Defining Software Quality Measures: A Systematic Approach Embedded in the Objectives, Principles, Attributes Framework

Gary N. Bundy and James D. Arthur

TR 92-38

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

July 29, 1992

Defining Software Quality Measures: A Systematic Approach Embedded in the Objectives, Principles, Attributes Framework

Gary N. Bundy
The MITRE Corporation
7525 Colshire Drive, MS: Z645
McLean, VA 22102
USA
(703) 883-5560

James D. Arthur
The Department of Computer Science
Virginia Tech
Blacksburg, VA 24061
USA
(703) 231-7538

Abstract

Currently, software quality measures and metrics are being developed in isolation and often without the benefit of a guiding framework. In this paper we describe a systematic process for identifying measurement approaches and defining corresponding metrics that definitively support software quality assessment. That systematic process embodies five well-defined steps that reflect quality assessment within a framework which links the achievement of desirable software engineering objectives to the use of appropriate principles, and the use of principles to the manifestation of desirable product attributes. Ada is the language we have chosen to examine; the Ada package is used to illustrate the identification and definition process.

Keywords: Metric Development, Metrics, Measures, Software Quality Assessment, Indicators, OPA Framework, Assessment Procedure

Category: Techniques and Methods

1 Introduction

In the late 1800s Lord Kelvin recognized the crucial role of metrics in the management process. Stated in paraphrased form, his contention is that "you cannot manage what you cannot measure." Today, metric-based analysis appears to be the most promising approach to controlling the software development process and producing a quality product. Some of the more well-known metrics include Halstead's Software Science Measures [HALM77], McCabe's Cyclomatic Number [MCCT76], and Henry and Kafura's Information Flow Metric [HENS81]. Unfortunately, many metrics currently in use are non-intuitive, non-instructive, and lack that fundamental basis for understanding the implications of one measurement value as compared to another [KEAJ86, EJIL87]. While we too advocate use of metrics, we do so with a fundamental belief that metrics must be developed according to

(1) a systematic procedure that stresses the the identification of proper measurement approaches and the corresponding definition of valid metrics, within

(2) a guiding framework that embraces intuitive measures and provides a basis for reasoning about the implication and ramifications of those measures grounded in software engineering concepts.

In this paper we outline the Objectives, Principles, Attributes (OPA) framework for software quality assessment. The OPA framework defines a set of linkages which relate the achievement of software engineering objectives to the use of principles, and the use of principles to the presence or absence of desirable attributes. Code properties are used to measure the extent to which attributes are either present or absent in the code. Propagating computations along the sets of defined linkages provides measures of principle usages and an indication of the extent to which software engineering objectives have been achieved during the development process.

Following the OPA framework presentation we provide a detailed description of a five-step process for developing software quality measures and metrics within that framework. Relative to our discussions, Ada is assumed to be the implementation language. Accordingly, we describe the first step as it applies to the entire Ada language; steps two through five, however, are discussed in relation to one particular Ada language construct, the package. Although many other Ada constructs have been considered in our work, e.g. loops, conditionals, tasks and generics, length restrictions prevent their discussion.

In the last section of this paper we describe to what extent the metric definition process has been utilized and the impact it is now having on our current research efforts.

