Measuring Software Quality in ADA Packages: An Objectives, Principles, Attributes Framework

By Gary N. Bundy and James D. Arthur

TR 90-42

Measuring Software Quality in Ada Packages: An Objectives, Principles, Attributes Framework

Gary N. Bundy and James D. Arthur

The Department of Computer Sciences
Virginia Tech
Blacksburg, VA 24061

Abstract

This paper describes preliminary results stemming from a research effort focusing on assessing the quality of an Ada-based product. The presentation emphasizes

(1) the formulation of a metric development procedure that starts with the identification of crucial Ada constructs and terminates with the formulation of metrics that support software quality assessment based on the usage of those crucial elements, and
(2) the derivation of several indicators and one specific metric for assessing the software engineering impact of using Ada packages.

The assessment process assumes a guiding framework based on linkages among software engineering objectives, principles that support the realization of those objectives, and product attributes induced by the usage of the principles.

CR Categories and Subject Desecrators: D.2 **[Software Engineering]**: D.2.8 Metrics, D.2.9 Management - Software Quality Assurance

General Terms: Software Quality Assurance, Software Quality Indicators, Software Quality Metrics, Ada Packages, Metrification Procedure

Additional Keywords: Indicators, Metrics, Objectives, Principles, Attributes

Measuring Software Quality in Ada Packages: An Objectives, Principles, Attributes Framework

Gary N. Bundy and James D. Arthur

The Department of Computer Sciences
Virginia Tech
Blacksburg, VA 24061

1.0 Introduction

Developing measures for assessing software quality has been a continuous problem in computer science and software engineering. A literature survey of metrics reveals that there are many metrics available for measuring software. Some well documented metrics include Halstead's Software Science [HALSM77], McCabe's Cyclomatic Number [MCCAT76], and Henry and Kafura's Information Flow [HENRS81]. A major criticisms of many of these metrics is the lack of a "clear specification of what is being measured" [KEARJ86]. Another author notes that a desirable attribute, missing from most metrics, is that software metrics should "empirically and intuitively describe software behavior" [EJIOL87]. A first step in addressing such criticism can be found in Arthur and Nance's 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 then used to measure the extent to which attributes are either present or absent in the code [ARTHJ87]. 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.

Currently, the only block structured language to which the OPA framework has been applied is Pascal [FARNM87]. This paper describes the findings of applying the OPA framework to a second block structured language, Ada. Ada, like Pascal, has many conventional language constructs, e.g. loops, decisions, records, etc. Unlike Pascal, however, Ada provides significant extensions in support of separate compilation, information hiding, exception handling, and

concurrency [BOOCG83]. Because of its widespread use within mission critical Department of Defense applications, and because Ada is said to promote "better" software engineered products through its language facilities, the authors have initiated a research effort to examine Ada relative to the OPA framework. The initial phase of this effort involves three tasks:

- the classification and categorization of Ada constructs,
- the identification of new attribute/property pairs, and
- the development of metrics for identified attribute/property pairs.

The work described in this paper describes the first task applied to the entire Ada language and tasks two and three applied to one important Ada component, the package. Although other Ada components such as tasking and generics are also significant, length restrictions prevents an adequate discussion of them all. In the course of this paper, the OPA framework will be outlined, a metric development procedure used to identify attribute/property pairs and define metrics for Ada components will be discussed, and the procedure's application to the Ada language and Ada packages in particular will be described. Finally, ongoing and future work in the development of an automated Ada analyzer will be discussed.

2.0 Background and Approach

2.1 OPA Framework

The OPA framework was developed in response to a need for evaluating software development methodologies, and in particular, their adequacy and effectiveness [ARTHJ87]. As illustrated in Figure 1, the framework is based on the argument that a set of objectives can be defined that responds to project level requirements. Correspondingly, the selected development methodology should emphasize a similar set of objectives. Achieving these objectives requires adherence to certain principles that characterize the process by which the product is developed. Adherence to a process governed by those principles should result in a product that possesses attributes considered desirable and beneficial.

