Requirements for a Software Maintenance Methodolgy

Richard E. Nance, James D. Arthur, and Benjamin J. Keller

TR 90-4

Technical Report SRC-90-001*

Requirements for a Software Maintenance Methodology

Interim Report: Subtask 1

Richard E. Nance
James D. Arthur
Benjamin J. Keller

Systems Research Center
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061-0251

18 January 1990

*Cross referenced with TR CS-90-4, Department of Computer Science, Virginia Tech.

This report was supported by the Naval Surface Warfare Center under contract #N60921-83-G-A165 B048

ABSTRACT

Software maintenance, although widely recognized as the most costly period in the life of a system, is given only passing consideration in life-cycle models. An extensive literature review shows the relationship between the development and maintenance phases to be ignored to a large extent. The Abstraction Refinement Model (ARM) describes the dependency of software maintenance on the quality of development documentation and depicts the adaptive and perfective maintenance forms as relying on earlier design and requirements documents to a greater degree than corrective and preventive maintenance. The ARM is effective in laying the foundations for a software maintenance methodology, particularly in explaining the role of reverse engineering. Coupling the ARM with the Objectives/Principles/Attributes procedure for the evaluation of software development methodologies proves effective in drawing the contrast with maintenance requirements, which are specifically identified for further study and assessment.

CR Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.7 [**Software Engineering**]: Distribution and Maintenance.

General Terms: Documentation, Management.

Additional Key Words and Phrases: Methodologies, reverse engineering, model, maintenance.

Table of Contents

1. Background

Prior research [NANR89, pp. 3-4] as well as other sources [LINI88, p. 241; RIDM88] have confirmed the tendency of the software engineering research community to ignore software maintenance, while at the same time acknowledging that it is by far the most costly phase in the life-cycle. This report, the first in a set to be produced under N60921-83-G-165 B048: Development of an AEGIS Maintenance Methodology, is intended as one step toward rectifying this oversight.

1.1. Task Organization

The project organization for the development of an AEGIS maintenance methodology identifies three subtasks:

1. Definition of maintenance methodology requirements, investigating the differences in approaches, techniques, and methods that contrast the maintenance and development phases of the software life-cycle.
2. Development of a model of the AEGIS maintenance process, based on earlier and on-going work by NSWC and contractors, and through active guidance and participaiion of an Advisory Panel.
3. Specification of the maintenance methodology based on the general requirements for maintaining time-critical embedded systems but also recognizing specific needs of the AEGIS application domain.

This report describes the work performed during Subtask I, which extends over the period from 16 August to 15 December 1989.

1.2. Approach

Underpinning this work is the research conducted in a prior project exploring software development documentation potentials using recent technological advances, such as hypermedia, mass storage technology, and knowledge representation. That project, "Documentation Production and Analysis Under Next Generation Technologies," (N60921-83-G-A165 B043-01), proposes the Abstraction Refinement Model (ARM) as a basis for depicting iterative refinement of development documentation in the specification of a software system. The ARM demonstrates the critical importance of development documentation in the early maintenance activities and characterizes the objectives of reverse engineering more clearly than any model to date. Linking maintenance to development more explicitly than prior models, the ARM also underscores the necessity for a software maintenance methodology for time-critical embedded systems, which typically have long lifetimes.

In bringing such a methodology into existence for the AEGIS Combat System, the approach taken in Subtask 1 has centered on an extensive analysis of software engineering literature, seeking to:

(1) compare and contrast (with emphasis on the latter) the activities of development and maintenance,

(2) recognize explicit treatments of software maintenance for time-critical embedded systems, and

(3) derive requirements for an "ideal methodology" that could form the target for a process model and an AEGIS maintenance methodology to be developed by the Advisory Panel.

This third objective is most important. The project team does not intend to develop the AEGIS maintenance methodology; rather, its function is to facilitate that development by those who know the application domain the best.

1.3. Report Organization

A comprehensive description of software maintenance is provided in Section 2, which includes some material from the final report of the prior project cited above. This material is supplied for completeness. Section 3 begins with a definition of "methodology," emanating from software development research that stipulates the importance of software engineering principles and objectives. The major influencers of requirements for a software maintenance methodology are identified in Section 4. The methodology requircments are listed in Section 5, followed by a brief summary and conclusions in Section 6.

2. Views of Software Maintenance

The definition of "maintenance methodology" requires a clear understanding of software maintenance. Particularly important is an understanding of the need to govern or manage the maintenance activities through the use of a methodology. The following section reviews the definitions and descriptions of software maintenance that are found in the software engineering literature.

2.1. Definitions of Software Maintenance

The difficulty with the term "software maintenance" is the implication which comes from the original usage of "maintenance." We think of maintenance as performing repairs on hardware, or changing parts (such as replacing the oil filter, or changing the oil in a car's engine). The need for maintenance is generally predicated on wear or aging of mechanical parts (certainly true of the oil filter). A natural extension of this view to software implies that "wear" or "deterioration" are factors in the need for software maintenance. Clearly, in the strict physical sense, wear brought about by software use does not occur.

What then is the "wear" of software? Software does not have the physical properties of hardware; it may be stored or represented using physical media, but the medium is *not* the "software." Software is a logical object which cannot suffer from use or entropy (beyond that caused by forgetfulness or erasures of magnetic media). Hence, the extension of the root causes of hardware maintenance to software seems unwarranted.

Perhaps the original perspective of maintenance requires adjustment. Consider, for example, a car engine as a functional mechanism. Each part fulfills some sub-function of the engine. The oil filter has the function of removing impurities from the oil. When prolonged usage of an oil filter contributes to increased accumulation of particles in that filter, the filter decreasingly fulfills its function, and must be replaced. From this perspective maintenance is performed because certain parts of the engine are not adequately performing their functions.

This latter perspective extends more naturally to software. A software system is initially described in terms of desired properties (including functionality), which are to be evident in the behavior of a subsequent realization. The development process, through a number of specification transformations, produces that which should demonstrate the desired properties. Failing to demonstrate the desired properties is the cause for software maintenance.

Why software does not exhibit a desired property is another question. A car's engine might not function correctly because of wear and aging. However, we have noted that this type of problems does not occur in software. From a software perspective, though, two root causes underlie need for maintenance: the software is not built properly in the first place (or other maintenance is performed improperly), or the desired properties have changed. These causes also impact how maintenance is to be performed, which is a subject for subsequent discussion.

Understanding that software and hardware maintenance are differentiated by their root causes, the definition of maintenance can now be stated. The essential points are found in several references [GAMS88], [FLEN88], [CANG86], [LINI88]:

(1) maintenance is the process of making changes to existing software systems,
(2) changes are made to the software to induce the desired properties, and
(3) the changes should be consistent with respect to existing system requirements and to each other.

Effectively, software maintenance is the set of activities which make changes to software to better match the desired system. While a methodology describes how these changes are to be made, process and life-cycle models describe changes in the context of the software life.

2.2. Life-Cycle Descriptions

Descriptions of software evolution are referred to as "life-cycle" models. These models try to represent the process by which a software system evolves. Included are the activities which recur throughout the life of the system (leading to use of the term, "cycle"). Technically, these models should make no mention of either development or maintenance, but focus instead on the abstract activities which occur in both. Despite this, most life-cycle models tend to ignore maintenance [SCHN87].

Criticism focusing on the tendency to ignore maintenance is well deserved. Consider, for example, the waterfall life-cycle model [BOEB76]. In the waterfall life-cycle model the process "flows" down through a number of development stages into the "final" stage of maintenance (Figure 1). This model, although touted as the classical "life-cycle" model has no cyclic aspect. Instead the model shows the "complete" process of development with maintenance added almost as an afterthought. The de-emphasis on maintenance is representative of a lack of understanding of what maintenance is and its significance, especially when even the most conservative estimates project that software maintenance contributes over 67% of the life-cycle costs [ZELM79].

Although other life-cycle models do meet our expectation of being cyclic, they are often deficient in other ways. Models such as Gidding's Domain Dependent software life-cycle model [GIDR84], and Boehm's Spiral Model [BOEB86] portray the cyclic nature, but focus primarily on development activities, assuming that these are also adequate to describe maintenance. As expressed by Peters [PETL89] this view implies that maintenance is a "miniature development cycle." Although permitted to continue, this perception is recognized as being incorrect [CHAN88].