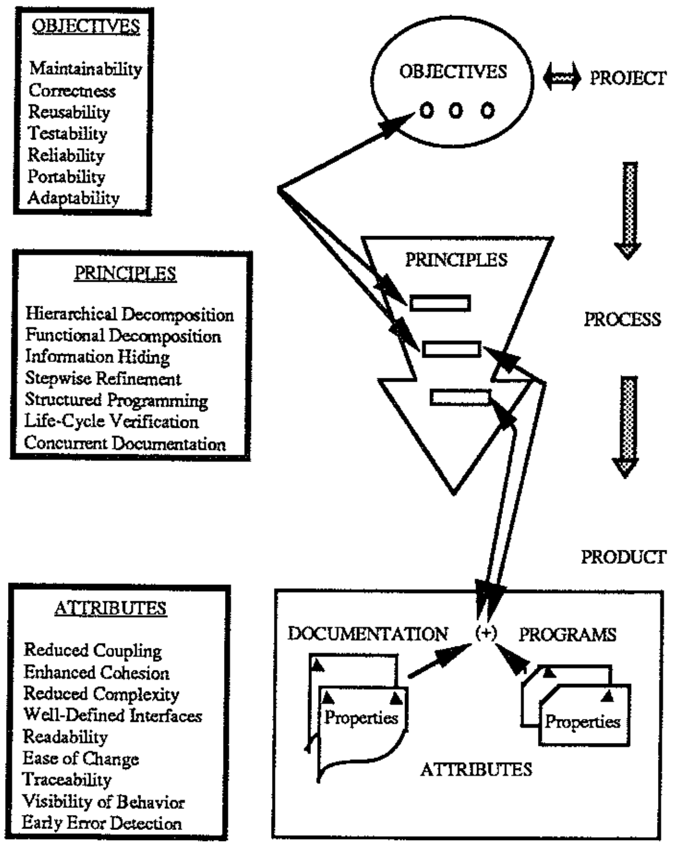

Figure 1
Illustration of the Relationship Among Objectives, Principles, Attributes in the Software Development Proces

2 Objectives, Principles, Attributes Framework

The Objective/Principles/Attributes (OPA) framework [ARTJ90] characterizes the *raision d'etre* for software engineering; that is, it embodies the rationale and justification for software engineering. As illustrated in Figure 1, the framework enunciates definitive linkages among project level objectives, software engineering principles, and desirable product attributes. In particular, it advances the following rationale for software development:

- a set of objectives can be defined that correspond to project level goals and objectives,
- achieving those objectives requires adherence to certain principles that characterize the process by which the product is developed, and
- adherence to a process governed by those principles should result in a product that possesses attributes considered to be desirable and beneficial.

Underlying this rationale is a natural set of relations, depicted in Figure 2, that link individual objectives to one or

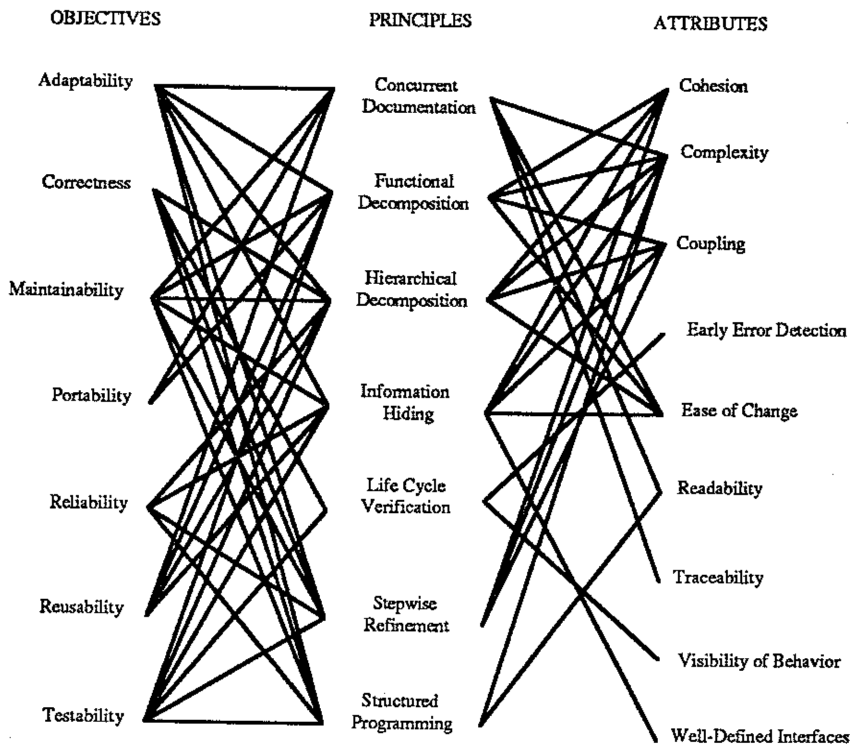

Figure 2
Linkages Among the Objectives, Principles and Attributes

more principles, and each principle to one or more attributes. For example, to achieve maintainability one might employ the principle of information hiding in the development process. In turn, employing information hiding will result in a product that exhibits a well-defined interface.

A natural question at this point is: *How does one determine if, and to what extent, a product possesses desirable attributes?* The answer lies in the observation of product properties, i.e. observable characteristics of product. For example, the use of global variables indicates that a module's interface is not necessarily well-defined [DUNH80]. The number of global variables used relative to preferable forms of communications, e.g. parameter passing, indicates the extent to which the interface is ill-defined.

