A PROCEDURAL APPROACH TO THE EVALUATION OF SOFTWARE DEVELOPMENT METHODOLOGIES

by

Ashok V. Dandekar

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

APPROVED:

Dr. J.D. Arthur, Chairman

Dr. R.E. Nance

Dr. S.M. Henry

September 1987

Blacksburg, Virginia

A PROCEDURAL APPROACH TO THE EVALUATION OF SOFTWARE DEVELOPMENT METHODOLOGIES

by

Ashok V. Dandekar

Dr. J.D. Arthur, Chairman

Computer Science and Applications

(ABSTRACT)

This thesis presents a procedural approach to evaluating software development methodologies. The evaluation procedure adopts a unique approach based on the fundamental assumption that the requirements govern the process by which software is constructed. To begin with, this research partitions desirable software characteristics into three categories, viz., objectives, principles and attributes. The thesis claims (claims are substantiated with literature references) that there exist definitive relationships (or linkages) among the software objectives, principles and attributes. These linkages form the foundation of the evaluation procedure. The procedure constitutes two processes, top-down process and bottom-up process. These processes are used to assess the software product and the employed software development methodology. The top-down process begins by identifying the objectives and travels down through principles and product attributes; thus, evaluating the adequacy of the methodology. The bottom-up process, on the other hand, starts at the attribute level and goes up through principles and objectives. The bottom-up process highlights the effectiveness of the methodology. Attributes are identified in the product via properties. This research establishes several properties (called factors in the report) for each attribute. A measurement approach is also presented to help assess the extent to which attributes are present. The feasibility and validity of the evaluation procedure are illustrated through the analysis of two real life methodologies.

Key Words: Software Development Methodology, Evaluation Procedure, Objectives, Principles, Attributes, Properties, Linkages, Factors, Metrics.

ACKNOWLEDGEMENTS

I am deeply grateful to Dr. James D. Arthur for providing inspiration and support in performing research activity. Also, his enormous patience, constant encouragement, guidance and advice throughout the writing process has made this voluminous report a reality. Working with him has been a very fruitful and enjoyable experience.

I am also grateful to Dr. Nance for providing support, encouragement and valuable advice. Thanks are also due to Dr. Henry for her willingness to be on my committee.

Table of Contents

List of Illustrations

List of Tables

1.0 INTRODUCTION

The editors of the first NATO report [NAUP69] stressed, as early as 1969, the need for software development based on the types of theoretical foundations and practical disciplines traditional in established branches of engineering. During the past decade software engineers have come to recognize circumstances that are collectively known as the "software crisis". Symptomatic of the crisis are the dramatic escalation of software costs [BOEB76] and schedules, completion dates being set but rarely kept [BROF74], waste and duplication, and disgruntled users. Subsequently, as software systems grew larger, quality became suspect.

As a response to the software crisis, a set of software development techniques, methods and methodologies has evolved. These techniques deal with software as an engineering product that requires planning, analysis, design, implementation, testing and maintenance.

According to Boehm [BOEB76], software engineering involves "the practical application of scientific knowledge to the design and construction of computer programs and the associated documentation required to develop, operate and maintain them." Subse-

quently, software engineering has emerged as a scientific discipline and is now considered an integral part of computer science. Although it is difficult to provide a universally acceptable definition of software engineering, there is a general consensus that software engineering implies the disciplined and skillful use of suitable software development methodologies and tools, as well as a sound understanding of certain basic principles. These methodologies and tools are employed for the development of software and its maintenance throughout its life-cycle [ROSD75].

A study of software engineering practices has resulted in the development of a number of software design and development methodologies based on accepted software engineering principles, (e.g. hierarchical decomposition, information hiding, stepwise refinement, etc.) which address various aspects of the software life-cycle. The software industry has benefited from the proliferation of various design and development methodologies. For example, Jackson [JACM75] and Warnier [WARJ74] methodologies describe approaches for designing software while Program Design Languages (PDLs) provide design specification capabilities based on structures that resemble the software target language [HENS85]. With the numerous methodologies now on the market, the user's dilemma is choosing one that will "deliver the goods". This is a difficult problem because each methodology adopts a different approach, recommends different tools and is applicable in a different environment.

This research effort focusses on the dilemma described above. The effort is aimed at helping users 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 methodology within a given environment. These concerns have provided motivation for this research and have

stimulated the evolution of *a procedural approach to evaluating software development methodologies.*

To develop a foundation for discussion, this thesis first addresses the basic question - What is a methodology? It also makes a clear distinction between a method and a methodology. Several examples are cited for each. The second question addressed is - Which methodology is better and in which environment? A study of these questions has led to a systematic approach for evaluating software development methodologies [ARTJ86].

The approach presented in this research suggests that it is possible to measure the adequacy of a methodology on a comparative scale (say, from 1 to 10). A procedure is presented which, when applied to a methodology, will yield a value assessing the adequacy of that methodology. Serving as a basis for the evaluation procedure are certain basic software engineering objectives that can be attained through the application of certain principles which, in turn, induce particular attributes in the resulting product. For example, to achieve the objective of maintainability one employs the principle of hierarchical (or functional) decomposition which induces a "coupling" attribute in the software product. Linkages have been established between these objectives, principles and attributes.

Although the above mentioned procedure establishes and utilizes linkages among methodology characteristics, relating those characteristics to a product (code and documentation) is achieved via product properties. Product properties, which indicate the presence or absence of an attribute, are identified, linked to methodological attributes, and provide the basis for assessing to what extent a methodology is followed. In some sense this assessment reflects the effectiveness of the methodology.

In chapter 2, this thesis presents a detailed discussion of methodologies, tools and environments. It also provides an in-depth study of three major software engineering methodologies. Chapter 3 discusses the evaluation procedure from a linkage viewpoint by presenting the established relationships among software objectives, principles and attributes. Chapter 4 then presents properties and discusses the measurement of properties using metrics. Also, a detailed discussion of factors in code and documentation is given. Chapter 5 briefly discusses an application of the evaluation procedure to two methodologies and differentiates between software engineering constraints and software engineering objectives. Chapter 6 discusses an important relationship between software engineering and systems engineering. Chapter 7 presents the author's contribution to the research as well as future work and new applications for the evaluation procedure.

2.0 BACKGROUND

This chapter is intended to familiarize the reader with the terms method, methodology, tools, environment, and software life-cycle. The motivation is to provide a sound background that will enable the reader to understand better the concepts fundamental to software development methodologies and to facilitate the comprehension of material presented in subsequent chapters. The examples presented in this chapter are intended to illustrate the fundamental principles of software development methodologies and to provide a level of understanding that can assist the user in navigating through the world of software engineering.

2.1 What is a methodology

In software engineering the words "method" and "methodology" are often used synonymously. In the literature, distinction is rarely made between the two terms; each has been defined in a number of ways by different authors.

According to Cameron [CAMJ83] a method is a process or a procedure for attaining an objective, or it is a way of doing things. At each step of the method, a number of decisions are made about the system or the software which is being developed. A method implies and is characterized by the ordering and organization of such decisions. From a software engineering perspective, developing software is a method [ARTJ86]. It has an element of decision making at every step. The word "decision" is interpreted in a wide sense. It can be an explicit statement of a fact about the subject matter or it might be a policy decision followed throughout the development process. A few possible examples of decisions are given below:

- defining a certain abstract data type,
- specifying a database system,
- deciding data storage formats,
- defining an output format, and
- decomposing system functions into sub-components.

Such decisions have to be made throughout the development task. A method organizes the development by ordering and organizing decisions. At the lowest level of decision making, writing a precise algorithm is a method. To summarize, a method provides three elements:

- a set of decision(s) to be made,
- how they are made, and

- the sequence in which they are made.

A prime example of a software design method is top-down design. The general approach in top-down design is to make decisions that consider the overall design goals and constraints, group them into levels, make decisions at the highest level, and then proceed iteratively to the lower levels.

A methodology, on the other hand, is defined as the study or science of methods [CAMJ83], or a collection of methods [HENS85] for achieving a specific task(s). A methodology is also defined as a process of identifying various steps [RAMC78] in the development of a system. In the case of software development, these steps are usually well defined. Each step is supported by analytical techniques, software tools, design aids, and programming guidelines.

In general, a methodology suggests several complementary methods and a set of rules or constraints for applying them. In more specific terms, a methodology

- consists of methods for accomplishing specific task(s) within the overall framework of an 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.

One particular characteristic of a methodology addresses decision dependency: if decision "Y" depends on the outcome of decision "X", then "X" should precede "Y". This characteristic, for example, embodies the separation of specification decisions from implementation decisions. How a specification is fitted into a hardware/software environment obviously depends on the constituent parts of a specification.

Methodologies are generally classified according to the concepts involved and the way they organize the developmental effort. Software engineering methodologies commonly address particular stage(s) of the software life-cycle [RAMC78], e.g., requirements specification, design, implementation, testing, debugging, maintenance and so forth. Most methodologies are also supported by analytical techniques, software tools, design aids, and programming guidelines.

After understanding the distinction between a method and a methodology, one may ask - why should one study methodologies? The next section discusses this question.

2.2 *Why Study Methodology?*

This section presents a discussion of methodologies from the perspective of what constitutes a good methodology and how important it is to study a methodology from analytical and critical view points. In general, a software design methodology can be characterized by the following criteria:

- its general application,
- its ease of use, and
- its consistency.

The first criteria, generality of the methodology, is determined by the size of the domain of application. For example, one may ask - whether a methodology is applicable to multiple environments or to a particular environment. The second criteria, ease of use,

is the lack of difficulty in properly utilizing a methodology. In reference to the third criteria, a methodology is consistent if it gives appropriate guidance in all decisions to be made. Conversely, if a methodology gives poor advice for some decisions, it is inconsistent [RAJV85]. More intuitively, the role of a methodology is to guide the user/designer in the software development process by indicating what decision should be made at specific points in time and describing the pertinent information needed to arrive at that decision.

Even with the availability of adequate methodologies, the process of selecting the most appropriate methodology for a particular application can be error prone. As described by Hamilton [HAMM79] typical risks include:

- comparing methodological techniques addressing very different problems,

- comparing techniques intending to address a problem, but not effectively addressing them, and

- comparing techniques with respect to ill-defined requirements.

The user always faces a dilemma regarding the selection of a right methodology. Once the methodology is determined, however, then so are methods and (hopefully) tools for maximizing the chances of satisfying the methodological goals.

Selecting an adequate methodology is advantageous from several perspectives. A good methodology helps reduce the number of errors committed at each development stage. It also reduces "ripple effect" errors, that is, an error made in one stage inducing multiple errors in the subsequent steps. A methodology also requires that a critical analysis be performed at every step in the software development process. This substantially reduces

the effort needed for final testing and validation. Thus, in addition to correct results, a good methodology offers a considerable saving in man-hour cost.

Because there are a multitude of methodologies, however, evaluating each methodology and then individually comparing them presents a major problem. Nevertheless, it is necessary that one thoroughly understands how each methodology works, how it is applied, what restrictions are applicable, and which support environments are appropriate. Moreover, one must understand a methodology from both an analytical and critical point of view. This understanding includes an appropriate knowledge levels about accompanying tools.

The above discussion presents a justification for studying methodologies. Based on an understanding of the need to study methodologies, the following section presents a discussion of few software development methodologies.

2.3 Methodologies, Tools and Environments

A typical software development methodology usually addresses one or more phases of software life cycle. The life cycle phases addressed by different methodologies vary. Before proceeding to specific software development methodologies, it is first appropriate to familiarize the reader with the life cycle phases and their implications.

2.3.1 Software Life Cycle

As typified by [BOEB84, CAVW78, PETL81, RAMC78, TEID74] the software life cycle consists of six separate stages through which software development passes. The phases are:

- Requirements analysis,
- Specification,
- Design,
- Implementation,
- Testing, and
- Maintenance.

Although these phases are precisely defined, their "spheres of influence" often overlap. For example, aspects of requirements and specifications may be interleaved, testing and maintenance may be carried out in the same phase. It is also common to find subdivision of these phases in smaller or more detailed phases; e.g. the design phase is often split into preliminary design, final design and so forth.

Software development starts with the requirements analysis phase. In the requirements analysis stage, the user discovers a need for some software system; the nature of the need is analyzed, the requirements for a system that would satisfy these needs are established. Other aspects such as processing time, costs, and error probability are considered among

the basic requirements before an appropriate course of action is chosen. Requirements analysis helps the system designer to better understand the problem and the trade-offs among conflicting constraints.

The second phase of the life-cycle is called the specification phase. In this phase functional descriptions of the software are developed and the constraints on structure and resources are indicated. This stage is also known as the definition stage. The inputs and outputs are defined during this phase as well as details about data storage (disk or tape), output formats and so forth. A functional specification document is created and later used throughout the development project. The more precise the specifications, the less likely are errors, confusion, or later recriminations.

The third stage of the life cycle is that of design. Given a functional specification the design process provides a formulation of system components and how they should interact. Activities of the design stage include :

- identifying functions indicated during requirements analysis,

- assigning algorithms to these functions,

- creating modules by grouping these functions according to well-defined criteria, e.g., better interaction, parallel processing etc., and lastly

- formulating algorithms, data structures, and interconnections among modules and data structures.

The design process is strongly influenced by the programming language used to implement the system, but is not concerned with syntactic aspects of the implementation

language or the level of detail inherent in expression evaluation and assignment statements. In short, during the design phase the system representation is developed in sufficient detail, but stops short of making all decisions associated with programming.

The fourth stage in the life cycle process is implementation. Implementation is the transformation of system design into a software entity that can be compiled on the target computer. The implementation process involves:

- programming - which is writing the code from algorithms,
- testing - where individual pieces of code are tested,
- integration - where these pieces are put together to form sub-assemblies,
- debugging - where software errors in the code are detected and removed, and
- re-design - where design is modified if necessary and changes are made to the original system to make it conform to specifications.

The next stage in the life cycle process involves testing the system that has been implemented. Testing is done to ensure that everything is as per specifications and the results are as per the requirements. During testing, the system is presented with data that is expected during "like" conditions. Testing is often divided into three distinct operations: module testing, integration testing and systems testing. In module testing each module is subjected to the test data supplied by the programmer. A test driver simulates the software environment of the module by containing dummy routines to take the place of actual routines that the tested module calls. During integration testing groups of components are tested together. The systems test focusses on assessing the behavior of the

total software system. Closely related to testing are verification and validation (V & V) activities. A system is validated by showing that it performs according to its specifications. A system is verified if it can be proven that the program meets its specifications. A verified system is correct relative only to the initial specifications and assumptions about the operating environment. The testing stage is quite time consuming and may even comprise half of the total effort [PRER82].

The last stage of software development life cycle is maintenance. The maintenance phase includes all aspects of software modification after the system is installed. It involves the location and repair of bugs, modification of existing functions, and so forth. This phase also includes the enhancement of capabilities and adaption of software to new processing environments. The maintenance phase consists primarily of following four activities:

- Corrective maintenance - which includes diagnosis and correction of one or more errors,

- Adaptive maintenance - which modifies software to properly interface with a changing environment,

- Perfective maintenance - which occurs after a software is successfully executing and usually includes recommendations for new capabilities, modifications of existing functions, and the incorporation of general enhancements, and lastly,

- Preventive maintenance - when software is changed to provide a better basis for future enhancements [PRER82].

The above six stages of software life cycle are captured in the "waterfall" model of software life cycle (Figure 1 on page 16) developed by Boehm [BOEB84]. As illustrated in Figure 1 on page 16 additional activities implied by the software development process are:

- verification and validation to eliminate as many problems as possible at each stage, and

- iterations among phases.

Every software development methodology addresses at least one of the life cycle phases. Though the ultimate goal of all methodologies is the same, the path chosen to achieve a goal is often different, and the means (tools and environments) for reaching the objective is also different. The following discussion introduces three software development methodologies, their tools and environments.

2.3.2 Examples of methodologies, tools and environments

As important as methodologies are [BERG81, PETL77], rarely is a clear distinction made among software tools, environments and methodologies. Moreover, these terms are often used interchangeably. Complicating the issue is the fact that tools exist which function as environments and support a software development methodology, e.g., Structured Analysis and Design Techniques (SADT) [ROSD79] and the Software Development System (SDS) [DAVC77]. In general, an environment can be viewed as the parts of a system that the user perceives. This perception extends beyond the user interface to facilities (tools) that provide ancillary user support. It includes the psychological "feel" of the system as well as the details of its functionality. In essence, an envi-

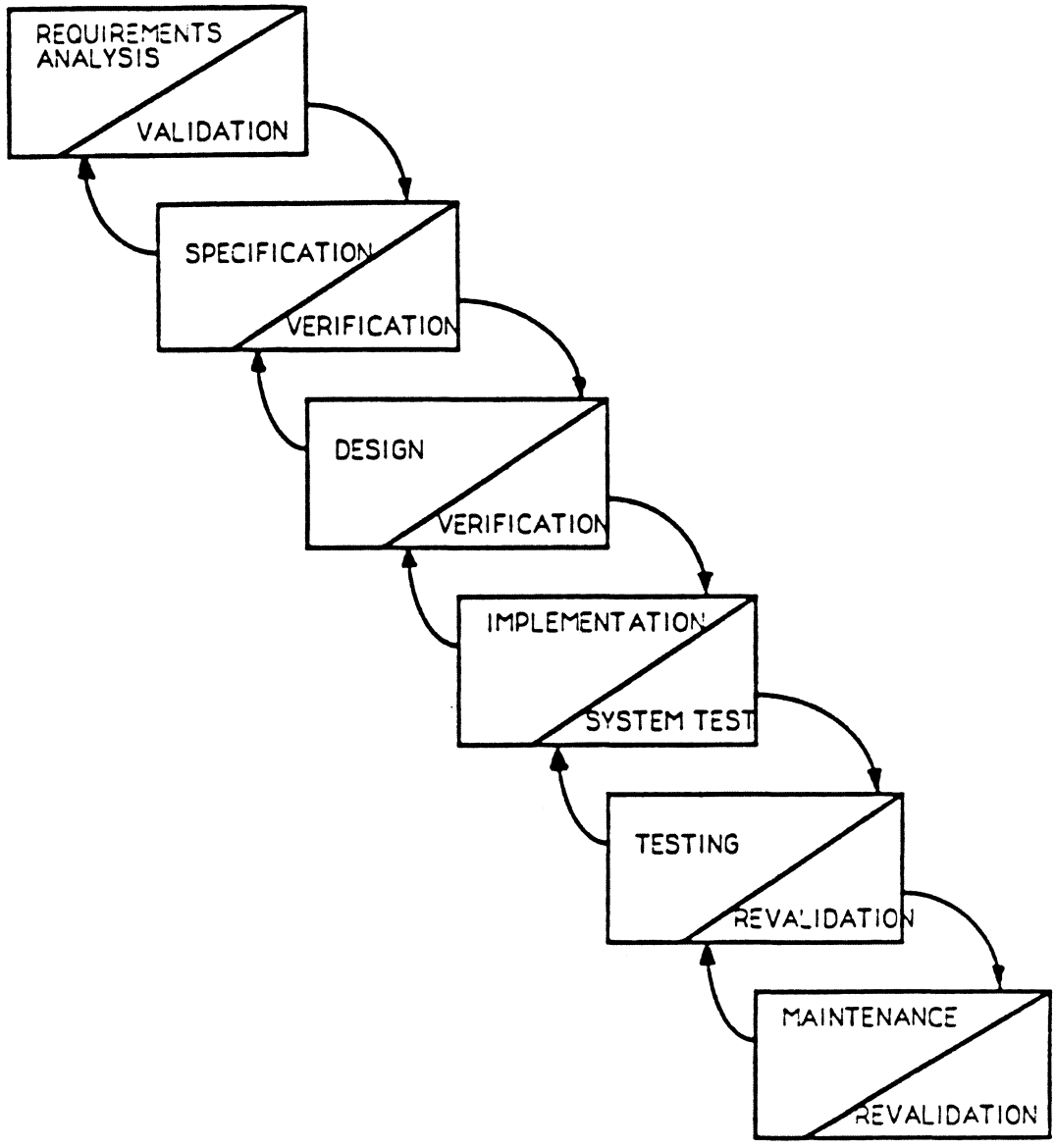


Figure 1. Boehm's Waterfall Model

ronment defines how one expresses a computation as well as what one can express. Hence, it is natural to view tools as elements of an environment that supports a given methodology [ARTJ86]. With these distinctions in mind, three software development methodologies are presented below to further familiarize the reader with methodological characteristics.

Over the past decade numerous software development methodologies have been devised. Among them the more prominent ones are: Structured Analysis and Design Techniques (SADT) developed by SofTech, Software Requirements Engineering Methodology (SREM) developed by TRW, and User Software Engineering (USE) designed and developed by A.I. Wasserman.

Each of the above mentioned methodologies is designed to achieve certain goals and is guided by accepted software engineering principles. The application of these principles to a software development process results in the achievement of desired goal(s). Each methodology demands its own set of tools and techniques for achieving its stated objectives. For each methodology the following elements are highlighted:

- the methodology's fundamental concepts, and

- its relationship to the software life cycle process.

Based on a set of linkages described in Chapter 3 and information collected from the literature survey, additional elements emphasized are:

- the interrelationships between its objectives, principles and attributes, and

- the properties identifiable in the produced product.

Please refer to Appendix A for details on additional software development methodologies.

2.3.2.1 Structured Analysis and Design Technique (SADT)

SADT is developed by D. Ross and his colleagues at SofTech. As the name implies, SADT is more than just a design method. SADT addresses the requirements analysis phase of the software life cycle. In SADT, Structured Analysis (SA) is built on top-down decomposition. The SADT philosophy [ROSD79] is that the functional specification of the system should be analyzed in as much detail as possible without letting implementation decisions intrude. When a stage is reached such that it is no longer possible to analyze the system, then a decision is made and documented. In SADT, the Structured Analysis Language provides a top-down, detail specification of a structured analysis model. Formulation of language constructs are supported by a graphical interface. Thus, SADT consists mainly of three elements:

- a set of methods that assist the analyst in understanding a complex subject (characteristic of a methodology),
- a graphical language for communicating that understanding, and
- a set of management and human-factors considerations for guiding and controlling the use of the methods and the language.

As a whole, SADT is a self-contained environment that supports a methodological approach to the requirements definition phase of the software development life cycle process.

SADT: Analysis of Objectives, Principles and Attributes: A careful examination and literature review reveals that SADT methodology embraces the following objectives, principles and attributes [ADDD85, DICM78, ROSD77, ROSD77a, ROSD79]. The objectives stressed in the methodology are maintainability, reusability, reliability, portability and adaptability. In order to achieve these objectives the principles enunciated are hierarchical decomposition, functional decomposition, information hiding, stepwise refinement (iterative enhancement) and documentation. Application of these principles imply product attributes like coupling, cohesion, complexity, well-defined interface, ease of change, traceability, visibility of behavior and early error detection.

An SADT model is a graphic representation of the hierarchical structure of a system, decomposed with a special purpose in mind. A model is structured so that it gradually exposes more detail. The SADT approach is one of strict decomposition by data or function. The SADT mechanism notation provides a concrete way of utilizing the principle of information hiding, as well as abstractions such as monitors, abstract data types and so forth. The principle of stepwise refinement is reflected where the SA maxim is applied recursively. Successive iterations are carried out until the design is final. The principle of documentation is also applied to help achieve the goals of the methodology. Procedural documentation is a requirement in SADT along with the procedural structuring of modules; final documentation is also advised [GRIS78].

The application of these principles induces attributes like reduced coupling and enhanced cohesion at the code level. Other desirable attributes expected to be present in the code are: reduced complexity, traceability, visibility of behavior, and ease of change.

2.3.2.2 *Software Requirements Engineering Methodology (SREM)*

The second software development methodology presented is Software Requirements Engineering Methodology (SREM). The SREM methodology, developed at TRW by M.W. Alford [ALFM77, ALFM85] is used for generating software requirements for large, real-time, unmanned weapon systems. SREM addresses the requirements specification stage of the software life cycle. SREM consists of a formal requirements description language called the Requirements Specification Language (RSL) and a set of automated tools that operate on the RSL description to check for completeness and consistency and to generate simulations for validation of the correctness of the requirements. A graphical notation called R-Nets is used in conjunction with RSL to express parallel operations, to specify explicit interfaces to other subsystems, and to tie validation assertions to particular points in the specification.

SREM, through an application of its defined methodology, strives to achieve the following goals:

- a structured medium or language for the statement of requirements,

- an integrated set of computer-aided tools to assure consistency, completeness, and correctness, and

- a structured approach for developing the requirements in this language, and for validating them using the tools.

Thus SREM has all the requisites of a methodology. It has a set of methods that is used to develop requirements specifications, and it provides software tools like RSL and R-Nets for depicting embedded computer system components in a real-time environment.

SREM: Analysis of Objectives, Principles and Attributes: A review of literature on Requirements Engineering [ADDD85, ALFM77, ALFM85, BELT77, ROMG85, RZEW85, SCHP85] indicates the following stated objectives: reliability, testability, portability, correctness, reusability, maintainability and adaptability. To attain these objectives the principles employed are : functional decomposition, stepwise refinement, information hiding, documentation, and life cycle verification. SREM chooses to use functional decomposition over hierarchical decomposition because most requirements are stated at the sub-function level or below and the ease of design testing predicated on such a decomposition. The principle of stepwise refinement is applied through an iterative process of analyzing the problem. SREM emphasizes the importance of documentation in that the requirements must be systematically documented. Other principles used to achieve the objectives are: information hiding, and life cycle verification.

Based on linkages among principles and attributes described in Chapter 3 the application of above principles is expected to affect following attributes at the code level: complexity, ease of change, traceability, coupling, cohesion, and well-defined interface.

2.3.2.3 User Software Engineering (USE)

The USE methodology is developed by A.I. Wasserman [WASA82]. USE addresses the specification phase of the software life cycle and is directed towards the specification and development of interactive informations systems (IIS). USE is designed to develop reliable, easy to use and less costly IISs. USE takes into consideration factors ranging from programming practices to hardware selection and from system definition to psychological factors. USE attempts to combine the systematic approach to software development inherent in the life cycle approach with the construction techniques successfully used in

the INTERLISP [TEIL81] and Smalltalk [KAYA77] environments. USE is supported by a set of tools in the Unix environment specifically directed toward meeting the needs of IIS. The environment consists of three parts: 1) a user interface, 2) a database, and 3) the operations mapped from the user onto the database.

The steps involved in developing an IIS through the USE methodology are summarized below:

- identify system objectives and constraints, including conflicts of interest among user groups,

- model the existing system using a requirements analysis method (Structured Systems Analysis for instance),

- construct a conceptual model of the database,

- produce a system dictionary containing the names of all operations, all data items, and all data flows, and

- review the analysis results within the development group and with the users and customers [ADDD85].

To date, five automated tools have been developed. The tools are:

- Transition Diagram Interpreter (TDI)- a tool for encoding transition diagrams,

- Troll- a tool that provides a relational algebra-like interface to a small relational database system,

- RApid Prototypes of Interactive Dialogues (RAPID)- a tool combining TDI and Troll that permits the rapid construction of partial systems,

- Programming LAnguage for INteraction (PLAIN)- a procedural programming language to support the definition and manipulation of relational data bases, and

- USE Control System - a tool that supports a modular organization of software system.

In summary, the USE methodology provides the developer of interactive information systems with the methods and tools that improve the quality of such systems and the process by which they are built.

USE: Analysis of Objectives, Principles and Attributes: A review of literature [ADDD85, WASA75, WASA82] on User Software Engineering indicates that the methodology has the following stated objectives: reliability, maintainability, testability, portability and adaptability. The methodology embraces the use of hierarchical decomposition, functional decomposition, documentation, stepwise refinement, structured programming and information hiding as principles for achieving the above mentioned objectives.

At the code level one can expect the presence of following attributes to be affected: complexity, coupling, cohesion, readability, traceability, and ease of change.

In summary, individual tools can support methodological approaches to solve a given task. Tools and simple methods combine to provide a more extensive and universally applicable methodology. It is often the case that this combination constitutes an environment, that is, not only suggesting a methodological approach, but also providing support facilities for decision processes and information analysis.

2.4 *Toward Assessing and Comparing Methodologies*

When one has a variety of methodologies with their own individual sets of tools and techniques, an unavoidable problem of comparison between the methodologies arises. In particular, one desires to select a methodology that is most appropriate for a given application through an evaluation of methodological characteristics. The evaluation process should not be just descriptive, but also prescriptive, i.e., it should be able not only to pinpoint the weaknesses, but also suggest remedies for eliminating those weaknesses. The remedies must be practical in the sense that they can be implemented with little difficulty.

One approach to comparing dissimilar software development methodologies is the development of an assessment procedure based on the fundamentals of software engineering, but what are those fundamentals? In simple terms, the principal aim of software engineering is the production of high-quality software within the constraints of the available resources in time, money, people and machinery. According to F. Bauer [BAUF72] the goal of software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines. This definition outlines the requisites of a methodology which are:

- embodiment of sound engineering principles,
- economic production of software,
- reliable and efficient on existing computers, and

- assessable over the entire life span of the software.

To achieve the above goal there are a number of objectives which are commonly identifiable in various methodologies and which are in consonant with the above definition of software engineering. To achieve each objective, however, certain principles must be enforced by the methodology. In turn, the employment of such principles in the development process induces certain attributes in the product (programs and documentation). The existence of these attributes imply a corresponding existence of identifiable product properties.

Clearly, a linkage exists between objectives, principles, attributes and properties. This perceived set of linkages is precisely the foundation needed to evaluate dissimilar methodologies on common terms. Such linkages are depicted in Figure 2 on page 26 [ARTJ86] and are discussed in detail in the following chapter.

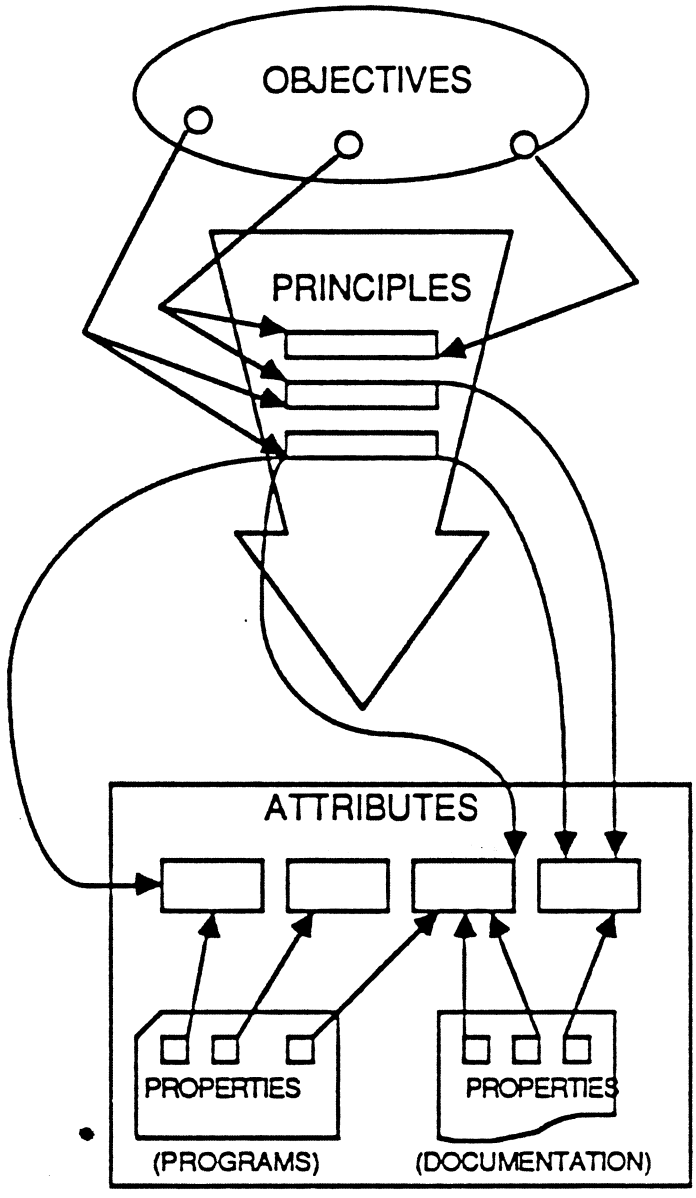


Figure 2. Objectives, Principles, Attributes and Properties

3.0 On Evaluating Software Development Methodologies

A primary focus of the research described in this thesis is the evaluation of software development methodologies. Exploiting the background material in Chapter 2, this chapter discusses the relationships among software engineering objectives, principles and attributes, followed by a description of the evaluation process which explains how these relationships (henceforth called linkages in this thesis) provide a fundamental basis for a procedural approach to evaluating methodologies. Together, these two components (linkages and evaluation process) form the backbone for a procedural approach to evaluating software development methodologies.

3.1 Establishing Linkages among Software Engineering Objectives, Principles and Attributes

The goal of software engineering as defined by Bauer [BAUF72], is, "the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines." According to this definition, the software design process and the underlying support methodology should be based on sound engineering principles in order to economically create software on existing computers which (software) is efficient and reliable over the entire life of the software.

The evaluation of such a methodology is predicated on the existence of fundamental sets of linkages among those "sound engineering principles". The sets of linkages are based on the fact that objectives can be identified within any methodology, from which one can deduce principles that guide the software development process itself. Working together, the objectives, principles and the software development process provide for the creation of programs (including documentation) that have distinct desirable attributes.

It is the hypothesis of this research that there exist clear-cut, well-defined relationships among the stated objectives of a methodology and the software development principles adopted to achieve these objectives. That is, to achieve a particular objective of a methodology, one must adopt and utilize a specific set of software engineering principles. The application of those principles ensures the attainment of desired objectives stated by a methodology, and thereby, establishes a set of linkages among objectives and principles. For example, to achieve the objective of maintainability, one is expected to apply the principles of hierarchical and/or functional decompositions, information hiding, stepwise refinement, structured programming, and documentation. In turn, the ap-

plication of a software development principle induces specific attributes in the product; hence, a set of relationships among the principles and attributes subsequently exists. For example, the application of stepwise refinement is expected to reduce coupling, enhance cohesion, and reduce complexity. This research claims that the espoused objectives of a methodology are inextricably tied to the application of specific software engineering principles, which in turn, predictably induce attributes in the final product.

As evidenced in Chapters 1 and 2, the terms objectives, principles, and attributes have been frequently used. In retrospect, Chapter 1 presents the motivation for this research, while Chapter 2 describes the foundation of software engineering. The following sub-section provide details of the above mentioned terms and examines their types, their meanings, how can one use them and how they are related to each other.

3.1.1 Software Engineering Objectives

Each methodology should espouse a particular set of well-defined goals. These goals often vary among methodologies because systems developed within methodological frameworks differ in emphasis and usage. That is, systems are generally developed with a particular use in mind. There is no one set of standards to which all methodologies and systems must adhere. Some systems are built to carry astronauts into space; they must be reliable. Other systems may have extensive life-spans and require design and implementation focused around maintainability, correctness, and adaptability. Nonetheless, system designs are based on methodologies to facilitate the attainment of crucial goals and objectives. Expanding on the relationships between systems and methodologies, the remainder of this section presents a brief discussion of the seven most widely accepted software engineering objectives.

A survey of software literature [ARTL85, BATD77, BOEB78, BOWT85, NAUP69, McCJ77, ROSD75] reveals a number of software engineering objectives. Out of these, seven are commonly recognized in numerous methodologies. They are:

- Maintainability,
- Correctness,
- Reusability,
- Testability,
- Reliability,
- Portability, and
- Adaptability

These objectives are presented below in detail.

3.1.1.1 Maintainability

Intuitively, maintainability, is defined as the ease with which a system can be corrected when "bugs" are discovered during the system's productive lifetime. More formally, maintainability is the ease with which corrections can be made to respond to recognized inadequacies [ARTJ86, BOEB78, YOUE79].

