NATO
STANDARD FOR
THE DEVELOPMENT OF REUSABLE
SOFTWARE COMPONENTS

Volume 1
(of 3 Documents)

- Development of Reusable Software Components
- Management of a Reusable Software Component Library
- Software Reuse Procedures

Issued and Maintained by:

NATO COMMUNICATIONS AND INFORMATION SYSTEMS AGENCY

(Tel. Brussels (2).728.8490)

This document may be copied and distributed without constraint, for use within NATO and NATO nations.

Table of Contents

Section	Title	Page
	Table of Contents	ii
	List of Figures	iv
	List of Tables	v
PART I	**INTRODUCTION AND BACKGROUND**	
Section 1	**Introduction**	**1-1**
1.1	Purpose and Scope	1-1
1.2	Guide to Using this Manual	1-1
Section 2	**Applicable Documents**	**2-1**
Section 3	**Basic Reuse Concepts**	**3-1**
3.1	Definitions	3-1
3.2	Expected Benefits of Reuse	3-2
3.3	Dimensions of Reuse	3-4
3.4	Forms of Reuse	3-5
3.5	Issues in Achieving Reuse	3-6
PART II	**STANDARD**	
Section 4	**Requirements Analysis**	**4-1**
4.1	Requirements that Encourage Reuse	4-1
4.2	Requiring Reusability	4-2
4.3	The Role of Domain Analysis	4-4
4.4	Requirements Specifications as RSCs	4-6
Section 5	**Design Principles**	**5-1**
5.1	Transition from Requirements Analysis	5-1
5.2	Models, Architectures, and Interfaces	5-2
5.3	Designing for Modification	5-6
5.4	Design Methods	5-8
5.5	Designs as RSCs	5-11
5.6	Selecting CASE Tools	5-11

Table of Contents (Continued)

Section	Title	Page
Section 6	**Detailed Design and Implementation**	**6-1**
6.1	Transition from Design to Code	6-1
6.2	Program Structuring	6-3
6.3	Interfaces	6-6
6.4	Parameterization	6-9
6.5	Handling Errors and Exceptional Conditions	6-11
6.6	Efficiency	6-14
6.7	Detailed Coding Standard	6-17
Section 7	**Quality Assurance and Test**	**7-1**
7.1	Evaluation Activities	7-1
7.2	Metrics	7-2
7.3	Test Procedures	7-4
7.4	Problem Resolution	7-5
Section 8	**Documentation**	**8-1**
8.1	Application of Conventional Documentation Standards	8-1
8.2	Documentation for the Reuse Library	8-2
8.3	The Reuser's Manual	8-4
8.4	Formal Specification	8-6

APPENDIX

Section	Title	Page
Appendix A	**Ada Coding Standard**	**A-1**
A.1	Identifiers	A-2
A.2	Format and Layout	A-5
A.3	Commentary	A-8
A.4	Types and Subtypes	A-11
A.5	Named Numbers, Constants, and Literals	A-15
A.6	Expressions	A-16
A.7	Control Structures	A-18
A.8	Exceptions	A-20
A.9	Program Structure and Compilation Units	A-22
A.10	Parameters	A-24
A.11	Tasks	A-26
A.12	Other Areas	A-28

List of Figures

Figure	Title	Page
Figure 5.1 -	Layered Data Base Management Architecture	5-5
Figure 6.2 -	Using a Filter Routine to Localize Dependencies	6-6
Figure 8.3 -	Example Outline for a Reuser's Manual	8-5

List of Tables

Table	Title	Page
Table 6.1 -	A Taxonomy of RSC Interface Types	6-7

PART I

INTRODUCTION AND BACKGROUND

Section 1

Introduction

The *Standard for the Development of Reusable Software Components* is designed to provide guidance in the creation of software products with maximum potential for software reuse.

The following subsections describe the purpose of this manual and explain how to use it effectively.

1.1 Purpose and Scope

To achieve the benefits of software reuse, it is essential to have an understanding of the specific activities to be performed at each step in software development; this manual provides that guidance to NATO, host nation, and contractor personnel.

Software reuse is an important aspect of controlling and reducing software cost and improving quality. The practice of software reuse can be significantly increased through the use of an appropriate standard in identifying software intended for reuse and developing that software. The *Standard for the Development of Reusable Software Components* is a prescriptive document designed to provide concrete reuse guidance. It assist the user in structuring a software development process that leads to the development of reusable software components.

The manual is intended for use by NATO, host nation, and contractor personnel. NATO and host nation program offices will use the guidance in establishing IFB requirements and in guiding contractors. Contractors will use it in establishing project-specific development practices.

This is one of a set of three manuals developed by NACISA to provide guidance in software reuse. This manual specifically addresses the *creation* of reusable software components. The other two documents are standards for the management of a library of reusable components, and for carrying out a project that takes advantage of the library to reuse existing software.

1.2 Guide to Using this Manual

This manual provides specific guidance, organized by software life-cycle activity, to form a basis for establishing individual project practice.

The *Standard for the Development of Reusable Software Components* is organized in two parts. Part I provides an introduction to the manual and a brief discussion of general concepts of software reuse to provide a basic frame of reference for the reader. Part II is the actual standard. Its major sections address requirements analysis, design principles, detailed design and

implementation, quality assurance and test, and documentation. Appendix A provides detailed reusability guidelines for the Ada programming language.

Within Part II, each regularly-numbered paragraph forms part of the standard, and is considered mandatory in meeting the reuse objectives addressed by this manual; any deviation must be justified and approved. The standard is augmented by a number of guidelines (indicated by paragraph numbers beginning with the letter "G"). Guidelines support the standard, identifying specific (potentially alternative) approaches to meeting the standard. Compliance with specific guidelines is not considered mandatory; however some effective approach to meeting the standard must be selected.

Because this manual offers alternative approaches, project managers should use this it as a basis for generating project-specific reuse guidance, for incorporation in other software development practices adopted for the project.

Section 2

Applicable Documents

Many existing materials provide valuable guidance that augments this standard.

The following reference documents are cited in this manual:

Booch, G. *Software Components with Ada*. The Benjamin/Cummings Publishing Company, 1987. ISBN 0-8053-0610-12.

Contel Corporation. *Standard for Management of a Reusable Software Component Library*. NATO contract number CO-5957-ADA, 1991.

Contel Corporation. *Standard for Software Reuse Procedures*. NATO contract number CO-5957-ADA, 1991.

Dynamics Research Corp. *ADAMAT Reference Manual*. 1987.

McCabe, Thomas J. "A Complexity Measure." *IEEE Transactions on Software Engineering* (Dec. 1976): 308-320.

Nissen, J. and P.J.L. Wallis (Eds). *Portability and Style in Ada*. Cambridge University Press, Cambridge, United Kingdom, 1984.

Numerous other references were used in developing this standard, and provide additional guidance in developing reusable software. Some that may be valuable to the user of this manual are:

Gautier, R.U. and P.J.L. Wallis (Eds). *Software Reuse with Ada*. Peter Peregrinus Ltd. on behalf of the Institution of Electrical Engineers, London, United Kingdom, 1990.

The Software Productivity Consortium. *Ada Quality and Style: Guidelines for Professional Programmers*. Van Nostrand Reinhold, New York, USA, 1989.

SofTech, Incorporated. *RAPID Center Standards for Reusable Software*. Document number 3451-4-012/6.4, 1990.

SofTech, Incorporated. *RAPID Center Reusable Software Component (RSC) Procedures*. Document number 3451-4-326/4, 1990.

SofTech, Incorporated. *Ada Reusability Guidelines*. Document number 3285-2-208/2, 1985.

Section 3

Basic Reuse Concepts

Software reuse offers tremendous benefits in cost savings and quality; however, it requires technical understanding, changed approaches, and an understanding of potential obstacles.

This section provides a frame of reference for understanding the benefits and challenges of software reuse. It introduces the terminology and concepts used in the remainder of the manual and explains the goals underlying the guidance provided herein.

3.1 Definitions

A consistent terminology is used throughout this and companion manuals.

The following are definitions of the key terms used in this manual:

Reuse—the use of an existing software component in a new context, either elsewhere in the same system or in another system

Reusability—the extent to which a software component is able to be reused. Conformance to an appropriate design and coding standard increases a component's reusability.

Reusable software component (RSC)—a software entity intended for reuse; may be design, code, or other product of the software development process. RSCs are sometimes called "software assets".

Reuser—an individual or organization that reuses an RSC

Portability—the extent to which a software component originally developed on one computer and operating system can be used on another computer and/or operating system. A component's reusability potential is greater if it is easily portable.

Domain—a class of related software applications. Domains are sometimes described as "vertical"—addressing all levels of a single application area (e.g., command and control) and "horizontal"—addressing a particular kind of software processing (e.g., data structure manipulation) across applications. The potential for reuse is generally greater within a single domain.

Domain analysis—the analysis of a selected domain to identify common structures and functions, with the objective of increasing reuse potential

Library—a collection of reusable software components, together with the procedures and support functions required to provide the components to users

Retrieval system—an automated tool that supports classification and retrieval of reusable software components, also called a "repository"

Software life cycle—The series of stages a software system goes through during its development and deployment. While the specific stages differ from one project to the next, they generally include the activities of requirements specification, design, code, testing, and maintenance.

3.2 Expected Benefits of Reuse

Software reuse clearly has the potential to improve productivity and hence reduce cost; it also improves the quality of software systems.

Productivity Improvement. The obvious benefit of software reuse is improved productivity, resulting in cost savings. This productivity gain is not only in code development; costs are also saved in analysis, design, and testing phases. Systems built from reusable parts also have the potential for improved performance and reliability, because the reusable parts can be highly optimized and will have been proven in practice. Conformance to standard design paradigms will reduce training costs, allow more effective practice of quality disciplines, and reduce schedule risk.

Reduced Maintenance Cost. Even more significantly, reuse reduces maintenance cost. Because proven parts are used, expected defects are fewer. Also, there is a smaller body of software to be maintained. For example, if a maintenance organization is responsible for several different systems with a common graphic user interface, only one fix is required to correct a problem in that software, rather than one for each system.

Improved Interoperability. A more specialized benefit is the opportunity to improve interoperability among systems. Through the use of single implementations of interfaces, systems will be able to more effectively interoperate with other systems. For example, if multiple communications systems use a single software package to implement the X.25 protocol, it is very likely that they will be able to interact correctly. Following a written standard has much less guarantee of compatible interpretation.

Support for Rapid Prototyping. Another benefit of reuse is support for *rapid prototyping*, or putting together quick operational models of systems, typically to get customer or user feedback on the capability. A library of reusable components provides an extremely effective basis for quickly building application prototypes.

Reduced Training Cost. Finally, reuse reduces training cost, or the less formal cost associated with employee familiarization with new assignments. It is a move toward packaged technology that is the same from system to system. Just as hardware engineers work with the same basic repertoire of available chips when designing different kinds of systems, software engineers will work with a library of reusable parts with which they will become familiar and adept.

Industry Examples. All of these benefits lead directly to lower-cost, higher-quality software. Some industry experiences have shown such improvements:

- **Raytheon Missile Systems** recognized the redundancy in its business application systems and instituted a reuse program. In an analysis of over 5000 production COBOL programs, three major classes were identified. Templates with standard architectures were designed for each class, and a library of parts developed by modifying existing modules to fit the architectures. Raytheon reports an average of 60% reuse and 50% net productivity increase in new developments.

- **NEC Software Engineering Laboratory** analyzed its business applications and identified 32 logic templates and 130 common algorithms. A reuse library was established to catalogue these templates and components. The library was automated and integrated into NEC's software development environment, which enforces reuse in all stages of development. NEC reports a 6.7:1 productivity improvement and 2.8:1 quality improvement.

- **Fujitsu** analyzed its existing electronic switching systems and catalogued potential reusable parts in its Information Support Center—a library staffed with domain experts, software engineers, and reuse experts. Use of the library is compulsory; library staff members are included in all design reviews. With this approach, Fujitsu has experienced an improvement from 20% of projects on schedule to 70% on schedule in electronic switching systems development

- **GTE Data Services** has established a corporate-wide reuse program. Its activities include identification of reusable assets and development of new assets, cataloguing of these assets in an automated library, asset maintenance, reuser support, and a management support group. GTE reports first year reuse of 14% and savings of $1.5 million, and projected figures of 50% reuse and $10 million savings, in telephony management software development

- **SofTech, Inc.** employs a generic architecture approach in building Ada compiler products. Compilers for new host and target systems can be developed by replacing only selected modules from the standard architecture. This has led to productivity level of 50K lines of code per person-year (10-20 times the industry average). This is typical of compiler developers, as this is a field in which reuse is accepted practice.

- **Universal Defence Systems (UDS)**, in Perth, Australia, develops Ada command and control applications. The company began its work in this business with a reuse focus, and has developed a company-owned library of 396 Ada modules comprising 400-500 thousand LOC. With this base, UDS developed the Australian Maritime Intelligent Support Terminal with approximately 60% reuse, delivering a 700 thousand LOC system in 18 months. A recently begun new project anticipates 50-70% reuse based on the company's asset library.

- **Bofors Electronics** had a requirement to develop command, control, and communications systems for five ship classes. As each ship class was specific to a different country, there are significantly different requirements for each. In order to benefit from reuse, Bofors developed a single generic architecture and a set of large-scale reusable parts to fit that architecture. Because of a well-structured design, internal reuse, and a transition to Ada and modern CASE tools, Bofors experienced a productivity improvement even in building the first ship—from 1.3 lines of code (LOC) per hour previously to 3.28 LOC per hour. Improvements are much greater for

subsequent ships, with a projected productivity of 10.93 LOC per hour for the fifth ship, which is expected to obtain 65% of its code from reuse.

3.3 Dimensions of Reuse

Reuse has several dimensions; the guidance in this manual supports all of these.

Compositional versus Generative Approaches. Approaches to reuse may be classified as either *compositional* or *generative*. Compositional approaches support the bottom-up development of systems from a library of available lower-level components. Much work has been devoted to classification and retrieval technology and to the development of automated systems to support this process. Generative approaches are application domain specific; they adopt a standard domain architecture model (a generic architecture) and standard interfaces for the components. Their goal is to be able to automatically generate a new system from an appropriate specification of its parameters. (The Fourth Generation Languages [4GLs] used in the commercial world can be considered an example of generative reuse.) Such approaches can be highly effective in very well understood domains, but significant effort is required to develop the initial model.

Small-scale versus Large-scale Reuse. Another dimension is the scale of the reusable components. Reuse on a small scale—for example, use of a library of mathematical functions—is practiced fairly widely today. The effort saved from a single reuse is not great; payoff comes from the widespread reuse that is possible. On a large scale, entire subsystems (for example, an aircraft navigation subsystem or a message handling subsystem) may be reused. Here the saving from a single reuse is great; many thousands of lines of code may be reused. However, the opportunities for reuse of a given component are more limited. Large-scale reuse can pay for itself even if a component is only reused once or twice, because of the amount of effort saved.

As-is Reuse versus Reuse with Modification. Components may be reused as is, or may required modification. Generally reusable components are designed to be flexible—for example, through parameterization—but often modification is necessary to meet the reuser's requirement. Modifiability—the capability of a software component to be easily modified—is particularly important in reusable software.

Generality versus Performance. Sometimes there is a trade-off between component generality and performance. A component designed to be general and flexible will often include extra processing to support that generality. Appropriate reusability guidelines help avoid this penalty; guidelines for the reuser can provide mechanisms for coping with performance problems that may arise.

3.4 Forms of Reuse

Reusable components are not necessarily code; they can be specifications, designs, code, tests, or documentation.

Specification Reuse. Reuse of specifications is particularly relevant when aiming for large scale reuse. Large-scale reuse requires up-front consideration during the requirements definition activity. If an entire subsystem is to be designed for reuse, this should be made explicit from the start. The specification is then reusable in systems that will reuse the component, guaranteeing that requirements will match. Reuse of specifications greatly increases the likelihood that design and code will also be reusable. Furthermore, reuse of specifications can reduce time spent on requirements definition and help ensure interoperability, even if neither design or code are reused.

Design Reuse. Sometimes a design can be reused even when the code cannot; for example, the code may not be in the required programming language, or it may have inappropriate environment dependencies. Design reuse can save significant effort in one of the most costly life-cycle phases, provided that the design is specified so as to facilitate reuse. Furthermore, the design phase establishes the software architecture that provides a framework for reuse. Reuse of the software architecture will provide significantly greater code reuse opportunities by establishing a standard functional allocation and uniform interfaces.

Code Reuse. The greatest payoff comes from reuse of actual code. Clearly this is possible only when the specification and design are also reusable. Reusable code should be accompanied by its associated life-cycle products—its requirements and design specifications, its tests, and its documentation—so the reuser will not have to regenerate them.

Test Reuse. Ideally, a reusable code component should be accompanied by test cases that can be used to test it in the environment in which it is reused. A less obvious point is that tests can be reusable even when code is not, with reusable test cases accompanying specification reuse. An example might be the reuse of a specification and a set of test cases for a particular communications protocol. Even if the design and implementation differ from the original, specification and test reuse will save effort and help ensure correctness and interoperability.

Documentation Reuse. Documentation is a major source of software development cost. To be most valuable, a reusable component must be accompanied by appropriate documentation items. Clearly, reuse of a specification or design is only meaningful when the component is in a written form. However, other documentation such as users manuals may also be reusable, even when the code is not.

3.5 Issues in Achieving Reuse

Reuse involves significant change to traditional practice; there a number of challenges to be overcome in achieving its full benefits.

Identifying Opportunities for Reuse. A major technical issue is simply identifying opportunities for reuse. A software engineer may know that similar software has been written before; finding it is another matter. Reuse libraries help solve this problem. Once a component is found, it may be hard to determine if it is indeed a fit, and hard to modify it if change is required. Often software that appears to be reusable in fact will not be—it has inappropriate interfaces, hidden dependencies, inflexible functional limitations, or is simply so difficult to understand that the engineer will be better off simply starting over. The objective of software reusability guidelines is to help avoid these problems.

Investment. Making software that is reusable generally requires investment above and beyond that required for a one-time system. This effort goes into making the software more flexible, ensuring its quality, and providing additional documentation required. Each organization must make decisions about how the investment is supported.

The "Not Invented Here" Syndrome. Sometimes developers are unwilling to reuse software. Software engineers enjoy the creative aspects of their profession, and can feel that these are diminished when reusing software. Management encouragement, training, and perhaps other incentives can help engineers shift to a view of creativity that involves larger "building blocks"—reusable software components.

Estimating and Measuring. Estimating and measuring software development activities has always been difficult, but there are some organizational methods in place that work relatively well. These traditional methods will require modification in a reuse environment, and little data is available to support that modification.

Contractual, Legal, and Ownership Issues. There are a number of contractual, legal, and ownership issues that impact software reuse. Today's usual contracting methods can create a disincentive for contractors to reuse existing software or to provide software for reuse by others. Legal issues arise over liabilities and warranties. Responsibility for maintenance must be identified.

These organizational challenges are, for the most part, outside the scope of this set of manuals. Each organization must develop its own solutions. Managers must be aware of the challenges and address them if reuse is to succeed.

PART II

STANDARD

Section 4

Requirements Analysis

The requirements analysis phase lays the groundwork for software reuse. Attention to reuse at this point can have a major impact on the extent to which later project products are reusable.

This section presents guidance for activities that can be performed in the requirements analysis phase to support the development of reusable software.

Development of requirements is often done in whole or in part by the program customer (NATO or a host-nation program management office). These guidelines apply to customer activities as well as to contractor activities.

The following subsections address establishing requirements that encourage reuse of existing software, requiring the development of reusable software, the role of domain analysis in the requirements phase, and the handling of the requirements specification as an RSC.

4.1 Requirements that Encourage Reuse

The requirements specification must recognize and encourage software reuse.

Reuse begins at the requirements stage. The software requirements lay the groundwork for reuse by providing a statement of the required functionality and performance. Requirements must identify any required or expected reuse, and must not inhibit other reuse.

4.1.1 Specify only what is really needed; overspecified requirements inhibit reuse.

In many cases opportunities for software reuse are lost because the system requirements unnecessarily preclude them. The best opportunities for reuse arise when the system specification requires only the necessary functionality and performance, and allows the system designer to select operational specifics. This gives the designer the freedom to identify reusable components that can help provide the needed capability. An extremely explicit specification—for example, one that describes precise screen layouts or report formats—is unlikely to correspond to any existing software, and to thus require new development.