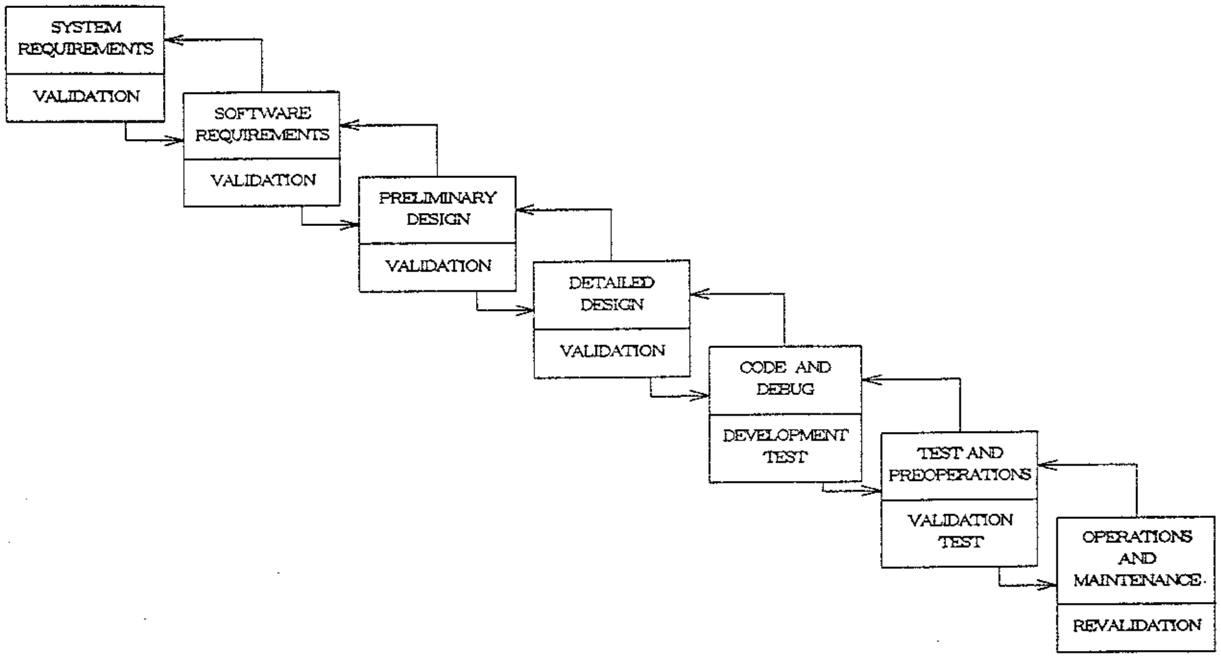


Figure 1. Waterfall model of the Software Life-cycle.

Models like the ones mentioned above are deficient in two significant areas. First, they emphasize development over maintenance. As systems continue to age, the impact of the development phase lessens and its relative importance decreases. Secondly, the models lack an explicit connection between the development and maintenance activities. These models represent a lack of understanding as to the impact that development exerts on maintenance. To some extent these deficiencies can be explained by a lack of understanding of maintenance and maintenance activities.

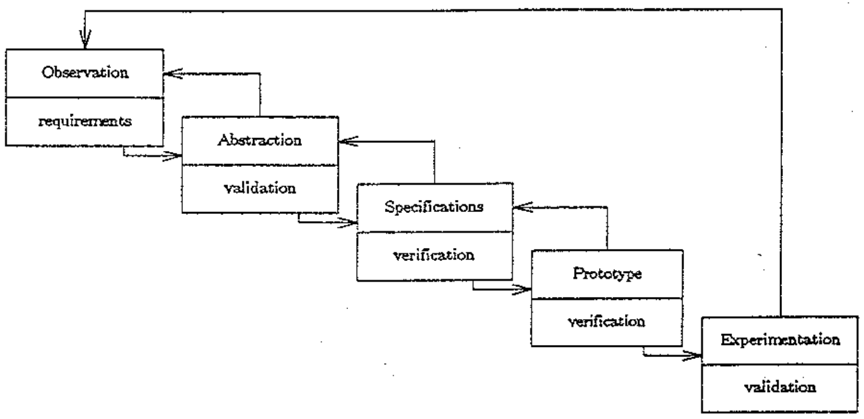


Figure 2. Gidding's Domain dependent software life-cycle model.

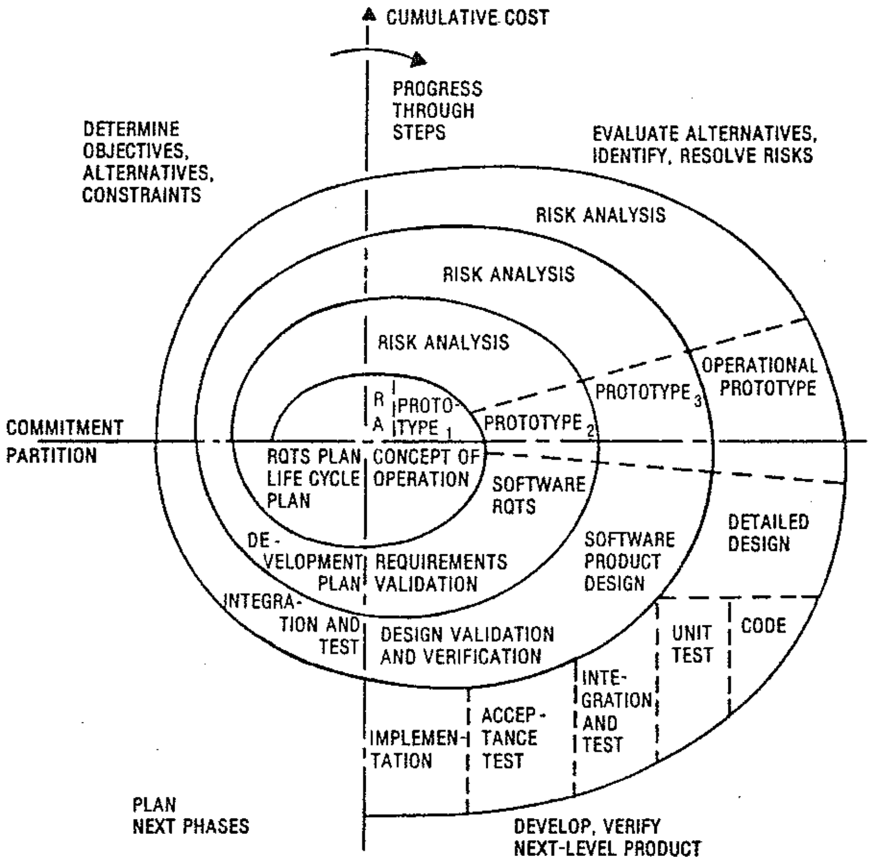


Figure 3. Boehm's Spiral Model.

2.3. The Maintenance Forms

In the attempt to better understand what maintenance is and how it should be approached, a number of maintenance forms have been identified. The classic forms are derived from the relationship between the software product and the system requirements. More recent forms are process oriented or motivated by the types and methods of changes made. Both classes of maintenance forms are considered, but the requirements-driven forms appear to have the most impact on maintenance methodologies. Unless stated otherwise, the remainder of this report is confined to the requirements-driven forms.

2.3.1. Requirements Driven

The classic forms of maintenance are categorized as: corrective, preventive, perfective, and adaptive [SWAE76]. The first three forms are characterized by the system requirements remaining constant, where the maintenance effort focuses on improvements consistent with those requirements. More definitively, they are:

(1) corrective - the correction of faults in a software system

(2) preventive - the addition of facilities to make a software system more robust, and

(3) perfective - the improvement of functionality of a software system to better meet the current requirements.

The fourth form addresses changes in the system requirements, i.e.,

(4) adaptive - the addition of functionality to a software system to accommodate changing requirements.

The four forms seem to partition maintenance activities clearly. Because people often argue that preventive maintenance is in some sense perfective, literature citations sometimes list only three forms of maintenance.

We find it convenient to combine maintenance forms. In particular, both corrective and preventive are code-level (or behavioral level) activities requiring an algorithmic perspective; perfective and adaptive, however, are more pervasive, requiring a design-level or higher perspective of the system.

2.3.2. Process Driven

Other categorizations of maintenance have been offered. These categorizations are concerned with "how" maintenance activities are to be carried out. Derived directly from the maintenance process, these are termed "process-driven forms."

One set of process-driven forms is defined by Gamalel-Din and Osterweil as "software alteration processes," [GAMS88]. These forms are denoted as "backbone maintenance," "spare parts maintenance," "black box maintenance," and "copy and adapt."

Backbone maintenance changes the software without altering its structure. To accomplish backbone maintenance the code and the effects of the changes must be understood. Without this understanding, changes can be both difficult and disastrous.

Spare parts maintenance is much like hardware maintenance, in that a new component is substituted for an old one. The new and old software components must have identical functionality and interfaces. Internal details of the components are not needed to perform this form of maintenance.

Black box maintenance involves the development of new software which cleanly interfaces with the old system. Understanding how the old system interacts with the new components is sufficient.

Copy and adapt maintenance uses existing software components which conform closely to the needs of a new version. Components must be closely analyzed for side effects (via non-local variables) to avoid problems.

In addition to the above, "design recovery" is sometimes listed. This however, is indeed a "process" rather than a maintenance form. Design recovery is a specialized form of reverse engineering used to capture missing design information from the code.

Another discussion of process-driven maintenance forms comes from Lin and Gustafson [LINI88]. The forms of maintenance defined in [LINI88] are characterized in terms of the number of additions, alteration and deletion made to the software product. The categories include corrective, adaptive, "retrenchment," "retrieving," pretty-printing, and documentation. Corrective and adaptive are the same as Swanson's [SWAE76]. Retrenchment is the "commenting-out" of statements, and retrieving the "un-commenting in" of these statements. The others are self-explanatory. Perfective maintenance is not included because they assert that the intent of maintenance is not well represented by metrics.

Although this classification of maintenance forms is interesting, it does not seem as informative as the process-driven forms discussed earlier. The characterization of maintenance with respect to the effect the changes have on the software structure seems preferable.

