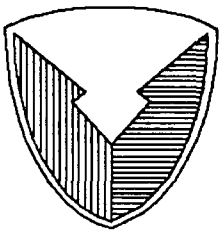
(2)



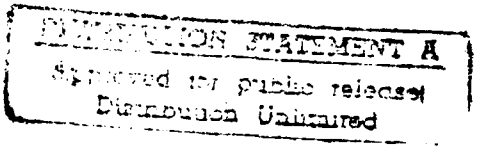
CECOM

CENTER FOR SOFTWARE ENGINEERING

SOFTWARE PROCESSING ENGINEERING
TECHNOLOGY DIVISION

Subject: - **SOFTWARE REUSE METHODS**

Final Report



CIN:C04-087OO-0001-00

JULY 1990

CLEARANCE OF INFORMATION FOR PUBLIC RELEASE

SUBMIT FORM IN TRIPLICATE

TO: Commanding General
U.S. Army CECOM
ATTN: AMSEL-IO
Fort Monmouth, NJ 07703

FROM: Director
CECOM Center for Software Engineering
ATTN: AMSEL-RD-SE-AST-SS
Fort Monmouth, NJ 07703-5000

DATE: 2 Nov 90

In compliance with AR 360-5, Public Information Policies, as supplemented, the attached manuscript/abstract is submitted for clearance for public release. (Material should be in triplicate if local clearance is requested, in 6 copies if clearance through Headquarters, AMC, is required. See paragraphs (A) through (J) below).

Section I. DESCRIPTION

TITLE
PAPER: Software Reuse Methods
ABSTRACT

AUTHOR(S): Steve Goldstein

NAME OF PERIODICAL
(If for publication): include country if outside CONUS
For submission to NTIS and/or DTIC

EXT NO: 22606

NAME OF CONFERENCE OR SYMPOSIUM (If for presentation):

DATE AND PLACE OF CONFERENCE:

DATE CLEARANCE REQUIRED: 2 Dec 90

PAPER _____DOES
__X____DOES NOT CONTAIN
CLASSIFIED INFORMATION

MATERIAL _____DOES __X____DOES NOT CONTAIN ANY OF THE FOLLOWING MATTER REQUIRING AMC CLEARANCE. IF SO, INDICATE AND EXPLAIN ON SEPARATE SHEET:

(A) Information which is, or has the potential to become an item of national or international interest.

(B) Information on subject of potential controversy among the military services or with other federal agencies.

(C) Information on new weapons or weapon systems or significant modifications or improvements to existing weapon systems, equipment or techniques. Unofficial prior publication of such information does not constitute authority for official release.

(D) Information on significant military operations, potential operations, operation security, and military exercises.

SEL FORM 1012, 1 Oct 1985
(Supersedes SEL Form 1011 and SEL Form 1012, 1 Jan 79)

(E) Information on military applications in space; nuclear weapons and the components of such weapons, including nuclear weapons effects research; chemical warfare and defensive biological and toxic research; high energy lasers and particle beams technology; and nuclear biological, chemical (NBC) defense testing and production, policy, programs and activities.

(F) Information and materials involving critical military technology.

(G) Information concerning communications security, electronic warfare, signal intelligence, and computer security.

(H) Subject matter involving or referring to other service interests or those of other government activities outside AMC.

(I) Information on tests, studies or experiments not yet officially approved by appropriate echelon concerned. For example, studies or tests directed by DA, AMC, or other agencies outside CECOM.

(J) Subject matter which by its nature implies official positions or scientific attitudes of higher headquarters or agencies outside CECOM.

Section II. DISTRIBUTION CONTROL

If document contains technical data information, a Distribution Statement must be applied. Indicate the appropriate statement in this section.

Approved for public release; distribution is unlimited.

Section III. VERIFICATION

The attached manuscript does not contain classified information. Additionally, understanding the hostile intelligence in open source publications, the undersigned have individually conducted an OPSEC review, find its release clearly consistent with the AMC OPSEC Program, and verify that it_____DOES__X___DOES NOT contain any of the matter itemized in paragraphs (A) though (J).

Activity OPSEC Officer
(Typed name and title) Signature/Date
JOSEPH ARETINO
AMSEL-RD-SE-CRM-SE-O

Division Chief
(Typed name and title) Signature/Date
EDWARD J. GALLAGHER JR. GM-14
C, Software Eng Tech Branch, SPET Div, CECOM , CSE

Activity Security Manager
(Typed name and title) Signature/Date
JOSEPH ARETINO
AMSEL-RD SE CRM-SE-O

Associate Director
(Typed name and title) Signature/Date
for MARTIN I. WOLFE GM-15
C, Software Processing Eng Tech Div, CECOM , CSE

SEL FORM 1012, 1 Oct 1985
(Supersedes SEL Form 1011 and SEL Form 1012, 1 Jan 79)

SOFTWARE REUSE METHODS

Final Report

PREPARED FOR: U S ARMY CECOM
CENTER FOR SOFTWARE ENGINEERING
AMSEL-RD-SE-AST
FORT MONMOUTH, NJ 07703

PREPARED BY: ITT RESEARCH INSTITUTE
4600 Forbes Boulevard
Lanham, Md 20706

JULY 1990

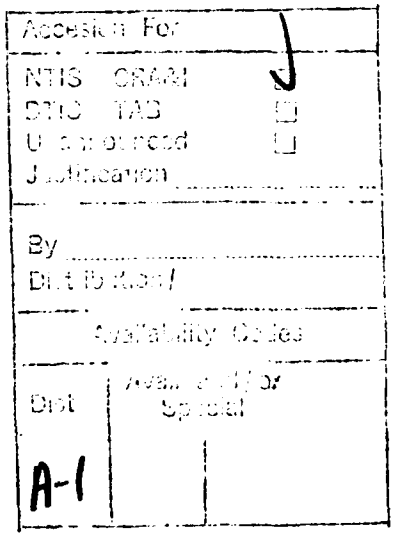


TABLE OF CONTENTS

1.0 INTRODUCTION

Given the increasing number of computerized, software driven systems being designed and implemented throughout the Department of Defense (DoD) and industry, reusability of software has become a critical endeavor. To better prepare software engineers and computer programmers to address the challenge of software reuse, the U.S. Army Communications and Electronics Command (CECOM) has undertaken a program to investigate different software reuse methods. This effort will provide guidelines on reuse strategies for software developers.

1.1 DEFINITION OF REUSABILITY

It is first necessary to clarify the concept of reusability. Specifically, it is necessary to distinguish reusability from portability. For the purposes of this report, we will consider reusability to be the extent to which the services provided by a software unit can be used by other software units. We will define portability as the ease with which a software unit can be transferred to other hardware or operating systems. From this point of view, reusability differs from portability in the following respect: reusability involves using a unit in another software environment, whereas portability involves using a unit in another hardware or operating system environment.

The portability of a system depends on how it is designed and implemented, for instance, whether all system dependencies are localized in one place. It is the nature of low-level routines to interface directly to the hardware and/or operating system. This has the tendency to make these routines very hardware specific, and hence non-portable. Conversely, it is not uncommon to encounter a very portable routine that is built around an application-specific database. Such an application would not be reusable. This suggests that reusability and portability are independent of each other.

In Ada the unit of reuse is the library unit. This is because in Ada reusable software is incorporated into an application through a context clause. A context clause designates the library unit that is to be made visible to the compilation unit. We can speak of the potential for reuse of a program unit nested inside a library unit or secondary unit, but in doing so we are considering that program unit as a potential stand-alone library unit.

1.2 SCOPE OF THIS REPORT

Reusability is widely believed to be a key to improving software development productivity and quality. The reuse of software components amplifies the software developer's capabilities. It results in fewer total symbols in a system's development and in less time spent on organizing those symbols [Kaiser 1987]. The opportunity to reuse software is frequently proposed as one strategy for reducing the cost of developing and enhancing the reliability of complex large-scale applications. However, software reuse usually requires more intense intellectual effort in the initial development of a part and in many instances a decrease in its performance efficiency. For embedded real-time Mission Critical Computer Resource (MCCR) systems, the tradeoff between increased reliability and decreased performance may determine the degree of reusability possible [Gargaro 1988].

The planned reuse of software has been practiced since the advantages for common libraries were recognized in the early days of high-level programming languages. The libraries were usually restricted to include only mathematical and statistical routines that implemented well-defined, numerical algorithms. However, since then, software reuse technology has not progressed to the same level of sophistication as its hardware counterpart. This results from the lack of discipline and formalism in the design and implementation of reusable software. Often, reusability is relegated to an implementation activity that is left to the discretion of the individual programmer [Gargaro 1988].

As part of the ongoing effort to promote the development of reusable software, following issues are examined in this report:

- Reusability Characteristics
- Domain Analysis
- Domain-Independent Approaches
- Domain-Specific Approaches
- Cost/Benefit Analysis for Software Reuse
- Reuse Metrics.

2.0 REUSABILITY CHARACTERISTICS

Reusability is not a binary function. A procedure may be fully reusable, that is, it can be used as is in a different application from which it was written. Or a procedure may require modifications to be reused. Thus, the question is not whether a particular software component is reusable, but the degree to which the software unit is reusable. We can attempt to quantify this by defining the constructs that promote reusability.

In evaluating reusability, we can distinguish three categories of reusability factors:

- design factors
- package factors
- presentation factors.

Design factors are structural components that are relevant to reuse. Package factors are reuse design elements that are unique to Ada packages. Presentation factors are those factors that affect ones ability to reuse a software unit.

2.1 DESIGN FACTORS

Software reuse is affected by several design factors. The following design factors are discussed:

- level of abstraction
- genericness
- size
- reusability of components
- cohesiveness
- coupling
- degree of information hiding.

Level of Abstraction. The level of abstraction represented by a library unit is inversely related to its reusability. The higher the level of abstraction, the less likely it is that it will be reusable. This is because a library unit at a higher level of abstraction is more sophisticated than one at a lower level. For example, one common representation of a map is as a non-directed graph. But a non-directed graph is at

a lower level of abstraction than a map, and therefore can be used in other contexts as well, whereas the possibilities of the use of a map are more limited. A road map would be a more complex and constrained form of a map, but it is at a higher level of abstraction and is less reusable than a map.

Most low-level program units, units found at the lowest level of abstraction (e.g., device drivers, graphics primitives), are very reusable. Low-level program units tend to have a simple, general-purpose interface, be small in size, and perform a single function. These characteristics of low-level software can be empirically applied to reuse. Mid-level program units (e.g., windowing functions, statistical sub-routines) are likely to have a broader function, be larger in size, and have a general purpose interface. They typically are cohesive units that are not coupled with other units. Reusability at this level is high. These units can also be structured for easy transportability by constraining all of the interrupts to a lower level. High-level program units are usually a melange of low-level, mid-level, and "other stuff" supporting a specific application. They are usually large, marginally cohesive, functionally complex, functionally specific, and interdependent with other routines and structures. Rarely are these program units fully reusable. To what degree these units are reusable depends largely on the reusability of their components. For example, a procedure that is built solely with reusable low-level and mid-level program units, must itself be reusable. This does not imply, however, that a reusable procedure has reusable components.

Genericness. Genericness (in the Ada sense) is directly related to the reusability of a library unit. This is a binary decision: either the unit is generic or it is not. A generic unit can be instantiated for any of a given set of types, objects, and operations. In effect, by being made generic a library unit is put at a lower level of abstraction. Thus, the sheer fact that a library unit is generic increases the likelihood of its reusability.

Size. The size of a software unit is inversely related to its reusability. The larger the software unit, the less likely it is that it will be reusable. This is particularly true of systems in which space is tightly constrained. In general it is preferable to have many small, reusable components than few large ones.

