NATO STANDARD FOR SOFTWARE REUSE PROCEDURES

Volume 3
(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)

Table of Contents

Table of Contents (Continued)

List of Figures

List of Tables

PART I

INTRODUCTION AND BACKGROUND

Section 1

Introduction

The *Standard for Software Reuse Procedures* is designed to provide guidance for software projects that wish to practice reuse by making significant use of reusable software components available in the NATO Reuse Library.

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

1.1 Purpose and Scope

The reuse library provides a foundation for reuse, but it will not in itself assure that reuse is achieved; specific procedures are required to guide projects in its effective use.

Software Reuse is an important aspect of controlling and reducing software costs and improving quality. The practice of software reuse can be significantly increased through the use of an appropriate standard for reusing existing software. The *Standard for Software Reuse Procedures* is a prescriptive document designed to provide concrete guidance for practicing reuse through use of the reuse library.

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

This is one of a set of three manuals developed by NACISA to provide guidance in software reuse. This manual specifically addresses the practice of software reuse on a development project. The other two documents are standards for the creation of reusable software components and the management of a library of reusable components.

1.2 Guide to Using this Manual

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

The *Standard for Software Reuse Procedures* manual 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 frame of reference for the reader. Part II is the actual standard. Its major sections address the reuse library, requirements analysis, design principles, detailed design and implementation, quality assurance and test, and documentation.

Within Part II, each regularly numbered paragraph forms part of the standard, and is considered mandatory in meeting the reuse objectives address 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, indentifying 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 will want to use this manual as a basis for generating project-specific guidance, for incorporation in other software development practices adopted for the project.

Section 2

Applicable Documents

This is one of a set of three documents, specifically addressing the reuse of existing software in projects. The other two documents provide standards and guidelines for the creation of reusable software components and for the management of a software reuse support organization:

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

Other references that may be of value to readers of this manual include:

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

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.

Section 4

The Reuse Library

The NATO reuse library is a source of RSCs intended for the use of NATO, National, and Contractor personnel in developing specific application systems.

The services the library provides include:

- Acquisition and classification of RSCs
- Support in identify RSCs that are potentially usable in software development efforts
- General organizational focus for reuse
- Coordinated configuration management of RSCs

The library's contents include all classes of RSCs - requirements, designs**,** documentation, tests, etc. as well as code. These may have come from other NATO programs, or may have been acquired commercially or from the public domain.

Specific procedures for the use of the library are documented in the user's manual for the library support tools. The following subsection provides an overview of those procedures. Subsequent subsections provide standards and guidelines for use on a specific project.

4.1 Reuse Library Overview

The library provides tools and procedures to enable software developers to identify and obtain RSCs that can be used in their applications.

The reuse library is run by and for individual NATO programs. RSCs are collected and refined incrementally. The intent is to build reusable products for current and future programs to improve productivity.

4.1.1 Expected User Interactions

It is anticipated that the user will use the library and library support tools to identify and obtain RSCs. In exchange for receiving RSCs, the library asks that feedback be given on the effectiveness of the RSC and its accompanying documentation. This feedback will be solicited from the user after some agreed-upon period. The feedback can be both positive and negative. The purpose of feedback is two fold; to provide other reusers with lessons learned and to improve the quality of the RSC, library support tools, and library procedures.

The user is also expected to submit problem reports concerning RSCs; submit new and enhanced RSCs; notify the library of changes made to RSCs; notify the library of user changes of address, telephone number, and project; and notify the library of point-of-contact changes for supported projects.

4.1.2 RSC Quality Standards

The library provides RSCs of the highest quality possible. They have identified a minimum acceptance criteria that the RSC must meet in order to be included in the library. Through a system of incremental refinement, the RSCs are expected to improve through reusers refining documentation, improving test coverage, and resubmitting improved RSCs to the library.

4.1.3 RSC Classification Scheme

RSCs are organized in the library according to faceted classification schemes. See the appendix on classification scheme in *Standard for Management of a Reusable Software Component Library* for more information. When an RSC is first received by the library, the applicable domains for the RSC are identified and the RSC is classified, based on its documentation, according to a domain-specific classification scheme.

PART II

STANDARD

Section 5

Reuse Library Procedures

5.1 Administration Procedures

It is critical to successful reuse efforts that the library track reuse experiences.

In order to provide effective configuration management of RSCs, the library must track the uses of the RSCs.

5.1.1 Register reuse occurrences and experiences with the Reuse Library.

It is important for the maintenance and configuration management of an RSC that the library know of all reuses of the RSC. Information on problem reports and new releases of RSCs can only be forwarded from the library to contact points for identified reuses.

G5.1.1.1 Obtain user accounts with the reuse library for each individual on the project.

It is important to identify the specific user of an RSC so the library can obtain feedback. In cases where the library supports multiple projects, it is also important to identify the users project in order for the library to coordinate maintenance and configuration management of the RSC.

G5.1.1.2 Avoid accepting copies of RSCs from sources other than the reuse library.

Note that this applies only to RSCs directly supplied by the library. New releases and problems with licensed RSCs shall be handled directly by the licensor.

G5.1.1.3 If an RSC is obtained from a source other than the library, notify the library of its use.

RSCs once in circulation may be more obtainable from other sources. These can be used, but inform the library so updates and problem reports are forwarded to you.

G5.1.1.4 Provide feedback to the reuse library on experience with RSCs and library support tools.

Provide both positive and negative feedback. The feedback concerned with using particular RSCs is needed by other reusers when they are considering selecting an RSC. The feedback will serve as lessons learned to help others when they are reusing the RSC. Feedback is also used to identify areas of improvement for the RSCs, the classification scheme, and the library itself.

G5.1.1.5 Notify the reuse library of all problems encountered with RSCs.

It is essential that all problems noted with RSCs be reported to the library, even if the problem is not critical to the application in which the problem is discovered. Other users of the RSC may be affected by the problem in a critical manner, and may not be aware of the existence of the problem.

G5.1.1.6 Submit new and enhanced versions of RSCs to the reuse library.

The only way the library can grow and improve is if users submit new and enhanced RSCs to the library. The library is based on the foundation of incremental improvement of RSCs. Through reuse and enhancement, RSCs become refined and improved.

When new versions are submitted, be sure to identify the RSC that provided the basis for the new RSC. The library tracks that information for configuration management and problem reporting purposes.

5.2 Searching for RSCs

It is important to identify the RSC which most closely meets the requirements as either an exact match or as a sound basis upon which to build a new RSC.

A major inhibitor to reuse is the time and effort required to locate RSCs that meet the reusers' needs. The reuse library has searching mechanisms based on the faceted classification scheme for overcoming that hurdle.