2.4. Models of the Maintenance Activities

Earlier discussion of life-cycle models describes them as deficient in their treatment of maintenance. Other descriptions of the maintenance process are more suitable. In general, these are models of "software evolution" rather than pure maintenance. Described below are the Re-engineering Cycle of Bachman [BACC88], the process-centered software life-cycle paradigm [GAMS88], the "maintenance paradigm" of Wild and Maly [WILC88], the integrated life-cycle model of Yau [YAUS88], and the Abstraction Refinement Model [NANR89][KELB90].

The Re-engineering Cycle is actually another life-cycle model. It has both forward engineering and reverse engineering aspects. The forward engineering side is much like the water-fall model and leads from specification to system operation. The reverse engineering side allows the "reversal" of forward engineering, "lifting" the information from lower levels to higher ones (see Figure 4). The intention of the re-engineering cycle is to have tools which support both forward and reverse activities [BACC88].

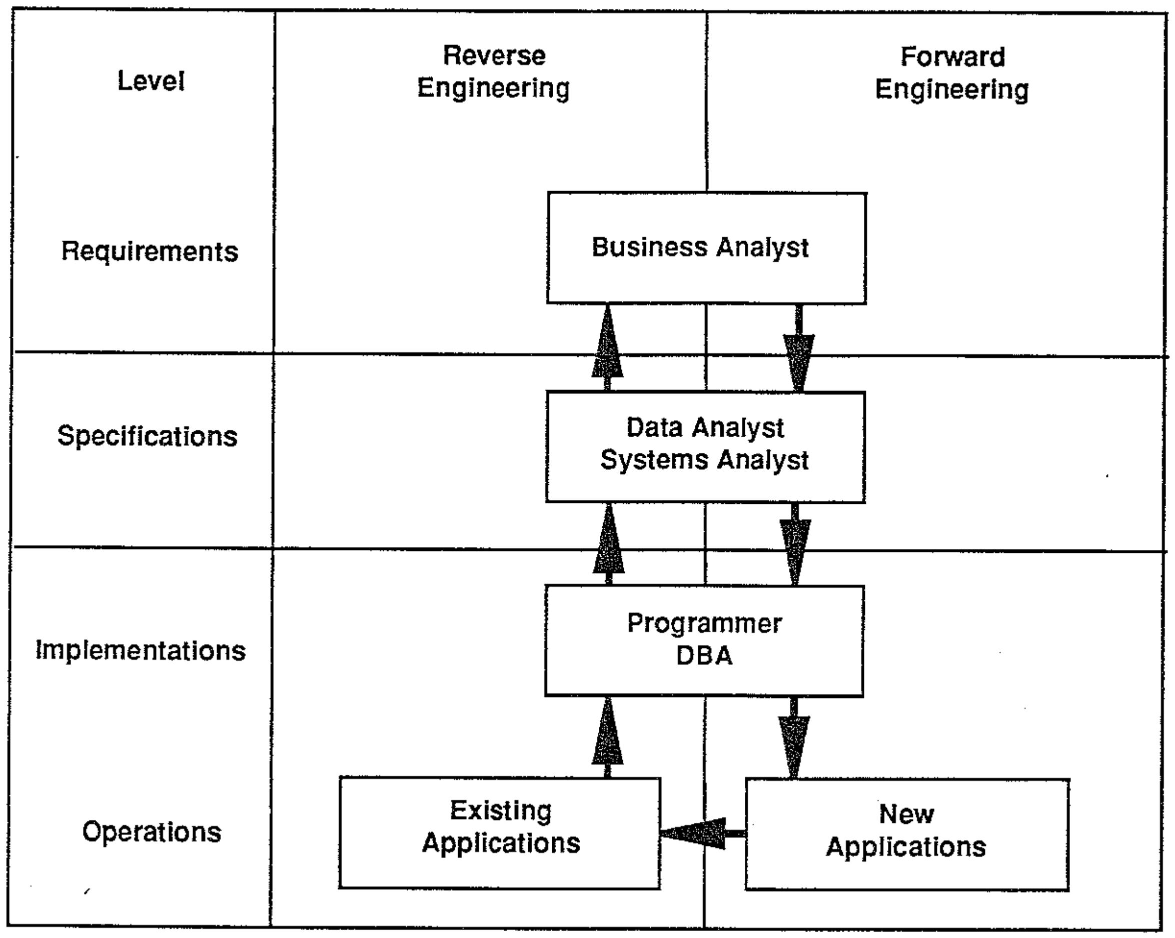


Figure 4. Re-engineering Cycle.

The process-centered software life-cycle paradigm [GAMS88] is more complicated than the others. The basis for the model is "process programming," through which the development and maintenance process is integrated with a tailored environment. (We note that the authors confuse the distinction between methodology and process, and appear to be referring to process when methodology is more appropriate.) The model implies that the process should be well-defined and completely constructed prior to the development of the software. The model has three connected loops: the process static cycle, in which the process is developed; the product cycle, in which the process executes to act on the product; and the process/product dynamic cycle, in which either the process or product can be modified while still executing.

Wild and Maly [WILC88] suggest the "maintenance paradigm based on closure." By closure they mean completeness of information relative to a task. Each step in their paradigm is complemented by a closed (complete) set of information. The requisite steps are understanding, analysis and design, and implementation. In the understanding phase each participant obtains the understanding necessary to perform the designated task (this information differs among manager, analyst or programmer). During the analysis and design phase, the design is optimized to balance the factors contributing to the design. The implementation phase realizes the design, reusing existing code when possible. Wild and Maly's main goal is to define an environment which supports this paradigm, emphasizing the capture and documentation of design decisions. The combination of the paradigm with the environment supports explicit representation of the maintenance process.

Explicit representation of the process is an important aspect of the final two models. Both describe the maintenance process in terms of transformations on descriptions of the system. In the integrated life-cycle models, graphs are used to represent the system and graph rewriting dominates the process representation of maintenance [YAUS88] [YAUS84]. The Abstraction Refinement Model (ARM) is based on an algebraic structure, and represents the process by transformations among descriptions [KELB90] [NANR89].

The integrated life-cycle model is intended to serve as a history of development and maintenance. It provides a documentation context of the current state of the system. One advantage of having a particular representation is that it can be used as a reference in maintenance activities. This model can, however, be described in terms of the ARM which has better defined properties.

The ARM has two basic objects: system descriptions and design decisions. A system description is the collection of documentation and code which completely defines the system. An example is the collection of documentation which makes up the System Requirements Specification to use the terminology of DoD-STD-2167A [DSSD85]. Systems descriptions are represented diagrammatically as circles. A design decision is an alternative for refining a system description. In diagrams design decision alternatives are represented by lines (as shown in Figure 5), where the lower system description is the result of the decision or a *realization* of the upper description. The upper description is an *abstraction* of the lower one.

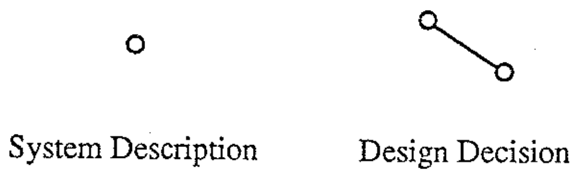


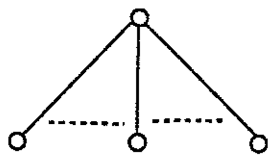
Figure 5. ARM Representation of System Descriptions and Design Decisions

The structure of the ARM is built from these objects. The basic properties can be stated without getting into excessive details. Two rules characterize the structure:

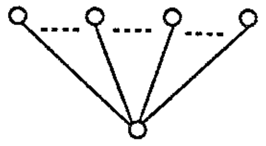
(1) From any one system description it is possible to have multiple design decisions (Figure 6a),

(2) Any one system description could result from alternative design decisions (Figure 6b).

The exact structure is called a lattice, which has the consequence that any pair of system descriptions has a *least common abstraction*.



6a. Multiple design decisions from any system description.



6b. System description as the result of many possible design alternatives.

Figure 6. Relationships among Design Decisions and System Description.

The *descriptive context* of a system is a sequence of system descriptions connected by design decisions (also called a path). For a path to be a context two requirements must be met:

(1) only one decision can be made at each system description, and

(2) at each system description only decisions consistent with the System Requirements Specification should be made.

(The execution of a decision is represented by an arrow following a design decision as shown in Figure 7.) The descriptive context of a system is a possible path from the System Requirements Specification to the current realization of the system. Therefore, the second requirement ensures the correctness of the system realization.