Reusability of components. The higher the percentage of code that consists of reusable modules, the greater its reusability. Many times it is desirable to extract sections of code from a previous software development. If that prior effort was comprised of many reusable software modules, it is likely that one could find the needed elements in a completely autonomous form.

Cohesiveness. Cohesiveness is the measure of focus in a software unit. It is directly related to reusability. A routine that performs a single function or operates on a single object is cohesive. Cohesive

modules tend to be easily extracted for reuse in other routines. The less cohesive a module, the more likely it will do things that are not needed by the new host application. As a rule, it is much better from a reuse point of view to have a large number of operations, each of which performs one atomic function, than to have few operations, each of which performs more than one atomic function.

Coupling. Coupling is inversely related to reusability. When one removes a software component from its host environment, one needs to extract all the units intertwined with that component. These units may be lower level subcomponents, data structures, files, or complimentary routines (e.g., push and pop). These inseparable relationships between a software component and other software units are defined as coupling. Obviously, the more ties a software component has, the less likely one could reuse that component.

In the case of library units, the number of library unit dependencies is inversely related to reusability. At the present state of compiler sophistication, importing other library units into a program only increases the size of the executable image. It is unnecessary and wasteful to add to the size of a program if the extra baggage is unused. Moreover, this consideration is related to the size of a library unit: the larger the library unit, the more library unit dependencies it is likely to have.

Degree of Information Hiding. A library unit's degree of information hiding is directly proportional to its reusability. A library unit specification should contain the minimum information to allow that library unit to be used. If it is encumbered with extra baggage, there is less incentive to use it.

A more serious situation concerns compilation dependencies. If compilation unit A "withs" library unit B in A's specification, and if then the specification of B is recompiled, the specification of A must be recompiled (along with its body and any subunits). If the specification of A is recompiled, any compilation unit that "withs" A must be recompiled as well. However if A "withs" B in A's body, or if one of A's subunits "withs" B, and if then the specification of B is recompiled, only the secondary unit that "withs" B needs to be recompiled (along with any of its subunits). In other words, hiding of context clauses in library unit bodies or subunits limits the spread of recompilation overhead. This increases the user's incentive to reuse the library unit.

2.2 PACKAGE FACTORS

The reuse variables that are germane on the package level are:

- sufficiency/completeness of operations
- presence of exceptions.

Sufficiency/Completeness of Operations. A package that does not provide at least a sufficient set of operations for a given type is not as reusable as one that does, and one that provides a complete set is even more reusable. Booch [1987A] defines a sufficient set of operations as being available when "the component captures enough characteristics of the abstraction to permit meaningful interaction with the object;" and he defines a complete set of operations as being available when "the component interface captures all characteristics of the component". The operations in both sufficient and complete sets of operations should be primitive, that is, they should be operations "that can be efficiently implemented only with access to the underlying representation of the object" [Booch 1987A]. Completeness not only refers to program units, but to other constructs visible to the user. These include types, constants, global variables, and exceptions that are found in the library unit specification. For example, users of most reusable components will need certain common constant values (such as a null value of a given type) at some time. The presence of constant declarations in the package interface indicates that such needs have been foreseen.

Presence of Exceptions. Similarly, the presence of exception declarations in the package interface enhances its reusability. Exceptions are even more likely to be needed than constants. Most operations can have exceptional conditions associated with them and so should raise exceptions at the appropriate occasions. Moreover, it is far more meaningful for a locally declared exception to be raised than a predefined exception. In fact, if an exception name is chosen with care, it can provide the user of the package a great deal of information.

2.3 PRESENTATION FACTORS

Presentation factors also affect reuse. The following are of particular interest:

- clarity of identifiers
- simplicity
- presence of documentation

- neatness
- desirability.

Clarity of Identifiers. The clarity of identifiers is directly related to reusability. Ada sets no limit other than the width of a line on the length of an identifier. Thus, the programmer has no excuse for using cryptic identifiers. A great deal of thought should be put into identifier names (something that is all too frequently neglected). Clear, terse, identifier names make it more obvious to the user of a software unit what the unit is designed to do. If identifiers are cryptic, a programmer is more likely to avoid using a software unit because he or she does not know what the unit can do.

Simplicity. Simplicity is also directly related to reusability. Learning the proper use of a program unit results in a certain labor cost. The simpler the unit, the lower the labor cost. Also, there is a frustration factor associated with the simplicity of a unit. Users have little patience for overly complex software and hence will not want to use such a unit.

Presence of Documentation. Similarly, the presence of documentation is directly related to reusability. If a software module is well documented, then it is clearer how to use it. A potential user is more likely to make use of a documented unit than one that is not documented.

Neatness. Neatness refers to the "look" of a program unit and thus is directly related to reusability. Software should be easy to look at. In general, a programmer should make liberal use of spaces and blank lines, indent logical blocks (e.g., loops, branches), and employ a consistent scheme for utilizing upper and lower case. Furthermore, comments should be positioned around (not in) logical blocks of code. An easy-to-read program unit, would definitely be reused more than a cumbersome looking unit.

Desirability. Desirability is directly related, and in fact crucial to reusability. One could make the argument that all software is reusable. It would be simple to write a higher level program that encapsulates any routine, hence, that routine would be reused. But, from a reuse standpoint, we are only interested in software that one would want to reuse. Desirability, therefore, is the degree to which a particular software element is useful.

All of the factors discussed above and their relation to reuse are summarized in TABLE 1.

TABLE 1. RELATION OF SOFTWARE ENGINEERING VARIABLES TO REUSABILITY

Software Engineering Variables	Directly Related	Inversely Related
Design Factors		
Level of Abstraction		X
Genericness	X	
Size		X
Reusability of Components	X	
Cohesiveness	X	
Coupling		X
Degree of Information Hiding		X
Package Factors		
Sufficiency/Completeness of Operations	X	
Presence of Exceptions	X	
Presentation Factors		
Clarity of Identifiers	X	
Simplicity	X	
Neatness	X	
Presence of Documentation	X	
Desirability	X	

3.0 DOMAIN ANALYSIS

This section addresses domain analysis, a potentially powerful process that facilitates high levels of reuse. If an application is to operate within a specific domain, it can benefit from prior knowledge about that domain. This is what domain analysis provides. The knowledge resulting from a domain analysis may then be used to refine the environment in which the application is to be developed. In this way, domain analysis may be used as a fundamental step in creating real reusable components.

Domain analysis generalizes all systems in an application domain by means of a domain model, or domain language, which transcends specific applications; this model identifies the commonalities and laws of an individual domain in a manner that is conducive to the specifications, requirements analysis, and actual design of a software system. In some cases the domain model may result in a library of closely related software components.

The product of a domain analysis is not universally agreed upon. One school of thought [Prieto-Diaz 1987] suggests that domain analysis produces a library of reusable modules that share a certain commonality and predefined relationships based upon the domain in which they are to interact. Another school of thought [Neighbors 1984] suggests that a domain-specific language containing predefined objects and procedures that are abstractions of groups of objects and procedures from common "executable" languages is produced by a domain analysis.

The software community has made only a few forays into the realm of domain analysis. These efforts consist of several untested methodologies and handful of prototypes. The most extensively tested system, Draco by James Neighbors [Neighbors 1984], has a base of a dozen small and moderate domains.

The remainder of this section contains a discussion of the methodology of domain analysis in general terms; an enumeration of the participants' roles and responsibilities for both views of domain analysis; an overview of pertinent related efforts, experiences, and proposed methodologies; and finally, a summary of the pros and cons of domain analysis.

3.1 THE NEIGHBORS APPROACH TO DOMAIN ANALYSIS

If a domain exists that can acceptably describe the objects and operations of a new system, the systems analyst has an environment that facilitates the development of the specifications and requirements of that system. This is the reuse of analysis and is potentially the most powerful form of reuse. This is also the goal of domain analysis. Reuse of analysis is powerful because the other forms of reuse, reuse of design (also known as reuse of information) and reuse of code, may be made, in some degree, to follow from reuse of analysis.

The first step of domain analysis is to define the domain to be analyzed. This is an application-dependent process. The next step is to gather information about the domain with which to identify the common attributes of the domain that may be reused. This is done either by analyzing a group of existing applications within the domain or by developing a group of applications within the domain (a costly and time-consuming option). The objects, operations, and relationships that are consistent throughout the domain are refined into a domain-specific language with which to reason about the domain. This language is also known as the domain model.

The requirements and specifications of a new application will be done in a domain language. This provides the basis for an abstract design which, through refinements, is made into a group of abstract algorithms. These refinements are at least partially automated in most proposed and existing systems.

Development of an application using this methodology requires that a processor be created which interacts with a programmer and produces executable code for the application. This processor must operate on the domain model. Ideally, a generic processor might be developed which allows a domain model as an input.

3.2 THE PRIETO-DIAZ APPROACH TO DOMAIN ANALYSIS

Not all supporters of domain analysis advocate the development of a domain model suitable for automated software life-cycle support. Many support the development of libraries of coded components based on the domain analysis. The basic theory behind the two schools of thought is the same. Both gather information and common attributes on the domain. Both look for the objects, operations, and relationships that are consistent throughout the domain. It is after the information has been gathered that they differ.

Aspects of the domain are stored in a library that contains special information about the relationships among the software components within the library. These strong relationships allow for the selection of some components based on previous selections.

The development of the library from the raw domain information takes place in a series of discrete steps. First, each component (components may be requirements, designs, objects, data structures, or algorithms) is generalized into an abstract form to fit the entire domain. The abstract components are then arranged in groups. The relationships between the abstract components are generalized to fit the entire domain. Finally, the relationships are formed as rules that guide the use of the components.

The encoding of domain knowledge into reusable structures accomplishes four upstream reuse objectives: feature-based selection, constraint-based analysis, domain-driven completion, and domain-driven refinement. Feature-based selection is the selection of objects and associated components based on descriptions of their known features. Constraint-based analysis consists of analyzing the requirements and specifications from the user based on domain-oriented dependencies and relationships. Domain-driven completion fills in the missing requirements and specifications to guarantee completeness. Relationship constraints and operations can be used to pick appropriate refinements for more detailed specifications and designs. This is domain-driven refinement.

3.3 THE PROBLEM OF KNOWLEDGE ACQUISITION

Most researchers in domain analysis feel that the most difficult problem in the production and subsequent use of a domain model is the extraction of the domain-specific knowledge that is to be represented in the model. The formulation of the domain knowledge often requires a thorough understanding of one or more application domains, the objects of the domain, the relationships between those objects, and the ways of processing those objects. This knowledge is difficult and costly to acquire. Two approaches have been taken to achieve this domain knowledge: in one effort the analyst "picks the brains" of one or more domain experts [Neighbors 1984]; in the other effort the analyst analyzes existing systems within the application domain [Horowitz 1984]. Both techniques are expensive and time consuming.

The problem of extracting knowledge from a source that is not familiar with domain analysis is very similar to the problems knowledge engineers have experienced with populating the rule-bases of expert systems. In fact, expert system developers have found that most of the effort and expense often go toward formulating the production rules. To reduce this expense, expert system developers have utilized a variety

of techniques with varying degrees of success. Not the least of these efforts has been the creation of expert systems to guide a domain expert in the formulation of rules. To date, however, there is no known method of removing all of the problems associated with knowledge acquisition. The domain analysis community should carefully follow the expert system community and its efforts, because any useful technologies should be applicable within both fields.

3.4 DEFINING THE DOMAIN

A common problem many domain analysts have encountered is the difficulty associated with defining the boundaries of domains. Most applications deal with multiple domains, e.g., a missile defense system would have to deal with both missile and database domains. If domains are made too small and objects or operations are separated into separate domains beneficial relationships between objects and operations are lost.