In "bottom-line" parlance: (1) the achievement of software engineering objectives is directly linked to the use of specific principles, (2) as a consequence of using these principles desirable attributes are induced in the product, (3) by observing product properties to determine the extent to which attributes really do exist in the product, one can ascertain the extent to which particular principles are governing the development process, and in turn, the extent to which stated software engineering objectives are achieved.

Effectively, through its property/attribute pairs and linkages relating attributes to principles and principles to objectives, the OPA framework supports a well-defined, systematic approach to examining product and process quality. To date, we have defined and substantiated (through published results of independent researchers) 33 linkages among the seven prominent software engineering objectives and nine principles, 24 linkages among principles and attributes, and 84 property/attribute pairs [ARTJ87].

3 The Five Steps to Metric Development

The OPA framework defines a set of linkages that relate the achievement of software engineering objectives to the use of principles, and the use of principles to the presence or absence of desirable product attributes. Nonetheless, that framework provides only the basis and rationale for metric development. The five steps outlined below, when applied within the OPA framework, leads one to the realization of viable measurement approaches, and subsequently, to the definition of suitable metrics.

Step 1. *Identifying, Categorizing, and Classifying Crucial Language Components:* The categorization and classification of individual language components supports and encourages independent analysis with respect to each component.

Step 2. *Understanding the Rationale for Component Inclusion:* Language component rationales often motivate the necessity for including the component in the language definition as well as provide insight into the theoretical uses of a component.

Step 3. *Assessing Component Importance from a Software Engineering Perspective:* From a software engineering perspective, the impact of using a particular language construct or component is significant within the OPA framework. Such information, found in language rationales and language reviews from literature, provides insight into a component's contribution to achieving software engineering objectives, supporting accepted software engineering principles and inducing desirable product attributes.

Step 4. *Identifying the Impact of Component Usage on Desirable Software Engineering Attributes:* This forth step entails the identification of all possible uses (abuses) for each language component and recognition of why and how such uses (abuses) impact product attributes.

Step 5. *Identifying Properties, Defining Indicators, and Formulating Measures and Metrics:* The activities in this final step is to determine the various uses (abuses) of each code component, formally link the code component use (e.g. properties) to the impacted attribute(s), and then define a metric whose value reflects the justification linking each property/attribute pair.

The remaining subsection provide a detail description of the five step process as applied to the Ada language. For purposes of illustration and brevity, steps two through five are discussed relative a single virtue of only one Ada component: the package.

3.1 Identifying, Categorizing and Classifying Crucial Language Components

The first step in defining an OPA-based procedure for assessing the quality of an Ada-based product is to identify those language components deemed necessary and crucial to the assessment process. Such first steps often involve a categorization scheme that permits a language to be analyzed at the individual component level and then to be viewed from analytical perspectives based on aggregated components. In concert with this approach, the initial categorization scheme employs partitioning criteria proposed by Ghezzi and Jazayeri [GHEC82]; that is, the partitioning of language components along specific functional boundaries. In particular, an Ada program can be viewed as possessing data types, statement level control structures, and unit level control structures.

Based on suggestions of Wichmann [WICB84b], Ada language constructs can be further partitioned relative to constructs defined in Pascal. That is, within functional boundaries an Ada construct can be further delineated based on whether it has a Pascal counterpart; and if not, whether it can be easily added to Pascal, or represents a new language feature having a significant influence on the language design issues. Data aggregates, user-defined types, looping, and decision constructs are members of the first set. Partial array assignment, exit statements, and named loops are representatives of the second set. Packages, generics, tasking, and exception handling are each members of the third set. The Pascal oriented categorization is particularly significant because it allows extension of previous research results reported by Farnan [FARM87] and Dandekar [DANA87], so that research focuses on those language constructs and semantic components found in the Ada language but not in Pascal.

Assuming that Farnan and Dandekar have necessarily and sufficiently analyzed conventional language constructs, the critical Ada language constructs requiring additional examination are:

- Data Types
  - Strings
  - Record Discriminants

- Statement Level Control Structures
  - Partial Array Assignments
  - Exit Statements

- Named Loops with Exits
- Block Structures

• Unit Level Control Structures
  - Subprograms
    - Default Parameters, Name Overloading, Parameter Passing
  - Packages
    - Specification
    - Body
  - Generics
  - Tasking
    - Concurrency Specification
  - Exception Handling.