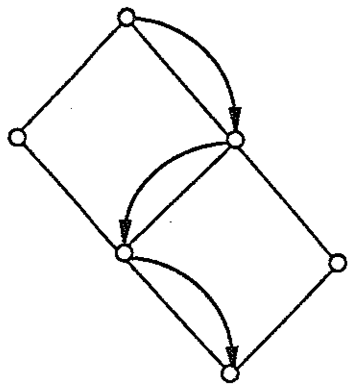


Figure 7. A Path through the ARM Structure

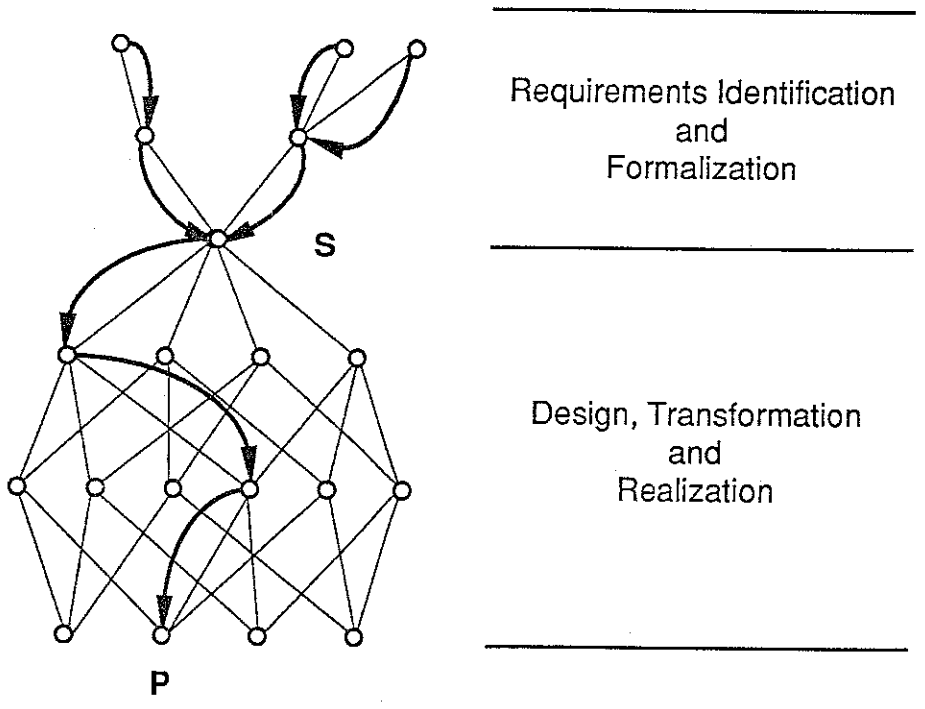


Figure 8. ARM Characterization of Development

The ARM characterizes development as the construction of the descriptive context of the system (see Figure 8). Development begins with a phase of requirements identification and formalization, for which the goal is to construct an initial system requirement specification (SRS). This is achieved in several ways, including taking the "intersection" of loosely defined specifications (shown in Figure 8 as two arrows merging into one system description). Once the initial specification has been found, the design, transformation, and realization phase begins. The goal of this phase is the realization of the system in executable form. A necessary byproduct of this process is the descriptive context.

For maintenance the system realization serves as the starting point and the context as the source of information. The existence of a complete context is dependent on the methodology and process used in development. An incomplete context could result from:

(1) deficiencies in the development methodology, or

(2) inability of personnel to execute the development process following the methodology (lack of training, inadequate tools).

Frequently, design decisions are not recorded, so repetition of the decision process might be made during maintenance.

The ARM characterization of maintenance (Figure 9) has two phases: an analysis phase and a transformational phase. The analysis phase produces a strategy for a modification using the following steps:

(1) problem identification,

(2) determination of an objective,

(3) the evaluation of alternative solutions to reach the objective, and

(4) the selection of the solution.

In the transformational phase the selected solution is implemented by first reverse engineering to a common abstraction of the existing system and the desired realization. The strategy from the analysis phase should assist in the identification of this abstraction.

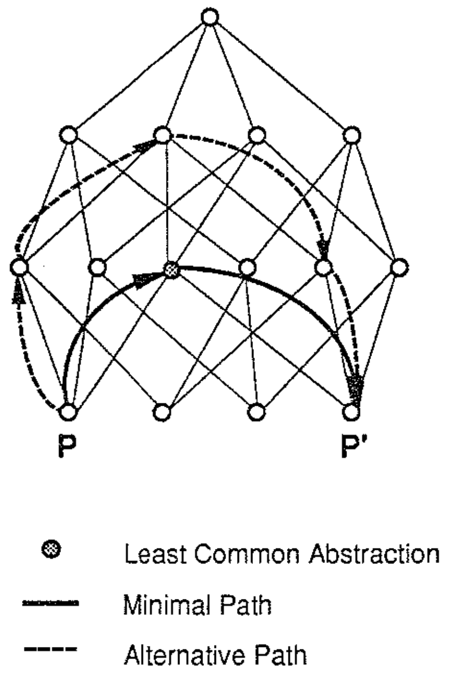


Figure 9. ARM Characterization of Maintenance

It is not always the case that the least common abstraction is in the context or even that the context is available. In the first case the least common context element (the least common abstraction in the context) can be used. From this point the realization can be found utilizing the solution from the analysis phase. If the context does not exist, the common abstraction and context must be constructed so that subsequent maintenance activities have a known reference.

Notice that reverse engineering contracts the original context to allow more alternatives. The context associated with the least common abstraction is that which permits the least number of alternatives to reach a set of realizations that include the target and the source. Working from the least common abstraction may be efficient (the minimum number of realizations), but locating the LCA could be quite difficult.

Once a common abstraction is achieved, forward engineering begins from that point. Forward engineering uses the strategy outlined in the analysis phase to find the desired realization. Reconstruction of the context is again a byproduct which benefits future maintenance.

The models considered in this section provide a description of the maintenance process. Particularly important is the ARM, which describes the dependence of maintenance on development and the maintenance process itself. This model is important in our discussion of the factors affecting a maintenance methodology.

3. Methodologies

Fundamental to the research presented in this report is a common understanding of what constitutes a"methodology". Although widely used, the term"methodology" 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.

3.1 What Constitutes a Methodology

To describe the term "methodology", a distinction must be made between it and the term "method." In [FREP77], 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., writing a statement of requirements. In general, a method specifies three elements:

- what decisions are to be made,
- how to make them, and
- in what order they are to be made.

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

- organizes and structures the tasks comprising the effort to achieve a global objective, establishing the relationships among tasks,
- defines methods for accomplishing individual tasks (within the framework of the global objective), and
- 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 in specific 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.

3.2. Software Development Methodology

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 10 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 be judged as achieved by observation at the project level (perhaps only at project completion). 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 may be achieved by taking steps that reduce reliability.

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.

3.3. The Objectives, Principles, Attributes (OPA) Framework

Influenced by Fritz Bauer's original definition of software engineering [BAUF72] and reflecting the above description of software engineering objectives, principles and attributes, a rationale for linking objectives, principles and attributes is founded on the philosophical argument that:

The *raison 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.

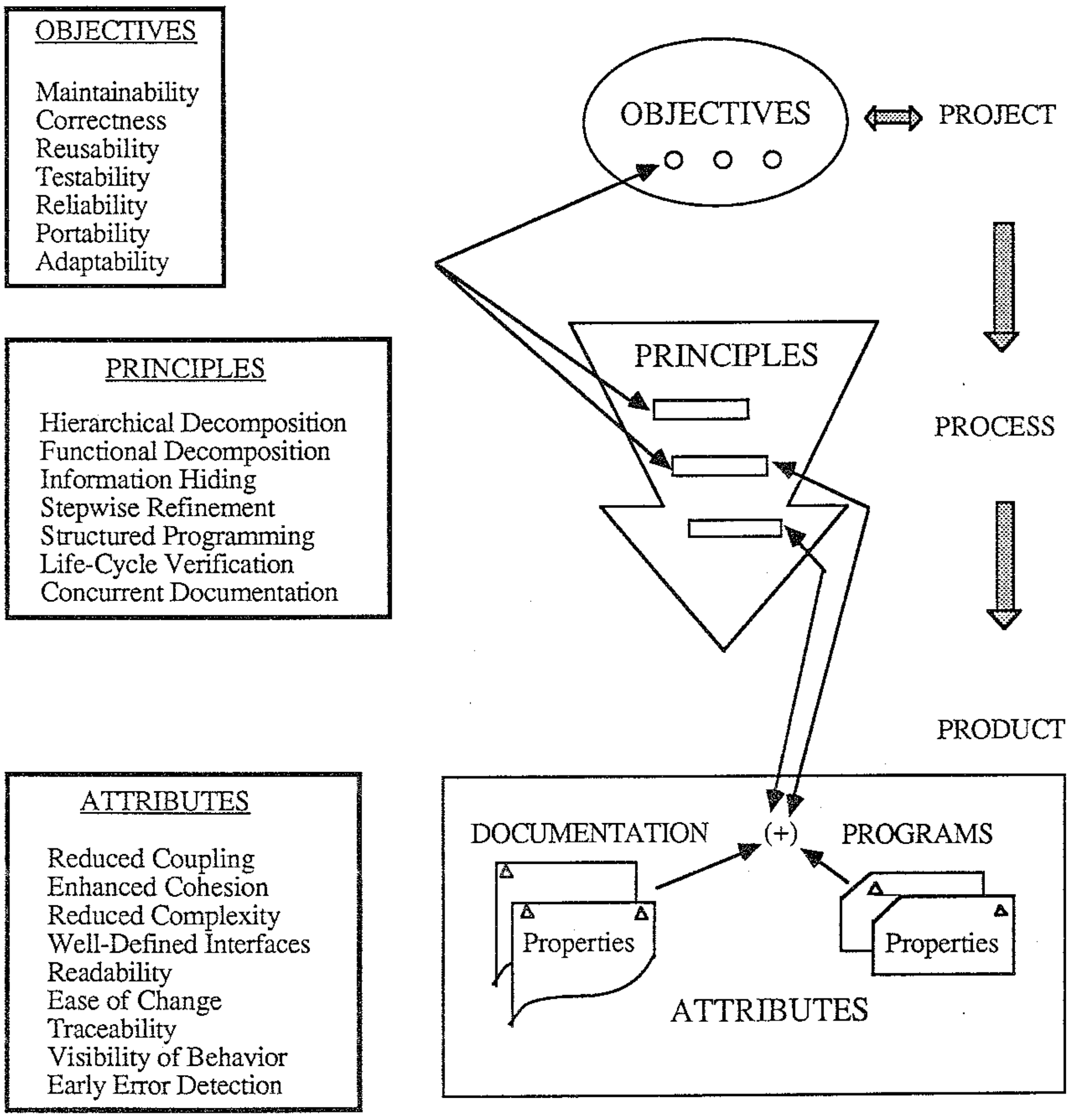


Figure 10. Linkages among Software Engineering Objectives, Principles and Attributes

This philosophy, exemplified by Figure 10, is tempered by practical concerns:

- While a set of software engineering objectives can be identified, this set might not be complete, and additions and modifications should be permitted.
- Objectives can be given different emphasis within a methodology or in applications of a methodology.
- Attributes of a large software product might be evident in one component yet missing in another.