G4.1.1.1 Examine each requirement for necessity. Be sure it is a requirement and not a part of the design solution.

Particularly when the requirements specification is developed by someone other than the software designer, there is a tendency to include aspects of the solution that are not in fact requirements. Such requirements constrain the developer's options to offer a more cost-effective solution based on reuse. A point-by-point examination of each requirement which asks "Is this really something we need, or can we ask the developer

to propose his own solution?" will help eliminate overspecified requirements. An independent party may be able to perform this review more effectively.

G4.1.1.2 Offer contractor(s) an opportunity to review the system specification to identify potential changes that could increase opportunities for reuse.

When the requirements specification is prepared before going out for bids on a program, it is often first issued as a firm requirement. Instead, consider providing a draft specification to potential offerors with a request that they identify areas in which requirements could be modified to permit reuse of available software. When the specification is prepared jointly by the customer and the contractor as part of the contract effort, it is important that software development personnel are involved to provide a similar insight.

4.1.2 Specify any required or expected reuse.

Sometimes the organization developing the requirements specification will be aware of existing software that can or should be reused in developing the new system. If reuse is a requirement, it must be specified as such. For any such requirement, it must be possible to determine whether it is met.

G4.1.2.1 Provide a reuse goal.

In most cases, there will probably not be a particular software component that *must* be reused. However, it may be desirable to encourage the developer to practice reuse. (This is particularly likely to be true once a significant library of RSCs has been developed.) In this case, a goal percentage of reuse might be specified. This would indicate the percentage of the overall system code resulting from reuse rather than from new development. Such a metric requires some thought to determine how reuse is measured (e.g., lines of code, number of functions reused, number of requirements reused), how compliance will be assessed, how reuse with modification is counted, etc.

G4.1.2.2 Provide an incentive for reuse.

If it is not desired to require reuse, it might be appropriate to encourage it through some sort of contractual incentive, whenever possible.

4.2 Requiring Reusability

To ensure the development of reusable software, any specifically needed reusability must be explicitly required.

If the software to be developed for a system is to have reuse potential in future systems, this should be stated as a requirement. There are two possibilities—(a) the software should generally be developed so as to facilitate reuse, and (b) specific parts of the software must be reusable in a given set of circumstances. In either case, the requirements must be established appropriately and explicitly.

4.2.1 Establish explicit and verifiable reusability requirements in the contract.

Little is to be gained by simply asking developers to make their software reusable, or by providing them with a reusability guidelines document like this one. Reusability must be specified as a project requirement. Requirements are only meaningful when they are explicitly stated and objectively measurable. A requirement for reuse must explain what reuse means, that is, how to know it if you've got it. There must be a way of assessing whether the requirement was met.

G4.2.1.1 Specify anticipated scope of reusability.

A reusability requirement can be made more explicit when the desired scope of reuse is known. For example, it may be known that a software component will need to be reused on another platform, or that it must be adapted to a different communications interface. In such cases, this can be made an explicit requirement. Desired flexibility or parameterization can also be specified. For example, the system under development might require a window-oriented user interface. If it is anticipated that the interface will be reusable in other systems, the contractor can be tasked to provide a window package that is tailorable to other applications, rather than one that provides only the specific screens needed for the original system. In effect, known desired RSCs become program requirements in their own right.

G4.2.1.2 Specify conformance to a reuse standard.

If a reuse standard such as this one is available, conformance can become a contractual requirement. In this case the developer should be expected to describe how conformance will be ensured. Alternatively, the developer might be asked to provide his own organization-specific or project-specific standard, again describing how it will be used.

G4.2.1.3 Specify needed documentation.

Reusable software components, to be most usable, require documentation beyond normal program documentation. Section 8 of this manual addresses this requirement. Any such documentation need should be contractually required.

G4.2.1.4 Identify/require tests for reusability.

Like any other requirement, reusability requirements should be testable. An understanding of how conformance to such requirements will be measured should be established during requirements specification. When an explicit requirement, such as the ability to operate on another platform or the ability to be parameterized differently, is known, this can be tested directly. When the requirement is conformance to a standard, the test will probably consist of quality assurance (QA) inspections, perhaps aided by some automated coding standards checker.

G4.2.1.5 Provide for subsequent maintenance of RSCs.

Maintenance of reusable software should be addressed. One approach is that RSCs are simply placed in the library and then reused as is, with the reuser responsible for any fixes. An alternative is that the library organization will want continued maintenance by

the original developer. If this is the case, the program requirements should allow for such a maintenance activity.

4.3 The Role of Domain Analysis

Domain analysis identifies functional commonality across a range of systems or subsystems, and can influence the choice of requirements.

Domain analysis is an activity above and beyond the effort normally carried out during a requirements analysis phase for a particular program. It is the analysis of a class of systems in a particular application domain, as well as of anticipated future requirements and evolving technology. Its objective is the identification of a common architecture and common functions and interfaces that apply across the domain. Once these are identified, software that is constructed according to this common architecture has greatly enhanced potential for reusability in future systems.

Alternatively, a domain analysis may already have been done for the application area. Any such products should be examined for potential utility in developing the new system.

4.3.1 Consider future reuse opportunities when analyzing system requirements.

A complete or partial domain analysis can increase the reuse potential of the software by making it a better fit to future requirements. The benefits of a domain analysis activity should be analyzed, and an effort undertaken if appropriate. A partial analysis of reusable subsystems and interfaces can also be worthwhile.

G4.3.1.1 Evaluate the appropriateness of a domain analysis.

Either a customer or a development contractor may make the decision that a domain analysis is desirable, either independently or in conjunction with a particular development effort. Some of the criteria in making such a decision are:

- Will the organization be building more systems in the same domain, and consequently be able to profit from the availability of a standard architecture and parts that fit that architecture?
- Is the technology of building systems in this area sufficiently well understood that a satisfactory standard architecture is a realistic expectation?
- Has the developer built similar systems, and thus gained the experience to ensure that the products of the domain analysis are usable?
- Is there a mechanism for requiring/ensuring that the products of a domain analysis are in fact used? For example, if a contractor is tasked to develop a standard architecture, is there a way to see that other contractors use it?
- Is there a way to amortize the cost of the domain analysis across the organizational elements that will profit from it?

If these questions are generally answered in the affirmative, a domain analysis is probably well worthwhile.

G4.3.1.2 Identify reusable subsystems and interfaces.

Without the time and expense of a full domain analysis, it is still possible to identify elements that a system may have in common with future applications in its class, and to design those with particular attention to reuse. This identification can be done relatively quickly by someone familiar with the organization's future directions in the application domain. Candidates for such treatment might include interfaces to standard devices or to other systems, communications protocols, user interface packages, graphics packages, specific graphics such as maps, application-specific algorithm packages, message handling, etc. Attention to future requirements can help build a powerful library of application parts as a by-product of an ongoing effort.

4.3.2 Make use of existing domain analysis products.

A domain analysis for the selected domain may have been conducted previously by the customer or the development organization. If so, a recommended architecture, an identified set of common functions, and potentially reusable parts to implement those functions may be available. Taking advantage of these where appropriate will save money in the development effort. More importantly, it will support the effort at commonality that was begun by undertaking the domain analysis initially.

G4.3.2.1 Evaluate the suitability of existing products.

In evaluating existing domain analysis products, some of the questions that should be asked are:

- Is the system to be built a representative member of the domain addressed by the domain analysis? Will the architecture be suitable?
- Exactly what is available? If there is only a model architecture and a parts list but no detailed design or code components available for reuse, cost savings will probably not be great.
- If detailed design or code components are available, how much of the required functionality do they supply?
- What level of standardization has the architecture achieved? Is there benefit to be gained by using the architecture and/or interfaces in order to promote commonality across an organization?
- Have the products been proven in practice? What level of confidence is realistic?

Even if the overall architecture from a previous domain analysis is not applicable, some of the available reusable parts, interfaces, etc. may be useful and should be evaluated.

G4.3.2.2 Provide feedback and additions/modifications to the "owner" of the existing products.

An architecture and its component parts evolve with use. Each user of these products should contribute to that evolution by providing feedback on the effectiveness of the products and by supplying any added or modified parts to the library.

4.4 Requirements Specifications as RSCs

Requirements are reusable components; they should be chosen and expressed so as to facilitate reuse.

Requirements specifications consist of a number of individual requirements, many of which represent requirements that will also occur in other systems. These requirements are thus reusable and should be treated as RSCs. Reuse of the design and implementation that result from the requirement is far more likely if the requirement itself is reused in the specification for the new system. Even if the design and implementation cannot be reused, a reused statement of the requirement lessens the chance of error and incompatibility.

4.4.1 Express requirements so as to facilitate their reuse.

Reusable requirements must be easily identifiable as such, and must be easily extractable from the requirements specification for incorporation in a reuse library.

G4.4.1.1 Separate requirements with reuse potential from system-specific requirements.

To the extent possible, system-specific requirements should be isolated from reusable requirements, rather than combining both aspects in a single requirement. For example, consider a system with a requirement for a window-oriented user interface. If the requirement for the general "look and feel" and operation of the windows, controls, etc. is separated from the requirement for the specific displayed information, the former can be reused. If a single requirement describes the general as well as the specific aspects of the user interface, it cannot be reused without significant modification. It may also be desirable to use some notational means to distinguish requirements with likely reuse potential from system-specific requirements.

G4.4.1.2 Ensure that requirements for identical functionality are expressed identically.

Sometimes reuse opportunities are not recognized due to differences in the way requirements are described. This can happen either across parts of a single system or across systems. For example, two systems may need to display maps of the same parts of the world with the same information and operations. If the two capabilities are described by two different specification writers in their own words, it is difficult to recognize that the capabilities are the same. If identical wording is used, such recognition is easier. An even better approach is to use only a single statement of the requirement, either as an independent document referenced by both specifications or by reference from one specification to the other.

G4.4.1.3 Employ a uniform means for representing requirements specifications.

If an organization wishes to reuse requirements across systems specifications, it can be inhibited if each specification employs distinct formatting, organizational, and notational conventions. An organization that adopts a uniform style for specifications can reuse requirements far more easily.

G4.4.1.4 Develop requirements specifications in a machine-readable form.

Extraction and reuse of requirements specification components are much easier when the requirements can be automatically extracted from machine readable text. If requirements must be retyped, there is opportunity and perhaps incentive to modify them, thus losing the desired commonality.

Section 5

Design Principles

Software reusability is greatly impacted by software design approaches and decisions; this section provides guidance in the effective use of design approaches to facilitate reusability.

This section presents guidance for the use of software engineering principles and methods to achieve software reusability. It deals with general principles and top-level design choices, whereas the next section (Section 6) presents more specific guidance on detailed design and implementation.

The subsections address: the transition from requirements analysis; the role of models, architectures, and interfaces; designing for modification; design methods; designs as RSCs; and selection of Computer-Aided Software Engineering (CASE) tools to support design for reuse.

5.1 Transition from Requirements Analysis

The design phase must carry through the reuse requirements established earlier, and must establish the framework for implementing them in later phases.

The top-level design activity allocates requirements to components of the software architecture. This allocation must preserve and facilitate the reuse requirements established in the requirements specification. This is a critical activity in determining the overall reuse potential of the software, as it defines the reusable components to be built.

5.1.1 Identify and define all reusable entities in the top-level design.

The design process must provide for a software implementation of the previously identified requirements. In general, design can be broken down into top-level design and detailed design. The top-level design phase develops a software architecture which defines the software components and their interfaces, while the detailed design develops the specific component designs. Because top-level design is the time when the specific system components are determined, it is the time when potential reusable components are identified. All reusable requirements must be addressed, and any other reusable components must be identified.

G5.1.1.1 Map reusable requirements to reusable design components.

While there is not a one-for-one mapping between the requirements and the software components, each reusable requirement should be mapped to one or more reusable components. A single component should not combine reusable and system-specific requirements; separation should be retained.

G5.1.1.2 Identify additional reusable components during the top-level design phase.

As the design is developed, components will be identified that, while not explicitly addressing one of the reusable requirements, clearly have reuse potential. This is particularly true of lower-level components in the design hierarchy. All components identified during design should be considered from the standpoint of reuse potential.

G5.1.1.3 Indicate which components are intended as RSCs.

Components intended for reuse are subject to special consideration—for example, conformance to this standard. Consequently, they should be clearly distinguished from system-specific components. Use some notational mechanism in the specification to identify or separate intended RSCs.

G5.1.1.4 Provide explicit requirements traceability.

Requirements tracing identifies the mapping from each requirement to its implementation in the top-level design, and subsequently, to the detailed design, code, and tests. An explicit requirements trace ensures that all reusable requirements are mapped to design elements. It also provides the basis for later reflecting these relationships in a reuse library, so that library users can locate all software components associated with a particular requirement.

5.1.2 Implement required generality and modifiability in detailed design.

Once the top-level design activity has identified the components to be developed, detailed design develops the specific algorithms to implement them. At this level, the generality and modifiability necessary to support reuse are designed.

G5.1.2.1 Identify the appropriate parameterization for each component.

Parameterization identifies the range of data on which a component will operate, and allows for execution-time tailoring of specific processing. The detailed design activity should examine the requirements not only for the specific application, but to meet the component's reuse goals. Each component identified in the top-level design as an RSC should be subject to such an analysis.

G5.1.2.2 Follow detailed design guidelines for RSCs.

Section 6 of this manual provides detailed design guidelines for reusable software components. These should be followed for each component intended as an RSC, in order to maximize the generality, flexibility, and modifiability of the component.

5.2 Models, Architectures, and Interfaces

The software architecture establishes the framework for reuse; use of modelling, layering, and interface design principles helps ensure an effective structure.

Software reusability is first and foremost a design issue. If a system is not designed with reusability in mind, component interrelationships will be such that reusability cannot be

attained no matter how rigorously coding or documentation rules are followed. The software architecture identifies the major software components and their interfaces. These components are reusable to the extent that they implement well-defined, complete functions that may occur elsewhere. The architecture should be designed to facilitate this.

5.2.1 Use models to provide a mapping of functional requirements to implementation decisions.

A model provides a common viewpoint for stating software needs and matching these needs against existing capabilities. It provides a structure and a vocabulary for organizing and describing capabilities in a uniform manner. In a sense, all software architectures reflect *some* model that the designer has formulated; our objective here is a logical, well-structured model that can serve as a basis for understanding between developers and reusers.

For example, if a designer wants to create reusable software for manipulating graphic displays, he first has to define a model of the semantic functions needed to establish and manipulate a certain kind of display interaction. He might choose as a model the idea that a form will be presented to a display user who will be asked to fill in the fields of the form, following prompts on the display which indicate what kind of data is to go in each field. This familiar paradigm helps establish a basis for reusability. The concepts of form layout and allowable values for each field provide natural bases for parameterization. A designer of a new system can ask himself whether any capabilities needed for his system can be provided within the framework of a fill-in-the-blanks form. If so, he can then specify the tailoring he needs in terms of form layout and field values.

G5.2.1.1 Base models on real-world concepts.

The most effective models reflect real-world concepts, as they map readily to our understanding of the problem at hand. For example, the display interaction model described is readily understandable to potential reusers because it corresponds to a real-world concept—filling in the blanks on a form. A potential reuser can instantly form an initial judgement of whether the capability can or cannot meet his needs. If it appears to be what he needs, he can then examine the component in more detail to support a decision. In contrast, if the RSC is simply described as an "interactive display package", the reuser might have to expend significant effort simply to form an initial judgement of its utility.

G5.2.1.2 Generalize models to increase reuse potential.

A real-world model increases the understandability of the RSC's intent; added generality will increase its reuse potential. The display interaction model described above would have limited reuse potential if it supported only the specific form layout and field contents were limited to those required by the initial application. The designer seeking to make a reusable component would begin with the concept of the particular form(s) he wishes to design, then generalize the capability so as to allow the form layout and field contents to be varied. This creates an effective, easy-to-understand RSC with wide applicability.

5.2.2 Use a layered architecture to separate concerns and thus to isolate reusable subsystems.

A layered architecture contributes to reusability by separating concerns into discrete layers that can be separately replaced to tailor the software function and performance. It partitions the system into an ordered series of layers, each of which implements a specific abstraction. Each layer provides a set of services to the layer above it. It, in turn, uses the services provided by the layer below to carry out its own work. In effect, the layers below a given layer define an abstract machine that is used by that layer to carry out its functions. Each layer has specific, defined interfaces to the layers directly above and below it in the hierarchy.

G5.2.2.1 Define architecture layers to support portability and to allow adaptability to anticipated change.

A layered architecture should be designed so that individual layers represent aspects of the solution that are subject to anticipation—for example, changes in platform and operating system (at lower levels) or changes in user interface (at higher levels). Layers can then be independently replaced to tailor the function or performance of the system, thus limiting the effect of a change to a relatively small part of the system. Since it is rarely the case that a moderate to large system can be reused exactly, limiting the effects of the required changes can significantly reduce the effort required to modify the system for reuse.

For example, consider the layered architecture for a simplified data base management system shown in Figure 5.1. The bottom layer, the physical device layer, does the physical input/output (I/O) to the data storage devices. This may include issuing actual I/O device commands, handling device interrupts, and dealing with device errors. This layer supports the abstraction of an idealized I/O device that simply and reliably reads or writes data blocks from one or more of a large set of possible physical device addresses. If the entire data management system was to be reused in another system that had a different type of physical data storage device, only this layer would need to be changed to deal with the new device. The next layer, file I/O, represents the data storage as a set of named files that may be created, destroyed, read from, or written to. A change in the directory structure, or in the handling of variable length files, involves only this layer. The data access layer provides access to particular pieces of information, using the file I/O facilities to store and retrieve the data items. Modifications to the data access method involve only this layer. The top layer, data manipulation, carries out specific operations on the data in the data base. Only the top layer needs to be changed to change the way data is manipulated.

G5.2.2.2 Use existing industry-standard layered architectures where appropriate.

In some fields there are existing layered architecture models that are becoming industry standards. An example is the International Standards Organization (ISO) Open System Interconnect (OSI) model of a communications system. This model uses a seven-layer architecture to separate the various aspects of the communications process. By conforming to this architecture, application developments in the communications domain will experience more opportunities to reuse existing software and will produce more software that can be reused in future programs.