We recognize that the above categorization does not cover all Ada specific language components, but stress that the intentions are to examine only those that are most prominent from a software engineering perspective. Bundy [BUNG90] offers a more detailed explanation of identifying, categorizing, and classifying Ada language constructs with respect to software quality assessment within the OPA framework.

3.2 Understanding the Rationale for Component Inclusion

Before employing code structure analysis as part of a software quality procedure, one must acquire a firm understanding of why particular language constructs have been included in a language definition. In some cases, the rationale might simply be that a specific capability is needed, e.g., looping. From the perspectives of software engineering and software quality assessment, however, of particular interest is the rationale for including constructs like generics, packages, and block structures that are purported to support desirable product design and development capabilities. For Ada, the language designers have provided the *Rationale for the Design of the Ada Programming Language* [ADAR84]. Published papers describing research and development efforts and books describing usage techniques provide additional insights into the proposed uses of Ada language components. Using packages as a representative example, the next paragraph outlines the type of information the authors have sought in synthesizing an adequate understanding for including particular language elements in the definition of Ada.

According to [ADAR84] packages are one mechanism through which the programmer can group constants, type declarations, variables, and/or subprograms. The intent is that the programmer will use packages to group related items. From a software engineering perspective, this particular use of packages is appealing because it promotes code cohesion [ROSD86]. Packages are also a powerful tool in supporting the specification of abstractions. The ability to localize implementation details and to group related collections of information is a prerequisite for defining abstract data types in a language. Again, from a software engineering perspective, the capability to specify abstract data types and to force the use of predefined operations to modify data structures promotes reliability, portability, and maintainability.

3.3 Assessing Component Importance from a Software Engineering Perspective

To exploit the OPA framework one must determine each individual component's contribution to the achievement of desirable software engineering objectives, its support in the use of accepted software engineering principles, and/or its ability to impart desirable software engineering attributes to the encompassing product. The authors note that the impact of a component on product quality can be beneficial or detrimental. For example, operator overloading generally enhances program readability [WICB84a, GHEC82]. If used indiscriminately, however, it can have the opposite effect [GHEC82].

From an Ada standpoint, the literature abounds with citations attesting to the "software engineering goodness" of Ada language constructs. In particular, Ada packages are extremely important in achieving a quality, software engineered product. Ada packages support four definitional abstractions: named collections of declarations, subroutine libraries, abstract state machines, and abstract data types. One particular abstraction, abstract data types, is fundamental to supporting the software engineering principle of information hiding [ADAR84]. That is, packages defining abstract data types provide the type declaration for an abstract data type and methods for manipulating the data type. What is hidden from the user is the sequence of coded instructions supporting the manipulative operations. Also, the user is forced to modify the abstract data type through the specified operations. This form of information hiding is particularly beneficial when maintenance is required because it tends to minimize the "ripple effect" that change can have. As also discussed by Booch [BOOG83, BOOG87], packages are crucial in supporting modularity, localization, reusability, and portability, all of which are highly desirable from a software engineering perspective.

3.4 Identifying the Impact of Component Usage on Desirable Software Engineering Attributes

In the third step described above language components are associated with rather abstract software engineering qualities like maintainability, reliability, information hiding, and modularity. To implement an assessment procedure within the OPA framework, however, those language components must be aligned with less abstract entities, i.e. the software engineering attributes. This fourth step in the metric development process is crucial in that it establishes such linkages by identifying the impact(s) of each language construct on one or more (less abstract) software engineering attribute. This fourth step is illustrated below by considering the impact of packages relative to selected software engineering attributes.

As a basis, the authors examined the four proposed uses of packages in linking package properties to software engineering attributes. For example, packages that contain only type declarations indicate code cohesion [ROSD86]. The other three proposed uses are packages to define abstract data types, packages to define abstract state machines and packages to define subprogram units. Although all four of these uses induce desirable attributes in the developed product (see [GANJ86, EMBD88, BOOG87], respectively), improper use of packages can also have a negative impact on the desirable product attributes. For example, the use of packages to group type declarations has diminishing returns when too many type declarations are exported. This misuse hinders ease of change because program units must be unnecessarily checked for possible impacts caused by changes to declaration packages.