There are a variety of reasons why maintainability is an essential and important objective of software engineering. Computer software is often changing; there are always er-

rors to fix, enhancements to add, and optimizations to make. In addition to existing problems whose solutions mandate software changes, the act of changing the software itself can introduce additional problems. Moreover, it is unreasonable to assume that software testing will uncover all latent errors in large software systems. It is highly likely that errors will surface after the system is in use and considered to be functioning properly. Latent errors are often due to changes in inputs, or perhaps due to changes in the environment not originally envisioned during the system design process.

Other reasons that necessitate maintainability are the demands for new capabilities. When the software is being developed, the designer must foresee such modifications, enhancements and extensions, and structure the software in a way that facilitates the installation of the new capabilities.

The objective of maintainability is achieved through the application of certain principles of software development, e.g. hierarchical and functional decompositions, information hiding, structured programming, stepwise refinement and so forth. These principles are discussed later in detail.

3.1.1.2 Correctness

The second objective, correctness, can be defined as strict adherence to specified requirements; that is, the software must be a solution to the right problem and must work under all stipulated circumstances. Intuitively, correctness is the extent to which software satisfies its specification requirements and fulfills the user's objectives [ARTJ86].

Primarily, correctness can be assessed by testing the system or observing the system in use. It is emphasized that correctness should not be confused with reliability. According

to Wasserman [BATD77] correctness is a programmer's notion that the software performs according to the specifications. Reliability, on the other hand, is more commonly perceived as software doing what is desired when it is desired. For example, correct programs may be unreliable if they run on unreliable computers or when the actual software specification differs from the perceived one.

Correctness is normally based on the following assumptions:

- the machine which interprets the software will function properly, and

- that the data inputs to the system will be correct [PARD72a].

From this perspective, correctness can be viewed as a function of completeness and consistency [ARTL85]. Completeness recognizes those software characteristics that provide a full implementation of the required system functionality. At the code level there are a number of measurements that reflect completeness. One of the better ways to understand and improve correctness is by tracking the number and types of functions either omitted or inadequately defined during the definition and design phase. Additionally, completeness depends on the decision structure and error handling facilities. The erratic application of control structures leads to the next measure of correctness-consistency.

Consistency begins with the definition and design process. That is, if different analysts use different design techniques, e.g. one group uses HIPO (hierarchical input process output) diagrams while other group uses Nassi-Schneiderman diagrams, design consistency becomes suspect. Standards serve to select a consistent method or way of doing things. Similarly, code consistency is an important factor. A uniform style of indenting IF-THEN-ELSE or CASE constructs fosters correctness [ARTL85].

In general, one achieves correctness through the application of the following software development principles: hierarchical decomposition, stepwise refinement, structured programming, and life cycle verification. These principles will be discussed later in the chapter.

3.1.1.3 Reusability

The third objective is reusability. Reusability is defined as the extent to which the developed software can be used in other applications [BOEB78]. Reusability is of prime importance in those modules which implement major functions of a system. The development of new systems in application areas such as command and control and management information systems invariably involves implementing a high percentage of logic that is common to all such systems. The developed software should possess desirable qualities that make reuse in generic applications feasible and practical. In general, the use of existing software is restricted to mathematical subroutine libraries, operating system facilities such as I/O handlers and various commercially available report generation and data management systems. Extensive use (or reuse) of off-the-shelf software modules significantly lessens the risk of introducing errors as well as shortens the time required for software development.

Reusability can also be viewed as a function of generality, modularity, hardware dependence, and self-documentation [ARTL85]. Generality is a primary requirement of any reusable module and is a measure of the software characteristics that help expand the usefulness of a function beyond the existing module. Modules that perform functions like data conversions, table searching, and sorting are considered general in nature. Each such module has a specific function and well-structured interface. The second factor that influences reusability is modularity. Typically, single-function, well-structured modules

are reusable [ARTL85]. Modularity encourages highly independent software components tailored to supporting only one function. A third factor that affects reusability is hardware dependence of the code. For example, software written in assembly language is often rendered non-reusable on other hardware. Lastly, for any software to be reusable, it must be documented. Self-documentation explains the function of the software. For example, the code in reusable modules should not use cryptic data names.

In general, reusability can be achieved through the application of the following software development principles: hierarchical and functional decompositions, information hiding, and documentation. These principles are also discussed later in the chapter.

3.1.1.4 Testability

The fourth objective, testability, is the ability to evaluate conformance with requirements. It can also be viewed as the effort required to test a program to insure that it performs its intended function. A software product possesses the testability characteristic to the extent that it facilitates the establishment of acceptance criteria and supports evaluation of its performance [BOEB78]. Testability is influenced by two primary elements:

- how test criteria are defined, and

- how one determines that the product satisfies the criteria previously set.

In addition to the above two criteria testability is also influenced by modularity. Modular programs are more testable as compared to those which are not modularized. In particular, it is easier to define a systematic testing process for modular programs. The

rationale behind this statement is that modularity introduces a partitioning which helps in testing modules independent of each other. From a measurement perspective, testability is often assessed through flow-charts. An evaluator who has a thorough understanding of flow-charts and system design can detect faulty logic, undetected error conditions and so forth from the corresponding flow-chart representation.

At this point, however, a few words of caution for the programmer are appropriate. One major point that should be emphasized is that testability is designed and coded into the system - it cannot be added later. With regard to actual coding, one must remember that the use of GOTOs in the code makes it more difficult to test because of the structure violation [DIJE68]. Also, excessive nesting (nesting levels of three or more) levels are difficult to understand and, therefore, difficult to test.

In addition, a programmer can facilitate source code testing by inserting diagnostic and intermediate numerical results in the software. To cite a positive example, suppose a program has universal expressions or equations like:

$$\sin^2 x + \cos^2 x = 1,$$

where sin(x) and cos(x) are computed separately. Such results can be printed or displayed as intermediate results. These results, when correct, give the designer a go-ahead. Negative examples are the utilization of a combination of global and local variables, and the overlapping use of variables for different functions. Such programming style makes it difficult to adequately test software systems.

One can achieve testability through the application of the following software development principles: hierarchical and/or functional decompositions, information hiding,

structured programming, stepwise refinement, and life cycle verification. A more detailed discussion of these principles is presented later in this chapter.

3.1.1.5 Reliability

The fifth objective is reliability. Reliability is defined as the extent to which software can be expected to perform its intended function with the required precision [ARTJ86, BOEB78]. Intuitively, reliability is the error free performance of software over time. That is, software should be resilient to errors in the input data.

From a reliability perspective, one normally expects that once in service, software will continue to operate correctly, even though the input conditions are not precisely the same as the test inputs. Thus, reliability must simultaneously prevent failure in conception, design, and construction as well as recover from failure in operation or performance. As with testability, it is absolutely essential to build reliability from the start because it is not possible to add it in the end.

In general, reliability is determined by the requirements and then introduced into the system design. The reliability of a software product depends both on the number of design errors in the software and on the use of the software in a given application. This concept is different from that of software correctness, which is primarily concerned with the consistency between the program code and its specification.

With regard to technical systems, reliability can be measured in stochastic terms. Hecht [HECH77] defines three principle functions of software reliability: measurement, estimation, and prediction. Reliability measurement uses failure interval data obtained by running a program in its actual operating environment. Reliability estimation uses fail-

ure interval data from a test environment. Reliability prediction uses program characteristics (not failure intervals) to determine software reliability and commonly considers factors such as program size, complexity and so forth. The four more prominent areas to which these software reliability measures have been applied are: Systems engineering, Project management, Operational software change management, and Software engineering technology evaluation [MUSJ84].

Generally, reliability is influenced by the following three characteristics: accuracy, consistency and modularity [ARTL85]. Accuracy (in this context) is concerned with the accuracy of data, computed results, and the avoidance of error buildup in iterative routines. Accuracy is best legislated during the definition and design phases of development. The second factor to influence reliability is consistency. Consistency provides uniform design, code, and test techniques. Standards for design techniques, coding, naming and documentation serve to improve consistency; in turn consistency helps ensure reliability. Lastly, modularity affects reliability because modular units can be easily tested, repaired and replaced, thus enhancing reliability.

One can achieve reliability through the application of the following software engineering principles: hierarchical and/or functional decompositions, information hiding, structured programming, stepwise refinement, and life cycle verification. These principles will be discussed later in the chapter.

3.1.1.6 Portability

The next widely accepted software engineering objective is portability. Portability is defined as the ease with which software can be transferred to another environment [ARTJ86, BOEB78]. In general, portability is the property of a system that permits it

(the system) to be mapped from one environment to a different environment [STEM78]. The characteristic of portability is synonymous with self-containment. One may ask - can a software product stand by itself in a different environment, or to what extent does it require additional resources?

Some common examples of portable software are programs that

- use standard library functions and routines selected for universal availability, and

- use operating system functions minimally or not at all.

Additionally, documents written to require only minimal reference to other documents for comprehension are also portable.

The original requirements play an important role in specifying the need for portability. Portability is viewed as a function of hardware independence, modularity, and software system independence [ARTL85]. The first factor, hardware independence, conveys the capability of software to compile and run on different kinds of machines. Hardware independent software should pose few problems when compiled and executed on different machines. The second factor that influences portability is modularity. Modularity encourages a highly independent structure among software components. For example, the UNIX operating system has been ported from the PDP 11/45 to the IBM and Amdahl mainframes and also to virtually every microprocessor. The original UNIX system design isolates I/O to just a few software modules. Only these select few modules have to be recoded for each target machine. The remainder of the operating system and tools are written in a portable language called "C". Finally, the last factor to affect portability is software system independence. This factor captures the application program's dependence on the software environment, and hence, the program's portability. For ex-

ample, operating systems, data-base management systems, extensions to the coding language and so forth are in many cases dependent on the software environment.

Portability can be achieved through the application of the following software development principles: hierarchical decomposition, and documentation. These principles will be discussed later in the chapter.

3.1.1.7 Adaptability

The seventh and last major software engineering objective is adaptability. Adaptability is defined as the measure of the ease with which software can accommodate to change [ARTJ86]. Poole defines adaptability as the ease with which a program can be altered to fit different user images and system constraints [POOP72]. In the software engineering literature, adaptability is commonly referred as flexibility. According to Gilb [GILT77] adaptability is "open-ended flexibility", that is, the degree to which module interfaces are designed to accommodate future changes. One comparable concept, Gilb says, is the way FM radios are often "prepared for" stereo FM, or the way cars are prepared for safety belts. Stevens [STEW81] also refers to adaptability as flexibility and defines it as the ability to adapt to new requirements not currently specified for the program.

The difficulty in achieving adaptability is that it is almost impossible to code for all functions that will be needed by the software in the future. Additionally, future needs are often the most ill-defined; one often finds it difficult to get everyone's agreement and approval on future needs. Nonetheless, adaptability is enhanced by allowing for all possibilities, rather than designing a solution specific to today's particular situation. The designer should assume that the current specifications may change, and accordingly, choose alternatives that will be least affected if changes do occur. On the other hand,

there is a temptation to provide for adaptability by trying to generalize functions far beyond what is needed. Because all future requirements cannot possibly be implemented at one time (and many turn out to be different when they finally occur), one must be careful not to go overboard in implementing unnecessary "nicities".

Factors that contribute to adaptability often include consistency, expandability, generality, modularity, and self-documentation [ARTL85]. Adherence to standards indicates consistency; in turn, consistency implies adaptability. Expandability, on the other hand, recognizes the software attributes that provide for expansion of the data or the program's functions. Enhancements are an expansion of the program's capabilities, hence adaptability goes hand in hand with expansion. Data expansion can be achieved through use of a data dictionary while functional expansion is attained through use of generic designs and code. The next factor affecting adaptability is generality. From the adaptability point of view, generality connotes the ability to expand the usefulness of a given function beyond its present scope. For example, driver modules with a high functional density should also be fairly general, and therefore adaptable. The fourth factor to influence adaptability is modularity. Modularity supports a systematic structure of highly independent modules, each of which has a well-defined interface tolerant to external changes. In general, for software to be truly adaptable, it needs the flexibility which comes from modular, interchangeable components. Lastly, self-documentation helps increase the understanding of the logic flow. In turn, the achieved understanding promotes adaptability.

One can attain adaptability through the application of the following software development principles: hierarchical and/or functional decompositions, information hiding, structured programming, stepwise refinement, and documentation.

In summary, the above seven objectives are the most commonly espoused objectives in the software engineering literature. Each methodology should attempt to achieve some of these objectives. As mentioned in the previous discussion, these objectives can be attained through the application of related software development principles. The following presentation briefly discusses these principles.

3.1.2 Software Engineering Principles

The previous section presents a discussion of the common objectives aspired by a typical software development methodology. The hypothesis mentioned earlier claims that the attainment of these objectives is accomplished through the application of software engineering principles. This thesis attempts to show the existence of direct relationship(s) among the objectives and the principles. By virtue of these relationships, one knows exactly what principles yield which objectives.

Various software engineering principles are mentioned in the literature. The principles listed below are associated with the creative process by which programs and documentation are produced. According to McLure [BATD77], the creative process can be visualized as constructing a bridge to cover the distance between the methodology objectives and the final product. Each principle attempts to tackle the problem in its own way. For example,

- In stepwise refinement the starting point is an abstract program, which, if it could be implemented, would solve the whole problem [JACM76],

- In functional decomposition the starting point is a dissection of the whole problem into fewer pieces [JACM76],

- Structured programming advocates that any program can be written using just a few constructs [DIJE65], and

- Information hiding is wherein a module is associated with a design decision which the module hides from higher level routines that appeal to it.

Seven prominent software engineering principles are commonly used in methodology development. These principles are listed below:

- Hierarchical Decomposition,

- Functional Decomposition,

- Information Hiding,

- Stepwise Refinement,

- Structured Programming,

- Documentation, and

- Life Cycle Verification.

The following text presents a brief discussion of each of the seven principles.

3.1.2.1 Hierarchical Decomposition

In the late 60's and early 70's, hierarchical structuring or decomposition was the "buzzword" [PARD74]. The principle of hierarchical decomposition, in recent years, is

more commonly known as top-down specification. The term top-down refers to the direction of the steps taken in the design and development process. Top-down design has been defined as a system where decisions are made first which concern the highest-level functions of the software. These decisions are then successively refined until the level of existing resources is reached [BATD77, DeMT79, FAIR85, McCC75, PARD71, ROSD75, TAUR77]. In simple terms, in hierarchical decomposition software systems are usually divided into many components where each component solves a subproblem that partition the original problem

Hierarchical decomposition employs the techniques of analysis and synthesis - that is, taking the components apart and putting them together [ROSD75]. While one decomposes a subject by recognizing the different subparts of which it is composed, care must be taken to examine the overall decomposition and look for similarities among the lower-level constituents with an eye toward recomposing collections of them into larger constructs. For example, pure decomposition would never result in recognizing subroutines that are needed in separated parts of the decomposition.

Generally, in top-down design, the strategy consists of the following three steps:

- determine what decisions are to be made,

- rank the decisions in importance, and

- design the structure by making the most important decisions first. That is, defer detail decisions as long as possible. At each level, choose the structure that preserves the most options for later decisions.

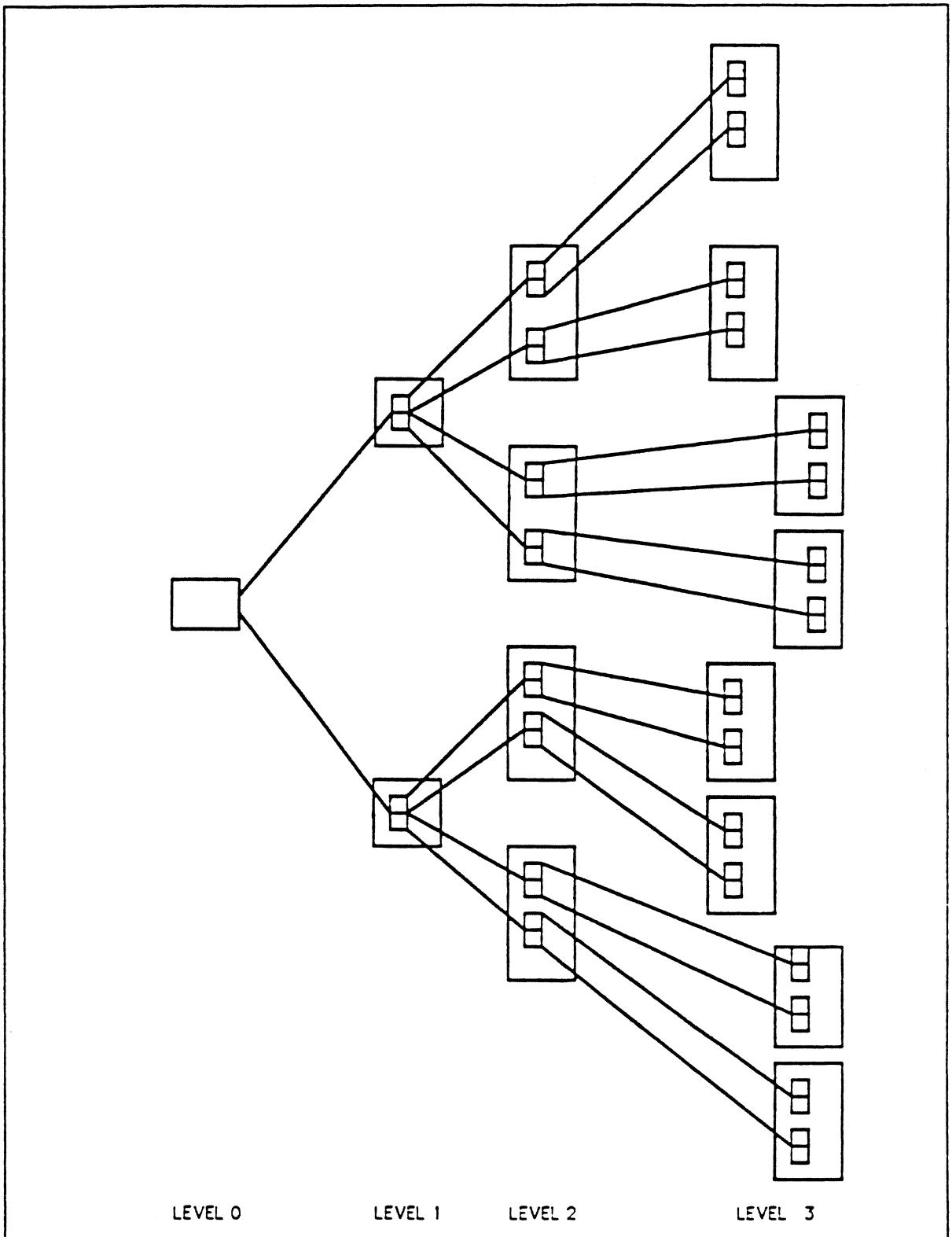


Figure 3. Hierarchical Decomposition

Typically, the highest level is given functional responsibility for the entire process. As shown in Figure 3 on page 44, the decomposition process is the division of a task or module into sub-tasks or sub-modules. In the best case of decomposition, these sub-modules are independent of each other. This independent nature of modules helps the system to be adaptable to the changes in the environment.

Wasserman [BATD77] suggests that the top-down approach to problem-solving is valuable, because it serves to provide a structure that facilitates intellectual mastery of the problem. The structure of the problem solution is then reflected in the resulting program structure. Also, via top-down development, a user can visualize the top level interfaces very early during the system development. Changes can then be easily integrated early in the development cycle.

The application of hierarchical decomposition is expected to induce following attributes at the product (code and documentation) level: low coupling, high cohesion, reduced complexity, and ease of change. These attributes will be discussed later in detail.

3.1.2.2 Functional Decomposition

The second fundamental principle is functional decomposition. The principle of functional decomposition states that the components of a system should be partitioned along functional boundaries. Characteristically, in any design, a system is broken down into components to make the system more manageable. One of the criteria used for the decomposition of a system is called "functional decomposition" which is the process of identifying and separating the components based on functional differences. For example, individual types of I/O devices or processing modules may be broken down as separate components because it is easier to deal with them individually.

Often components derived in the above mentioned fashion are not closely related and can be designed in such a way as to be independent of one another. This greatly facilitates the design process by allowing the design of functionally different components to proceed independently and often in parallel. In addition, it increases the inherent modifiability of the system. It is this kind of decomposition or modularization that has resulted in standard mathematical subroutines and in proposals for standard business-oriented modules, all of which can be used as a base for more complex software systems.

In using functional decomposition, one begins with the ultimate function to be performed; this function is then divided into sub-functions by decomposing it with respect to time order, data flow, or some other criterion. The choice of "what to decompose with respect to" has a major impact on the "goodness" of the resulting product.

One can expect to find the following attributes in the code when the principle of functional decomposition is applied. The attributes are: coupling, cohesion, complexity, and ease of change. These will be discussed in more detail later.

3.1.2.3 Information Hiding

The third software engineering principle is information hiding. The concept of information hiding is first introduced by Parnas [PARD71, PARD72a]. Parnas suggests that the concept of information hiding helps produce flexible software and recommends the following three steps:

- the identification of items that are likely to change (these items are called "secrets"),

- locating specialized components in separate modules, and

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

The concept of information hiding has been highly acclaimed as a software principle in the literature [DeMT79, FAIR85, PETL81, PRER82, ROSD75]. When a software system is designed using the information hiding approach, each module in the system hides or conceals the internal details of its processing activities; modules communicate with each other only through well-defined interfaces. In other words, modules are specified and designed so that information (procedure and data) contained within a module is inaccessible to other modules that have no need for such information. This encapsulation of information helps improve the flexibility of software. Besides encapsulating difficult and changeable decisions, other candidates for information hiding include:

- complex data-structures (stacks, pointer structures), their internal linkages, and the implementation details of the procedures that manipulate them,

- the format of control blocks such as those for queues in an operating system, and

- shifting, masking, and other machine dependent details [FAIR85].

The use of information hiding provides greatest benefits when modifications are required during testing, and later during software maintenance [PRER82]. Since most data and procedures are "hidden" from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

The principle of information hiding when applied is expected to induce the following attributes in the code: coupling, cohesion, complexity, well-defined interface, and ease of change; each of these will be discussed separately in the next section.

3.1.2.4 Stepwise Refinement

The fourth principle, stepwise refinement, is one of the oldest and most widely used methods of program design. It is also known as "stepwise program development" and "successive refinement" [FAIR85]. Stepwise refinement is a careful statement of the good techniques long practiced by the better programmers. It is a variant of the top-down approach [GILP83]. Through stepwise refinement one develops a program by successive refinement steps, each of which partitions a portion of the potential program text into several parts. Initial steps deal with large program segments and broad design decisions, while later steps deal with small program segments and detail design decisions. In the refinement process additional details are incorporated at each step. Nevertheless, the design decisions are postponed as long as possible. This technique allows the designer to argue convincingly that the resulting software is consistent with the design specifications.

As originally described by Wirth [WIRN71], stepwise refinement involves the following activities:

- decomposing design decisions to elementary levels,
- isolating design aspects that are not truly interdependent,
- postponing decisions concerning representation details as long as possible, and
- carefully demonstrating that each successive step in the refinement process is a faithful expansion of previous step.

On the application of stepwise refinement principle, the resulting product is expected to exhibit the presence of following attributes in the code: coupling, cohesion, and complexity. A brief discussion of the attributes present at the code level is presented in subsequent paragraphs.

3.1.2.5 Structured Programming

The fifth principle is structured programming. The concept of structured programming is introduced by E.W. Dijkstra [DIJE65, DIJE72] and involves the use of only three constructs for programming. These constructs are shown in Figure 4 on page 50.

As seen from the Figure 4 on page 50 the three forms allowed are:

- Sequence, which indicates the order in which execution should take place, e.g. execute S1, then execute S2,

- Choice, which allows selection of one path out of two on some condition (C) being true, e.g. IF-THEN-ELSE statement, and

- Iteration, which is same as the loop. On some condition C; if false, calculation proceeds to next step along arrow labeled false. If C is true, then S is executed and C is tested again. Thus S is repeatedly executed while C continues to be true. Examples of iterative constructs include the WHILE <condition> DO in Pascal and DO loop in Fortran.

According to Dijkstra, and Boehm and Jacopini [BOEC66], any programmable calculation can be programmed using only the above three constructs. With the use of these forms, they claim, the resulting program is easily understood by programmers - even at

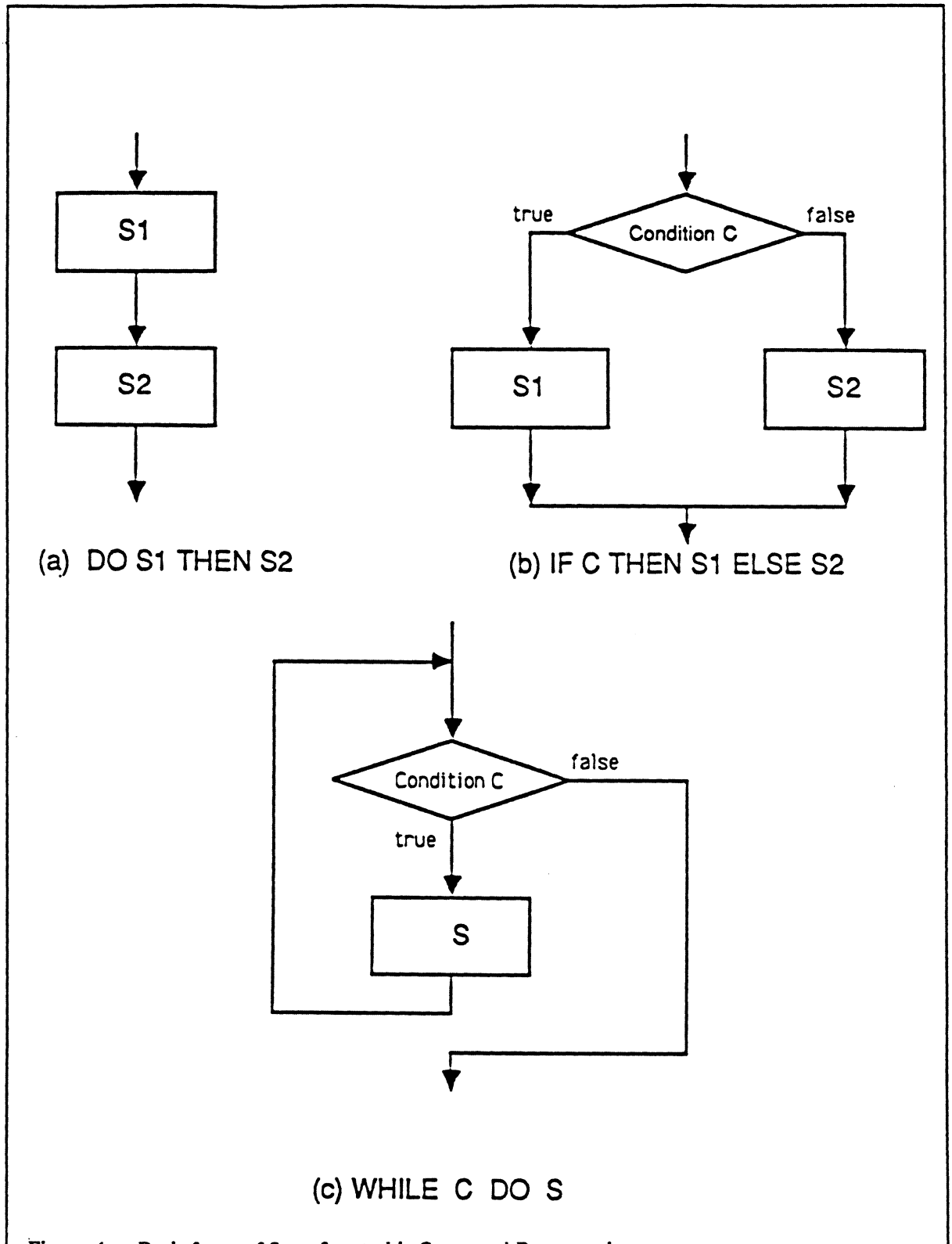


Figure 4. Basic forms of flow of control in Structured Programming

first reading. Justification for the above statement follows from the fact that there are fewer intricacies to be deciphered. Thus, maintenance personnel can quickly discover the portions of the program that require maintenance or modification. In contrast, program structures that have many blocks intricately interconnected are difficult or impossible to comprehend fully.

The application of structured programming is expected to induce the following attributes in the resulting code. They are: complexity and readability. The attributes will be discussed later in this chapter.

3.1.2.6 Documentation

The sixth principle, documentation, is the management of supporting documents (system specifications, user manual etc.) throughout the life cycle of a software system [ARTJ86]. The principle of documentation emphasizes that the complete system must be fully recorded, so that each decision taken in the development can be traced back to its reasons, and every statement in the final code can be traced to a corresponding element in the problem specification. Horowitz [HORE86], suggests that documentation permits traceability through all phases of the software life cycle.

According to Bauer [BAUF72], documentation is the information about the software system available in writing. Arbitrary pieces of paper cannot be a part of documentation. To be effective, documentation must have purpose, content and clarity. Documentation is part of the software development process, inseparable from the analytical, design, programming, coding, and testing phases, and must be integrated into these activities. Tausworthe [TAUR77] terms this as the concurrent documentation principle because the definition, design, coding and verification phases of development cannot be regarded as

complete until the documentation is complete. This view, which reflects the importance and place of documentation, is taken because good documentation is inextricably bound up in each facet of the project, from conception to design, coding, testing etc.

System or project documentation is classified into four main categories:

- The user's manual - which is subdivided into three parts: introductory manual, reference manual, and operator's guide,

- The conceptual description - which is developed as the project proceeds. It serves as an introduction, overview, and specification of the project,

- The design documentation - which describes the current state of the project during the design phase. It defines the input for the construction phase, and

- The product documentation - which describes the current state of the project during the construction and maintenance phase. The program itself is the part of the product documentation [BAUF72].

From this perspective, there must be guidelines or standards stating what information is considered part of which document. Moreover, because documentation may not remain static for long, the life-span of current documentation must be considered. Each time there is a change to a system, the change must be recorded in what is called a revision of the document(s). The documentation must have a detailed record of the past, the present and the future activities related to the system.

The use of the principle of documentation is expected to induce the attributes of complexity, readability, ease of change, and traceability in the resulting product.

3.1.2.7 Life Cycle Verification

The last software engineering principle considered is that of life cycle verification, that is, the verification of requirements throughout the design, development, and maintenance phases of the life cycle [ARTJ86]. This principle is also termed as confirmability [ROSD75].

The principle of life cycle verification directs attention to methods for assessing whether stated goals have been achieved [ROSD75]. Applied to design issues, life cycle verification is the verification to ensure that the structuring of a system is done in accordance with the requirements. Applied to development issues, life cycle verification is making sure that the correct notations are used such that the notations require explicit specification of constraints which affect the correctness of a design or implementation (e.g. data declarations that specify range of values, units of value, as well as modes of representation). Basically, life cycle verification is making sure that nothing has been overlooked.

Tausworthe [TAUR77] defines life cycle verification as that aspect of development asserting that the program response falls within the acceptable limits of functionally specified behavior. It (1) testifies that design and production activities conform to program requirements and project standards, (2) generates test procedures and conducts tests to evaluate the program behavior, (3) identifies all anomalies for corrective action, and (4) ultimately certifies the program is ready for user operation.

The application of life cycle verification is expected to induce the following attributes at the code level. The attributes are: visibility of behavior and early error detection.

In summary, the seven software engineering principles discussed above are instrumental in achieving desired objectives of a methodology. As mentioned earlier, to achieve an objective, one applies a specific set of software engineering principles. The application of these principles is evident at the product level. That is, specific software engineering attributes are induced in the product as a direct result of the application of a software engineering principle. The next section presents a discussion of these software attributes.

3.1.3 Software Attributes

The earlier discussion of software engineering objectives and principles focuses on system development through the use of methodologies. It presents the software development process from more theoretical perspective as compared to the practical aspect of producing a product via the association of objectives and principles. At the product level, one is interested in identifying the software and documentation characteristics that are desirable and beneficial. Such characterics are called attributes. This research claims that once a software engineering principle is selected (to achieve a certain objective), the application of that principle directly influences product characteristics; that is, it induces specific attributes in the code which are a direct consequence of the application of that particular principle. For example, use of hierarchical decomposition is expected to reduce coupling and complexity while enhancing cohesion and ease of change. Similarly, the application of structured programming is expected to enhance readability of the code and reduce the complexity.

The following discussion is directed towards presenting the nine commonly identified software attributes. They are:

- Coupling,
- Cohesion,
- Complexity,
- Well-defined Interface,
- Readability,
- Ease of Change,
- Traceability,
- Visibility of Behavior, and
- Early Error Detection.

The above attributes are individually discussed below.

3.1.3.1 Coupling

The first software attribute is coupling. The concept of coupling is introduced by L.L. Constantine [STEW74]. According to Constantine coupling is a measure of the strength of association established by a connection from one module (or routine) to another where that connection is defined by a reference to some label or address defined elsewhere.

From the above definition, it is evident that coupling primarily results from one or more connections between modules. A connection exists when an element of code references a location in memory defined elsewhere. In more general terms, a connection exists when two elements of code reference the same location in memory, usually to share a data item [STEW81]. Two common situations that give rise to coupling are: use of global variables in code, and use of parameters. The use of global variables introduces connections between all the modules that refer to the common global variables whereas the use of parameters directly connects modules through the parameters.

The connections discussed above introduce interdependence among modules. This dependence can adversely affect the modifiability of the software because if one wishes to change a module he (or she) may inadvertently introduce changes in the dependent module(s). Such a phenomenon is called the "ripple effect". Obviously, a programmer would like to minimize the dependency among modules in order to reduce the possibility of the "ripple effect". A programmer would therefore, wish to eliminate coupling. Software cannot function, however, without some amount of coupling. Some connections must exist among modules in a program, or else they would not be part of the same program. Subsequently, coupling cannot be eliminated. A programmer at best can only minimize coupling. From this view point it is not only important to identify coupling, but also to identify factors that affect the degree of coupling.

The degree of coupling between modules depends on a number of factors that can be manipulated by the programmer, e.g. how complicated the connection is, the type of connection, the size of connection, and the type of communication via the connection. Subsequent discussion presents the dependence of coupling on these factors.

The first factor, complexity of connection, is determined by whether the connection is due to use of a common environment (such as global variables) or due to use of parameters. According to Constantine [STEW74], connections due to common environment are less complex from the coupling viewpoint than those introduced by use of parameters. The reason for such a complex coupling is that every element in the common environment, whether used by particular module or not, constitutes a separate connection along which errors or changes can propagate.

The second factor that influences the degree of coupling is the type of connection between modules. Connections are partitioned into two categories: 1) connections that address or refer to a module as an entity by its name, and 2) connections that refer to internal elements of a module. The first category, reference to a module by its name, is commonly encountered in COBOL. Such a connection introduces less coupling than the second category which is referencing internal components of a module [STEW81]. In the second category (referencing internal component), one needs to take into account the entire content of that module to correct an error or make a change. The additional effort required in such a situation introduces extra coupling.

The third factor that affects the degree of coupling is the size of connection. The size of connection is determined by the amount of data passed between two modules - the more data, higher the coupling. The fourth factor influencing the degree of coupling between modules is the type of information passed via a connection. Two types of information is passed across a connection, e.g. data and control. Data is simple information such as computational items or simple arguments. Control, on the other hand, is the information from which decisions are made in a subordinate or superordinate module. Control information is also termed as a "flag" or "switch". Coupling associated with passage of data items is less severe than that caused by passage of control [YOUE79]. According

to Constantine [STEW74], modules must pass at least data to another module(s) or they cannot functionally be part of a system. Data coupling is sufficient for a system to function. The conclusion is that all communication of control is not only extraneous, but also introduces needless additional coupling.

To summarize, a programmer strives for low coupling because,

- the fewer the connections there are between modules, the less chance there is for the ripple effect,

- one should be able to change a module with minimum risk of having to change another module; that is, each change should affect as few modules as possible, and

- while maintaining a module, one does not want to worry about the interal (coding) details of any other module [PAGM80].