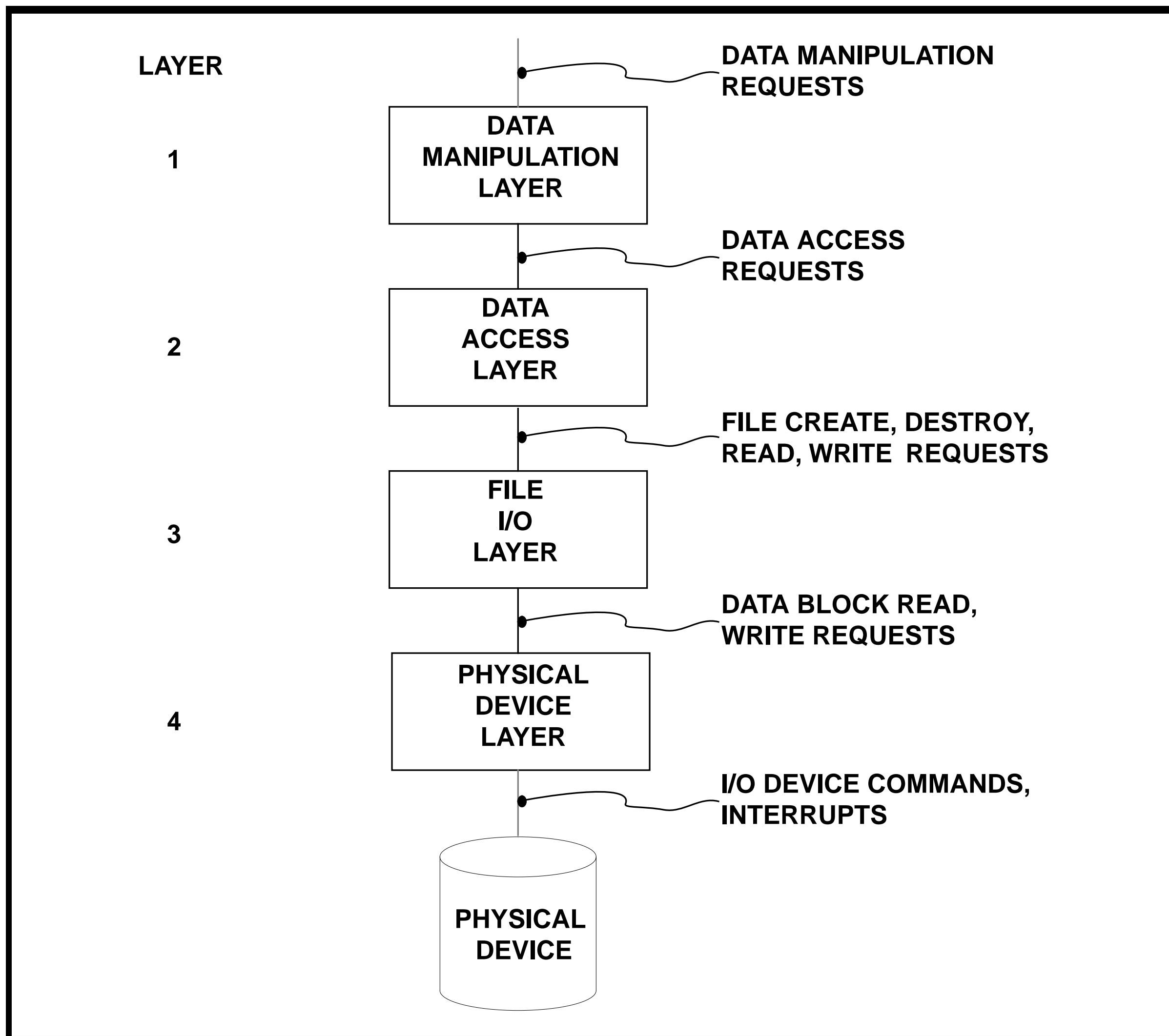


Figure 5.1 - Layered Data Base Management Architecture

5.2.3 Clearly identify and specify interfaces for all RSCs.

In a sense, a component's interfaces specify the component. The component can be thought of as a "plug-in" part, where the reuser must provide the correct "socket". The component's interfaces describe how to construct that socket, and determine how easy or difficult it is to do so. The interfaces to all components intended for reuse should be identified during the top-level design phase. They should be clean, concise, and clearly specified. Where possible, they should conform to existing industry standards. (This section presents general principles for effective RSC interface identification; Section 6, which addresses detailed design and implementation, provides detailed guidelines for the design of RSC interfaces.)

G5.2.3.1 Identify interfaces that should be standardized.

The interfaces to all components specifically intended as RSCs should be identified during top-level design. The interfaces of a component form a standard with which reusers must comply, as in "all systems desiring to incorporate this component must conform to the following specification ...". Thus, the designer of a reusable component is, in effect, determining design constraints for future systems. The interfaces of an RSC

should be carefully designed to facilitate their use as such a standard, and should be fully documented. The designer of such an interface should keep this viewpoint in mind; thinking of the RSC as a software product can help.

G5.2.3.2 Design interfaces of reusable components according to strict criteria of necessity and sufficiency.

The interfaces of a reusable component should be sufficient to provide the necessary flexibility and generality, but should not be so extensive as to limit the potential situations in which the component can be reused. There is generally a trade-off between simplicity and generality. Simpler interfaces make a component easier to reuse, because it is easier to provide the required connections. However, interfaces are the means whereby generality is supported—for example, by providing various parameters to tailor functionality. Before the interfaces can be identified, it is necessary to determine the degree of generality that is desired for the RSC, i.e., how much variation will be supported. Then each potential interface can be examined to determine if it is necessary to providing that generality. In some cases, the interface required to implement the desired generality may be so complex that the generality requirement must be relaxed.

G5.2.3.3 Where possible, use existing standard interfaces.

Some interfaces to well-known kinds of components have become industry standards; additional accepted standards may exist within a particular organization. An example of such an industry standard is the Structured Query Language (SQL) interface to relational data base management systems. Even if an off-the-shelf data base management system is not used, development of an SQL interface to the custom data base will permit later substitution of a commercial product.

G5.2.3.4 Specify standard interfaces so as to permit reuse of the specification.

Interface specifications for reusable components are themselves reusable entities. If the component is reused, its specification should be reused in the specification of the new system. Because the description of interfaces can be complex (for example, consider the description of a communications protocol), the interface should be available for physical extraction and insertion in a new document. Thus, its description should be easily isolated from other parts of the design specification and should be machine readable.

5.3 Designing for Modification

Reusable components must often be modified by the reuser; building in capability for modification must be considered at the design stage.

Reusable components can often be reused as they are, but in many cases some modification by the reuser will be necessary. This is particularly true over time, as requirements and technology evolve. Thus, modifiability is an important principle in designing for reuse. The designer of a reusable component can greatly enhance its reusability by giving attention to modifications that are likely to be required and designing to facilitate them.

5.3.1 Make design choices that facilitate modification.

Certain design choices can greatly facilitate modification by appropriate addition of generality and isolation of areas of likely change. The design of a reusable component should consider subsequent modification needs consistent with the anticipated reuse of the component.

G5.3.1.1 Identify potential/likely modifications when designing.

The need for modification of a reusable component can arise for a number of reasons. The designer of the RSC should devote some attention to explicitly identifying such reasons. Some questions that might be asked are:

- Will the RSC be ported to other platforms and/or operating systems?
- Is future commercially available software likely to be usable for some of the functions provided by the RSC?
- Will the RSC have to communicate with new kinds of devices not supported in its original design?
- Will workstation replacement impact the RSC?
- Will the RSC need to be adapted to a different user interface "look and feel" (e.g., a change from an Open Look to a Motif interface)?
- Will reuser systems require a different response to errors and anomalies?
- What kinds of functional additions are most likely?

Some of these anticipated changes may be handled by building flexibility into the RSC via parameterization. However, some will require the reuser to modify the software. These should be identified during the design phase.

G5.3.1.2 In designing for future modification, weigh development cost versus anticipated reuse benefits.

Building in the capability for modification can increase development cost, and should not be undertaken without consideration. Some guidelines to consider are:

- If supporting a modification initially is not more costly, do so. Don't create needless obstacles to reuse.
- If supporting a modification now is more costly but provides for a modification that is certain or very likely to be needed in a planned reuse, evaluate the cost of adding modifiability versus rewriting the component later. Generally, adding modifiability initially will be the better choice in this case.
- If supporting a modification now is more costly and addresses no known future need, but might generally increase the component's reuse potential, evaluate the modification based on general cost versus benefit to the organization, recognizing that it is an investment that may or may not be justified.

G5.3.1.3 Build in the capability for anticipated modifications.

Once anticipated modifications are identified and the decision to support them is made, the software design can take advantage of that knowledge. Areas of change should be clearly isolated and easy to find. For example, machine and operating system dependencies should be isolated for easy replacement. The layered architecture principles discussed earlier provide a mechanism for localizing areas of potential change.

5.3.2 Design for a long lifetime.

Reusable software components can be expected to have a long lifetime as they are reused from one system to the next. This means that they will be maintained, modified, and improved by many people. This demands that the software be particularly suited to modification—that it be easy to find the area to be changed, easy to formulate the required change, and easy to implement the change with "breaking" the rest of the software. Design for this long-term support is a major principle of developing reusable software.

G5.3.2.1 Follow established design principles that support modifiability and maintainability.

There are well-accepted principles of software design that lead to more modifiable, maintainable software. This manual includes some of these; others are available in software engineering texts. A specific reuse standard should be adopted for all RSC development. Verification of compliance with this standard should be part of the project Quality Assurance activity.

5.4 Design Methods

An appropriate design method can provide guidance in organizing software so as to enhance the reuse potential of individual components.

A design method appropriate to the development of reusable components will provide guidance for packaging software in self-contained objects or abstractions that can be reused with little or no modification. Most modern design methods address packaging in a way that supports reusability.

For example, object-oriented (OO) design methods provide such a focus, designing a system by identifying its constituent objects, and packaging each object with all its associated operations. The principle of OO design, closely related to earlier work in abstract data types, provides a natural basis for reuse. Components that are packaged in this way, because they are complete in themselves, can be more easily used in other contexts. Even when a full object-oriented approach is not used, following some of the key principles of OO design will support reusability. For a discussion of object-oriented design, the reader is referred to (Booch 87) or other OO texts.

More traditional structured decomposition methods can also support software reusability, by focusing on clean interfaces with low coupling and high cohesion. These methods give the

designer more leeway in deciding which entities to group together, and can lead to diminished reuse potential unless appropriate attention is given to packaging and abstraction principles.

Note that, while much of the guidance in this document refers to “objects”, the same principles apply to traditional structured methods.

5.4.1 Employ principles of packaging and abstraction.

The way a software system is organized—i.e. the way functions are allocated to software components—determines the reuse potential of those components. Modern software engineering principles provide guidance in organizing the software effectively. P*ackaging* refers to the way operations and data are grouped into software entities—i.e. into modules, subprograms, or packages. *Abstraction* refers to the “logical completeness” of such a package, and to the effectiveness of the way the package presents the user with necessary information while hiding implementation details. OO techniques provide an excellent way to design packages that implement useful, reusable abstractions.

G5.4.1.1 Package reusable objects with their associated operations.

This is the key OO packaging guideline, equally applicable to traditional design methods. OO packages are focused on objects; for example, a radar might be an object. All operations associated with the object are grouped with the object itself. A package implementing a radar object would include all the software routines necessary to control the radar and to process the signals it receives. Such a package, because of its logical completeness, clearly has high reuse potential. In the radar example, another system using the radar would very likely be able to reuse the radar package.

G5.4.1.2 Separate the package’s user view from implementation detail. Consider reuse when defining the user view.

The principles of abstraction and information hiding mean that the essential operations on the object are made visible to users of the package, while the details necessary to implement those operations are hidden. Thus, in the radar package example, the user would be able to create radar objects to correspond to the radars in his system and to carry out the operations required to control the radars and collect the data they return. He would not need to be concerned with internal details, like the way I/O to the radar is handled or the way data is represented within the package. However, it requires some thought to design an effective user view. The radar package returns complex information to the calling program—the signal information returned by the radar. There are many ways this can be done. At what intervals will the radar data be read? Should one call return information from only a single scan, or should data be collected in a buffer that can be made accessible to the caller? Should the radar package attempt to correlate the data, or will the caller do that? What checking will be done in the radar package, and what by the caller? How will the signal data be represented? The answers to these questions should be selected with reuse in mind. It is not enough to interface effectively with the system currently being designed; future systems’ needs should be considered.

G5.4.1.3 Identify opportunities for inheritance.

OO design principles view objects as potentially existing in a hierarchical organization, such that lower-level objects are refinements of, and inherit characteristics of, a higher level object. For example, an object class "animal" might have a subordinate class "bird". "Bird" would inherit properties and functions from "animal", and have additional ones appropriate only to birds. In the radar example, a general radar object might be provided, with a subordinate object for a particular type of radar. The subordinate object would provide only those operations unique to that particular type of radar, inheriting the general radar-handling operations from its parent type. Clearly this sort of structure leads to reuse. In a system with five different radar types, much of the code need exist only once, in the general radar package. Furthermore, the general package has reuse potential even in systems with different types of radars than those in the original system.

G5.4.1.4 Avoid design constructs that do not map to implementation language (e.g. multiple inheritance to Ada).

Some programming languages are explicitly designed to implement particular concepts. For example, an OO language will have explicit mechanisms for defining object classes and class relationships, and for implementing inheritance. However, the chosen implementation language may not fully support all of the desired design principles. In particular, many (including Ada) do not support multiple inheritance—an inheritance hierarchy involving more than one level of subordination. It is important when selecting a design approach to understand how design constructs will map to the implementation language. Some OO concepts can be mapped to languages that do not support them directly. For example, languages that do not explicitly support "packages" as a program component can still support a package organization with the addition of some human procedures to implement needed controls. However, a design that depends on multiple inheritance will not easily map to a traditional language. In selecting a project design approach, this issue should be considered.

G5.4.1.5 Consider the use of an accepted OO methodology.

As noted in the preceding guidelines, adoption of some OO-related principles is important to facilitate reuse. A somewhat separate question is whether a project should select a full object-oriented approach. This would constrain all of the software to be designed according to OO principles, and would assume mapping to a full OO implementation. Such an approach can be effective, but it requires a significantly greater change to the existing mindset of software engineers. It is not hard to adopt some of the key principles as a packaging guide, but viewing the entire design as a set of interacting objects is a challenge to traditional engineers. A move toward full OO design and implementation must be accompanied by training, as well as by some ongoing support from an experienced source during early stages of use. A full OO approach, if used effectively, will lead to more overall reuse potential for the software; used ineffectively, it will make no difference.

5.5 Designs as RSCs

Design specifications have reuse potential in their own right, and should be treated as RSCs.

Design specifications can clearly be reused when the associated RSC is reused. They also have potential for reuse when the code itself is not reused (perhaps, for example, because of a different language requirement). It is important to organize and manage design specifications to facilitate this.

5.5.1 Express component designs so as to facilitate their reuse.

Designs should be expressed so that individual design components can be easily extracted and incorporated into other design specifications.

G5.5.1.1 Employ a uniform design notation.

Reuse of design specifications (and often, therefore, of the resulting code) can be inhibited if the design specification is expressed inconsistently with the specification for the potential reuser system. Adopting a uniform style for structuring, formatting, and describing design components in specifications can facilitate reuse across an organization.

G5.5.1.2 Represent designs in a machine-readable form.

Extraction and reuse of design specification components is much easier if it can be done automatically. If design specifications must be retyped, there is a likelihood that changes will be introduced.

G5.5.1.3 Provide a means of linking the design RSC to associated requirements and code RSCs.

Design documentation should explicitly identify the requirement(s) that resulted in the design component, and should clearly indicate which code RSCs will implement the design. This will facilitate reuse of corresponding RSCs, and will also provide needed linkage information for later incorporation of the RSCs in a reuse library.

5.6 Selecting CASE Tools

Software engineering tools can do much to support compliance with this standard and the resulting creation of reusable software components.

CASE tools include tools supporting requirements definition, design, and coding. Tools of particular importance to reuse are those that support or enforce design methods and principles. Compiler selection also deserves attention, as it has a significant impact on efficiency tradeoffs.

5.6.1 Use design tools to implement a consistent methodology.

The use of well-chosen CASE tools can support the creation of reusable components. Many CASE tools are commercially available. Each supports one or more aspects of the software life cycle and each makes some decisions about the methodology to be employed. It can be difficult

for the CASE "shopper" to select a suite of tools that work well together to support a selected methodology. It is all too easy to select several tools, that, while effective individually, are virtually impossible to integrate. The best approach is to first establish the methodology suitable for the overall project, considering all activities from requirements analysis to maintenance, and to then select tools that can be integrated to support that selected methodology in a consistent manner. It is generally better to do without automated support than to attempt to use a tool that does not fit the overall methodology; use of a badly fitted tool can actually be harmful. While the overall problem of constructing an effective tool set is beyond the scope of this manual, the following guidelines identify some of the points to be considered in ensuring that the tool set supports reuse goals.

G5.6.1.1 Provide for a machine-readable and portable design representation.

As noted above, machine-readable designs facilitate design, and hence code, reuse. This is straightforward when dealing with text. However, most modern CASE tools include graphic descriptions of component designs and interrelationships. These can be difficult to extract and incorporate into other specifications, particularly if the reusing system is not using the same design tool. Ideally, the design tool should produce graphics in some transportable form. At a minimum, they should be compatible with conventional word processors or publishing systems on the particular computer system.

G5.6.1.2 Support construction of a larger component from smaller components.

Reuse involves the composition of components—building a larger component from smaller building blocks. This composition is performed at the design level, as the desired reusable parts are identified. The design tool should allow design representations to be merged in some way. This includes the capability to integrate the graphic depiction of one component in the depiction of another.

G5.6.1.3 Include a mechanism for aiding and/or guaranteeing requirements/design/code/test correspondence.

Earlier guidelines have noted the importance of requirements/design/code/test correspondence. CASE tools that support the generation and management of specifications and code often provide automated mechanisms for representing such correspondence, and sometimes for checking it (e.g. for ensuring that each requirement maps to one or more design components). These capabilities, if available, are particularly supportive of reuse objectives.

G5.6.1.4 Consider tools that can automatically check for conformance to at least portions of the coding standard.

This manual sets forth, in the next section and in the appendix, a standard for writing reusable code. Some of the specified requirements can be either automatically enforced or automatically checked. Including such a capability can help ensure compliance and cut down on some of the manual checking required.

5.6.2 When selecting a compiler, consider its support for software reusability.

There is, in general, a trade-off between the generality desired in a reusable component and its runtime performance. Software designed for reusability will often have capabilities that are unnecessary for a particular user of that component. This can lead to memory-utilization and/or execution-time inefficiency. However, the compiler selection can help minimize the impact of reuse features. (Proper use of the programming language also helps, as discussed in later sections.)

G5.6.2.1 Examine the compiler's optimization capabilities.

Reuse can lead to additional code, both because unwanted capabilities are provided and because some decisions are deferred to runtime (e.g. "if this is message type X, then check ..."). Much of this excess code can be eliminated by a compiler with good optimization capabilities. Some of the specific capabilities to examine are:

- Constant folding. This is an optimization that replaces a variable by an expression that represents its value. This is effective, for example, when an RSC has a parameter that is not constant for all uses, but is constant for any single use. Constant folding can allow the compiler, if it can be made aware of the specific value, to propagate that value throughout the code, eliminating many computations.

- Dead code elimination. This optimization eliminates code that the compiler can determine will never be executed. This works in conjunction with a constant to eliminate entire code paths. In the case of "If this is message type X, then ...", the compiler may be able to determine that that message type will not appear in this particular use of the RSC, and thus to eliminate the check itself and all code to be performed as a result of the check.

- Support for inline procedures. Some languages (including Ada) allow the programmer to specify that a particular procedure is to be effectively inserted inline in the object program when it is called, rather than the usual approach of generating a call to a single instance of that procedure. This capability, which trades space for time, is appropriate for small procedures. The extra memory taken up by the inline expansion can be preferred to the procedure call overhead. Inline procedures provide more opportunities for constant folding, and can be effective in handling some of the small procedures that can result from following generality principles.

- Optimization for generic procedures. Some languages (including Ada) include a construct called a generic procedure. A generic is, in a sense, a procedure template that can be used to build specific procedures depending on the data to be handled. Such a specific procedure is created by *instantiating* the generic. For example, a generic sort procedure would include all the mechanisms required to sort a list of values. It would then be instantiated for each particular data type for which a sort procedure was desired, e.g. character strings, fixed point numbers, or personnel records. Such instantiation is explicitly called for by the programmer when developing his program. Generics thus provide parameterization in program construction, rather than at runtime. Generics are a

valuable capability for reuse, but compilers differ in their ability to optimize. Some considerations are whether or not a compiler allows multiple similar generic instantiations to be implemented as a call to a single copy (a difficult optimization) and to what extent generic expansion takes advantage of constant folding and dead-code elimination opportunities.

Compiler vendors will occasionally implement special optimizations requested by a customer—for example, to take advantage of a particular hardware feature or to support a particular usage style. If one's organization is a major customer of the vendor, such capabilities are worth considering.

G5.6.2.2 Examine the compiler's capability to limit the units to be included as the result of a link.

Normally, a compiler and linker will include all referenced program units whether or not they are needed. A linker that can eliminate those units not actually used can save significant code space. For example, a programmer using only a SINE function from a package of transcendental functions, with an intelligent linker, would need to incorporate only the SINE function and any subprograms it references. This seemingly simple improvement can be difficult to implement in cases where subprogram dependencies are complex. When examining a compiler, the question is not simply "Does the linker eliminate unnecessary program units?", but "To what extent does it do so?"

Section 6

Detailed Design and Implementation

The detailed design and implementation activities must carry out the design decisions made earlier, at the same time incorporating characteristics that specifically contribute to component reusability.

Detailed design and implementation are the activities that actually produce reusable code components. It is important that a specific standard be followed to ensure that none of the planned-for benefits is in fact lost by inappropriate mapping to code, and to maximize reuse potential of the software.

The following subsections address transition from design to code, program structuring, interfaces, parameterization, handling errors and exceptional conditions, and efficiency.

The guidance presented in this section is, insofar as possible, language independent, so that the manual is usable with different languages. Specific guidance for the use of Ada is included as Appendix A.