5.2.1 Define your search requirements in terms of the classification scheme.

G5.2.1.1 Obtain a report of the classification scheme and identify the facets of interest for your search.

From the domain collections supported by the library, identify the domains of interest for the target application. From the classification scheme identify the facets applicable to the point in the life cycle of the development process. For example, during design, *function* and *object* may be the only facets of interest, while during coding, those facets as well as *language* and *platform* may be of interest.

G5.2.1.2 Select terms in the classification scheme that most closely match the system requirements.

Define the search in terms of the facets. For example, identify the function the RSC must perform, such as *sort*; and identify the abstract object involved, such as *queue*.

Many retrieval tools support synonyms and recognize similarities or closeness between terms. If no terms in the classification scheme match the requirements, look for close terms. For example, a closeness exists between the term search and sort because some search algorithms include a sort algorithm.

G5.2.1.3 Use the requirements to limit or broaden the search.

Initially, search only for absolute requirements; specify the criteria the RSC must meet. Based on the results of the search, refine the candidate list.

If no RSCs are found based on the search criteria, relax the requirements by ranking the requirements in priority order, removing the lowest priority requirement from the search list, and searching again. Repeat this process until some candidate RSCs are found. Note, any RSC selected from this list will now have to be modified to meet the base requirements.

If too many (defined as more than the user is willing to search through) RSCs are found, expand the requirements by adding "would like to have"s (that is, features that would be nice in the RSC, but are not requirements) to the search criteria. Continue adding requirements until the candidate list is of manageable size.

5.3 Selecting RSCs

The library provides various information about RSCs which aids in the selection of specific RSCs for use in application systems.

Having identified a list of RSCs that meet the requirements, the next step is to select the best RSC for the application.

5.3.1 Use the support material provided by the library to decide which RSCs to select.

The library provides various material about each RSC, which is aimed at aiding the user in selecting among several RSCs.

G5.3.1.1 Examine the RSC's abstract before committing to the RSC's use.

Each RSC in the library has an abstract associated with it. The abstract contains high-level general information about the RSC. It will contain all known and potential "show stoppers" so the RSC can be quickly selected or eliminated from consideration.

The abstract will also identify assumptions and guarantee parameters within the RSC. It defines the scope in which the RSC is assured to work. Example phrases which may appear in abstracts are: "This RSC assumes nonrecursive processing." and "Parameters passed to this RSC must be in the range of 1 to 1000."

G5.3.1.2 Examine the general quality and reusability rating and the summary of recommendations for RSCs under consideration.

The quality rating and recommendation summary is based upon expert opinion and actual reuse experience with the RSC. Examine the summary of recommendations for descriptions of enhancements and problem reports. The fact that an RSC has outstanding problem reports should not automatically disqualify it from consideration. Read the problem report description to determine the applicability and severity of the problem in order to determine if the RSC should be considered.

The quality rating and rational may contain a synopsis of the feedback that the library has received from other users about the RSC. The feedback can provide useful insights and hints on reusing the RSC. The feedback is generally anonymous; however, direct contact with past reusers may be possible through the library.

G5.3.1.3 Where possible, select RSCs with acceptable maintenance and quality metrics.

For each RSC, various software quality metrics may be available in the quality and reusability rating. Examples are:

Number of reuses—known instances where the RSCs has been incorporated into an application system

Number of inspections—the number of times users have considered the RSC for reuse

Complexity—generally based on McCabe's method[1] of evaluating software complexity

Number of problem reports—known outstanding defects with the RSC

These metrics present a rough estimate of the reusability of the RSC and help eliminate unsuitable candidates. Rules of thumb for evaluating these metrics include:

- Look for RSCs with a high number of reuses.
- Be suspicious of RSCs with a high number of inspections and a low number of reuses.
- Look for RSCs with low complexity values.
- Select RSCs with a low number of problem reports.

There may be a correlation between these metrics which will aid the selection process. For example, an RSC with high complexity or a high number of problem reports will probably have a low number of reuses. If that is not the case, then it is likely that the RSC has great reuse potential and should seriously be considered, for it means that the RSC is being reused in spite of these obstacles.

Inform the library of RSCs with a high number of inspections and a low number of reuses because these RSC may be misclassified.

5.3.2 Select complete RSCs.

RSCs are separate entities. Often, however, they are not complete or independent. For example, an RSC may be a top-level graphic design diagram with no detail decomposition associated with it, or the RSC may be a software code module that has no test plan. In both these cases, the RSC is complete, but related information is missing.

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

G5.3.2.1 Choose independent RSCs or RSCs that dependent only on other RSCs.

Often an RSC will not be independent; it may depend on other program elements. For example, in Ada, program units may "with" other program units, which creates dependencies between them. Without the "with"ed program unit, the RSC cannot be compiled and used. The RSC's reuser manual will provide guidance on obtaining or developing the needed "with"ed program unit, but given the option, choose an RSC that is dependent on other RSCs.

The dependencies between RSCs are tracked by the library and may be examined by potential reusers through the library.

G5.3.2.2 Choose RSCs that have a full set of documentation.

Given the choice between RSCs which include test plans, test procedures, and design documentation and those that do not, all other considerations equal, choose the RSC with the documentation. Keep in mind that the design documentation for an RSC may be a separate RSC in the library.

5.4 Extracting RSCs

Most RSCs are obtained directly through the library and include all the information necessary for their reuse.

RSCs extracted from the reuse library are delivered with the following items:

- Reuser's manual
- Test plans, objectives, scripts, expected results
- The RSC itself
- General quality and reusability rating and rationale
- Summary of recommendations

5.4.1 Follow reuse library procedures when extracting RSCs.

G5.4.1.1 In cases where extraction is not made directly through the library, follow the RSC's extraction directions to obtain it.

In some cases, the RSC will not be directly obtainable through the library. Examples of these are RSCs that have to be licensed or are controlled by an external mechanism such as a CASE or Configuration Management (CM) tool.

The library support tools will still help identify these components and upon a request for such an RSC, will supply directions for obtaining the RSC.

G5.4.1.2 Specify a feedback date to provide feedback to the library.

When a user requests an RSC, the request is recorded as a potential reuse. It is necessary for library configuration management to determine if the request culminated as an actual reuse in a target system. The feedback date that the user agrees to at the time of the request is the mechanism the library uses to make this determination.

The feedback date is usually 3 months from the date of request. This typically gives the user adequate time to decide to use the RSC and to gain experience using it. The feedback session generally amounts to a telephone call between the library and the user in which the library asks if the component was reused and, if so, what experience did the reuser have using the RSC.