Domains that are made too large may become too complex to allow for the completion of the domain analysis. As the number of components within a domain increases, the potential number of relationships increases exponentially. While the actual number of relationships does not approximate the exponential growth of the potential relationships, it too will quickly become too large to be managed effectively when the domain grows too big. Another problem with creating over-sized domains is the large number of components and relationships that must be reckoned with during the creation of an application. As the percent of components used decreases (compared to the components which may potentially be reused), the utility of the domain decreases.

3.5 DISCUSSIONS OF EXISTING EFFORTS

This section will discuss the following:

- ROSE
- Raytheon Experiment
- CAMP
- McCain Approach
- Prieto-Diaz Approach
- Draco.

3.5.1 Reuse of Software Elements (ROSE)

ROSE [Lubars 1987] is a two-part application development system. The front end is a System Design Assistant that accepts specifications and requirements of the application from the user and, utilizing a library of design schemas, domain types, and domain constraints, produces a set of useful algorithms and type expressions and an abstract design. The back end is a design assistant that combines the abstract algorithms, data types, and representations to generate executable code.

The libraries upon which the front end of ROSE operates are analogous to a library of domain models. The front end uses the user interface to select the appropriate domains that supply the abstract data types and operations, and assists in the development of the abstract design. Two separate libraries are used to do this. A library of domains is referenced through descriptions of domain objects and attributes, and this library in turn references a design component library that provides the abstract design with the help of some user-supplied associated design schemas.

3.5.2 The Raytheon Experiment

A methodology to reuse code in business applications was developed at Raytheon [Lanergan 1979]. While business applications might be considered a rather large domain, the development of the library and classification of the components placed within the library was done using domain analysis. They found that most modules fall into one of three major classes within the business application domain. They also found that most programs could be described by a combination of three logic structures, although as many as 85 logic structures were defined. The common functions and the relationships between them were defined to form the domain model. The abstractions of these functions and their relationships were used to form the three logic structures.

While the Raytheon experiment did not follow a classic domain analysis paradigm, it is similar to the ROSE method discussed above.

3.5.3 The Common Ada Missile Package (CAMP)

Ten tactical missile systems, coded in various languages, were analyzed for common components. The analysts identified existing systems within the application domain, performed a functional decomposition and functional abstraction to break those systems into their abstract component parts, and then developed reusable components that were based on the common abstract components.

3.5.4 The McCain Approach

An approach to domain analysis is discussed by McCain in his paper, "A Software Development Methodology for Reusable Components" [McCain 1985]. He divides domain analysis into four separate sequential processes. First, reusable entities are determined. These entities include objects and related independent operations. Relationships between these operations are then defined. These relationships define search paths for the retrieval of library components.

An approach to domain analysis devised by G. Arango is discussed by Prieto-Diaz. Arango utilizes a similar approach to McCain but spends a great deal more effort first bounding the domain in order to limit the analysis. Neither McCain nor Arango have tested their approaches to date.

3.5.5 The Prieto-Diaz Approach

Ruben Prieto-Diaz [Prieto-Diaz 1987] defines a domain analysis approach which defines pre-domain analysis and post-domain analysis activities as well as the actual domain analysis description. He also breaks his domain analysis down into three separate operations.

The pre-domain analysis activities consist primarily of defining the domain. In addition to defining the scope of the domain, his domain definition includes identifying the source of the information about the domain and tailoring the domain analysis itself.

The post-domain analysis activities consist of the production of the guidelines with which the reusable components produced by the domain analysis are to be used. This step is considered to be part of the domain analysis process by some other researchers.

The three steps involved in performing the domain analysis are: identification of the objects and operations, abstraction of those objects and operations into a domain-specific application independent form, and classification of those objects and operations. The abstraction of the objects and operations includes the definition of the relationships that bind them together. The classification of the objects and operations includes the formal definition of the domain language.

This approach is only in the research phase and has not been tested to date.

3.5.6 Draco

A domain analysis approach which supports reuse of design is that of Draco; the system was devised by Neighbors [Neighbors 1984]. Draco performs a domain analysis that produces both a domain language and a group of tools that support the use of the domain language. The system designer writes a program in the domain language, and the tools, with help from the user, produce an executable version of the application in a conventional language.

The results of a domain analysis and domain design are the domain description, which consists of five tools: a parser, a pretty printer, transformations, software components, and software procedures. The parser consists of the external syntax of the domain and the prefix internal form, which is the actual data that is manipulated by Draco. The pretty printer is a tool that produces the mapping from internal program fragments to the external syntax for the domain. The transformations are source-to-source transformations on the objects and operations of the domain. These transformations represent the rules of exchange between the objects and operations of the domain and are correct regardless of the implementation chosen for the object or operation. The software components specify the semantics of the domain. There is one component for each operation and object in the domain. The components make implementation decisions. Each component consists of one or more refinements that represent different implementations for the object or operation. The software procedures are used when the knowledge for a certain domain-specific transformation is algorithmic in nature.

This system is built in six stages. These stages are:

1) Specify the domain.

2) Perform a domain analysis.

3) Create a domain language.

4) Create the parser and pretty printer.

5) Define the transformations.

6) Define a specific computerized system based on the domain language.

Stages one through three might be considered to all be contained within the definition of a domain analysis, if the standard broader definition is used. A quality domain analysis is crucial for correct operation of Draco, because the tools are directly generated from the domain language that is derived from the domain analysis.

J. G. Rice advocated a system called the Automatic Software Generation System (ASGS) which is similar to Draco [Rice 1981].

3.6 SUMMARY AND CONCLUSIONS

Domain analysis may prove to be an effective tool in providing reuse because of the wide spectrum of knowledge that may be reused. If a domain analysis has been performed on a domain in which a new application is to be developed, the systems analyst has a framework on which to hang the specifications for the new application. This reuse of analysis information is a powerful form of reuse because its effects are felt throughout the development process. The design of the application, the components used in the actual coding of the application, and the relationships between those components may all be provided within the libraries developed in the domain analysis and mapped from the specification written in the domain language. The ability to reuse these designs and components that were developed independently is the strength of domain analysis.

Domain analysis is a relatively new process, and several problems in the process remain unresolved. Not the least of these is the lack of a standard methodology with which domain analysis should be performed. The three most prevalent deficiencies with existing methodologies are the following:

- defining the domain
- acquiring knowledge about the domain
- generating concrete goals and uses for the product of the domain analysis.

No research effort has suggested a technique to determine when a domain analysis is complete or when the knowledge of an idea is fully encapsulated. Nor have any efforts suggested a technique that determines when a domain is oversized and requires division into multiple domains. These problems have not surfaced in the prototypes because all domains have been made sufficiently small. The incompleteness of a domain is more likely to surface as more applications are developed and the more complexity is added to those applications. To date, the existing prototypes have been used for developing only a few applications, and those applications have been relatively simple.

It is widely agreed upon that the knowledge about a domain is costly and difficult to acquire. This problem could be minimized if domain analysis were to become a widely accepted technique, because the domain experts would have more experience. This is being seen with structured design now, as the vast

majority of those software engineers who are knowledgeable in an application domain are also knowledgeable in structured design. As long as relatively few software engineers are familiar with domain analysis, either those with the knowledge of the domain must learn about domain analysis or those with the knowledge of domain analysis must learn about the domain. This transfer of knowledge is both difficult and costly, as the experiences of knowledge engineers developing expert systems show.

It is important to realize that the science of domain analysis is targeted specifically toward the capture of information about a specific application domain, and that, while that information is a viable resource that may be reused in the development of future applications within that domain, the perceived use of the fruits of a domain analysis vary widely. Domain analysis based paradigms have been shown to be capable of providing the following reusable products:

- templates for the systems analysis of the application which result in a design of the application
- a mapping to a library of applicable constructs for the application, including procedural components and objects
- complete CASE tools which guide the systems analysts, systems designers, and programmers through the entire development life-cycle
- a high-level, domain-specific language and a translator to a common executable language (C, Ada or FORTRAN).

Because of this diversity in the potential use of domain analysis, there can be no uniform assessment of the amount of risk or the potential for gain from using domain analysis. The potential benefits and the amount of risk resulting from a given type of domain analysis can best be evaluated for the specific reuse approach being implemented.

The science of domain analysis is a new, relatively unexplored field in which neither the extent of the benefits nor the depth of the pitfalls is fully known. Virtually all stages of the software design life-cycle can be supported by reusable components spawned from the domain analysis. This could include automatically generated applications. Domain analysis has the potential to reduce the effort required for individual applications by a tremendous amount. Yet, questions about the amount of effort required to perform a domain analysis, and the degree of usefulness of the components, tools, and designs available for reuse through domain analysis, raise doubts as to the final net gains of domain analysis.

4.0 DOMAIN-INDEPENDENT APPROACHES

Domain-independent approaches are those approaches that propagate reuse across unrelated application areas. The reusable elements in this category are considered to be general purpose. Most general purpose elements are component based. Therefore the predominant domain-independent approaches deal with producing, using, validating, finding, cataloging, managing, and maintaining general-purpose components. Some examples of these components include linked list managers, sort routines, and math functions. The three major domain-independent approaches identified in this section are:

- libraries/repositories
- commercial components
- a design methodology for producing reusable components.

All of the domain-independent approaches may also be applied to a specific domain. For example, CAMP uses a repository that is domain specific.

4.1 COMPONENT LIBRARIES/REPOSITORIES

Interest in software development cost savings and the potential for reuse of Ada code have given rise to a number of efforts in the U.S. to create software repositories and libraries of software components. Efforts to develop such libraries have varied in sponsorship, management, breadth of focus, and accessibility.

The following topics are examined in this section:

- factors to be considered in the development of software component libraries and repositories
- existing component libraries and U.S. repositories
- lessons learned and recommendations for future developments.

4.1.1 Desirable Characteristics of Software Component Libraries/Repositories

In literature on reuse, and from experience with existing libraries, a few factors stand out as important to the development of software libraries.

- Accessibility of Code. To encourage reuse, code must be easily accessible to both contributors and users. Ease of access includes the knowledge that the repository exists,

the ability to find individual programs or components that fit reuse requirements, and the means to obtain code in a reasonable amount of time at a reasonable cost. The medium in which code is available is also important to the accessibility of the repository, because many developers have a limited hardware configuration. Security restrictions may be a barrier to easy access of code and so must be carefully considered.

- Ease of Reuse. After code has been located which seems to fit the requirements for a particular application, the user must be able to quickly determine what modifications, if any, are required for the code to exactly fit functional specifications. The developer must also be able to spot areas of concern for portability.

- Quality Control. The user should be able to distinguish among different versions of the same software component or program, and also to understand the differences between different implementations of the same application (e.g., two different sort routines). The user should know whether the code has ever been tested and used and on what hardware configuration.

- Management and Acquisition Incentives to Reuse. Management strategies must be developed both to encourage the development of code for a software library, and the use of the resulting code in other efforts. The development of code for the software library may be accomplished by a single contractor, or it may be the result of many contractors' efforts. Contractual requirements or incentives may spur the use of software libraries.

- Documentation. Documentation should be provided that describes how to use the library component. The component documentation should include an abstract of the component, and a description of its interface. Examples of the components' use should also be included. Finally, documentation should enable the user to contact either the code developer or the repository maintainer to report bugs and problems, and reported bugs or limitations should be clearly indicated in the repository.

In the following sections the characteristics of existing sets of reusable components will be weighed against these criteria.

4.1.2 Existing Software Component Libraries/Repositories

In the U.S., efforts to establish collections of reusable components have been undertaken both by the Government and in the commercial sector. Early efforts to build software repositories were not restricted to particular application areas or technologies. More recent work has focused on the development of components for restricted application areas, such as software for missiles and avionics, and software for management information systems.