Several authors have classified coupling into five distinct classes in order to provide the programmer with guidelines for minimizing coupling in code [MYEG75, PAGM80, STEW74, STEW81, TAUR77]. The degree of coupling associated with these five classes may be represented on a spectrum from a low (desirable) coupling to a high (undesirable) coupling. The five classes are:

- Data Coupling,

- Common Coupling,

- Control coupling,

- External Coupling, and

- Content Coupling.

The first class of coupling, data coupling, introduces the lowest coupling, whereas the fifth class of coupling, content coupling, is the highest coupling and therefore should always be avoided. The degree of coupling increases as one moves from data coupling toward content coupling in the above list. A pictorial representation of the above classes of coupling is shown in Figure 5 on page 60. The first class of coupling is data coupling. Data coupling is the lowest form of coupling. It exists if 1) one module calls another module, and 2) if all input and output communication is in the form of arguments or parameters passed through call-sequence interface, and 3) if all such parameters are data (not control) elements. Constantine [STEW74] has shown that this data coupling is sufficient for any program to function.

The second class of coupling is called common coupling. Common coupling exists when modules (or routines) reference data held in a "common pool", such that the common data is accessible to all modules. For example, global variables in Pascal and common data in Fortran personify the common pool concept. The use of such common data creates coupling because it binds together the entire set of modules using the common pool, irrespective of whether the modules are related or not.

The third class of coupling is called control coupling. Two modules are control coupled if one module passes a flag or a set of flags (control data) as argument(s) to the other [TAUR77]. Such connection(s) is undesirable because it directly influences the functioning of the receiving module, thus affecting the independence of two modules.

The fourth class of coupling, external coupling, exists when two modules refer to a common data structure. External coupling is also called stamp coupling [PAGM80]. External coupling is considered high because the entire usage or content of a submodule

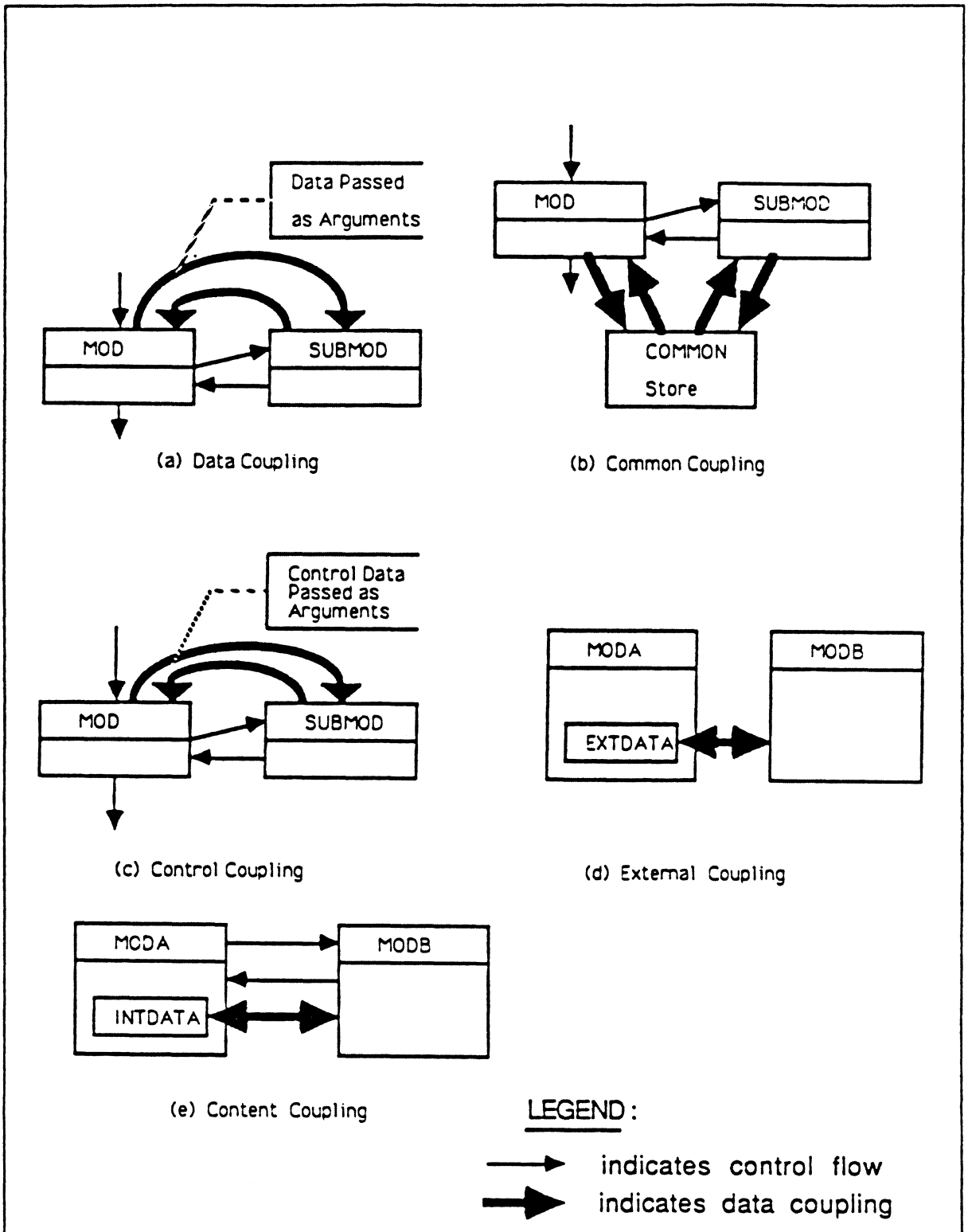


Figure 5. Classes of Coupling

may have to be taken into account to correct an error, or to make a change, or to verify that it does not create side effects.

The last class (and the worst) of coupling is content coupling. Two modules are said to exhibit content (also known as pathological) coupling if one refers to the inside of the other in any way, i.e. a direct reference to the contents of another module, by either modifying a statement in the other module, or accessing a set of internal data not externally declared [TAUR77]. Another example of content coupling is when modules share the same physical code. Assembly languages commonly indulge in such practices. It is obvious that content coupled modules are very much dependent on each other and a small change can cause disastrous effects on the other.

3.1.3.2 Cohesion

With coupling, one is concerned with intermodule relatedness or connections. For the second software engineering attribute, cohesion, one is more concerned with intramodular relatedness. The concept of cohesion is first introduced by Constantine [STEW74, YOUE79] and is described as the measure of the strength of functional association among elements within a module. Cohesion is sometimes referred as binding [MYEG75] and modular strength [TAUR77]. Unlike coupling concepts, with cohesion each module is considered in isolation. The consideration is - how tightly bound or related the module's internal elements are to one another.

A common analogy for cohesion is that of cement that holds itself and other elements together. Module cohesion is conceptualized as the cement that holds the processing elements together. Similarly, a cohesive module is a collection of statements and data items that are treated as a whole because they are closely related. In other words,

cohesiveness reflects integrity of a module. A cohesive module performs a single task within the module, requiring little interaction with modules being performed in other parts of the program. Cohesiveness, therefore, is a desirable software characteristic because it gives rise to independence among modules. Hence a programmer should strive for maximum cohesion in code.

A survey of literature presents seven different levels of cohesion commonly encountered in code [MYEG75, PAGM80, STEW74, STEW81, TAUR77, YOUE79]. The discussion of these levels provides the programmers with a set of guidelines for achieving maximum cohesion in code. The seven levels of cohesion are listed below in order of increasing strength of cohesion:

- Coincidental cohesion,
- Logical cohesion,
- Temporal cohesion,
- Procedural cohesion,
- Communicational cohesion,
- Sequential cohesion, and
- Functional cohesion.

The first level of cohesion is called coincidental cohesion. Coincidental cohesion is displayed by a module when there is no meaningful relationship between its components other than, as a coincidence, they lie in the same module. This is the worst type of

cohesion. Such a situation may be described as a random module. However, occurrence of this type of a module is rare, because it normally results from an attempt to divide code arbitrarily.

The second level of cohesion is logical cohesion. A module is called logically cohesive if its elements contribute to activities of some general category, i.e., they fall into the same logical class of similar or related functions. For example, a module called INPUTALL, which includes all processing elements that need input operations will contain logically cohesive code. It could include reading from a disk file, or a terminal or a magnetic tape. All of these are input operations and therefore logically cohesive. These types of modules are more cohesive as compared to coincidentally cohesive modules because they (logically cohesive modules) have at least some commonalty among their elements. In general, however, these modules do not perform a single function, but rather one of several different (but logically similar) functions.

The third level of cohesion is called temporal cohesion. A module is temporally cohesive if its components are involved in activities that are related in time. Temporal cohesion is similar to logical cohesion, except that the elements are related in time. That is, temporally bound elements are executed in the same time period. In such a case, there are no parameters or control data that determine which components are executed and which are not. Typical examples of such modules are: initialization modules, termination modules, and house-keeping modules. Elements in an initialization module are logically bound because initialization represents a logical class of functions. In addition, these elements are related in time. These modules are higher in degree of cohesiveness than logically cohesive modules because the same set of elements are always executed within the same time frame (i.e., no parameters and logic to determine which elements to execute).

The fourth level of cohesion is called procedural cohesion. Modules in this category have elements that are related with respect to the procedure of the program. Such modules normally result when the problem to be solved is first flow-charted and then partitioned into modules that represent two or more blocks on the flow-chart. Procedurally associated components are elements of a common procedural unit. They are combined into a module of procedural cohesion because they are found in the same procedural unit. The common procedural unit may be an iteration (loop), or a decision process, or a linear sequence of steps. Procedural cohesion often cuts across functional lines. A module with mere procedural cohesion may contain only part of a complete function, one or more functions plus parts of others, or even fragments of several functions.

Communicational cohesion represents the fifth level of cohesion. Communicational cohesion occurs in a module where the components are related through procedural cohesion and additionally, communicate with one another. In other words, the elements contribute to activities that use the same input or output data. The cohesive units either reference the same set of data or pass data only among themselves. This type of cohesion is higher on the scale than procedurally cohesive modules because the module components have the additional bond, that is, they share the same data. It is common to find communicational cohesiveness in business and commercial applications. For example, a module to print and punch a transaction file or a module that accepts data from several sources, transferring and assembling them into a report line exhibits communicational cohesion. There is a potential problem with communicational cohesion which is the temptation to share code among the activities within it. This can make it difficult to change one activity without destroying the other.

Next higher on the scale of cohesion is sequential cohesion, where the output data (or results) from one element serves as the input for the next element. In terms of data flow,

sequential cohesion combines a linear chain of successive or sequential transformations of the data. Constantine [YOUE79] conducted a series of experiments at IBM in 1968, 69 and conjectured that sequential cohesion is stronger than communicational cohesion. A sequential module may contain more than one function which makes it weaker than the best cohesion which is functional cohesion.

Functional cohesion is at the top of the strength scale and is considered the strongest type of cohesion. In a functionally cohesive module, all of the components are related directly to the performance of a single function. That is, all the elements of the module contribute to the execution of one and only one problem-related task. In other words, every element of processing is an integral part of, and is essential to the performance of a single function. Thus, a functionally cohesive module does not contain any extraneous elements. A useful technique for deciding whether a module is functionally cohesive is to write a sentence describing the purpose (function) of the module, and examining the sentence. If the sentence is a simple declarative statement (one verb and no commas), and if there are no words relating to time or sequence (e.g.; "first", "next", "then", "after", "otherwise", "when", "if" etc.) then the module is functionally cohesive.

To summarize, cohesion is a desirable attribute; that is, one should strive to maximize cohesion in the code. This can be achieved by striving toward functionally cohesive modules.

3.1.3.3 Complexity

The third software attribute to be considered is complexity. Complexity is one of the most frequently discussed issues in computer science. Controlling program complexity

is often an underlying objective of software engineering techniques. The literature offers a variety of definitions for software complexity. To cite a few, complexity is:

- the software attribute that decreases the program's clarity [ARTL85],

- the amount of remaining difficulty, once the problem is established, algorithms and data structures selected, and an initial version of the program is prepared [DUNH78], and

- the presence of anything which cannot be traced back, as a whole, to a corresponding factor in the problem or one of the implementation constraints which modify it [GRIS78].

In summary, complexity can be defined as an abstract measure of work associated with a software component [HENS85]. It is needless to say that a programmer should strive to reduce complexity in code.

Most of the problems in software development occur because of human beings (programmers and analysts) that make mistakes. The reason behind their mistakes is their limited capacity for complexity. Factors affecting complexity are:

- the amount of information that must be understood correctly,

- the accessibility of information, and

- the structure of information [YOUE79].

The first factor to affect complexity involves the consumption of information. By "amount of information" what is meant is the number of bits of data, that a programmer

must deal with in order to comprehend the problem. In simplest terms, it is related to the number of statements or arguments that are presented to the programmer at one time. In the software sense, this factor could be related to the size of a module. Generally, a 100-statement routine is more difficult to understand than a 10-statement routine.

The second reason for complexity is accessibility of information. Sometimes the accessibility of information is more important than the amount of information. For example, certain information about the software must be understood by the programmer in order to write or interpret the code correctly. Adequate comments increase the accessibility of information and decrease the possibility of error. Standardization also helps improve the situation in the sense that it can reduce complexity.

The last factor "structure of information" can be a key issue from the software complexity perspective. The main point is that information is less complex if it is presented in a linear fashion and more complex if presented in a nested fashion. Similarly information is less complex if it is presented in a positive fashion, rather than in a negative fashion. Both of these concepts have application in the writing of code. For example, it is known that certain forms of nested IF statements are considerably more difficult for an average person to understand than an equivalent sequence of simple IF statements. Similarly it is well-known that Boolean expressions involving NOT operators are generally more difficult to understand than an equivalent expression without the NOT operator.

The problem of program (or code) complexity has been approached by authors from different aspects; viz.:

- The psychological complexity of programs - a measure of human factors that affect software development [WEIL74],

- The graph-theoretic (cyclomatic) complexity - a control flow representation of program [McCT76],

- The effort equation [HALM77],

- The information flow complexity [HENS81], and

- The computational complexity - formal specification of algorithm structure, efficiency and application [WAHB84].

The approach taken in this thesis to measure complexity (and other attributes) is based on assessing product properties and is discussed later in Chapter 4.

3.1.3.4 Well-Defined Interface

The fourth software attribute is well-defined interface. An interface is defined as a shared boundary where independent systems meet and act on or communicate with each other [DALN83]. For example, when one uses the computer there is an interface between the user and the computer. The physical interface involves the particular I/O device (such as a terminal). In the case of a software module (or a procedure) the parameter list serves as an interface. The designer considers the nature and complexity of each interface to determine any effect on the development of resources, cost and schedule. The concept of an interface is interpreted to mean:

- hardware (e.g. processor and peripherals) that executes the software and devices (e.g. machines and displays) that are indirectly controlled by the software,

- software that already exists (e.g. database access routines, subroutine packages, and operating system) and must be linked to "new" software,

- people who make use of the software through terminals or other I/O devices, and

- procedures that precede or succeed the software as a sequential series of operations [PRER82].

Of the four interpretations discussed above, this research concentrates on the interface aspect at the code level. This type of interface signifies the specific connections that routines, modules and subprograms have with each other. The number of parameters passed across the boundaries of two procedures or functions constitute an interface between them. Each programmer must strive to keep each interface simple, clear, and well-defined. A simple and well-defined interface is essential for smooth operation of any software. To the contrary, an obscure interface will have an adverse effect on the software.

Generally, software is divided into subprograms, modules or routines. The special place where a subprogram "hooks" into the software is called an interface point. Data and the flow of (execution) control are transferred at the interface point [MARD84]. The guidelines for a well-defined interface (point) suggest that a module's interface should be defined and used so that

- all communication is done through the interface,

- parameters lists are kept small, and

- inputs are separated from outputs [MARD84].

The first guideline for a well-defined interface states that all communication (with other parts of a program) must be through the interface. Communicating through the interface requires that 1) each input and output is defined and described, 2) all required data is passed in when a subprogram or module is called, and 3) all results are passed back when the subprogram returns. Such an approach to communication reduces code errors. For example, incorrect definitions of FORTRAN COMMON data can give a subprogram wrong data when called. The stored data values as well as the executable statements may be correct, but the subprogram operates incorrectly because data definitions refer to the wrong storage locations.

The second guideline for a well-defined interface concerns parameter lists. Keeping parameter lists small reduces human complexity of an interface [MARD84]. According to Stevens [STEW81], the number of parameters in a parameter list should not exceed three or four; fewer parameters improve clarity and simplicity of the interface.

The last guideline to a well-defined interface refers to separation of inputs from outputs. This technique allows one to segregate what is needed (or accepted) from that which is produced. Programmers should list input parameters first and output parameters last; those parameters that are both input and output belong in the middle of the parameter list. Some languages, such as Pascal, enforce input/output separation; others like FORTRAN do not.

In the next chapter, some additional factors that contribute to a well-defined interface are identified and discussed.

3.1.3.5 Readability

The fifth software attribute is readability. The attribute, readability, is to some extent related to human complexity; that is, readability directly affects the human complexity of the code. Human complexity is concerned with the understandability of the material. If code is found to be readable then the human complexity of the code is low. In other words, if the readability is good then the user can understand the code better. More formally, readability is defined to be the difficulty in understanding a software component [ARTJ86].

Many factors affect code readability which, in turn, affect our actions toward code units. For example, as large software systems develop over expanding time periods, programmer turnover on projects increases. In general, when a change has to be made to such a system in the absence of the original programmer, it is often easier to rewrite an entire software module than to attempt to make the modification. In such circumstances the importance of readability stands out.

The above crisis emerges if the software to be modified is difficult to understand. On the other hand, if the software is easily readable (and therefore understandable), such a crisis can be averted. The importance of the readability of code is highly stressed in the programming literature [ARTL83, BAUF72,BULR76, GILP83, KERB78]. One of the primary ways to improve readability of code is through the use of comments. Well-written comments in the code make it easier for the programmer to understand and comprehend author's logic. Obviously, in the absence of comments, maintenance personnel have to rely possibly on cryptic code to understand the original intent of the software module. Even more detrimental, however, is the presence of bad comments - they are likely to mislead the maintenance personnel and cause the code to be misinterpreted.

One must be aware that comments are generally helpful in improving readability, but the use of "excessive" comments often have an adverse effect on readability. There are, of course, a number of factors that aid in improving readability and detecting "excessive" comments. These items are presented in the next chapter.

3.1.3.6 Ease of Change

The sixth software attribute is ease of change. Ease of change is defined as the ease with which software accommodates enhancements and extensions [ARTJ86]. Some authors call it expandability and define it as the ability of software to provide for the expansion of data or program functions [ARTL85]. Other authors describe it as changeability which is the ease with which the logic of a program can be changed [GANC77].

Generally, once code is written, it does not remain static; changes, extensions, enhancements, and expansions become necessary with the passage of time. These modifications normally involve extending the program's capabilities. A programmer attempts to incorporate these new capabilities with the least amount of effort by expanding functions of the existing modules. Such an approach to altering code often results in a situation similar to that of loading a camel. That is, at first loading a camel is easy, then it becomes harder to find a place to strap on more cargo. Finally, the camel collapses from the sheer weight of its burden. Programmers should make an effort to avoid such situations. If the code is designed to facilitate "ease of change", a programmer will not find himself in the above situation because "ease of change" implies the capability to make changes in the code in a controlled manner, that is, the alteration of relevant portions or segments of the code, without affecting other code or logic in the thrust of coding for change.

A common technique for incorporating ease of change is through the use of a data dictionary and data base management system. Such techniques usually increase the ability to expand a program beyond its current data requirements. Data dictionaries provide common naming conventions and data descriptions, which are then combined to form the record structures required by all of the system's programs. Hence, if field "X" needs to increase to seven numeric positions, one simply changes the data dictionary, and generates the revised data structures in order to achieve a particular goal.

Other techniques useful in enhancing "ease of change" are the use of distinct functions, and "sandwiching". Using distinct, separate functions or routines assist in making changes because changes to independent routines do not force changes in other routines. Another technique, called "sandwiching", is first introduced by Parnas [PARD79], and addresses the conflict resulting when two programs want to call each other. In such situations, one of the programs is sliced into two parts in such a way that allows the programs to use each other. In the next chapter the reader is exposed to details of "sandwiching" and a few other techniques that facilitate "ease of change" in the code.

3.1.3.7 Traceability

The seventh software attribute is termed traceability. Traceability is defined as the software attribute that provides a link from requirements to the implemented program [ARTL85]. In other words, traceability is the ease in retracing the complete history of a software component from its current status to its design inception [ARTJ86].

For any software, it is desirable to trace the requirements to certain parts of the physical or logical system. The importance of traceability is realized when there is a need to find the origins of an error by tracing conditions back to the design and further to the re-

quirements definition. One can check whether there is an error in the definition or whether the designer overlooked something in the requirements. Although error conditions can arise under many circumstances, getting to the source of the error is always easier if the product possesses the attribute of traceability.

Methods that help programmers introduce traceability at the code level include:

- references to project documentation, and
- references to calling procedures.

The above items are discussed in more detail in the next chapter.

3.1.3.8 Visibility of Behavior

Software is characteristically less tangible than hardware. Code production, often used as a primary indicator of completeness, conveys nothing as a measure of the completeness of requirements, design and how well the code implements the design. Therefore, until one has reached the system testing phase the true development status does not become obvious. At this stage of development, however, problem correction is exceedingly expensive and time consuming. The root of this problem lies in the way a software system is produced. A software system can be produced by successive refinement of system requirements through high level system specification, and subsequently, into a more detailed representation of a system solution until the final solution (the checked out program representation) is reached. Software developed in the above fashion possesses the attribute of visibility of behavior, that is, the provision of review process for error checking [ARTJ86].

The lack of behavior visibility makes a system more complex and therefore more difficult to manage. Consequently, one spends more time fixing errors than it would be required if the software possesses the attribute of visibility of behavior.

The presence of visibility of behavior attribute is exhibited at the code level through items like certification levels, validation and verification (V & V) awareness, and V & V procedures. These items are to be discussed in the next chapter.

3.1.3.9 Early Error Detection

The ninth and the last software attribute is "early error detection". The attribute of early error detection is defined as an indication of faults in the requirement's specification and design prior to implementation [ARTJ86].

The importance of identifying "bugs" in specification and design prior to implementation is well-known. Generally, after the design stage, system functions are well-defined and are ready for implementation. If errors in the specification and design stages are allowed to propagate to the implementation level, they (the errors) can have disastrous effect on the product. Therefore, during specification and design it is essential that the subsystems are verified to be consistent and in concert with overall system objectives. Moreover, all discrepancies and mismatches should be corrected before they propagate into the next level.

The attribute of early error detection can be achieved through certification levels in the code, suggestions of V & V awareness, presence of V & V procedures, indication of V & V enforcement, and V & V result accessibility in the code and documentation. That

is, validation and verification should be recognized as an indispensable element and strictly enforced.

Thus far, the discussion in this chapter has revolved around the software engineering objectives, principles and attributes. The reader is exposed to the meanings of these terms, their usage, and their application. Additionally, the discussion also presents examples to provide the reader with an intuitive feel for these terms as well as how they are applied to yield desirable effect on the software. The next section attempts to establish a correspondence among these terms; that is, the relationships (linkages) among the objectives, principles and attributes are established.

3.2 Linkages

Recall that the primary focus of this research is to formalize a procedure for the evaluation of software development methodologies. The rationale for the evaluation procedure described in this research utilizes objectives, principles, and attributes, and is based on the philosophical argument that:

> A set of objectives can be defined that should be postulated within any software engineering methodology, from which certain principles are derived that characterize the process by which software is created. Adherence to a process governed by those principles should result in a product (programs and documentation) that possesses attributes considered desirable and beneficial [HENS85].

It is precisely the above argument that serves as the motivation for linking objectives to principles and principles to attributes. The thrust of this research is to establish concrete linkages (relationships) among software engineering objectives, principles and attributes, based on literature support, intuition and common sense. In other words, this research attempts to justify that software engineering objectives can be achieved by the application of software engineering principles. When these principles are applied one can predict certain product attributes that are induced by the application of the above chosen principles.

The above concept is illustrated by the following example. Consider the single objective, maintainability. To remind readers, maintainability is the ease with which corrections can be made to recognized inadequacies. This research claims (claims later substantiated through literature references) that maintainability is achieved through the application of the following software engineering principles: hierarchical decomposition, functional decomposition, information hiding, stepwise refinement, structured programming, and documentation. Following discussion presents a justification for establishing a correspondence between maintainability and each of these principles.

The first software engineering principle that assists in the achievement of maintainability is hierarchical decomposition. The use of hierarchical decomposition implies the partitioning of a system in smaller, manageable modules. Such a division allows multiple use of common designs and programs, and also makes it easier to incorporate changes, expansions and extensions to the system [WASA76].

The second software engineering principle that helps in achieving maintainability is functional decomposition. According to J. Hosier [HOSJ78], the divisionalization of a system based on functional units promotes maintainability because "functional

separateness" of the constituent modules facilitates easier changes, extensions, and expansions to the original system.

The third software engineering principle that promotes maintainability is information hiding. Dickover [DICM78] suggests that the encapsulation of major design decisions in information hiding helps enhance maintainability. Additionally, according to Pressman [PRER82], the use of information hiding as a design criteria provides better benefits when modifications are required during testing, and later during software maintenance.

The fourth software engineering principle that helps achieve maintainability is stepwise refinement. In support of a correspondence between stepwise refinement and maintainability, Gilbert [GILP83] says that stepwise refinement is a design strategy that seeks to implant ease of understanding and ease of maintenance into the program text.

The fifth software engineering principle to support maintainability is structured programming. Structured programming advocates the use of only three programming constructs, i.e., sequence, choice, and iteration. With the use of such few constructs, the resulting program is easily understandable because it contains fewer intricacies. Thus, programmers can quickly detect which portions of the program that require maintenance or modifications [GILP83].

Finally, the principle of documentation is also linked to maintainability. As Wasserman [WASA76] states, a good design document is not only essential for writing software, but also for the maintenance of the software over time.

The above discussion provides supportive arguments for establishing a linkage between maintainability and each of the following six software engineering principles: hierarchical decomposition, functional decomposition, information hiding, stepwise refinement,

structured programming, and documentation. It can therefore be concluded that if one seeks maintainability as an objective, the methodology must support a software development process that embraces the above six principles.

Table 1 on page 81 summarizes relationships between software engineering objectives and principles. Appendix B provides literature references for each of the linkages shown in Table 1.

To achieve a software engineering objective one must adhere to a software development process guided by fundamental software engineering principles. In turn, the application of these principles induces certain attributes in the product. To continue with the earlier example, consider the single software engineering principle functional decomposition. In functional decomposition, a system is partitioned along functional boundaries. The use of functional decomposition technique induces the following software attributes in the final product: reduced coupling, enhanced cohesion, reduced complexity, and ease of change.

The first software attribute affected in the code by the application of functional decomposition is coupling. Functional decomposition entails divisionalization of code into modules in a controlled manner. Such controlled partitioning reduces the association between modules, thereby reducing coupling [PRER82].

The second software attribute affected by functional decomposition is cohesion. In functional decomposition the dissection of the system is done on functional boundaries. Modules resulting from this division are expected to perform a single task so that little interaction exists with other modules in the program [PRER82]. Thus, functional decomposition enhances cohesion.

Table 1. Linkages Between Objectives and Principles

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

The third software attribute affected by functional decomposition is complexity. Functional decomposition is simply divide and conquer technique applied to programming. When a program is divided into two independent parts, complexity is reduced dramatically [BERG81b].

Lastly, the attribute, ease of change, is induced by application of functional decomposition. Functional decomposition enhances product flexibility; that is, it facilitates the making of drastic changes to one module without a need to change others [PARD72a].

The preceding discussion establishes a correspondence between functional decomposition and each of the following software engineering attributes: coupling, cohesion, complexity, and ease of change. With respect to the maintainability example, Figure 6 on page 82 illustrates the espoused linkages. Table 2 on page 83 provides a complete set of principle/attribute relationships. Literature references for each of the linkages shown in Table 2 on page 83 can be found in Appendix C. From an overall, global perspective, Figure 7 on page 85 summarizes the entire set of linkages among software engineering objectives, principles and attributes.

The author notes that it is not the intent of this thesis to present supportive arguments for each of the linkages shown in Figure 7 on page 85. An interested reader is advised to refer to linkage references in Appendices B, and C. Moreover, additional discussion on the evaluation procedure is found in ARTJ85, HENS85 and ARTJ86.

This author has significant contribution in validating the linkages among the objectives, principles, and attributes. All the objective/principle and principle/attribute likages were validated through literature references by the author. More detailed description of author's contribution is listed in Chapter 7.

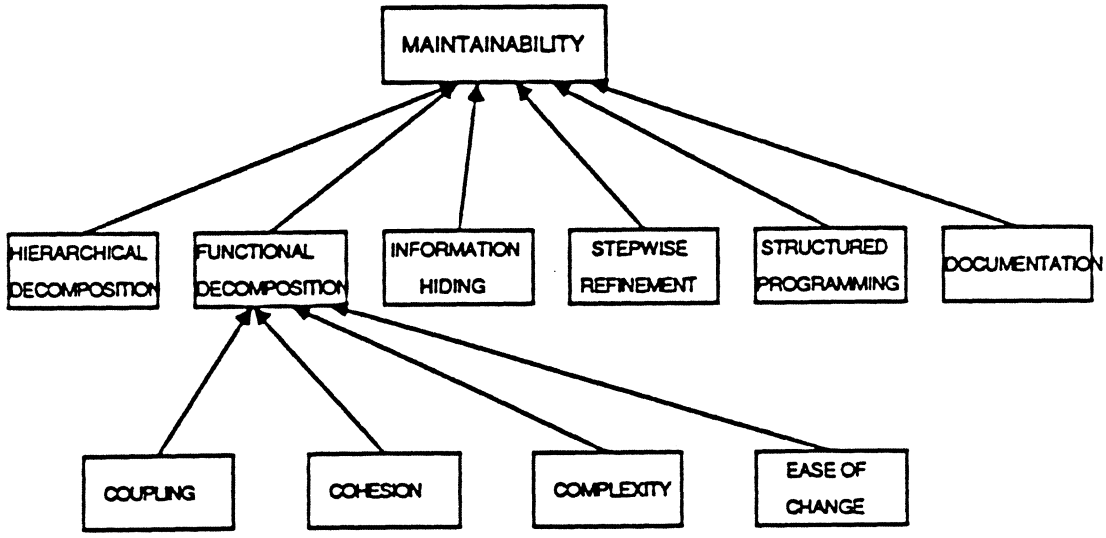


Figure 6. Linkages with respect to Maintainability

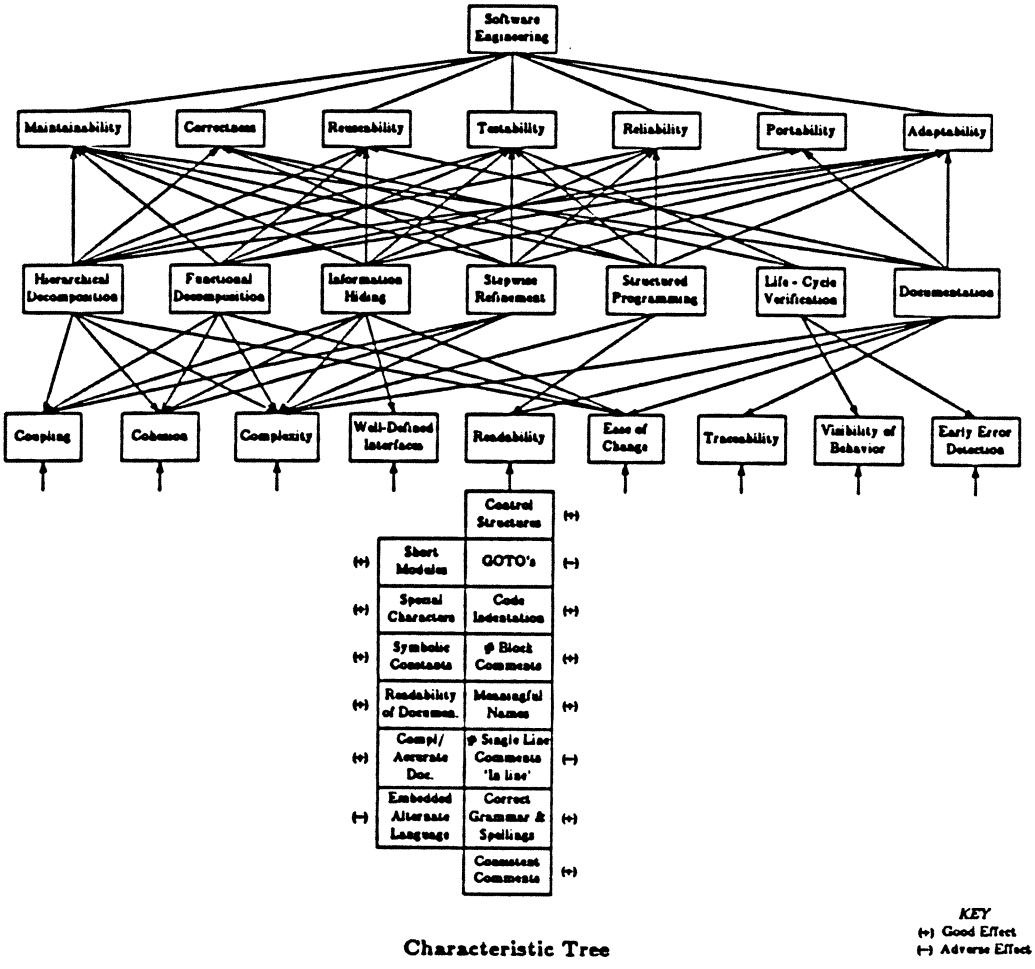
Table 2. Linkages Between Principles and Attributes

Principle	Effect on Attributes
Stepwise Refinement	Coupling/Cohesion enhanced Reduced Complexity
Hierarchical Decomposition	Ease of Change Coupling/Cohesion enhanced Reduced Complexity
Structured Programming	Enhanced Readability Reduced Complexity
Functional Decomposition	Ease of Change Coupling/Cohesion enhanced Reduced Complexity
Information Hiding	Extensible Software Coupling/Cohesion enhanced Reduced Complexity Well-defined Interface
Documentation	Enhanced Readability Reduced Complexity Traceability Ease of Change
Life Cycle Verification	Visibility of Behavior Early Error Detection

To summarize, this chapter has concentrated on presenting the motivation behind this research, explaining the background material, introducing the software engineering objectives, principles and attributes, and the linkages among them. The next issue to address is how these linkages are utilized in assessing the adequacy of a methodology. That is, given a methodology, how do the relationships described above enable one to determine if that methodology possesses the necessary qualities for attaining its stated goals and objectives.

3.3 Assessment

Having established linkages among the software engineering objectives, principles and attributes the next topic of interest is how the linkages can be utilized in assessing whether a methodology possesses the appropriate software engineering characteristics crucial to software development, and the extent to which they (the characteristics) are present in the product. An examination of the above will proceed in two phases. The first phase deals with "how to" assess the adequacy of the methodology while the second phase addresses "how well" the methodology succeeds in achieving its goals. Formally, the "how to" assessment is viewed as a top-down evaluation process, while the "how well" approach is considered a bottom-up process. Together, these two approaches (top-down and bottom-up) not only provide a basis for judging the adequacy of a software development methodology but also the extent to which the software product (resulting from the application of the methodology) conforms to the stated goals of the methodology. The top-down and bottom-up approaches are discussed below in detail.



Characteristic Tree

Figure 7. Characteristic Tree

3.3.1 The Top-Down Evaluation Process

As the name suggests, the top-down evaluation process begins at the highest level of characteristics (the objectives) in the characteristics tree (Figure 7 on page 85), i.e. identification of the objectives of the methodology. The evaluation process then moves to the next lower level in the software engineering hierarchy, and addresses the appropriateness of the software engineering principles. The last level visited in the top-down approach is the software attributes. This step-by-step examination of the objectives, principles and attributes adopted in the top-down evaluation process allows one to evaluate a methodology in terms of its goals, and the sufficiency and completeness of its supporting tool set for achieving those goals. The top-down evaluation process is reflected in the three major steps described below.