A broad review of software engineering literature [BERG81, CLEP84, GAFJ81, JACM75, LISB72, PARD76, PARD72, SCOL78, WARJ76] leads to the identification of seven objectives commonly recognized in numerous methodologies:

- Maintainability -- the ease with which corrections can be made to respond to recognized inadequacies,
- Correctness -- strict adherence to specified requirements,
- Reusability -- the use of developed software in other applications,
- Testability -- the ability to evaluate conformance with requirements,
- Reliability -- the error-free performance of software over time,.
- Portability - the ease in transferring software from one host system to another, and
- 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.

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 creative process producing high quality programs and documentation.

- Abstraction -- defining each program segment at a given level of refinement.
    - Hierarchical Decomposition -- components defined in a top-down manner.
    - Functional Decomposition -- components partitioned along functional boundaries.
- Information Hiding -- insulating the internal details of component behavior.
- Stepwise Refinement -- utilizing a convergent design.
- Structured Programming -- using a restricted set of control constructs.

- Concurrent Documentation -- creation and management of supporting documents (system specifications, user manual, etc.) throughout the life cycle.
- 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 Figure 11.

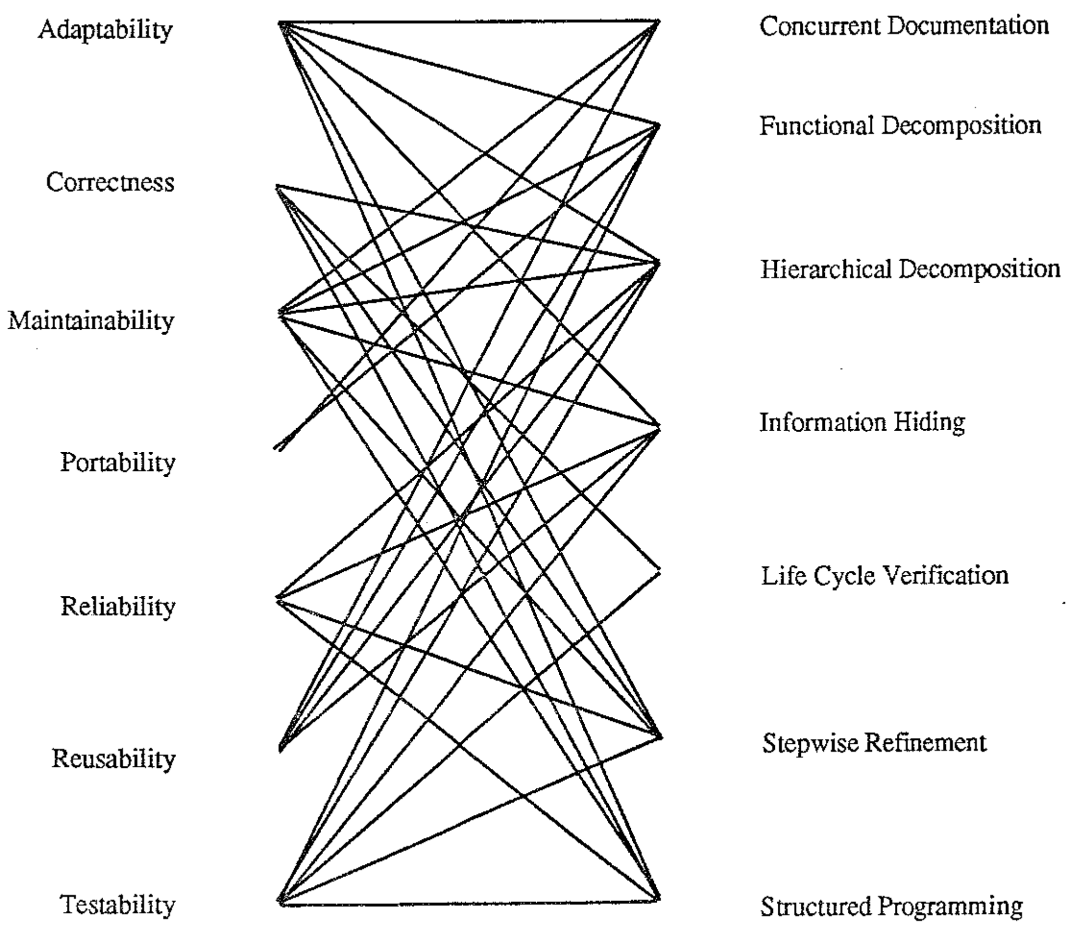


Figure 11. Linking Objectives to Principles

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.

- Cohesion -- the binding of statements within a software component.
- Coupling -- the interdependence among software components.
- Complexity -- an abstract measure of work associated with a software component relative to human understanding and/or machine execution.
- Well-defined Interfaces -- the definitional clarity and completeness of a shared boundary between a pair of components (hardware or software).
- Readability -- the difficulty in understanding a software component (related to complexity).
- Ease of Change -- the ease with which software accommodates enhancements or extensions.
- Traceability -- the ease in retracing the complete history of a software component from its current status to its design inception.
- Visibility of Behavior -- the provision of a review process for error checking. specification and design prior to implementation.

The relationships among attributes and principles are denoted in Figure 12.

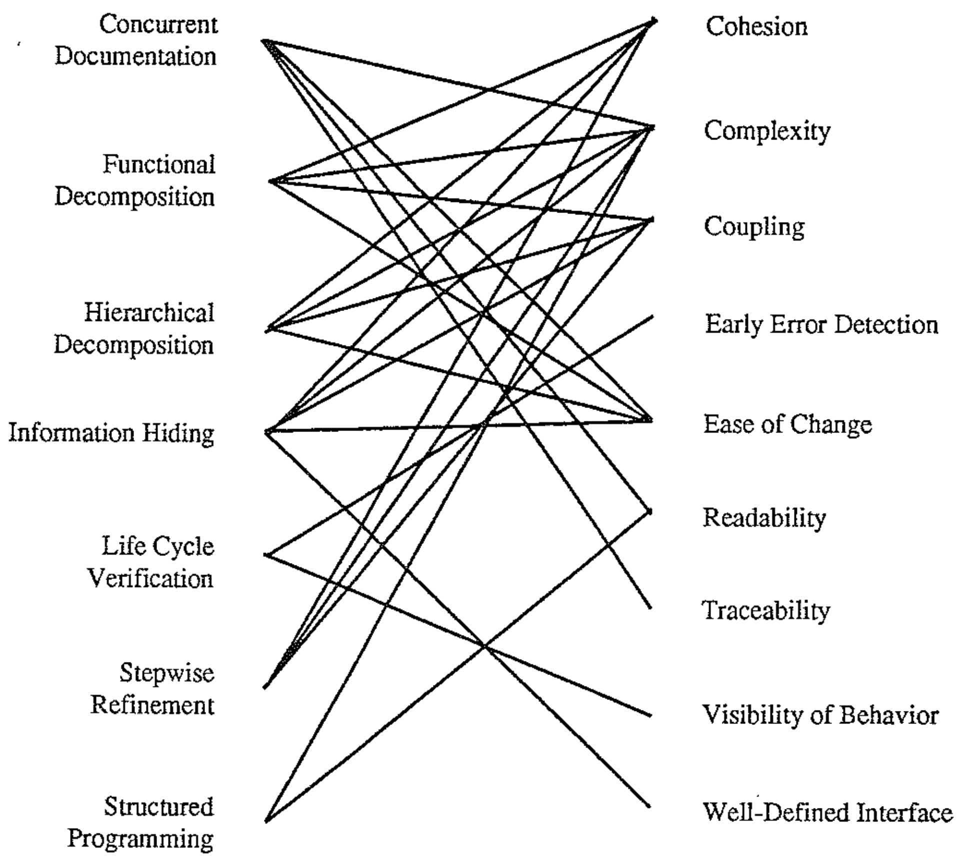


Figure 12. Linking Principles to Attributes

The software development process, illustrated in Figure 10, 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 particular code and documentation *attributes*. From a more detailed perspective, Figures 11 and 12 define the precise set of linkages relating objectives to principles and principles to attributes.