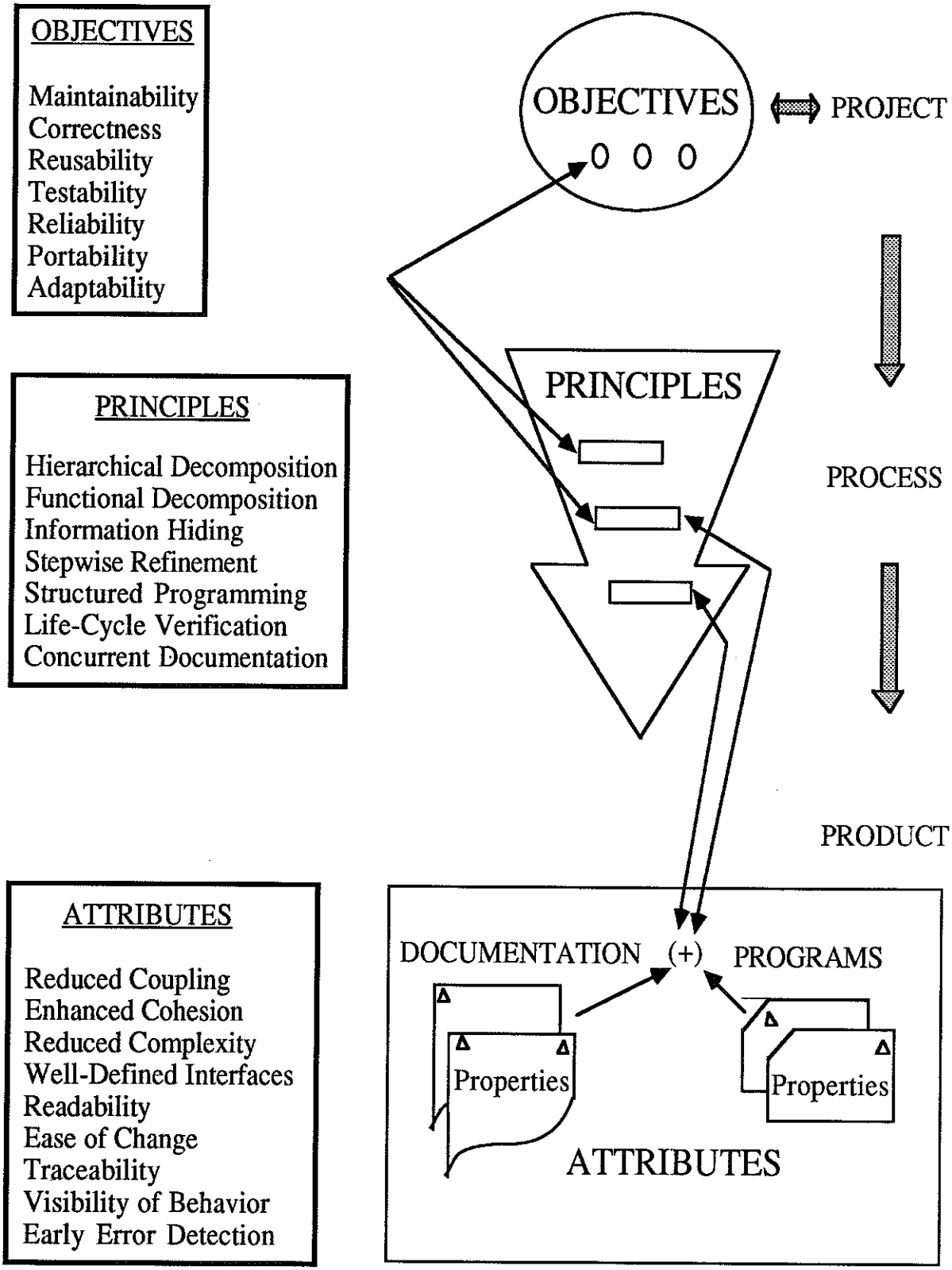

Figure 1

The Objectives, Principles, Attributes Framework for Software Quality Assessment

Underlying this rationale is a natural set of relations that link individual objectives to (one or more) principles and each principles to (one or more) attributes. Referring again to Figure 1, one achieves the objectives of a software development methodology by applying fundamental principles which, in turn, induce particular attributes in the product. Product properties support the definition of indicators (a property/attribute pair) whose measurements reflect the presence (or absence) of particular attributes. Assessing software quality, and ultimately the achievement of designated objectives, relies on:

- recognizing the extent to which attributes are present or absent, and
- analyzing the attribute information within the linkage framework.

More specifically, measuring and/or predicting software quality starts with the computed evidence of attributes (based on properties found to exist within the software product). Next, using those computations and the defined set of linkages among principles and attributes one identifies the set of principles needed to induce the observed product attributes and assesses the extent to which those principles were adhered to in the software development process. Finally, using the set of objectives/principles linkages in conjunction with the assessments from the previous step, one determines the degree to which objectives have been realized in constructing the software product. Comparing the measured achievement levels to defined acceptance levels provides a sound and defensible basis for judging the quality of the software product.

2.2 The Seven Steps to Metric Development

Given the OPA framework described above, the following seven steps provide a methodical process for developing metrics that support software quality assessment within that framework.

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.

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.

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.

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

5. *Identifying Product Properties Reflective of Presence (or absence) of Attributes:* The important activity in this step is to determine how to detect the various uses (abuses) of each component in the code. For each use (abuse) we must identify code that attest to the attribute impact determined in step four.

6. *Defining Product/Attribute Indicators:* This step formally links the code properties identified in step five to the affected attributes identified in step four and justifies such linkages.