4.1.2.1 The Ada Software Repository (ASR)

The Ada Software Repository (ASR), established in 1984, is a collection of general information, Ada programs, tools and educational material which is available on the Defense Data Network (DDN), a national electronic network maintained by the DoD. The ASR currently contains more than 1,500 files. The ASR is intended to promote the reuse of Ada programs, tools, and components, and to promote Ada education by providing several working examples of programs in source form for people to study and modify. This repository contains only Ada code and Ada-related information, but is otherwise unrestricted in application focus. The ASR contains programs and components for graphics, communications, database management, mathematical functions and text manipulation, as well as benchmarks, programming tools and metrics. The ASR also contains general information such as the list of validated compilers and the text of DoD directives concerning Ada.

The ASR receives sponsorship and some funding from two organizations: the U.S. Army Information Systems Command and the DoD's Software Technology for Adaptable Reliable Systems (STARS) Joint Program Office. A support contractor handles dissemination of copies of the repository and interface with the public. Even now, only one individual handles the software acquisition, software review, electronic mailing list, newsletter preparation, Master Index preparation, internal and on-line database maintenance and points of contacts.

All material in the ASR is considered in the public domain. It is accessible to users in a number of ways. Direct, on-line access to the ASR is currently available only to users of the DDN. DDN users may electronically transfer any file in the ASR. Users may also obtain copies of the ASR on magnetic tape, floppy disk, and CD ROM at a reasonable cost. Finally, a hardcopy directory of the repository and its contents is available from the ASR support contractor. To promote the availability of its products, the ASR has established links with a number of other organizations in the Ada community, including the Ada Information Clearinghouse and AdaNET. These organizations provide additional distribution points for information concerning the ASR.

The ASR provides descriptions of available programs both via an on-line documentation system which may be easily adapted for use with typical database management systems, and a hardcopy index. A regular newsletter provides information on new releases. At a more detailed level, each piece of software has an associated prologue file and item description file in a standard format, providing information on the version, date, author, environment and any review that has been performed.

Quality control has been the most criticized facet of the ASR's operation. The software is provided "AS IS", without any warranty concerning its validity. The prologue for each software component contains a point of contact for the author of the code. No formal screening is done in most cases before placing items into the ASR, other than checking to ensure that the required prologue is complete. Screening and reliability information on a piece of software may be provided after its release. Each item has one or more comment files, stored with the software, which relate comments received from users. In the case of software upgrades, both old and new versions of software are kept for some time, until reports from users have been received that the new version is reliable and can be trusted to the same level as the old; at that time, the old version may be removed. It is intended that, once a critical mass of software has been received in the ASR and the quality items become well-known, then a separate collection of only the quality items will be established.

Some DoD organizations, including Worldwide Military Command and Control System (WWMCCS), STARS, and the Defense Communications Electronics Command, have already mandated submission of contractor-created software to the ASR. Other software has been submitted by individuals in academia, industry and other government agencies.

4.1.2.2 Computer Software Management Information Center (COSMIC)

The Computer Software Management Information Center (COSMIC), founded in 1966, is operated by the University of Georgia for the National Aeronautics and Space Administration (NASA). COSMIC contains more than 1,000 programs and components in a variety of application areas, implementation languages and environments. COSMIC was started with the goal of technology transition; it was hoped that the results of research and development funded by NASA would benefit U.S. businesses as well, and would maintain public support of NASA efforts. COSMIC is staffed by approximately 15 people from the University of Georgia.

The distribution of programs in the COSMIC inventory is restricted to the continental U.S. for the first year after their receipt. Then, if permission is granted by NASA and the author of the software, distribution is unlimited. COSMIC is not directly accessible via an electronic network; instead, it publishes a catalog of available programs from which interested users may order. The catalog is well indexed by topic and provides a brief description of each program, along with information on its development environment, processing requirements, and its cost. Programs are available in a variety of digital formats. Software programs or components are individually priced based on their size, documentation, and application area. Documentation is unbundled from software and sold for the cost of reproducing it.

Although COSMIC attempts to compile and test all code received, this effort is in fact limited by the hardware environments available to COSMIC personnel. Each author is required to provide test data or benchmarks with which to test the program submitted. COSMIC has been receiving an increasing number of utility programs that are difficult to test. Version control is maintained, and both old and new versions are made available. There are no formal documentation standards, but software has been rejected because of inadequate documentation. COSMIC acts as a buffer between the author and user and provides the first tier of technical support. If that is not sufficient, COSMIC's technicians call the author and attempt to work out problems. With the author's permission his or her name may be provided to a user.

NASA's requirement that software developed for it be made available to COSMIC has not guaranteed submission of code to the Center. COSMIC personnel often have to take the first step in monitoring trade journals, NASA Tech Briefs, and so forth, for news of code that has been developed under NASA contracts, and then contacting the contractors or offices responsible.

4.1.2.3 Common Ada Missile Packages (CAMP)

Common Ada Missile Packages (CAMP) is an Air Force software technology project focusing specifically on software reusability. CAMP was initiated in 1984 and currently contains 452 operational flight software parts in Ada for tactical missiles, as well as a prototype parts engineering system to support parts identification, cataloging, and construction. CAMP differs from the ASR and COSMIC in a number of ways. First, it is narrow in application focus. Second, the development of both its software and the associated tools for its use have been the responsibility of a single contractor. Third, the software in it is considered "militarily critical" and so is subject to limited distribution.

CAMP software is distributed by an Air Force facility, the Data & Analysis Center for Software (DACS). Because the CAMP software is considered "militarily critical", it is subject to export control regulations which make distribution cumbersome. Distribution via electronic network is prohibited. The DACS handles the necessary Government paperwork required to distribute limited distribution software while staying within moderate costs. CAMP also requires that users complete a new form, agreeing that the Government software will not be resold to the Government, or sold as a separate entity in competition with commercial products. This agreement, however, allows CAMP component users to be compensated for time spent modifying CAMP software for use on a Government contract.

The CAMP components are all packages or subprograms usable in a stand-alone fashion. To help the user find the appropriate component for his requirements, CAMP has developed a prototype parts engineering system that enables the user to identify, catalogue, and customize components. This system is currently hosted on a Symbolics 3620 using the ART expert system, but will be implemented in Ada and re-hosted on a DEC computer to increase accessibility.

Because CAMP software was developed by a single contractor over a two year period, the problems of quality control, configuration management, and use of documentation standards are not as great as for large repositories such as the ASR and COSMIC. All CAMP components were tested before release, and the components have been demonstrated in the subsequent development of a real-time embedded system.

As an incentive for reuse, the CAMP software was distributed to more than 125 Government agencies and contractors between September 1985 and May 1988. The CAMP parts were acquired by the Government with unlimited rights. The parts are being considered for a number of other systems, including NASA's Space Station, the Advanced Tactical Fighter, and the Advanced Air-to-Air Missile.

4.1.2.4 AdaNET

AdaNET is a cooperative effort among NASA, the Ada Joint Program Office, the U.S. Department of Commerce, and the University of Houston to provide electronic access to Ada software, tools, information, and education, and to expedite the transfer of technology to industry, small business and academia, as well as to government agencies. The contract to develop AdaNET was awarded in 1987.

One of the first goals of AdaNET will be to link with existing sources of Ada code and information to make such repositories more widely available to the public. Although there is currently no charge for accounts on the AdaNET system, which is accessible via GTE Telenet, it is intended that AdaNET services eventually be self-funded. Later goals of AdaNET include the provision of value-added services and products, including Ada "starter" packages, documentation and tutorials, seminars and workshops, development and management tools, and telephone assistance. It is also planned that AdaNET be a pointer for other sources of information about Ada products and services. AdaNET has participated in major Ada conferences as a vendor in order to raise awareness of its products and its plans for the future. Because AdaNET currently offers no software products of its own, it serves more as a redistribution point for other repositories than as a repository itself.

At present, AdaNET offers access to the ASR products. It is expected that access to software developed by the STARS Foundation contracts will be provided as well as access to the COSMIC software catalog. There is currently no additional screening of software accessible via AdaNET and no version control or configuration management other than that which is performed by the repositories with which AdaNET is linked.

4.1.2.5 The Booch Taxonomy

Another source of reusable components is that proposed by Grady Booch in his recent book Software Components with Ada. In this book Booch advocates developing families of reusable components rather than single components [Booch 1987A]. For one application, a linked list of unbounded length may be appropriate; for another application, space constraints may put a limit on the size of a linked list. In both cases, the interface of the package handling the linked list manipulations should be the same; but the implementations of the linked list would differ in the two cases. The advantage of having identical interfaces is that if the needs of the application change, for example, limits must be placed on the size of the linked lists, one can simply "unplug" the unbounded package and "plug in" the bounded one. Thus families of reusable components are more flexible than components of which there is only one representative of each kind.

Booch recommends classifying components by the following distinctions:

- Bounded - Unbounded
  - A bounded component is one whose size is static. For example, a list implemented as an array is of fixed size.
  - An unbounded component is one whose size is dynamic. For example, a list implemented as a linked list is of variable size.
- Unmanaged - Managed - Controlled
  - An unmanaged component is one that does not provide garbage collection. In this case, any garbage collection is performed automatically by the runtime system.
  - A managed component is one that provides garbage collection for sequential systems.
  - A controlled component is one that provides garbage collection for concurrent systems.
- Noniterator - Iterator
  - An iterator is an operation that does something to all members of a homogeneous class, such as an array. A noniterator component is one that does not have such an operation.

- An iterator component is one that does provide an iterator.

- Sequential - Guarded - Concurrent - Multiple

  - A sequential component is one that is designed to be used in a sequential application.

  - A guarded component is one that preserves the integrity of data in a concurrent environment by means of devices, such as semaphores, that are visible to the user.

  - A concurrent component is one that preserves the integrity of data in a concurrent environment by making access to the protected data sequential.

  - A multiple component is one that preserves the integrity of data in a concurrent environment and allows simultaneous readers but only sequential writers.

Given these distinctions, Booch develops a taxonomy of reusable components. Booch sells a representation of these components through WIZARD Software (see Section 4.2, Commercial Components). GRACE component, distributed by EVB, Inc., also follow this taxonomy. The main disadvantage of the Booch components is that a given development effort is likely to need only one component of a given type. Yet as things stand now, the developer must purchase the entire package.

4.1.2.6 Other Library Systems

The STARS program has investigated many software libraries issues. As a result of that investigation, four prototype library systems were developed;

- Reusable Library Framework (RLF)
- Reusable Ada Packages for Information systems Development (RAPID)
- Rapid Search and Retrieval (RSR) system
- GENeralized Embedded SYstem Specification (GENESYS) tool.

Except for RAPID, each of these systems was developed as a STARS Foundation project, and copies are available to the public by way of the Naval Research Laboratory (NRL).

The primary objective of the RLF was to provide for an intelligently guided search through a library of software components, and more generally, a knowledge based approach to the management of software artifacts that apply to a particular application domain. The basic achievement of the RLF project is the provision, in a demonstrable prototype, of a general framework for the construction of domain-specific libraries of Ada software components.

The main goal of the RAPID Center is to promote software reuse. Plans for the RAPID Center include policy recommendations, administration guidelines, standards, and user guidelines. The RAPID system also contains a library system of reusable components.

The software life-cycle needs to incorporate the role of software reuse in order to reduce development costs and increase software reliability. The U.S. Army Information System Engineering Command recognizes this fact. The SIDPERS-3 RAPID Center is a proof-of-concept that will be applied to the development of SIDPERS-3. The current version will support only software components that are Ada code. Because the benefits of reuse can be achieved early in the life-cycle, future versions should be extended to include not only code, but also design, specifications, and documentation.