3.4. Application of the OPA Framework to Maintenance Methodologies

The OPA framework described above provides significant insights into what one might expect in and from a maintenance methodology. Comments on such expectations and related concerns are provided below:

(1) The objectives of a particular methodology are, to a large extent, reflective of requirements imposed by a set of system specifications. Within the maintenance framework, for example, a requirement to provide access to development documentation commensurate with the maintenance form underlies an objective of realizing desired changes in an efficient and effective manner. Identifying requirements and recognizing their role within a methodological framework is crucial because one must be able to assess the impact of constraining or sacrificing a particular requirement. The significance of this statement becomes even more evident when one considers that a single requirement often impacts several objectives.

(2) Recognizing the impact of requirements on methodological objectives is not, in and of itself, sufficient to define or guide a maintenance activity. Because maintenance tasks within the time-critical embedded systems domain can have wide-spread impact, a well-defined, systematic approach to performing maintenance activities is essential. Moreover, that approach must recognize, encourage and support the achievement of system-wide objectives through the enunciation of complementary methods and techniques.

(3) Intrinsic to a maintenance methodology, and providing the driving force behind many of the maintenance activities, is a complementary set of principles that work together in achieving the objectives emphasized by that methodology. For example, the principles of Scope Delineation and Varied Abstraction (discussed in Section 4) support the identification of source components requiring maintenance, while the principle of Change Propagation (also discussed in Section 4) is used to ensure maintainability in the resulting (or target) components.

(4) Finally, the OPA framework provides a basis for arguing the importance of a well-defined, systematic approach to performing software maintenance and of the crucial role of principles in supporting the maintenance process. One should recognize, however, the significance of principles in the shaping of the maintenance environment. The effective utilization of methodological principles often require the use of tools with specific capabilities. A study of such principles should yield requirements for tool capabilities, and subsequently, assist in the identification and selection of tools appropriate for the maintenance environment.

4. Factors Influencing a Maintenance Methodology

The previous sections of this report have centered on understanding the perceptions of maintenance derived from published works. Additionally, we have examined methodologies primarily within the context of software development. Using this material as a backdrop enables us to confront the challenge of defining a maintenance methodology. The immediate goal is to identify a set of requirements for such a methodology.

These requirements are driven by a number of influencers. The influencers are formed from within and beyond the maintenance phase: development activities, software environments, problems specific to maintenance, process models and the OPA framework. All affect the maintenance process and the execution of maintenance activities.

4.1. The Development Influencers

In the Abstraction Refinement Model characterization of development and maintenance, the descriptive context is the interface between the two. This context is a complete documentation of the system which serves as both a definition of the system and a record of its creation. We contend that the single most significant influence on maintenance (cost and effort) is the quality of the development process; the second is the quality of development documentation.

That development documentation is a primary influencer should not be surprising. Documentation consists of descriptions of the product and the process through which it is developed. Recognition of this fact can be seen in the work of Yau [YAUS84], where a major part of research on methodologies is focused on representing the context.

Development influencers are found in both the structure and content of the documentation. The structure of documentation can facilitate information search and comprehension. The content is important simply because in the absence of the needed information comes the requirement that it must be constructed or rediscovered.

The influence of development through documentation suggests that a maintenance methodology should take advantage of existing development documentation and also seek to overcome deficiencies in that documentation. Subsequently, a methodology must provide two mechanisms: (1) means to access documentation and (2) means for the reconstruction of unavailable or incomplete documentation. (Note that these requirements are not unique to development.)

4.2. The Tool and Environment Influencers

The influencers in tools and environments are interesting in that they can be viewed as "bottom-up" influencers. This "interest" stems from the fact that a top-down influence of methodology on tools is more typical than the reverse.

Two reasons exist for considering the ("bottom up") tool influencers. The first is that published research emphasizes tools and environments as opposed to methodologies. The second is that the existence and design of tools may reflect a number of underlying principles that admit extraction and generalization in the definition of a methodology.

The tools applicable to maintenance are many and varied. Rather than attempting an exhaustive listing, we resort to a description. This classification originates in the General Services Administration as a partial list of the types of tools available [ROMD86]. Figure 13 shows the adapted classification with definitions of the major categories.

(1) Information gathering - these tools are used in the capture of information needed for maintenance.

- Version comparison
- Data file comparison
- Code analyzers
  - Data flow tracers
  - Control flow tracers
  - Change analysis tools
- Browsers with
  - Structure outliners: procedural , control, data, I/O [SCHN87, p.306],
  - Presentation of delocalized information [SCHN87, p.305], and
  - Capability of selective abstraction [FLEN88,p.57] [CANG86].

(2) Reorganization and modification tools - tools which are used to reorganize, restructure, modify or translate software.

- Reformatters
- Restructurers
- Data standardization tools
- Syntax-directed editors
- Translators

(3) Documentation database construction tools - tools used in the building and modification of a database of captured information as part of the documentation of the system.

- Documentation aids
  - Re-documentation tools
- Cross-referencing tools

(4) Verification tools - these tools are used to verify the correctness of the documentation (including programs).

- Test data manipulation (capture) tools
- Test coverage monitors
- Quality assurance tools for documentation and programs.

Figure 13. Maintenance Tool Classification

The influence of tools on a maintenance methodology is primarily through "principles" (as in the OPA). Underlying these tools are principles which might be candidates for the methodology. The classification shown in Figure 13 is used to organize potential principles in terms of the identified tools.

4.2.1 Information Gathering Tools

Under the umbrella of "information gathering tools" are three types: comparators, code analyzers, and browsers. Each serves a separate role in terms of information gathering. The first two capture or create new information while the latter expands the presentation of information.

A principle underlying the use of *comparators* is the identification of changes from one version of a data file or document to the next. These tools provide an after-the-fact means of studying the change. Although the stated principle is sound, supporting tools are somewhat lacking.

Code analysis tools are used to capture information about the static characters and dynamic behavior of software. Included are simulators, which allow data- or control-flow tracing, and utilities that determine the effect of a proposed change. These tools are based on a principle of understanding both the software and the potential effects of any changes.

The third form of information gathering tools is, oddly enough, *presentation tools*. These tools are called browsers, and allow the presentation of documentation in a number of formats. Particularly important is the possibility of information selection (possible delocalized) and the level of abstraction for presentation. A principle underlying such browsing tools asserts the need for a view of the system that is both selective and from the right perspective for the task at hand.

4.2.2 Reorganization and Modification Tools

Like the information gathering tools, the reorganization and modification tools can be allocated into three classes. These include reorganization utilities, editors, and translators. The relationship of the first two to maintenance methodology principles are more apparent than for translators.

Reorganization tools are meant to increase the ease of understanding documentation and programs. Tools in this class include reformatters (or pretty-printers) and restructurers (that transform a program to an equivalent form with "nicer" structure). Also included are data standardization tools, for example an alias elimination tool. The underlying principle promoting this class of tools is to improve the "readability" of documentation.

Despite the typical view of editors as text editors, advanced forms are more useful. These editors might take advantage of the software structure or other properties to help in effecting less disruptive changes. A type of editor which fits this mold is the syntax-directed editor. Incorporation of features such as those outlined for browsers might be quite useful. The underlying principle here is that "each change should be implemented in a manner consistent with the existing software and level of abstraction."

Documentation database construction tools are aimed at the capture of information in "document" form, and also the construction of linkages among the "documents." Also important are re-documentation tools that allow the addition of information missing from development or created during maintenance. The incremental addition of information is a crucial function of such tools, since often the information collected is incomplete [WILC88]. "Collect all relevant information into a structured form, even if the information is incomplete," is a principle which requires such tools.

Verification tools are directed at the process. The focus is on the verification of the system with respect to the initial specification. The underlying principle supported by these tools is that quality assurance should be applied to increase certainty of correctness in the developed system.

Methodology principles establish the requirements for tools. In this section we have "inverted" the relationship, i.e., examined tools in the attempt to extract the underlying principles. This tact is necessary since far more is written about tools than about maintenance methodologies.

4.3. Specific Maintenance Considerations

In addition to the "information" influencers in development, and the "principle" influencers among tools, the "task" influencer of maintenance is also evident. A maintenance methodology must be concerned with the composition and execution of the maintenance tasks. The ARM characterization of maintenance decomposes the process into analysis and transformational phases and an identification and ordering of tasks within each phase should be provided by a methodology. The separation of tasks fits naturally into the framework of the ARM. This division serves as our primary guideline for discussion.

Each maintenance task has an origin. A user encounters a problem, or desires new capabilities, and submits a request that the software be modified. The challenge of maintenance is to accomplish the requested modification without the introduction of new problems. The analysis phase is concerned with identifying the source of the request and designing the modification to meet the request. The transformational phase then executes the modification to achieve the desired target. As recognized by Yau [YAUS84], a maintenance methodology must provide guidance for such tasks.

4.3.1. Analysis Phase

Within the analysis phase the goal is to identify the problem, the source of the problem and a strategy for solution. The strategy is then employed in the transformational phase to realize the solution. The methodology should address the problems associated with both phases.

The identification of the cause for software maintenance (the "problem") is the first task at hand. Generally, a software trouble report (or change request) provides a vague indication of the problem. Depending upon the form of maintenance required, identification of the "problem" may require a great deal of analysis or very little work at all. Once the problem has been located and identified, a solution strategy must be formed.