6.1 Transition from Design to Code

Transition from design to code must follow through with and build upon the structures defined earlier, and must recognize that code quality is of vital importance in reuse.

Many of the key software structuring and packaging decisions have been made in earlier phases. In the transition from design to code, it is important to preserve these decisions in the implementation. Code quality is also extremely important. Reusable components must be particularly robust, consistent, and maintainable. A detailed standard is essential to achieving this quality.

6.1.1 Maintain correspondence from design to code.

Correspondence from design to code must be correct and traceable. It must preserve all decisions made earlier, in particular those affecting the allocation of function to software components and the specification of interfaces. Leeway is permissible in the implementation detail, but the "block box" form and function of the component must exactly comply with the specification. Any deviation must be handled through an established configuration management process so that requirements and design specifications are updated to reflect the change.

G6.1.1.1 Consider automated support.

Today some CASE tools are beginning to be available to assist in the mapping from design to code. For example, such a tool might construct templates for each procedure,

including all parameterization identified in the design specification, and might provide the control flow constructs to implement a design algorithm. Use of a tool like this can help ensure that code corresponds to design, and can save effort during the coding phase. Automated tools can also, as noted earlier, help with automated traceability from design to code.

G6.1.1.2 Consider the use of specific mapping guidelines appropriate to the design method and source language used.

In the implementation phase, the software engineer's task is to map each design entity to an implementation in the source language. With a well-defined design notation (e.g., a program design language or a graphic representation), it is possible to specify explicit guidelines for mapping design constructs to the source language. For example, if the design approach uses object-oriented packaging concepts, it would be necessary to specify the mapping of these packages to a FORTRAN implementation. If the design is specified in an Ada Program Design Language, implementation in another language (e.g., PASCAL or C) would require guidance in how to map such features as Ada generics.

6.1.2 Emphasize code quality and reusability properties.

Many of the key decisions affecting reusability—for example, the definition of RSCs and their interfaces—will have been made in earlier life-cycle phases and need only be followed through on at this point. However, the detailed design and coding phases still have an important role in implementing the detailed quality and reusability properties required in an RSC. An RSC must be of exceptionally high quality; it must be robust, must perform efficiently, and must be easy to modify and maintain. It must also conform to special requirements specifically intended to build in the flexibility and generality necessary for reuse.

G6.1.2.1 Employ a detailed-design and coding standard to maximize reusability.

A concise standard should be employed to ensure the desired code quality and reusability. This manual provides such a standard. This section includes language-independent guidance for detailed design and implementation; language-specific guidance is provided in Appendix A. A project should tailor this guidance to its specific situation.

G6.1.2.2 In selecting the programming language, consider quality and reusability factors.

If a project has the opportunity to select the programming language to be used, it is desirable to consider the support for high-quality, reusable component construction provided by a candidate language. Some of the considerations include:

- Its support for the structuring concepts described in this and the preceding section (e.g., object-oriented programming, implementation hiding, packaging and abstraction)

- Its support for extended parameterization through a facility such as Ada's generic procedures

- Its ability to guarantee error-free, robust software through such mechanisms as type checking
- Its support for the creation of readable, modifiable, and maintainable programs (e.g., structured programming support, readable identifiers, and modularity)

Also, the anticipated language requirements of future systems should be considered. Other things being equal, it is clearly best to select a language that future systems will use if reuse is an objective.

6.2 Program Structuring

Choice of program structuring determines the reuse potential of individual components.

From the reuse perspective, the goal of program structuring (grouping functions and interfaces into individual software components) is to group together those entities that will typically be reused together, minimizing dependence on those entities that are not likely to be involved in that reuse. Traditional software engineering principles of modularity, information hiding, and separation of concerns apply.

6.2.1 Carry through design principles in the implementation.

The design developed according to these guidelines will incorporate design principles influencing the organization of software functions and data. The implementation must retain these properties. This requires particular attention when the language does not directly support some of the design constructs.

G6.2.1.1 Map design constructs to implementation language.

A specific programming language mapping should be defined for design constructs. In particular, it is important to understand:

- How will an object be represented in the programming language? If a single language entity corresponding to an object (i.e., a grouping of the object's definition and operations) is not available, so that the object must in fact consist of several program entities, what human or configuration management procedures are needed to ensure the object's integrity?
- Will inheritance be implemented? If so, how? Does the language provide the desired capability, or must some preprocessor support be provided? Are human controls necessary to ensure it is done correctly?
- How will information hiding be implemented? If the language does not provide an explicit mechanism to restrict visibility, how can one ensure that no programs access "hidden" detail?

Answers to all these questions should be developed before beginning implementation.

G6.2.1.2 Each RSC should implement a single, complete object.

An RSC should implement a complete object (i.e., an abstraction). It should provide the reuser with the full set of operations needed to create and manipulate the object. Without this completeness, users may find it necessary to modify the RSC, resulting in additional work and loss of commonality. Such a complete object should include operations (i.e. subprograms) for:

- Creation—includes both creating and initializing an object
- termination— provides a means of ending the life of the object, essentially making it inaccessible in the remainder of its scope
- Conversion—allows for the change of representation from one abstract type to another. For example, a conversion function would take as its input two fixed point numbers and produce as its output a single value of a programmer-defined type *Complex*. In this case, the internal representation of the complex object is not accessible for user manipulation.
- State inquiry—allows a user to inquire about boundary conditions. Consider an abstract file type. Boundary conditions cover whether the file is empty or whether its maximum capacity has been reached. Other boundary conditions refer to end-of-line and end-of-page states. Non-boundary state-inquiry functions might return the position in the file (line and column number), the status of the file (open, closed), and the mode of the file (read-only, write-only, read-and-write).
- Input/output representations—support printing or modification of object values for debugging purposes. The objective is not to debug the object package but to allow the user to debug his application. For example, printing out an entire stack, he may discover that he has omitted a “push” operation.
- State change—allows the user to modify the contents of an abstract object—e.g. to negate a complex number

G6.2.1.3 Separate externally visible characteristics and operations from implementation-specific details.

The principles of layered design, separation of concerns, and information hiding all essentially refer to the concept of making certain information visible (i.e. accessible) to reusers of an RSC while hiding (making inaccessible) other information. This supports the view of an RSC as a “software chip”—a black box whose function and interfaces are well-defined while no knowledge of its implementation is required. For example, a stack package would provide capability for a reuser to create objects of stack type, and would provide all the needed operations to manipulate the stack. However, the reuser need never know whether the stack is implemented as an array, a linked list, or any other mechanism. This separation allows the implementation details to change without requiring changes to programs that use the package.

When implementing an RSC, it is important to clearly identify those characteristics that should be visible from those that can and should be hidden. The programming language

should then be used to effect this separation. As noted above, this may require project control procedures to augment the language capabilities.

6.2.2 Control machine and implementation dependencies.

Machine and implementation dependencies limit the scope of reuse of an RSC. Components with such dependencies can be reused, but only by replacing the dependent portions with code suitable to the new situation. Such dependencies should be minimized; where they exist, they must be isolated and documented.

G6.2.2.1 Minimize dependencies within RSCs.

The previously presented design guidelines suggest that not all parts of a system can realistically be planned as RSCs. Intended RSCs should be identified during the requirements and top-level phases, and implemented according to this standard. When creating the initial design architecture, it is appropriate to separate activities with machine, operating system, or other dependencies from those with more general reuse potential. Sometimes this will result in RSCs with no dependencies. Often, though, implementing a complete abstraction will require the inclusion of some dependencies (e.g., dependence on a particular data base management system interface). These should be minimized; when necessary they should be treated according to the next two guidelines.

G6.2.2.2 Isolate dependencies that must occur within RSCs.

When a dependency is necessary, it should not permeate the RSC, but should be isolated as well as possible to minimize the change required to replace the dependent code. For example, suppose a message handling package makes use of a relational DBMS to store messages. Many operations in the package will need to access the DBMS. However, including DBMS calls throughout the code can create a major dependency on the particular DBMS. The objective is to be able to replace the DBMS without having to change all of the procedures in the package; thus a single format for DBMS calls is required. This can be implemented by building a “filter routine” that all the other procedures actually call, which in turn generates the appropriate calls to the particular DBMS. If the DBMS is replaced, changes are localized in the filter routine. Figure 6.1 illustrates this approach. (In the specific case of the DBMS, the SQL interface standard has been developed precisely to provide such a uniform interface.)

G6.2.2.3 Document all machine and implementation dependencies.

Once the dependencies in an RSC have been identified and localized, they should be documented for the reuser. This documentation should, first, clearly identify any dependencies the RSC has. This information should be in a consistent place with each RSC, as it is one of the first things a potential reuser will want to know. Second, the documentation should explain how to make the necessary modifications to the isolated dependent code. In the DBMS example above, this documentation would describe how to implement a replacement filter package when the DBMS is replaced. It would also provide an estimate of how much work is required to make the change.

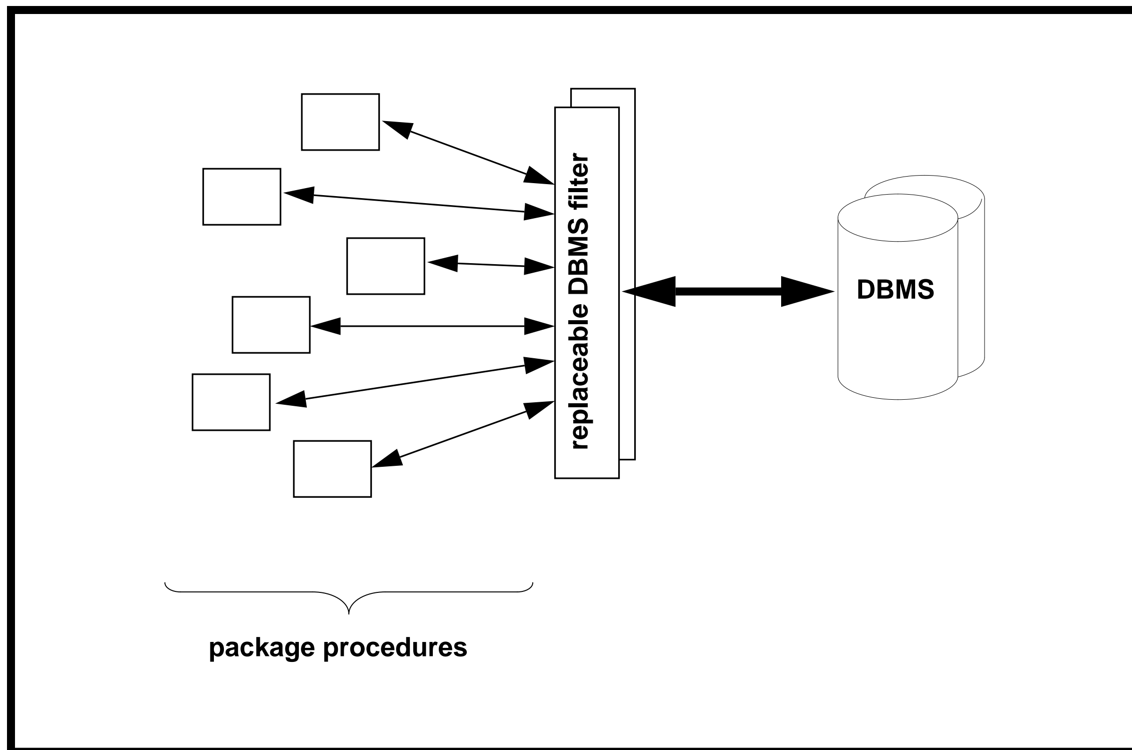


Figure 6.2 - Using a Filter Routine to Localize Dependencies

6.3 Interfaces

A component's interfaces, which define its connections to the outside world, establish the framework within which it can be reused.

In a sense, a component is defined by its interfaces, in that it is completely specified by them. For the potential reuser of the component, the interface definition provides a basis for determining whether reuse is possible and a guide for using it if it is.

6.3.1 Develop well-documented interface specifications for all potentially reusable components.

The physical integration of an RSC into a new application requires the reuser to supply a correct matching interface for each interface of the RSC. This requires a full understanding of exactly what interfaces exist and of how each must be treated. It is important to recognize that interfaces are not only explicit subprogram parameters; any coupling of the RSC to its environment constitutes an interface. This includes such constructs as shared global data and task interaction.

G6.3.1.1 Explicitly enumerate the interfaces in the component's prologue and documentation.

The interface specification provides the "directions" for component installation. It should be presented in a consistent format both in the source program itself (in the prologue, the comment lines preceding the rest of the source program) and in the

component documentation. It should include all interfaces that exist, whether explicit parameters or otherwise.

G6.3.1.2 For each interface specified, provide a text description of its meaning, a type specification, a range of allowable values, and an explanation of what will occur when a value is outside the allowable range.

A consistent format should be selected for presenting the indicated information. The meaning of the interface should tell the user how the particular interface influences the functioning of the RSC. The type specification should be in terms of the source language, so that the user knows how to declare a variable that matches the RSC's expectation. Range and boundary conditions explain the limits of applicability of the component.

G6.3.1.3 In the interface descriptions, include the types of interfaces identified in Table 6.1.

This table provides a checklist for ensuring that all interfaces are recognized and properly documented.

1. RSC is called via subprogram call by reuser.
2. RSC calls subprogram of reuser.
3. RSC is a task invoked by a reuser task.
4. RSC is a task that invokes a reuser task.
5. RSC shares memory with a reuser subprogram.
6. RSC is a task sharing memory with a reuser task.
7. RSC communicates with reuser via a shared file, with one always reading and the other always writing.
8. RSC communicates with reuser via a shared file with simultaneous access by both.
9. RSC communicates with reuser via a message passing or "mailbox" mechanism.

Table 6.1 - A Taxonomy of RSC Interface Types

G6.3.1.4 Consider providing a formal algorithmic description of the relationship between input and output parameters.

The relationship between input and output parameters defines the RSC in the sense of "if I put this in, I can expect this out". Depending on the nature of the RSC, this relationship can often be expressed algorithmically or formally in a way that provides a very good, concise description of the component's function. This is particularly true in components that perform mathematical operations.

6.3.2 Provide only the necessary generality; fewer, simpler interfaces support reuse more effectively.

In general, a component with fewer, simpler interfaces is more reusable than one with a greater number of more complex interfaces, because it is more easily incorporated into the new application. Fewer interfaces also reduce the potential for error. However, the interfaces must be adequate to support the required generality of function.

An example of a sufficient level of generality might be a polynomial evaluation function that is parameterized to allow the user to specify the order of the polynomial. An example of excess generality might be a function that was able to evaluate all possible mathematical expressions. Such a function would almost certainly have an excessively complex interface and involve excessive runtime overhead.

G6.3.2.1 Consider the principles of coupling and cohesion in designing component interfaces.

The concepts of *coupling* and *cohesion* are useful in considering the interfaces of a software component intended for reuse. Cohesion means that functions and data provided by the reusable component are closely related. A component with high cohesion is one that implements a single functional abstraction. Coupling refers to the extent to which a component depends on other components. A component with high coupling has many such dependencies, and a component with low coupling is one with simpler, fewer interfaces. Established measures of coupling and cohesion are an objective way of analyzing the reusability of a component from the standpoint of the necessity and sufficiency of its interfaces.

G6.3.2.2 Develop QA procedures that explicitly examine the interfaces of components intended for reuse.

Such procedures should assess each interface for its necessity and sufficiency to meet established requirements, and might require application of an explicit metric such as a coupling and cohesion measure.

6.3.3 Provide reuser "hooks" for dealing with boundary conditions.

In some cases, it is not clear in designing a reusable component what action the user would prefer in the event of a boundary condition. For example, if the reuser attempts to store an element in a queue that is full, he may wish to receive some signal of an error. He may also wish to expand the queue space or delete some earlier elements of the queue. A component that provides these alternatives in its user interface has additional flexibility. Each interface should be examined from the point of view of what alternative reuser actions might be desired, and an appropriate implementation selected to provide the desired flexibility.

G6.3.3.1 Allow the RSC reuser to provide a subprogram to be executed when boundary conditions occur.

If the programming language provides a mechanism for this, it can give a great deal of flexibility. There are two ways this can be accomplished. If the language allows subprograms as parameters to other subprograms, the RSC can call that subprogram

when the condition occurs. This approach allows the reuser program to tailor the boundary condition behaviour at runtime. Alternatively (e.g., in Ada), the RSC can be implemented as a generic, with the reuser's boundary condition subprogram provided as a generic formal subprogram parameter. This approach binds the boundary condition handling selection at compilation time. In a language without generics, a similar effect can be achieved with conditional compilation.

G6.3.3.2 Raise an exception that the RSC reuser can handle as appropriate to his needs.

If the programming language provides exceptions (a means of signalling an exceptional condition by transferring execution control to a routine specifically provided to handle that condition), this can be another way of handling boundary conditions flexibly. The RSC might also provide auxiliary routines for getting additional information about what caused the exception.

G6.3.3.3 Include output parameters that provide information about whether a subprogram completes its action appropriately, and if not, why not.

This approach does not explicitly provide the reuser with a way to control the behaviour of the RSC. It does, however, provide information he may need to react to its behaviour.

6.4 Parameterization

Specific use can be made of component parameterization to increase the component's reusability potential.

In addition to the usual parameterization required to implement a component's functionality, additional parameterization may be useful to enhance reusability.

6.4.1 Provide a means to supply system-specific parameter values prior to execution time.

Certain information about the system configuration or operating environment for an RSC is variable from one use of the RSC to another, but is fixed for any single use of the RSC. The RSC implementation should identify such information and implement a means for the reuser to provide that information when incorporating the RSC in his application. Providing such parameterization before execution time increases execution efficiency by permitting more compiler optimizations (see *G5.6.2.1*) to be performed.

G6.4.1.1 Use generic constants to supply system parameters.

If the selected programming has generic procedures, generic parameters provide a means to parameterize the component at the time the generic is instantiated. A generic constant parameter is simply a constant value supplied for each instantiation of the generic. For example, a generic stack package might include a constant parameter that specifies the stack depth. The programmer using that package to create a stack supplies the desired constant value for his application.

G6.4.1.2 Use compile-time variables or a source code preprocessor to supply system parameters.

These methods essentially provide parameterization that "modifies" the view of source code as seen by the compiler, without actually changing the source itself. Some languages (e.g. PL/I) provide compile-time variables that allow the compiler to substitute a constant value for the variable while it is compiling the code. Alternatively, a source code preprocessor can scan the source code and substitute a parameter value for some target string. Either method would allow an RSC that contains the parameter *Stack_Depth* to be compiled with a constant value for *Stack_Depth*, without any modification to the RSC source code.

G6.4.1.3 Identify required changes in RSC source code.

A final alternative is to require the reuser to supply the change in the RSC source code itself. This is less desirable, as it is better to avoid a requirement to modify the source. If this approach is used, it is important to make the directions very explicit in the RSC's documentation, in its header, and at the location of the source to be changed. Comments like "!!! WARNING—REPLACE THIS WITH AN INTEGER CONSTANT INDICATING STACK DEPTH !!!" are appropriate. Also, the RSC implementation should include a check to be sure that a value was supplied, so that the RSC does not attempt to execute with an unknown or zero value.

6.4.2 Ensure that all objects are initialized.

A package implementing an object provides a capability for a reuser to create objects (program variables with appropriate associated operations) of that type. Safety of the package is improved by making it impossible for the reuser to create uninitialized objects. Initialization should be a necessary part of object creation, rather than a separate and therefore optional activity. If the package does not guarantee initialization when creating objects, it would have to check for it on every call to one of the operations on the object, an obviously inefficient approach.

G6.4.2.1 Provide default values for each element of the object type in the type definition.

The easiest way to initialize objects, if the language permits it, is for the package implementing an object type to specify that type with default initial values. All objects declared with that type simply acquire the initial value.

G6.4.2.2 Execute initialization code as an automatic (i.e., outside the control of the reuser) part of object creation.

Another way the object package can initialize objects is to provide an explicit initialization routine that it calls as part of the creation process. This is appropriate either when default values as part of the declaration are not supported by the language, or when the desired initial values are not readily specified as constants, but require some form of computation.

G6.4.2.3 Require initial values to be supplied as part of the object creation operation.

A third alternative is to require the reuser to supply the initial value as parameters in object creation. The object creation operation would have additional parameters for each element of the type. For example, a call like

```
New_Value := Complex_Number (3,-5);
```

would create a complex_number object called *New_Value* with initial value 3-5i.