G5.4.1.3 Identify in which application the RSC will be reused.

For long term maintenance of the RSC, it is important that when the potential reuse is identified as an actual reuse, the program Computer Software Configuration Item (CSCI) and Configuration Software Component (CSC) the RSC is used in be identified. New releases and problem reports dealing with an RSC will not be sent to individuals; they will be sent to program Software Configuration Control Boards (SCCBs) and CM organizations.

G5.4.1.4 Notify the reuse library when extraction problems are encountered.

In order to help the library improve, notify the library of any problems encountered in obtaining the RSCs. The library also solicits feedback on its services and performance.

G5.4.1.5 Extract related RSCs which complete the selected RSC.

Many reuse support tools allow the extraction of all the related RSCs that complete the RSC being extracted. For example, when extracting an RSC that is an Ada package specification, you may specify the extraction of RSCs that are the Ada package body, separate program units declared in the package, and associated design documentation.

5.5 Using RSC Documentation

The documentation provided with an RSC can significantly facilitate its use.

5.5.1 Make maximum use of the documentation that accompanies the RSC.

Many RSCs will come with the traditional software documentation associated with the RSC type, such as test plans. All RSCs are delivered with a reuser’s manual.

G5.5.1.1 Follow the guidance supplied in the reuser’s manual when integrating the RSC in the target application.

The reuser’s manual is intended to provide a complete description of how to use the RSC. It includes instructions and examples for integrating the RSC into the target application. The following figure contains an example outline for a reuse’s manual.

REUSER'S MANUAL

1. INTRODUCTION
   - purpose of the document
   - overview of the component
2. FUNCTION
   - operation
   - scope
3. INTERFACES
   - RSC specification
   - external references and parameters
   - interfaces by class
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 5.1 - Outline for a Reuser's Manual

G5.5.1.2 Don't modify the RSC documentation to conform to target system standards.

Often RSC documentation will be different from the form required for the rest of the system. This difference is acceptable providing the differences are approved and documented.

Unless required information is completely missing, incorporate RSC documentation as it is. In cases where information is missing, supply the missing information as an addendum to the RSC documentation, and submit the addendum to the library for incorporation in future releases.

G5.5.1.3 If possible, reconcile project documentation guidelines with those in use for the library RSCs.

To avoid the previous problem of RSC documentation not conforming to system standards, where possible, define the system documentation standards to conform to those used by the library, or propose changes to the library guidelines if appropriate. “Appropriate” means this and foreseeable future projects will benefit from the change.

Section 6

Requirement Analysis

The requirements specification of a software system has a direct impact on the amount of reuse that is possible, creating or eliminating opportunities for reuse of library RSCs.

Awareness of available RSCs can help guide requirements specification to prevent inadvertently precluding reuse opportunities. For example, consistency between system interfaces, such as communication protocols, could be ensured through reuse of the requirements specifying the protocol.

The following subsections address the role of reuse during Requirements Analysis, using Domain Analysis products and RSC Architectures and Subsystems.

6.1 Role of Reuse and Reuse Library

The requirements phase should identify opportunities for reuse and establish requirements that support them during the requirements analysis phase.

6.1.1 Establish reuse requirements and identify reuse opportunities as early as possible.

The requirements establish the framework in which reuse of library components is possible. The goals of requirements analysis are to define system requirements that are complete, consistent, measurable, and implementable. Reuse aids in meeting these goals.

Reuse of existing requirements statements can have a number of advantages, including:

- Consistency among related systems
- De facto standards set in place
- Increased reliability due to use of a proven implementation
- Overall risk reduction

Planned reuse of designs and code also has advantages, including:

- Reduced scope of the requirements analysis task
- Cost and schedule savings
- Risk reduction through increased confidence in system specification

G6.1.1.1 Before committing to specific requirements, examine the library for available components that may apply.

Requirements specifications themselves are potential RSCs, and may be available in the library. In order to incorporate existing design or code components, the requirements may have to be modified.

Knowledge of available RSCs and their architectures/functions may have an influence on the specification of requirements for the system.

Do not ignore the fact that the library may already contain existing lists of requirements for similar applications/systems as a result of previous domain analyses/ implementation efforts. The earlier in the project that reuse is begun, the greater the potential benefits.

G6.1.1.2 Avoid overspecifying requirements.

Requirements statements which overspecify implementation details can preclude reuse of available RSCs. For example, if a requirement states that a system, or a part thereof, “shall read an array...,” this might preclude reuse of available queueing or linked-list objects/components which might be capable of performing the same functions with lowered time and resource costs.

G6.1.1.3 Specify system requirements in terms of the library’s classification scheme.

Use object and functional names that are consistent with the library’s classification scheme. Using the classification scheme terminology increases the likelihood of finding components that can be reused. Note, there is likely to be more than one classification scheme within the library, based on the existence of different domains. This must be taken into account when specifying system requirements.

The project and/or the library should generate a set of procedures for specifying requirements using the existing library classification scheme.

G6.1.1.4 Select development methods and tools which support reuse.

Most design methods support reuse, but the degree to which reuse is supported will vary with different design methods. For example, structured methods support reuse at a functional level. Reuse of low-level functional units is easy and is currently practiced in many organizations. Programs designed using functional methods typically benefit from reuse mostly in the detailed design, coding, and unit testing phases.

Reuse for programs designed using object-oriented methods typically spans the software life cycle, since the design method is chosen during the requirements analysis phase.

G6.1.1.5 Identify any specific reuse requirements.

Any specifically desired reuse should be identified as a requirement. Make these requirements in terms of classes of components, not specific components (unless the quality and reliability of the component is explicitly and directly known), specifying requirements that existing components can meet.

6.2 Domain Analysis Products

Domain analysis can provide a means to maximize reuse of existing software as well as reuse within the project.

Domain analysis is the process of identifying and making explicit the knowledge about some class of problems--the problem domain--to support the description and the solution of those problems.

The steps that are undertaken as part of domain analysis are:

Knowledge Acquisition—Information regarding the class of problems is collected and analyzed in terms of describing the objects contained within and outside of the specified domain(s).

Domain Definition—The objects identified during the knowledge acquisition phase are analyzed in order to determine the specific boundaries of the described domain(s).

Model Formulation—The objects identified as belonging to the domain under analysis are modeled to gain further understanding of their roles within the domain.

Model Evolution—The model is further analyzed and refined to develop a taxonomic model of the domain, which is used to demonstrate the model's organization and semantics.