In the first step of the top-down evaluation process, two sets of objectives are identified. The first set of objectives is based on software needs and requirements. These objectives are normally reflected in the description of a desired software system. The second set is comprised of objectives adopted by the software development methodology. These are the stated objectives of the methodology. If the comparison of the two sets of objectives indicates a mismatch, then one should question whether the methodology being considered is appropriate. After all, one should not expect to achieve objectives that are not enumerated by the methodology.

The second step in the top-down evaluation is the investigation of the software development process that links objectives to principles. That is, given a stated set of methodological objectives, one can identify (using linkages between objectives and principles) which principles should be supported by the methodology. Three possible cases exist in such an identification. The first case is where the methodology is overprincipled,

that is, principles are present without corresponding objectives. In the second case, the methodology may be underprincipled, that is, the objectives exist without supporting principles. In the third case, the methodology is adequately supported by the principles. A software development methodology that exhibits either case one or case two characteristic also has a potential for creating unwanted problems.

The third step in the top-down evaluation process involves a formulation of the set of expected product attributes. This formulation is based on the fact that the software engineering principles define the process by which a software product is produced and induce a specific (desired) set of product attributes. Obviously, the expected set of product attributes should reflect those desired by the software engineer.

Thus, one can utilize the top-down assessment in order to establish the adequacy of a software development methodology. The inadequacy of a methodology may be reflected at any of the above three steps, and will have a detrimental effect on the overall adequacy of the methodology.

3.3.2 The Bottom-Up Evaluation Process

While the top-down approach brings to light the deficiencies in the software development methodology, the bottom-up approach enables one to determine the extent to which a software product conforms to the objectives of a software development methodology. In other words, the bottom-up evaluation process reveals product attributes and allows one to assess, to some extent, the effectiveness of the methodology; that is, the capability of producing desired results.

The top-down approach provides a set of attributes which is expected as a direct consequence of applying the methodology. However, it is possible that the set of expected attributes does not match those determined to be present in the product. Such an irregularity can be identified using the bottom-up approach described below.

The bottom-up evaluation process, as the name suggests, starts at the lowest level (the product level) in the software engineering hierarchy shown in the characteristics tree (Figure 7 on page 85). Like the top-down approach, the bottom-up evaluation process also relies on the existence of linkages among software engineering objectives, principles and attributes.

To begin with, the bottom-up evaluation process involves the identification of software attributes in the product resulting from the application of a software development methodology. This computed set of attributes is compared with the attributes derived (and expected) from the top-down evaluation process. If these two sets of attributes do not match, one can infer that there is an improper use of software engineering principles in developing the product. Several reasons may exist, however, as to why the supported principles may not be properly employed:

- the personnel involved in the development process are ill-trained, and/or

- the effectiveness of methodology is questionable.

Based on the evaluation procedure, one can examine the exploited software development process in order to pin-point possible reasons for the failure of attaining the expected attributes. That is, using the computed set of attributes and the linkages among principles and attributes one can compute the actual set of principles used in the software development process. This set of principles is then compared with the set of principles

needed to support the desired methodological objectives. If the two compared sets are in agreement then the methodology is expected to achieve its goals. On the other hand, if the two sets are not in harmony then there is a possibility that the software development process has fallen short of meeting desired methodological objectives. One must then question whether the fault lies with the personnel expertise or insufficient tool support for the enunciated principles.

In summary, the top-down and bottom-up evaluation processes together are not only descriptive in nature, but are also prescriptive. That is, they not only indicate potential problem areas but also indirectly suggest remedies for rectification.

3.4 Assessment Based on a Common Yardstick

In the previous section, the reader is presented with the software methodology assessment procedure wherein a given methodology is tested for fulfillment of its goals. The procedures used for this test are the top-down and bottom-up evaluation processes. Once a methodology is assessed for adequacy of its goals (using the top-down evaluation process), and its objectives, principles and attributes have been established, the next major issue is to measure these characteristics (objectives, principles and attributes) based on a common framework.

This research suggests that one appropriate measurement approach be based on an arbitrarily selected scale of 1 through 10, where a score of 1 is considered to be poor and the score of 10 is the best (or most desirable). A score of 5 is the mid-range and does not reflect the "goodness" or "badness" of a characteristic (objective, principle or attri-

bute) under consideration. The scores 6,7,8, or 9 reflect increasingly desirable qualities with respect to the measured quantity. Scores like 4,3,2, and 1, on the other hand, reflect increasingly undesirable qualities. This 1-10 approach provides a uniform basis for measuring software engineering characteristics inherent to a product, and at the same time, reflects the variability with which these characteristics are stressed during the software development process. The variability among software engineering objectives, principles and attributes can be reflected when they are measured on a common basis. In other words, the computation reflects the emphasis. Such a measurement enables one to compare the extent to which a characteristic (objective, principle or attribute) is present.

We note that this research does not deal with the question of what the ideal scores are for different characteristics nor how they should be achieved in a methodology. Such a problem falls in the software engineer's purview. In this section, we are presenting a method by which a characteristic's presence is measured on a common scale so that they can be compared with one another.

A secondary issue closely tied to the common yardstick concept is the use of weighted computations. In the current evaluation procedure determining the emphasis placed on an objective, principle or attribute starts with the product and utilizes a weighted computational process. The first items to which weights are associated are code properties. For example, the presence of a GOTO statement in the code indicates that the attribute readability is adversely affected. The question of "how adversely it is affected" is a controversial issue due to its subjective nature. For example, we feel that the presence of a GOTO statement is three times more harmful for readability than the presence of multiple exits from a loop. Although other researchers may perceive the need for different weight distributions, the point we wish to convey is that the evaluation procedure does

provide a mechanism for capturing the variability of influence among measured quantities. Hence, by using weights one can assess the extent to which each characteristic is present. The 10-point scale together with the weighted averages allow the scores for individual characteristic to remain in the range 1 through 10. These scores reflect the emphasis placed by the methodology on various characteristics. Also, when the scores are on a common scale comparison among the entities is justified.

To summarize, in this chapter the reader is exposed to the various software engineering objectives, principles and attributes, their meanings, and applications. This chapter also introduces the concept of linkages to tie the objectives, principles and attributes to one another, followed by a discussion on how the methodologies are evaluated for adequacy and completeness. Two approaches, top-down and bottom-up, are presented in support of the evaluation process. Finally, a procedure for the assessment of various characteristics based on a common yardstick is presented. Most of the discussion in the chapter is substantiated with examples.

In concert with the material presented in this chapter, the next major issue is the identification of various attributes in the code and documentation. For example, the code is likely to possess attributes like coupling, cohesion, complexity, readability and so forth. The question is how does one identify or detect the presence or absence of these attributes at the code level. The next chapter discusses what are the surface qualities that one needs to look for in the code to establish the presence of these attributes.

4.0 Product Properties

This thesis, in chapter 1, presents the motivation and need for developing a procedural approach to evaluating software development methodologies. Chapter 2 presents the background material which, after distinguishing between a method and a methodology, familiarizes the reader with several software development methodologies, and lays the foundation for the introduction of software engineering objectives, principles and attributes. Chapter 3 outlines in detail the various software engineering objectives, principles and attributes and the linkages among them. Chapter 3 also describes how the linkages are utilized to assess the adequacy and the effectiveness of a software development methodology using the top-down and bottom-up evaluation procedures, respectively. Finally, a discussion is given which focusses on a common yardstick for assessing the influence of properties on various attributes.

After having introduced the linkage approach and the evaluation procedure, the next major topic, and focus of this chapter, is the measurement of product attributes. This topic can be partitioned into two issues - the first issue involves identifying the existence of attributes while the second issue addresses the measurement of those attributes.

Issue one, the identification of code and documentation attributes, is a non-trivial task. Unfortunately, attributes are intangible objects. Hence, to establish their existence one needs first to address the following sub-items:

- the identification of code and documentation properties whose presence implies existence of certain attributes, and

- the definition of relationships among properties and attributes.

Two examples of code properties that are related to attributes are the number of routines and the average length of a routine. One can argue that the average length of a routine is related to the attribute complexity. The rationale being, the longer the routine the more complex the code is to understand.

Once a set of properties is established and the relationships among properties and attributes are defined, the second issue concentrates on measuring the extent to which an attribute is perceived present based on the assessed presence of related properties. To compute the presence of a property one needs to resort to the concept of metrics which is introduced later in this chapter.

4.1 Background and Motivation

One benefit of the evaluation procedure is that it allows one to assess the adequacy of a given software development methodology with respect to achieving its stated objectives. This assessment procedure is performed through a top-down evaluation process. Effectively, assessing the adequacy of a methodology means that methodological suffi-

ciency or completeness is determined with regard to its stated objectives, principles, and attributes.

Although it is recognized that having a well-defined, adequate methodology can produce software products exhibiting desirable characteristics, in the development environment there may be problems in achieving the goals of a methodology. In short, methodologies do fail in attaining their goals. The reason(s) for the failure of a methodology could be any one or a combination of the following:

- wrong design or an inadequate methodology, i.e., the goals of a methodology are not in harmony with those derived from needs and requirements analysis,

- improper application of the methodology, or

- untrained personnel.

Of these three causes or problems, the first problem can be detected by the top-down analysis which reveals the adequacy of the methodology. If the reason for the failure of a methodology does not lie within the methodology per se, then it lies with either an improper application of the methodology or untrained personnel. These two aspects are reflected in assessing the effectiveness of the methodology. The effectiveness of a methodology is directly mirrored in the resulting product and, as such, can be assessed by examining the product. Correspondingly, product examination based on properties assessment form a foundation for determining the effectiveness of a methodology.

4.2 *Attribute/Property Relationship*

In order to assess the extent to which an application of a methodology follows its intended use, one needs to start at the product level. At this stage, the problem narrows down to the evaluation of the code and code documentation. This situation can be translated as follows: Given the project software, project documentation, project requirement specifications and methodology specification, the question is - to what extent do the code and code documentation conform to the specified software development methodology?

One answer to this question can be based on an assessment of the presence or absence of product attributes, that is, the software and documentation characteristics that are desirable and beneficial (e.g., low coupling, high cohesion, low complexity, etc.). The primary issue then is identifying these attributes in the product. It can be seen that the attributes cannot be identified directly. That is, by simply observing the product one cannot identify the presence or absence of code (or documentation) attributes. The reason is that the attributes are intangible and often subjective qualities. For example, one can not establish the presence or absence of coupling or complexity or any other attribute with a mere look at the code or code documentation.

In determining the presence or absence of attributes, one has to search for certain specific qualities of the code and code documentation. For example, the use of global variables in the code indicates the presence of coupling. Similarly, the use of certification levels in code documentation indicates visibility of behavior. Surface qualities of the code and code documentation that are indicative of the presence or absence of an attribute

are called properties. The difference between properties and attributes is that properties are readily observable (or recognizable) while attributes are not.

In addition to identifying code properties one must also ascertain the relationship between a property and attribute as well as determine its effect on the attribute. That is, the presence or absence of a property can have a positive (favorable) or a negative (adverse) effect on the attribute. For example, the following properties influence cohesion (to remind the reader, one endeavors to enhance cohesion in the code). The type of effect (positive or negative) they have on cohesion is shown with each property:

- Use of switches as parameters (-),
- Use of control structures (+),
- Multiple entry points in a routine (-),
- Fan-in to a routine (+),
- Fan-out (-), and
- Modularization (+).

The first property (use of switches as parameters) has an adverse effect on cohesion. According to De Marco [DeMT78] the downward passing switch is probably the simplest test of poor cohesion. This is true because when a switch is passed downward the controlling module is not aware as to how and how many modules are going to be affected by this switch. On the contrary, when a switch is passed upward, the passing module knows exactly which (one) module will be affected.

The second property (use of control structures) has a favorable influence on cohesion [TROD81]. When a control structure is used to represent some complex logic, that piece of code is more like a unit with all relevant code encompassed in it. This homogeneity of code enhances cohesion.

The next property that affects cohesion is the use of multiple entry points in a routine. This property has an adverse effect on cohesion. If a routine has multiple entry points, the implication is that there are pieces of code performing multiple functions. In turn, this indicates reduced functional cohesiveness [PERE85].

The fourth property to influence cohesion is fan-in to a routine [TROD81]. Fan-in has a positive effect on cohesion. The rationale behind this type of influence is that the more fan-in to a routine the more specific and well-defined its function. This is precisely the goal of cohesion.

The next property listed is fan-out and has an adverse effect on cohesion. Multiple calls from a module indicate a tendency toward too much control within a single module [CARD86, STEW81]. The problem may be a missing design level. The function and control contained within the original module may need to be divided.

The last property mentioned above that influences cohesion is modularization. The main goal of modularizing a design is to divide the software into pieces that are functionally cohesive [TROD81]. This property, therefore, has a favorable influence on cohesion.

It should be mentioned that the collective effect (or influence) of all the properties related to an attribute imply the existence of that attribute. For example, the overall effect of the six properties (mentioned above) imply the existence and the extent to which cohesion is present in the code.

The reader is advised to refer to Appendix D for further information about the sets of properties related to various attributes. Appendix D addresses each attribute individually and indicates the set of properties that is related to each attribute.

The previous examples illustrate that the presence of properties can have either a positive or a negative effect on an attribute. Correspondingly, the absence of properties can also have a favorable or adverse effect on attributes. For example, absence of code indentation has an adverse effect on readability and complexity, i.e., the lack of code indentation reduces readability and increases complexity. The absence of "Go To" statements, on the other hand, helps improve readability and reduce coupling, both of which are favorable (positive) influences [DeMT78, DIJE68].

In addition to the above, properties can also have a positive effect on one attribute while the same property may have a negative influence on another attribute. For example, the use of structured data types as parameters has a favorable effect on the "well-defined interfaces" attribute but has an adverse effect on coupling. The structured data type (as a parameter) helps improve the well-defined interface because all the information (in a data structure) is passed as a single parameter which makes the interface simpler. On the other hand, it has an adverse effect on coupling because extraneous information is being passed to other modules [TROD81].

Thus far, in this chapter the reader is presented with the notion of properties, why they are needed, and how they can be related to attributes. The above discussion also describes how the presence or absence of a property can have a favorable or an adverse effect on attributes. A particular case, where one property has a positive effect on one attribute while having a negative effect on another, is also presented. The next section

distinguishes between code and documentation properties. Also, the concept of metrics for measuring properties is introduced in the discussion.

4.3 Types of properties

The previous discussion acquaints the reader with the definition of properties and how they are useful in determining the presence of an attribute. The following discussion is aimed at familiarizing the reader with the different types of properties, and a basis for their classification.

Product properties can be viewed as exhibiting either

- Logical, or
- Numerical characteristics.

The first group consists of properties whose assessment values reflect responses like "yes", "no", and "sometimes". Representative elements of "logical" properties are determined from questions like:

- Is the source code indented,
- Are symbolic constants used in the code, and
- Is there consistency in use of variable names in code and documentation?

Properties that elicit logical responses are often subjective in nature and can be difficult to verify. On the other hand, properties that exhibit numerical characteristics are much less subjective and are easy to verify. The responses to these properties are in numerical quantities. Some typical examples of "numerical" properties are:

- Number of routines in the software,
- Average number of lines of code (LOC) per routine,
- Number of global variables, and
- Total program length.

Independent of their logical or numerical qualities, product properties can be partitioned into two categories, viz;

- Code properties, and
- Documentaton properties.

The above distinction is based on the criteria of whether a property is found in the source code (executable program instructions) or whether it is represented in documentation. An example of a code property might be the number of parameters passed by a routine while an example of a documentation property might be the use of block comments. Documentation properties can be further sub-divided into:

- Comment (code documentation) properties, and
- System documentation properties.

The first subcategory, comments or code documentation properties, are internal to the code. They are reflected in comments or explanations written in the code. Such documentation includes: single line comments, block comments, and in-line comments. These comments also include program header comments and module/routine header comments. The second subcategory of documentation properties are found in system documentation which includes overall project document, design document, testing/maintenance document, users' manuals and all other system documents. This form of documentation is external to the code. The following discussion first presents properties associated with the code followed by the documentation properties.

As mentioned earlier, code properties are the ones present in the source code. Some typical examples of the code properties are:

- Use of structured data types (SDTs) as parameters,
- Use of dynamic structures,
- Use of recursive code, and
- Multiple exit points from a routine.

The first property mentioned above, the use of structured data types as parameters, can be related to the attributes of coupling and well-defined interface. It should be noted that this property has opposite effects on coupling and well-defined interface. That is, the use of structured data types has a positive (favorable) effect on well-defined interface but a negative (undesirable) effect on coupling. The rationale is that when a data structure is passed as a parameter the interface is more compact, concise and therefore well-defined. On the other hand, an SDT induces extra coupling as a result of passing extraneous data

items which are not necessary for computation, and thereby, increasing coupling [LOHJ84, TROD81].

The second code property listed above is the use of dynamic structures. This property has a favorable influence on ease of change while having an undesirable effect on complexity. The use of dynamic structures enhances ease of change because they (dynamic structures) alleviate problems of insertion and deletion of components [DALN83]. On the other hand, dynamic structures change in size during the execution of a program, thereby, increasing complexity [GROP83, WEIL74].

The third property enlisted is the use of recursive code. The recursive nature of code increases complexity because in the case of an error it is difficult to debug the code easily [BOEB84].

The fourth property, multiple exits from a routine, has an adverse effect on readability, complexity and cohesion. The reason for a negative effect on readability and complexity is that the multiple exits destroy the structuredness of a program. By definition, a module with a single exit has an effective complexity of zero. Also, readability is adversely affected because of the multiple exits [ARTJ86]. Multiple exits from routines also adversely affect cohesion. According to Perry [PERE85], program modules must remain single purpose, with single entry and exit points. A program cannot jump into the middle of a module nor can it leave the module except via its sole exit. Such modules are called cohesive modules.

The reader is advised to refer to Appendix E for a detail list of properties, which attribute(s) they affect, how those attributes are influenced by them. Also, each relationship between a property and attribute(s) is substantiated with literature references.

In Appendix E these properties are referred as factors that exhibit presence or absence of an attribute.

The second category of properties is documentation related. These properties appear both within the code and outside of code. Documentation inside the code refers to various types of comments which include block comments, header comments, single line comments, and in-line comments. Similar to the code properties, a documentation property can also be related to one or more attributes. Properties related to documentation in code include:

- Use of comments,
- Use of "excessive" single line comments,
- Use of block comments,
- Use of comments consistent with code functions, and
- Use of comments referencing project documentation.

The first code documentation property is the use of comments. Comments favorably influence readability and complexity. According to Weissman [WEIL74], comments can increase the ability to understand and maintain programs. They also aid in enhancing clarity and precision of the code [CONR76].

The second documentation property listed above is the use of "excessive" single line comments. Comments, if excessive in number, adversely affect complexity and readability. The reason for the adverse effect is that too many comments can obscure the executable code. Also, there is no need to restate what is already obvious [ARTL85].

The third code documentation property is the use of block comments. According to McCracken [McCD76], the use of block comments makes a program more understandable, thereby, favorably affecting the readability and the complexity of the code [McCD76].

The fourth code documentation property listed is the use of comments consistent with code functions. Such comments have a favorable effect on both readability and complexity. According to Kernighan and Ritchie [KERB78], the trouble with comments that do not accurately reflect the code is that the code cannot be examined critically - this affects complexity and readability.

The last documentation property in the above list is the use of comments referencing project documentation. Such references in code enhance traceability because traceability implies overall relationships to other routines which is reflected via comments referencing code documentation.

The above discussion presents the code properties and the code documentation properties, both of which are present within the code. The reader is again advised to refer to Appendix E for detailed listing of properties and their relationships to attributes. These relationships are substantiated through literary references.

After presenting the properties exhibited in the documentation internal to the code (code documentation), one needs to discuss the properties intrinsic to the documentation external to the code. Such documentation includes users' manuals, design documentation, product documentation, operations documentation and so forth. A detailed discussion of external documentation is presented in Chapter 3. These external documents are collectively referred to as system documentation in the following discussion.

As mentioned above, the system documentation exhibits certain documentation properties. The properties whose presence can be detected in system documentation are:

- Completeness,
- Accuracy, and
- Readability.

The first system documentation property is completeness. Completeness of documentation refers to the extent to which the information presented in the documentation provides full account of the design, implementation, and operational (including maintenance) features of the system. That is, the documentation should provide a detailed record/report of the development of requirements, preliminary and detailed designs, implementation of the final design and finally maintenance of the system.

The second system documentation property is accuracy. Documentation accuracy indicates consistency among documentation elements and between documentation and the final software product. If the documentation does not reflect the exact behavior of the product the documentation will be termed inaccurate.

The third property of documentation external to the code is that of readability. The readability of documentation is the ease with which one can understand and comprehend the contents of documentation. Readability normally depends the appropriateness of information, consistency of formats, cross-references and various other factors.

The above three system documentation properties can be detected by employing an evaluation technique that uses a questionnaire, which when answered, indicates not only

the presence or absence of one of the three properties but also the extent (on the 10-point scale) to which the properties are present.

An assessment of these properties (completeness, accuracy and readability) is accomplished through the evolution of factors that are related to the properties. For example, accuracy and completeness can be assessed by identifying incomplete or missing information, traceability between requirements and design, and between design and implementation, revision numbers, inclusion of memory size and so forth. The factors that assess completeness and accuracy are currently found to be the same. We have therefore combined these two properties into one which we call completeness/accuracy of the system documentation.

Similarly, readability can be assessed by observing additional factors in (or qualities of) the documentation. The factors that influence readability include: traceability, usage of presentation tools and change bars, consistency of format between documents for table of contents, lists of illustrations, and sections of paragraphs. Appendix F shows the factors that enhance accuracy/completeness and readability. Appendix F also presents a sample questionnaire that enables one to determine the presence of factors that are related to the documentation properties.

Once the presence of attributes in the software product (code and documentation) is established via properties, the next logical concern is the measurement of these properties. The measurement technique should be able to measure the degree of presence of the attributes in the software product. Such a measurement of properties is accomplished through the use of metrics.

4.4 *Metrics*

The discussion in this chapter, thus far, has suggested how the presence of attributes in the product can be determined. To remind the reader, at the lowest level in the software characteristics tree (see Figure 7 on page 85) are the code attributes. To assess the extent to which the methodology has been effectively applied, one needs to measure the degree to which attributes are present or absent in the product.

According to Gilb [GILT77], there exist two important stages in software attribute measurement. They are:

- agreement on the measuring concept, and

- finding a sufficiently accurate tool for measuring the attribute in a real system.

Effectively, this amounts to identifying or defining indices of merit that can support quantitative comparisons and evaluations of software. To carry out these comparisons and evaluations, parameters are needed. They are provided by the metrics.

The term "metric" is defined as the measure of the extent or degree to which a product (this research concentrates on code and documentation) exhibits a certain (quality) characteristic [BOEB78]. Some metrics provide very important insights and decision criteria for both the developer and potential users of a software product; others provide information which is interesting, perhaps indicative of the potential problems. The judgement as to its potential benefit is, of course, dependent on the users for which the evaluator is assessing the product.

A simple example of a metric can be as follows. Program development techniques often prescribe that the length of routines (modules, procedures, functions) should be manageable in terms of the lines of code (LOC). A number of arguments supporting this idea are present in the literature. Constantine [YOUE79] argues that the executable LOC in a routine should be restricted to 30. Beyond this limit the understandability, readability, and complexity are adversely affected. The number 30 emerges from the fact that a standard output page can accommodate about 30 LOC and a human is said to comfortably understand that much code at one time. This value can be used for comparison with the average routine length in the code. If the average LOC in a routine is over 30 then one can conclude that the software attributes readability and complexity are adversely affected. On the other hand, a shorter routine (LOC < 30) should enhance both complexity and readability of the code.

Now, consider another example where one needs to compute a metric to assess the degree of presence of an attribute, e.g. the software attribute of coupling. The reader is aware that coupling depends on the amount of communication between routines. Earlier discussion has established that one factor influencing coupling is the number of global variables referenced.

As discussed in the previous chapter, the computation of the following metric is done in such a way that it results in a score on the 10-point scale. A metric to compute the effect of global variables referenced on coupling is as follows:

$$x = \frac{\text{(\# Global Variables Referenced)}}{\text{(\# Global + \# Local Variables Referenced)}}$$

$$\text{Scale} = 10 - 9x$$

One can observe from the above computation that in the absence of global variables the metric would yield an ideal score of 10. As the number of global variables referenced increases (relative to the number of local variables referenced), the value of the metric would drop. Use of local variables in a routine would reduce the coupling caused by global variables to a certain extent. The worst case scenario (i.e., high coupling) occurs when there are no local variables in a routine. In other words, when the entire communication between routines is through global variables, the degree of coupling is maximum and the reflected metric value is lowest, i.e. a "1".

The use of a 10-point scale is suggested for metrics in Chapter 3. One can easily assess the "goodness" (positive effect) or "badness" (negative effect) of the attribute by simply looking at the number on the 10-point scale. As discussed earlier, a score of over 5 suggests a good effect (e.g., low coupling, high cohesion etc.) of the attributes. On the other hand, a score of less than 5 indicates a bad effect (e.g., high coupling, low cohesion etc.) of the attribute. A score of 5 does not provide any information.

One must be aware that a property which influences more than one attribute may have different metrics for each of the affected attributes. For example, use of structured data types (SDTs) affects the attributes of coupling and well-defined interface. Use of SDTs introduce extraneous coupling between routines. On the other hand, use of SDTs makes interface more compact and concise. Because of these variable effects one needs to compute the scale differently in the above two cases. Appendix G presents several metrics used to evaluate two Navy software development methodologies (These methodologies are explained in the next chapter).

Gilb [GILT77] suggests that even where there is a stable metric available for some software characteristic, the measuring tools will tend to be varied, tailor-made to fit the

system, and constantly changing as the years pass. This rate of change in the measuring instrument, which can be observed in all scientific, engineering, and business applications is probably attributable to the following factors:

- the need for greater accuracy for measurement,
- the need for more reliable measuring,
- the need for more economic measuring, and
- the invention or development of instruments which satisfy the above needs.

To summarize, a metric allows one to measure the presence of an attribute and also the extent of its presence. One must be cautioned not to be biased when designing a metric. A biased metric design will not yield a correct picture of the extent to which an attribute is present.

In summary, this chapter presents the notion of product properties. To start, the motivation for the use of properties is presented. The motivation is followed by the relationships between attributes and properties. A number of examples are cited to justify these relationships. Later sections then present a more detailed discussion on the two types of properties; viz.: code properties and documentation properties. Finally, the last section introduces the concept of metrics, their purpose, and their use.

Currently, this thesis has presented the reader with a description of software engineering objectives, principles and attributes, the evaluation procedure and how the evaluation is achieved through the use of properties and metrics. The next chapter deals with the ap-

plication of the evaluation procedure to two (2) software development methodologies and a discussion of the results.

5.0 Application and Ramifications

The first four chapters of this thesis have presented the reader with the motivation behind the procedural approach to evaluating software development methodologies, the background material, the evaluation procedure itself and the requisite measurement techniques that are instrumental in the evaluation process. To remind the reader, the main purpose of this research is to evolve a procedure that enables one to evaluate and compare software development methodologies. As mentioned earlier, the evaluation procedure is aimed at identifying both the strengths and the weaknesses of a software development methodology. In addition to pointing out methodological weaknesses, the evaluation procedure can also suggest remedial actions.

5.1 Application of Evaluation procedure

With the above concepts in mind, the evaluation procedure has been applied to two Navy methodologies. These methodologies are referred to as methodology A and meth-

odology B (and Systems A and B, respectively) in this chapter. The following material, extracted and summarized from [ARTJ85] and [NANR85], presents a brief introduction of the two methodologies.

Methodology A

The goal of methodology A is to control the design process for a complex of operational computer programs so that the resulting product is reliable, and correct. The program development process adopted by methodology A is a top-down process by which system requirements are allocated into successively more detailed functional performance requirements. These performance requirements are then mapped into top level software design specifications which, in turn, are allocated to detailed modules and data base designs.

The basic code unit in system A is called a procedure. A typical procedure performs some small unit function and has only one entry point and one exit point. Procedures are grouped into units called modules. Data is structured into messages. A message is defined to be a formatted unit of data exchange between modules.

Methodology B

The goal of methodology B, enunciated in the documentation, is to achieve product maintainability, adaptability, and reliability. The review also reveals that the principles of hierarchical decomposition, functional decomposition, life cycle verification, and structured programming are espoused by the methodology. This methodology includes a software production and maintenance procedure with associated production support tools.

In system B, the smallest basic unit of code is called a task. User programs are represented by task set chains (TSC) which are composed of computer program tasks. The interface between system control and user programs consists of system control/task interfaces.

Drawing from the overview above, later discussion in this chapter presents a detailed analysis of the two methodologies stemming from an application of the evaluation procedure. The following section presents the sets of data used in both the top-down and bottom-up processes of that evaluation.

5.1.1 Sets of Data Used

The evaluation of a methodology requires information describing the methodology itself, as well as information derived from the application of the methodology in a project. Methodology description and project requirements usually provide information regarding the guidelines, conventions and/or standards to be applied in developing the software product. Often, objectives and principles are identified in these instructions.

Project documentation, source code, and code documentation are all sources of information related to principles and attributes. Within the program performance specifications and interface design specifications, information is often found regarding the degree to which a methodology is understood and applied in a particular project. The surface properties of program design specifications and the source code provide the input to metrics for assessing code properties - either in terms of objective elements or subjective opinion.

The following documentation items were studied in order to evaluate methodologies A and B.

- Four software development methodology documents for

    1. identifying the pronounced software engineering objectives, principles, and attributes, and

    2. assessing the effectiveness of each methodology through the objective/principle/attribute linkages defined by the evaluation procedure, and

- Eight software system documentation items 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 [ARTJ86, ARTJ87].

The following section describes the application of the evaluation procedure to methodologies A and B. It involves the application of top-down and bottom-up approaches to both the methodologies. The following presentation is a re-statement and a summary of the discussion found in [ARTJ86].

5.2 *Analyzing the Methodology and the Product*

To remind the reader, the evaluation process consists of two steps:

- a top-down analysis of the methodology, and
- a bottom-up examination of the product.

The initial step in the evaluation process is to perform a "top-down" analysis of methodologies A and B. For each methodology, this top-down analysis reveals the set of pronounced software engineering objectives, principles, and attributes. Because both methodologies appear to be in the process of evolving, a clear statement of their respective methodological objectives is lacking.

Nonetheless, the documentation of methodology A appears 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. Table 3 on page 118 illustrates the pronounced software engineering objectives, principles, and attributes [ARTJ86]. Using metric values and properties, a corresponding "bottom-up" examination of product A provides some interesting results. The Kiviat charts displayed in Figure 8 on page 119 illustrate the extent to which each attribute is assessed as present in the product. The reader should note that the complexity attribute attains the rating 8.0 on the 10-point scale. The high rating of 8.0 indi-

cates that the complexity is reduced to a considerable extent. It is 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 8 on page 119(b) 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 8 on page 119(c) depicts the results of emphasizing these principles in the software development process. In particular, reliability (6.7) is rated as the major software development objective. Although correctness is also stressed by methodology A, attaining correctness necessitates life-cycle verification; this principle is neither emphasized by methodology A, nor evident in the software product. As illustrated by Figure 8 on page 119 other objectives and principles are given some emphasis during the software development process for system A. One may deduce that because the objectives and principles are not explicitly stressed in methodology A, the associated product suffers [ARTJ86].

For methodology B, the top-down analysis reveals that the objectives stressed in the corresponding documentation are maintainability, adaptability, and reliability. The principles emphasized are structured programming and documentation. 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.

Table 3. **Pronounced Objectives, Principles and Attributes**

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

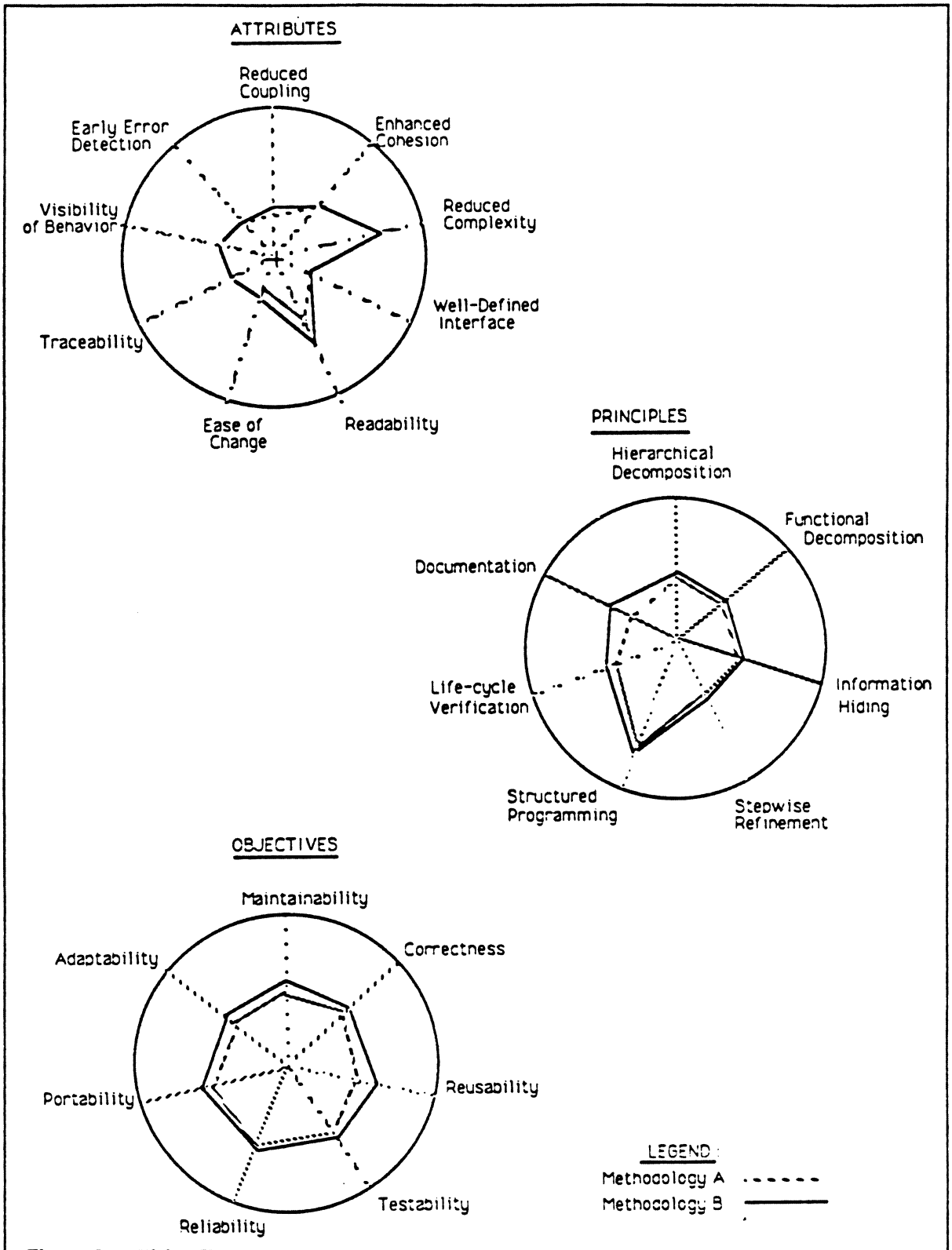


Figure 8. Kiviat Charts

Subsequent analysis of product B and a "bottom-up" propagation of the results through the linkages defined by the evaluation procedure reveals that structured programming is the most prominent principle (8.3), closely followed by documentation (7.0). Moreover, the evaluation also indicates that the implicitly assumed principles of methodology B are highly utilized - stepwise refinement, hierarchical decomposition, functional decomposition, and information hiding rate 6.9, 6.7, 6.7 and 6.3 respectively. Finally, the results imply that during the development of product B the objectives of maintainability, adaptability, and reliability are most emphasized. The above assessments are illustrated in Figure 8 on page 119.