6.4.3 Provide a means to remove unnecessary checks.

A "safe" reusable component will likely include a number of checks to ensure that the component does not accidentally crash. For example, there may be checks on expressions for division by zero, for exceeding some bounds (e.g., -pi to +pi), for checking that a queue is empty or full, for hardware read errors when reading from disk, etc. Depending on the instantiation of the RSC, these checks may not apply because the case being checked for can be guaranteed to never arise. An optimizing compiler may not necessarily recognize such checks as dead code. In order to help the compiler remove these checks and make the resulting code more efficient, some means of indicating that a check may be suppressed is desirable.

G6.4.3.1 Use a generic Boolean constant to suppress unwanted safety checks.

A generic Boolean constant (see the description of generic constants in *G6.4.1.1*) can cause execution of unwanted checks to be skipped at runtime. The compiled code will effectively be "***if*** *FALSE*, ***then*** *check* ...". A good compiler will remove the ***if*** statement, as well as the checking code.

G6.4.3.2 Use a compile-time variable or a preprocessor to remove unwanted safety checks.

A compile-time variable or a preprocessor (see *G6.4.1.2*) can be used to cause the compiler to not compile the checking code based on a value supplied at compilation time.

6.5 Handling Errors and Exceptional Conditions

Reusable components must be especially robust and safe; attention to error handling adds assurance that they will be.

Users of a reusable component can be expected to differ in their desire for what should happen if the component encounters an error. The component should provide the flexibility to meet these different needs.

Note: Several of the following paragraphs refer to the concept of *raising an exception*. This capability (sometimes referred to by other names) is explicitly present in many, though not all, programming languages (including Ada). Workarounds can allow a similar capability in other languages, as noted in the first guideline below. It refers to the capability to specify blocks of

code (called exception handlers) to which control will be transferred in the event of certain types of execution errors. This transfer may happen automatically (i.e., under the control of the runtime system)—for example, on detection of an attempt to divide by zero. It may also happen under program control. A procedure may—for example, on detecting an invalid parameter value—explicitly raise an exception. Exception handlers can occur in the subprogram that raises the exception, or in the calling program (and so on up the chain). Normally, reusable subprograms will return exceptions to the caller.

6.5.1 Provide a mechanism for returning control to the calling program when parameters are invalid or other anomalous conditions occur.

Reusable components must be "safe", i.e., they should guarantee against crashing. One aspect of implementing this safety is to reject calls that will result in failure. This should be done in an orderly way, returning control to the calling program with some indication of what is wrong.

G6.5.1.1 When the language provides exceptions, use them to inform the caller of problems.

Exceptions provide the cleanest way of handling problem conditions. They are clearly distinguished from other code, they isolate exception handling behaviour so it can be easily modified, and they provide the caller with flexibility in handling the problem.

G6.5.1.2 If exceptions are not supported, but procedure parameters are, use a procedure parameter to indicate a reuser routine to call in the event of error.

Some languages allow procedure names to be passed as parameters, permitting the called RSC to call that procedure. This is an effective way to allow the user to specify an action—a procedure to be called—when problems occur. Naming conventions should be used to clearly indicate that this parameter is an error-handling procedure.

G6.5.1.3 If no other mechanisms are available, use return-code parameters to indicate problem conditions.

Return-code parameters are status or indicator parameters returned by an RSC to indicate successful or unsuccessful completion, and to indicate reasons for unsuccessful completion. This mechanism provides the desired capability to the calling program, but it does not require the calling program to use it. If the calling program does not include code to check the return parameter value, it will not be informed of the problem. Correct behaviour by the calling program requires a check after each call to the RSC, which is somewhat time-consuming and unwieldy. A lexical convention should be used to distinguish these parameters as return code parameters.

6.5.2 Use exception handling to implement "safe" RSCs.

Exception handling, by whatever mechanism chosen to implement it, should be used to ensure the safety of an RSC and to provide the reuser with the information needed to support error-free use in all cases.

G6.5.2.1 Provide checks for all assumptions the component depends on to operate correctly.

Instead of defining an RSC such that its reuser is responsible for ensuring that certain assumptions are satisfied, specify exceptions that are raised when these assumptions are violated. For example, instead of saying "Don't call the *Stack.Pop* function if the stack is empty," say "*Stack.Pop* raises *Stack.Empty* if called when the stack is empty." Raising an exception allows the reuser to decide what to do about the situation. In essence, the usability of the RSC is extended to include empty stacks.

G6.5.2.2 Recognize instances in which runtime checks are too costly, and instead raise an exception if a violation occurs.

In some cases the preceding guideline is costly to implement, because some assumptions are expensive to check. For example, a lookup RSC that uses binary search must assume that the table being searched is sorted. If it is not sorted, the search will return incorrect results. Checking that the table is sorted is expensive, and since speed is the reason for using binary search, making such a check would be contrary to the purpose of the RSC. Instead, an exception can be raised when unordered table elements are encountered during execution.

G6.5.2.3 Where possible, provide an alternative means to avoid raising an exception.

Exceptions are raised to indicate the existence of conditions that prevent an RSC from producing its normally expected result. An RSC is more reusable (reusable in a wider variety of situations) if functions are available to check for the presence of such a condition in advance of calling the RSC. For example, a stack package should, in addition to the *Stack.Empty* exception, provide a function *Stack.Is_Empty*, which returns *True* if stack is empty. Similarly, a file handling package should provide the function *Is_End_of_File* as well as the exception *End_of_File* to indicate when there are no more items to be read. These functions help control program logic, and are particularly useful in a tasking environment—e.g., for specifying a guard on an Ada ***accept*** statement:

```
select
    when not Stack.Is_Empty =>   -- closed if nothing is on stack
    accept Get (...) do
        Item := Stack.Pop;             -- no exception will be raised
    end Get;
or
    when not Stack.Is_Full =>       -- closed if stack is full
    accept Put (...) do
        Stack.Push (Item);             -- no exception will be raised
    end Put;
end select;
```

If such functions are not available, it is awkward to provide the equivalent capability by just using exceptions.

Note, however, that such functions are not practical if the amount of computation needed to decide whether the exception will be raised is comparable to performing the

operation that will raise the exception. For example, in a matrix inversion RSC, it is not practical to try to determine in advance if the matrix has an inverse, since the way to do that is to try to invert the matrix. In such a case, simply raise an exception when executing the function.

G6.5.2.4 If reinvocation after an exception is raised would be too costly, consider an alternative mechanism to allow the user to correct the error.

One purpose in raising exceptions is to allow the user to correct the condition that caused the exception and then retry the operation (as opposed to simply trying to fix the problem within the RSC). This is practical if it is relatively cheap to retry the operation. For example, when writing a tape, if the end-of-volume is encountered, it is quite reasonable to raise an exception, since not much work is lost by calling the write routine again after a new tape has been mounted. In some cases, however, considerable computation will be lost. In this case, it is better to attempt a fix by calling a reuser-provided procedure (specified as a procedure parameter).

G6.5.2.5 When additional information is available describing the nature of an exception situation, provide a subprogram to return all available information.

It is sometimes the case that a lot of information can be provided to a reuser when an exception situation is encountered. For example, tape drives can signal a variety of error conditions, ranging from lack of a write ring to parity error, to end of tape, etc. Instead of defining one exception for each possible error condition, it may be reasonable to handle all the situations with a single exceptions—e.g., DEVICE_ERROR—and then provide an additional subprogram that can be called to obtain all information available about the reason for raising an exception. Providing such a subprogram is more general than providing a global variable that can be read after an exception is raised, since any computation involved in producing the information need not be done unless the reuser wants the information. If such a subprogram is provided, it should be defined as part of the RSC interface.

6.6 Efficiency

Performance penalties can result from added generality, and can discourage reuse; these can be largely overcome by appropriate development techniques.

Reusability can impact performance by adding runtime decision-making and hence additional processing. These difficulties can be addressed by effective choice of language constructs, by choice of binding time, by using language processing tools effectively, and by allowing for tuning by the reuser.

6.6.1 Use the implementation language effectively to minimize performance penalties.

Attention to performance impact during RSC design and development can go a long way toward minimizing performance penalties from reuse. It is important to select each implementation construct with a view to performance implications.

G6.6.1.1 Recognize high-impact constructs.

In any programming language, some constructs involve far more execution and space overhead than others. To some extent, this depends on the compiler (Guidance on compiler selection criteria is given in *G5.6.2.1.*); however, much of the information is independent of the implementation, and perhaps even of the language. Some constructs are inherently more expensive to implement. It is best to try to obtain specific information about this from the compiler vendor. However, some general rules of thumb are:

- Subprogram calls are expensive, due to the need to save and restore values of variables. If a subprogram activity involves only a few lines of code, an inline expansion approach will be more efficient. (Note that Ada ***inline*** procedures have the performance benefit of inline code without the negative impact on modularity and readability.)

- Tasking operations are expensive, due to the need to perform context switching. Some compilers provide a “fast task” that eliminates some of the overhead of more general tasks. If available and appropriate to the particular need, these can effectively reduce overhead, but at the expense of portability.

- Operations on data aggregates (records and arrays) are more expensive than operations on scalars. This is fairly obvious, but can be forgotten because the source code can still appear simple. A single-line assignment statement, if assigning one array of records to another, can result in the execution of hundreds or thousands of machine instructions.

- Exceptions are more expensive than parameter passing.

- Variable-length strings and aggregates are more expensive than fixed-length strings and aggregates.

- Linked-list data structures are more expensive than arrays.

- Hidden checks—e.g., to implement exception handling by testing for such situations as zero-divide before they occur—add cost.

G6.6.1.2 Where possible, allow reuser to make decisions prior to runtime.

While it is important to build generality into reusable components, it is desirable, where possible, to allow the reuser to customize the component prior to runtime. A primary way of doing this, if supported by the language, is by the use of generic procedures. Generics are instantiated with many of the reuser’s specific requirements, avoiding the need to test for alternatives at runtime. Alternative approaches include compiler-time variables and preprocessors. These are described in *G6.4.1.2*. Any such techniques to

bind parameter values before runtime will eliminate unnecessary checks during execution.

6.6.2 Provide opportunities for performance tuning.

Design RSCs for flexibility, modifiability, maintainability, and robustness. Recognize that performance tuning can occur later. Don't shortcut design goals. Instead, provide opportunities for subsequent performance tuning.

G6.6.2.1 Give the reuser the ability to tune later.

In many cases, the reuser can tune RSC performance simply by working with the existing parameterization capabilities. We have already presented a number of ways the RSC can support reuser tuning—e.g., by providing parameter values that suppress unnecessary checks and by providing alternative mechanisms for handling error conditions. Sometimes, however, this will not be enough; the reuser will want to actually modify the source code, perhaps even replacing key low-level routines by machine-code implementations. It is important to help the reuser who has this need to do so without "breaking" the code. Ways to do this include:

- Provide explicit guidance in what parts of the RSC have the greatest performance impact. It might be desirable to consider running an analysis to determine the execution-heavy areas. (For example, this might be a service performed by a reuse library organization.)

- Provide guidance in what code can be eliminated if the reuser's needs are specialized or if he is willing to guarantee correctness in certain areas and hence dispense with safety checks. (But be sure the risk of this is noted.)

- Identify any performance assumptions that are likely to be dependent on a particular platform, compiler, or runtime. Suggest any alternatives that might help performance if any of these are changed. (Often the original programmer has the best knowledge of this; he should capture that knowledge in guidance to the reuser.)

G6.6.2.2 Help the compiler optimize by correct choice of language constructs.

Guideline G.5.6.2.1 briefly discussed some common compiler optimization techniques. In general, most compilers can be expected to implement such techniques as constant folding and dead code elimination, though the extent of the support varies. A compiler's optimization opportunity is determined by the information available to it at runtime. In particular, any computation for which the compiler can already predict the result need not be performed. As noted earlier, for example, generic constant parameters turn into constant values (rather than variables) at runtime, thus providing code elimination opportunities. Often compiler vendors will provide some guidance to users on how to help the optimizer. These are worth following even if the RSC might be ported to an environment with a different compiler; most compiler developers employ similar optimization techniques.

6.7 Detailed Coding Standard

A language-specific coding standard provides detailed guidance for implementing the principles already identified.

The body of this manual is a standard that deals with requirements, design, and other issues that are largely independent of choice of programming language. To carry through these principles to the implementation, a language-specific coding standard is appropriate. Appendix A contains a coding standard for Ada. This manual can be used with other languages by replacing Appendix A with standards appropriate to those other languages.

Note: Techniques for implementing reusable code are, in many cases, identical to those for implementing *good* code. The same properties that make code good—modularity, maintainability, and portability—contribute to its quality and utility as an RSC. Appendix A includes many guidelines that support these general quality goals, in addition to those directly motivated by reusability.

Section 7

Quality Assurance and Test

Quality Assurance (QA) and test activities can help ensure the reuse potential of developed software.

This section provides guidance for QA and test activities to help ensure that the standards and guidelines presented in this manual are followed. This section should be used in conjunction with appropriate NATO STANAGs and AQAPs or other military standards quoted.

The following subsections address evaluation activities, metrics, test procedures, and problem resolution.

7.1 Evaluation Activities

A standard must be used to be successful; QA monitoring activities can help ensure that success.

Conformance to the project reuse standard should be among the specific aspects audited by QA. QA traceability review is particularly critical for ensuring that reusability properties are not lost in phase-to-phase transitions.

7.1.1 Develop a QA procedure for auditing conformance to the project reuse standard.

An explicit project requirement should be established based on this manual. Each paragraph of the standard should be addressed; guidelines should be selected as appropriate to the project situation and tailored as necessary (for example, to adapt to the selected programming language and documentation standards). The QA program should then include evaluation activities to ensure conformance to this project-specific requirement. These should be integrated with the QA activities that are already practiced.

G7.1.1.1 Recognize that QA has a continuing role; carry out appropriate evaluations at each phase of the life-cycle.

QA cannot wait until code is finished before evaluations begin. At each phase of the life-cycle, QA should carry out appropriate evaluations to ensure that the appropriate requirements for that phase are being followed. QA review should be included in each design review, with an established audit procedure for each product and activity. From the reuse perspective, it is particularly important that the guidance for the requirements and top-level design phases is followed, as these phases form the basis for all development of reusable software. If the guidance is ignored in these early phases, little benefit will be gained from following it in the detailed design and coding phases.

G7.1.1.2 Audit the process, not just the products.

QA evaluations and audits usually focus on products—specifications, code, etc. These are usually amenable to objective quality measures. It is important to also audit the process—the work activities that influence the quality of the product. For example, such evaluations might address:

- whether CASE tools are being used as expected, and whether they are serving the intended purpose
- whether changes to RSCs are strictly controlled by configuration management procedures
- whether testing is being carried out according to plan

G7.1.1.3 Use checklists and/or automated tools to support product audit.

Product evaluation should be objective, and should be assisted by a checklist to ensure that all key points are covered. Such a checklist also provides a mechanism for documenting audit results and preparing discrepancy reports. Automated tools can help with this activity (see subsection 7.2, below).

G7.1.1.4 Carry out requirements/design/code/test traceability procedures.

One activity that is normally part of a traditional QA program is tracing capabilities from requirements to design to code to tests, to ensure that no capabilities are lost and no changes are inadvertently introduced. This is especially important for reusable software components, as noted in earlier sections, as it is critical that RSCs conform to their specification and that linkages to earlier and later life-cycle products be explicit.

G7.1.1.5 Assess the effectiveness of the project reusability standard, and improve it as appropriate.

Throughout the QA activity, there will be opportunities to determine how effective the project reusability standard is. It is important to evaluate its effectiveness and make appropriate improvements. Note that if the standard was obtained from a customer or other organization, it may be necessary to have any such change approved. Even if change approval is not required, it would be helpful to provide the suggested improvements to that organization.

7.2 Metrics

Metrics can help measure reuse potential and expected payback.

Metrics can have three roles—measuring the reuse qualities of a component, providing an estimating and justification base in support of a reuse program, and providing information needed by a reuse library organization.

7.2.1 Use metrics to promote RSC quality and to further the practice of reuse.

Metrics essentially just means measurements. In a reuse organization, measurements can be made of the software itself—e.g., its extent of conformance to a reuse guideline—and of the

process—e.g., the savings in labor and cost that result from reuse. The real value of such metrics is not primarily to the initial development project but to the organization in general. Metrics help assure the quality of RSCs, help demonstrate the success of reuse, and help improve the ability to estimate future projects that take advantage of reuse.

G7.2.1.1 Collect RSC quality metrics.

As noted earlier, code quality is an important aspect of the effectiveness of an RSC. It can be very effective to establish a program of explicitly measuring the quality of RSCs by assessing them against an established set of guidelines, such as those provided in this manual. Note that it is important to define the specific set of measurements and the relevance of each of those measurements before beginning metrics collections. These metrics can then be used to help assess the reuse potential of each component. It might be appropriate to establish a project requirement that each component explicitly designated as an RSC achieve some minimum score on this metrics assessment.

G7.2.1.2 Consider use of a metrics tool.

Tools are available to automatically process source code to measure code quality. One well-known tool, available for several languages, is McCabe's complexity measurement tool (McCabe 76). Another, specific to the Ada language, is Dynamics Research Corporation's AdaMAT (DRC 87). These tools can significantly ease the task of measuring code quality, and hence permit a more ambitious metrics program.

G7.2.1.3 Collect reuse metrics.

Reuse metrics refer to measurements of the experience and impact of reuse. For example:

- What were the added costs associated with making software reusable?
- How often has a particular RSC been reused?
- What percentage of reused components were requirements, design, code, etc.
- How much RSC code modification occurred?
- How many lines of code in an application were obtained via reuse?
- How did reuse impact schedule?
- How did reuse affect development cost?

These metrics will help justify the cost of operating a reuse program, and will provide an estimating base for scheduling and costing future efforts.

G7.2.1.4 Collect any metrics required by the reuse library.

Often a reuse library organization will want to provide its "customers" with a measure of the quality of the RSCs it offers. The organization may implement this policy by requiring the submitting organization to provide the necessary metrics. If so, this should be integrated in the project metrics program.

7.3 Test Procedures

If reusability is to be considered a real program requirement, it must be tested as a requirement.

Testing is particularly important for reusable software components, to ensure that they have the quality and robustness expected of an RSC, as well as to ensure that explicit reusability requirements are met. The tests associated with RSCs are themselves RSCs, and should be delivered to the reuse library.

7.3.1 Test RSCs as independent products.

In its initial development, an RSC is part of an overall system. It will clearly be tested as part of the normal development process, through unit, integration, and system test. However, the RSC is also an independent product with a well-defined specification. It must be tested for conformance to that specification.

G7.3.1.1 Test each RSC as an entity.

Each RSC should have its own set of tests. These tests should be usable independently of the overall system; they cannot depend on other parts of the system outside the RSC to support the test. In a sense these are comparable to the unit tests performed on each software component in a normal development process. However, they should be much more formal than the usual unit test. Unit tests are often informal and cursory, because it is assumed that all requisite capability will be fully tested during system test. In a sense, the RSC is a system in its own right; it has a well-specified requirement that must be met. Tests should demonstrate conformance to that requirement.

G7.3.1.2 Test all RSC interfaces over their full range of values.

An RSC can be considered as a "black box" with a specified behaviour for a given range of inputs. It should be tested for all combinations of allowable inputs, across their full range of values. Such tests should be performed even when it is known that some of the conditions the RSC addresses will not occur in the overall system of which the RSC is initially a part.

G7.3.1.3 Test for conformance to any explicit reusability requirements.

Sometimes an RSC will have explicit reusability requirements such as "It shall be possible to port this component to the ABC hardware/operating system." or "The message handling package shall be extensible to support the XYZ message type." In such cases, this requirement should be tested like any other system requirement. An explicit portability requirement can be tested only by actually porting the software. An explicit additional function such as handling another message type can be tested only by implementing that capability. Such tests can be costly, but are imposed by such requirements. If a customer imposes requirements of this sort, it is important that the test implications be understood. In some cases, the customer might want to relax the requirement to something like "The component shall conform to all the portability guidelines in document QRS." This would limit the test requirement to a full audit for conformance to the referenced guidelines.