7. *Defining the Measure and Supporting Metric for each Indicator:* This seventh and final step requires that we identify an approach for measuring the indicator defined in step six, and then define a metric whose value reflects the justification linking each property/attribute pair.

3.0 Tailoring the OPA Framework to Reflect Ada Characteristics

The seven steps outlined in the previous section provide a basis for tailoring the OPA framework to reflect characteristics specific to particular programming languages. Relative to an Ada-based effort the authors have completed the application of all seven steps. The following

sections individually discuss each of those steps, starting with the identification of critical Ada language components and ending with a discussion of the metrics that are directly related to the software engineering attributes defined by the OPA framework. Due to length restrictions on the paper, the authors choose to focus the presentation of steps two through seven on the impact of Ada Packages.

3.1 Identifying, Categorizing and Classifying Crucial Language Components

The first step in transforming the OPA framework into a well-defined 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 start with a categorization scheme that allows a language to be analyzed at the individual component level and then to analytical perspectives based on aggregated components. In concert with this approach, the authors initial categorization scheme employed partitioning criteria by Ghezzi and Jazayeri [GHEZC82], that is, the partitioning of language component 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 a discussion by Wichmann [WICHB84b], 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 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 allowed the authors to build on previous research results of Farnan [FARNM87] and Danfekar [DANDA87], and subsequently, to focus research efforts on the study of those language constructs primarily found in the Ada language.

Assuming that Farnan and Dandekar have necessarily and sufficiently analyzed conventional language constructs, the critical Ada language constructs requiring additional examination relative to the OPA assessment framework 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; our intentions are to mention only those that are most prominent from a software engineering perspective. The authors refer the interested reader to [BUNDG90] for a more elaborate discussion on identifying, categorizing and classifying Ada language constructs as related to software quality assessment within the OPA framework.