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 assess the software engineering objectives, principles and attributes espoused by methodologies A and B. Of particular significance is the fact 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. An interested reader is advised to refer to NANR85 for a detailed evaluation of the two methodologies.

Thus far, the reader is presented with an application of the evaluation procedure and the discussion of its results. The Kiviat graphs illustrate, in comparative terms, the extent to which the objectives, principles and attributes are achieved in each methodology. One

question that commonly intrigues everyone, however, is why performance or efficiency is not considered a software engineering objective. The following discussion explains the omission of performance as a software engineering objective.

5.3 *Why Performance Is Not An Objective?*

Because one strives to achieve a high degree of efficiency or performance in a system, it is logical that one might assume performance as a software engineering objective. Therefore, the extension of performance as an objective of software development or a methodology is assumed reasonable. However, performance is not an objective - it is, in fact, a constraint. This section presents the arguments as to why performance is a constraint rather than an objective.

5.3.1 Linear Programming Concept

To understand the rationale behind the statement that performance is a constraint rather than an objective, one needs to first understand several concepts basic to linear programming (commonly known as LP) [GREM78, JOHR76]. LP is one of the most common and useful tools used in the operations research area for finding solutions to complex situations and problems encountered in many different environments. Some typical environments where LP is applied are production systems, transportation problems and so forth.

LP refers to a technique for the formulation and solution of problems in which some linear function of two or more variables is to be optimized subject to a set of linear constraints. The linear function to be optimized is refered to as the objective function. Optimization can refer to either the maximization or minimization of the objective function. For example, the objective may be to maximize a profit function or to minimize a cost function. Constraints are simply algebraic statements which, for maximization problems generally specify the limited availability of resources - time, labor, material, money, etc. - while for minimization problems they usually establish minimum quality or composition requirements. Let us now apply the LP concept to the software engineering objectives.

5.3.2 Performance : A Constraint

In Chapter 3 we list a number of software engineering objectives, viz; maintainability, correctness, portability, adaptability, testability, reliability and reusability. A methodology may stress all of these objectives equally or perhaps emphasize only on a few. For example, one may aim only for maintainability, portability and correctness. In such a case, the objective will be: To optimize M + P + C where variables M, P, and C respectively stand for maintainability, portability, and correctness. Another methodology may be aimed at achieving maintainability, portability, adaptability and reusability, where adaptability and portability are considered twice as important as maintainability and reusability. In such a case, the objective function will be to optimize M + 2P + 2A + R where variables A and R stand for adaptability and reusability respectively. The above objective functions are usually optimized with respect to some constraint like a given performance measure.

Performance (or efficiency) is described in a variety of ways by various authors. A few sample definitions of performance are:

- the measure of execution behavior of program (execution speed, storage speed) [MYEG75],

- the execution time, storage space, number of instructions, processing time [KOSD74],

- the effectiveness with which resources of the host system are utilized toward meeting the objective of the software system [DENJ73], and

- the amount of computing resources and code required by a program to perform a function [McCJ77].

To generalize the above, performance (or efficiency) deals with the utilization of a resource. One would normally endeavor to achieve a high degree of efficiency. It must be pointed out, however, that the achievement of high performance is not without side effects. That is, high performance has an adverse effect on the following software engineering objectives: maintainability, testability, adaptability, portability and reusability. Performance does not have an influence on correctness and reliability [McCJ80]. Figure 9 on page 126 [McCJ80] depicts the relationships between various software quality factors. A close examination of those relationships substantiate why performance is considered a constraint on the above objectives. That is, as performance goes up the achievement of objectives goes down!

Justifications for the adverse effect of high efficiency on the objectives of maintainability, testability, adaptability, portability and reusability are given as follows:

- Performance has an adverse effect on maintainability. Optimized code, incorporating intricate coding techniques and direct code, always provides better efficiency but poses problems for the maintenance of the system. From the other perspective, modularity and well documented high level code tend to improve maintainability but usually increases overhead and results in less efficient operation.

- Testability is also adversely affected by performance. The reason for this adverse effect is that the complex coding techniques used to gain performance often hinder the programmer throughout system testing. Moreover, in the case of an error the programmer often finds it difficult to trace the error because of the complex code.

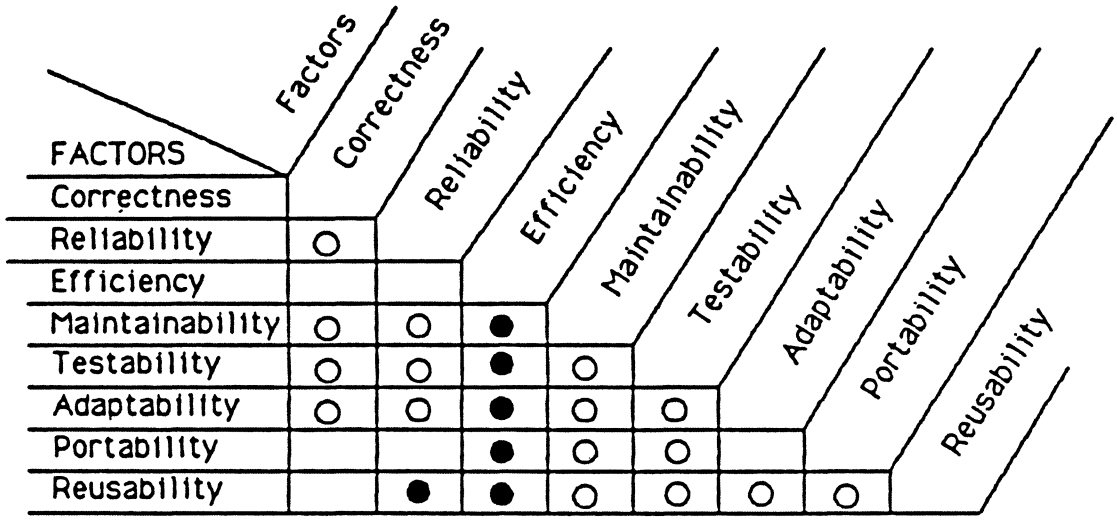
- The use of direct code, optimized system software and utilities improve the performance of the system. This has, however, a detrimental effect on portability because direct code and special utilities cannot be easily installed on other systems. Hence, portability of the original system is reduced.

- Adaptability is usually attained through generality. The generality required to achieve an adaptable system increases overhead and, correspondingly, decreases the efficiency of the system.

- Performance also has an adverse effect on reusability. Reusability also relies on generality of application which helps expand the usefulness of a function beyond the existing module. As mentioned above, however, generality gives rise to extra overhead which reduces the efficiency of the system.

In view of the above reasoning one can naturally perceive why performance is considered a constraint rather than an objective. If one wishes to achieve certain objectives, he must

compromise levels of performance. That is, maximizing a function of objectives is constrained by the level of performance one strives to achieve.



LEGEND

If a high degree of quality is present for factor, what degree of quality is expected for the other:

○ = High ● = Low

Blank = No relationship or application dependent

Figure 9. Relationship between Software Quality Factors

6.0 Software Engineering versus Systems Engineering

After exploring the domain of software engineering let us now examine the systems engineering area and investigate the similarities and differences between the two disciplines. The science of software engineering is aimed at creating software that is economical, correct, reliable and maintainable. Similarly, systems engineering endeavors to create systems that have several desirable characteristics. A few coveted characteristics pertaining to systems engineering are efficiency (or performance), reliability, testability and so forth.

6.1 Systems Engineering Life Cycle

A survey of literature [CUTM83, CUYR84, HIRS84, KATR84, RADJ84, SMIC83] reveals a significant commonalty between the software engineering and systems engineering disciplines. Systems engineering also employs a life cycle that is similar to the

software life cycle. Hirschhorn and Davis [HIRS84] argue that the systems engineering development life cycle is conceptually similar to the software life cycle presented in Chapter 3.

The following discussion first presents the phases of a systems engineering life cycle. As each phase is presented, similarities and correspondence between the software and the systems life cycles phases are highlighted. Rader [RADJ84] outlines a systems engineering life cycle which he calls digital life cycle. This life cycle includes the following steps:

- Requirements Definition,
- Architectural Design,
- Logic Design
- Detail Design,
- Construction,
- Testing and Integration, and
- Maintenance.

The first step of systems engineering life cycle is requirements definition. In this step, the complete description of the features of the system from a pure black-box perspective is presented. The corresponding step in the software life cycle is requirements specification.

The second step in the systems engineering life cycle is architectural design. Architectural design involves organization and division of system functions across computational modules. This step corresponds to the high-level design phase of the software life cycle.

Logic design and detailed design are the next two steps in systems life cycle. Together, these two steps support the detailed design aspect of the life cycle, wherein the abstract high-level design in the previous step is refined to more concrete ideas. The detailed design step of software life cycle parallels these steps.

The next step in systems engineering life cycle is construction. In this step the detailed design is implemented to create physical entities which can be networks, circuits and so forth. From the software aspect, construction resembles the coding and implementation.

The fifth step listed in the systems engineering life cycle consists of two main activities; testing and integration. Where testing involves verifying that the implemented hardware system fulfils system specifications while validating its performance, integration ensures that all the system components mesh together properly. Software testing is the corresponding step in the software life cycle.

System operation and maintenance constitute the last step in the systems engineering life cycle. In this step, the hardware system is put into operation. The maintenance aspect involves the incorporation of necessary extensions, expansions and enhancements. The corresponding step in the software life cycle is known as maintenance.

The above discussion of the systems engineering life cycle demonstrates an analogy between the two development cycles. The analogy is as follows: the requirements levels of both disciplines define the external behavior of the system under development and are thus equivalent; the architectural design of systems engineering life cycle and the high-

level design of software each defines the structure and architecture of the system; the logic and detail design levels of systems engineering life cycle correspond to software's detailed design in that both define the logical details of components; and finally, the construction phase of systems engineering life cycle defines the two-dimensional geometric representation of the hardware and which corresponds to the one-dimensional coding phase of software.

6.2 *Systems Engineering Objectives, Principles and Attributes*

Realizing the similarities between software and systems development life cycles one might ask - do "corresponding" objectives, principles and attributes exist on the systems engineering side? Once again a survey of the above literature reveals a number of systems engineering objectives, principles and attributes. Smith [SMIC83] and Cutler [CUTM83] outline the design objectives in hardware development and the principles employed to achieve those objectives. The authors have partitioned the systems engineering objectives in two categories; the objectives pertaining to design correctness and those that help improve design quality. Each objective is listed below according to its respective category.

- Design Correctness
    - Feasibility - ability to develop hardware system within available resources,

- Equivalency - the relationship between the conceptual design and the physical representation,

- Consistency - the extent to which the contents are traceable to the requirements, and lastly,

- Performance - the effectiveness with which resources of the host system are utilized toward meeting the objective of the hardware system.

- Design Quality

  - Testability - the ability to assess how the hardware conforms with the requirements,

  - Reliability - the extent to which hardware is expected to perform its intended function,

  - Modifiability - ability to make changes to a hardware system in a controlled manner,

  - Efficiency - the extent to which the system is adequately economical in its consumption of all resources,

  - Understandibility - the extent to which the purpose of the hardware product is clear to the evaluator,

  - Flexibility - the ease with which hardware can accommodate to change,

- Implementability - the ease with the hardware design is transformed into a physical entity, and lastly,

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

One can observe that the objectives listed above are similar (with the exception of performance) to those in software engineering. The only difference is that they apply to hardware products. For example, testability in relation to systems engineering is the ability to assess how the developed hardware conforms with the requirements. Reliability in hardware terms is the error-free use of hardware performance over time. Reusability is the use of a system or system components in other applications. Finally, implementability is the ease with which the conceptual design can be realized using physical elements.

To achieve the above objectives, Smith, Cutler and others [CUTM83, HIRS84, SMIC83] observe that the application of following development principles can assist in attainment of the objectives. They advocate use of following development principles:

- Hierarchical Decomposition - a method of designing a system by breaking it down into its components through a series of top-down refinements,

- Functional Decomposition - components are partitioned along functional boundaries,

- Abstraction - the ability to see the whole problem, ignoring irrelevant details,

- Structured Design - using a restricted set of design constructs,

- Hiding - insulating the internal details of component behavior,

- Confirmability - verification of requirements throughout the design, development and maintenance phases of the life cycle,

- Localization - bringing related items together in close physical entity, and lastly,

- Uniformity - the degree to which a design uses consistent notation and entities.

All of the above principles (with the exception of localization and uniformity) are deployed in the development of software. The following discussion presents how some of these principles are applied in hardware environment.

In hierarchical decomposition or top-down development, design proceeds through stages. At the highest level a behavioral description of the system is given that is further decomposed into description description and then translated into a structural description. The algorithmic description is then used to further refine the description into a structural description that includes the logic and topological characterization [CUYR84].

Another approach used in developing hardware systems is functional decomposition. In such a case the system is decomposed into various functional units. For example, the system may be divided into clock design and layout concerns [SMIC83]. We might add that both hierarchical and functional decompositions are basically forms of abstraction.

A third approach commonly employed in hardware development is called structured design [HIRS84]. This approach is similar to the structured programming principle of

software engineering, in that design constraints are imposed upon the designers to help solve the problem. Only certain design constructs are permissible.

The next systems engineering principle is hiding. The algorithms specified in systems engineering are realized through chip layout and fabrication. During the developmental process the higher levels of design tend to hide their details from lower level entities. Hence decisions at the lower levels are made with only minimal information of higher level details [CUYR84].

The principle of confirmability is similar to life cycle verification in software engineering. This principle assures that the structuring of the hardware system is done according to requirements.

When these principles are used, the resulting hardware product demonstrates a number of attributes. Some of the more commonly found attributes are connectivity, and complexity.

Connectivity, for example, is defined as the length of path between the components or elements of a circuit. Connectivity in systems engineering is analogous to the attribute coupling in software engineering. As the achievement of low coupling is desirable in software engineering, one endeavors to attain low connectivity in systems engineering.

Similar to the relationships among software engineering objectives, principles and attributes, there exist similar linkages among systems engineering objectives, principles, and attributes. That is, like the software engineering tree there exists a corresponding systems engineering tree that depicts relationships among systems engineering objectives, principles and attributes. Because it is not the intent of this research to discuss systems engineering discipline in detail, interested readers are advised to refer to [CUTM83,

CUYR84, HIRS84, SMIC83] for additional information. The point that we wish to emphasize, however, is that apart from the systems and software linkages mentioned above, there is also an interplay between the two relationship trees. That is, a correspondence or relationship exists between the two trees. In particular, a design decision in hardware environment can force constraints in the software domain, and vice versa. As discussed in Chapter 5, enforcing degrees of system performance illustrates the impact of system/software engineering design and development decisions.

To explicate the above relationship, let us consider the following scenario. Suppose one is designing a multiprocessor network with an objective of implementability. Obviously, the goal in this case is to design the system such that its implementation is easier. One of the systems design principles that can aid in achieving implementability is decomposition, that is, the hardware is designed in modules of manageable size. Such a decomposition in the hardware environment, however, forces a corresponding software design decomposition. Thus, the decomposition principle in systems engineering constrains the software design. If, for example, the systems design is network oriented and prohibits the use of procedure calls (for communication purposes) then one might consider message passing or communication through a global database as an alternative. Because message passing exacts a high price in terms of performance, one might logically choose to use a common global database. This choice, however, dictates software design fraught with excessive coupling among modules.

The above scenario describes how decisions in one domain affect decisions in another. Such an influence is unavoidable because of the close correspondence between the software and hardware environments.

To summarize, there exist similarities between the two domains of software engineering and systems engineering. Not only are there similarities but also close correspondence exists betweeen these two areas. In the following chapter we discuss the future research directions and possible applications of the evaluation procedure.

7.0 Conclusion

In Chapter 1 this thesis presents the reader with the motivation behind research that focuses on the development of a procedure for the evaluation of software development methodologies. Chapter 2 presents the background behind the research. The notion of software engineering objectives, principles and attributes is then introduced in Chapter 3, wherein all the objectives, principles and attributes are discussed in detail. Later in Chapter 3, the evaluation procedure, with its top-down and bottom-up processes, is presented. Chapter 4 discusses the evaluation procedure and how the software attributes are identified and measured in the code. Chapter 5 explains how the evaluation procedure is applied to two existing software development methodologies. The previous chapter examines the systems engineering area and investigates the similarities and differences between the two disciplines. This chapter presents author's contribution to this research and discusses future research directions and applications of the evaluation procedure.

7.1 Contribution

This author's contribution in this research lies mainly in three areas which are:

- validation of linkages among software engineering objectives, principles and attributes,
- establishment of properties in assessing the absence/presence of attributes, and
- synthesis of metrics for measuring properties.

The pertinent statistics for the linkages is as follows.

No. of Objectives	**No. of Linkages**
---	---
Objectives - 7	
Principles - 7	Objectives/Principles - 32
Attributes - 9	Principles/Attributes - 24
Properties - 44	Attributes/Properties - 89

This author validated (through literature references) all 32 linkages between objectives and principles and all 24 linkages between principles and attributes. Out of 89 linkages between attributes and properties approximately 95% have been validated. Most of the remaining 5% linkages are comment related and intuitive.

The second area of author's major contribution is establishment of properties for assessing the presence/absence of attributes. The process of identification of properties (or factors) is continuing.

In the synthesis of metrics for the measurement of properties this author played a significant role. Based on the metrics developed in Appendix G the evaluation of two Navy methodologies was carried out.

7.2 Future Application

Based on experiences gained through the development of evaluation procedure, its application to two Navy methodologies and the analysis of the results, let us now focus attention to some extensions, enhancements, and refinements that might be incorporated to improve the evaluation procedure. In admitting that this research is still in its infancy, we realize that considerable research effort needs to be expended so that, eventually, the evaluation procedure has better implementability.

Experience has shown that two major improvements are needed:

- automation of the evaluation procedure through
- better assessment factors and metrics.

The first major concern is automation. Though the analysis of data is automated, the data collection process is not. Currently, data collection is manual. Such a manual op-

eration has a number of disadvantages. For example, the manual process is time consuming, requires excessive manpower, is subject to bias, and to some extent, error prone. The major advantages of the automated procedure are: (1) reduced manpower requirement, and (2) elimination of bias due to human involvement.

One of the main obstacles to automation, however, is the absence of structured documentation. Documentation, in most cases, is unstructured and, therefore, causes major impediment to the automation process. Lack of structured documentation necessitates the evolution of documentation standards. Such standards should incorporate traceability between physical and logical units, and should also provide verification and validation information [PARD71]. Once the documentation standards are set, automated analysis of the documentation can be more easily and effectively implemented. The documentation standards can also facilitate use of compiler-type techniques, and use of software tools like LEX and YACC for the automated analysis of documentation.

The second major improvement mentioned above is the better assessment of the software product. At present, the assessment of product attributes is through properties which are the surface qualities of code and documentation (these properties are referred to as factors in this thesis report). Such an assessment is not quite accurate because of the subjectivity involved in many factors. Assessment can be more accurate if the research evolves better assessment factors. Also, the measurement of such properties is based on the use of metrics. Presently, not all the metrics have a sound basis for comparisons and lack a firm statistical underpinning.

Once the above concerns are resolved, applicability of the evaluation procedure will improve considerably so that the evaluation procedure can be used to monitor software quality. That is, one can examine a software product using the evaluation procedure and

assess its (product's and methodology's) pluses and minuses. The minuses can be eliminated by implementing prescriptive suggestion implied by the evaluation procedure; thus, achieving software quality assurance (SQA). In other words, suppose a methodology is designed to focus on maintainability. If the score for maintainability is low, one can trace the reasons for the lack of maintainability through the software characteristics tree (Figure 7 on page 85). Through the linkages one can assess the root cause of the deficiency and determine steps to help restore maintainability (or any other objective). Thus, one can utilize the evaluation procedure to ensure the presence of desirable characteristics in the product. At the same time, the procedure will indirectly suggest ways and means of weeding out the undesirable characteristics, thus, improving the overall quality of the software product.

7.3 Summary

Software design characteristics are difficult to evaluate. However, it is often necessary to assess the "goodness" of a software design methodology by evaluating the software characteristics present in the resulting product. A lack of established relationships among software characteristics and a lack of quantifiable concepts have hampered any truly meaningful evaluations of software and software design characteristics. The research documented in this thesis outlines a procedural approach to evaluating software development methodologies.

The approach chosen in this research to address the above problem is unique. The procedural approach is based on the fundamental assumption that the needs dictate the process by which software is constructed. After partitioning the software characteristics

into three categories, viz., objectives, principles and attributes, this research establishes concrete relationships among these characteristics. These relationships are referred to as linkages. Using linkages as the basis, an evaluation procedure is developed for assessing software products (code and documentation) from the perspective of recognizing the presence or absence of various product characteristics.

The evaluation procedure is based on a two phase process. The procedure uses top-down and bottom-up evaluation processes to assess the product and the employed software development methodology. The top-down process begins by identifying the objectives of the methodology. The process then addresses the appropriateness of the software engineering principles. The last level visited in the top-down process is the software attributes. This step-by-step examination of the objectives, principles and attributes allows one to evaluate the methodology in terms of the adequacy of its goals and objectives. The bottom-up process, on the other hand, highlights the effectiveness of a methodology. This process starts at the attribute level and goes up the hierarchy illustrated in the characteristic tree.

Attributes are identified in the product using properties. This research establishes an association among properties and attributes. The research also presents a measurement approach, based on standard metrics, which helps assess the extent to which a property is present in the product. Next, the thesis report discusses the propriety of considering performance as a constraint, and not an objective. Finally, a comparative study of the disciplines of software enginering and systems engineering is presented.

To summarize, the feasibility and the useability of the evaluation procedure have been illustrated through the analysis of two real life methodologies. Through this analysis, the flexibility and inherent power of the procedure is also illustrated. Moreover, agreement

between the results of the analysis and perceived opinion tend to confirm the validity of the evaluation procedure. The proposed extensions and refinements are needed to further improve assessment reliability and to facilitate the collection of unbiased data.

Appendix A. OVERVIEW OF SOFTWARE DEVELOPMENT METHODOLOGIES

ABSTRACT

A method is a process or a procedure for attaining an objective while methodology is the study or science of method. This paper focuses on the software engineering methodologies which use different modeling strategies viz. functional decomposition, data-structure design methods, data-flow design methods, top-down, bottom-up and those that are developed specifically for information processing.

Key Words and Phrases: axioms, mechanism, methodology, modeling, modularity, requirements specification, simulation, software engineering, specification language.

TABLE OF CONTENTS

1. INTRODUCTION

Model development is an important phase in the simulation model life-cycle. It will not be an exaggeration to say that model development phase is the backbone of the model life-cycle. Over the years model development methodologies have gone through various stages of transition.

This paper gives an overview of different simulation model development methodologies. The methodologies considered here (28 in number) basically fall into two groups. The first group consists of those methodologies which are not directly related to software engineering. The second group deals with software development methodologies.

The above two groups are presented in two sections in this paper. The first section deals with three different schools of thought, viz., Program Generator (PG) Approach, Systems Theoretic approach and Conical Methodology (CM). In the software engineering category (Section 2), 25 methodologies (summarized in Table 4 on page 151) were proposed to be studied. Due to non-availability of reference material some of these methodologies (marked with '*') could not be studied. Nevertheless these references are included in the bibliography.

2. MODEL DEVELOPMENT APPROACHES

2.1 Program Generator Approach

Program Generator (PG) is a software tool which aids in translating logic of a model (described symbolically) into a simulation language. PG is an interactive system in which the modeler provides a model description in a natural-language-like format as input. The PG then provides an output in the form of a simulation programming language (SPL).

S.C.Mathewson [1] has developed a PG named DRAFT in which model specification is generated using entity-cycle diagram. The editor accepts model description, points semantic errors and allows the user to correct them. A back-up copy is created which can be corrected later and stored. The model analyzer checks input for errors (which can be corrected on-line) and a coded file of entity interactions is produced. The program writer uses this file as input and produces a program code.

2.2 System Theoretic Approach

System theoretic approach provides a general theory of simulation modelling [2]. It relies on the formalism provided by general systems theory. The model is represented using set theory. This approach is applicable to both continuous and discrete systems.

The formalism separates static and dynamic model description. Each model component is characterized by descriptive variables, and a component interaction section identifies the dynamic relationships in an informal description.

The biggest advantage of this approach is its generality. It could be applied to a variety of areas, e.g. behavioral science, ecological system, computer system and so forth.

2.3 Conical Methodology

Conical Methodology (CM) was developed by R.E.Nance[3]. CM employs top-down definition of a simulation model coupled with bottom-up specification of that model.

The definition proceeds through a partitioning process. In CM the model is broken up into sub-models while adhering to hierarchy. The submodel at each level is characterized by its value and relational attributes. This process continues till no further sub-model is necessary. Thus a conical structure is produced.

In CM, the specification part is not yet complete.

References:

1. Mathewson, S.C. (1984), "The Application of Program Generator Software and Its Extensions To Discrete Event Simulation Modelling," IIE Transactions, Vol. 16. No. 1.

2. Zeigler, B.P. (1984), "System-Theoretic Representation of Simulation Models," IIE Transactions, Vol. 16, No. 1.

3. Nance, R.E. (1981), "Model Representation in Discrete Event Simulation: The Conical Methodology," Technical Report CS 81003-R, Dept. of Computer Science, Virginia Tech, Blacksburg, Va.

3. SOFTWARE DEVELOPMENT METHODOLOGIES

Table 4 on page 151 summarizes the software development methodologies which are useful in simulation model development. All these methodologies were intended to be studied. Unfortunately reference material was not available on all the topics. The topics which could not be studied are marked with '*'.

3.1 ACTIVE AND PASSIVE COMPONENT MODELLING: ACM/PCM

3.1.1. Introduction

Active and Passive Component Modelling (ACM/PCM) is a model development methodology developed by M.L.Brodie and E.Silva. ACM/PCM is mainly used for the design and development of database systems. This method uses abstraction(suppression of minor details) to manage complex problems and to ensure semantic integrity to design and develop information systems.

3.1.2. Concepts

ACM/PCM employs four forms of abstraction viz: Classification, Aggregation, Generalization and Association. These forms which are used for structure modelling are described below.

3.1.3. Structure Modelling

Classification refers to a collection of objects where object is an instance of object class and an object class is characterization of common properties of objects. Classification treats collection of objects as a higher level object class. Classification is used to identify, classify and describe objects in terms of object classes. In 'Aggregation' relationship between component objects is treated as higher level aggregate object. Generalization is a type of abstraction where relationship between category objects is treated

Table 4. Methodologies Summarized

Methodologies Mnemonic	Full Name of Methodology
ACM/PCM	Active and Passive Component Modelling
DADES	Data Oriented Design
DSSAD *	Data Structured Systems Analysis & Design
DSSD *	Data Structured Systems Development
EDM	Evolutionary Design Methodology
GEIS *	Gradual Evolution of Information Systems
HOS	Higher Order Software
IBMFSD-SEP *	Adaptation of IBM Federal Systems Division Software Engineering Practices
IESM *	Information Engineering Specification Method
ISAC	Information Systems Work and Analysis of Changes
JSD	Jackson System Development
Merise *	
NIAM	Nijssen's Information Analysis Method
PRADOS *	Projektabwicklungs- und Dokumentationssystem
REMORA	
SADT	Structured Analysis & Design Technique
SARA *	System ARchitect's Apprentice
SD *	System Developer
SA-SD	Structured Analysis and Structured Design
SDM	System Development Methodology
SEPN *	Software Engineering Procedures Notebook
SREM	Software Requirements Engineering Methodology
STRADIS *	STRuductured Analysis, Design and Implementation of Information systems
USE	User Software Engineering

An asterisk (*) indicates lack of reference material.

as a higher level generic object. Association refers to the relationship between member objects. This relationship is treated as higher level set object.

At the conceptual level objects are composed to form aggregates, sets etc. using aggregation, generalization and association. Objects can also be decomposed into components, categories, and members using the same forms of abstraction. Thus objects can be composed or decomposed using these abstraction forms repetitively. At the transaction level in structure modelling, objects and their relationships are identified. Structure diagrams, E-R diagrams are used to represent the objects and structural relationships graphically.

3.1.4. Behavior Modelling

Each operation is either a retrieval or an alteration. Retrieval operations are CREATE, FIND whereas alterations are achieved by INSERT, DELETE, and UPDATE. Three forms of control abstraction exist, viz., sequence (analogous of aggregation in structural abstraction), choice (analogous of generalization) and repetition (corresponds to association). These three forms are sufficient for most of the applications.

At the conceptual level behaviour modelling identifies actions for each object. An action is defined as an operation by which all properties of an object are satisfied. Thus, action is the only way of altering an object. An action is representative of the behaviour of an object. The scope of an action consists of all objects related to the three forms of structural abstraction.

The behavioral and structural properties of an object together form data abstraction which defines the semantics of an object. At the transaction level behavior modelling is identification, design and specification of transactions. A transaction is defined as an operation which allows more than one object. The scope of transaction contains all objects accessed by the transaction.

Strict invocation hierarchy is observed to ensure semantic integrity of database applications. Behavior modelling and structure modelling are done together in two steps. Together, they keep the level of details low. The first step is to design behavioral properties followed by the actions, and then the transactions are identified and related to objects. In behavior modelling, diagrams (called as object schemes and action schemes) are used in designing. Step two involves specification of actions and their details. Specificatio is prepared using specification language BETA.

ACM/PCM decomposes database life cycle in six distinct stages. The life cycle phases provide complete guidelines and framework for design and development of information systems. These phases are:

1. Requirements formulation and analysis,
2. Logical design and specification,
3. Implementation design,
4. Implementation - coding, testing,
5. Operation, monitoring and maintenance, and lastly,
6. Modification.

3.1.5. Conclusion

ACM/PCM is a modelling methodology for moderate to large size database-intensive applications. It focuses mainly on specification. The use of ACM/PCM is mainly in in developing large information systems like university registration, real estate management, hotel reservation and so forth.

References:

1. Brodie, M.L. and L. Silva (1982), "Active and Passive Component Modelling: ACM/PCM," Proc. IFIP WG 8.1 Working Conference on Comparative Review

of Information Systems Design Methodologies, Noordwijkerhout, The Netherlands, pp. 41-92.

3.2 DATA ORIENTED DESIGN (DADES)

3.2.1. Introduction

Data Oriented Design (DADES) methodology was developed by Antoni Olive. DADES is an abbreviation of DAta oriented DESign. It deals with the data collected, stored or produced by an information system (IS) rather than with processes performing functions in an IS.

3.2.2. Description of DADES

DADES has three main components which are:

(i) a formal language for requirements specification,

(ii) a method for validating logical consistency of the specifications, and

(iii) a method for verifying logical correctness of decisions.

3.2.3. DADES Specification Language

DADES specification language consists mainly of conceptual schema definition and inputs and outputs. The foundation of the conceptual schema is based on two concepts 'life span' and 'universe of discourse'. Life span defines the time interval in which the system exists or operates. Universe of Discourse (UD) is a set of information that an information system derives, receives or stores during its life span.

DADES uses relational model and relational algebra. A set of all tuples in a given relation in a UD is called a total relation. Conceptual schema is the classes of elements of UD of an IS and the relation among classes.

The specification of the language consists of the conceptual schema of UD, i.e., domains, relation schemes, assertion times, value sets of attributes, and derivation rules. The input/output definition consists of information contents, instants of arrival, location and value estimates.

3.2.4. Validation of Specifications and Verification of Decisions

DADES has provided a formal method for the validation of logical consistency of IS requirements. For the verification of decisions DADES employs deriveability analysis, i.e., the problem of determining whether or not an information set i can be derived from other information sets (1,....i).

3.2.5. DADES Methodology

DADES methodology combines the above components to form an integrated technique. DADES is composed of three phases. The phases are: (i) Preliminary Specification Phase, (ii) Specification Phase and (iii) Architectural Design phase. A description of these phases follows.

(i) Preliminary Specification Phase

Preliminary specefication phase is carried out after requirements analysis stage. It involves the following steps:

a) List input/output requirements - Main inputs and outputs are determined by analyzing the object requirements and the information needs,

b) Develop abstract conceptual schema - This serves as the basis for the development of conceptual schema in the next phase. This also serves as a basis for naming convention, and lastly,

c) Decide the naming conventions - It assigns a naming rule to each attribute of the abstract conceptual schema.

(ii) Specification Phase

In the specification phase, DADES specification language is used to specify the requirements of the IS. In practice a number of iterations have to be made to arrive at the final specification. The steps involved are:

(a) Development of conceptual schema - The inputs to this step are the abstract conceptual schema and the naming convention. The output is the final conceptual schema.

(b) Definition of input/output requirements - Taking the conceptual schema and preliminary input/output requirements list as inputs, a formal expression is derived for each input/output requirement.

(c) Definition of derivation rules - a derivation rule is specified for each derived relation.

(d) Validate specifications - The specifications are validated for consistency. Some refinement of the conceptual schema may take place.

(iii) Architectural Design Phase

In the architectural design phase, logical design of system data base and the design of system structure are carried out.

3.2.6. Conclusion

Thus, DADES provides consistent specifications where the consistency can be formally verified. DADES is still in research stage at the University of Barcelona, Spain.

References

Olive, A. (1982), "DADES: A Methodology for Specification and Design of Information Systems," Proc. IFIP WG 8.1 Working Conference on Comparative Review of Information Systems Design Methodologies, Noordwijkerhout, The Netherlands, pp. 285-334.

3.3. EVOLUTIONARY DESIGN METHODOLOGY

3.3.1. Introduction

Evolutionary Design Methodology (EDM) was developed by Messers Rzevski, Trafford and Wells. EDM is used for designing information systems. This methodology aims at providing techniques for controlling:

1. the quality of information systems, and

2. productivity of information engineering personnel involved

3.3.2. Description of the Methodology

The measures for quality are reliability, usability, modifiability, portability, robustness, efficiency and so forth. EDM has identified the factors which contribute to the quality of information systems and the productivity of information engineering personnel. They are:

i) Human error factors: factors critically depending on human errors (e.g. reliability, maintainability),

ii) Interdependence factors: factors depending on interdependence of information systems,

iii) Controlability and observeability factors,

iv) Requirements factors: factors relevant to completeness of specification and changes in requirements,

v) System environment factors, viz.,

a) computing environment factors: factors causing the system to depend on computing environment and thus affecting portability, and

b) data environment factors: factors causing occurrence of incorrect inputs and thus affecting system robustness, and lastly,

vi) Computational complexity factors.

The analysis is then carried out to decide which of the above factors affect or are controllable by the constituent activities. The constituent activities are :

a) Project management,

b) Requirements specification,

c) System environment specification,

d) Design,

e) Coding,

f) Testing,

g) Installation, and

h) Maintenance.

3.3.3. Information Systems Design

EDM advocates trial-and-error technique in designing information systems. The technique involves generating tentative solutions, selecting the best solution and documenting decisions. Creation and selection of these solutions is done by humans while checking and documentation is done by computers. EDM, therefore, involves man-machine interaction.

Information systems design involves two major activities, viz., design of man-machine system and design of software. The first activity, man-machine system design, involves the following:

1. Define inputs to the system, outputs from the system , the function to be performed to achieve desired output, document the decision,

2. Divide the system in distinct modules which can be designed and implemented independently. For each module define inputs and outputs,

3. Design commands to achieve the results, design output formats,

4. Decide which subsystem to design first,

5. Divide the subsystem in functions performed by the user and the computer. Define inputs and outputs for each function,

6. Design external data structures for the selected subsystem,

7. Design control structures for the selected subsystem, and lastly,

8. Summarize the design.

The second activity in information system design is software design. The software design consists of the following activities.

1. Formulating functional specification for software system by defining input, output data sets and functions to be performed,

2. Identifying data entities and data structures,

3. Dividing the system into subsystems and identifying inputs and outputs for each subsystem,

4. Designing data structures,

5. Designing control structures,

6. Define functional specifications for each subsystem,

7. Design software subsystem which supports first man-machine software system using same steps as for design of software system,

8. Design each subsystem separately using steps 3, 4, 5 and 6,

9. Continue this procedure till all modules are transformed into a procedure or a subroutine or a function, and lastly,

10. Implement the design.

3.3.4. Conclusion

It can be seen that EDM is a general methodology applicable to a wide variety of systems. Besides applying EDM to the design of large information systems it has also been applied to the design of engineering products and educational systems.