The solution strategy serves as a guide for the modification "engineering" to be made in the next phase. The nature of the problem affects the way in which a solution can be formed. A problem of corrective maintenance requires a different solution style from that of a problem of adaptive maintenance. In the first, changes must be made within the code; whereas the second dictates changes through the addition of new components external to the existing system. Obviously, the form of maintenance greatly influences the solution strategy, and the methodology must accommodate that fact. In addition, the impact of any change must be analyzed and accounted for in the solution strategy.

4.3.2. Transformational Phase

Supplied with an understanding of the problem and a solution strategy, the transformational phase embodies the solution realization. A methodology must define the methods by which the solution can be obtained. Such methods may include techniques for making changes within code or for developing new components. Some aspects of the methodology can be lifted directly from a development methodology; whereas others are modifications of these or are unique to maintenance.

A bigger challenge in making changes is ensuring that the changes do not introduce further problems in the software. Prevention of new problems is the reason for analyzing the effect of a change in the analysis phase; however, the changes made must follow the intent of the solution strategy. The use of quality assurance techniques is necessary both prior to and after the change, as well as for assistance in the design and execution of the code change activity.

4.4. Maintenance Requirements from the OPA Perspective

The OPA provides a framework in which the requirements can be expressed. The elements used are methodology objectives, the principles by which the objectives can be achieved, and the attributes of the resultant system (indicating success in achieving the stated objectives). The specification of objectives and principles provides a means by which the methodology can be loosely defined.

The objectives and attributes for a maintenance methodology correspond closely to those for development. One new objective can be stated as "realizing desired changes in an efficient and effective manner." This states the basic objective of maintenance, which the methodology must promote. As a consequence of this basic objective, a number of new principles emerge.

The previous sections advance the contention that maintenance introduces some unique concerns. While the objectives of development might prove sufficient for maintenance, the principles certainly do not. Four new principles are identified as requisite for realizing the basic objective of maintenance.

The first principle relates to the knowledge and understanding required before beginning a task.

Scope Delineation - The initiation of every task should be the identification of bounding (document) components.

This principle is aimed at the problem of finding the information needed for a task, ignoring irrelevant information, and limiting the scope of the work to be done. The "scope" of a task is the range of documentation needed ("vertical"), and the identification of software components affected by the specific task ("horizontal"). The browsing tools mentioned earlier suggest this principle:

(1) support presentation of delocalized plans [SCHN87, p. 305],

(2) partial access to multiple levels of documentation [FLEN88], and

(3) enable the hiding of unneeded information [FLEN88, p.57].

Wild and Maly [WILC88, p.81] add to this the argument that the "separation of relevant and irrelevant facts needs to be supported [by tools]."

The second principle is called the principle of varied abstraction:

Varied Abstraction - Representations that support multiple levels of abstraction and the transitions among them should be utilized in the maintenance process.

This principle states that only representations which can express different levels of documentation (specification) and the logical connections between them (design decisions) should be employed for maintenance documentation. Wild and Maly

[WILC88, p.81] support this principle by the statement: "There is a need to know design decisions and possible design alternatives." The concern here is with the knowledge needed for maintenance. Additionally, the importance of supporting transitions among levels, which is useful in development, becomes of paramount importance in maintenance.

A statement by Yau [YAUS88, p. 1129] suggests the following principle.

Common Representation - Environmental integration should be achieved through the use of a common representation (wide-spectrum language).

Subordinate to the principle of varied abstraction, the principle of common representation recognizes the impact of representation on the tools used to support a methodology. The discussion of browsing tools above also indicates the role of representation as well as abstraction with respect to tools. Tools for information hiding and presentation of delocalized plans could benefit from both common representation and abstraction.

The varied abstraction principle has two important aspects: use of multiple abstraction levels and the transitions among the levels. Movement among the levels of documentation used in development and maintenance must be facilitated. The principle also has important implications in terns of environments.

The third principle is also related to documentation:

Change Propagation - Recognition of the need to propagate specification changes through multiple levels of abstraction (i.e., throughout the maintenance document set).

This principle is based on the acknowledged need for consistency of documentation at various levels (i.e. code to design, etc. [ANTP87]). It recognizes that changes made at any level can have influence over documentation at higher or lower (or both) levels of abstraction. These changes must be propagated to the affected levels to ensure consistency. Adherence to this principle ensures the proper evolution of the maintenance set of documents (see [NANR89]).

Also affected by the Change Propagation principle is the problem of re-documentation. One interpretation requires the regeneration of missing documentation which is a recognized need throughout the literature:

"Preserve the knowledge gained in doing maintenance tasks even if this knowledge is incomplete" [WILC88, p.81].

Reverse Engineering enables extraction of organizational and operational rules from software [BACC88,p.49].

Re-documentation tools should allow for incremental documentation of the system [FLEN88, p.56].

Automatic documentation should be integrated with other re-documentation tools [FLEN88, p.57].

Both change propagation in existing documents, and re-documentation are attempting to ensure that the documentation is consistent throughout and accurately represents the current system. In a sense, Change Propagation is an extension of the Concurrent Documentation principle in software development.

The final principle emphasizes the measurement of product quality:

Quantification with Abstraction Resolution - Quantification of the product quality should be a constant goal: the potential for quantification is inversely related to the level of abstraction.

The importance of measuring product quality is widely recognized, and supported by the OPA framework. Also recognized is the need for quantified measurement of the "entropy" induced by maintenance [CANG86,p.320], and analysis of changes and the induced side effects [SCHN87,p.305]. The principle supports the application of quality assurance (metric based)tools discussed above.

4.5. The Abstraction Refinement Model Emphasis

Consideration of the ARM provides no added maintenance requirements beyond those already stated. However, the requirements can be stated more clearly or carefully using the descriptive qualities of the ARM. The model crystallizes our perspective of maintenance and maintenance methodologies.

The descriptive context of a system is an important aspect of the ARM. Context construction during development represents the documentation associated with the system creation. The maintenance methodology must modify the context to implement software modification, but these modifications should be made in a disciplined manner to preserve the qualities of the context.

The ARM characterizes modifications as forward and reverse engineering activities. A maintenance methodology should define such methods, guiding and controlling the use of methods and techniques. Methods for forward engineering naturally resemble those used in development (some adaptation may be required, however). Forward engineering in maintenance involves making direct changes within the existing software system where the freedom of development does not exist. Reverse engineering ranges from documentation searches to re-documentation, so methods are both easy and hard to define. Recognition of the limiting and confining characteristics of maintenance is important to defining a methodology.

Another aspect of the ARM that affects the maintenance methodology requirements is the notion of common abstraction. Common abstractions play two roles in the ARM. One is as a starting point for forward engineering to implement the modification. The second is as a marker point from which the context must be rebuilt. Considering the concerns expressed earlier, both roles are important to the methodology. The methodology should include methods which assist in the identification of a common abstraction.

The three points mentioned above are perhaps the most important influencers of the ARM on a maintenance methodology. The context represents the system documentation, a major influencer throughout the system life. Forward and reverse engineering are the basic activities applied in maintenance, and methods for them are certainly important. Common abstractions are intrinsic to the maintenance process and the means for identifying them are quite important.

5. Requirements for a Software Maintenance Methodology

The requirements for a software maintenance methodology are stated below. These are derived from the principles enunciated in Section 4.4. Each requirement is stated with a description and reference to the governing principles and relevant literature (if not touched upon in the description of the principles).

(1) *Access to development documentation commensurate with maintenance forms.*
Principles: Varied Abstraction, Change Propagation.
Purpose: A mechanism for accessing development documentation is the first need, but, secondly, this access should be guided by the form of maintenance. The need for accessing specific levels of documentation for particular maintenance forms is described by the Abstraction Refinement Model [NANR89].

(2) *Provide for decisions which maximize product availability (consider system availability).*
Principles: Varied Abstraction
Purpose: This requirement relates to the global objective of the maintenance process being both effective and efficient. The methodology should seek to ensure that quality is not sacrificed but changes are accomplished efficiently.

(3) *Each modification activity should include the following sub-activities:*
(a) Identify source and target (order is source dependent).
(b) Define and effect transformation process.
(c) Record source, process, and target.

(d) Test:

(1) Identify original test specifications

(2) Modify original test specifications for target.

(3) Revise test procedures and apply them.

Principles: Varied Abstraction, Scope Delineation.

Purpose: The major steps in making a modification are prescribed. Actually making the modification requires the identification of the source of the maintenance, the target of the maintenance (a solution) and the means for achieving that target. Complementing these activities is the recording of the source, target and the process (including design decisions) to ensure compatibility of the documentation with the programs. Finally, testing is required to ensure that the target was achieved and that it is correct.

(4) *Promote the identification of alternatives, the evaluation of alternatives (risk assessment), and support documentation of both.*

Principles: Varied Abstraction.

Purpose: The maintenance (and development) activities involve the consideration of a number of alternatives. The recording of these alternatives and their evaluation provide a justification of the selected change strategy that may be useful for later maintenance.

(5) *Recognize and resolve potential interference among concurrent maintenance activities.*

Principles: Scope Delineation.

Purpose: Maintenance activities might require changes which interfere or interact in some way. Recognizing and handling this interference helps ensure that combinations of modifications will have a positive result. Otherwise, these modifications might combine to form an undesired change.