The usual outputs of the domain analysis process are the following:

Functional Hierarchy—a hierarchical model of functional requirements

Entity-Relationship Model—a model showing the relations among and interfaces between the objects defined within the domain

Generic Software Architecture—the objects described in the E-R model represented as software components

Taxonomy—provides a classification scheme to define the objects within the domain

Standard Requirements—a set of generic requirements which any system addressing the problem domain must fulfill

Domain-Specific Language—a vocabulary for describing and classifying domain-specific components

Design and Development Guidelines—a generic development framework based upon the architecture and components that make up the domain under analysis

6.2.1 Consider carrying out a domain analysis for a particular application, or across a class of applications.

In some cases it may be appropriate to identify commonalty within a single application. In other cases, it may be appropriate to carry out the analysis across a broader class of applications within a given problem domain. The former case would probably occur when a company or agency has limited resources and only seeks to satisfy a limited number of specific application requirements. The latter case might occur when a company or agency has more available resources and wishes to lay the groundwork for a more ambitious or extended set of multisystem projects that span more than one application domain.

Availability of appropriate resources to perform the analyses as well as the extent of the company or agency's commitment to the application domain should be taken into account when deciding on the extent of the domain analysis or whether it should be performed at all. For example, an agency which intends to develop/field a large number of Command and Control systems over a number of years would probably benefit from a domain analysis of the broad field of automated message handling (AMH). A firm that is required to deliver a single numeric spreadsheet application as a part of a single office automation effort could probably forego such an effort in the spreadsheet domain.

G6.2.1.1 Use the library to evaluate any existing domain analysis products related to the domain of this system.

A domain analysis may already have been carried out for the domain of the system, in which case products may be available in the library.

The design and development guidelines should contain procedures for applying existing domain models to specific applications.

G6.2.1.2 Define explicit boundaries and scope for the domain to be analyzed.

If the boundaries of the domain are not explicitly defined, it is difficult or impossible to ascertain whether the analysis is, in fact, complete.

G6.2.1.3 Make sure the domain boundaries define a discrete application area.

A single system may contain many domains. A system does not necessarily map to a single domain. There may also be domains within domains. For the purpose of performing a domain analysis, make sure to explicitly define the scope of the analysis.

G6.2.1.4 When defining the classification scheme for the domain, adopt terms from related domains whenever possible.

This practice will enhance the possibilities for reuse across related, but nonidentical domains, for example, military AMH and commercial electronic mail systems.

G6.2.1.5 Use terms from the classification scheme consistently.

The more consistently the terms are applied and used, the more likely it becomes that commonalities and differences will be correctly identified, maximizing reuse potential.

G6.2.1.6 Select terms that span the domain, not just your single instance of the domain class.

Selection of terms that span the entire domain, rather than being application-specific can serve to enhance reuse across the entire domain. For example, referring to the generic term “message header information”, rather than to specific ACP 127 numbered format lines would tend to make the specified generic architecture useful for E-mail as well as military AMH systems. (If you don’t know whether the terms span the domain, you may not know enough to perform the analysis effectively, and you may want to reassess the decision to perform the analysis.)

G6.2.1.7 Define acceptance criteria for the model of the domain.

Identify how you will know you have completed the domain analysis. This includes identification of the products of the analysis, as previously stated, and acceptance/ verification criteria for their contents.

G6.2.1.8 Remove application-specific features from the architecture model.

Removal of application-specific features, like the removal of application-specific terminology, enhances the possibility of more wide-spread reuse of the domain analysis products. Include only features common to several applications in the domain.

G6.2.1.9 Specify assumptions made about the application domain.

The underlying assumptions made about the application domain, including the assumed requirements of the application domain, should be specified clearly. This will tend to prevent use of the generic architecture in an inappropriate manner. For example, if the necessity for multilevel security processing is assumed in a message-handling generic architecture RSC, it may not prove to be a cost-effective choice for a local electronic mail application.

G6.2.1.10 Specify any restrictions imposed by the application domain model.

It is vital to successful reuse that restrictions implied by the application domain model be explicitly specified. This will serve to prevent attempted reuse of the generic architecture in an inappropriate manner which might lead to increased, rather than decreased cost and effort. For example, inappropriate reuse of a data base management architecture whose retrieval algorithms are inefficient for very large numbers of records can be prevented by explicitly stating and explaining this restriction.

G6.2.1.11 Include recommended use guidelines.

Specify a set of guidelines, similar to a “reusers manual” for the generic architecture. This will act to prevent use of the RSC in an inappropriate manner.

6.3 RSC Architectures and Subsystems

Extremely high payoff can be realized through the reuse of existing architectures and subsystems.

6.3.1 Reuse existing architectures and subsystems whenever possible.

Reuse of large-scale components, whether the architectures only or the implementation of those architectures in code, can greatly benefit the project. However, if not done carefully, it can also greatly increase risk.

G6.3.1.1 Examine potential reusable architectures and subsystems for compatibility with the planned interfaces and design approach.

The set of architectures and subsystems included in the library should be examined with potential reuse in mind. The potential reuse areas include:

- “Closest-fit” analysis to determine whether an existing architecture/subsystem can be used with little or no modification
- Examination of the library for smaller RSC components that can be combined into subsystems that address system requirements

G6.3.1.2 Evaluate the impact of potential architecture/subsystem reuse.

The impact of using an RSC architecture/subsystem should be assessed. The procedure for performing such an analysis should include:

- Guidelines for performing a cost benefit analysis (trade-off analysis) of reuse with modifications versus development from scratch
- Guidelines for estimating the project/schedule impacts based upon identified RSCs and modifications or enhancements

G6.3.1.3 Integrate RSC architectures with other outputs of the requirements analysis task.

When the component fits the development method (for example, structured components with structured development or object-oriented components with object-oriented development) integration is simplified. The object or module to be integrated can be made to fit into the appropriate place in the module hierarchy with relative ease, in a manner similar to project-developed code.

When the component does not fit the development method (for example, structured component with object-oriented development) integration is more difficult. Some suggested integration approaches include:

- Encapsulation of a structured component within an abstract data-type object with visible interfaces to facilitate integration into an object-oriented system
- Repetition of an object-oriented component at appropriate places within a structured module hierarchy

The key to integration using components created by means of dissimilar design approaches may be granularity of the choice of components. Large granularity of reused components such as entire subsystems will probably be easier to integrate by means of “pipe-fitting” than smaller programs/objects where multiple interfaces may have to be modified extensively to allow them to work successfully together.