7.3.2 Manage tests as RSCs themselves.

RSC tests form a valuable addition to a reuse library. The reuser who can obtain tests in addition to the code RSC has both a way to evaluate the RSC for his situation and a significant help in meeting his own test requirements.

G7.3.2.1 Establish traceability back to requirements.

Tests should be directly traceable to requirements. It is not enough (for a nontrivial RSC) to simply say “This test goes with RSC X”. There should be a number of RSC tests each keyed to a particular RSC requirement. This is particularly important because it allows the reuser who might wish to adapt the RSC’s functionality or make partial use of its features to tailor the test set accordingly.

G7.3.2.2 Do not build in platform/system dependencies in the tests that are not present in the RSC itself.

RSC tests should be at least as portable as the RSC itself. They should not rely on system characteristics or debugging/analysis aids that will not be available to the reuser. These tools can support unit test of the RSC within the initial project, but they cannot substitute for the formal RSC tests.

G7.3.2.3 Provide information on any modifications to the tests that would be required if the RSC is modified.

Clearly, not all modifications to an RSC can be anticipated; however, many can be. As noted in an earlier section, the RSC designer should provide guidance to the reuser in how to make these anticipated changes. Changes in the RSC will require corresponding changes to the tests, and corresponding guidance should be provided. This might be as simple as providing the same kinds of parameterization to a test that are provided in the RSC—essentially a generic test that is instantiated just like a generic RSC—or it might involve replacing parts of the test procedure, data, or expected results. The key is simply to capture the test designer’s knowledge about likely modifications.

7.4 Problem Resolution

To ensure that they have an impact, QA and test activities must be accompanied by a problem-resolution procedure.

All problems uncovered in QA and test activities must be tracked and resolved. Such resolution may consist of a requirement for corrective action, or may permit the deviation with justification.

7.4.1 Document and resolve all deviations from established policy or discrepancies with requirements.

Problems may arise either from failure to comply with an established policy (e.g., the reuse standard) or from failure to meet a program requirement (e.g., testing shows that functionality differs from that established in the requirements specification). In either case, the problem must be identified and tracked through resolution as part of the QA process.

G7.4.1.1 Document and manage problems as part of each QA evaluation activity.

Section 7.1 identifies QA evaluation activities to be performed, recognizing that these are especially important in the development of reusable software components. In each of these evaluation activities, there should be a step involving the identification and logging of all deviations. Subsequently, the open problems should be tracked and reviewed at regular intervals. Part of each program milestone should be a review of all outstanding problems and their resolution status. Each problem should have an established resolution date, and a means for verifying that the problem has been resolved. If any problems remain outstanding when an RSC is delivered to the customer or to the reuse library, these should be clearly documented with the delivery.

G7.4.1.2 Establish a mechanism for handling problems reported by the reuse library.

After an RSC is submitted to the reuse library, the library's procedures may include reporting back to the submitting organization on any problems encountered with the RSC. Whether or not the submitting organization has maintenance responsibility for the RSC (something determined by the library's policy), the problem report requires handling. If the RSC is still in use, either in ongoing or completed systems, it is necessary to determine whether the problem will occur in those systems and, if so, to treat it like any other problem encountered during the life cycle.

G7.4.1.3 Identify criteria for allowing deviations.

Sometimes a deviation, either from policy or from requirements, may be justified. There should be a specific QA policy for judging such requested deviations. In the specific case of conformance with the reuse standard, some of the possible justifications for deviation might be:

- The reuse goal implied by the standard is met in another manner not explicitly addressed in the standard document.

- The known scope of reuse is narrower than the general case addressed by the standard (e.g., there is no anticipated need to port the component to a different processor) and hence conformance can be relaxed.

- Compliance with some aspect of the standard is very costly in terms of human resources (e.g., full testing for reusability) or in terms of component performance (e.g., adding some kinds of component generality). Management decides to relax the reuse goal rather than make the extra investment.

Any such proposed deviation should be subject to an established review process. Any approved deviation should be documented, with rationale, when closing the problem.

Section 8

Documentation

Documentation of reusable software provides a key part of its reuse value.

Documentation of reusable software serves a dual role; it fills the traditional role of documentation, and also provides explicit guidance to the reuser.

The following subsections address the application of conventional documentation standards, documentation for the reuse library, the reuser's manual, and the role of formal specification methods.

8.1 Application of Conventional Documentation Standards

The normal documentation accompanying a reusable software component is, in effect, part of the RSC, and must conform to comparable standards.

Reusable components are normally developed in the course of carrying out a contract, and thus contractually-required documentation is normally prepared. If there is an objective of reusing the software, particular attention should be given to various aspects of this documentation to facilitate this reuse.

8.1.1 When developing the normally required documents, address the specific needs of the reuser.

The system documentation for an RSC serves a dual purpose; in addition to serving as a deliverable on the development contract, it supports the reuser. It must provide the reuser with guidance in understanding and using the RSC, and also must itself serve as reusable documentation components. It must be organized to allow the reuser to quickly access the information he needs, to understand that information, and to extract it for reuse.

G8.1.1.1 Comply with any accepted documentation standards of the potential user community.

If it is the case that most organizations in the potential user community follow a particular documentation standard (e.g., for the U.S. DOD, MIL-STD-490A) the reuse potential of the documentation is significantly enhanced by conformance to that standard. This may be worth adopting as a project policy for RSCs even if not otherwise contractually required.

G8.1.1.2 Use consistent organization and formats.

Clearly, a consistent approach to organizing and formatting system specifications will support the reuser. A consistent structure makes the documents more readable and understandable, helps the reuser find the relevant parts of the document, and increases the likelihood that the documentation will be reusable in his program. Some kind of

documentation standard (if possible, using and perhaps augmenting a military standard) is necessary to ensure this consistency.

G8.1.1.3 Organize documentation so that it is separable into the same units as the reusable code components—documentation for each RSC should be complete and self-contained.

When a reuser elects to reuse a particular RSC, he should be able to access the documentation specifically applying to that RSC, without having to "untangle" it from surrounding documentation. This means that the RSC's requirements, design, test, and user (if applicable) documentation should stand alone, not be full of dependencies and references to other parts of the document. (Such documentation interdependency is likely to lead to code interdependency, which interferes with reusability.)

G8.1.1.4 Ensure that documentation remains consistent with code.

The documentation for an RSC is far more heavily used than most design documentation. Often, once a design specification is written and passed through the design review, it is never referred to again. Code often deviates from the specification without any update to the specification. This situation is not acceptable for reusable component documentation. Because the design specification (and other documentation) will be propagated to each system using the RSC, and will actually be read by reuser's desiring to understand the RSC, it is essential that it be consistent and accurate.

G8.1.1.5 Supply all documentation in machine-readable form.

RSC documentation can be far more easily reused if it can be mechanically extracted from its initial document and inserted in another one. Ideally, this documentation should be representable in a form not dependent on a particular choice of word processor.

G8.1.1.6 Recognize that documentation should be understandable by others.

Completeness, clarity, and understandability are goals in all system documentation, but are particularly important in fostering reuse. As noted above, reusers may be among the few who actually read design specifications with a need for real understanding. It may be desirable to develop a set of project guidelines specifically addressing documentation style.

8.2 Documentation for the Reuse Library

Special documentation is required for components submitted to a reuse library organization.

If documentation is to be submitted to a reuse library, the library will usually require special-purpose documentation to help the library support classification, identification, and retrieval of the component. This documentation is usually separate from the normally required documentation developed by a project.

8.2.1 Provide a “quick look” view of the component’s functionality.

It must be possible for the library staff to obtain a quick understanding of the general function and characteristics of an RSC, so that the component can be properly classified in the library. Likewise, the potential reuser should be able to obtain a quick understanding to make an initial judgement of whether to further consider the RSC for reuse.

G8.2.1.1 Provide an abstract describing the function of the component for use in classification and searching for the component.

A component abstract is a brief statement of the function performed by a component. It should be limited to one or two paragraphs, so that it can be quickly browsed and can be displayed on a single screen during a lookup process. The library organization may specify the information to be included in the abstract. At a minimum, it should indicate the capabilities provided by the RSC, including an explicit list of operations provided.

G8.2.1.2 Identify any dependencies of the component.

One of the first things a potential reuser will want to examine, after finding the basic functionality acceptable, is a list of dependencies of the RSC. For example, does it depend on the availability of a particular DBMS, only work when data is represented in a particular form, or only work with Ada compilers with a particular tasking model? Any such dependencies should be identified in an easily found way; again, library policy may determine a format.

G8.2.1.3 Provide classification information as requested.

The library organization will ordinarily classify its RSCs in some manner to promote easy lookup. This can most easily be done if the submitting organization provides the necessary classifying information. This may, for example, consist of selecting from a list of terms for such aspects of the RSC as *function*, *object*, *source language*, etc. The specific requirement will be established by the reuse library.

8.2.2 Provide RSC assessment information.

The reuse library will wish to provide “customers” with information about the quality and usability of the RSCs. The developer is one of the sources of such information; any that is available should be supplied with the RSC.

G8.2.2.1 Include any available reusability and quality metrics.

As recommended in the preceding section, the project may have gathered metrics about the quality and reusability of intended RSCs. The library may require particular information, in which case that particular information should be supplied. In the absence of such a requirement, it is important to provide whatever is available.

G8.2.2.2 Identify any outstanding problem reports.

If any problem reports are outstanding against an RSC, this should be clearly explained when submitting the RSC. Normally, an RSC would not be submitted when it contains known errors. However, such reports might indicate desired improvements or areas in which additional testing or documentation would be desirable.

G8.2.2.3 Identify any recommended enhancements.

Often the RSC developer is aware of enhancements that would improve the component—improve performance, make it more robust, improve maintainability, or extend the scope of reuse. Such enhancements, and any information the developer may have about how to make the change, should be identified with the RSC submission. Enhancement requests from the RSCs initial users should be treated similarly. In many cases the library will have a reuser who wishes to make the recommended improvement; the result will then be available to the originating project, if desired.

8.2.3 Provide any special information needed by a potential reuser.

Sometimes special cases require the reuser to know something special about a component before attempting to reuse it. This might apply if there are some constraints on how the component is obtained or on the environments in which it can be used.

G8.2.3.1 Identify any commercial or legal restrictions on use of the component.

Sometimes the use of components is restricted. This might be because the component is a commercial product requiring purchase or special license. It might be because the component is classified in some way and its use requires special permission. Use of the component might require execution of some sort of nondisclosure agreement. In any such case, it is important to make this information immediately clear in the component's abstract and documentation. It is also desirable to identify the restriction in the source-code prologue, if possible.

G8.2.3.2 Explain how to access the component, if it is not physically available in the library.

Sometimes a component itself may exist but not be physically available from the library. This might occur in any of the cases noted above, or simply because some physical restriction (e.g., host machine dependence or limited storage resources) precludes its inclusion. In these cases, it is necessary to provide specific information about how to access the component.

8.3 The Reuser's Manual

Normal documentation does not fully meet the needs of the RSC reuser; additional support should be provided in a reuser's manual.

Section 8.1 discusses ways the normal project documentation can be made more useful to the reuser. However, it will still fall short of the support a reuser really requires. Additional reuser documentation should be provided, specifically directed to the concerns of the individual trying to evaluate, modify, and incorporate the RSC.

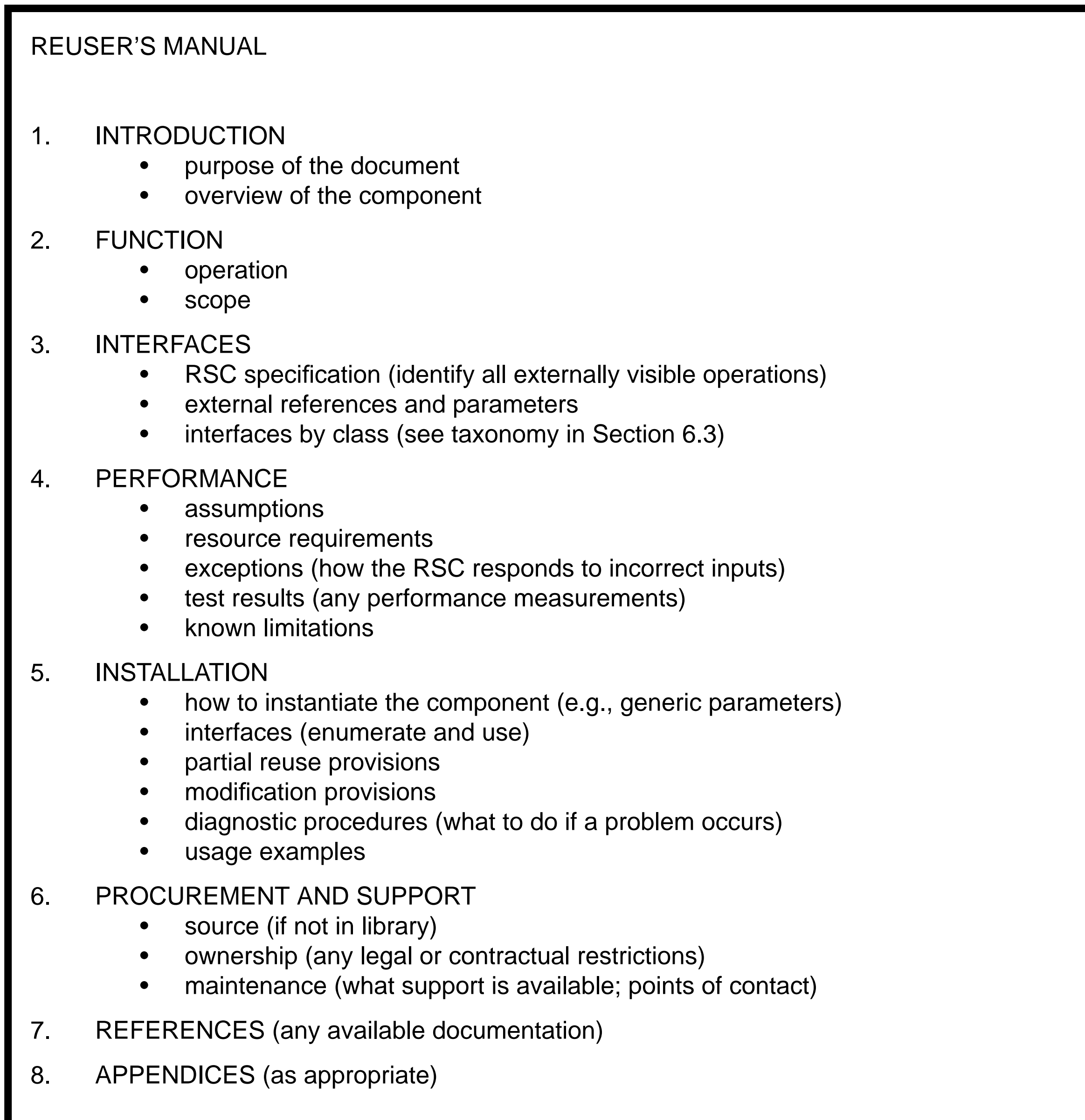
8.3.1 In addition to the usual documentation, provide documentation that specifically explains how to reuse the component.

Some additional documentation is essential to support the reuser. At a minimum, this might be simply the abstract identified in the preceding subsection. However, it is far better to develop a reuser's manual specifically intended to support reuse of the component. The reuser's manual

should include, in addition to an expanded description of the RSCs function, information on how to install, modify, and tune the component, and guidance on how to obtain support. If resources do not permit development of a reuser's manual, this information must be obtainable from the normal documentation.

G8.3.1.1 A reuser's manual should follow a standard format.

Figure 8.1 is a recommended outline for a reuser's manual. An individual organization may tailor this format as appropriate, consistent with library policy.



REUSER'S MANUAL

1. INTRODUCTION
   - purpose of the document
   - overview of the component
2. FUNCTION
   - operation
   - scope
3. INTERFACES
   - RSC specification (identify all externally visible operations)
   - external references and parameters
   - interfaces by class (see taxonomy in Section 6.3)
4. PERFORMANCE
   - assumptions
   - resource requirements
   - exceptions (how the RSC responds to incorrect inputs)
   - test results (any performance measurements)
   - known limitations
5. INSTALLATION
   - how to instantiate the component (e.g., generic parameters)
   - interfaces (enumerate and use)
   - partial reuse provisions
   - modification provisions
   - diagnostic procedures (what to do if a problem occurs)
   - usage examples
6. PROCUREMENT AND SUPPORT
   - source (if not in library)
   - ownership (any legal or contractual restrictions)
   - maintenance (what support is available; points of contact)
7. REFERENCES (any available documentation)
8. APPENDICES (as appropriate)

Figure 8.3 - Example Outline for a Reuser's Manual

8.4 Formal Specification

Formal specifications can support the reuse potential of RSC's developed for particularly critical applications.

Formal specification techniques, in general, have not proven an effective cost/benefit choice in software development. In certain areas— e.g., trusted software development—however, they can be valuable. Formal specification might be required for such RSCs.

8.4.1 Consider requiring formal RSC specifications in special cases.

Formal specifications can enhance the reuse potential of a software component by helping to guarantee its correctness and safety. However, the guarantee is not absolute and the methodology is costly, both in labor and in supporting tools. Their use should normally be considered only in special cases. (Such specifications may also be valuable to support the automated generation of systems from reusable parts, but this capability is still a research area.)

G8.4.1.1 Evaluate RSCs for potential formal specification on a case-by-case basis.

If formal specification is to be applied at all, evaluate each RSC independently to assess its desirability. Some of the factors to be considered are:

- Is the RSC likely to be reused in systems with a formal specification requirement, e.g., trusted systems or systems with a high reliability or safety requirement?
- Does the RSC implement a critical function that is relatively straightforward to specify/validate and relatively harmful if incorrect?
- How much cost is involved in the formal specification?
- Are trained personnel and tools available?
- Is the anticipated level of reuse so great that the investment initially is justified in terms of saving maintenance effort later?

G8.4.1.2 Couple formal specification with verification.

Formal specifications are most valuable when coupled with a verification method that assesses compliance with the specification. If a formal specification for an RSC is developed, such a verification should also be performed. Otherwise, not only is the correctness benefit lost; the reuser will have no certainty that the specification matches the implementation and therefore it will be useless to him.

G8.4.1.3 Plan for any formal specification from the beginning.

Formal specification and verification requires steps throughout the software life cycle, beginning with the requirements phase. It is necessary to identify up front those RSCs for which formal procedures are followed, so that the steps can be performed in sequence.

G8.4.1.4 Consider the validity of a formal specification in a new operating environment.

There are aspects of formal specification and verification that are dependent on characteristics of the host system, especially the operating system and any supporting software such as X-Windows or a DBMS. When the component is reused in a system with different characteristics, much of the specification and verification may be invalidated. It is important to consider this impact, both in assessing whether to carry out the activity and in structuring and documenting it so as to clarify the requirements to adapt it to a different environment.

G8.4.1.5 Capture any formal specification results for reusers.

If a formal specification and verification are carried out, clearly reusers will want the results for evaluation and incorporation in their systems. All results should be documented, organized consistently, and submitted to the library just as other RSC documentation is handled.

APPENDIX

Appendix A

Ada Coding Standard

What is now recognized as good modern software engineering practice, especially in the Ada community, includes many principles which inherently support software reuse; the Ada language itself was designed with software reuse as a principal objective. This section differs slightly from typical Ada coding standards because of its emphasis on software reuse. Some guidance has been added or given greater weight.

Each regularly numbered paragraph forms part of the standard, and is considered mandatory for achievement of intended reuse goals; any deviation must be justified and approved. The standard is augmented by a number of guidelines (indicated by paragraph numbers beginning with the letter "G"), which provide non-mandatory guidance on ways to implement the standard.

Standard paragraphs and guidelines specifically intended to support reuse (as opposed to supporting overall quality and reliability) are marked with an asterisk (*).

The guidance in this section is arranged in the following categories:

1. Names
2. Format and Layout
3. Commentary
4. Types and Subtypes
5. Named Numbers, Constants, and Literals
6. Expressions
7. Control Structures
8. Exceptions
9. Program Structure and Compilation Units
10. Parameters
11. Tasks
12. Other Areas

A.1 Identifiers

A.1.1 Choose identifiers to enhance readability and understandability.

Well-chosen identifier names make the difference between a highly readable program and a meaningless one. Identifiers should be chosen to convey meaning, avoid unclarity, and assist readability.

GA.1.1.1 Use mnemonic identifiers.

