction of properties.

The application of the evaluation procedure to the two Navy methodologies and associated products serves as an example of the first and second levels of evaluation, pointing to attributes that should be evident in the source code and documentation examined at the fourth level. The consistent results demonstrate the utility and versatility of the evaluation procedure. The incorporation of indicators and the consequent refinement of the evaluation procedure offers promise of a flexible approach that admits to change as the field of knowledge matures. In conclusion, the procedural approach presented in this paper represents a promising path toward the end goal of objectively evaluating software engineering methodologies.

References

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

2. Arthur, J.D., Henry, S.M. and Nance, R.E., "Immediate Software Development Issues for Embeded Systems Applications in Surface Combatants," Technical Report SRC-85-009, Systems Research Center, Virginia Tech, 1985.

3. Arthur, J.D. and Nance, R.E., "Developing an Automated Procedure for Evaluating Software Development Methodologies and Associated Products," Technical Report TR-87-16, Department of Computer Science, Virginia Tech, 1987.

4. Bailey, J.E., and Pearson, S.W. "Development of a Tool for Measuring and Analyzing Computer User Satisfaction," *Management Science*, Vol. 29, No. 5, May 1983, pp. 530-545.

5. Basili, V.R. and Weiss, D.M. "Evaluation of a Software Requirements Document by Analysis of Change Data," *Proceedings 5th International Conference on Software Engineering*, March 1981, pp. 314-323.

6. Basili, V.R. and Weiss, D.M. "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering*, Vol. 11, No. 2, February 1985, pp. 157-168.

7. Bauer, F.L. "Software Engineering," *Information Processing 71*, North Holland Publishing Company, 1972.

8. Bergland, G.D. "A Guided Tour of Program Design Methodologies," *Computer*, Vol. 14, No. 10, October 1981, pp. 13-36.

9. Cain, S. and Gordon, E., "PDL – A Tool for Software Design," *Proceedings of the 1975 National Computer Conference*, Vol. 44, 1975, pp. 271-276.

10. Card, D.N., Church, V.E., and Agresti, W.W., "An Empirical Study of Software Design Practices," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986, pp. 264-271.

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

12. Dijkstra, E.W., "Go To Statements Considered Harmful," *Communications of the ACM*, Vol. 11, No. 3., March 1968, pp.147-148.

13. Estrin, G., *et. al.*, "SARA, (System ARchitect's Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986, pp. 293-311.

14. Freeman, P., "The Nature of Design," *A Tutorial on Software Design Techniques*, Second edition, IEEE Computer Society Press, 1977, pp. 29-36.

15. Gaffeney, J. E., "Metrics in Software Quality Assurance," *Proceedings of the National ACM Conference,* November 1981, pp. 126-130.

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

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

18. Jackson, M., *Principles of Program Design*, London: Academic Press, 1975.

19. Kelly, John, "A Comparison of Four Design Methods for Real-Time Systems," *9th International Conference on Software Engineering,* Monterey, CA, March 1987, pp. 238-252.

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

21. Nance, R.E., Arthur, J.D. and Dandekar, A.V. "Evaluation of Software Development Methodologies," A Final Report of the Immediate Software Development Project, Systems Research Center, Virginia Tech, December 1985.

22. Naur, P. and Randell, B., (eds.) *Software Engineering*, Brussels: NATO Science Committee, 1969.

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

24. Parnas, D., "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 1, March 1976, pp. 1-9.

25. Parnas, D., "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM* , Vol. 15, No. 5, May 1972, pp. 330-336.

26. RADC, "Specification of Software Quality Attributes," Rome Air Development Center Technical Report RADC-TR-85-37, Vol. 1 (of three), Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, New York, February 1985.

27. Ross, D., "Structured Analysis: A Language for Communicating Ideas," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, January, 1977, pp. 16-34.

28. Sawyer, S.K. and Dawson, C.A. "Practical Application of Software Engineering Theorems and Methodologies to Software Systems Design Problems," UCCEL Corporation, Dallas, Texas, November 1984.

29. Scott, L., "An Engineering Methodology for Presenting Software Functional Architecture," *Proceedings of the Third International Conference on Software Engineering* , NY, 1978, pp.222-229.

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

31. Stevens, W.P., Myers, G.J., and Constatine, L.L., "Structured Design," *IBM Systems Journal*, Vol. 13, No. 2, pp. 115-139.

32. Warnier, J. *Logical Construction of Programs*, 3rd edition, trans. B. Flanagan, NY: Van Nostrand Reinhold, 1976.

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

Appendix 1

Abstraction - the definition of each program segment at given levels of refinement, and in terms of its relation as a unit to other program segments.

Adaptability - the ease with which software allows differing system constraints and user needs to be satisfied.

Cohesion - the degree to which the tasks performed by a single program module are functionally related.

Complexity - the degree or complication of a system or system component, determined by such factors as the number and intricacy of interfaces, the number and intricacy of conditional branches, the level of nesting, the types of data structures, and other system characteristics.

Correctness - the extent to which software is free from design defects and coding defects; that is, fault free.

Coupling - a measure of the interdependence among modules in a computer program.

Concurrent Documentation - the management of documents which may include the actions of synthesis and modifications to reflect current product status, document identification and acquisition, and document processing, storing, dissemination.

Function - a special purpose of an entity or it's characteristic action.

Functional Decomposition - a method of designing a system by breaking it down into it's components along functional boundaries.

Hierarchical Decomposition - a method of designing a system by breaking it down into it's components through a series of top-down refinements.

Hierarchy - a structure whose components are ranked into levels of subordination according to a specific set of rules.

Information Hiding - the technique of encapsulating software design decisions in modules in such a way that the module interfaces reveal as little as possible about the module's inner workings; thus, each module is a "black box" to the other modules in the system. The discipline of information hiding forbids the use of information about a module which is not in the module's interface specification.

Interface - a shared boundary to interact or communicate with another system component.

Maintainability - the ease with which maintenance of a functional unit can be performed in accordance with prescribed requirements.

Modular Decomposition - a method of designing a system by breaking it down into modules.

Modular Programming - a technique for developing a system or program as a collection of modules.

Modularity - the extent to which software is com- posed of discrete components.

Module - a logically separable part of a program.

Portability - the ease with software can be transferred from on computer system or environment to another.

Reusability - the extent to which a module can be used in multiple applications.

Software Development Cycle - the period of time that begins when a software product is conceived and ends when a product is no longer being enhanced by the developer.

Software Life Cycle - the software life cycle typically includes a requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and some- times retirement phase.

Software Quality - the degree to which a customer or user perceives that software meets his composite expectations.

Software Reliability - an attribute of software quality. The extent to which a program can be expected to perform its intended function.

Stepwise Refinement - a system development methodology in which data definitions and processing steps are defined broadly at first and then with increasing detail.

Structured Design - a disciplined approach to software design which adheres to a specific set of rules based on principles such as top-down design, stepwise refinement, and data flow analysis.

Structured Programming - a well-defined software development technique that incorporates top-down design and implementation, strict use of structures program control constructs, and often, use of chief programmer teams.

Testability - the extent to which software facilitates both the establishment of test criteria and the evaluation of the software with respect to those criteria.

Top-down - pertaining to an approach which starts with the highest level component of a hierarchy and proceeds through the progressively lower levels.

Top-down Design - the process of designing a system by identifying its major components, decomposing them into their lower level components, and iterating until the desired level of detail is achieved.

Validation - the process of evaluating software at the end of the software development process to ensure compliance with software requirements.

Verification - the process of determining whether the products of a given software development phase fulfill the requirements established during the previous phase.