3.2 Understanding the Rationale for Component Inclusion

Fundamental to basing a software quality assessment process on product structure is a firm understanding of why the corresponding language constructs have been included in a language definition. In some cases, the rationale might simply be that a particular 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 characteristics. For Ada, the language designers have provided the *Rationale for the Design of the Ada Programming Language* [ADARF84]. Published papers describing research and development efforts and books describing usage techniques provide additional insights into the proposed uses of particular language components. For example, Booch [BOOCG83] and Wichmann [WICHB84a] independently support Ichbiach's contention that the use of parameters with name notation

enhances program readability and reduces the overall complexity. 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 [ADARF84] 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 has appeal because it promotes code cohesion [ROSSD86]. 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. In addition to abstract data types, Ada packages can be used to group named collections of declarations, provide subprogram libraries, and implement abstract state machines [BOOCG83, GANNJ86, ADARF84, SHUMK88].

As can be inferred from above, identifying the connection between the rationale for including a particular language element in Ada and the impact its usage can have on achieving desirable software engineering characteristics is crucial to our research. In fact, such connections serve to motivate measures and metrics used in assessing to what extent such characteristics do exist in a product. The following two sections describe steps three and four of the metric development procedure and, within a software engineering framework, substantiates the importance of understanding the rationale behind including particular language elements in the definition of Ada.

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 [WICHB84a,GHEZC82]. If used indiscriminately, however, it can have just the opposite effect [GHEZC82].

From the Ada standpoint, the literature abounds with citations attesting to the goodness of Ada language constructs from a software engineering perspective. In particular, Ada packages are extremely important to 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 [ADARF84]. 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 [BOOCG83, BOOCG87] 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 identified with rather abstract software engineering qualities like maintainability, reliability, information hiding, modularity and so forth. 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 defined in Section 2. 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 the nine attributes outlined in Section 2.

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 [ROSSD86]. 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 [GANNJ86, EMBLD88, BOOCG87], respectively), improper use of packages can also have a detrimental 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 will 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 from 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, the definition of ADTs, because of the capabilities provided by packages, has additional beneficial effects in terms of reduced code complexity and improved readability. Without further elaboration let it suffice 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 Properties, Indicators, 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. Steps five and six identify and formally link product properties (language elements) to software engineering attributes. Because each identified property undeniably reflects either the presence or absence of a specified attribute, we refer to the property/attribute pair as an indicator. Finally in step seven, a measurement approach and supporting metric is defined for each indicator. These three steps are being discussed together because they are intrinsically tied together. To illustrate these steps of the metric development procedure, the *use of packages to define subprogram units* is discussed.

From step four of the metric development procedure, we recognize that the definition of subprogram units within a package has a positive impact on the attributes of cohesion, well-defined interface, and ease of change. An interesting similarity exists among the attributes identified for defining subprogram units in packages, defining abstract data types in packages, and defining abstract state machines in packages. The similarity of affected attributes suggests that the three uses need not be distinguished in steps five, six, and seven of the metric development procedure. Rather than disucss packages in general, the remaining text in this section focuses on the identification of properties indicative of the presence of the attribute cohesion relative to using packages in defining groups of subprograms.

To begin the process of steps five, six, and seven, we identify the properties associated with a specific language construct use and the attribute that usage affects. In the cohesion example, our 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 related (or functionally cohesive) as they should be. A very high utilization suggests that the subprograms are very related or 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, and a measurement approach and 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, we must relate 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:

$$\text{Sub Package Utilization} = \frac{\sum \begin{matrix}\text{package subprograms} \\ \text{referenced}\end{matrix}}{\begin{matrix}\text{"Withs"} \\ \text{to a Sub} \\ \text{Package}\end{matrix}}$$

$$\overline{\text{(total \# of "withs") * (\# of subprograms in the package specification)}}$$

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

The analysis on packages that define abstract data types and packages that define abstract state machines produced similar results to the analysis on packages that export subprograms, so the three uses were merged into one indicator. The working list of property/attribute indicators for Ada packages are enumerated in Figure 2. For further details on the indicators and metrics for Ada packages (and all Ada components) see [BUNDG90].