An important part of the requirements integration effort is the analysis of the cost benefit of using the RSCs identified. Some areas to be analyzed include:

- Definition of RSC architectures and the cost of fitting them into the traditional outputs of the requirements analysis task

- Cost of identifying and implementing procedures for identifying RSCs for later incorporation

- Cost benefit analysis (trade-off analysis) of reuse with modifications versus development from scratch

- The impacts of project/schedule cost estimates based upon identified RSCs and modifications or enhancements

Section 7

Design

Software design is the key phase in which reuse occurs; reuse that is not designed-in rarely happens.

Design takes advantage of any requirements reuse that has been identified, but goes far beyond that to identify parts that implemented the chosen solution.

The following subsections address using existing designs in new applications, integrating design components, and reverse engineering designs from code RSCs.

7.1 Designs as RSCs

Reusing design components is essential to achieving code reuse, and is valuable even if code is not reused.

Design RSCs can be used with or without accompanying code RSCs to achieve reuse benefits.

7.1.1 Consider using existing system and subsystem designs in their entirety.

When the design of an existing system or subsystem matches a significant number of requirements, consider using the entire design as an RSC. Examples might be such items as electronic mail systems, data base query or error handling subsystems from existing products. It should be kept in mind that the coarser the granularity of the reused RSC, the greater the potential savings in development time and effort.

G7.1.1.1 Examine available RSCs before beginning design, and identify potential reusable designs.

The first step is to identify what components can be reused, then acquire both design and code components. Be sure to include components which were derived from any reused requirement components.

G7.1.1.2 Assess each component's reuse status.

A list of possible statuses, based on representation and requirements, might be:

RSC TYPE	DESIGN REPRESENTATION	REQUIREMENTS MATCH
Design	Representations Match	Complete Match
Design	Representations Match	Incomplete Match
Design	Representations Differ	Complete Match
Design	Representations Differ	Incomplete Match
Code	No Design Representation	Complete Match
Code	No Design Representation	Incomplete Match

Table 7.1 - RSC Reuse Status

Based on the reuse status, decide whether or not the component must be modified; in general, modify as little as possible. Only consider modifying those that don't fully meet requirements.

G7.1.1.3 Don't modify design components unless absolutely necessary.

It is frequently possible, especially when dealing with Ada objects, to add additional levels of abstraction and encapsulation to existing RSC objects which adds new functionality or hides undesirable existing functionality. This sort of repackaging gains the advantages of modification without losing the benefits of using an unmodified RSC.

G7.1.1.4 Consider not requiring graphic representations for reused design components that are not already represented graphically.

It may be worthwhile to tailor project design representation standards specifically to address this point. Some sort of standardized off-page connector and a description of the appropriate interfaces to an existing RSC could save a great deal of time and expense during the design phase.

G7.1.1.5 Evaluate cost benefits of reusing components that must be modified.

Assess the cost effectiveness and disadvantages of RSC modification. Modified RSCs must be treated as a new development as far as design documentation is concerned unless the new component is going to be submitted to the library as a new RSC. In many cases, if a component is modified, there is no justification for the component not meeting system design standards and conventions. So potentially, there is a lot of re-work that may be necessary over and beyond developing design for new functionality.

G7.1.1.6 Consider reuse of only a design component when reuse of the corresponding code component is not possible.

Designs represented as Program Design Language (PDL) or as graphic representation can be implemented in many languages. If you are reusing a design components that has a corresponding code component in the language you need, use it, but do not hesitate to use the design component only, even in those cases where the programming language does not map.

G7.1.1.7 In the system design documentation, identify reused components. Include a brief description of the design representation if it differs from the rest of the system design. Include a reference to more information on the design method used by the RSCs.

It may be worthwhile to tailor project design representation standards specifically to address this point. Some sort of standardized off-page connector and a description of the appropriate interfaces to an existing RSC could save a great deal of time and expense during the design phase.

7.2 Designing for Reuse

7.2.1 Avoid constraining system design to unnecessarily preclude reuse.

Requirements and design statements which overspecify implementation details can preclude reuse of available RSCs. For example, if the design states that a module "shall receive an array containing X," this might preclude reuse of available queueing or linked-list objects which would perform the same functions with lowered time and resource costs.

G7.2.1.1 Treat reused design components as black boxes.

A "black-box" component is one in which the user of the component's services need have little or no knowledge of the component's inner workings. This implies that all of the component's services are available by means of standard, visible interface methods. Note that even if these services are not of immediate use, do not modify the RSC to remove them. Significant savings can be accomplished in later phases of development by using the services provided with the RSC which support debugging and testing.

G7.2.1.2 Honour the component's interfaces.

Do not attempt to access or modify internal state-data residing within the component by bypassing the component's visible interfaces. This means that archaic and intrusive programming practices must be avoided. For example: if a First In, First Out (FIFO) queue is the required design component, do not assume at the design phase that it will be implemented by means of an array whose individual entries can be accessed outside of the queueing abstraction.

G7.2.1.3 Use terms from the design component consistently in the rest of the design.

Attempt to map the project's names for objects and functions which interact with RSCs to the RSC's names for those same objects. For example, if the Queue_Package exports methods are called "Put_Item" and "Get_Item", do not refer to these methods elsewhere as "Push" and "Pop".

G7.2.1.4 Where possible, use terms from the classification scheme in the design.

Whenever possible, map the project module name to the terms that describe it in the classification scheme. For example, it would probably be preferable to refer to an object that encapsulates a Prioritized Queue RSC object as “Prioritized_Queue_Package” instead of “Data_Transport_Mechanism”.

7.3 Reverse-Engineering Designs from Code RSCs

7.3.1 Specify and follow a standardized method for performing reverse engineering.

A project standard method for performing reverse engineering of code RSCs into design objects should be created. All project personnel should be informed of this method.

For example, when generating a detailed design from code, extract PDL from the code first. Translate simple statements to English text representing the statement, keeping program unit structure intact. Keep complex statement structure as it is. Translate conditionals within complex statements to English text representation.

G7.3.1.1 Use automated support when available.

Where automated methods exist to extract design information from code units, such as automated PDL processors, use them.

G7.3.1.2 If possible, avoid mixing object-oriented design components with components developed using Structured techniques and vice versa.

When there is a choice, avoid using a mix of modules developed using different or contradictory design methodologies. When there is no option, keep in mind that there must be effort applied at the *design stage* to ensure consistency of interfaces and usage across the RSC components.

G7.3.1.3 Be careful to keep the design consistent with the code component.

The design object must be kept consistent with the actual code object if the design is to truly reflect reality. Areas which must be addressed include:

- Keep interfaces consistent with the code object.

- Keep overall terminology consistent with the code object.

G7.3.1.4 Submit the design to the library as additional RSC information.

Submit the design to the library as additional RSC material for possible reuse on other projects. Be sure to explain its origins and development history. Ensure that the link between the code component and the design component is maintained.