A name should indicate the meaning and purpose of the entity it represents. For example, the identifier *Message_ID* is more meaningful than the identifier *M* (assuming it refers to a message identifier and not to a square root!).

*GA.1.1.2 * Avoid jargon in identifier names.*

As far as possible, identifiers in reusable software should be general, related to the abstract concept of the function performed, rather than jargon peculiar to one application or another. An exception exists where a particular application is universally understood (e.g., mathematics, screen and menu management); in that case the terminology of that application is, effectively, a general term, applying to other applications by analogy.

GA.1.1.3 Choose identifiers in a way that reflects their relationship, if any, with each other.

Choosing identifiers that express the relationship among different entities makes it easier for the reader to remember the relationship and reduces the number of names to memorize.

GA.1.1.4 Select identifier names long enough to be meaningful, but not so long as to be unmanageable.

Identifiers should be long enough to be meaningful throughout their entire scope, but should not be so long that they are difficult to read at one glance. In general, identifiers should be between 5 and 20 characters. An excessively long identifier is hard to perceive as a single item. This distraction makes it harder to grasp the program structure, resulting in a less readable program.

GA.1.1.5 Use abbreviations for long words in an identifier.

Abbreviations help to shorten words but keep meaning intact, making the program more readable. Abbreviations should be used consistently throughout the system so that no confusion results from them. If abbreviations are used, a reference list, including their spelled out form, should be provided in the software documentation. Including such a list that might otherwise occur because of different interpretations of an abbreviation. A good place for this list is directly in the code, for example, in the “header” comments.

GA.1.1.6 Use underscores to separate words in an identifier.

The use of underscores to separate parts of an identifier that are intended to be seen as separate words (e.g., *Radar_Scan_Rate*) allows the reader both to perceive the separate words making up the identifier and to see the identifier as a single entity.

A.1.2 Use distinct identifiers in lexically nested scopes.

No identifier should be named the same as one in an outer enclosing scope. Although this is permitted by the Ada visibility rules, it is poor practice. A reader must know where an identifier comes from in order to understand the program logic. If identifiers are duplicated in nested scopes, it may not be immediately apparent which one is being used.

A.1.3 Select identifiers that clearly distinguish among Ada program entities.

Identifiers should be selected in such a way that their form or part of speech clearly identifies their role in the program. Such distinctions can materially aid in the understanding of the program by making clear the role of an identifier without requiring diversion of the reader's attention to search for the declaration of the identifier. Ideally, code should read like a book. The relationships of the entitles in the code should parallel those of the English language as much as possible.

GA.1.3.1 Use nouns for most object and function identifiers.

Object and function identifiers each represent individual data objects. Since data objects are the basic "things" on which computing is done, it is natural that nouns, which name things, be used to represent them.

GA.1.3.2 Use declarative statements for Boolean identifiers.

An identifier representing a Boolean object or function may, however, be a statement that can be true or false or a question answerable by "true" or "false." A Boolean object or function, in effect, asks a question about some condition that may occur. In English, we may ask a question by making a statement with a rising inflection at the end. Naming the Boolean as a statement (e.g., *Stack_Is_Empty*) allows that same linguistic technique to be used as the program is read.

GA.1.3.3 Use verb phrases for procedure and task entry identifiers.

Procedures and task entries are imperative statements that cause actions to happen. Naming them as verb phrases indicating the action accomplished makes their imperative nature clear.

GA.1.3.4 Consider a naming convention that distinguishes type names from identifier names.

In some cases, it may not be immediately obvious to a reader whether a name is that of a type or an identifier. (The only real confusion for a reader who is fluent in Ada is between type conversions and function calls.) Some organizations find it helpful to require a distinctive way of naming types, e.g., appending "*_Type*" to the name, as in "*Matrix_Type*". However, others find this convention unwieldy.

A.1.4 Use name overloading only to improve understandability, not to obscure it.

Overloading allow the same name to have multiple meanings. It may be used to show similarity and hide differences that are irrelevant to the user. However, it can also create substantial confusion, which must be avoided.

GA.1.4.1 Do not overload type names defined in package STANDARD.

Type names defined in package *STANDARD* are recognized by most Ada programmers almost as though they are language keywords. Overloading these (e.g., giving the name "Boolean" to a type representing employee records) would be very misleading.

GA.1.4.2 Limit the use of overloading to operators and widely used procedures or functions that perform similar operations on different types.

```
Cross_Product_Vector := (others => O);
function "*" (Matrix_A, Matrix_B: Matrix)
   return Vector;
...
Cross_Product_Vector := Position_Matrix * Transform_Matrix;
```

or

```
function TAN (Angle : Radians) return Small_Number;
function TAN (Angle : Degrees) return Small_Number;
```

GA.1.4.3 Never use overloading to give the same name to dissimilar operations.

The converse of the preceding guideline is that overloading should not be used to give the misleading impression that two operations are the same when in fact they are quite dissimilar. For example, using a function named "*Move*" to identify a function that moves a character string as well as to a function that moves a radar would be unreasonable. Both should have unique, more explicit names.

A.1.5 Minimize the use of *use* clauses.

Use clauses make identifiers visible without their declarations being present. This makes it difficult for readers and maintainers to find the declaration of an identifier. A widely accepted exception is **"use** Text I0". Instead, either qualify identifiers explicitly or use rename and subtype declarations to relate the identifiers to their qualified referents. In general, exceptions to this rule should not be permitted without independent approval.

A.2 Format and Layout

A.2.1 Format so as to make the program's logical structure apparent.

The goal of programming conventions is to achieve reliability and maintainability by making programs readable and understandable. An important element of readability is that the program structure and the lower-level structures within it be easily seen. The visual layout of the code can emphasize the program structure.

GA.2.1.1 There should be no more than one statement or declaration per line.

A second statement or declaration on a line can be easily overlooked by a reader who is scanning the program to find something particular.

GA.2.1.2 If it is necessary to break a statement or declaration across program lines, make the break(s) consistent and at the highest level of abstraction.

The way in which a long statement is broken may influence its interpretation. It should be broken at the highest level of abstraction—at the outermost level of nesting of the statement's operation. For example, if a statement adds three long parenthesized expressions, the break should occur between the expressions rather than in the middle of any of them. The statement, if broken at all, should break at all such points at the same level. For example, a statement adding three long expressions should use three lines to break between all of them, not just between the second and third. If lines are broken without regard to these logical abstractions, it is possible that they may be misinterpreted.

A.2.2 Use spacing and indenting to indicate program structure.

Spaces and blank lines, when used effectively, can greatly enhance program readability. Tools that support this need, such as source code reformatters, language-sensitive editors, and pretty printers, are readily available.

GA.2.2.1 Use blank lines to emphasize the grouping of related statements and declarations.

Blank lines allow adjacent blocks at the same nesting level to be distinguished visually.

GA.2.2.2 Use blank spaces to emphasize the structure within a statement or declaration.

Blanks should be used consistently before and after the highest level operators in a statement. In complicated statements or expressions, blanks may be used before and after operators at lower levels. This use of horizontal space to emphasize the structure within a statement or declaration aids the reader in grasping the composition of the statement or declaration.

GA.2.2.3 Do not use TAB characters for spacing or alignment.

Tab-stop settings are environment-dependent. It they change, the formatting or alignment will be disrupted.

A.2.3 Use consistent indentation to indicate program structure.

Ada programs are structured as a nested series of blocks. Indentation graphically shows the block structure of the program.

GA.2.3.1 Use indenting to indicate nesting level.

Declarations and statements on the same level (i.e., those belonging to the same declarative part, sequence of statements, or compound statement) should be equally indented. This is in accordance with the basic principle of making the underlying program structure visible.

GA.2.3.2 Matching key words should be vertically aligned.

Matching key words (***if-elsif-else-end if***, ***while-for-end loop***, ***declare-begin-exception-end***, ***case-end case***, ***record-end record***, ***generic-package-private-end***, ***package body-begin-exception-end***, ***generic-procedure***, ***procedure-begin-exception-end***, ***generic-function***, ***function-begin-exception-end***, ***task-end***, ***task body-begin-exception-end***, ***accept-do-end***, ***select-or-else-end select***) should be aligned vertically (i.e., beneath one another on separate lines). The key word ***then*** should be aligned vertically with ***if*** when it does not fit on the same line. The key word ***loop*** should be aligned vertically with ***while*** or ***for*** when it does not fit on the same line. This is a key technique for improving code readability. It allows easy visual matching of the parts of compound statements, making the nesting structure more clearly visible.

GA.2.3.3 Indent logically contained constructs within containing structures.

The statements and declarations contained within other statements or declarations should be indented consistently within the containing structure. This also helps to make the nesting structure of the program clearer.

GA.2.3.4 Indent succeeding lines of a continued statement.

This helps the reader to see that several lines constitute a single statement.

GA.2.3.5 Place block and loop identifiers to the left of the main text.

This improves visibility, allowing the reader to find quickly the beginning and end of a loop. Loops are more difficult to identify if their identifiers are not set off conspicuously.

A.2.4 Use letter case to facilitate readability.

A consistent use of letter case (upper and lower case) is essential to readability. It should be used as a mechanism to make program intent clearer—for example, by providing a way to distinguish identifiers from Ada reserved words. Identifier-naming conventions should also make consistent use of case.

GA.2.4.1 Write Ada reserved words entirely in lower case letters.

Text written with a high proportion of lower case letters is easier to read than all upper case text. Furthermore this convention, used in the Ada language reference manual and in most Ada texts, is widely accepted.

GA.2.4.2 Use capitalization to distinguish identifier names from reserved words.

Identifier names can be written with initial capitals, or all in capitals. The initial capital form is often considered easier to read; the all capitals form makes identifiers more distinctive. Whichever approach is chosen, it should be consistent.

If using initial capitals, capitalize the first word and each significant subsequent word. A good rule of thumb is to use the same case convention as in a title, for example, *Naming_Conventions_for_RSCs*.

GA.2.4.3 Use all uppercase letters in type and enumeration value identifiers.

Ada syntax does not provide an obvious way to distinguish between type conversions and function calls. Type names written in all uppercase can help avoid confusion in understanding program structure.

GA.2.4.4 Consider a notational convention that distinguishes type names from identifiers.

In some cases, it may not be immediately obvious to a reader whether a name is that of a type or an identifier. (The only real confusion for a reader who is fluent in Ada is between type conversions and function calls.) One way of distinguishing type names is given in *GA.1.3.4*, page A-3. An alternative is to use capitalization differently—for example, using initial capitals in identifiers and all capitals in type names.

A.3 Commentary

A.3.1 Use embedded commentary to enhance program readability and understandability.

In general, code should be written to be self-documenting. Commentary should augment clear structure and naming, not substitute for them. It should be reserved for those instances when additional explanation enhances the readability and understandability of the code.

GA.3.1.1 Be sure embedded commentary is grammatically correct.

This is particularly important when reusable components are propagated to many different systems. Ideally, embedded commentary should be subject to the same scrutiny that design documentation is.

GA.3.1.2 Use embedded commentary to describe the code in a higher level or generalized manner.

Embedded commentary can detract from program readability if it is too detailed. A knowledgable software engineer can extract the details from the code once the concept or abstraction has been outlined in the commentary. For example:

```
-- Search for the first occurrence of the requested item
Next_Position := O;
for Index in Message _Text'range loop
    Next_Position := Index;
    exit when Message_Text (Next_Position) = Requested_Item;
end loop;
```

*GA.3.1.3 *Use generally understood terminology in embedded commentary.*

Avoid application-specific jargon. When code implements a general function, describe it in general terms rather than application-specific terms. For example, a package implementing variable-length character strings should be described as such, rather than as a package for manipulating employee job descriptions (which may be its specific use in the initial application). Terminology from generally understood application areas such as mathematics, data bases, and user interfaces is acceptable.

GA.3.1.4 Use embedded commentary to explain any hidden but significant implications.

A comment should give the reader enough information to provide a general understanding of the code that follows. It may be important to know why something was coded in a particular way. Sometimes changing a particular code segment can have non-apparent effects. This information is vital, especially when many programmers have overlapping responsibilities.

GA.3.1.5 Position embedded commentary to enhance understandability.

To avoid disrupting the flow of the code, embedded commentary should be placed to the side of or between blocks of related statements. Comments on multiple lines should be aligned vertically.

*GA.3.1.6 Comment each **begin-end** block.*

Every ***begin*** corresponding to a subprogram, package, or task should have, on the same line, an attached comment indicating the name of the subprogram, package, or task. The matching ***end*** should also include the name of the subprogram, package, or task. This clarifies the structure of the program by allowing visual matching of ***begin***s with ***end***s. This practice helps focus attention on the boundaries of the subprogram or package. The reader can easily tell which code belongs to which program unit.

GA.3.1.7 Use comments to make declarations more understandable.

Comments should be attached to declarations when it is desirable to provide additional information about the purpose, properties, and usage of the item being declared that is not apparent from the name of the item. Attaching comments to declarations is a way of identifying the concept behind the items being declared. Sometimes there are characteristics of these items that are not obvious. These should be included in the comment. Note that not all declarations need comments; well-chosen names are better.

GA.3.1.8 Use comments to summarize the effect of statement groups.

Comments should be used to explain the net effect of a group of statements occurring at a lower level. In particular, there should be comments attached to the ***then*** and ***else*** clauses of an ***if*** statement explaining the net effect of each clause. There should also be a comment attached to each ***loop*** statement indicating the effect of one pass through the loop body. Comments offer immediate information about actions being performed within a control structure. When the parts of a control structure consist of several statements, a reader can easily get lost in the logic. A comment explaining the function of those statements will focus attention on the important issues. Comments on the ***then*** and ***else*** clauses reinforce the abstraction involved in the control structure by providing a place for an explanation of the effect of the entire group of statements making up the clause. This allows a first reading of the program to be made by reading only the comments below an ***if*** statement.

GA.3.1.9 Use embedded commentary to provide a rationale for the selection of critical algorithms.

When more than one approach to solving a particular problem is possible, the rationale for a particular approach should be briefly documented in the source code. Including this rationale can avoid a maintainer replacing the algorithm without understanding the consequences.

```
-- Sort the temporary-table.  An insertion sort was chosen due to the large size
-- of the table and high frequency of sorting.

for Index in Table'range loop
    ...
    Insertion Sort Algorithm
    ...
end loop;
```

GA.3.1.10 Use embedded commentary to identify and justify deviations from the coding standard presented in this document.

If compliance with this standard is normally required, it may be desirable to document deviations from it. This would suggest that there is a reason for the deviation, so that someone will not inadvertently "fix" it and undo the intent.

A.3.2 Provide a consistently formatted prologue for each program unit.

A prologue containing a consistently formatted set of comments should be required at the beginning of each program unit (package, subprogram, or task) explaining the purpose and function of the unit and its relationship to the rest of the software. Any special usage information should be included. A standard-form prologue provides information about the program unit in a form that is familiar and readily apparent to the reader. It provides a checklist to ensure that all relevant information is available. This immediate form of documentation is also more likely to be kept up-to-date than off-line documentation. Each project should establish a detailed standard for the content and format of prologues.

*GA.3.2.1 * Use the prologue to provide information specifically required for RSCs.*

The prologue for an RSC should prominently identify it as an RSC, and should indicate a point of contact for problem reports, enhancement requests, and maintenance. This ensures that application maintainers don't inadvertently duplicate library maintenance efforts or create new, independent versions that call for individual maintenance and configuration management. It also ensures that all users of an RSC benefit from problem reports and maintenance.

A.3.3 * Include any usage restrictions in the prologue.

The prologue for a reusable software component must warn the user of copyrights, licenses, or any other restrictions on the use of the component. Including this information in the prologue helps ensure that it will not be overlooked.

A.4 Types and Subtypes

A.4.1 Use types and subtypes to limit the program's properties to the desired purpose.

Types and subtypes restrict program behaviour in accordance with specific requirements. They are an important mechanism both for making intent explicit and for increasing implementation safety (by avoiding operation on inappropriate values).

GA.4.1.1 Use different types to represent different concepts or value sets.

The reader usually associates a particular type with a set of concepts related to a particular abstraction. If the same type is used to represent logically different concepts, this interferes with this process of understanding. For example, a program that includes:

```
type My_Range is Integer range 1 .. 31;
...
Day_of_the_Month:    My_Range;
Childs_Age:          My_Range;
```

interferes with understanding, compared with a program that includes:

```
type Day is Integer range 1 .. 31;
type Age is Integer range 1 .. 31;
...
Day_of_the_Month:    Day;
Childs_Age:          Age;
```

Note that the first example, apart from being counter-intuitive, makes it impossible to change the allowable range for *Childs_Age* without also changing the allowable range for *Day_of_the_Month*.

GA.4.1.2 Use subtypes to retain concept but restrict value set.

This simply tightens the control over the objects of the type. It forces an even more exact meaning and purpose to the objects. Therefore, subtyping can offer the reader a clearer picture of the relationships between objects and a better understanding of the program. For example:

```
type Temperature is Integer range -100 .. 100;
subtype Comfort_Zone is Temperature range 15 .. 25;
```

This makes it clear that *Comfort_Zone* represents a *Temperature*, but restricts values to a desired range. (Note that using two type declarations would not preserve the relationship between the two, and would not allow values of one type to be assigned to variables of the other.)

GA.4.1.3 Constrain all real types.

Definitions of floating point or fixed point types should include the **range** constraint. This models the abstraction more accurately, makes good use of range checking, and helps detect machine dependencies at compilation time. For example:

```
type Vacuum is digits 3 range 0.0 .. 14.7;
```

A.4.2 Use enumeration types to enhance understandability.

Types should be used to model the "real world" concept as clearly as possible. Programmers have traditionally had to use numbers to represent many concepts that are not really numeric; Ada enumeration types offer a far more readable alternative.

GA.4.2.1 Use enumeration types as opposed to integer ranges when that more clearly reflects meaning.

Enumeration types give the values of a type meaningful names. For example, rather than writing:

```
type Month is Integer range 1 .. 12;
This_Month: Month;
...
if This_Month = 3 then ...
```

use

```
type Month is (January, February, March, April, May, June,
               July, August, September, October, November, December);
This_Month: Month;
if This_Month = March then ...
```

Enumeration types are particularly effective in indexing arrays.

GA.4.2.2 Use two-valued enumeration types in preference to Booleans when that more clearly reflects meaning.

Examples of such two-valued pairs are *on/off*, *open/closed*, *up/down*, *yes/no*, *male/female*, etc. Values with these names are clearly more meaningful than *true/false*. They make the relationship of the pair explicit, and also eliminate the need for negation or testing for false values, thus reducing the potential for confusion and error.

A.4.3 Avoid anonymous types and subtypes where possible.

Declaring objects of anonymous types obscures the fact that each such object is of its own type, incompatible with others, regardless of the programmer's intent. An example of a permissible deviation from this guideline is declaring a reference table as a constant array, deliberately preventing any other object from having the same type, for example:

```
Days_in_Month : constant array (Month) of Day_Count :=
    (January => 31,
    February => 28,
    March => 31,
    April => 30,
    May => 31,
    June => 30,
    July => 31,
    August => 31,
    September => 30,
```

```
October => 31,
November => 30,
December => 31);
```

GA.4.3.1 If anonymous array types must be used, each such declaration should contain only one object.

Creating two objects of the same anonymous type, as in:

```
Stack1, Stack2: array (1..Max) of Element;
```

creates a strong, but false, impression that the objects are of the same type.

A.4.4 Use private types to hide implementation detail.

Private types are a good way of hiding unnecessary information. preventing the user from inadvertently including inappropriate dependencies. They offer security for the software, reducing the potential for misuse by allowing the user access only to abstract operations that are appropriate. If the representation of the values should change, only a change to the body will be required. It will not affect the use of the objects and, hence, not require any recompilations. These are particularly important properties for reusable software.

GA.4.4.1 Use limited private types to control unsafe usage and undesired modification.

Limited private types severely restrict the use of objects of that type, essentially restricting use to operations explicitly declared in the package defining the object. This provides total usage control, inhibiting the user's ability to corrupt the object or use it in an unanticipated way. In a reusable component, it also eliminates most undesired dependencies that other system components might create.

GA.4.4.2 To guarantee automatic initialization, implement private types as records with default values.