References:

1. Rzevski, G., D.B. Trafford and M. Wells (1982), "The Evolutionary Design Methodology Applied to Information Systems," Proc. IFIP WG 8.1 Working Conference, Noordwijkerhout, The Netherlands, pp. 427-474.

3.4. HIGHER ORDER SOFTWARE

3.4.1. Introduction

Higher Order Software (HOS) was developed by Margaret Hamilton and Saydean Zeldin while working on NASA project. This method was invented in response to the need for a formal means of defining reliable, large scale, multiprogrammed, multiprocessor systems. HOS is a system oriented rather than a software oriented methodology.

HOS is a formal methodology with six axioms as the basis, a given system and all of its interfaces, is defined as if it were one complete and consistent computable system. The software automatically generates program code, thus eliminating the need for conventional programmers.

3.4.2. Basic Elements

HOS includes following five basic elements. These elements and their relationships are depicted in Figure 10 on page 163.

1. a set of formal laws,
2. a specification language,
3. an automated analysis of the system interfaces,
4. layers of system architecture produced from the analyzer output, and
5. transparent hardware,

In HOS any software can be represented by a mathematical function with input as domain and output as the range. The whole system is assumed to be a function with a subfunction for each of the subsystems representing the system. Thus, each subsystem can be represented individually by a single function. A binary tree (with a function at each node) shows how functions are decomposed into subfunctions. The leaves may be

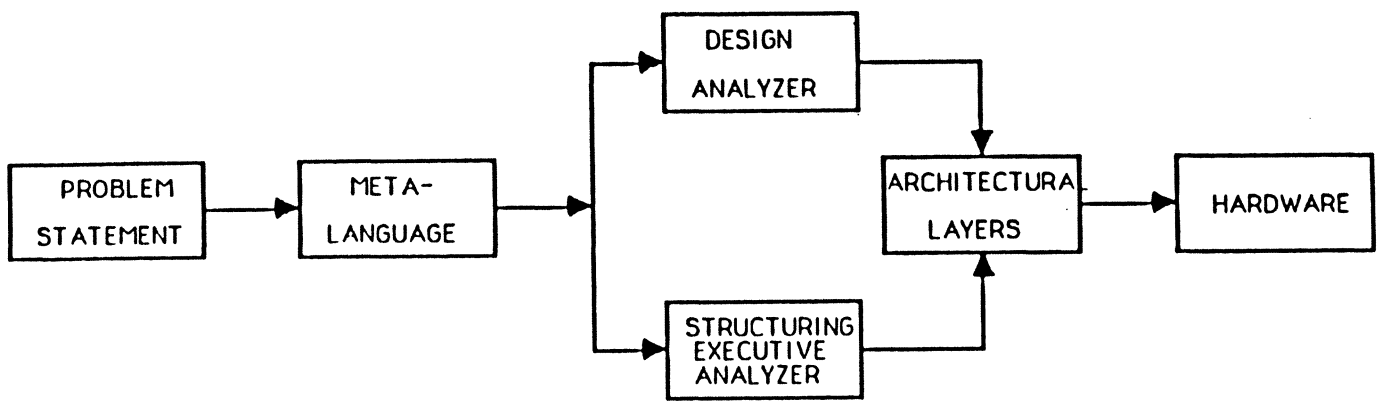


Figure 10. HOS Elements and their Relationships

primitive functions, functions already existing and stored in a library or functions obtainable from an external source.

3.4.3. Axioms

The design of software is based on six axioms which explicitly define a hierarchy of software control, wherein control is a formally specified effect of one software object on another. The six axioms are:

Axiom 1: A given module controls the invocation of valid functions on its immediate, and only its immediate lower level.

Axiom 2: A given module is responsible for elements of only its own output space.

Axiom 3: A given module controls the access rights to each set of variables whose values define elements of the output space for each immediate, and only each immediate, lower level function.

Axiom 4: A given module controls the access rights to each set of variables whose values define the elements of the input space for each immediate, and only each immediate, lower level function.

Axiom 5: A given module can reject invalid elements of its own, and only its own, input set.

Axiom 6: A given module controls the ordering of each of tree for the immediate, and only the immediate, lower levels.

3.4.4. Specification Language

HOS specification language separates properties of system performance from those of interface correctness. The language ensures reliable system decomposition. The analyzer checks the interface specifications written in HOS language. The automatic programmer converts HOS language from a specification level to the procedural level.

Design Analyzer checks it statically and the Structuring Executive Analyzer checks it dynamically.

HOS programs are normally built in modules and each module is checked with the Analyzers. HOS advises its user to run the model and examine it before a system is completely defined. This is done by simulating manually the portion which is as-yet defined.

3.4.5. Conclusion

HOS methodology is used in designing hardware and software, design of integrated circuits, complex radar systems and so forth.

References:

1. Hamilton, M. and S. Zeldin (1976), "Higher Order Software- A Methodology for Defining Software," IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, pp. 9-32.

3.5. INFORMATION SYSTEMS AND ANALYSIS OF CHANGES (ISAC)

3.5.1. Introduction

ISAC methodology was developed by Mats Lundeberg. ISAC is used for the specification of Information Systems (IS). The specifications are divided in the following three levels.

- Change analysis,
- Activity studies, and
- Information analysis.

In change analysis the designer finds reasons behind the problems. It starts with study of current situation and then studies the alternatives and ends with a development measure. The second level is activity studies. Activity studies separates information system from the organization. Also different ambition levels are studied in this step. The third level is information analysis. Information analysis describes the contents of the delimited information system. In this level the designer begins with the required results, then studies what is required to be done to achieve the results (precedence analysis), then the information sets are studied (component analysis) and finally the processes are described (process analysis). The above three levels are discussed below in detail.

3.5.2. Change Analysis

Change analysis comprises of three parts:

a) Analysis of needs, problems and current situation - This is achieved by first listing the expected problems and then analyzing the interest groups affected by these problems. These problems are then divided in various groups so that each

group can be dealt with separately. These groups are documented. The activities relevant to the problem are then illustrated by means of techniques like A-graphs, text pages for A-graphs, property tables and so forth. Then the objectives are described and a comparison of desires (objectives) with the availability is carried out. The differences provide the needs for changes.

b) Study of change alternatives - The change alternatives are created and described. These are summarized in a change alternative table. Each alternative is analysed and evaluated.

c) Choice of change approach - The decision about the choice of change alternative is taken.

3.5.3. Activity Studies

Activity studies is done through the following two steps.

a) Partitioning in information system - In this step, the information processing parts are described in more detail. Also the information subsystems are identified, classified and delimited.

b) Study of information subsystems - Each subsystem is individually studied. Cost-benefit analysis is carried out. Different ambition levels are generated and tested (desk test, field test) for each subsystem. In the end one ambition level which is most suitable is chosen.

An evaluation of the result of activity studies with the selected change approach is done.

3.5.4. Information Analysis

Information analysis is performed in three steps. These steps are:

a) Precedence and component analyses - Precedence analysis is the analysis of the information precedence relations. It is the analysis of information set. Component analysis deals with information set structure. It involves transition from activity studies. To achieve this transition input output information sets are extracted from A-graph of the subsystem and put in I-graph. The next step is to perform continued precedence analysis using I-graphs. In component analysis C-graphs are used.

b) Process analysis - The I-graph nodes represent information precedence relations. Here information processes are identified and described. Each description is documented in process tables.

c) Property analysis - In property analysis, the relevant property values are documented.

3.5.5. Conclusion

ISAC is a well-structured, result-oriented approach. It incorporates problem analysis, identification of affected groups and generation of high level activity graphs.

References:

1. Lundeberg, M. (1982), "The ISAC Approach to Specification of Information Systems," Proc. IFIP WG 8.1 Working Conference on Comparative Review of Information Systems Design Methodologies, Noordwijkerhout, The Netherlands, pp.173-234.

3.6. JACKSON'S SYSTEM DEVELOPMENT

3.6.1. Introduction

Jackson's System Development (JSD) methodology was developed by Michael Jackson. JSD covers complete technical development of a wide class of systems, from an initial statement of need through to their final implementation and subsequent maintenance. JSD is not based on stepwise refinement or top-down design, nor does JSD development start either by building a functional specification or a data model. Sequential processes are used as modelling tool.

3.6.2. Specifications

JSD develops formal system specifications in a number of distinct steps. The specifications are written in terms of sequential processes. JSD method involves six main steps. The steps are as follows:

(i) Entity/Action and entity structure step

(ii) Initial model step

(iii) Interactive function step

(iv) Information function step

(v) System timing step, and

(vi) Implementation step.

Steps (ii),(iii) and (iv) can be performed mostly in parallel as the decision in either step is independent of those in the other two. A description of the six steps follows.

(i) Entity/Action and Entity Structure Step

The purpose of this step is to define a model that is rich enough to support the functional requirements of the system. The model comprises of a set of action definitions and a number of disconnected sequential processes that describe the time constraints

between the actions. Attention is focused first on the subject matter of the system e.g. employees in a payroll system, cars in a car rally and so forth. The subject matter is dynamic, i.e., actions take place. Employees clock on, go sick, go and return from vacation, car passes different checkpoints etc.

In entity/action and entity structure step the modeler makes an abstraction of the subject matter. He concentrates on its time dimension. The abstraction is formal in that it is well defined and expressed in precise language. In 'JSD' the abstraction of subject matter is called 'model'. An 'action' is defined as an event in the world that forms the subject matter of the system and about which the system must produce or use information. An 'entity' is a person, organization or object that performs or suffers an interesting time sequence of actions.

Entity/action and entity structure step involves making a list of candidate actions, entities and producing following documentation.

- o Final list of actions and entities,
- o Action descriptions with associated attributes,
- o Entity/action cross reference, and
- o Minimum one structure diagram for each entity.

(ii) Initial Model Step

The initial model step and interactive function step realize the abstraction defined in the previous step as a set of processes within the system. These processes are called 'realized model'. The realized model is the basis of the specification and is a central part of the final implementation. Here, the actions are divided in two groups.

a) Those corresponding to events that are external to the system, and

b) Those that are automatically generated by the system.

In initial model step connection is established between external reality and realised model so that inputs are created to the realised model when external actions happen.

To distinguish between external world and realised model two terms, viz., level 0 and level 1 are introduced respectively. Input is collected for external actions to synchronize and coordinate the realised model with external reality. When an employee goes sick the employee process in the realised model gets an input that enables it to retain an accurate and up-to-date model of real employee.

A principle is worked out to detect an external action in the model. If an action is common to two or more structure diagrams in the model a way to replicate the input for that action is defined.

(iii) Interactive Function Step

Extra processes are added to the specification to generate inputs for the internally generated actions. Together the initial model step and the interactive step realize the abstraction defined in step 1 as a set of processes within the system.

Internal actions are generated in the model without interaction with external world, e.g. independent random number generator. The procedure is to find when the internal action should be generated. The answer may be in terms of other model actions or in terms of external input.

The relationship between level-0 objects, the model, interactive functions and the information function is shown in schematic diagram Figure 11 on page 172. The dotted lines show the interfaces on the behaviour of level-0 objects; these influences are not the concern of the developer.

(iv) Information Function Step

Extra processes are added to the specification to extract information from the model processes by data stream and state vector connection and to calculate the required outputs.

The outputs of the system are finally included in the specification. For each required stream of output, analysis is carried out to determine which processes in the

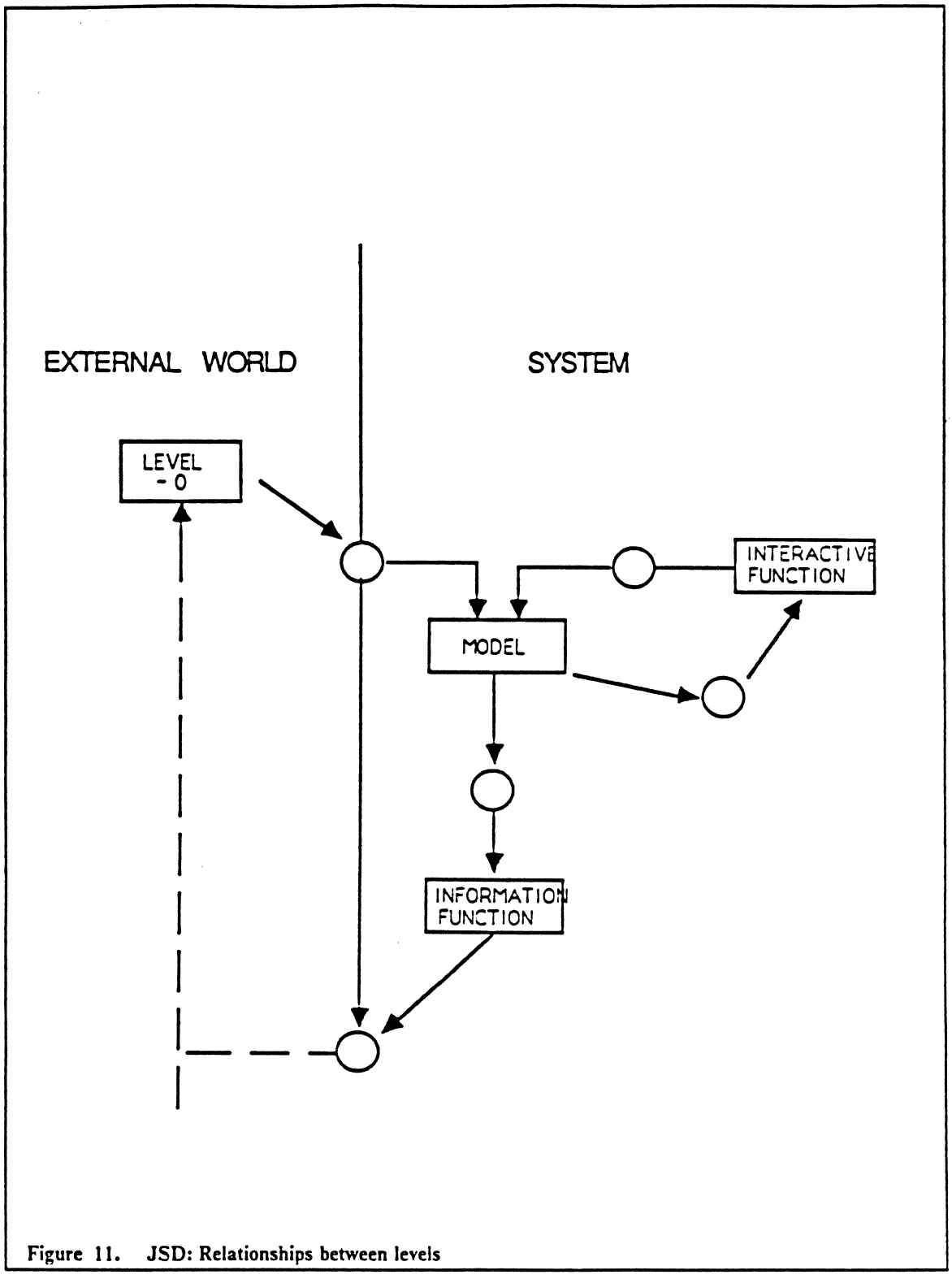


Figure 11. JSD: Relationships between levels

model contain necessary information and what type of connection (state vector or data stream) is appropriate for extracting it. Tentatively assuming a single function process, data stream and state vector connections from the model processes to function process are defined. The specification of function process is completed. A data structure is drawn of the stream of state vectors resulting from a state vector connection.

(v) System Timing Step

In system timing step timing constraints are expressed that must be satisfied in the implementation step. For each output stream, constraints are defined on the speed of the output and how up-to-date the information in the output needs to be. For each input stream constraints are defined on the input subsystem that must be satisfied to ensure that all the input is collected and all necessary orderings are preserved.

(vi) Implementation Step

The purpose of implementation step is to fit the specification developed in the system timing step to the target hardware/software environment so that it runs efficiently enough and meets the performance constraints defined in the system timing step. The main issues are the scheduling of the processes in the specification, the transformation of specification processes, and the organization of the storage of the state vectors of processes.

3.6.3. Conclusion

To put it in a nutshell JSD develops formal system specifications in a number of distinct steps. The specifications are written in terms of sequential processes. The early steps make a description or model of the relevant external reality while the later steps focus on functional requirements. The specifications are implemented in a series of mechanisable transformations.

References

1. Cameron, J.R. (1983), JSP & JSD: The Jackson Approach To Software Development, IEEE Computer Society Press Series, Silver Spring, Md.

3.7. NIAM : AN INFORMATION ANALYSIS METHOD

3.7.1. Introduction

Nijssens Information Analysis Method (NIAM) was developed by G.M. Nijssen. NIAM is an information analysis method which depends on top-down decomposition of system functions, followed by an analytical study of information flows and data relationships. NIAM has its own graphical notation. The semantics of the object system is built into the syntax of the data structure.

NIAM has its own definition language, Referential IDea Language (RIDL) which translates the special notation of Information Structure Diagrams (ISD) into machine processable forms.

3.7.2. Concepts

To reduce the complexity of transformation of information flows, NIAM uses functional decomposition technique. Decomposition is done into subfunctions till each subfunction cannot be divided further. The next step is to show the information flow with the help of information flow diagrams (IFD). Each level of decomposition is represented by an IFD, resulting in a hierarchy of IFDs. The IFDs give rise to ISDs.

NIAM differentiates between lexical objects and non-lexical objects. The lexical objects are defined as strings that can be uttered and which refer to an object while non-lexical objects are defined as real or abstract things in the object which are non-utterable.

The first step in modelling process is achieved by classifying lexical objects in lexical object types (LOT) and non-lexical objects in non-lexical object types (NOLOT). The sentences are decomposed into idea types and bridge types. To introduce abstraction NIAM describes constraints or rules which prescribe the behaviour of the system.

3.7.3. Diagrams

NIAM uses graphical notations to represent the Information Structure Diagrams (ISD). The diagrams represent the object and relationship in the real world or the object system. These diagrams are developed from the special notations developed for NIAM.

3.7.4. RIDL Language

RIDL language is an extension to the ISDs. The interpretation of a diagram into RIDL statement is a manual process. RIDL expresses the abstraction system. RIDL translates specifications of IFDs and ISDs. It also translates specifications of constraints which cannot be denoted graphically. Such constraints are called procedural constraints. RIDL allows specification of procedures. A procedure specifies performance of a function. A conceptual grammar expressed in RIDL is compiled and information system software is generated.

3.7.5. Conclusion

NIAM is a method to perform information analysis. NIAM can also be applied to project management approach provided the approach recognizes the need and the importance of information analysis.

References:

Verheijen, G.M.A. and J. Van Bekkum (1982), "NIAM: An Information Analysis Method," Proc. IFIP WG 8.1 Working Conference on Comparative Review of Information Systems Design Methodologies, Noordwijkerhout, The Netherlands, pp. 537-590.

3.8. REMORA METHODOLOGY

3.8.1. Introduction

Remora methodology was developed by C. Rolland and C. Richard. Remora methodology, named after French fish 'remora', is proposed for information systems design and management.

The design is divided into two steps viz. conceptual step and internal step. The conceptual step concentrates on the semantic aspect while the internal step deals with the technical aspects. The conceptual step or level represents conceptual schema and the internal step or level represents internal schema. The two schemas are described below.

3.8.2. Conceptual Schema

The conceptual schema requires two elements which are:

a) a conceptual model, and

b) a formal language.

The two elements are discussed below.

a) The Model

In the dynamic perspectives the model defines objects, events and operations. An object is a specific, lasting component. An operation is an activity that changes state of one or more objects at a given time. An event is defined as any happening at a given time. The model also defines three associations. They are: (1) modify (association between operations and objects), (2) ascertain (association between objects and events) and (3) trigger (connects events and operations). In modify, an operation changes the object while in ascertain a change in state of object is an event. In trigger, an event starts off an operation. The causal representation is shown in Figure 12 on page 178.

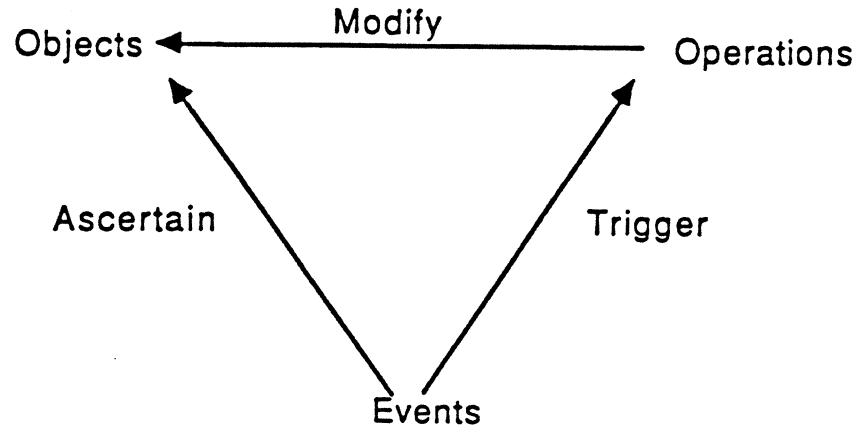


Figure 12. REMORA: Causal Representation

Remora uses the relational model to represent the conceptual model. Remora defines three types of relations viz: c-object, c-operation and c-event to represent the relational model.

C-object is in 3NF with each attribute having permanent dependency with the prime attribute. It represents the basic state (atomic) of the information system. C-operation expresses the smallest transformation that c-object undergoes. C-event represents a temporal aspect of real event class. The conceptual schema is a collection of the above three relation types.

b) ISDEL Language

Remora has defined a language ISDEL (Information System DEfinition Language) which is both DDL and DML. ISDEL has four basic constructions which are assertional expressions, relation predefined type (like record in Pascal), FOREACH control structure (to express manipulation of tuples) and language extensions (embodiment of functions like Total, Count, Average etc.).

3.8.3. Internal Schema

The internal schema constitutes of two subschemas which are: data saturation subschema and process synchronization subschema. In data saturation subschema, the designer defines the way data is stored in order to satisfy queries efficiently. It also takes care of future extension to the present database. In process synchronization subschema the designer defines how conceptual operations are gathered into modules, the sequencing rules for module execution and so forth.

3.8.4. Conclusion

Remora's correct tool specification in description of the problem and its (methodology's) dynamic nature renders it useful in various applications, viz.,

o distributed DBMS,

o office management for design of automated tools, and

o real time technical software design.

References

1. Rolland, C and C. Richard (1982), "The Remora Methodology for Information Systems Design and Management," Proc. IFIP WG 8.1 Working Conference on Comparative Review of Information Systems Design Methodologies, Noordwijkerhout, The Netherlands, pp. 369-426.

3.9. STRUCTURED ANALYSIS AND DESIGN TECHNIQUE

3.9.1. Introduction

Structured Analysis and Design Technique (SADT) methodology was developed by Douglas T. Ross. It SADT is useful for requirements analysis as well as for design. SADT is a general purpose modelling technique that is applicable to a wide range of problems, not just computer applications. SADT is a graphic representation of the system's hierarchic structure decomposed with a specific purpose in mind.

3.9.2. Mechanism

In SADT the modeler builds the model in a top-down fashion. Model is decomposed gradually to the required level of detail. The model gradually exposes more and more detail which is dictated by requirements analysis. A series of diagrams are used with system boundaries which illustrate gradual decomposition. Figure 13 on page 182 illustrates an example.

The diagram consists mainly of boxes (representing parts of a whole) interconnected by arrows (interfaces between parts). Each box is further blown up with additional details in the next diagram. Node numbers are used to denote relationship between diagrams. The top diagram is called Ao. Each box in a diagram is numbered. The node which gives more details of the box 3 of the Ao diagram is called A3. The decomposition of box 3 (on node A3) is called node A33.

Each activity is represented by a box on the diagram and labelled by a verb on the box. Arrows entering or exiting a box represent objects or information inputed or outputed respectively. Arrows entering the box are either input, control or mechanism. Exiting arrows are outputs (Figure 13 on page 182).

The side of the box the arrow enters determines its role. Input arrows at top and left of the box represent all the data required for the activity. Input on the left of the box

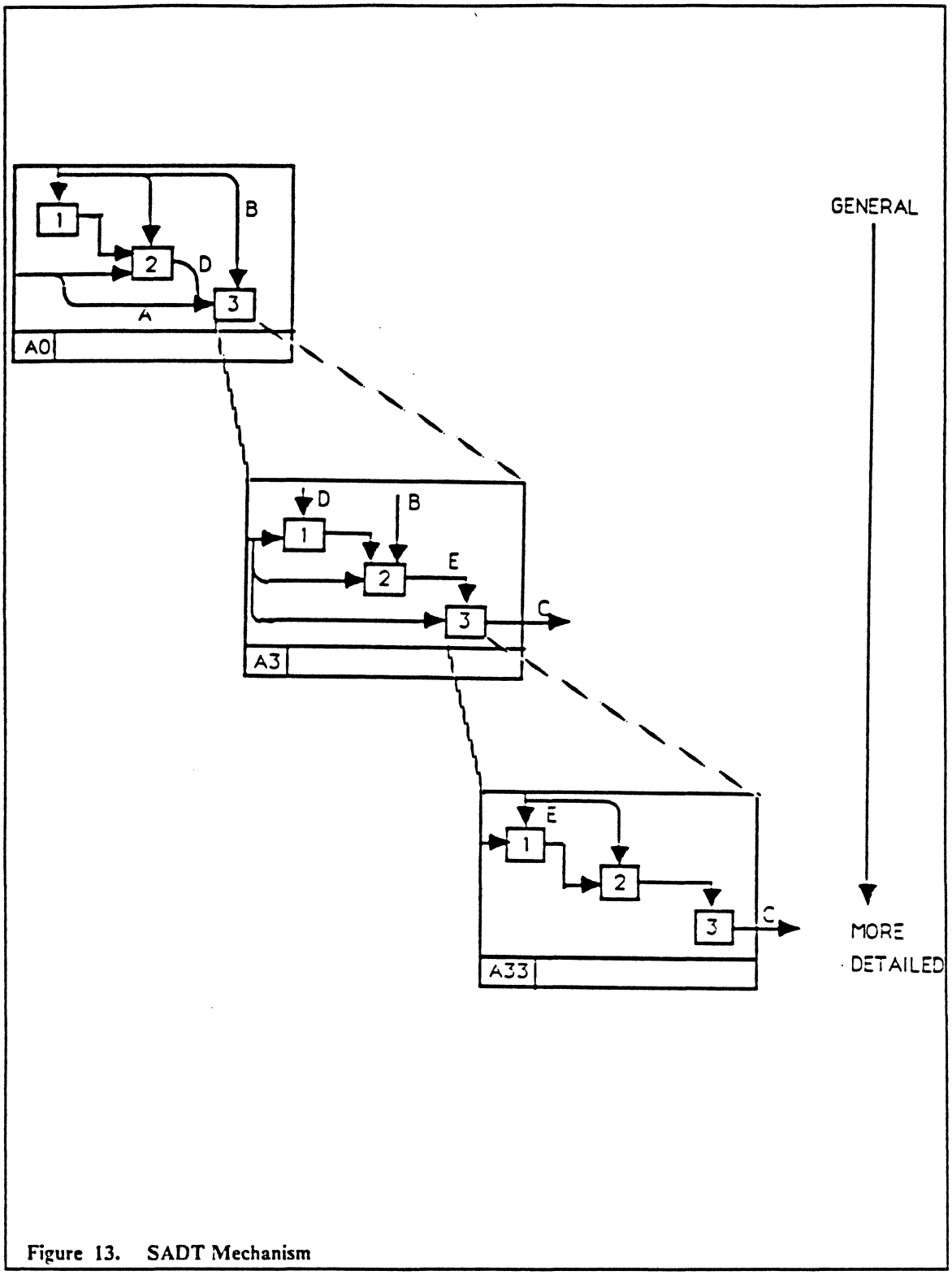


Figure 13. SADT Mechanism

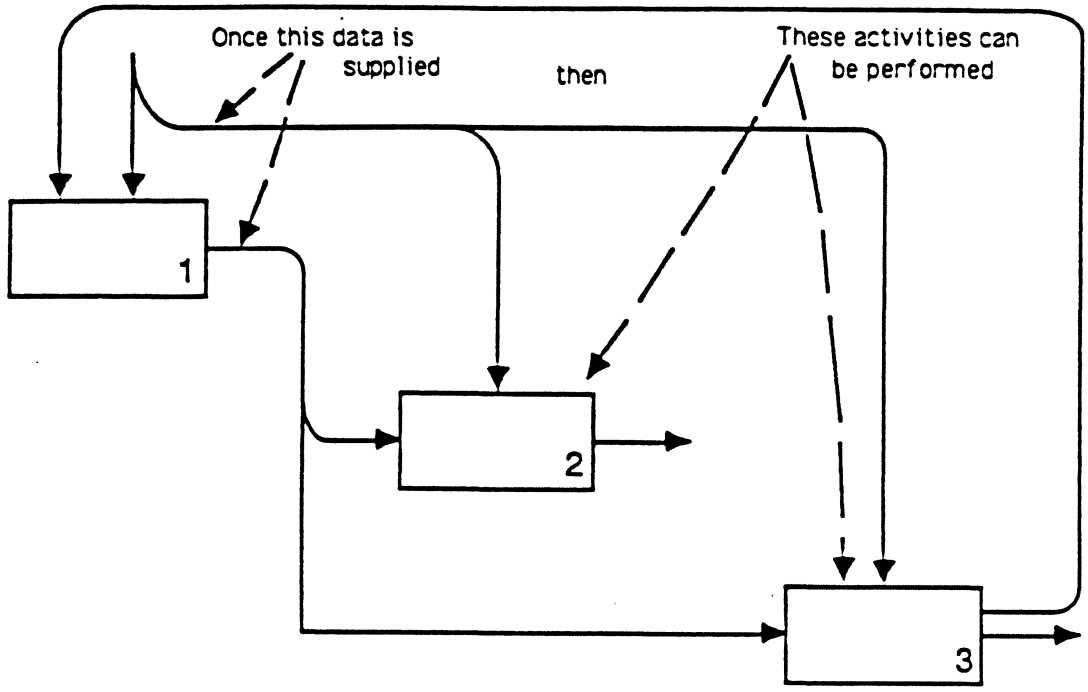


Figure 14. SADT: Activity Dependence

is converted into the output with the help of control on top of the box. The inputs, outputs, control along with the box represent what the system does. Mechanism (arrow entering the bottom of the box) represents how the activity is accomplished. Mechanism which is described by a name on the arrow can be a person, a process or a device. It could also be referenced by a separate SADT model.

3.9.3. Sequencing

The arrow indicates constraint relationship. It also shows a sequence in that an activity cannot be performed unless data from the previous/preceding activity is available. SADT also allows concurrency, that is, many activities could be running simultaneously. Figure 14 on page 183 explains the dependence of activities on the data supplied by the preceding activity or activities. The figure also illustrates simultaneous activities and feedback.

Arrows represent categories of data. At higher levels the categories are more general and as the system is decomposed each arrow is also decomposed (Figure 14 on page 183) and it gives more details. Arrows can also undergo a 'join' operation so that general data category is performed combining different arrows (Figure 14 on page 183).

3.9.4. Conclusion

SADT offers the conceptual and notational tool for expressing, evaluating and comparing design alternatives. It has been successfully implemented in simulating real-time telephonic communications design and military policy planning. SADT is compatible to contemporary design methodologies like Structured Design, Jackson's System Design and so forth.

References

1. Ross, D.T., G.L. McGowan and M.E. Dickover (1979), "Software Design Using SADT," In: G.D. Bergland and R.D. Gordon (editors), Tutorial: Software Design Strategies, Compsac.

3.10. STRUCTURED ANALYSIS

3.10.1. Introduction

Structured Analysis (SA), a tool used for communicating ideas in software development, was developed by D.T.Ross. SA combines a graphic language with the nouns and verbs of written language to provide a hierarchical model particularly well suited for the documentation of ideas and transfer of understanding. The function of structured analysis is to provide a framework through which ideas can be precisely represented and communicated in an efficient manner.

3.10.2 Mechanism

SA uses basic building block to document both objects (data) and events (activities). In the SA model, arrows represent input, output, control and mechanism relationships among the blocks. This technique presents 40 features which people understand and communicate ideas to the structured collection of diagrams and notation which constitute SA models.

An SA model consists of an organized structure of separate diagrams, each exposing only a limited portion of the object. Bounded pieces of subject matter are diagrammed using a box representing a transformation of an object or an activity. Arrows interconnecting these boxes travel from the output face of one to either the input face or control face of another box. Input activity to an object box creates the object. The output activity references such an object. For an activity; the input causes the activity; the output is transformed into an object. The control constrains the transformation to ensure that it applies only under the appropriate circumstances. Further the control constraints for activity boxes define and impose the structural decomposition of the model. An arrow terminating on the fourth face of the box specifies the mechanism for affecting the transformation diagrammed.

The component pieces, into which a subject is broken, are brought together by the interconnecting arrows to constitute the complete object. The SA model represents both the objects and activities of the system being described, using the same box notation, thus emphasizing the duality of activities and data. The graphical syntax provides a framework upon which all the aspects and relationships between concepts can be organized.

The SA language also includes non-graphic notation which provides a means of referencing the diagrams and the component boxes, and special graphical conventions used to simplify the illustrations.

3.10.3. Conclusion

SA is a rigorous demonstration and communication tool which forms the basis of Structured Analysis and Design Technique (SADT).

References

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

3.11. STRUCTURED DESIGN

3.11.1. Introduction

Structured design (SD) was developed by Larry L. Constantine and advanced by E. Yourdon and G. Myers. SD is concerned with the development of well-structured system, the ability to compare alternative designs and determine their relative quality and the transformation of data flow diagrams into program structures.

3.11.2. Methodology

The structured design method consists of concepts, measures, analysis techniques, guidelines, rules-of-thumb, notation and terminology. Reliance is placed upon following the flow of data through the system to formulate program design. The data flow is depicted through a special notational scheme which identifies each data transformation, transforming process and the order of their occurrence.

3.11.3. Modularity

One of the key issues in SD is modularity. SD regards a module as simply some named set of contiguous program statements such as a subroutine. SD uses special graphical notation which consists of:

1. Modules A and B,
2. A invokes B. B is subordinate to A, and
3. B receives an input parameter X (its name in module A) and returns a parameter Y (its name in module A).

In creating a design, major emphasis is given to the effectiveness of the modular decomposition. Modularity is measured by two yardsticks: (1) cohesion and (2) coupling. Cohesion refers to the extent to which the components of a module are conceptually related. Coupling refers to the way in which the modules are related to one another and

the form of interconnections that are used in the design, primarily the way in which data are communicated among modules.

3.11.4. Cohesion and Coupling

The categories in SD for cohesion (from strong to week) and coupling (from low to high) are listed below. Cohesion levels are:

- o Functional: module performs a single specific function, e.g. 'write a record to output file'.
- o Clustered: module is a group of functions sharing a data structure usually to hide its representation from the rest of the system; only one function is performed per invocation, e.g. 'symbol table with insert and look-up functions'.
- o Sequential: module action comprises of several functions that pass the data along, e.g. 'update and write a record'.
- o Communicational: module action consists of several functions operating on some data, e.g. 'print and punch a file'.
- o Procedural: module elements are grouped for algorithmic reasons, e.g. 'loop body'.
- o Temporal: module functions are all related in time, e.g. 'intialization'.
- o Logical: module can perform a general function (i.e. several logically related functions); an invoking parameter value determines the specific function, e.g. 'general-error-routine' called with an error number.
- o Coincidental: no real relationship between module elements that are grouped for packaging considerations, e.g.'common sub-expression'.

Coupling levels are:

- o Data: all communication between modules is via arguments that are the data elements.

o Stamp: communication between modules includes an argument that references a data structure (some of whose fields are not needed).

o Control: an argument from one module knowingly influences the flow of control of the other, e.g. a flag.

o External: reference is made to an externally declared data element.

o Common: modules reference an externally (i.e. common) declared data structure (some of whose fields are not needed).

o Content: a module references the content of the other.