Consider as a detailed illustration of the above, the use of packages to define abstract data types (the authors will refer to such packages at ADT packages). The benefits (relative to the inducement of desirable software engineering attributes) of ADT packages are enhanced cohesion (functional and logical), a well-defined interface to the ADT, and enhanced ease of change for program units "withing" the ADT package. The improved cohesion results from the grouping of the ADT declarations and access operations within one package. A well-defined ADT interface is achieved by using the package specification to house the subprogram specification for each ADT and then using private or limited private types to restrict access to the ADT. From a different perspective, because of the capabilities provided by packages, the use of ADTs has additional beneficial effects in terms of reduced code complexity and improved readability. Without further elaboration, it suffices to say that the definition of ADTs through packages embraces the use of abstractions that hide superfluous details from the ADT user.

In considering packages relative to their impact on product attributes, the authors have also examined several other uses (and misuses) of packages. They include (but are not limited to)

- "excessive" number of declarations in a declaration package,
- program unit access to declaration packages,
- "excessive" number of subprograms in a package,
- defining abstract state machines via packages,
- the use of packages to define collections of global variables, and
- the impact of unused packages.

3.5 Identifying Properties, Defining Indicators, and Formulating Measures and Metrics

The fourth step of the metric development procedure describes the impact that component uses and abuses have on the software engineering attributes. Step 5 identifies and formally links product properties (language elements) to software engineering attributes. Because each identified property undeniably reflects either the presence or absence of a specified attribute, the authors refer to the property/attribute pair as an indicator. Building on the relationship between the property/attribute pair, a measurement approach and supporting metric is defined. These three activities are being discussed together, as a single step, because they are intrinsically tied together. To illustrate Step 5 of the metric development procedure the remainder of this section focuses on the identification of properties indicative of the presence of the attribute cohesion relative *to the use of packages in defining groups of subprograms.*

To begin the process one identifies those properties associated with the use of packages to define subprogram units and the attribute(s) that usage affects. In the cohesion example, the task is to identify characteristics that a cohesive package would exhibit. One such characteristic is the utilization of subprograms defined within a package. In particular, each program unit that "withs" the package of subprograms utilizes a percentage of the subprograms. A very low utilization suggests that the subprograms grouped by the package are not as closely related (or functionally cohesive) as they should be. A very high utilization suggests that the subprograms are closely related or functionally cohesive.

The description presented in the previous paragraph suggests the identification of a property, the establishment of a link between a particular property and attribute, a measurement approach and a supporting metric. In particular, the property/attribute indicator is the "definition of packages that export subprograms relative to its positive impact on code cohesion." Hence, to effectively measure the cohesiveness of packages that export subprograms, one must examine the utilization of the subprograms by "withing" units. Intuitively, if the subprograms are sufficiently related, any unit that "withs" the package will use a majority of the subprograms. The indicative metric, calculated on a per package basis, is given with the following formula:

$$\text{Sub Package Utilization} = \frac{\displaystyle\sum_{\substack{\text{"Withs"}\\\text{to a Sub}\\\text{Package}}} \text{package subprograms referenced}}{(\text{total \# of "withs"}) \;*\; (\text{\# of subprograms in the package specification})}$$

(Note: Sub Package refers to a package that exports subprograms)

The analysis of packages that define abstract data types and of packages that define abstract state machines provides similar results. The current working list of property/attribute indicators for Ada packages is:

1. Definition of Declaration Packages
   - Cohesion (+)
   - Ease of Change (+)
2. Insufficient Decomposition of Declaration Packages
   - Ease of Change (-)
3. Definition of Packages that Export Subprograms
   - Cohesion (+)
   - Ease of Change (+)
   - Well-Defined Interface (+)
4. Units which "with" Packages that export Subprograms
   - Complexity (+)
   - Readability (+)
5. Definition of Packages that are never "withed"
   - Complexity (-)

For further details on the indicators and metrics for Ada packages (and all Ada components) see [BUNDG90].

4 Summary and Current Research

The Objectives, Principles, Attributes framework provides a formal basis for defining a software quality assessment procedure. The five-step procedure outlined above provides a guidance that enables one to identify and characterize the beneficial (or detrimental) impact that the use of a language construct can have on a product. Because this procedure relates such properties to software engineering attributes, a natural, complementary link to the OPA framework is established. Together, they embody the guidelines and techniques for identifying alternative measurement approaches and the definition of metrics that measure what is intended.

Using the five-step procedure we have currently identified 66 automatable metrics: eight are based on data type information, 12 exploit properties of statement level constructs, and 46 reflect assessment of unit level constructs like packages, tasks, subprograms and so forth. A code analyzer has been built and is currently being used to validate the 66 metric relative to their ability to assess product quality.