RSR employs a three-phase method to increase Ada code accessibility. The first phase establishes a powerful, easy to use, readily accessible, centralized STARS Repository. The second phase makes the same sophisticated repository technology generally available to the software development community so that others can establish their own repositories. The third phase will be the formation of a distributed repository comprising networked repositories resulting from the first two phases.

GENESYS is designed to assist in the assembly of individually tailored Ada Run-Time Support Environments (ARTSEs) from a library of vendor-supplied, third-party or custom-built Run-Time Elements (RTEs). In addition, GENESYS provides an attractive user interface to any library of reusable Ada software components. GENESYS supplements the traditional software development life-cycle in much the same manner as other reuse support tools. GENESYS should pay off over the entire life-cycle through

- productivity increases through reuse
- reduced development risk through multiple instances.

The potential for risk reduction by supporting multiple development paths for the same general class of functionality makes GENESYS a unique software reuse and tailoring tool for Ada development.

4.1.3 Library Set Up

Several different library systems have been described. How does the project manager set up his own library system? What are the problems that need to be solved? Here the primary issues are defined and some approaches are suggested.

Reusable libraries are still in their infancy as a technology. A study that investigated the underlying assumptions of software libraries states, "current operations will not scale up gracefully to handle our assumptions involving large numbers of components with large amounts of information about those components" [Hocking 1988]. Hocking goes on to point out that software solutions to simple problems have rarely scaled up to handle larger problems. If this is the case, a number of different solutions may have to be tried before the best solution is found. Only experimentation and use will shed insight into the fine grain of technical issues that need to be resolved in order for the best library system to emerge.

There are four primary issues related to libraries:

Search:	The mechanism/method that a library employs to facilitate the location of a desired component.
Retrieval:	The means by which a component is transferred from the library to the intended user.
Configuration Management:	The methods/policies used to control component changes in a library.
Administration:	Administrative needs that a staff or librarian will fulfill.

4.1.3.1 Search

After a fair number of components have been collected, the components need to be put into some classification scheme. Classification of components allows the user to begin to search for a component in a "logical" place.

Initially, a directory structure may be sufficient as a means of classifying components, but when a large number of components are collected, this method may become very cumbersome.

Another way of classifying components is through the use of a taxonomies. Taxonomies would provide a depth-first search for a component. Furthermore, taxonomies may be customized with categories that the user is familiar. There are many ways to classify software: by size, by the problem it is solving, by the characteristics of its solution, or by the domain the component belongs to. By selecting a set of attributes and describing all the software in the library in terms of those attributes, one can define a vocabulary to locate the components. This process is called faceted classification. A search strategy based

on faceted classification can easily be implemented with a database. Each facet defines a field in the database structure. A user can then query the database using a facet-based vocabulary.

The Rapid Search and Retrieval (RSR) system appears to have a good solution to this problem. The key to the system is, in essence, its universal symbol table. A unique symbol is assigned to each word. That symbol is constant in all documents. Rather than reducing the degree of indexing, this technique reduces the overhead of storing the original code and documentation. The unique symbol table facilitates indexing, which allows for an effective and efficient way to impose different taxonomies on the same set of software components.

4.1.3.2 Retrieval

Once a potential component has been found the next step is to obtain a copy. Before the library system is put into place, the program manager should decide if he wants to integrate the library system into the project development environment. If a library system is integrated into the development environment, it makes reuse easier. The software engineer can search through the library at his terminal. Obtaining a component may be as simple as copying it from one place to another. A drawback of this approach is that multiple copies of the same code may result. This would be a waste of memory space as well as a configuration nightmare.

If a library is not integrated into the development environment, then for the reuser to obtain a copy of the component, a variety of options exist. The librarian may send it to the reuser on a floppy disk or magnetic tape, or the component may be downloaded. This all depends on what resources are available to the project manager and how he decides to integrate the library.

4.1.3.3 Configuration Management

When setting up a library, one must define who will be allowed to use the library. There are a range of possibilities. One extreme would be to make the library available to the world, a public domain library. The other extreme would be to make the library available only to developers on one project. The first would be overkill and unnecessary; the other would be too limiting and restrict the benefits of reuse. A project manager's library system should at least be available to all the projects for which he is responsible. Another option is to make the library system assessable throughout a particular domain. These options will allow each project to benefit from reuse while keeping the library system to a

manageable size. After the library's domain is defined, a number of library requirements will naturally fall into place, e.g., who is allowed to use it, library staff size, distributed needs, etc.

What happens when a bug is found and a component is modified, or a component is enhanced, or a new implementation to a package specification is developed? Should the users be notified of every change? If so, how? If not, in what manner should they be notified? These are the questions configuration management addresses. Some example configuration management policies are listed below:

Do not store functionally redundant components.

Notify reusers of bugs found in the code.

Supply reusers with solutions to bugs when developed.

Supply reusers with coded solutions, if they have actually used the code.

Do not supply reusers with new components that have been modified to improve functionality. Consider it a new component.

Determine a way to link package specifications with different package bodies.

A library system that is integrated into the software development environment has the advantage of being linked to the reusers. Also an automated tracking system may be implemented. Notices, based on defined configuration management policies, could be sent directly to the reuser. However, the integration will be costly.

A library system that is separate from the development environment needs to set priorities for its policies. The objective is to keep manual work to a minimum, thus keeping administrative cost low.

4.1.3.4 Administration

Obviously, a library system does not just come into existence and maintain itself. A library staff is needed. Their responsibilities are many:

- Complete and test components that come from outside sources.
- Recommend enhancements to components.
- Analyze bug reports and correct problems; distribute change notices to library users.

- Upon acceptance of a software component, enter the component into the classification schemes.
- Track the experience gained from running the library.
- Collect cost data from inside and outside the organization to compare software reuse with conventional software development practices.
- Identify and extract components requested by the user.
- Log a variety of information for tracking purposes, e.g., software component use, search failures, suggestion boxes, and user accounts. The logs are used to determine areas in which the library needs to be improved.

Topic specialists may also be part of the library staff. Their purpose is to be very knowledgeable about the components in the library that are in a specific domain. The topic specialist keeps notes on problems, solutions, and other information about those components that are not represented in the database. Periodically these notes would be published.

It should be noted that reuse will not happen as a result of the implementation of a reuse library. To ensure that reuse is at least attempted, checkpoints need to be inserted in the development process.

4.1.4 Summary and Recommendations

The diversity of goals and approaches in the development of U.S. software repositories has made clear that with each of the considerations outlined above there are benefit tradeoffs.

The ASR, funded and staffed at a low level, places little emphasis on the quality of software submitted. It is a "grass-roots" repository that was initially established without fanfare. Its achievements, however, in terms of quantity of available software, have been immense given the low level of investment in it. ASR has now become so well known that it is being made accessible on other media through a number of second party vendors. The progress of the ASR can probably be attributed to the dedication of a few individuals and the relative lack of bureaucratic and contractual "red-tape" surrounding it.

COSMIC provides a much higher degree of quality control than the ASR. With the additional investment in quality, however, additional time is also needed to make software available. Turnaround time between submission of code to COSMIC and its accessibility to the user via the COSMIC catalog is apt to be far longer than that of the ASR. COSMIC personnel also mentioned that the Center's lack of publicity is a problem; even those within NASA are often unaware of the Center. Finally, the diversity of

application areas, implementation languages, and media represented by the COSMIC collection, even with its well organized catalog, present a challenge to users searching for reusable software.

The CAMP project has had the greatest focus on software development with reuse as an explicit objective, and it has already shown a great deal of promise. CAMP is the only repository that is comprised of components only, and not entire programs, and it is the only one that includes additional software to assist the user in tailoring components for use in various applications. One barrier to reuse of CAMP is its means of dissemination. Contractor users have complained about the length of time required to satisfy government requirements surrounding access to CAMP products, because of their militarily critical status.

Because AdaNET is fairly new, conclusions may be premature. However, if AdaNET's goals include adding value to existing repository products, its problems with configuration management will be even greater than those of the ASR and COSMIC. Ironically, AdaNET may also suffer from being publicized too much; users may expect products and services more quickly than could be reasonably expected of such a large undertaking. AdaNET's publicity underlines the necessity to advertise reasonable goals and a realistic schedule for achieving them.

Repository developers must define short term and long term goals and a realistic schedule of milestones to achieve the goals. The definition of repository goals is especially important if there are a number of sponsors or developers involved. If the need to show that software can be developed for particular applications is greater than the need to have a single set of standard components, the strategy adopted may be similar to that of the ASR. If, however, application areas are well-defined, the CAMP strategy may work best. The following are additional recommendations for the successful development of software repositories.

- Ensure adequate repository management. Support required includes configuration management, quality control, catalog generation, public relations, dissemination of components and documentation, and coordination with sponsors and developers.
- Define quality control standards, including standards for coding, documentation, and testing. These standards should be clearly stated in all literature describing the repository, so that users know the extent to which the repository contents have been screened.
- Assist users in finding the components or programs by providing a well-indexed, comprehensive catalog. Make the catalog and the components available both in hardcopy and electronically. The catalog should include, at a minimum, information on the application, development date and version, development environment and limitations or trouble reports.

- Make repository software as easy to obtain as possible. Sometimes cost is less a barrier than government procedures. The dissemination organization must be able to provide timely response to requests for library components.

- Publicize the repository as widely as possible to software developers, potential users, and government software acquisition organizations. Also, provide a means of obtaining feedback from users.

- Contact other repository developers to obtain specific information on software licensing agreements used, and agreements with government sponsors and contractors regarding the reuse of software. There are still many issues surrounding data rights which are important to understand before adopting a strategy.

The use of government contractual incentives for the reuse of software has not yet been demonstrated completely. Many feel that the availability of high quality products which are conducive to reuse provides adequate incentive to vendors with fixed-price contracts. Contractors will reuse code if there is an obvious cost benefit, and if competition is driven by the reuse philosophy. To achieve high quality products in the shortest time, repositories must have clear goals, be adequately funded and directed, and must be well publicized.

4.2 COMMERCIAL COMPONENTS

Private industry is beginning to supply the software community with reusable components or commercial components. The cost of the components is typically much less the development cost of a comparable component. Also, many manufacturers offer warranties, maintenance contracts, and/or user support. Thus it is prudent to survey the industry to determine whether the available components can be useful. Below is a sample of commercial components that can be purchased from their respective developers. A brief description of the component(s) is provided along with a point of contact.

Ada Components Catalog includes mathematical algorithms, control systems, graphics algorithms, board support packages, business, string processing, sorting algorithms and geometric algorithms. For additional information, contact: John Griffin, Iib Systems, Inc., P.O. Box 18173, Anaheim, CA 92187, (714) 528-6710.

The AdaSoft Components Kit (TACK). Current tools available within TACK include AdaMenus, which provides facilities for creating and displaying several kinds of menus, and AdaWindows, which provides facilities for creating and using windows. This software is available both in source and binary. For more information, contact: Mr. Jerry Horsewood, AdaSoft Inc., 9300 Annapolis Rd., Lanham, MD 20706, (301) 459-4696, adasoft@grebyn.com.

Computer Representatives International, Inc. (CRI), has developed and markets a database management system written in Ada. The DBMS is available as a standalone product and can be embedded in applications written in Ada. For more information, contact: CRI Incorporated, 5333 Betsy Ross Dr., Santa Clara, CA 95054, (408) 980-9898.

Software Technology, Inc, offers an implementation of the Graphical Kernal System (GKS) written in Ada and for use by applications written in Ada. Versions of GKS are available for systems including VAX/VMS, PC/MS-DOS, Macintosh, as well as training in the use and application of GKS. An Ultrix implementation is under development. For more information, contact: Geri Cuthbert, Software Technology, Inc., 1511 Park Avenue, Melbourne, FL 32901, (407) 723-3999.