(6) *Require and facilitate auditing of the maintenance process (metrics, and methodology).*

Principles: Quantification with Abstraction Refinement.

Purpose: Auditing of the maintenance process evaluates the success of the process in terms of the product quality. The methodology should support auditing to guarantee the correctness of the process and the quality of the product.

(7) *Support (enforce) uniformity in maintenance process/activity (procedures, documentation).*

Principles: Change Propagation.

Purpose: Providing for uniformity in the maintenance process means that the procedures used in the maintenance process are the same. The impact of uniformity in the process is uniformity in documentation. Uniformity in documentation increases the maintainability of the system by helping understanding.

(8) *Enable prioritization and coordination of maintenance forms and activities. (Interface with Configuration Management.)*

Principles: Change Propagation.

Purpose: Various maintenance forms attach not only a practical order, but also a theoretical order to tasks. The practical order captures the necessity for certain modifications being made first. The theoretical order implies that certain tasks should be performed first for efficiency and effectiveness. The methodology needs to recognize the ordering in the decision-making process.

(9) *Enforce recording of source, process, target, test documentation, decision alternatives, evaluation, and final decision.*

Principles: Scope Delineation, Change Propagation.

Purpose: To provide for the documentation necessary for future maintenance.

(10) *Enable, promote and enforce the quantification of the process and product quality.*

Principles: Quantification with Abstraction Resolution.

Purpose: Metrics can play a major role in the auditing process if provided proper support.

6. Summary and Conclusions

Software maintenance has suffered from inattention, the improper analogies with hardware, and negative connotations associated with the term "maintenance." The life-cycle models acknowledge the maintenance phase, but do little else to describe the attendant activities or relate them to development activities. In the attempt to derive requirements for an "ideal" methodology, we have reviewed the treatments of maintenance in the software engineering research literature, considered the several definitions, and noted the categorization of maintenance forms. Of primary interest are the models of the maintenance process that depict the development/maintenance relationship.

The objectives, principles, attributes (OPA) characterization of software development methodologies is seen as applicable to maintenance. The fundamental maintenance objective - effective and efficient change - leads to the statement of four maintenance-related principles: scope delineation, varied abstraction (with common representation), change propagation, and quantification with abstraction resolutions.

The culmination of the research in Subtask 1 is the statement of requirements in Section 5. Each requirement is described in terms of its purpose and the underlying principle(s). Using this frame of reference, the AEGIS maintenance process can be examined and refined to conform with application domain needs and management constraints.

7. References

[ANTP87] Antonini, P., Benedusi, P., Cantone, G., and Cinitile, A. "Maintenance and Reverse Engineering: Low-Level Design Documents Production and Improvement," in *Proceedings of the Conference on Software Maintenance - 1987*, IEEE Computer Society Press, 1987, 91-100.

[ARAG85] Arango, G., Baxter, I., and Freeman, P. "Maintenance and Porting of Software by Design Recovery," in *Proceedings of the Conference on Software Maintenance - 1985*, IEEE Computer Society Press, 1985, 42-49.

[BACC88] Bachman, C., "A CASE for Reverse Engineering," *Datamation* **34**, 13 (July 1, 1988) 49-56.

[BAUF72] Bauer, F.L., "Software Engineering," *Information Processing*, North Holland Publishing Company, 1972.

[BERG81] Bergland, G.D., "A Guided Tour of Program Design Methodologies," *Computer*, Vol. 14, No. 10, October 1981, pp. 13-36.

[BOEB76] Boehm, B.W., "Software Engineering," *IEEE Transactions on Computers* **C-25**, 12 (December 1976) 1226-1241.

[BOEB86] Boehm, B.W., "A Spiral Model of Software Development and Enhancement," *ACM Software Engineering Notes* **11**, 4 (August 1986) 14-24.

[CANG86] Cantone, G., A. Cimitile, P. Maresca, "A New Methodological Proposal for Maintenance," Microprocessing and Microprogramming 18, (1986) 319-332.

[CLEP84] Clements, P. C.,"Function Specifications for the A-7E Function Driver Module," NRL Memorandum Report 4658, Naval Research Laboratory, Washington, D. C., November 1984.

[DSSD85] DoD-STD-2167A, *Defense System Software Development*, United States Department of Defense, June 4, 1985.

[FLEN88] Fletton, N.T. and M. Munro, "Redocumenting Software Systems Using Hypertext Technology," in Proceedings of the Conference on Software Maintenance - 1988, IEEE Computer Society Press, 1988, 54-59.

[FREP77] Freeman, P.,"The Nature of Design," *A Tutorial on Software Design Techniques*, Second edition, IEEE Computer Society Press, 1977, pp. 29-36.

[GAFJ81] Gaffeney, J. E.,"Metrics in Software Quality Assurance," *Proceedings of the National ACM Conference*, November 1981, pp. 126-130.

[GAMS88] Gamalel-Din, S.A. and L.J. Osterweil, "New Perspectives on Software Maintenance Processes," in Proceedings of the Conference on Software Maintenance - 1988, IEEE Computer Society Press, 1988, 14-22.

[GIDR84] Giddings, R.V., "Accommodating Uncertainty in Software Design," *CACM* **27**, 5 (May 1984) 428-343.

[JACM75] Jackson, M., *Principles of Program Design*, London: Academic Press, 1975.

[KELB90] Keller, B.J., *An Algebraic Model of Software Evolution*, Master's Thesis, Department of Computer Science, Virginia Tech, (to be completed) Spring, 1990.

[LINI88] Lin, I.-H. and D. A. Gustafson, "Classifying Software Maintenance," in Proceedings of the Conference on Software Maintenance - 1988," IEEE Computer Society Press, 1988, 241-248.

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

[NANR89] Nance, R. E., B. J. Keller, and D. Boldery, "Documentation Production Under Next Generation Technologies," Technical Report SRC-89-001, Systems Research Center, Virginia Tech, 15 February 1989.

[PARD72] 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.

[PARD76] 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.

[PETL87] Peters, L., *Advanced Structured Analysis and Design*, Prentice- Hall, 1987.

[RIDM88] Ridgway, M., "Curriculum Shortfall," Datamation, 15, December 1988, 77.

[ROMD86] Roman, D., "Classifying Maintenance Tools," Computer Decisions, 18 (June 30, 1986) 35,40-41,68-71.

[SCHN87] Schneidewind, N. F., "The State of Software Maintenance," *IEEE Transactions on Software Engineering* **SE-13**, 3, (March 1987) 303-310.

[SCOL78] Scott, L.,"An Engineering Methodology for Presenting Software Functional Architecture," *Proceedings of the Third International Conference on Software Engineering*, NY, 1978, pp.222-229.

[SWAE76] Swanson, E.B., "The Dimensions of Maintenance," *Proceedings of the Second International Conference on Software Engineering*, 1976, 492-497.

[WARJ76] Warnier, J., *Logical Construction of Programs*, 3rd edition, trans. B. Flanagan, NY: Van Nostrand Reinhold, 1976.

[WILC88] Wild, C. and K. Maly, "Towards a Software Maintenance Support Environment," in Proceedings of the Conference on Software Maintenance - 1988, IEEE Computer Society Press, 1988, 80-85.

[YAUS84] Yau, S.S., Methodology for Software Maintenance, Final Technical Report RADC-TR83-262, NTIS AD-A143-763/1, February 1984.

[YAUS88] Yau, S., R. Nicholl, J. Tsai, and S. Lin, "An Integrated Life-Cycle Model for Software Maintenance," *IEEE Transactions on Software Engineering* **14**, 8, (Aug 1988) 1128-1144.

[ZELM79] Zelkowitz, M.V., A.C. Shaw, and J.D. Gannon, *Principles of Software Engineering and Design*, Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION: Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): Systems Research Center SRC-90-001	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Systems Research Center; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: Naval Surface Warfare Center
6c. ADDRESS (City, State, and ZIP Code): 320 Femoyer Hall, Virginia Tech, Blacksburg, Virginia 24061-0251	7b. ADDRESS (City, State, and ZIP Code): Dahlgren, Virginia 22448-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Naval Surface Warfare Center; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code): Dahlgren, Virginia 22448-5000	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Requirements for a Software Maintenance Methodology, Interim Report: Subtask 1

12. PERSONAL AUTHOR(S)
Richard E. Nance, James D. Arthur, and Benjamin J. Keller

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Interim	FROM 16 Aug 89 TO 15 Dec 89	1990 January 18	36

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

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

Software maintenance, although widely recognized as the most costly period in the life of a system, is given only passing consideration in life-cycle models. An extensive literature review shows the relationship between the development and maintenance phases to be ignored to a large extent. The Abstraction Refinement Model (ARM) describes the dependency of software maintenance on the quality of development documentation and depicts the adaptive and perfective maintenance forms as relying on earlier design and requirements documents to a greater degree than corrective and preventive maintenance. The ARM is effective in laying the foundations for a software maintenance methodology, particularly in explaining the role of reverse engineering. Coupling the ARM with the Objectives/Principles/Attributes procedure for the evaluation of software development methodologies proves effective in drawing the contrast with maintenance requirements, which are specifically identified for further study and assessment.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code); 22c. OFFICE SYMBOL

DD Form 1473, JUN 86 Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified