METRICS FOR ADA PACKAGES: AN INITIAL STUDY

Many novel features of Ada present programmers with a formidable learning task. The study of four first-time Ada programmers suggests that a background in the software engineering practices supported by Ada is necessary to learn to use the features of the language.

J. D. GANNON, E. E. KATZ, and V. R. BASILI

When programmers begin to learn a new language, they often start by using those features of the language that have appeared in other languages they already know. For example, Fortran or Cobol programmers should have no trouble learning to use PL/1's DO statement for bounded iteration since this concept appears in slightly different forms in both languages. However, repetition while a condition holds is a novel concept for these programmers, and they are unlikely to use DO statements for this purpose. This is confirmed in a study of 102,397 statements in PL/1 programs that turned up 7385 DO statements, but only 11 DO WHILE statements [4]. Elshoff concluded that

> the most basic, general form of the DO statement is not used. The fact that the programmers do not know of its existence is a primary reason.

Modules allow programmers to group related data and/or procedures and to limit the amount of information that is accessible to the rest of the program [9]. Splitting a program into modules should localize the effects of program changes to correct errors or to improve the implementation (i.e., making it more robust or more efficient). In addition, since modules are usually self-contained, they can be reused from project to project. The designers of Ada® [7] recognized three major uses for modules:

1. a named collection of declarations that makes a group of types and variables available much like a Fortran common block;
2. a group of related subprograms that provides a library facility;
3. an encapsulated data type that provides the names of the type and its operations, but hides the details of the representation of objects of the type and implementation of the type's operations.

While the first two uses are familiar to many programmers, the third use is not supported by many commonly used programming languages. Strong syntactic clues are available to help programmers decide what objects comprise the first two kinds of modules (e.g., all types and constants, a collection of global variables, or a set of utility routines), but fewer hints are available to aid in grouping objects in problem-oriented terms [8]. Deciding what objects to encapsulate in a system is a formidable challenge.

HYPOTHESES

One of the lessons of structured programming is that simply providing users with "gotoless" programming languages does not result in structured programs being written. Users need to understand the ideas of top-down development and stepwise refinement to produce structured programs. Our main hypothesis was that a similar phenomenon was likely to occur with Ada packages. Users lacking a thorough familiarity with the ideas of information hiding and data abstraction were unlikely to use Ada packages to write programs that exhibited these properties.

Compiler-generated messages help programmers remove syntactic errors from their programs. Other warning messages, produced by the compiler or its run-time support system, may also be used to induce programmers to test with more data (e.g., test cover-

Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

This work is supported by the Office of Naval Research and the Ada Joint Program Office under grant N00014-82-K-0225.

age metrics) or alter the style of their programs (e.g., by including more comments). Metrics based on packages can be used to characterize the structure of a program, and to indicate how resistant the system is to changes in representations of data objects or the implementations of operations. Warnings based on these metrics might result in programmers rethinking their use of packages. However, care must be exercised since messages associated with novel constructs often provoke inappropriate responses. In a language in which programmers had to explicitly request the right to access nonlocal identifiers instead of inheriting them automatically, programmers responded to messages about lack of access to any identifier by making all nonlocal identifiers visible in each procedure [5].

ADA PACKAGES AND UNITS

In Ada, modules are implemented as packages. A package consists of two parts: a specification and a body. The specification, which contains declarations, is further divided into visible and private parts. Identifiers declared in the visible part can be used by other units, whereas those declared in the private part can only be used in the package body. For example, the following package specification exports the name of the type `Rational` with operations /, +, etc. The representation of a rational number by a record containing two integers is hidden from users in a *private* part.

```
package Rational is -- specification
  -- visible part
  type Rational is private;
  function "/" (X,Y: integer)
    return Rational;
  function "+" (X,Y: Rational)
    return Rational;
  ...
private
  type Rational is
    record
      Numerator, Denominator: integer;
    end record;
end;
```

The package body contains implementations of operations and declarations of types whose names appear in the corresponding specification part. Nothing declared in the package body is visible outside the package. However, package bodies and specifications can use information from other packages' specifications.

Packages are not required to hide the representation of data. The specification for `Rational` numbers above could have been declared as follows:

```
package Rational is -- specification
  -- visible part
  type Rational is
    record
      Numerator, Denominator: integer;
    end record;
  function "/" (X,Y: integer)
    return Rational;
  function "+" (X,Y: Rational)
    return Rational;
  ...
end;
```

However, if the representation is defined in the visible part of the specification, any other units that can see the package can manipulate the representation of the data (e.g., may access the `Numerator` or `Denominator` field of any `Rational` object). Changes in the representation, therefore, might have a tremendous effect on those units.

Encapsulating data types in packages allows the definition of objects and their associated operations. Hiding the representation of the data either in the private part of the package specification or in the package body limits the effects of changes in the representation.

Ada programs are collections of compilation units: predefined units, package specifications, and others (i.e., subprogram, package, and task bodies). Where a package is defined is another important decision. A package might be generally available to any other unit in the environment. It may be defined in a library restricted to a project or group that will limit the package availability to a subset of units. The author of a package also has the option of defining packages within other units limiting the scope to the defining unit. By choosing an appropriate location for a package's definition, the package's author can limit the scope of possible changes.

A package is visible in a unit if one of the following occurs: First, the package is named in a `with` clause at the beginning of the unit. Second, the package is visible in the unit's parent unit. Items declared in the package can be made directly visible with a `use` clause in the same manner. However, in this article, we concentrate on general visibility as opposed to direct visibility. Reducing package visibility would lower the number of possible bindings [1] between the unit and the package.

PACKAGE METRICS

There are many simple characterizing metrics that provide a sketch of the system: the number of packages declared, and the number of generic packages and the number of times each is instantiated. In

addition to these simple metrics, two more elaborate metrics are discussed below.

Component Access Metric

When selection operations are applied to composite objects outside package bodies, details of data representation are spread throughout the program. Distributing representation information rather than centralizing it in private parts of package specifications makes programs more difficult to change. If the type of the composite object is not defined locally, changes in representation to enhance program capability or efficiency could involve many statements in many compilation units.

For example, consider the following Ada fragment containing the visible type T2 having two array components (A and B) with identical numbers of elements with the same type (T).

```
-- original representation
N: constant integer := ...;
type T1 is array (1..N) of T;
type T2 is
  record
    ...;
    A: T1;
    B: T1;
    ...;
  end record;
```

If we introduced a new type, NewType, a record of two elements of type T, that permitted the two array components of the previous example to be combined into a single component (C) in T2

```
-- new representation
N: constant integer := ...;
type NewType is
  record
    A: T;
    B: T;
  end record;
type T1 is array (1..N) of NewType;
type T2 is
  record
    ...;
    C: T1;
    ...;
  end record;
```

then all references to the A and B components of type T2 variables (e.g., V) would have to be changed.

```
-- original representation
                V.A(...)
                V.B(...)
-- new representation
              V.C(...).A
              V.C(...).B
```

The ratio of component accesses of objects with nonlocally defined data types to lines of text in a program unit measures the unit's resistance to changes. Subunits of a package are considered part of the package; therefore, any component accesses to the objects defined in the package in these units are considered local.

Package Visibility Metric

Another means of measuring packages is to look at their visibility to other units in the system. We examine two versions of a system—the current one and one where the `with` clauses may have been moved to lower the visibility.

Definitions. Each package has the following visibility measures:

1. *Used.* Number of units where information from the package is accessed or changed.
2. *Current.* Number of units where the package is currently visible.
3. *Available.* Number of units where the package could be made visible by adding a `with` clause, given the current unit structure.
4. *Proposed.* Number of units where the package is visible given the current unit structure and the `with` clauses in their lowest possible positions.

Only those units that are part of the current system are included in our measures. In addition, package bodies and their subunits are not included in the measures for that package because they are part of its implementation. These values can be computed during the design of a system or after the system has been completed.

Examples. Figures 1 and 2 illustrate different views of a small system containing the following components: `package P` defining a `procedure X`; and `procedure A` defining subunits B, C, and D. In Figure 1, there is a `with` clause for P in A; therefore, P is currently visible in four units. (X is not included in these measures because it is a subunit of P.) However, P.X is only used in two units, B and D. In Figure 2, we propose moving the `with` clause from A to B and D, limiting the visibility of P to three units. For a system in which C is a subunit of B, that is the lowest location for the `with` clauses.

Visibility Ratios. Two interesting visibility ratios for the systems in Figures 1 and 2 are given in Table I (p. 620).

Each of these ratios has an upper bound of one and a lower bound of zero. The ratio of units in which a package is accessed to those in which it could be made available (UA) measures the perceived generality of the package. If UA(P) is high,

```
package P is
  procedure X;
end P;
-------------
separate (P)
procedure X is
begin
  ...;
end X;
-------------
with P;
procedure A is
  procedure B is separate;
  procedure D is separate;
begin
  ...;
end A;
```

```
separate(A)
procedure B is
procedure C is separate;
begin
  ...;
  P.X;
  ...;
end B;
-------------
separate (A.B)
procedure C is
begin
  ...;
end C;
-------------
separate (A)
procedure D is
begin
  ...;
  P.X;
  ...;
end D;
```

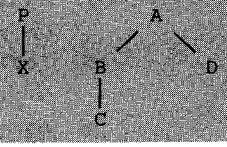

FIGURE 1. Global Visibility of Package P

```
package P is
  procedure X;
end P;
-------------
separate (P)
procedure X is
begin
  ...;
end X;
-------------
procedure A is
  procedure B is separate;
  procedure D is separate;
begin
  ...;
end A;
```

```
with P;
separate(A)
procedure B is
procedure C is separate;
begin
  ...;
  P.X;
  ...;
end B;
-------------
separate (A.B)
procedure C is
begin
  ...;
end C;
-------------
with P;
separate (A)
procedure D is
begin
  ...;
  P.X;
  ...;
end D;
```

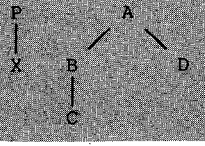

FIGURE 2. Limited Visibility of Package P

TABLE I. Visibility Ratios

Full name	Notation	Meaning	(1)	(2)
Used/available	UA(P)	Perceived generality	0.5	0.5
Proposed/current	PC(P)	Excess visibility	0.8	1.0

`package P` may support objects manipulated throughout the system. However, if UA(P) is low (i.e., P could be available much more widely than is necessary), the designers may have mistakenly believed the package to be more generally useful than it is.

The ratio of the number of units in which a package must be visible to those in which it is visible measures the success of the design decision to minimize visibility. If PC(P) is high, P's visibility is limited given the current system structure.

Had the design goal been minimizing the visibility of packages used, the second subunit structure would be better. The ratios should be used to indicate which packages should be examined more closely, not to replace the need to understand why design decisions have been made.

A CASE STUDY

We studied a subset of an existing ground-support system for a satellite, which was redesigned and implemented in Ada [2]. With the help of the original designers of the system, requirements were developed for a subset system that included an interactive operator interface, graphic output routines, and concurrent telemetry monitoring.

This was an early Ada development to examine the effect of using Ada in an industrial environment. The programming team consisted of four programmers with diverse backgrounds. The lead programmer had substantial industrial experience in the application area and was fluent in Fortran and assembler languages. The senior programmer had less experience in the application area, but wider exposure to languages. The junior programmer was a recent computer science graduate who had been trained in modern programming languages and design methods. The programmer/librarian was a novice programmer who had taken a single course in Fortran programming.

Since none of the programmers was familiar with Ada, a one-month training period preceded the start of the project. They viewed 15 hours of videotaped lectures given by Ichbiah, Firth, and Barnes. A six-day in-house course by a consultant was spread over a period of weeks to allow team members to complete assignments, the last of which was a 500-line team project in which they encapsulated new data types in packages. Another half day was spent reviewing the programming practices they were expected to use: design and code walk-throughs, structured programming, information hiding, etc.

The bulk of the development of this system was done with the Ada/Ed interpreter between February and December 1982. Some testing was done on the ROLM compiler in the summer of 1983. The lack of production-quality compilers and access to their parsers prevented the programming team from fully testing their system, and our reporting package metric values to them during the case study. Thus the study cannot confirm our hypothesis that reporting metrics will result in changes in system structure. However, the system structure and use of packages can be studied to determine if programmers with typical Ada training make effective use of packages.

The Program

The final program contained 4375 text lines (excluding comments and blank lines) of Ada. The system included 11 packages contained in 19 units and a main program with 29 subunits. Some attempts were made to decompose the functions of the subunits; therefore, as many as four nesting levels of subunits are used in the system. Figure 3 shows the structure of the system. Of the 11 packages defined, one's body was not written, and it was never used.

The packages were of four types:

1. two common blocks exporting only definitions,
2. three libraries exporting only functions,
3. four encapsulated data types exporting private type definitions and operations, and
4. two data types exporting the representation of the type.

Although these numbers seem to indicate that the package feature was used appropriately, closer examination refutes this conclusion. Of the four packages defining encapsulated data types, two were device drivers, another was a mathematical function, and the remaining package definition was neither completed nor used. Device drivers and mathematical libraries are common modules in existing software systems—no new fully encapsulated types were defined.

Five of the 11 packages were generic, but each was instantiated only once. The generic parameters were primarily ranges for arrays and precision for real numbers. The programmers used the standard

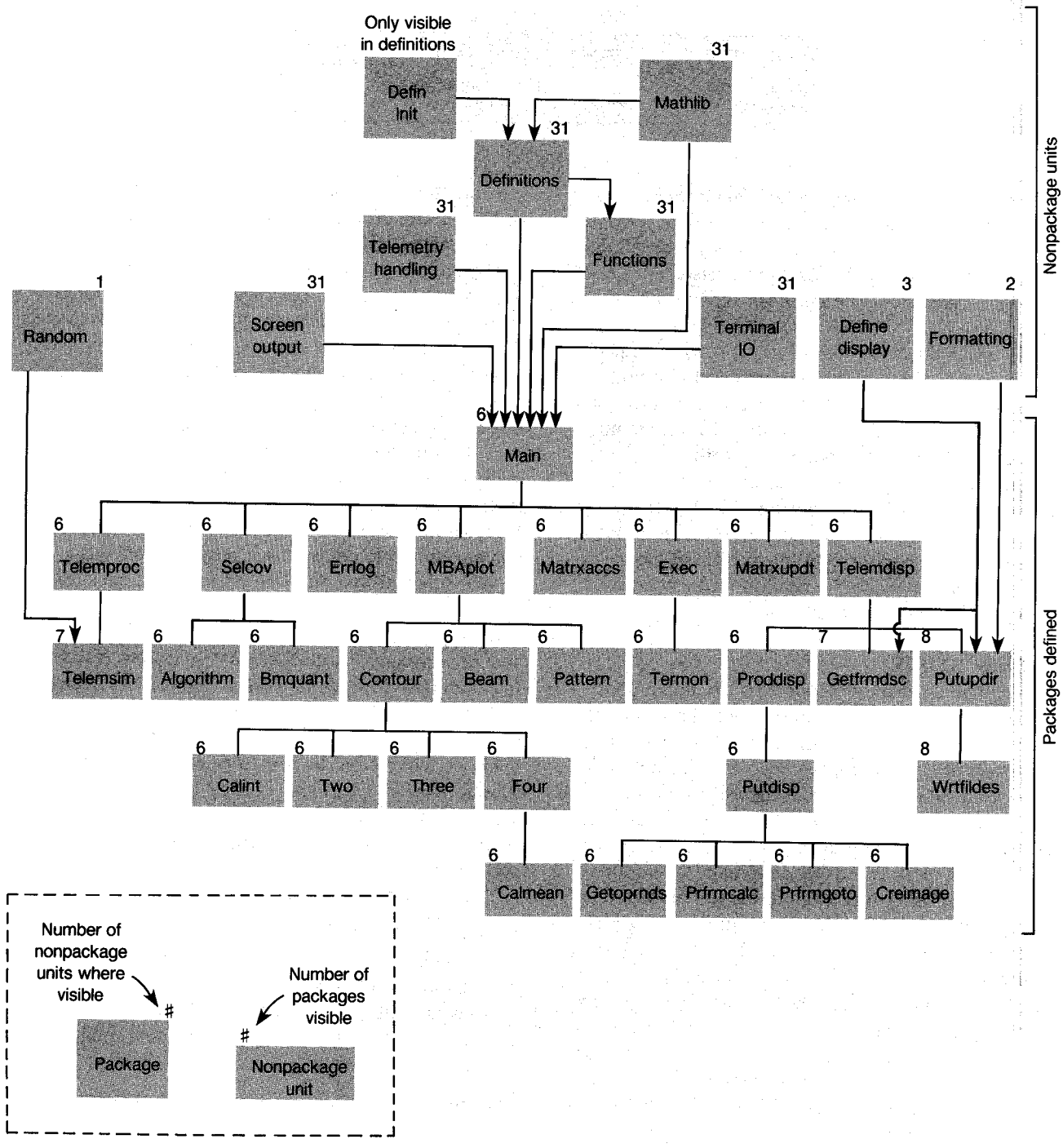

FIGURE 3. Actual System

sequential_io package in various instantiations, but they only used one instantiation for each of the generic packages they defined. Of those five instantiations, two were in one other package, and three were in the main program. The programmers seemed to view packages as global entities.

A more alarming discovery was that only two of the team members (those with experience in the widest variety of programming languages) defined any packages. The other two team members used the conventional packages provided by the other two programmers (i.e., the device drivers and the mathematical subroutines) and the standard Ada packages, but wrote none of their own. Even though the requirements specified that an antenna beamforming network had binary treelike connectivity

TABLE II. Component Accesses by Programmer

Metric	Programmer 1	2	3	4	Total
Text lines	706	1904	1648	117	4375
Component accesses					
All data	159	171	140	0	470
Only packaged data	120	124	61	0	305
Excluding packaged data	39	47	79	0	165
Accesses per text line					
All data	0.23	0.09	0.08	0.00	0.11
Only packaged data	0.17	0.06	0.04	0.00	0.07
Excluding packaged data	0.06	0.02	0.05	0.00	0.03

and a binary tree package specification was written, different internal functions manipulating a different representation of binary trees were written instead of providing a package body to match the specification.

COMPONENT ACCESSES

Table II summarizes nonlocal component accesses for each programmer based on all modules written by the programmer. The total number of component accesses, the accesses to packaged data, and the accesses to nonpackaged data are each included.

These metrics show that on average more than 1 of every 10 lines (0.11) of text contained a reference to a component of an object with an externally defined type. Roughly twice as many references are made to packaged data as are made to unpackaged data, which suggests that the more complex data types might have been packaged but not hidden. However, Programmer 3 made more references to components of unpackaged data.

Table III summarizes the component accesses by package for selected packages. Note that Package 3 has 217 of the packaged component accesses. This is not surprising considering that the package contains global data and types. The majority of the remaining packaged component accesses were to Package 7, which provides some types shared by several related units. If the data in these two packages were hidden, the number of packaged component accesses and the effect of changes to the packages would be greatly diminished. However, changes to the representation at this time would affect many other units.

TABLE III. Component Accesses by Package

Package	Number of units	Number of accesses
3	13	217
2	1	9
8	2	14
7	6	46
5	1	19

The values in Table II indicate that the first programmer's code should be relatively difficult to change since about one of every five lines contained a component access. We selected one of this programmer's modules and made the trivial modification discussed in the "Component Access Metric" section (page 618). To make this change in the representation, 11 program changes [3] were required in the module we selected. In addition, 10 program changes were needed in five other modules that encompassed two of the four major subsystems in the program. The record type containing these components could have been encapsulated in a package definition. Then, the same change in the representation would require a change in the private part of the package specifications and a total of 4 program changes in two functions of the package body. No other modules would be affected.

PACKAGE VISIBILITY

An examination of the visibility of the packages within the other units indicates that the system structure did not minimize package visibility. The visibility ratios for the 10 packages that were used are given in Table IV.

Although the main program used only two packages, six packages were named in both *with* and *use* clauses there. Most of the packages were viewed as global data or functions that were accessible everywhere. This view is consistent with the Fortran style of programming most familiar to the programmers.

Note that the UA column is fairly low for all the packages except Package 3, which contains type definitions and constants used throughout the system. The low values of UA suggest that most packages could be defined locally to groups of units.

The PC column demonstrates the programmers' view of the role of packages in the system. Five of the packages had minimal visibility; however, the rest of the packages are excessively visible. This is

TABLE IV. Package Visibility Vectors and Ratios

Package	Used	Proposed	Current	Available	UA	PC
1	9	13	30	44	0.2	0.4
2	4	9	33	45	0.1	0.3
3	20	30	32	46	0.4	0.9
4	1	31	33	47	0.0	0.9
5	11	30	30	47	0.2	1.0
6	4	9	30	47	0.1	0.3
7	7	13	30	47	0.1	0.4
8	3	3	3	48	0.0	1.0
9	1	1	2	48	0.0	0.5
10	1	1	1	48	0.0	1.0

yet another example of the programmers' global data approach to package definition.

CONCLUSIONS

The case study demonstrates what might happen when programmers who are experienced in an application area but lack training in modern software development practices begin to use Ada. Despite training efforts that are similar to those that are likely to be used in a typical industrial setting, only traditional modules like device drivers and mathematical libraries were defined. Encapsulated types were declared only by programmers with the widest exposure to different languages, but even the programmers' prior success in working with these languages does not guarantee success with Ada. A good background in the software engineering practices that Ada supports is probably necessary to learn to use the full capabilities of the features of the language—simply teaching professional programmers Ada is not enough.

We mistakenly assumed that because packages were being declared the programming team was using Ada effectively. Had our package metrics been applied during the case study, they might have helped the programmers better understand how to use packages and alerted us to their problems. Package visibility is a rather crude metric that can be used during design to check that the system architecture does not simply make all packages visible to all program units. Lowering the visibility will probably decrease the scope of any changes made to the package. However, even if package visibility is restricted, packages may still export type definitions that permit programmers to access the components of composite objects. Program units that directly access components of objects are likely to be difficult to change.

Metrics that track the use of packages during system development treat the symptoms and not the problem; however, we expect many early developments will have these symptoms. These metrics and those described in [6] may help in the transition to using Ada effectively.

Acknowledgments. M. V. Zelkowitz, J. B. Bailey, E. Kruesi Bailey, and S. B. Sheppard were the other monitors of the case study and have contributed to the work reported here.

REFERENCES

1. Basili, V.R., and Turner, A.J. Iterative enhancement: A practical technique for software development. *IEEE Trans. Softw. Eng. SE-1*, 4 (Dec. 1975), 390–396.
2. Basili, V.R., Katz, E.E., Panlilio-Yap, N.M., Ramsey, C.L., and Chang, S. Characterization of a software development in Ada. *Computer 18*, 9 (Sept. 1985), 53–65.
3. Dunsmore, H.E., and Gannon, J.D. Experimental investigation of programming complexity. In *Proceedings of the ACM/NBS 16th Annual Technical Symposium* (Gaithersburg, Md., June). ACM, New York, 1977, pp. 117–125.
4. Elshoff, J.L. An analysis of some commercial PL/1 programs. *IEEE Trans. Softw. Eng. SE-2*, 2 (June 1976), 113–120.
5. Gannon, J.D., and Horning, J.J. Language design for programming reliability. *IEEE Trans. Softw. Eng. SE-1*, 2 (June 1975), 179–191.
6. Hammons, C., and Dobbs, P. Coupling, cohesion, and package unity in Ada. *Ada Lett. 4*, 6 (May–June 1985), 49–59.
7. Ichbiah, J.D., Barnes, J.G.P., Heliard, J.C., Krieg-Bruckner, B., Roubine, O., and Wichman, B.A. Rationale for the design of the Ada programming language. *SIGPLAN Not. 14*, 6 (June 1979), 8–12.
8. Ledgard, H.L. Packages: A method for software decomposition. 1985.
9. Parnas, D.L. Information distribution aspects of design methodology. In *IFIP 71*, C.V. Freiman, Ed. North-Holland, Amsterdam, 1971, pp. 339–344.

CR Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*modules and interfaces*; D.2.8 [**Software Engineering**]: Metrics—*complexity measures*; D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types; modules, packages*; K.6.1 [**Management of Computing and Information Systems**]: Project and People Management—*training*
General Terms: Human Factors, Languages
Additional Key Words and Phrases: Ada, case study

Authors' Present Address: J.D. Gannon, E.E. Katz, and V.R. Basili, Dept. of Computer Science, University of Maryland, College Park, MD 20742.