GRACE (Generic Reusable Ada Components for Engineers) is a library of 275 reusable software components based on commonly used data structures. The only requirement for its use is a validated Ada compiler. For additional information, contact: EVB Software Engineering, Inc., Frederick, MD. (301) 695-6960.

Math Advantage, a library of reusable Ada components, is available in the new 3.0 release. This version is useful for vector and matrix manipulation, as well as signal and image processing. Contact Quantitative Technology Corp. for more information at (503) 626-3081.

Numerical Algorithms Group, Inc. Offers mathematical components in its NAG Ada Library. Package units include basic arithmetic, input/output, numbers, and error trapping. For additional information contact Numerical Algorithms Group, Inc., Downers Grove, IL (312) 971-2337.

The Booch Components (referenced in Section 4.1.2.5), are sold as a set by WIZARD Software. There are 501 packages in this collection, totalling just under 150,000 lines of Ada. The collection includes structures (e.g., stacks, strings, queues, lists, trees), tools (e.g., filters, pipes, sorts, searches and pattern matching packages), and subsystems (components that denote a logical collection of cooperating structures and tools). For additional information, contact Wizard Software, 2171 S. Parfet Court, Lakewood, CO 80227, or call (303) 987-1874.

4.3 A DESIGN METHODOLOGY FOR PRODUCING REUSABLE COMPONENTS (OBJECT-ORIENTED DEVELOPMENT)

There are numerous methods of developing software. Reusable software can be developed using any development method. There is one method however, that seems to facilitate reusable code. That method is called object-oriented development. Object-oriented development tends to produce software that is loosely coupled, and highly cohesive. This method also promotes development at multiple levels of abstraction. These properties coincide with primary characteristics of reusable software. Therefore, object-oriented development is a good choice when trying to produce or consume reusable code.

Object-oriented design is a method of partitioning a software system. It states that partitioning should be based on objects, not functions: each module in the system should be built around one class or object. The term "object-oriented design" has lately been replaced by the term "object-oriented development" because it has been realized that an object-oriented mentality must be present all the way through the software development cycle, not just in the design phase. Accordingly, we will begin with a discussion of object-oriented requirements analysis.

Defenders of object-oriented design contend that an object-oriented library unit is more likely to be reused. This is because object-oriented library units tend to be highly cohesive and loosely coupled to the outside world. A properly designed object-oriented library unit is built around a given type and exports at least a primitive set of operations for that type, and so is highly cohesive. Moreover, a well designed object-oriented library unit has a tightly controlled interface: it exports only the minimum needed by the outside world and hides the rest, and it imports only what it needs from the outside world. This ensures that object-oriented library units will be loosely coupled.

4.3.1 Object-Oriented Requirements Analysis

An object-oriented requirements analysis enhances the traceability between the requirements and the design. Although the designer is free to improve on the view of the world presented by the analyst, an object-oriented requirements analysis will still look a lot more like an object-oriented design than a functionally-oriented requirements analysis will. Since many functions may pertain to a single object and many objects may embody a single function, it is likely to be extremely difficult to achieve a straightforward mapping between functional requirements and object oriented design. The mapping between object-oriented requirements and object-oriented design, however, is likely to be much more direct.

A requirements analysis provides a default framework for a design. The designer may choose to accept this default. In that case, the design and coding phases would consist of nothing more than filling in the gaps left by the requirements analysis. But the designer is not obligated to do this. The designer may instead combine objects identified in the requirements analysis phase into larger objects, or he/she may break them apart into smaller objects. The situation is no different from functional requirements: there may be a one-to-one, a many-to-one, or a one-to-many relationship between a functional requirement and the subprogram(s) that implements it. The essential thing is that the requirement be satisfied in some way.

It has been claimed that it is more difficult to derive an object-oriented design from a functionally oriented requirements analysis (such as a structured analysis) than from an object-oriented requirements analysis. Why should this be? To answer this question we must recognize that the software development process from design through coding is a continual process of making explicit what was only implicit in the original requirements. This is true in the first place of derived requirements. Derived requirements, as the name implies, are implied by the original system requirements. The same is true of the actual code: the data structures and processes that constitute the completed system constitute the "cash value" of its requirements. The full implications of the original system requirements, therefore, cannot be understood until the coding phase has been completed.

Because the ratio of requirements to lines of code is almost always very small, each requirement typically summarizes an enormous number of implementation features. Any two complete sets of system requirements will of necessity summarize all the features of the implemented system. The difference between them will not be in the features summarized but in the way the features are grouped together. Thus, each requirements analysis will organize the system from a unique point of view.

Now an object-oriented design has very specific needs: objects must model the real world, and operations must be subordinate to objects. What this means is that an object-oriented design will need implementation features that are grouped together in a certain way. What is to insure that, out of all the possible groupings, these features will be grouped together in the way an object-oriented design needs them? Not unless we introduce object-oriented thinking from the start can we maximize the likelihood that serviceable objects and operations will come out of the requirements analysis phase. The entities identified by a functionally oriented requirements analysis method are likely to be the wrong collections of implementation features--too general, too specific, or simply irrelevant to the needs of an object-oriented design. This is because functionally oriented requirements analysis methods are really looking for different things than object-oriented requirements analysis, so that, even if both completely express a system's implementation features, it is unlikely that the way in which those features are organized will be as useful for an object-oriented design.

Functionally oriented requirements analysis and object-oriented requirements analysis also organize the system requirements differently at a higher level. The former organize them by function, breaking down general operations into more specific ones; the latter organizes the requirements by object, breaking down higher-level objects into lower-level ones. Thus, not only are the objects and operations identified in an object-oriented requirements analysis more likely to be serviceable for an object-oriented design, but the structure of a system is more likely to fit in with an object-oriented design if that design is based on an object-oriented requirements analysis.

Object-oriented requirements analysis uses basically the same set of techniques as object-oriented design. The main difference between them is that requirements analysis focuses on required objects, attributes, and operations, whereas design fills in the details.

4.3.2 Object-Oriented Design

An obvious strategy for reuse of high-level routines is to structure them with reusable modules. A contemporary method for doing this is object-oriented design. Object-oriented design restricts an application to the manipulation of objects. Objects tend to be general purpose, and are constructed with other lower level objects that are themselves reusable.

Given the need for object-oriented requirements analysis as a front end to object-oriented design, let us first of all attempt to clarify just what the object-oriented designer is looking for. The goal of object-oriented design is to identify the following:

- classes
- subsystems
- objects
- states
- attributes
- operations.

The basic concept of object-oriented development is the notion of a class. A class is an entity that is characterized by a set of attributes and/or a set of operations. If the structure of the class is visible, then it is characterized by the data structure in terms of which it is defined, the implicit operations that pertain to that data structure, and any further operations that are explicitly provided in the library unit defining the

class. If the structure of the class is not visible, then it is characterized only by the operations that are explicitly provided in the library unit defining the class, as well as any operations (such as equality) that may still be implicitly available.

Other concepts are based on the notion of a class:

- A subsystem is a collection of classes. For example, in an aircraft simulator all classes pertaining to the aircraft components may be grouped together into the same subsystem.
- An attribute is a property of a class. For example, "location" is an attribute, because it is a property of the class "aircraft." An attribute is an instance of a class in its own right, which may in turn be characterized by other attributes. For example, "location" may be an instance of the class "coordinates," which is characterized by the attributes "latitude" and "longitude."
- An object is an instance of a class other then an attribute. An example of an object is "Air Force One," which is an instance of the class "aircraft."
- A state is an instance of all the attributes of an object. Suppose the class "aircraft" were characterized by the attributes "altitude," "air speed," "heading," and "location." Then the state of an aircraft would be the particular values of these attributes, for example, "1000 feet," "100 knots," "180 degrees," and "40 degrees north by 160 degrees west."
- An operation is a process that either changes or provides information about the state of an object. For example, the operation "accelerate" would change the state of an aircraft; the operation "current speed" would provide information about the state of the aircraft.

Operations can be divided into constructors, selectors, and iterators: Constructors change the state of an object; selectors provide information about the state of an object; iterators are performed on homogeneous complex objects (e.g. arrays, linked lists) and either change or provide information about the state of every element in the object. Thus, iterators are either constructive or selective.

The most important of these concepts are class, attribute, and operation. It is important to keep in mind the conceptual differences between them. The benefits of object-oriented design can be fully realized only if it is indeed classes that are encapsulated in software modules, not attributes or operations disguised as classes. Conceptual clarity is important. Attributes are properties; operations happen over time; classes are things that have attributes and perform operations.

4.3.3 Object-Oriented Coding

How can these entities be mapped onto Ada library units? A subsystem has no Ada equivalent. A class is typically an abstract data type defined in a package (for example, a linked list package), and an

object is typically a variable, constant, or task of that type. Sometimes it is advisable to represent a class by a generic package--what Booch calls an "abstract-state machine" package [Booch 1987B]. In this instance, an object of the class would be an instantiation of the generic package, and so would be a package. Attributes are represented by generic formal types. States are represented by the possible values of those types, expressed in Ada by literals or aggregates. Operations would be implemented by functions, procedures, and entries. All Ada functions are selectors, and iterators are represented by generic procedures. TABLE 2 provides Ada equivalents to object-oriented entities.

The principal Ada design tool is the package. An object-oriented package will be built around a single type and will define the type in terms of the operations that can be performed on objects of the type. One can distinguish five kinds of object-oriented packages:

- Open, in which the central type declaration is visible
- Private, in which the central type is a private type
- Limited, in which the central type is limited private
- Opaque, in which the central type is limited private and implemented as an access type, the full type declaration of whose designated subtype is located in the package body [ARM 3.8.1 (3)]
- Closed, in which the central type is declared in the package body.

Orthogonal to this taxonomy is another distinction based on what the package exports. From this point of view, packages can be classified into the following types:

- Class exporting, in which the central type is declared in the visible part of the package specification
- Object exporting, in which an object of the central type is declared in the visible part of the package specification
- Operation exporting, in which at least one operation of the central type is declared in the visible part of the package specification. It should be pointed out that a package can export an operation in two ways--explicitly or implicitly. When a type is declared, certain operations are implicitly declared for that type. These are comprised of the following [ARM 3.3.3 (2)]:
  - Basic operations [ARM 3.3.3 (3 - 7)]
  - Predefined operators [ARM 4.5 (2, 6)]
  - Enumeration Literals [ARM 3.5.1 (3)]

TABLE 2. ADA EQUIVALENTS TO OBJECT-ORIENTED ENTITIES

Object-Oriented Entity	Ada Equivalent
Subsystem	< none >
Class	Type Generic package
Object	Variable Constant Task Package
Attribute	Generic Formal Type
State	Literal Aggregate
Operation	
Constructor	Procedure Entry
Selector	Function Procedure Entry
Iterator Constructive Selective	Generic Procedure

- Derived subprograms [ARM 3.4 (11 - 14)].

Explicitly declared operations for a type are subprograms each of which meets the following conditions [ARM 7.4.2 (4)]:

- At least one of its parameters and/or its result is of the type in question.
- The operation is declared in the specification of the same package in which the type is declared.

An open package exports all of its implicitly declared operations. A private package exports only those implicitly declared operations which do not depend on the knowledge of the full type declaration [ARM 7.4.2 (1 - 3)]. A limited package exports the same implicitly declared operations as a private type with the exception of the basic operation of assignment and the predefined equality operator [ARM 7.4.2 (1, 3)].