The Ada language does not guarantee initialization of objects when they are created, and erroneous situations could arise since a user cannot initialize objects of private types. This does not restrict the design. Any type can be implemented as a record with a single component of the desired type, allowing an initial value to be specified. The additional layer of data structure is transparent to the user and, with an optimizing compiler, will not affect efficiency. This additional safety provision is particularly important in RSCs.

*GA.4.4.3 * To avoid user's dependency on an object's implementation, implement private types using incomplete type declarations and access types.*

For example:

```
package Stacks is
type Stack is private;
...
private
type Specific_Stack;
```

```
type Stack is access Specific_Stack;
end Stacks;
```

The actual implementation of *Stack*s is deferred to the package body, where *Specific_Stack* is defined. Thus, changing the implementation will not require recompilation of the package specification and all the user's programs that depend upon the package.

A.4.5 *Use type declarations to help support component portability.

Type declarations can help eliminate implementation dependencies. These dependencies arise from such characteristics as the minimum and maximum representable values, floating point representation, implemented fixed point precision, and system clock period. If the range or precision of values for a type is not defined explicitly in the type declaration, the actual range or precision will be implementation dependent, causing the software to perform differently (or not at all) when ported to a different environment. Explicit specification of these properties can control this problem.

*GA.4.5.1 * Fixed-point type declarations should include a length clause for the attribute **'small**.*

A length clause specifies the precision to be used. Without it, the accuracy would be implementation-dependent. For example,

```
type Degree is delta 0.1 range -360.0 .. 360.0;
for Degree'small use 360.0 / (2**31);
```

provides an implementation-independent provision for *Degree*. (However, note that some implementations may not be able to support a particular precision. The above example assumes a 32-bit fixed-point representation. If the implementation cannot support the precision, the program will not compile, and hence not be portable as is. An alternative is to state the length in terms of a system-defined constant (*SYSTEM.STORAGE_UNIT* - 1) in the declaration. This will let the code compile on any machine, but its operation will vary depending on the implemented precision.

GA.4.5.2 For real types, specify the minimum accuracy/precision necessary to meet problem requirements.

The more stringent the accuracy (for float types) and/or precision (for fixed types) requirements, the more limited is the range of environments to which a component can be ported. Furthermore, excessive accuracy/precision can sometimes result in an unexpected leap in memory or execution cost, due to the need for an alternative implementation (e.g., a two-word representation). An analysis of accuracy/precision needs should address the potential reuse scope, but this is not the place to add unneeded capability. Once an accuracy/precision is determined, the component's operating characteristics should be defined accordingly.

A.5 Named Numbers, Constants, and Literals

A.5.1 Use named numbers and constants wherever appropriate for clarity, maintainability, and safety.

Named numbers and constants can more fully express the intent of the programmer, thus making the program easier to understand and helping to guarantee that it is used as intended.

GA.5.1.1 Declare an object as a constant if it is not intended to change during execution.

This makes the programmer's intent more obvious to the reader. It also promotes safety by disallowing incorrect attempts to change the value. It can also be used to isolate system-dependent features (for example screen size, terminal colour, or function keys) to simplify maintenance.

GA.5.1.2 Except for a few special cases (for example, the numbers 0 or 1), use named numbers or constant numeric objects rather than numeric literals.

This has two advantages. First it provides a name for the number that can help explain its significance. Second, it provides a single point for changing the value of the number if that becomes necessary, thus supporting maintenance and modification. Note that each of these advantages may apply in the case of a string constant versus a string literal. On the other hand, string literals are often self-documenting, making the program more readily understandable than if a named constant were used.

A.6 Expressions

A.6.1 Structure expressions to avoid ambiguity, reduce complexity, and increase clarity.

GA.6.1.1 Use parentheses to make operator precedence easily visible to the reader.

In a long expression it can be difficult for the reader to see the intended evaluation order, e.g.:

```
Earnings := Salary * Base_Hours + Salary * 1.5 * Overtime_Hours - FICA -
            State_Tax - Local_Tax;
```

is difficult to interpret at a quick glance. A more readable alternative is:

```
Earnings := (Salary * Base_Hours + Salary * 1.5 * Overtime_Hours)
            - (FICA + State_Tax + Local_Tax);
```

GA.6.1.2 Do not build in dependencies on the evaluation order of operands.

In many instances, the order in which operands of an expression are evaluated is not defined by the language, and is therefore implementation-dependent. For example, consider:

```
Value := Function1 (Value1) + Function2 (Value2);
```

It may appear unimportant whether the left or right operand of the "+" operator is evaluated first. However, suppose *Function2* is called first, and in the course of its processing it changes *Value1*. This would cause a different result than if the evaluation order were reversed. This may work correctly for the life of a component in its initial system (perhaps because the implementation will always call *Function1* first), but malfunction when the component is reused with another compiler. It may be necessary to break the statement into two statements to control evaluation order.

GA.6.1.3 Use Boolean expressions in preference to the equivalent control structures.

For example,

```
return Possible_Prime >= Smallest_Prime;
```

is clear and concise, while

```
if Possible_Prime < Smallest_Prime then
   return False;
else
   return True;
end if;
```

adds an unnecessary branch construct to the code and makes the assignment seem more complicated than it is.

GA.6.1.4 In referring to elements of record aggregates, use named associations instead of positional ones.

Named associations make explicit which components of the record are being referenced, and thus make the program more readable. When positional associations were to be used, it is necessary to look back to the declaration to determine this correspondence.

A.7 Control Structures

A.7.1 Select control structures to support understandability of algorithms.

*GA.7.1.1 Avoid the use of labels and **goto** statements.*

Labels and ***goto*** statements are generally agreed to be opposed to the principles of good software engineering practice, and are rarely necessary. Some uses that may be justified are:

- In some algorithms it might be desired to jump out of a loop back to its beginning in order to restart it.
- Sometimes a program is developed by translating from an algorithm originally designed for another language. It may be very difficult to determine how to restructure the algorithm to avoid a ***goto***, and safer to simply use the same structure.

Any use of labels and ***goto****s* should be subject to independent scrutiny to determine if it cannot be better avoided. Also, the scope of the label should be limited by using a ***begin-end*** block, so that transfers to the label cannot occur from elsewhere.

GA.7.1.2 Be sure that each loop's exit condition is explicit and apparent.

Most loop exits are determined by the bounds on a ***for*** clause or the condition on a ***while*** clause, making it clear how the loop will terminate. However, it is legal to write a loop without such a clause. This should occur in only two cases:

- The loop includes an ***exit*** statement.
- The loop is part of a task body intended for continuous execution.

In both instances, the ***loop*** statement should be commented to that effect. Comments may also be appropriate at the point at which ***exit*** statements occur, if ***exit***s are nested and it is not readily apparent where control will go.

*GA.7.1.3 Avoid deeply nested loop **exit** statements.*

As discussed in *GA.7.1.2*, difficulty of understanding increases with depth of nesting of ***exit***s.

*GA.7.1.4 Do not use **case** statements when **if** statements are more appropriate, and vice versa.*

An ***if*** statement should be used when selection of alternatives depends on a test of Boolean conditions. Use of a ***case*** statement in this situation adds unneeded complexity. A ***case*** statement should be used when the selection is based on the value of a single variable or expression of a discrete type other than Boolean. Using instead a sequence of ***if***s again adds unneeded complexity.

Choices of a ***case*** statement are mutually exclusive, while ***if*** statements allow the possibility that one test depends on another. Correct use of these control structures makes algorithms more understandable.

GA.7.1.5 Use control structures instead of highly complex expressions to make algorithms more understandable.

For example, a single expression could be used to calculate the number of days in a month, taking into account the complicated rules for leap years. A reader who didn't know these rules would have no way of verifying the correctness of the algorithm. On the other hand, the individual rules could be expressed as **if** or **case** statements, presenting each rule individually and allowing room for explanatory comments.

A.7.2 Select control structures that promote safety.

*GA.7.2.1 Do not use **when others** as a programming shortcut.*

The use of ***when others*** reduces the compiler's ability to detect omitted cases and as a result may allow erroneous values to go through. This makes use less safe and maintenance more difficult. However, the use of ***when others*** is preferable when enumerating the other values would be cumbersome because of their number.

A.8 Exceptions

A.8.1 Use exceptions only to provide the clearest and most flexible way of handling unusual situations.

Exceptions are included in Ada as a way of breaking control flow when unusual errors occur that prevent the execution of the normal statement sequence. They allow for separation of the code for handling unusual situations from the normal code flow, making both aspects easier to understand.

GA.8.1.1 Do not use exceptions as a normal control flow mechanism.

If the situation handled by an exception can be expected to occur during normal processing, an exception is not the best construct. This technique obscures flow of control when it should be explicit. Also, exceptions are a somewhat expensive programming construct and are often not a major target of compiler optimization (since they are expected to occur infrequently). Using them as regular programming mechanism adds unneeded overhead.

GA.8.1.2 Use caution in interpreting the occurrence of predefined exceptions.

There may be many reasons why a particular predefined exception is raised. It is not always clear at any one point exactly what those circumstances are. Making assumptions about these circumstances may later lead to difficulties in determining the cause of errors. For example, the exception ***Numeric Error*** will be raised by an attempt to divide by zero. However, the exception handler should not assume that every ***Numeric Error*** means a division by zero; it may just as well have been raised by an overflow condition.

GA.8.1.3 Do not explicitly raise a predefined exception.

This simply obscures the programmer's intent. Presumably a specific condition is anticipated by the programmer, and it should be represented by a specific user-defined exception. Also, this "exception overloading" can cause a problem if the user of the procedure that raises the exception wants to treat that exception differently than unanticipated occurrences of the same predefined exception.

GA.8.1.4 Clearly indicate all possible exceptions and the conditions under which they may be raised.

This can be accomplished by declaring all the exceptions in one place, with appropriate comments, or by including a section in the prologue discussing exceptions.

A.8.2 *Provide user flexibility in the way exceptional conditions are handled.

Users of a reusable software component may have varying needs for the handling of exceptional conditions. A component that provides alternatives has greater reuse potential.

GA.8.2.1 *Where possible, provide an alternative means to avoid raising an exception.*

In some instance in which a component might be reused, the reuser might wish to avoid the occurrence of exceptions. If it is possible to straightforwardly check to determine if an exception would occur, offer such a check function to the user. For example, a stack package might include a check function *Stack.Is_Empty* in addition to the defined exception *Stack_Empty*. A user who wishes to avoid an exception could call the check function before a call to *Stack_Push*. (Sometimes such check functions are too costly to be practical.)

GA.8.2.2 ** Provide a way to let the user correct problems.*

The RSC should raise an exception if recovery can be easily accomplished outside the RSC. If raising an exception would result in loss of information, provision should instead be made for the user to supply a subprogram to correct the problem.

GA.8.2.3 ** In a reusable component, recognize that users' desired exception handling will vary.*

If exceptions are handled within a reusable component, a generic formal subprogram parameter will allow the user to provide the specific operation to be performed by the exception handler. With this approach, a default handler should also be provided. Alternatively, the reusable component can propagate the exception to the caller. In this case, the exception is a part of the component's defined interface.

A.8.3 *Do not build in implementation dependencies through use of exceptions.

Components intended for reuse should not explicitly raise, or depend upon, implementation-defined exceptions. Such exceptions make software non-portable.

A.9 Program Structure and Compilation Units

A.9.1 Structure a program to maximize the independence and modularity of its components.

Programs should be composed of independent, modular components. Such a structure enhances understandability, makes the program more maintainable, and maximizes the reuse potential of individual components.

GA.9.1.1 Emphasize the principles of coupling and cohesion.

A general principle of good programming says that actions within a program unit should be highly cohesive. This is maximized when a subprogram carries out a single function. The second basic principle is that of minimal coupling between separate program units—i.e., simple interfaces with little interdependence. These principles are mutually supporting.

GA.9.1.2 Limit subprogram length to a manageable level—perhaps two pages.

An individual project should establish a length limit for subprograms, with exceptions subject to some independent approval. A recommended length is two pages, exclusive of any header comments that precede the start of the subprogram specification. Limiting a subprogram to two pages means that the subprogram listing can be opened up so the entire subprogram structure can be seen at one time. It also has the advantage of forcing the program to be broken up into "bite-sized" pieces, encouraging modularity and cohesion. If necessary, the Ada ***inline*** code feature can minimize any performance impact of this convention.

GA.9.1.3 Use separate compilation to add clarity and maintainability.

Separately compiled subprograms facilitate maintenance by minimizing the recompilation required by a change. Subunits—separately compiled bodies of program units declared within another program unit—also provide a mechanism for limiting length of program units.

A.9.2 Use packages to implement objects.

Packages provide a mechanism for implementing abstract objects (as in object-oriented programming methods). Packages should be used to create complete objects by grouping the data representation with the appropriate operations.

*GA.9.2.1 * Provide a basic set of operations for each object.*

The following basic operations should be provided if applicable, unless there is a specific desire to exclude that capability from the user.

- Create and initialize an object.
- Determine an object's value and attributes.
- Assign/change the value and attributes of an object.

- Destroy an object.
- Convert between types.
- Visit (iterate through) all elements of a compound object.

*GA.9.2.2 * Do not split object implementations across packages.*

Objects should not be split across packages at the same level. The object definition and its operations should be in a single package. The package may, however, make use of layered packages to partition the object into layered objects. There should be well-defined interfaces between layers. An example is the layered database management system with separate layers for transaction processing, data manipulation, logical I/O, and physical I/O. Each layer effectively implements a different object.

A.9.3 Hide unnecessary detail and capability.

The principal of information hiding dictates that only the information and capability the package user needs should be made visible and available to him. Other information and capability should be hidden. This helps make programs more understandable by presenting only "interesting" information to a reader. It also supports safe usage by avoiding inadvertent change or misuse of package data.

GA.9.3.1 Include only visible properties of an object in the package specification; hide all non-visible details in the package body.

A package specification should include only those properties (i.e., types, variables, and subprograms) intended for use by the package user. The additional types, variables, and subprograms required to implement the package should be in the package body, where they are not accessible by the package user.

GA.9.3.2 Use the private part of a package specification only for information that must be in the private part, not as a general hiding mechanism.

Only the full definitions of private types and deferred constants should be in the private part of a package specification. Any other hidden declarations should be in the package body. Both hide the declarations from the user; however, a change to a declaration in the private part will require that any dependent packages be recompiled. This is an unnecessary dependency that can be avoided by placing these declarations in the body.

GA.9.3.3 Hide the use of machine and implementation dependent features in package bodies.

If these dependencies are localized, moving to another machine will require changes only to the body of the package and not to all places that use the information.

A.10 Parameters

A.10.1 Make the purpose and use of each parameter clear.

Parameters define the interfaces of a software component. The user needs to know how each parameter will be used by the component, and how to provide a correct value for each parameter.

GA.10.1.1 Name parameters to clearly identify their meaning to the subprogram.

Parameter names should give some indication of the role the particular parameter plays in the subprogram. For example, instead of:

```
procedure Divide (Value1 : in Integer;
                  Value2 : in Integer;
                  Value3 : out Integer);
```

use

```
procedure Divide (Dividend : in Integer;
                  Divisor : in Integer;
                  Quotient : out Integer);
```

GA.10.1.2 Use types to constrain the allowable values for parameters.

Ada's type matching capabilities can help avoid unallowable parameter values, making parameter requirements clear and eliminating the need for the subprogram to check explicitly. For example, the preceding procedure could use a type other than *Integer* to constrain *Divisor* to a nonzero value.

GA.10.1.3 Make parameter modes explicit.

Ada provides that when no parameter mode is stated, the default mode is ***in***. However, using the default mode makes the reader wonder whether the omission was intended. An exception to this rule applies to function parameters; the mode ***in*** is the only legal mode, so it need not be declared explicitly.

A.10.2 *Use generics to enhance reuse potential.

Ada generics increase the parameterization potential of a component, and hence add flexibility that makes the component reusable in a wider range of circumstances.

*GA.10.2.1 * Explicitly consider whether each intended reusable component should be a generic.*

Each component intended for reuse should be considered from the standpoint of what range of reuse is possible—i.e., in what ways can the procedure be varied and still be useful. If this analysis indicates that generic parameterization can increase reuse potential without unacceptable performance overhead or impact on understandability, the component should be implemented as a generic.

GA.10.2.2 * *Use generic formal constants (in preference to generic parameters) for values that do not change within a given application.*

Parameters that vary from one target application to another, but do not vary within a particular application, should be generic formal constants and not subprogram parameters. A value that does not change throughout an application is effectively a constant (compare with the conventions in Section 5, Named Numbers, Constants. and Literals). Passing such a value as a parameter in each subprogram call adds needless overhead. Making it a generic parameter on instantiation avoids this overhead and allows the compiler to treat it as a constant by applying standard optimization techniques.

GA.10.2.3 * *Use generic parameters to support tailoring of a component for an individual instantiation.*

In general, parameters used to tailor a component to an application should be implemented as generic constants, types, and subprograms. Parameters used in the operation of the component should be implemented as subprogram parameters.

A.11 Tasks

A.11.1 Use Ada tasks to implement parallel processing.

GA.11.1.1 Use separate tasks to perform synchronization.

A task should be either an active task or a synchronizing task, but not both. Active tasks spend most of their time performing particular actions in order to achieve their objective, and will, on occasion, interact with other tasks. Synchronizing tasks control these active tasks, usually to provide some kind of protection from undesirable results. For example, they may protect a data structure from simultaneous reading and writing. The major advantage of tasking is that it allows for the separation of function and control. A task that performs both the active and synchronizing roles violates this principle. It will be confusing to the reader, and will make the job of maintaining the software more difficult.

GA.11.1.2 Within an accept statement, carry out only those actions that must occur during the rendezvous.

This minimizes the time during which the calling task is idle as the result of a rendezvous.

A.11.2 Avoid implementation dependencies in the use of tasks.

GA.11.2.1 Make no assumptions about the order or speed of execution of different tasks except that implied by Ada rendezvous semantics.

Ada semantics provide no assurances about the relative speed or order of execution of different tasks. Any program that depends on a specific order may work in some cases and not in others. In particular, it may well work differently when ported to a different operating environment. To avoid such latent errors, no assumptions should be made about the order of execution.

GA.11.2.2 Do not rely on pragma PRIORITY to implement task synchronization.

Task priorities defined by pragma *PRIORITY* should be used only to indicate relative degrees of urgency, not for task synchronization. Priorities do not imply that a lower priority task should be interrupted to execute a higher priority task. Also, in a multiprocessor environment, the lower priority task may execute simultaneously with the higher priority task on another processor. Thus assumptions about how *PRIORITY* influences execution order should be avoided.

GA.11.2.3 Keep task priorities simple and understandable.

The number of different task priorities should be kept small, and each priority level should be given a name by declaring constants of predefined type *PRIORITY.* A program that uses many different priority levels is harder to understand because it quickly becomes difficult to assess the effect of the priorities. Task processing is more clearly understood if constant objects are used for the priorities instead of numeric literals, because they indicate more of a relative ordering than an exact one. The program is more portable if these constant names are used, because only the values of

the constants would need to change as a result of a change to the predefined type *PRIORITY.*

A.12 Other Areas

A.12.1 Minimize the use of machine-dependent and implementation-dependent features.

Machine-dependent and implementation-dependent features should be used only to permit access to specific machine features, or when determined to be critical to achieving required performance. These features make a program less portable, and also add detail that may overwhelm the reader. Any use of these features should be subject to independent review.

GA.12.1.1 If any machine- or implementation-dependent features must be used, localize them in a package.

If these features are used, they should be localized inside packages with comments that explain why they are needed and what effect they have. These features are normally difficult to understand and require some additional explanation; comments prevent misunderstandings. Localizing them into packages isolates them for modification and helps reduce dependencies.

A.12.2 Avoid suppression of built-in checks unless required for performance.

Built-in checks should not be suppressed (by using the generic function *Unchecked_Conversion* or the generic procedure *Unchecked_Deallocation*) unless required to achieve acceptable program efficiency. If such suppression is necessary, it should be done only after the program has been proven reliable and then only after all other means of achieving acceptable efficiency have been exhausted. If checks are omitted, this should be clearly documented in the prologue.

A.12.3 Make pragmas conspicuous by their position in the source text.

If pragmas can be seen, they are more likely to be understood. It they are not positioned in a way that calls attention to them, they run the risk of being overlooked.

A.12.4 Explicitly close all files.

Relying on the operating system or runtime environment to close files at termination introduces an unnecessary implementation dependence. In particular, exception handlers must close any open files.