We also have two other research efforts that use procedures similar to the one presented here. The focal points of these efforts, however, have been to identify and formulate document quality metrics and process metrics. Currently, thirty-two document quality measures form a basis for assessing the accuracy, completeness and usability of documentation. Of these thirty-two, ten have been automated. Nine process metrics that span the software development life-cycle are now being investigated. In each case, a step-based procedure defined relative to the OPA framework has been instrumental in the identification and definition of such metrics.

References

[ADAR84] *Rationale for the Design of the Ada Programming Language*, Minneapolis, MN: Honeywell Systems and Research Center, 1984.

[ARTJ90] Arthur, J.D. and R.E. Nance, "A Framework for Assessing the Adequacy and Effectiveness of Software Development Methodologies," *Proceedings of the Fifteenth Annual Software Engineering Workshop*, Process Improvement Session, Greenbelt MD, December 1990.

[ARTJ87] Arthur, James D. and Richard E. Nance, "Developing an Automated Procedure For Evaluating Software Development Methodologies and Associated Products," Technical Report SRC-87-007, Systems Research Center and Department of Computer Science, Virginia Tech, 1987.

[BUNG90] Bundy, Gary N., "The Objectives, Principles, Attributes Approach for Measuring Software Quality in Ada Based Products," M.S. Thesis, Computer Science Department, Virginia Polytechnic Institute and State University, July, 1990.

[BOOG83] Booch, Grady, *Software Engineering with Ada*, Menlo Park, CA: The Benjamin/Cummings Publishing Company, 1983.

[BOOG87] Booch, Grady, *Software Components with Ada*, Menlo Park, CA: The Benjamin/Cummings Publishing Company, 1987.

[DANA87] Dandekar, Ashok V., "A Procedural Approach to the Evaluation of software Development Methodologies," M.S. Thesis, Computer Science Department, Virginia Polytechnic Institute and State University, September, 1987.

[DUNH80] Dunsmore, H.E. and J.D. Gannon, "Analysis of the Effects of Programming Factors on Programming Effort," *Journal of Systems and Software*, Vol. 1, No. 2, February 1980, pp. 141-153.

[EMBD88] Embley, David W. and Scott N. Woodfield, "Assessing the Quality of Abstract Data Types Written in Ada," *Proceedings: 10th International Conference on Software Engineering*, April 1988, pp. 144-153.

[EJIL87] Ejiogu, LEM O., "The Critical Issues of Software Metrics--Part 0. Perspectives on Software Measurements," *SIGPLAN Notices*, Vol. 22, No. 3, March 1987, pp. 59-64.

[FARM87] Farnan, Mark A., "The Automation of a Set of Code Metrics for Pascal," M.S. Project, Computer Science Department, Virginia Polytechnic Institute and State University, September, 1987.

[GANJ86] Gannon, J. D., E. E. Katz, and V. R. Basili, "Metrics for Ada Packages: An Initial Study," *Communications of the ACM*, Vol. 29, No. 7, July 1986, pp. 616-623.

[GHEC82] Ghezzi, C. and Mehdi Jazayeri, *Programming Language Concepts*, New York, John Wiley & Sons, Inc., 1982.

[HALM77] Halstead, Maurice H., *Elements of Software Science*, New York: Elsevier North-Holland, Inc., 1977.

[HAMC85] Hammons, Charles and Paul Dobbs, "Coupling, Cohesion, and Package Unity in Ada," *Ada Letters*, Vol. 4, No. 6, May/June 1985, pp. 49-59.

[HENS81] Henry, Sallie and Dennis Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering*, Vol. 7, No. 5, September 1981, pp. 510-518.

[KEAJ86] Kearney, Joseph K., et al., "Software Complexity Measurement," *Communications of the ACM*, Vol. 29, No. 11, November 1986, pp. 1044-1050.

[MCCT76] McCabe, Thomas J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, December 1976, pp. 308-320.

[ROSD86] Ross, Donald L., "Classifying Ada Packages," *Ada Letters*, Vol. 6, No. 4, July/August 1986 , pp. 53-65.

[WICB84a] Wichmann, B. A., "Is Ada too Big? A Designer Answers the Critics," *Communications of the ACM*, Vol. 27, No. 2, February 1984, pp. 98-103.

[WICB84b] Wichmann, B. A., "A Comparison of Pascal and Ada," *Comparing and Assessing Programming Languages*, Englewood Cliffs, NJ: Prentice-Hall Inc., 1984.