It seems clear that the first of these types is characteristic of open, private, limited, and opaque packages, because the central type declaration is visible. It also seems clear that the second of these types is another variant of the open package, because in order for an object to be visible in a package specification, its type must also be visible. Finally, the third type pertains to open, private, limited, opaque, and closed packages. Combining this distinction with the one above, we can produce the following classification shown in TABLE 3. Examples of each package can be found in Appendix A.

TABLE 3. PACKAGE TYPES

	class exporting	object exporting	operation exporting
open	x	x	x
private	x		x
limited	x		x
opaque	x		x
closed			x

4.3.4 Risks of Object-Oriented Development

Any advocate of object-oriented development is likely to run into a certain amount of resistance. In any profession there is a certain reluctance to change. People have built successful careers on certain procedures with which they are comfortable. Adopting the adage, "If it works, don't fix it," people may see no need to do things any differently than the way they have always been done.

Object-oriented development is still a very new technology. It is best accomplished with an object-oriented requirements analysis as a front end. This part of object-oriented development is even newer and thus relatively untried. There are few analysts around with genuine experience in object-oriented requirements analysis. Hence, there may be a substantial learning curve for any group of software engineers intending to use object orientation for the first time.

A related problem is that software engineers have for years been trained in functional methods. This, coupled with the fact that functional decomposition is probably easier to perform with less training than object-oriented development, means that designers on a project are likely to be more comfortable with functional than with object-oriented approaches.

Another difficulty a program manager may encounter when considering object-oriented development is fitting it into the software life-cycle. It was argued above that the most suitable front end to object-oriented design is object-oriented requirements analysis. Often, however, a program manager simply inherits a functionally specified system in which the major components are also broken down functionally. It may be difficult to fit object-oriented development into a life-cycle which has already progressed along a different path.

All of the above cautions center around the newness of object-oriented programming. But if novelty were always a reason for not doing something, no progress would ever be made in any field. In order to be persuaded to embrace a new technology, managers must be convinced that the long-term benefits are substantial enough to outweigh the short-term liabilities. This is probably the case with object-oriented development. Object-oriented systems, when done in the right way, exhibit a coherent design, are loosely coupled internally, are clear and easy to maintain, are modularized in a meaningful way, are easy to debug, and are likely to have reusable components. An investment of the time and resources to become comfortable with object orientation will probably pay for itself in the long run.

5.0 DOMAIN-SPECIFIC APPROACHES

Reusable elements whose reuse is restricted to a defined application area are called domain-specific approaches. The payoff when using these approaches is usually higher than when using domain-independent strategies. This is because the percentage of reusable elements in a given application can be greater in the domain-specific approaches.

The component approaches to reuse that are defined in the previous section also apply to domain-specific environments. The only difference is that the reusable components are not as general-purpose as in the domain-independent case.

Another style of reuse approaches is adaptive reuse. Adaptive approaches try to reuse the framework of the software. This includes the design as well as the top layers of an application. Component reuse structures the application around the components. Conversely, adaptive reuse maintains a fixed structure and plugs in, deletes, or alters components to change the functionality of the system.

In most cases, domain-specific components can be used to feed the adaptive approaches. This is because the adaptive approaches advocate a top-down software design where component methods are generally bottom-up. Combining the two types of approaches provides the potential for automating the software coding process.

The domain-specific reuse methods discussed in this paper are:

- generic architectures
- constructors.

5.1 GENERIC ARCHITECTURES

Generic architectures are a form of adaptive reuse. Generic architectures are upper-level structures that contain reusable software and design components along with common interface mechanisms. They provide the high-level design and code as well an application-specific set of components to be adapted, swapped, or deleted. Thus a new application could experience huge savings through the reuse of most of the design and code from a generic architecture.

Generic architectures can be thought of as a form of standardization at the design level. One could make an analogy with computer hardware. An IBM-AT compatible computer will always have a chassis, a 286 mother-board with expansion slots, and a power supply. These common elements define the generic architecture of the system. Hardware components can be adapted, swapped, or deleted from one specific configuration to create a second configuration that satisfies totally different needs. For example, a desktop publishing system requires a high resolution graphics card driving a large screen black and white monitor, a hard drive, a floppy drive, a controller, I/O ports, a mouse, and a postscript laser printer. An image processing workstation would be configured with a frame grabber, CCD camera, high resolution color monitor and graphics card, a dry ink plotter, I/O ports, a track ball, a larger hard drive, a floppy drive, a controller, and extended memory. These applications have dissimilar functionalities and require drastically different hardware support. Yet both configurations are based on a generic architecture of the IBM-AT compatible computer.

There are several criteria that govern when to use generic architectures. First, there must be several applications to be developed that are similar in nature such that a generic architecture can be applied. Also these applications should be funded from the same source. This is because the up-front cost of creating a generic architecture is high. Thus several reuses are required to offset the initial costs and save long-term dollars. Furthermore, it should be anticipated that the technology in the application domain be stable over the life of the program. Rapidly growing technologies could foster some radical changes in the requirements of the systems. Changes in requirements could shatter a generic architecture.

The level of reuse for generic architectures is high. This is due to the sheer volume of elements that are reusable. The design, the upper-level code, and the lower-level components are all potentially reusable. Also the data structures, communication protocols, and other software interfaces are standardized across applications. This translates into savings in future developments as well as system-wide maintenance. Training costs are reduced because the man-machine interface will be fairly consistent among the applications spawned from the same generic architecture.

A key feature of generic architectures is that all the functionality of the system is known before much code is written. The opposite is true of a bottom-up component reuse strategy. Also the adaptive nature of generic architectures lends itself to an iterative style of development where one would code, test, and provide feedback. Thus generic architectures facilitate the development of rapid prototypes. These prototypes are also useful to validate the generic architecture.

For generic architectures to be feasible, the application domain needs to be well understood, and the potential applications in the domain need to be identified. Consequently, a quality domain analysis

needs to be completed prior to the design. This domain analysis must be performed by an experts in the application domain and by individuals who are experienced in domain analysis.

The cost for doing a domain analysis is high and cannot be budgeted. The cost cannot be budgeted because it is difficult to determine when a domain analysis is complete, and an incomplete or marginal quality domain analysis may produce an invalid architecture. Note that the feasibility of using the generic architecture is not fully determined until the domain analysis is complete.

Generic architectures have a moderate amount of flexibility in the handling of requirements changes. The domain analysis identifies or tries to anticipate the requirements for future applications. Such requirements pose little threat to the generic architecture because the architecture is designed to accommodate all of the current and future requirements for the systems. However those requirements that were not forecasted by the domain analysis may compromise the architecture. This danger is prevalent in domains that were ill conceived, or where the technology is experiencing rapid growth. Using our hardware example, the IBM-AT architecture cannot adapt to the new micro channel technology and therefore cannot be used.

Once the decision is made to use generic architectures, it must be enforced. This is because most of the expense of developing a generic architecture is incurred near the beginning of the project. These battles are best waged prior to spending all the funds on a detailed domain analysis.

5.2 OTHER DOMAIN SPECIFIC METHODS

Other reuse methods that are unique to the realm of domain-specific approaches are discussed:

- constructors
- structural models

5.2.1 Constructors

A component constructor is a software system that facilitates the development of application software by producing components based on user requirements [McNicholl 1988]. There are three parts to a constructor.

a domain-specific library of skeleton components called meta-parts

an interface to the user that allows the user to build applications through a collection of components

a set of rules and tools that instantiate a meta-part.

Requirements are input by the user. The constructor analyzes this information, searches the library for the appropriate meta-part, and then instantiates the meta-parts thereby generating a compilable component. For a schematic representation of a constructor, see Figure 5.

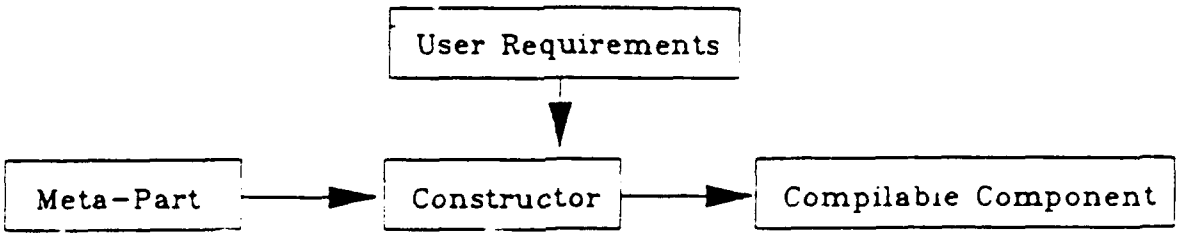


FIGURE 1. THE CONSTRUCTOR AND RELATED PARTS

Each item in Figure 5 will be explained in detail:

- meta-part

  The meta-part is the general form of a component and is either the complex Ada generic or the schematic part. A complex generic part may require data types, operators, and subprograms for instantiation. It may also require a complex defaulting scheme. Simple generic parts require only a small number of data types for instantiation. Schematic parts consist of a "blueprint" for construction, and a set of construction rules for building a specific instance of the part. Ada generics are used whenever possible. But not all possible templates can be captured as Ada generics. Specifically, generics cannot handle situations in which only the structure, not the content, remains the same (cf. Section 5.1.1.1). In these instances, schematic parts are used in place of Ada generics.

- user requirements

  The requirements are entered based on questions elicited by the constructor's user interface.

- constructor

  The constructor analyzes the requirement data and extends the meta-part to generate the code for the component.

- component

  The resulting component is an instance of the component's general form.

The creation of a constructor requires a close interaction between the constructor developer and the component developer. The constructor developer creates the user interface and the constructor, and the component developer generates the meta-parts.

The CAMP project determined that constructors would be beneficial for efficient implementation of nine of their components and developed nine component constructors. The following discussion describes constructors based on CAMP's experiences.

The meta-part for CAMP's finite state machine is a schematic part, because the variable number of states and transitions are difficult to capture in generic units [McNicholl 1988]. The meta-part for the autopilot is made up of complex generic parts. The constructor assists the user in creating a correct instantiation. For the Kalman Filter Constructor a combination of generics and schematics is used. The user's options determine the implementation.

CAMP's experiences with the constructor show its feasibility within narrow domains, and its use does increase productivity. In its current state, it is not very portable. Because the constructor is closely tied to the software component, any significant changes to the component will require modifications in the constructor. Thus, constructor maintenance can be a costly concern.

A software developer working in the domain of the constructor, can build a large portion of the desired application with the constructor. This can result in a large cost savings in the design, coding, and maintenance phases of a software development.

5.2.2 Structural Models

There are two premises of structural models. One is that there will exist a solution that can be described as a series of recurring patterns. The other is that these patterns can be generalized to other applications within a broader domain. This translates to two levels of reuse. The first is the reuse of the recurring patterns within the application. The second is the reuse of those patterns between applications.

Structural models operate predominantly at the design and code levels of the software life-cycle. At the design level, a structural model will define a set of recurring patterns, and a grouping strategy with

which these patterns apply. Code for the structural model would consist of a set of software templates. These templates are skeleton packages that require a user to fill in specific type definitions and formal procedural parameters. If a generic is used in lieu of a template, a large number of generic formal parameters and subprograms would need to be instantiated. This would add an unnecessary level of complexity to the code and obscure the essence of the recurring patterns. This is discussed further in Appendix F.

The first step in developing a structural model, is a domain analysis. As with the generic architecture, the domain analysis bounds the domain, defines the requirements for software within the domain, and identifies commonality between elements in the domain. Initially, in a structural model, the domain is defined as the application itself. The domain analysis consists of defining requirements for the given application, and identifying any recurring patterns within the application. Requirements are typically defined as part of the software life-cycle. Thus, the net effort required in a domain analysis for structural models is reduced to finding the recurring patterns.

Once the recurring patterns are established, software templates based on these patterns are generated. A framework is then designed to interface the instances of the patterns with the application.