3.11.5. Design

The design process in SD is split into:

i) general program design, and

ii) detailed design.

While the general program design deals with what functions are required for the program (or programming system), the detailed design deals with the implementation of functions. SD uses bubble charts based on data flow and transforms them into Structure Charts.

3.11.6. Conclusion

In SD the flow of data through a system is the key to the program design. The system specification is used to produce the data flow diagram, the diagram to develop the data structure chart, the chart to develop the data structure and all of the pieces to reinterpret the system specification. SD is useful in architectural design and provides the modeler with a high level system architecture.

References

1. Stevens, W.P., G.J. Myers and L.L. Constantine (1979), "Structured Design," In: G.D. Berglang and R.D. Gordon (editors), Tutorial: Software Design Strategies, IEEE Computer Society, Compsac.

3.12. SYSTEM DEVELOPMENT METHODOLOGY

3.12.1. Introduction

System Development Methodology (SDM) was developed by PANDATA , a software company in Netherlands. SDM basically provides guidelines for development of computerized systems. SDM describes a life cycle of a system from basic inception to the delivery of the system. SDM is a comprehensive study of the methods required for the development of the systems.

3.12.2. Methodology

SDM prescribes a sequence of seven stages. The seven stages are:

1. Definition Study,
2. Preliminary Design,
3. Detail Design,
4. Programming,
5. Testing,
6. Data Conversion and System Implementation, and
7. Maintenance and Support.

In Definition Study stage , a user raises a request and a problem is defined . In order to define the problem accurately a number of iterations are performed. During this stage the problem is formulated, possible solutions are evaluated, cost/benefit analysis is carried out, functional requirements are established. This is the beginning of system design efforts. This stage also involves specifying system inputs, outputs and functions.

In Preliminary Design stage , identification of system functions is carried out. The whole system is considered as a single entity. System environment and subsystems are defined, flowcharts indicating interactions between subsystems are prepared. Logical

data base structure and access technique are designed. System software and hardware configuration are specified. The resulting Preliminary Design report contains a system overview and subsystem design requirement.

In the third stage , Detail Design is carried out. Here the basic design is expanded and a detailed design of all program modules, information files, manual procedures etc. is done. The specifications evolved as a result of this stage are precise , explicit and in maximum detail. In this stage all computer logic is specified in detail, software utilities, common routines specified, input/output interfaces are defined.

In the fourth stage, the actual program code is generated. All system components are coded in detail. Coding also involves preparing source program files, compiling and assembling the programs. Debugging programs and data are also included in this stage. Programming begins with detailed specifications and ends with preparation of all system components, prior to testing.

The 'testing' phase starts with a debugged program. Detailed test plans and procedures are developed in this stage. The system is tested thoroughly employing sound testing principles including test planning, test monitoring, saturation testing etc.

Data conversion and system implementation begins with a fully tested program. In this stage conversion and implementation plan is established. Conversions are documented. Operations personnel are trained on the hardware, software and the new system.

In the last stage of operation, maintenance and support, system's performance is monitored and evaluated. Failures are prevented. Change requests are processed. Maintenance procedures are recommended.

3.12.3 Modularity and Structured Design

SDM advocates modularity and structured design. The system is split into three phases. Adherence to the sequence of phases will automatically incorporate modularity in design.

During phase 1, development takes place at the system level. The system is considered as the design entity. During phase 2, the system is divided into subsystems. Each part is individually developed and implemented independently. In the third stage the automated processes are further divided into modules and programs.

3.12.4 Conclusion

SDM is widely used in project planning, project control and in general project management.

References

1. Hice, G.F., W.S. Turner and L.F. Cashwell (1978), System Development Methodology, North-Holland, Amsterdam, The Netherlands.

3.13. SOFTWARE REQUIREMENTS ENGINEERING METHODOLOGY (SREM)

3.13.1. Introduction

SREM is a requirements methodology developed as a part of U.S. Army Ballistic Missile Defense software by TRW, Inc. SREM was evolved for the generation of software requirements for large, real-time, unmanned weapons systems.

SREM combines language support software and steps for generating and evaluating the software requirements. The language used for stating requirements in SREM is called Requirements Statement Language (RSL); it also has graphical notation called R-Nets for describing concurrency of operations and automated tools to support the creation and checking of requirements documents.

3.13.2. Goals

SREM specifies three goals viz:

a) development of a language for statement of requirements,

b) tools to ensure completeness, consistency, correctness, and

c) developing requirements in the above language.

3.13.3. Key Concepts

SREM defines seven key concepts which are

i) Real time software is tested by inputting an interface message resulting in output message and using the resulting data for processing another message.

ii) Processing is defined in terms of relationships of input and output messages, processing steps, intermediate data produced and utilized.

(iii) The performance is defined in terms of a test which assures testability and removes ambiguity regarding the requirement. A sequence of processing steps

beginning arrival of message at the interface to termination of process is called 'PATH'.

(iv) A requirements network (R-Net) is drawn to illustrate the different paths taken by processing. It clearly indicates relationships between paths.

(v) Use formal language RSL for the requirements statement

(vi) Using automated tools to ensure consistency and completeness of requirements and validating their correctness. They are integrated into an REVS (Requirements Engineering and Validation System).

(vii) The intermediate products produced by the methodology steps are evaluated for completeness.

3.13.4. Methodology Steps

SREM methodology involves various steps. The resulting intermediate products are evaluated for their completion. The assumption of the methodology is that a DPSPR (Data Processing Subsystem Performance Requirement) collects system functions and performances after they are allocated to the data processor.

Step 1: Translation

In translation step the DPSPR is translated and interpreted, the DPSPR summary (in RSL) is entered into REVS database, R-nets are drawn, data and processing steps are generated with traceability back to DPSPR, the consistency and completeness of requirements are analysed, DPSPR problem reports generated, and the remaining requirements generation activities are planned. Translation is complete when the source of every functional requirement and performance requirement is identified or included in DPSPR problem report and DPSPR requirement is mapped onto some processing.

Step 2: Decomposition

In decomposition step all requirements on the data processor identified in the interface specification are accounted for in the processing description and the form of the performance requirements and the engineering trade studies to obtain the performance bounds are identified. Performance requirement is defined as a set of three activities viz: identifying the requirements form, specify variables to be measured and record the decisions made to relate the accuracy and timing back to the specification of DPSPR performance requirements.

Step 3: Allocation

Allocation determines the sensitivities of the path performances to the DPSPR performances. A trade-off is required to be set between the accuracies and timings of different paths which have to satisfy the requirements and to select allocation which does not impose major constraint in any direction. This step uses functional simulators e.g. weapon system which uses simulation by processing messages. REVS is used to generate the simulator. It assures traceability of the simulation and the requirements. The requirements and the tests are written using RSL.

Step 4: Analytical Feasibility Demonstration

In this step an algorithm is used for each type of processing. If an algorithm is not existing it is developed. The activities are building simulators. REVS is useful in combining algorithms into an analytical simulator which is driven with realistic data produced by simulating an environment. It is not necessary that the above sequence must be followed in the development and validation of requirements.

3.13.5. Conclusion

SREM is the combination of language, support software and steps. SREM is useful in solving real-time problems.

References

1. Alford, M.W. (1977), "A Requirements Engineering Methodology for Real-Time Processing Requirements," IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, pp. 60-69.

3.14. USER SOFTWARE ENGINEERING

3.14.1. Introduction

USE methodology was developed by A.I.Wasserman. USE was developed in order to evolve interactive information systems (IIS) which are reliable, easy to use and less costly. User Software Engineering (USE) takes into consideration factors ranging from programming practices to hardware selection and from system definition to psychological factors.

USE uses a number of tools to achieve its goal. The tools are TDI (Transition Diagram Interpreter), Troll, PLAIN (Programming LAnguage for INteraction). The details of these tools are described later.

3.14.2. Methodology

USE methodology has the following steps.

(i) Requirements analysis,

(ii) User/program dialog design,

(iii) Creation of a mock-up of the user/program dialog,

(iv) Specification of system operations,

(v) Preliminary relational database design,

(vi) RAPID construction of partial systems,

(vii) Formal specification of system operations,

(viii) System design at architectural & module levels, and

(ix) Implementation in PLAIN.

The above steps are described in detail below.

(i) Requirements Analysis is the first step in USE methodology.

This is achieved through:

a) Identification of system objectives and constraints,

b) Modeling the system,

c) Constructing conceptual model of the database,

d) Producing a system dictionary containing names of all operations, data items, and

e) Reviewing the analysis results.

This step is a determinant of system quality, including reliability, usability.

(ii) Design and modification of user/program dialog

Design is the most critical stage in designing IIS. It uses transition diagrams to pictorially represent language definition. In this step the aspects that are incorporated in the design are: system is kept interactive, underlying aspects of the computer are hidden from the user, system provides on-line assistance, input requirements are tailored to user characteristics, response time is consistent for same request.

The third step is creation of a mock-up of the user/program dialog. In this step TDI is used to encode transition diagrams. TDI accepts one or more transition diagrams as input and produces an executable program. It helps build prototype of the system.

The next step involves specificationof system operations. The specifications provide a link between the user's concepts of the system and its actual implementation. USE provides two views of specification.

a) User view - It is detailed but informal description of the system. Its aimed at comprehensiblity for the reader and completeness of the system description.

b) Developer view - It provides a view of the system functions. This view defines a high level program structure, identifies program models and shows logical data flow between modules.

(v) Preliminary relational database design

In this step the data base is specified as a set of normalized relations. The existence of relational DBMS simplifies implementation of prototype systems.

(vi) RAPID construction of partial systems

The partial system is constructed from TDI and Troll (a tool that provides a relation algebra-like interface to a relational database system). This step helps link data manipulation and/or routines written in high level languages to user/ program dialog.

(vii) Formal specification of system operations

The specification thus consists of a set of transition diagrams, the database specified as a set of normalized relations and the operations.

(viii) System design at architectural and module levels

The design process involves developing the program structure and logic that describes how to build the system subject to stated constraints. The architectural design involves creation of overall program structure and the logical data design. This stage leads to the decomposition of the system into a set of modules with well-defined input and output interfaces.

(ix) Implementation in PLAIN

PLAIN (Programming LAnguage for INteraction) is the implementation language of USE. PLAIN provides a number of facilities which include a set of relational database management, support for strings and string handling, exception handling and pattern matching.

The software support system for the USE methodology is Unix. USE control system is a command driven tool which permits the user to create modules and systems, to modify aspects of the module, to compile one or more modules, to examine the status and history of the development and to run the concurrent version of the system.

3.14.3. Conclusion

USE methodology provides the developer of interactive information systems with a method and tools that improve the quantity of the system and the process by which they are built.

References

Wasserman, A.I. (1982), "The User Software Engineering Methodology: An Overview," Proc. IFIP WG 8.1 Working Conference on Comparative Review of Information Systems Design," Noordwijkerhout, The Netherlands, pp. 591-628.

4. CONCLUDING REMARKS

We have seen a number of model development methodologies, many of which are applicable to a particular type of problem. But their usefulness in simulation is obvious. In short it is difficult to find a method which would be an asset in every design problem, because each design problem is unique in some way and each method has its limitations.

Methods are important but the skill lies in adopting a method or methods to suit one's problem. Therefore the methods contribute to a certain extent, it is the modeler who produces design, methods do not.

Also their successful application occurs in supportive environments. The management elements (planning, scheduling, control systems etc.) must all be present and effective. In larger systems, these non-technical factors assume greater importance.

BIBLIOGRAPHY

Alford, M.W. (1977), "A Requirements Engineering Methodology for Real-Time Processing Requirements," IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, pp. 60-69.

Brodie, M.L. and L. Silva (1982), "Active and Passive Component Modelling: ACM/PCM," Proc. IFIP WG 8.1 Working Conference on Comparative Review of Information Systems Design Methodologies, Noordwijkerhout, The Netherlands, pp. 41-92.

Cameron, J.R. (1983), JSP & JSD: The Jackson Approach To Software Development, IEEE Computer Society Press Series.

Campos, I.M. (1978), "Concurrent Software Systems Design Supported by SARA at the Age of One," Proc. 3rd International Conference on Software Engineering Atlanta, Ga., pp. 230-242.

Evers K.H., R.E. Bachert and P.R. Santucci (1981), "SADT/SAINT: Large Scale Analysis Simulation Methodology," Proc. Winter Simulation Conference, Atlanta, Ga., pp. 185-191.

Hamilton, M. and S. Zeldin (1976), "Higher Order Software- A Methodology for Defining Software," IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, pp. 9-32.

Hice, G.F., W.S. Turner and L.F. Cashwell (1978), System Development Methodology, North-Holland, Amsterdam, The Netherlands.

Lundeberg, M. (1982), "The ISAC Approach to Specification of Information Systems," Proc. IFIP WG 8.1 Working Conference on Comparative Review of Information Systems Design Methodologies, Noordwijkerhout, The Netherlands, pp. 173-234.

Marmor-Squires, A. (1978), "An Annotated Bibliography of Structured Analysis and Design," Infotech State of the Art Report, Vol. 1, pp. 229-256.

Martin, J. (1984), System Design from Provably Correct Constructs, Prentice-Hall, New Jersey.

Mathewson, S.C. (1984), "The Application of Program Generator Software and its Extensions to Discrete Event Simulation Modelling," IIE Transactions, Vol. 16, No. 1.

Mueller-Merbach, H. (1983), "Model Design Based on the Systems Approach," Journal of Operational Research Society, Vol. 34, No. 8, pp. 739-751.

Nance, R.E. (1979), "Model Representation in Discrete Event Simulation: Prospects for Developing Documentation Standards," In: N. Adam and A. Dogramaci (eds.), Current Issues in Computer Simulation, Academic Press, pp. 83-97.

Nance, R.E. (1981), "Model Representation in Discrete Event Simulation: The Conical Methodology," Technical Report CS 81003-R, Department of Computer Science, Virginia Tech, Blacksburg, Va.

Olive, A. (1982), "DADES: A Methodology for Specification and Design of Information Systems," Proc. IFIP WG 8.1 Working Conference on Comparative Review of Information Systems Design Methodologies, Noordwijkerhout, The Netherlands, pp. 285-334.

Oren, T.I. (1982), "Computer-Aided Modelling Systems," In: F.E. Cellier (ed.), Progress in Modelling and Simulation, Academic Press, pp. 189-203.

Oren, T.I. and B.P. Zeigler (1979), "Concepts for Advanced Simulation Methodologies," Simulation, Vol. 32, No. 3, pp. 69-82.

Parnas, D.L. (1978), "Some Software Engineering Principles," Infotech State of the Art Report, Vol. 2, pp. 237-248.

Porcella, M., P. Freeman and A.I. Wasserman (1983), "Ada Methodology Questionnaire Summary," ACM SIGSOFT Software Engineering Notes, Vol. 8, No. 1, pp.51-58.

Ramamoorthy, C.V. and R.T. Yeh (1979), Tutorial: Software Methodology, IEEE Computer Society, Long Beach, Ca.

Retti, J. (1982), "A Framework for Interactive Engineering of Simulation Models for Time-Oriented Systems," In: R. Trappl (ed.), Cybernetics and Systems Research, North-Holland, Amsterdam, pp. 199-204.

Rolland, C. and C. Richard (1982), "The Remora Methodology for Information Systems Design and Management," Proc. IFIP WG 8.1 Working Conference on Comparative Review of Information Systems Design Methodologies, Noordwijkerhout, The Netherlands, pp. 369-426.

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

Ross, D.T., G.L. McGowan and M.E. Dickover (1979), "Software Design Using SADT," In: G.D. Bergland and R.D. Gordon (editors), Tutorial: Software Design Strategies, Compsac.

Rzevski, G., D.B. Trafford and M. Wells (1982), "The Evolutionary Design Methodology Applied to Information Systems," Proc. IFIP WG 8.1 Working Conference on Comparative Review of Information Systems Design Methodologies, Noordwijkerhout, The Netherlands, pp. 427-474.

Schruben, L. (1982), "System Modelling with Event Graphs," Technical Report No. 498, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, NY.

Stevens. W.P., G.J. Myers and L.L. Constantine (1979), "Structured Design," In: G.D. Berlang and R.D. Gordon (editors), Tutorial: Software Design Strategies, IEEE Computer Society, Compsac.

Verheijen, G.M.A. and J. Van Bekkum (1982), "NIAM: An Information Analysis Method," Proc. IFIP WG 8.1 Working Conference on Comparative Review of

Information Systems Design Methodologies, Noordwijkerhout, The Netherlands, pp. 537-590.

Wasserman, A.I. (1975), "Some Principles of User Software Engineering for Information Systems," IEEE Digest of papers, Compcon, Spring, pp. 172-175.

Wasserman, A.I. (1982), "The user Software Engineering Methodology: An Overview," Proc. IFIP WG 8.1 Working Conference on Comparative Review of Information Systems Design, Noordwijkerhout, The Netherlands, pp. 591-628.

Willis, R.R. (1974), "Structured Model Development Techniques," Simuletter, Vol. 5, No. 4, pp. 41-52.

Winchester, J.W. and G. Estrin (1982), "Requirements Definition and its Interface to the SARA Design Methodology for Computer Based Systems," Proc. AFIPS National Computer Conference, Houston, Texas.

Zeigler, B.P. (1984), "Multifaceted Modeling Methodology: Grappling With the Irreducible Complexity of Systems," Behavioral Science, Vol. 29, pp. 169-178.

Zeigler, B.P. (1984), "System-Theoretic Representation of Simulation Models," IIE Transactions, Vol. 16, No. 1.

Zeigler, B.P. and Roy Rada (1984), "Abstraction in Methodology: A Framework for Computer Support," Information Processing and Management, Vol. 20, No. 1-2, pp. 63-79.

Appendix B. LINKAGES BETWEEN OBJECTIVES & PRINCIPLES

Objectives and principles and how they are related

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

ADAPTABILITY

Achieved by:

(a) Hierarchical Decomposition (*)

(b) Functional Decomposition (*)

(c) Information Hiding (*)

(d) Stepwise Refinement (*)

(e) Structured Programming (*)

(f) Documentation (*)

References:

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

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

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

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

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

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

NOTES

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

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

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

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

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

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

7. PP193/(ii): Flexibility is a function of modularity, self-documentation etc. {Documentation}

   Modularity provides a structure of highly independent modules, having sharply defined interfaces that are tolerant to external changes. A module possessing functional strength and data coupling has little to fear from the outside and its internal logic is often so easily understood that enhancing the module becomes simple.

   Self-documentation identifies the software attributes that explain the function of the software. Without a clear understanding of the code, the programmer will have a hard time identifying where to insert new functions or to change or delete old ones.

MAINTAINABILITY

Achieved by:

(a) Hierarchical Decomposition (*)

(b) Functional Decomposition (*)

(c) Information Hiding (*)

(d) Stepwise Refinement (*)

(e) Structured Programming (*)

(f) Documentation (*)

References:

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

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

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

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

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

NOTES

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

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

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

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

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

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

CORRECTNESS

Achieved by:

(a) Hierarchical Decomposition (*)

(b) Stepwise Refinement (*)

(c) Structured Programming (*)

(d) Life-cycle verification (*)

References:

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

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

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

NOTES

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

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

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

REUSABILITY

Achieved by:

(a) Hierarchical Decomposition (*)

(b) Functional Decomposition (*)

(c) Information Hiding (*)

(d) Documentation (*)

References:

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

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

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

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

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

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

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

NOTES

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

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

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

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

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

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

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

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

TESTABILITY

Achieved by:

(a) Hierarchical Decomposition (*)

(b) Functional Decomposition (*)

(c) Information Hiding (*)

(d) Stepwise Refinement (*)

(e) Structured Programming (*)

(f) Life-cycle Verification (*)

References:

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

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

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

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

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

(vi) Bauer, F.L. (1975), Software Engineering: An Advanced Course, Springer Verlag, New York, NY.

NOTES

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

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

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

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

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

6.PP156/(ii) Stepwise refinement and modularity are closely aligned with abstraction.

PP289/(vi): One of the advantages of a modular approach to the construction is that the problems of testing and debugging are greatly simplified. {Stepwise Refinement}

RELIABILITY

Achieved by:

(a) Hierarchical Decomposition (*)

(b) Information Hiding (*)

(c) Stepwise Refinement (*)

(d) Structured Programming (*)

References:

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

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

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

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

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

NOTES

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

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

3. PP82/(i): Structured Programming is a programming discipline which was introduced with reliability in mind. {Structured Programming}

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

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

PORTABILITY

Achieved by:

(a) Functional Decomposition (*)

(b) Documentation (*)

References:

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

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

NOTES

1. PP123/(i): Portability is a function of five underlying metrics: generality, hardware independence, modularity, self-documentation, and software system independence. Modularity provides a structure of highly independent modules. Independent modules are less likely to have input/output statements that are one of the banes of portability. {Functional Decomposition}

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

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

Appendix C. LINKAGES BETWEEN PRINCIPLES & ATTRIBUTES

Principles and attributes and how they are related

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

HIERARCHICAL DECOMPOSITION

Induces:

(a) Coupling (*)

(b) Cohesion (*)

(c) Complexity (*)

(d) Ease of Change (*)

References:

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

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

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

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

NOTES

1. PP158/(i): The concept of module independence is the direct outgrowth of modularity and the concepts of abstraction and information hiding.
   Independence is measured using two qualitative criteria: cohesion and coupling. {Coupling, Cohesion}

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

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

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

FUNCTIONAL DECOMPOSITION

Induces:

(a) Coupling (*)

(b) Cohesion (*)

(c) Complexity (*)

(d) Ease of Change (*)

References:

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

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

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

(iv) Parnas, D.L. (1972), "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM, Vol. 15, No. 12, pp. 1053-1058.

NOTES

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

2. PP158/(i): The concept of module independence is the direct outgrowth of modularity and the concepts of abstraction and information hiding.
   Independence is measured using two qualitative criteria: cohesion and coupling.
   Cohesion is a natural extension of the information hiding concept. A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program. {Coupling, Cohesion}

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

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

INFORMATION HIDING

Induces:

(a) Coupling (*)

(b) Cohesion (*)

(c) Complexity (*)

(d) Well-defined Interface (*)

(e) Ease of Change (*)

References:

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

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

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

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

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

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

NOTES

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

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

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

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

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

   This method (information hiding) addresses real issues such as how to design for ease of change. {Ease of Change}

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

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

- Identification of the items that are likely to change. These items are termed "secrets,"
- Location of the specialized components in separate modules, and
- Designing intermodule interfaces that are insensitive to the anticipated changes. The changeable aspects or "secrets" of the modules are not revealed by the interface.

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

6. PP195/(vii): Much of the complexity of embedded real-time software is associated with controlling special-purpose hardware devices. Many designers seek to reduce this complexity by isolating device characteristics in software device interface modules, thereby allowing most of the software to be programmed without knowledge of device details. {Complexity}

STEPWISE REFINEMENT

Induces:

(a) Coupling (*)

(b) Cohesion (*)

(c) Complexity (*)

References:

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

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

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

NOTES

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

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

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

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

STRUCTURED PROGRAMMING

Effect on derived attributes:

(a) Complexity (*)

(b) Readability (*)

References:

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

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

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

NOTES

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

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

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

LIFE_CYCLE VERIFICATION

Induces:

(a) Visibility of Behavior (*)

(b) Early Error Detection (*)

References:

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

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

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

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

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

NOTES

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

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

3. PP23/(iii): Confirmability (Life-Cycle Verification)- Confirmability is a principle that directs attention to methods for finding out whether stated goals have been achieved. Applied to design issues, confirmability refers to the structuring of a system so it is readily tested. It must be possible to stimulate the constructed system in a controlled manner so its response can be evaluated for correctness. Applied to notational matters, confirmability means that a notation should require explicit specification of constraints that affect the correctness of a design or implementation (e.g., data declarations that specify range of values and units of value as well as mode of representation). Applied to the practice of software engineering, confirmability refers to the use of such methods as structured walkthroughs of designs, egoless programming, and other methods that help to ensure that nothing has been overlooked.

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

5. PP110/(v): Ref. Fig. 2. Continuous verification ensures that errors are found early and at least cost. {Early Error Detection}

DOCUMENTATION

Achieves:

(a) Reduced Complexity (*)

(b) Improved Readability (*)

(c) Improves Ease of Change (*)

(d) Traceability (*)

References:

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

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

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

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

NOTES

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

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

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

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

Appendix D. ATTRIBUTE/FACTORS RELATIONSHIPS

Software Engineering Attributes

and

their Related Assessment Factors

COUPLING

Influenced by:

(a) Use of global variables (-)

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

(c) Use of switches as parameters (-)

(d) Use of parameters (-)

(e) Use of parameterless procedure calls (-)

(f) Types of parameters- control vs data

(1) Control (-)

(2) Data (+)

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

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

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

COHESION

Influenced by:

(a) Use of switches as parameters (-)

(b) Use of control structures (+)

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

(d) Multiple exit points from loops (-)

(e) Multiple exit points from routines (-)

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

(g) Fan out (-)

(h) Modularization (+)

COMPLEXITY

Influenced by:

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

(b) Use of control structures (+)

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

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

(e) Use of dynamic structures (-)

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

(g) Multiple exit points from loops (-)

(h) Multiple exit points from routines (-)

(i) Fan out (-)

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

(k) Use of recursive code (-)

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

(m) Use of embedded alternate language (-)

(n) Use of code indentation (+)

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

(p) Length of routine/module (-)

(q) Total program length (-)

(r) Modularization (+)

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

(t) Use of block comments (+)

(u) Use of comments (+)

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

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

(x) Completeness/Accuracy of documentation (+)

WELL-DEFINED INTERFACE

Influenced by:

(a) Use of global variables (-)

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

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

(d) Use of parameterless procedure calls (-)

READABILITY

Influenced by:

(a) Use of control structures (+)

(b) Use of symbolic constants (+)

(c) Use of special characters (+)

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

(e) Multiple exit points from loops (-)

(f) Multiple exit points from routines (-)

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

(h) Use of embedded alternate language (-)

(i) Use of code indentation (+)

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

(k) TLOC > ELOC (+)

(l) Length of routine/module (-)

(m) Use of block comments (+)

(n) Use of comments (+)

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

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

(q) Grammatically correct comments/spellings (+)

(r) Completeness/Accuracy of documentation (+)

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

(t) Use of parentheses around conditions (+)

(u) Multiple statements on one line (-)

EASE OF CHANGE

Influenced by:

(a) Use of global variables (-)

(b) Use of parameters (-)

(c) Use of dynamic structures (+)

(d) Use of symbolic constants (+)

(e) Fan out (-)

(f) Modularization (+)

(g) Use of "sandwiching" (+)

(h) Completeness/Accuracy of documentation (+)

TRACEABILITY

Influenced by:

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

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

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

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

VISIBILITY OF BEHAVIOR

Influenced by:

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

(b) Awareness of validation & verification (+)

(c) Procedures of validation & verification (+)

(d) Enforcement of validation & verification (+)

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

EARLY ERROR DETECTION

Influenced by:

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

(b) Awareness of validation & verification (+)

(c) Procedures of validation & verification (+)

(d) Enforcement of validation & verification (+)

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

Appendix E. ASSESSMENT FACTORS

The product attributes to which they are related and why

USE OF GLOBAL VARIABLES

Affect:

(a) Coupling (-)

(b) Well-defined Interface (-)

(c) Ease of Change (-)

References:

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

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

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

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

NOTES

1. PP98/(i): Whenever two or more modules interact with a common data environment, those modules are said to be common-environment coupled. Each pair of mod-

ules which interacts with the common environment is coupled- regardless of the direction of communication or the form of reference. {Coupling}

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

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

4. PP111/(iv): Common Coupling

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

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

USE OF STRUCTURED DATA TYPES AS PARAMETERS

Affects:

(a) Coupling (-)

(b) Well-defined Interface (+)

References:

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

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

NOTES

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

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

USE OF STRUCTURED DATA TYPES AS VARIABLES

Affects:

(a) Complexity (+)

References:

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

NOTES

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

USE OF SWITCHES AS PARAMETERS

Affects:

(a) Coupling (-)

(b) Cohesion (-)

References:

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

NOTES

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

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

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

USE OF PARAMETERS

Affect:

(a) Coupling (-)

(b) Ease of Change (-)

References:

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

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

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

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

NOTES

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

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

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

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

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

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

USE OF "EXCESSIVE" # PARAMETERS

Affect:

(a) Well-defined Interface (-)

References:

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

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

NOTES

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

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

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

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

USE OF PARAMETERLESS PROCEDURE CALLS

Affect:

(a) Well-defined Interface (-)

(b) Coupling (-)

References:

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

NOTES

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

TYPES OF PARAMETERS- CONTROL vs DATA

Affect:

(a) Coupling

Control (-)

Data (+)

References:

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

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

NOTES

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

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

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

USE OF CONTROL STRUCTURES

Affect:

(a) Cohesion (+)

(b) Complexity (+)

(c) Readability (+)

References:

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

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

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

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

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

NOTES

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

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

3. PP115/(iii): Cohesion - The goal here is to strive for modules whose elements (statements and functions) are highly related. {Cohesion}

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

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

USE OF "EXCESSIVE" NESTING OF CONTROL STRUCTURES

Affects:

(a) Complexity (-)

(b) Readability (-)

References:

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

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

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

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

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

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

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

NOTES

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

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

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

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

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

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

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

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

"EXCESSIVE" USE OF CONTROL STRUCTURES

Affects:

(a) Complexity (-)

References:

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

NOTES

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

   DD = (Total # Decisions)/ ELOC

   Higher the DD, higher the complexity. {Complexity}

USE OF DYNAMIC STRUCTURES

Affects:

(a) Ease of Change (+)

(b) Complexity (-)

References:

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

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

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

NOTES

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

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

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

USE OF SYMBOLIC CONSTANTS

Affects:

(a) Readability (+)

(b) Ease of Change (+)

References:

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

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

(iii) Shneiderman, B. (1977), "Measuring Computer Program Quality and Comprehension," International Journal of Man-Machine Studies, Vol. 9, No. 4, pp. 465-478.

NOTES

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

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

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

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

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

6. PP466/(iii): Commenting, mnemonic variable names and modular programs have a significant (favorable) impact on program comprehension. {Readability}

USE OF SPECIAL CHARACTERS

Affects:

(a) Readability (+)

References:

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

NOTES

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

USE OF PARENTHESES AROUND CONDITIONS

Affects:

(a) Readability (+)

References:

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

NOTES

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

USE OF MEANINGFUL NAMES FOR ROUTINES, VARIABLES

Affect:

(a) Readability (+)

(b) Complexity (+)

References:

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

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

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

(iv) Shneiderman, B. (1977), "Measuring Computer Program Quality and Comprehension," International Journal of Man-Machine Studies, Vol. 9, No. 4, pp. 465-478.

NOTES

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

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

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

4. PP466/(iv): Commenting, mnemonic variable names and modular programs have a significant (favorable) impact on software comprehension. Complexity, Readability}

MULTIPLE ENTRY POINTS IN A ROUTINE

Affect:

(a) Coupling (-)

(b) Cohesion (-)

References:

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

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

NOTES

1. PP88/(i): Use of multiple entry points guarantees that there is more than the minimum number of interconnections for the system. {Coupling}
   If a module has multiple entry points, it is implied that there are pieces of code in the module that are performing single specific functions which in turn implies that the module is not functionally cohesive. {Cohesion}

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

MULTIPLE EXIT POINTS FROM LOOPS

Affect:

(a) Complexity (-)

(b) Readability (-)

(c) Cohesion (-)

References:

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

(ii) Parker, C.S. (1987), Understanding Computers and Data Processing: Today and Tomorrow, HRW Publishers, New York, NY.

NOTES

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

2. PP299/(ii): An extremely important fact about the control structures is that each of them permits only one entry point into and one exit point out of the structure. This is sometimes called the one-entry-point/ one-exit-point rule. The one-entry-point/one-exit-point convention encourages a modular programming approach that makes programs more readable. {Readability, Cohesion, Complexity}

MULTIPLE EXIT POINTS FROM ROUTINES

Affect:

(a) Complexity (-)

(b) Readability (-)

(c) Cohesion (-)

References:

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

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

NOTES

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

2. PP110/(ii): Program modules must remain single purpose, with single entry and exit points. A program cannot jump into the middle of a module, nor can it leave the

module except via its sole exit. (Such) Modules that perform a single logical function are called cohesive. {Cohesion}

MULTIPLE STATEMENTS ON ONE LINE

Affect:

(a) Readability (-)

References:

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

NOTES

1. PP395/(i): There should be no multiple statements on one line. They affect readability.

FAN-IN TO A ROUTINE

Affect:

(a) Cohesion (+)

(b) Coupling (-)

References:

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

NOTES

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

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

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

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

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

FAN-OUT

Affects:

(a) Ease of Change (-)

(b) Complexity (-)

(c) Cohesion (-)

References:

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

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

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

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

NOTES

1. PP97/(i): Multiple calls from a module indicate a tendency toward too much control within a single module. The problem may be a missing level. The function and

control contained within the original module may need to be divided. {Ease of Change, Complexity, Cohesion}

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

3. PP111/(iv): Since modules are parents, just how many child modules should each control? Span of control refers to the number of subservient modules controlled by a parent module. Spans of five to nine are considered ideal. Spans of one or two are too few, and spans of 12 to 15 are too many. {Ease of Change, Complexity}

GOTO's & USE OF GOTO's

Affect:

(a) Complexity (-)

(b) Readability (-)

(c) Coupling (-)

References:

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

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

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

NOTES

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

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

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

USE OF RECURSIVE CODE

Affects:

(a) Complexity (-)

References:

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

NOTES

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

USE OF NEGATIVE/COMPOUND BOOLEAN EXPRESSIONS

Affects:

(a) Complexity (-)

References:

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

NOTES

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

USE OF EMBEDDED ALTERNATE LANGUAGE

Affect:

(a) Readability (-)

(b) Complexity (-)

References:

Justification:

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

{Readability}

USE OF CODE INDENTATION

Affect:

(a) Readability (+)

(b) Complexity (+)

References:

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

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

NOTES

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

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

"EXCESSIVE" USE OF CODE INDENTATION

Affect:

(a) Readability (-)

(b) Complexity (-)

References:

Justification:

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

TLOC vs ELOC

Affects:

(a) Readability (+)

References:

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

NOTES

1. PP 26/(i): It is possible to write an executable statement in one or more lines. In this case a program with more lines of code is more readable than one where TLOC = ELOC.

   (TLOC = > Total Lines Of Code)

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

LENGTH OF ROUTINE/MODULE

Affects:

(a) Complexity (-)

(b) Readability (-)

References:

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

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

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

NOTES

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

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

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

TOTAL PROGRAM LENGTH

Affects:

(a) Complexity (-)

References:

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

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

NOTES

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

2. PP767/(ii): The features of program that can qualify for use in the measurement of program complexity are:
   a) the total # imperative statements
   b) the average depth of 'IF' nesting
   c) the total # lines of source code
   d) the total # entry points
   e) the total # boolean variables declared
   f) the average # parameters passed

g) the average # SLOC jumped by a forward transfer of control

h) the total # lines of annotation in the source code. {Complexity}

MODULARIZATION

Affects:

(a) Cohesion (+)

(b) Ease of Change (+)

(c) Complexity (+)

References:

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

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

(iii) Shneiderman, B. (1977), "Measuring Computer Program Quality and Comprehension," International Journal of Man-Machine Studies, Vol. 9, No. 4, pp. 465-478.

NOTES

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

2. PP76/(ii): I have alluded to the need for modularity in program design as a means toward organizing the program into subdivisions (which can be considered sepa-

rately) to cope with complexity during the development phase, and to cope with side effects when later changes or corrections are made. {Complexity}

3. PP466/(iii): Commenting, mnemonic variable names and modular programs have a significant (favorable) impact on the program comprehension. {Complexity}

USE OF "EXCESSIVE" NUMBER OF ROUTINES

Affects:

(a) Complexity (-)

References:

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

NOTES

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

USE OF "SANDWICHING"

Affects:

(a) Ease of Change (+)

References:

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

NOTES

1.PP 190/(i): "Sandwiching" resolves the conflict resulted when two programs want to use each other. {Ease of Change}

USE OF BLOCK COMMENTS

Affects:

(a) Complexity (+)

(b) Readability (+)

References:

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

NOTES

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

USE OF COMMENTS

Affects:

(a) Readability (+)

(b) Complexity (+)

References:

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

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

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

(iv) Shneiderman, B. (1977), "Measuring Computer Program Quality and Comprehension," International Journal of Man-Machine Studies, Vol. 9, No. 4, pp. 465-478.