4.0 Future Work

Experience has taught us that manually collecting the raw data necessary for computing the defined metrics is labor intensive and error prone. Correspondingly, a current major research thrust focuses on the development of an automated analyzer that can be used to assess the quality of Ada products. The "front-end" part of the analyzer will accept the Ada code and use both syntactic and semantic analysis to generate a structure and characteristics file. The structure and

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 (-)

Figure 2

Property/Attribute Indicators for Ada Packages

characteristics file will then be processed to generate a "raw" data file. Contents of the "raw" data file will be used to support metric computations. Examples of "raw" data are the number of global variables referenced by a program module, total lines of code, average lines of code per subprogram, and the average number of parameters passed per subprogram call. The "back-end" of the analyzer takes the raw data generated by the Ada specific front-end (along with documentation quality data and data extracted from the development process) and generates reports detailing the presence of attributes, use of principles, and achievement of objectives. The current research effort has been planned for the next two years and includes validating the analyzer via monitoring an Ada based development project.

5.0 Summary

The authors believe that the Objectives, Principles, Attributes framework is an intuitive approach for software quality assessment of Ada packages as well as all Ada-based products. The

crucial components of Ada that are not found in Pascal can be integrated into the OPA Framework through the seven step procedure for metric development outlined in this paper. Once integrated, it will then be possible to easily and effective assess Ada's impact on the software engineering objectives, principles, and attributes. Because Ada is designed to support software engineering activities, evaluating Ada code for the achievement of objectives, use of principles, and presence of attributes will be important to both developers of Ada-based products and the purchasers of Ada-based products. Instrumental to this assessment are the indicators and metrics for the identified crucial components of Ada and the development of the Ada analyzer to facilitate the automated assessment process.

List of References

[ADARF84] *Rationale for the Design of the Ada Programming Language*, Minneapolis, MN: Honeywell Systems and Research Center, 1984.

[ADARM83] *Reference Manual for the Ada Programming Language*, Ada Join Program Office, ANSI/MIL-STD-1815A, 1983.

[ARTHJ87] 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.

[BUNDG90] 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.

[BOOCG83] Booch, Grady, *Software Engineering with Ada*, Menlo Park, CA: The Benjamin/Cummings Publishing Company, 1983.

[BOOCG87] Booch, Grady, *Software Components with Ada*, Menlo Park, CA: The Benjamin/Cummings Publishing Company, 1987.

[DANDA87] 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.

[EMBLD88] 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.

[EJIOL87] 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.

[FARNM87] 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.

[GANNJ86] 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.

[GHEZC82] Ghezzi, C. and Mehdi Jazayeri, *Programming Language Concepts*, New York, John Wiley & Sons, Inc., 1982.

[HALSM77] Halstead, Maurice H., *Elements of Software Science*, New York: Elsevier North-Holland, Inc., 1977.

[HAMMC85] Hammons, Charles and Paul Dobbs, "Coupling, Cohesion, and Package Unity in Ada," *Ada Letters*, Vol. 4, No. 6, May/June 1985, pp. 49-59.

[HENRS81] 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.

[KEARJ86] Kearney, Joseph K., et al., "Software Complexity Measurement," *Communications of the ACM*, Vol. 29, No. 11, November 1986, pp. 1044-1050.

[MCCAT76] McCabe, Thomas J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, December 1976, pp. 308-320.

[ROSSD86] Ross, Donald L., "Classifying Ada Packages," *Ada Letters*, Vol. 6, No. 4, July/August 1986 , pp. 53-65.

[SHUMK88] Shumate, Ken and Khell Nielsen, "A Taxonomy of Ada Packages," *Ada Letters*, Vol. 8, No. 2, March/April 1988, pp. 55-76.

[WICHB84a] 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.

[WICHB84b] Wichmann, B. A., "A Comparison of Pascal and Ada," *Comparing and Assessing Programming Languages*, Englewood Cliffs, NJ: Prentice-Hall Inc., 1984.