7.4 Integrating Design Components

Existing components can be reused effectively and safely only when their

design is integrated with that of the overall system.

7.4.1 Integrate RSC's design with overall system design.

Choice of integration approaches depends on the compatibility of the design methodologies and the interface match of the components.

G7.4.1.1 Evaluate "as is" integration versus integration with modification.

The project should provide specific guidance, based on objective criteria, on whether or not to modify an RSC, including:

- Explicit procedures for performing cost benefit analysis on the modification effort
- Explicit guidance regarding how to weigh the various modification issues, based upon the factors affecting a particular program
- Explicit guidance regarding resolution of possible documentation inconsistencies

G7.4.1.2 Verify correct conformance to the RSC's interfaces.

It is vital to verify that the existing RSC's interfaces are capable of supporting (or approach supporting) the requirements of the system. It is also vital to ensure that the interfaces are used and integrated in accordance with their stated design and function. This may require either modification or encapsulation of the RSC to achieve.

G7.4.1.3 Select an integration approach compatible with overall project design methodology and goals.

When the component fits the development method (for example, structured components with structured development or object-oriented components with object-oriented development), integration is simplified. The object or module to be integrated can be made to fit into the appropriate place in the module hierarchy with relative ease, in a similar manner as project-developed code.

When the component does not fit the development method (for example, structured component with object-oriented development), integration is more difficult. Two suggested integration approaches are:

- Encapsulation of a structured component within an abstract data-type object with visible interfaces to facilitate integration into an object-oriented system
- Repetition of an object-oriented component at appropriate places within a structured module hierarchy

The key factor to integration using components created by means of dissimilar design approaches may be the granularity of the choice of components. Components with large granularity such as entire subsystems will probably be easier to integrate by means of pipe-fitting than smaller programs/objects where multiple interfaces may have to be modified extensively to allow them to work successfully together.

G7.4.1.4 Consider the use of Computer Aided Software Engineering (CASE) tools to support design integration.

CASE tools can frequently be used to great advantage to support design integration. They can, however, have drawbacks as well.

- Possible advantages to CASE utilization:
    - CASE tools often perform analysis on the design, collect metrics, and check consistency automatically.
    - CASE tools can often consistently generate both graphic and textual documentation as well as code skeletons with less effort required.
- Possible disadvantages to CASE utilization:
    - CASE products are sometimes “king of the mountain” tools, closed systems that don’t recognize other design approaches or another tool’s design products, making integration difficult or impossible.
    - CASE tools sometimes require extensive user training and have lengthy learning curves.

Section 8

Implementation

The implementation phase carries out the reuse planned in earlier phases, continues the reuse started in earlier phases, and may identify additional opportunities for low-level component reuse.

Implementation involves the integration of code components into the evolving system. Guidelines must address ways of doing this. Modification issues are particularly important.

The following subsections describe integrating code components, modifying code components, testing code components, and effects of RSC integration on CM.

8.1 Integrating Code Components

Careful attention to code integration ensures a consistent and coherent product.

8.1.1 Select an integration approach compatible with overall project design methodology and goals.

When the component fits the development method (for example, structured components with structured development or object-oriented components with object-oriented development), integration is simplified. The object or module to be integrated can be made to fit into the appropriate place in the module hierarchy with relative ease, in a similar manner as project-developed code.

When the component does not fit the development method (for example, structured component with object-oriented development), integration is more difficult. Two suggested integration approaches are:

- Encapsulation of a structured component within an abstract data-type object with visible interfaces to facilitate integration into an object-oriented system
- Repetition of an object-oriented component at appropriate places within a structured module hierarchy

G8.1.1.1 Clearly identify reused modules or parts of modules with consistent in-line commenting.

When included RSCs or portions of RSCs are clearly labeled as such, a great deal of unnecessary maintenance and testing effort can be avoided. This procedure makes it clear to the developers/testers/maintainers exactly which portions of the code are to be tested/altered versus those which either should be left alone or may require additional documentation and quality assurance (QA) steps in order to modify.

G8.1.1.2 Establish coding standards that facilitate the ease and safety of code reuse.

Practices that can be standardized to facilitate ease and safety of reuse include:

- Choose and enforce naming conventions that make identifier-name conflicts unlikely. Some implementation languages (such as, Ada) makes this a less serious problem, but the lack of naming conventions can still cause confusion to human readers.

- Tasking and exception mechanisms should be used in a standard and consistent manner. Enforced uniformity of usage of these items will simplify and speed up both implementation and maintenance efforts.

- Avoid global data items, especially those that are shared. Communication of these items is better handled by means of formal visible interfaces since it frequently becomes difficult or impossible to forecast accurately how a black-box RSC may alter the state of these items at any given time.

8.2 Modifying an RSC

RSC modification must be carefully controlled to avoid unnecessary cost and risk.

8.2.1 Control and document all RSC modifications.

It is often better to modify an existing RSC than to start from scratch, but such modifications must be done carefully to avoid risk.

Whenever possible, do not modify the RSC itself, but, instead, encapsulate it in a new object which can provide additional required functionality or limit access to undesirable functionality.

There are some trade-offs to be considered when deciding whether or not to modify an RSC. The advantages to RSC modification include:

- Ownership. A local project developer feels personally responsible for the modified code.

- Modification results in a correct and exact fit with no unnecessary or inappropriate parts.

- The efficiency of the code can frequently be increased.

- Modifications can be made according to standards and in a manner consistent with the rest of the project.

The disadvantages to RSC modification include:

- It is costly in terms of both time and resources to modify an existing, functioning module.

- Modification of existing RSCs can introduce errors.
- Modification invalidates prior tests and requires resources and time to retest.
- Modification invalidates future upgrades and releases of the component. Minimally, it will require additional analysis to determine compatibility with new releases, and will probably require reiteration of the modifications to the new release to make it consistent with the rest of the system.

G8.2.1.1 Establish an explicit procedure for assessing proposed RSC modification.

The procedure should provide specific guidance, based on objective criteria, on whether or not to modify an RSC, including:

- Explicit procedures for performing cost benefit analysis on the modification effort
- Explicit guidance regarding how to weigh the various modification issues, based upon the factors affecting a particular program

G8.2.1.2 Modify RSC materials as little as possible.

Whenever possible, do not modify the RSC itself, but, instead, encapsulate it in a new object which can provide additional required functionality or limit access to existing undesirable functionality.

Developers must be given explicit guidance to avoid any unnecessary cosmetic or stylistic modification that is not absolutely required. State and enforce restrictions against the “Well, as long as we’re in here...” mind set.