NOTES

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

CD = (# of Comments)/TLOC

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

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

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

4. PP466/(iv): Commenting, mnemonic variable names and modular programs have a significant (favorable) impact on program comprehension. {Complexity, Readability}

USE OF "EXCESSIVE" # SINGLE LINE COMMENTS IN LINE

Affects:

(a) Readability (-)

(b) Complexity (-)

References:

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

NOTES

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

USE OF COMMENTS CONSISTENT WITH CODE FUNCTIONS

Affect:

(a) Readability (+)

(b) Complexity (+)

References:

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

NOTES

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

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

GRAMMATICALLY CORRECT COMMENTS/SPELLINGS

Affect:

(a) Readability (+)

References:

Justification:

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

CERTIFICATION LEVELS IN "COMMENTS"

Affect:

(a) Visibility of Behavior (+)

(b) Early Error Detection (+)

References:

Justification:

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

USE OF COMMENTS REFERENCING PROJECT DOCUMENTATION

Affect:

(a) Traceability (+)

References:

Justification:

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

USE OF COMMENTS REFERENCING "who called me"

Affect:

(a) Traceability (+)

References:

Justification:

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

COMPLETENESS/ACCURACY OF DOCUMENTATION

Affect:

(a) Readability (+)

(b) Complexity (+)

(c) Ease of Change (+)

References:

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

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

NOTES

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

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

gram runs, how to load and start after failure, how to keep the program maintained, etc. {Complexity, Ease of Change}

Appendix F. QUESTIONNAIRE TO ASSESS DOCUMENTATION PROPERTIES

EVALUATION OF DOCUMENTATION

DEFINITION:

SOFTWARE REQUIREMENTS - A subset of system performance objectives allocated to the software.

LOGICAL DESIGN - The Decomposition of a system into into its functional primitives and the interactions among them.

PHYSICAL DESIGN - The allocation of the logical design components and interactions to physical (procedural/data) components and events (procedure calls, scheduling, etc.).

1. IS THERE TRACEABILITY BETWEEN:

- Software requirements to/from logical design _____
- Logical design to/from physical design _____
- Logical design to/from testing procedures _____
- Physical design to/from CODE _____

0 ☞ None
1 ☞ Occasionally
2 ☞ Usually
3 ☞ Almost always
4 ☞ Excellent

EVALUATION OF DOCUMENTATION

SYSTEM NAME:

Document Name	Standards and Practices							Testing		
	REQUIREMENTS	DESIGN	PROGRAMMING	CORRECTNESS ANALYSIS	SOFTWARE REQUIREMENTS	LOGICAL DESIGN	PHYSICAL DESIGN	REQUIREMENTS	PLANS	PROCEDURES

GENERAL DOCUMENT EVALUATION

Is there obviously missing or incomplete information?
Select a 50 page segment of the document at random.
Count the total number of TBS or TBD paragraphs or sections. _____

Are appropriate presentation tools used in the document?
Select "primitive description" segments at random until you have located five for which some "presentation aid" (e.g. flow chart, structure diagram, decision tree, structured English, pseudo-code, etc.) would be helpful.

A. In how many of the five selected cases did the document actually employ some "presentation aid"? _____

B. In how many cases was the choice optimal or near optimal? _____

Do change bars (or some equivalent indication) appear to have been used effectively (Yes, No or N/A)? _____

Are revision numbers explicitly noted in the document? _____

A. Can these revision numbers be easily associated with the appropriate revisions of associated documents (e.g. If I have a certain revision of a design document, can I easily determine the specific revision of the requirements document with which it is compatible?)? _____
Rate as follows: 0 = Easily _____
1 = With some difficulty
2 = With great difficulty or only with some uncertainty

B. Use the same scale to rate the ease with which a specific revision of a document can be associated with:

1) A specific platform _____
2) A configuration or baseline _____

GENERAL DOCUMENT EVALUATION

Are the following attributes included where appropriate?

A. Memory Size _____
B. Timing Requirements _____
C. Precision/Scaling _____
D. Min/Max Ranges _____

(Rate as follows: 0 = N/A 1 = Never 2 = Occasionally
3 = Sometimes 4 = Usually 5 = Almost Always)

Rate the ease with which this document can be checked for consistency or completeness as follows: _____

1 = In an automated fashion
2 = In a manual fashion with assistance from the format, cross-references, presentation, etc. of the document
3 = Only by "brute force"

Does the document contain a table of contents? _____
An index? _____
Are data elements (in addition to processing elements) included in the index? _____

Within the document, select 10 words which should be indexed. How many of them are? _____

For 10 randomly selected table-of-contents entries, are the page numbers referenced accurately? _____

Is a consistent format used between documents:

A. Table of contents? _____
B. Lists of Illustrations/Tables/etc.? _____
C. Section order? _____
D. Glossary? _____
E. Index? _____
F. Paragraph/Section/Subsection Numbering Scheme? _____
G. Abstract? _____
H. Overview? _____
I. Cross-Reference Matrix to other documents? _____
J. List of Acronyms? _____
K. Appendices? _____

Is there a cross-reference between each Appendix and the sections which reference it? _____

GENERAL DOCUMENT EVALUATION FOR PPS/PDS

Select "primitive description" segments at random until five are located which require some input which is produced by a primitive which is described elsewhere in the same or another document.

A. How easily can all sources of the input be located? _____
Rate as follows: 0 = Easily
1 = With some difficulty
2 = With great difficulty or only with some uncertainty

B. Is the input referred to by the same name in all references (Yes, No or Unable to Locate)? _____

Do the same for outputs and locate other segments which use that output. A)_____ B)_____

Locate descriptions at three different hierarchical levels that address:

		FOUND/NOT FOUND
1)	The level that represents the entire document	_____
2)	Some intermediate level	_____
3)	A primitive level subfunction	_____

For each of the above which was "found":
(Par = Parent Inter = Intermediate Prim = Primitive)
(Answer Y/N/NA)

		Par	Inter	Prim
a)	Is the level of detail appropriate for the level in the hierarchy?	_____	_____	_____
b)	Do all pertinent interfaces appear to be documented?	_____	_____	_____
c)	Are interface elements indicated?	_____	_____	_____

Appendix G. METRICS SPECIFICATIONS

Each metrics computation is defined in the following equations. Note the following points with respect to these equations:

- All scales have a minimum value of 1 and a maximum value of 10. If their computations take them out of those bounds, they are given the extreme allowable value within that range. To increase readability, this fact is not stated in the following computations.

- Operands which are all capital letters denote a value which is computed over the global class of programs.

- A number of the metrics from the original specification involve subjective analysis. Such metrics cannot be calculated using an automated tool, and have therefore been ommitted from this project. The original numbering of the metrics, however, is kept for consistency. This explains the "missing" metrics numbers.

2. COUPLING

Objective: Reduce coupling

2.1. Number of global variables referenced

$$y = \frac{\text{number of global variables referenced}}{\text{number of global and local variables referenced}}$$

$$scale = 10 - 9y$$

2.2. Numbers of parameters passed

$$a = \frac{\text{total number or parameters}}{\text{total number of calls}}$$

if $a < 9$ $\quad z = 10 - a$
if $a \geq 9$ $\quad z = 1$

$$y = 5 + (z - 5) \times \frac{\text{total number of parameters}}{\text{number of global variables referenced}}$$

if number of calls $= 0$ $\quad scale = 7.5$
if number of calls $\neq 0$ $\quad scale = y$

2.3./2.4. Types of parameters passed

y = number of calls that have switches

if $y = 0$ $\quad scale = 10$
if $0 < y < 5$ $\quad scale = 6 - y$
if $5 \leq y$ $\quad scale = 1$

2.5. Number of entry points in a program

$scale = 10$

2.7. Number of structured data types passed as parameters

$$scale = 5 + \frac{\text{number of structured data type parameters}}{\text{total number of parameters}} \times 5$$

3. COHESION

Objective: Enhance cohesion

3.1. Number of control structures (CS)

$$\text{ave loc per CS} = \frac{\text{TOTAL LINES OF SOURCE CODE FOR ALL PROGRAMS}}{\text{TOTAL CS AT LEVEL 0 FOR ALL PROGRAMS}}$$

$$\text{expected CS at level 0} = \frac{\text{total loc}}{\text{ave loc per CS}}$$

$$\textit{scale} = \frac{\text{total loc enclosed by CS at level 0}}{\text{total loc}} \times 10 \times \frac{\text{expected CS at level 0}}{\text{number of CS at level 0}}$$

3.2. Division of code into logical units which perform single specific functions

$$\textit{scale} = 5 + \frac{\text{total loc enclosed by CS at level 0}}{\text{total loc}} \times 5$$

4. WELL-DEFINED INTERFACES

Objective: Improve software/software and software/hardware interfaces

4.1. Number of global variables

$$y = \frac{\text{number of unique global variables referenced}}{\text{number of unique global variables referenced + number of unique parameters}}$$

$$scale = 10 - 9y$$

4.2. Use of parameters in procedure calls

$$y = \frac{\text{number of unique parameters}}{\text{number of unique global variables referenced + number of unique parameters}}$$

$$scale = 10 - 9y$$

4.3. Use of parameterless procedure calls

$$y = \frac{\text{number of unique parameterless calls}}{\text{number of unique calls}}$$

$$scale = 10 - 9y$$

4.4. Use of excessive parameters

$$y = \frac{\text{total number of parameters}}{\text{total number of calls}}$$

if $y \leq 5$	$scale = 5$
if $5 < y < 10$	$scale = 5 - (y - 5)$
if $y \geq 10$	$scale = 1$

4.6. Number of data structures used with respect to intra-routine communication

$$y = \frac{\text{number of unique structured data type references}}{\text{number or unique variable references}}$$

if $y \geq 0.8$	$scale = 10$
if $0.5 < y < 0.8$	$scale = 5 + 6y$
if $0.2 \leq y \leq 0.5$	$scale = 10 - \frac{1.5}{y}$
if $y < 0.2$	$scale = 1$

5. COMPLEXITY

Objective: Reduce complexity

5.4. Length of procedures

$$\text{average procedure length} = \frac{\text{total lines of code}}{\text{total number of procedures}}$$

$y = 50/\text{average procedure length}$

if $y > 1$ $\quad scale = 5 + \frac{5}{y}$
if $y \leq 1$ $\quad scale = 5y$

5.5. Number of control structures used

Scale is computed as in 3.1

5.6. Maximum nesting level

Compute the following *for each* control structure at level 0:

if maximum nesting level ≤ 2 $\quad y = 10$
else

$$y = 10 - 10 \times (\text{maximum level} - 2) \times \frac{\text{loc enclosed at level 3}}{\text{loc enclosed at level 0}} \times \frac{\text{loc enclosed at level 0}}{\text{total loc}}$$

$$scale = \frac{\text{sum of every y}}{\text{number of control structures at level 0}}$$

5.7. Number of goto statements used

$y = 5 - 2 \times$ number of gotos

if number of gotos $= 0$ $\quad scale = 10$
if number of gotos $\neq 0$ $\quad scale = y$

5.8. Number of block comments

$$scale = 1 + \text{number of block comments} \times \frac{\text{number of block comment lines}}{\text{total number of comment lines}}$$

5.9. Number of single line comments

$$y = \frac{\text{number of single line comments}}{\text{total loc}}$$

$$scale = 10 - 20 \times (y - 0.2)$$

5.13. Number of structured data types used

Scale is computed as in 2.7

5.16. Number of if statements

$$y = \frac{\text{TOTAL LINES OF SOURCE CODE FOR ALL PROGRAMS}}{\text{TOTAL NUMBER OF IF STATEMENTS}}$$

$$z = \frac{\text{total loc}}{y}$$

if number of ifs $\leq$ z $scale = 10$
if number of ifs $>$ z $scale = 10 - 3 \times (\text{number of ifs} - z)$

6. READABILITY

Objective: Enhance readabilty

6.1. Use of control structures

$$z = \frac{\text{TOTAL LINES OF SOURCE CODE FOR ALL PROGRAMS}}{\text{TOTAL NUMBER OF CONTROL STRUCTURES}}$$

$$\text{expected number of control structures} = \frac{\text{total loc}}{z}$$

$$y = \frac{\text{expected number of control structures}}{\text{actual number of control structures}}$$

if $y \leq 0.5$	$scale = 1$
if $0.5 < y \leq 1.0$	$scale = 10 - (1 - y) \times 20$
if $1.0 < y \leq 1.5$	$scale = 10 - (y - 1) \times 20$
if $y > 1.5$	$scale = 1$

6.2. Use of goto statements

Scale is computed as in 5.7

6.4. Use of block header comments

Scale is computed as in 5.8

6.6. Use of single line comments

Scale is computed as in 5.9

6.10. Module length

Scale is computed as in 5.4

6.12. Use of symbolic constants (literals)

$$y = 8 - \frac{\text{number of symbolic constants referenced}}{2}$$

if number of symbolic constants referenced = 0 $\quad scale = 10$
if number of symbolic constants referenced $\neq$ 0 $\quad scale = y$

7. EXTENSIBLE SOFTWARE

Objective: Enhance extensibility

7.1. Use of global variables

Scale is computed as in 2.1

7.5. Use of distinct functions within a single module

Scale is computed as in 3.2

METRICS EXCEPTIONS

This appendix lists each denominator in the metrics equations (in Appendix G) and stipulates the action to take in the case that it is 0.

Coupling

Metric	Condition	Action
2.1	number of global and local variables = 0	scale = 10
2.2	number of global variables = 0	scale = 1
2.7	total number of parameters = 0 and number of sdt parameters = 0	scale = 5
	total number of parameters = 0 and number of sdt parameters ≠ 0	scale = 10

Cohesion

Metric	Condition	Action
3.1	TOTAL CS AT LEVEL 0 FOR ALL PROGRAMS = 0 number of CS at level 0 = 0	expected CS at level 0 = 0 scale = 5

Well-defined Interfaces

Metrics	Condition	Action
4.1	# of unique global vars + # of unique parameters = 0	scale = 10
4.2	# of unique global vars + # of unique parameters = 0	scale = 5
4.3	# of unique calls = 0	scale = 10
4.4	total # of calls = 0	scale = 10
4.6	# of unique variable references = 0	scale = 5

Complexity

Metric	Condition	Action
5.4	total number of procedures = 0	no action (impossible condition)
5.6	total lines of code = 0	no action (impossible condition)
	loc enclosed at level 0 = 0	scale = 10
	# of cs at level 0 = 0	scale = 10
5.8	total comment lines = 0	scale = 1
5.9	total loc = 0	no action (impossible condition)
5.16	TOTAL NUMBER OF IF STATEMENTS = 0	z = 0
	y = 0	no action (impossible condition)

Readability

Metric	Condition	Action
6.1	TOTAL NUMBER OF CS = 0	expected number = 0
	z = 0	no action (impossible condition)
	actual # of control structures = 0 and number of goto's = 0	scale = 10
	actual # of control structures = 0 and number of goto's ≠ 0	scale = 1

Appendix H. BIBLIOGRAPHY

ADDD85 Addleman, D.R., M.J. Davis and P.E. Presson (1985), "Specification Technology Guidebook," RADC Technical Report CDRL A003.

ALFM77 Alford, M.W. (1977), "A Requirements Engineering Methodology for Real-Time Processing Requirements," IEEE Transactions on Software Engineering, Vol. SE-3. No. 1, pp. 60-69.

ALFM85 Alford, M. (1985), "SREM at the Age of Eight: The Distributed Computing Design System," IEEE Computer, Vol.18, No. 4, pp. 36-46.

ARTJ85 Arthur, J.D., S.M. Henry and R.E. Nance (1985), "Immediate Software Development Issues for Embedded Systems Applications in Surface Combatants," Technical Report C8-85-19, Department of Computer Science, Virginia Tech, Blacksburg, Va 24061.

ARTJ86 Arthur, J.D., R.E. Nance and S.M. Henry (1986), "A Procedural Approach to Evaluating Software Development Methodologies: The Foundation," Technical Report C8-86-24, Department of Computer Science, Virginia Tech, Blacksburg, Va 24061.

ARTJ87 Arthur, J.D. and R.E. Nance (1987), "Developing an Automated Procedure for Evaluating Software Development Methodologies and Associated Products," Technical Report TR-87-16, Department of Computer Science, Virginia Tech, Blacksburg, Va 24061.

ARTL83 Arthur, L.J. (1983), Programmer Productivity, John Wiley, New York, NY.

ARTL85 Arthur, L.J. (1985), Measuring Programmer Productivity and Software Quality, John Wiley, New York, NY.

BAKF78 Baker, F.T. (1978), "Structured Programming in a Production Programming Environment," In: C.V. Ramamoorthy and R.T. Yeh (eds.), Tutorial: Software Methodology, IEEE Computer Society Press, Long Beach, CA.

BASV75 Basili, V.R. and A.J. Turner (1975), "Iterative Enhancement: A Practical Technique for Software Development," IEEE Transactions on Software Engineering, Vol. SE-1, No. 4, pp. 390-396.

BATD76 Bates, D. (ed.) (1976), Structured Programming, Infotech State of the Art Report, Infotech, England.

BATD77 Bates, D. (ed.) (1977), Software Engineering Techniques, Infotech State of the Art Report, Infotech, England.

BAUF72 Bauer, F. (1972), "Software Engineering," Information Processing 71, North Holland Publishing Co., Amsterdam, Holland.

BAUF75 Bauer, F.L. (ed) (1975), Software Engineering: An Advanced Course, Springer Verlag, New York, NY.

BELT77 Bell, T.E., D.E. Biller and M.E. Dyer (1977), "An Extendable Approach to Computer-Aided Software Requirements Engineering," IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, pp. 6-15.

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

BERG81a Bergland, G.D. and R.D. Gordon (1981), "Software Design Strategies," IEEE Compsac, pp. 79-92.

BERG81b Bergland, G.D. (1981), "Structured Design Methodologies," Proc. 15th Annual Design Automation Conference, pp. 475-493.

BOEB76 Boehm, B. (1976), "Software Engineering," IEEE Transactions on Computers, Vol. C-25, No. 12.

BOEB78 Boehm, B.W. et al. (1978), Characteristic of Software Quality, North-Holland Publishing Co., Amsterdam, Holland.

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

BOEC66 Boehm, C. and G. Jacopini (1966), "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," Communications of the ACM, Vol. 9, No. 5, pp. 366-371.

BOWT85 Bowen, T.P., G.B. Wigle and J.T. Tsai (1985), Specifications of Software Quality Attributes, RADC-TR-85-37, Vol I.

BROF74 Brooks, F.P. (1974), "The Mythical Man-Month," Datamation, Vol. 20, No. 12.

BULR76 Bullen, R.H. (1976), "Program Modularization," In: D. Bates (ed.), Structured Programming, Infotech State of the Art Report, U.K.

CAMJ83 Cameron,J.R. (1983), JSP & JSD: The Jackson Approach to Software Development, IEEE Computer Society Press.

CARD86 Card, D.N., V.E. Church and W.W. Agresti (1986), "An Empirical Study of Software Design Practices," IEEE Transactions of Software Engineering, Vol. 12, No. 2, pp. 115-139.

CAVW78 Cave, W.C. and A.B. Salisbury (1978), "Controlling the Software Life Cycle," IEEE Transactions on Software Engineering, Vol. SE-4. No. 4. pp. 326-334.

CAVW84 Cave, W.C. and G.W. Maymon (1984), Software Life-Cycle Management, Macmillan, New York, NY.

CONR76 Conway, R., D. Gries and E.C. Zimmerman (1976), A Primer on Pascal, Winthrop Computer Systems Series, Cambridge, MA.

CONS86 Conte, S.D., H.E. Dunsmore and V.Y. Shen (1986), Software Engineering Metrics and Modules, Benjamin Cummins, Menlo Park, CA.

CUTM83 Cutler, Mel (1983), "An Overview of VLSI Intersected with Software Engineering," IEEE Compcon Digest of Papers, pp. 260-262.

CUYR84 Cuykendall, R. et. al. (1984), "Design Synthesis in VLSI and Software Engineering," The Journal of Systems and Software, Vol. 4, No. 1, pp. 7-12.

DALN83 Dale, N. and D. Orshalick (1983), Introduction to Pascal and Structured Design, D.C. Heath & Co., Lexington, MA.

DAVC77 Davis, C. and C. Vick (1977), "The Software Development System," IEEE Transactions on Software Engineering, January 1977.

DeMT78 De Marco, T. (1978), Structured Analysis and System Specification, Yourdon Press, New York, NY.

DeMT79 De Marco, T. (1979), Concise Notes on Software Engineering, Yourdon Press, New York, NY.

DENJ73 Dennis, J.B., G. Goos, P.C. Poole et. al. (1973), Advanced Course of Software Engineering - Lecture Notes, Springer-Verlag, New York, NY.

DICM78 Dickover, M.E., C.L. McGowan and D.T. Ross (1978), "Software Design Using SADT," In: J. Hosier (ed.), Structured Analysis and Design, Infotech International Ltd., Vol. II.

DIEH84 Dietel, H.M. (1984), An Introduction to Operating System, Addison-Wesley, Reading, MA.

DIJE65 Dijkstra, E.W. (1965), "Programming Considered as a Human Activity," Proc. IFIP Congress, pp. 213-217.

DIJE68 Dijkstra, E.W. (1968), "Go To Statement Considered Harmful," Communications of the ACM, Vol. 11, No. 3, pp. 149-150.

DIJE72 Dijkstra, E.W., o.J. Dahl and C.A.R. Hoare (1972), Structured Programming, Academic Press, New York, NY.

DUNH78 Dunsmore, H.E. and J.D. Gannon (1978), "Programming Factors - Language Features that Help Explain Programming Complexity," Communications of the ACM, Vol. 21, pp. 554-560.

FAIR85 Fairley, R.F. (1985), Software Engineering Concepts, McGraw-Hill, New York, NY.

GANC77 Gane, C. and T. Sarson (1977), Structured Systems Analysis, Improved System Technologies, Inc., NY.

GILP83 Gilbert, P. (1983), Software Design and Development, Science Research Associates, Chicago, IL.

GILT77 Gilb, T. (1977), Software Metrics, Winthrop Publishers, Cambridge, MA.

GREM78 Greenberg, M.R. (1978), Applied Linear Programming, Academic Press, New York, NY.

GRIS78 Griffiths, S.N. (1978), "Design Methodologies - A Comparison," Structured Analysis and Design, Infotech International Limited, U.K., Vol. 11, pp. 133-166.

GROP83 Grogono, P. (1983), Programming in Pascal, Addison-Wesley, Reading, MA.

HALM77 Halstead, M. (1977), Elements of Software Science, Elsevier, New York, NY.

HAML78 Hammond, L.S., D.L. Murphy and M.K. Smith (1978), "A System for Analysis and Verification of Software Design," Proc. Compsac, Chicago, IL.

HAMM79 Hamilton, M. and S. Zeldin (1979), "The Relationship Between Design and Verification," The Journal of Systems and Software, Vol.1, pp. 29-56.

HANP77 Hansen, P.B. (1977), The Architecture of Concurrent Programs, Prentice-Hall, Englewood Cliffs, N.J.

HECH77 Hecht, H. (1977), "Measurement, Estimation, and Prediction of Software Reliability," In: Software Engineering Technology, Vol. 2, pp. 209-224, Infotech International, England.

HENK81 Heninger-Britton, K.L., R.A. Parker and D.L. Parnas (1981), "A Procedure for Designing Abstract Interfaces for Device Interface Modules," Proc. 5th IEEE International Conference on Software Engineering, pp. 195-204.

HENS81 Henry, S.M., D. Kafura and K. Harris (1981), "On the Relationships among Three Software Metrics," Proc. ACM Workshop/Symposium on Measurement and Evaluation of Software Quality, pp. 81-88.

HENS85 Henry, S.M., J.D.Arthur and R.E. Nance (1985), "A Procedural Approach to Evaluating Software Development Methodologies," Technical

Report C8-85-20, Department of Computer Science, Virginia Tech, Blacksburg, Va.

HIRS84 Hirschhorn S. and A.M. Davis (1984), "Parallels between Software and VLSI Engineering," The Journal of Systems and Software, Vol. 4, No. 1, pp. 27-37.

HOFG85 Hoffnagle, G.F. and W.E. Beregi (1985), "Automating the Software Development Process," IBM Systems Journal, Vol. 24, No. 2, pp. 102-120.

HORE86 Horowitz, E. and R.C. Williamson (1986), "SODOS: A Software Documentation Support Environment - Its Definition," IEEE Transactions on Software Engineering, Vol. SE-12, No. 8, pp. 849-859.

HOSJ78 Hosier, J. (1978), Structured Analysis and Design, Infotech International Ltd., England.

JACM75 Jackson, M.A. (1975), Principles of Program Design, Academic Press, NY.

JACM76 Jackson, M.A. (1976), "Data Structure as a Basis for Program Design," In: D. Bates (ed), Structured Programming, Infotech, England.

JOHR76 Johnson, R.D., and B.R. Siskim (1976), Quantitative Techniques for Business Decisions, Prentice-Hall, Englewood Cliffs, NJ.

KATR84 Katz, R., W. Scacchi and P. Subramanyam (1984), "Environments for VLSI and Software Engineering," The Journal of Systems and Software, Vol. 4, No. 1, pp. 13-26.

KAYA77 Kay, A. and A. Goldgerg (1977), "Personal Dynamic Media," IEEE Computer, Vol. 10, No. 3, pp. 31-41.

KERB78 Kerninghan, B.W. and P.J. Plauger (1978), The Elements of Programming Style, McGraw-Hill, New York, NY.

KERB81 Kerninghan, B.W. and P.J. Plauger (1981), Software Tools in Pascal, Addison-Wesley, Reading, MA.

KOSD74 Kosy, D. (1974), "AF Command and Control Information Processing in the 1980s: Trends in Software Technology," RAND R-1012-PR.

LANR81 Lanergan, R.G. and D.K. Dugan (1981), "Software Engineering with Reusable Designs and Code," IEEE Compcon Digest of Papers, Fall Conference, pp. 296-303.

LISB76 Liskov, B.H. (1976), "A Design Methodology for Reliable Software Systems," In: P. Freeman and A.I. Wasserman (eds.), Tutorial on Software Design Techniques, IEEE Computer Society Press, Long Beach, CA.

LOHJ84 Lohse, J.B. and S.H. Zweben (1984), "Experimental Evaluation of Software Design Principles: An Investigation into the Effect of Module Cou-

pling on the System and Modifiability," Journal of Systems and Software, Vol. 4, pp. 301-308.

MARD84 Marca, David (1984), Applying Software Engineering Principles, Little, Brown and Co., Boston, MA.

McCC75 McClure, C.L. (1975), "Top-Down, Bottom-Up, and Structured Programming," IEEE Transactions on Software Engineering, Vol. SE-1, No. 4, pp. 397-403.

McCD76 McCracken, D.D. (1976), A Simplified Guide to Structured COBOL Programming, John Wiley, NY.

McCJ77 McCall, J.A., P.K. Richards and G.F. Walters (1977), "Factors in Software Quality," RADC TR-77-369, Vol. I.

McCJ80 McCall, J.A. and M.T. Matsumoto (1980), "Software Quality Measurement Manual," RADC-TR-80-109, Vol. I and II.

McCT76 McCabe, T.J. (1976), "A Complexity Measure," IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, pp. 308-320.

McTJ80 McTap, J.L. (1980), "The Complexity of an Individual Program," Proc. AFIPS National Computer Conference, pp. 767-771.

MILG56 Miller, G.A. (1956), "The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information," Psychological Review, Vol. 63, pp. 81-97.

MYEG75 Myers, G. (1975), Reliable Software Through Composite Design, Petrocelli/Charter, New York, NY.

MUSJ84 Musa, J.D. (1984), "Software Reliability," In: C.R. Vick and C.V. Ramamoorthy (editors), Software Engineering Handbook, Van Nostrand Rheinhold Co., New York.

NANR85 Nance, R.E., J.D. Arthur and A.V. Dandekar (1985), "Evaluation of Software Development Methodologies," A Final Report of the Intermediate Software Development Project, The Department of Computer Sciences, Virginia Tech, Blacksburg, Va 24060.

NAUP69 Naur, P. and B. Randell (eds.) (1969), Software Engineering, NATO Science Committee Report, Brussels, Belgium.

OSTL79 Osterweil, L.J., J.R. Brown and L.G. Stucki (1979), "ASSET: A Life Cycle Verification and Visibility System," The Journal of Systems and Software, Vol. 1, No. 1, pp. 77-86.

PAGM80 Page-Jones, M. (1980), The Practical Guide to Structured Design, Yourdon Press, New York, NY.

PARC87 Parker, C.S. (1987), Understanding Computers and Data Processing: Today and Tomorrow, HRW Publishers, New York, NY.

PARD71 Parnas, D.L. (1971), "Information Distribution Aspects of Design Methodology," Proc. IFIP Congress 71, Vol. 1, pp. 339-344, North-Holland Publishing Co.

PARD72 Parnas, D.L. (1972), "A Technique for Software Module Specification with Examples," Communications of the ACM, Vol. 15, No. 5, pp. 330-336.

PARD72a Parnas, D.L. (1972), "On the Criteria to be Used in Decomposing Modules," Communications of the ACM, Vol. 15, No. 12, pp. 1053-1058.

PARD74 Parnas, David (1974), "On A Buzzword: Hierarchical Structure," Proc. IFIP Congress, pp. 336-339.

PARD79 Parnas, D.L. (1979), "Designing Software for Ease of Extension and Contraction," IEEE Transactions on Software Engineering, No. 3, pp. 184-196.

PERE85 Perry, E. (1985), Systems Analysis, Design and Development, HRW Publishers, New York, NY.

PETL77 Peters, L.J. and L.L. Tripp (1977), "Comparing Software Design Methodologies," Datamation, Vol. 23, N0. 11, pp. 89-94.

PETL81 Peters, L.J. (1981), Software Design: Methods and Techniques, Yourdon Press, New York, NY.

POOP72 Poole, P.C. and W.M. Waite (1972), "Portability and Adaptability," In: M. Beckman et al. (eds.) Advanced Course on Software Engineering, Springer Verlag.

PRAT84 Pratt, T.W. (1984), Programming Languages - Design and Implementation, Prentice-Hall, Englewood Cliffs, NJ.

PRER82 Pressman, R.S. (1982), Software Engineering A Practitioner's Approach, McGraw-Hill, New York, NY.

RADJ84 Rader, J. (1984), "VLSI and Software Engineering: Guest Editor's Introduction," The Journal of Systems and Software, Vol. 4, No. 1, pp. 3-6.

RAJV85 Rajlich, V. (1985), "Stepwise Refinement Revisited," The Journal of Systems and Software, Vol. 5, No. 1, pp. 81-88.

RAMC78 Ramamoorthy, C.V. and Raymond T. Yeh (1978), "Introduction," In: C.V. Ramamoorthy and R.T. Yeh (editors), Tutorial: Software Methodology, IEEE Computer Society, Compsac.

RAMC84 Ramamoorthy, C.V., A. Prakash, W. Tsai and Y. Usuda (1984), "Software Engineering: Problems and Perspective," IEEE Computer, Vol. 17, No. 10, pp. 191-210.

ROMG85 Roman, G. (1985), "A Taxonomy of Current Issues in Requirements Engineering," IEEE Computer, Vol.18, No. 4, pp. 14-23.

ROSD75 Ross, D.T., J.B. Goodenough and C.A. Irvine (1975), "Software Engineering: Process, Principles, and Goals," IEEE Computer, Vol. 8, No. 5, pp. 17-27.

ROSD77 Ross, D.T. (1977), "Structured Analysis (SA): A Language for Communicating Ideas," IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, pp. 16-34.

ROSD77a Ross, D.T. and K.E. Schoman (1977), "Structured Analysis for Requirements Definition," IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, pp. 69-84.

ROSD79 Ross, D.T., G.l. McGowan and M.E. Dickover (1979), "Software Design Using SADT," In: G.D. Bergland and R.D. Gordon (editors), Tutorial: Software Design Strategies, IEEE Computer Society, Compsac.

RZEW85 Rzepka, W. and Y. Ohno (1985), "Requirements Engineering Environments: Software Tools for Modelling User Needs," IEEE Computer, Vol.18, No. 4, pp. 9-13.

SCHP85 Scheffer, P.A., A.H. Stone and W.E. Rzepka (1985), "A Case Study of SREM," IEEE Computer, Vol.18, No. 4, pp. 47-54.

SHNB77 Shneiderman, B. (1977), "Measuring Computer Program Quality and Comprehension," International Journal of Man-Machine Studies, Vol. 9, No. 4, pp. 465-478.

SHNB80 Shneiderman, Ben (1980), Software Psychology, Winthrop Publishers, Cambridge, MA.

SMIC83 Smith, C.U. and J.A. Dallen (1983), "A Comparison of Design Strategies for Software and VLSI," IEEE Spring Compcon Digest of Papers, pp. 263-268.

STEM78 Stern, M. (1978), "Some Experience in Building Portable Software," Proc. International Conference on Software Engineering, pp. 327-332, May 10-12, Atlanta, GA.

STEW74 Stevens, W.P., G.J. Myers and L.L. Constantine (1974), "Structured Design," IBM Systems Journal, Vol. 13, No. 2, pp. 115-139.

STEW81 Stevens, W.P. (1981), Using Structured Design, John Wiley, New York, NY.

TAUR77 Tausworthe, R. C. (1977), Standardized Development of Computer Software, Prentice-Hall, Englewood Cliffs, NJ.

TEID74 Teichroew, D. (1974), "Improvements in the System Life Cycle," Proc. IFIPS Congress, pp. 972-978.

TEIW81 Teilman, W. and L. Masinter (1981), "The INTERLISP Programming Environment," IEEE Computer, Vol. 14, No. 4, pp. 25-33.

TROD81 Troy, D.A. and S.H. Zweben (1981), "Measuring the Quality of Structured Design," The Journal of Systems and Software, Vol. 2, pp. 113-120.

WAHB84 Wah, B.W. and C.V. Ramamoorthy (1984), "Theory of Algorithms and Computation Complexity with Applications to Software Design," In: C.R. Vick and C.V. Ramamoorthy (eds.), Software Engineering Handbook, van Nostrand Rheinhold Co., NY.

WARJ74 Warnier, J.D. (1974), Logical Construction of Programs, H.F. Stenfert Kroese B.V.,Leiden, Holland.

WASA75 Wasserman, A.I. (1975), "Some Principles of User Software Engineering for Information Systems," Digest of Papers, COMPCON 75 Spring, IEEE Computer Society, pp. 49-52.

WASA76 Wasserman, A.I. (1976), "On the Meaning of Discipline in Software Design and Development," In: P. Freeman and A.I. Wasserman (eds), Tutorial on Software Design Techniques, IEEE Computer Society, Long Beach, CA.

WASA82 Wasserman, A.I. (1982), "The User Software Engineering Methodology: An overview,' Proc. IFIP WG 8.1 Working Conference on Comparative Review of Information Systems Design, Noordwijkerhout, The Netherlands, pp. 591-628.

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

WEIL74 Weissman, L. (1974), "Psychological Complexity of Computer Programs: An Experimental Methodology, ACM SIGPLAN Notices, N0. 6, pp. 25-36.

YOUE79 Yourdon, E. and L.L. Constantine (1979), Structured Design, Prentice Hall, New Jersey.

ZOLJ81 Zolnowski, J.C. and D.B. Simmons (1981), "Taking the Measure of Programming Complexity," Proc. AFIPS National Computer Conference, pp 329-336.

The vita has been removed
from the scanned document