G8.2.1.3 Document the changes to RSC materials internally as part of the change log for the component.

Provide a list of changes in the RSC’s formal change log as well as in internal commentary.

G8.2.1.4 Document the steps externally so they can be repeated, if necessary, when new versions of the RSC are released.

Document all changes made to a given RSC in external, as well as internal, documentation. This can prevent confusion and save large amounts of time and effort, especially in cases of disaster recovery.

G8.2.1.5 Notify the library that you have modified the RSC and why.

The RSC library must be notified of any changes applied against an RSC. The library can and should serve as an additional repository for change documentation.

8.3 Testing RSCs

Testing of RSCs should be done routinely, similar to regression testing, not only when modifications have been made.

8.3.1 Perform unit and integration testing of RSCs.

When appropriate test material comes with the component, it should be used to verify the correct functioning of the RSC as part of normal project unit and integration testing. Any modifications to the RSC under test must be tested at the same time. Submit any new tests generated for inclusion in the library as RSC-related material.

When appropriate test material does not come with the component, it must be generated by the project in accordance with project testing standards. The project may wish to consider only requiring minimal testing for any functions which are part of the RSC, but not utilized by the project.

8.4 Effects on CM

The reuse of software developed elsewhere imposes additional CM requirements.

8.4.1 Incorporate RSC's into overall project CM.

CM procedures must recognize that a reusable component also exists elsewhere, and provide the necessary links.

G8.4.1.1 Establish an explicit procedure for evaluating new releases (of RSCs that have been reused) from the library.

This standard also imposes Testing requirements/impacts. Regression testing will probably be necessary to determine whether a new release is acceptable. The procedure must address such topics as:

- Decision criteria for determining whether, how, and when to accept a new release
- Understanding the reason for any changes to the RSC
- Assessing the impact of new releases on the developing system

G8.4.1.2 Augment the project's configuration system to provide necessary information for RSCs.

The project's configuration control system must be augmented to maintain and track the following types of information:

- Version control data
- Problem-report tracking

- Regression-testing data and test results
- Support for providing information required by the library organization, such as records of modifications and user/developer feedback reports

Section 9

Quality Assurance and Test

An effective QA and test program can help ensure that reuse standards and guidelines are followed and avoid risk and cost/schedule problems.

QA for programs in which reuse is an objective must include explicit procedures to verify that reuse standards/guidelines are followed.

The following subsections address using RSC test material, plans that foster reuse, and problem resolution.

9.1 Using RSC Test Material

Tests available from the library with RSCs can enhance the savings available through reuse.

9.1.1 Take advantage of test materials supplied by the library.

The library should contain test materials for each RSCs which will be included when the RSC is extracted from the Library. This test material will vary depending on the type of RSC. For example, a design RSC will have different test material than a code RSC. The following table outlines the test material that should accompany RSC:

RSC TYPE	TEST MATERIAL
Requirement Lists	Test Objectives and Test Plans
Design Diagrams	Test Objectives and Test Plans
Program Design Language	Test Objectives, Test Plans, and Test Procedures
Code	Test Objectives, Test Plans, Test Procedures, Test Data, Test Results, and Test Support Programs (Test Drivers and/or Stubs)

Table 9.1 - RSC Test Material

G9.1.1.1 Review the RSC test material for applicability, coverage, understandability, and implementability in the target system.

RSCs, because they make no assumptions about the system they will eventually become a part of, often contain extraneous "baggage" that may not be used in the target system. Areas in the test plans and procedures that deal with unnecessary functionality of the

RSC need to be identified and documented as such in the overall system-test documentation.

Verify that the functionality of the RSC that is used in the system is adequately and completely covered in the test material.

Assess the clarity of instructions and objectives. Look for inappropriate assumptions about how the component will be used.

Look for instructions and/or objectives that cannot be implemented in the target system. Examples may involve tests dealing with timing constraints that cannot be met or interrupts that cannot be generated.

G9.1.1.2 Augment any identified deficiencies in RSC test material and submit new versions to the reuse library.

Based on the review, augment the test material to address the deficiencies. The augmented material should be separate, not integrated with the rest of the test material, unless the deficiency addressed is applicable to future reusers. Augmented test material that is integrated because of its applicability to future reusers should be submitted to the library.

In the target test material, identify the RSC test objectives and test cases that do not apply, with an explanation and justification of why they do not apply. Do not simply modify the RSC test documentation to delete nonapplicable parts because they may reappear with future releases of the RSC. Keep in mind that if test cases are not identified as nonapplicable, they must be executed and passed during testing.

G9.1.1.3 If the RSC has been modified, modify the RSC test material correspondingly.

Review the test documentation to verify that the test objectives and test plans still reflect complete coverage on the modified RSC.

Generate any new test objectives, plans, and procedures needed. Directly incorporate the new test material since modified RSCs are treated as new components. Future releases of the original component should not directly affect the test material.

G9.1.1.4 Execute RSC unit test procedures before beginning integration test.

RSCs must be tested prior to incorporation into a target system. In many cases, since the RSC has already been developed and fielded in other systems, this may be thought unnecessary. It is not. There are enough differences in machine architectures and compilers to make unit testing prior to integration a fundamental and minimum risk-reduction mechanism.

G9.1.1.5 Incorporate RSC test documentation in overall project test documentation.

In the project test documentation, explicitly identify RSC test documentation. Specify any modifications that have been made. Identify nonapplicable parts and include a rationale and justification for the nonapplicability.

G9.1.1.6 Provide feedback to the reuse library on the RSC test material and submit any RSC-related problem reports.

It is important that the library be apprised of problems with RSC test material as well as problems with RSCs uncovered during testing. This will help others reusing the same components.

9.2 QA Plans that Foster Reuse

QA activities must monitor conformance to reuse standards and guidelines.

Reuse activities should be addressed as part of QA's overall role in monitoring the software development process and in approving key milestones.

9.2.1 Ensure that QA procedures address reuse standards and guidelines.

Project standards and procedures that do not explicitly support reuse may inadvertently prevent reuse. Standards and procedures, particularity QA standards and procedures, must ensure that reuse is explicitly addressed in order for maximum reuse benefits to be achieved.

G9.2.1.1 Develop QA inspections and checklists to ensure reuse is considered at each stage in the life cycle.

This reuse manual specifies that reuse issues be addressed at each stage of the life cycle. A QA checklist confirming such things as the library was searched and domain analysis considered significantly increases the likelihood that reuse will be considered. Check lists should address items such as:

- Technical review of the requirements includes a check for overspecific requirement statements
- Performing a domain analysis is considered
- Where possible, terms from domain-specific library classification schemes are used in requirements and design
- The selected design methods and tools do not inhibit reuse

G9.2.1.2 Develop QA procedures for ensuring that specific reuse requirements, if present, are clear, testable, and verifiable.

As in the case of all requirement statements, reuse requirements must be clearly stated with no ambiguity in meaning, they must be testable in that some procedure or process can be applied that yields objective results, and they must be verifiable in that the testing results must clearly indicate fulfillment of the requirement.

Early resolution of unclear or untestable reuse requirements is necessary in order to prevent the reuse requirement from causing more damage in the forms of cost and schedule overruns.

Reuse objectives that may be specified as good ideas or overall project goals that become requirements can burden the developer more than reuse saves.

G9.2.1.3 Define procedures to ensure that modified and enhanced RSCs are submitted to the library.

Modify QA inspection and review procedures to include checking that changed RSCs are submitted back to the library. Without QA inquiry, the library may never know when RSCs are changed and future reusers of the RSC may have to reinvent the change.

G9.2.1.4 Define QA procedures to ensure that all RSC-related problem reports are submitted to the reuse library.

Modify the SCCB (Software Configuration Control Board) procedure to include a check for this. Include problem reports that come from all aspects of the RSC; design material, test material, reuser manual, etc.

G9.2.1.5 Perform QA review of selected RSCs for compliance to target application requirements.

The QA procedure of code inspection should be applied to RSCs that have been selected for integration in the target system in the same manner they are applied to newly developed code.

G9.2.1.6 Require QA participation and sign-off in decisions to modify or not to modify RSCs.

The decision to modify or enhance an RSC has significant impact in the areas of development cost, schedule, testing, CM, and maintenance. QA involvement in this process is critical to ensure that:

- All alternatives are considered
- Cost/benefit analysis are performed
- An informed decision is made

Obtain early QA approval on nonmodified RSCs which do not conform to project standards, in order to avoid schedule and cost impacts due to QA vetoing the decision later. Include QA approval on the nonapplicability of RSC test objectives and test plans as well.

G9.2.1.7 Require QA participation in development inspections and reviews with emphasis on identifying reuse areas.

QA personnel are likely to see a wide range of different developer's design/code. They should be trained to identify common functionality areas that might be targets for reuse. This reuse can be either in the form of utilization of existing RSC's in place of developed software or in the identification of portions of the design/code that should be developed as new RSCs.

G9.2.1.8 Define the documentation requirements and format for RSCs and their documentation within the target application.

Often RSCs and their support documentation will differ from the form required for the rest of the system. This difference is acceptable providing the differences are approved and documented. The means of documenting the differences and the mechanism for ensuring that the differences are approved and documented falls in QA's domain.

The nonconforming RSCs and documentation cannot be simply incorporated without explanation. There must be documentation explaining the deviation. In particular, the standards for the unit development folder must allow for the identification of units that are RSCs, the software design documents must explain the RSC's design notation if it differs from the rest, and the software test documents must explain the RSC's test material if it differs from the rest.

9.3 Problem Resolution

Special problem-resolution procedures may be necessary for RSCs.

9.3.1 Ensure that problem-resolution procedures address reusable software.

Problems discovered with RSCs must be handled differently from problems discovered by nonreused code. This difference is due to the fact that coordination and additional CM are required for the RSC.

G9.3.1.1 Identify program-specific problem-reporting procedures for RSCs.

Specific detailed procedures which identify such things as what information needs to be sent to the reuse library, where to send the problem reports, and how to verify that the problems are still valid should be defined.

G9.3.1.2 Publish the reuse-related problem-reporting procedures in a manner similar to the other project engineering procedures.

The RSC problem-reporting procedure should be available to all program engineers, possibly as part of a software development plan, CM plan, or QA plan.

G9.3.1.3 Include the reuse-related problem-reporting procedures in the project or organizational training program.

Project specific and organizational training programs should include all reuse-related standards and procedures, particularly, the problem-reporting procedure.

G9.3.1.4 Identify and budget for maintenance responsibility for RSCs.

Maintenance of an RSC is the responsibility of the submitter of the RSC to the reuse library. Maintenance of RSCs which have been modified, unless accepted by the reuse library as a new RSCs, is the responsibility of the organization which has modified the RSC.

The reuse library is responsible for ensuring that the problems get fixed and that there is no duplication of effort in fixing the problems.

G9.3.1.5 Identify a single focal point in the organization to coordinate RSC maintenance.

This person should be a member of the organization's SCCB and identified in project management plans, maintenance plans, etc. as responsible for coordinating the maintenance of RSCs used in the target system. This includes funnelling all problem reports and new releases through the focal point.

Section 10

Documentation

Documentation for reused software must become part of the overall system documentation.

As with all software, reused software must be documented in the target system. No assumptions can be made about the long-term maintenance of an RSC, so the project must plan and document as if the RSC will be maintained within the target system.

10.1 RSC Documentation

10.1.1 Develop project-specific policy on documentation requirements for RSCs.

G10.1.1.1 Where possible, incorporate the RSC documentation as it is.

A consistent policy needs to be established which allows RSC documentation to be incorporated with as little effort as possible while still providing adequate information for the maintainer of the system.

G10.1.1.2 Clearly identify all instances of reuse in the system documentation.

The RSC name or ID should be specified along with the domain name, the version number, and date of extraction. Identify whether modifications have been made, and if so, list the modifications. Also, list any assumptions made about the RSC's behavior in the system.

G10.1.1.3 In cases where new documentation for RSCs is developed, follow the guidelines defined by the reuse library.

In the case where an RSC is received with no design or test documentation, the documentation must be generated for inclusion in the system documentation. This documents the RSC as an RSC. The new documentation generated must follow the standards defined by the library, not those defined by the project. The new documentation should be submitted to the library.

G10.1.1.4 Define the documentation requirements and format for RSCs which do not conform to project standards.

When RSC support documentation differs from the required form for the rest of the system, there must be documentation explaining the deviation. In particular, the software design documents must explain the RSC's design notation and the software test documents must explain the RSC's test material.

G10.1.1.5 Treat the documentation for modified RSCs as nonmodified RSC documentation.

The documentation for modified RSCs, even if it is modified, should be treated in the same way as other RSC documentation. There should be no cause to make the documentation conform to project standards. Any modification made to the documentation should follow the requirements defined by the reuse library.

G10.1.1.6 Explicitly identify RSC test documentation in overall system test documentation.

Specify that the test material is from an RSC, that modifications have or have not been made, and that certain test objectives are nonapplicable, if necessary. Include a rationale and justification for the nonapplicable test objectives.