Because the domain analysis is minimal for a structural model, the up front cost, hence the risk is less than most other reuse schemes. Conversely, there is no guarantee that the patterns will translate well into future applications.

A generic architecture is appropriate when the applications to be developed are planned. In this situation it is preferable to define the requirements and the design for all the applications simultaneously. If the number and function of future applications is not known, then a structural model should be considered. For instance, if software is being developed for an application that is to be implemented in a number of disjoint organizations (e.g., across DoD agencies or services), then the in-depth domain analysis needed to develop a generic architecture would be virtually impossible due to the amount of coordination necessary, and the unknowns of various implementation schedules, budgets, and staff assignments. In this case, use of structural models would be more practical and less risky. The focus in this case would be the current application with the consideration of future applications.

The idea of structural models is a very powerful concept because it really achieves reuse on two levels--within an application and between applications. Its power comes from the detection of patterns internal to any application of a given type and iterating instances of that pattern. It is a domain-specific concept and requires some knowledge of the intended application domain to be used.

6.0 COST/BENEFIT ANALYSIS FOR SOFTWARE REUSE

Four cost/benefit issues are addressed in this section:

- the economics of reuse
- software cost models with a reuse component
- estimating the cost/benefits of reuse

6.1 ECONOMICS OF REUSE

There are two areas that are relevant to the economics of reuse. These areas are the production of reuse, and the consumption of reuse. In this section the factors that influence cost will be delineated for both reuse production and consumption.

Production of Reuse

At the beginning of most software efforts, a decision must be made as to whether software units are to be made reusable. This determination can be aided by looking at the factors influencing the costs.

The first factor to consider is the initial investment in reuse. For components this is the cost to produce a reusable component versus the cost to produce a non-reusable functionally equivalent component, and/or the start-up cost for a component library. With adaptive reuse the initial costs can include a domain analysis, a generic architecture, a structural model, a constructor, a template generator, and/or supporting tools.

Another major factor is the number of times a reusable element is to be used. It costs more to produce a reusable element than an equivalent non-reusable one. This is because the development of reusable code requires more time in the planning and coding stages. Thus, it is important to anticipate the number of potential reuses for a given software element. By reusing a software module, one is, in effect, distributing the developed cost for that software among the various applications. Therefore, if a software element is used three times, then the cost of developing that element can be divided by three, with each third being charged to each usage. The number of reuses will determine the pay-back in the development phase.

The extra time expended to create reusable code is offset somewhat by the amount of debug time saved in the testing phase; it is easier to localize bugs when working with reusable modules since one can zero in on small blocks of code. Another factor enhanced by reusability is maintainability. Reusable components are easy to maintain because they are easy to localize. They also can be easily extracted for testing. Furthermore, once a reusable software element is validated, all instances of that element are valid. The maintenance aspect alone could justify the creation of reusable software.

Consumption of Reuse

Including a reusable unit in an application is the consumption of reuse. When addressing the cost effectiveness of reuse consumption, some additional cost factors need to be examined.

One factor that should be considered is the extra cost of identifying, finding, and evaluating reusable components. Finding a component to fit your needs is not a trivial task. Hence this task may be expensive.

There is also the cost of the learning curve for using a reusable unit. The learning curve cost refers to the extra time and effort required for the proper usage of the unit. Good documentation will reduce this cost significantly.

Finally, there is the cost of working within reuse-imposed design constraints. Reusable modules can restrict the design of the software they support. The greater the number of reusable components used in a given application, the greater the number of design constraints imposed by the reused software. Such constraints affect the structure, operation, and function of the application. At times these constraints are beneficial to the development effort. At other times an extra software layer needs to be developed to interface the reusable component with the application.

6.2 SOFTWARE COST MODELS WITH A REUSE COMPONENT

This section describes the methods used to account for reuse in current software cost models. The following topics are discussed:

- accounting for reusable components in new cost estimates
- estimating the development of reusable code
- deficiencies in the current cost models.

Selecting a Cost Model

The term "cost model" is a conventional one, but it should be recognized that cost models estimate personnel effort and schedule duration for software project activities and life-cycle phases. There are few discriminating factors when determining which model is the most appropriate within a unique environment. Most automated models will run on an IBM PC (or compatible) and estimate operational support costs in addition to development. (Appendices B - E provide an overview of the contractual arrangements, costs, hardware requirements, and developer points-of-contact for several well-known models.)

Of first concern to managers when determining the cost for a new project is which model will give the best ballpark figure. Few comprehensive studies have been performed that demonstrate the differences between software cost estimation models in view of their ability to provide reasonable estimates. Accuracy claims made by the developer are difficult to substantiate. In addition, differing perceptions exist on how to estimate software cost for new technologies such as Ada. The following discussion will focus on the empirical studies performed to date that address model accuracy. The discussion is divided into two areas: 1) predicting personnel effort and 2) predicting schedule duration.

Personnel Effort

For estimating software development costs, two studies have demonstrated that different models have better expected accuracy for different classes of applications [FERENS 1989] [IITRI 1989]. An IITRI study targeted eight completed Ada projects and compared the effort predicted by six cost models to the actual project resources expended by their respective developers. Projects targeted in the test case study consisted of three different types of applications: command and control (4 projects), tools/environment (3 projects) and avionics (1 project). TABLE 4 summarizes the model performances that were based on a comparison of estimated to actual effort. An analysis of the results based on application type demonstrated that model performance varies for different types of applications.

TABLE 4. TEST CASE STUDY RESULTS [IITRI 1989]: PERSONNEL EFFORT

Evaluation Criteria	Model	Performance (Within 30%)	Range
Overall Accuracy of Effort	SoftCost-Ada	4 out of 7	0% to 13%
	SASET	4 out of 8	-29% to 29%
	SPQR/20	3 out of 8	-22% to 19%
	COSTMODL	2 out of 6	-25% to - 1%
	PRICE S	0 out of 8	
	SYSTEM-3	0 out of 8	
Overall Consistency of Effort	SYSTEM-3	5 out of 8	-14% to 28%
	PRICE S	5 out of 8	-26% to 22%
	SoftCost-Ada	4 out of 7	-13% to - 2%
	COSTMODL	3 out of 6	-29% to 10%
	SASET	4 out of 8	-15% to 27%
	SPQR/20	3 out of 8	-20% to 21%
Model Accuracy on Command & Control Applications	SASET	3 out of 4	- 7% to 29%
	SPQR/20	3 out of 4	-22% to 19%
	SoftCost-Ada	2 out of 4	6% to 13%
	COSTMODL	2 out of 4	-25% to - 1%
	PRICE S	0 out of 4	
	SYSTEM-3	0 out of 4	
Model Consistency on Command & Control Applications	PRICE S	4 out of 4	-26% to 0%
	SASET	3 out of 4	-15% to 1%
	SYSTEM-3	3 out of 4	-14% to 26%
	SPQR/20	3 out of 4	-20% to 21%
	SoftCost-Ada	2 out of 4	- 8% to - 2%
	COSTMODL	2 out of 4	- 1% to 10%
Model Accuracy on Tools/Environment Applications	SoftCost-Ada	2 out of 2	0% to 2%
	SASET	1 out of 3	-29%
	COSTMODL	0 out of 1	
	PRICE-S	0 out of 3	
	SYSTEM-3	0 out of 3	
	SPQR/20	0 out of 3	
Model Consistency on Tools/Environment Applications	SoftCost-Ada	2 out of 2	-13% to -11%
	PRICE S	1 out of 3	22%
	SYSTEM-3	1 out of 3	28%
	COSTMODL	0 out of 1	
	SASET	0 out of 3	
	SPQR/20	0 out of 3	

* COSTMODL is an automated implementation of Ada COCOMO, IOC version.

Results were evaluated for consistency by comparing the project's actual effort to the estimated effort after a computed mean value was applied to each model estimate. An analysis of model efforts for consistency was performed to establish if results were consistently high or consistently low, eliminating differences between the perspectives of the person deriving the inputs to each model and the model developer. This process involved the following steps:

1. A percentage of actual effort to model effort was calculated.
2. The two extremes were discarded to achieve a truer sampling of percentages.
3. A mean value of the remaining percentages was computed and applied to the given model's estimates.
4. The relative error for each project was recalculated using the adjusted efforts.

The results of this process when applied to each model are illustrated in Tables 4 through 6.

Models were also applied using nominal (average) values for input ratings while providing actual project values for model input parameters that must be estimated early in the life-cycle, and for which there is no associated average value. The nominal inputs reflect the level of knowledge about a new development prior to contract award. An evaluation of the results of the study showed model performances varied with differing amounts of project information. Some had surprisingly good results with minimum information. A summary of the test case study results for nominal runs is provided in Table 6.

The results are somewhat indicative of the databases that were used to develop and validate cost models (See TABLE 5). SoftCost-Ada has a database that is comprised of a large number of commercial projects. PRICE S, SYSTEM-3, and SASET appear to be based on DoD software development environments.

Because of the size of the database and the nature of the programs targeted in the test case study, it is difficult to make positive conclusions with regard to model accuracy. One can, however, identify trends that may be supported in future studies. Outside validation studies should ideally be used to supplement a model user's own analysis.

TABLE 5. OVERVIEW OF DATA USED TO DEVELOP/CALIBRATE COST MODELS

Cost Model	Data
Ada COCOMO	Calibrated using two completed TRW Ada projects which had been developed using full DoD software acquisition standards [IITRI 1989].
SoftCost-Ada	Approximately 30 software projects developed by 12 different organizations within five aerospace firms during the period spanning 1982 through 1987 [IITRI 1989].
PRICE S	Software projects at RCA Morristown Surface Radar Division, including airborne, ship, and ground radar projects [IITRI 1987].
SASET	Martin Marietta software development data consisting of more than 300 completed projects and some selected Navy data [IITRI 1989].
SYSTEM-3	Data points on 50 management information systems and command and control systems [IITRI 1987].

TABLE 6. TEST CASE STUDY RESULTS FOR NOMINAL RUNS [IITRI 1989]: PERSONNEL EFFORT

Evaluation Criteria	Model	Performance (Within 30%)	Range
Overall Accuracy of Effort	SASET	4 out of 8	-24% to 29%
	SYSTEM-3	3 out of 8	-17% to 28%
	COSTMODL	2 out of 6	-25% to -24%
	SoftCost-Ada	2 out of 7	-27% to 14%
	PRICE-S	2 out of 8	-14% to -8%
	SPQR/20	1 out of 8	-27%
Overall Consistency of Effort	COSTMODL	3 out of 6	-23% to 30%
	SoftCost-Ada	3 out of 7	0% to 28%
	SASET	3 out of 8	-24% to 7%
	SYSTEM-3	3 out of 8	-26% to 13%
	SPQR/20	1 out of 8	-14%
	PRICE-S	1 out of 8	-29%
Model Accuracy on Command & Control Applications	SASET	3 out of 4	- 7% to 29%
	SYSTEM-3	3 out of 4	-17% to 28%
	COSTMODL	2 out of 4	-25% to -24%
	PRICE S	2 out of 4	-14% to - 8%
	SoftCost-Ada	2 out of 4	-27% to 14%
	SPQR/20	1 out of 4	-27%
Model Consistency on Command & Control Applications	SASET	3 out of 4	-24% to 7%
	SoftCost-Ada	2 out of 4	12% to 28%
	SYSTEM-3	2 out of 4	-11% to 13%
	COSTMODL	2 out of 4	-23% to 30%
	SPQR/20	1 out of 4	-14%
	PRICE-S	1 out of 4	-29%
Model Accuracy on Tools/Environment Applications	SASET	1 out of 3	-24%
	COSTMODL	0 out of 1	
	SoftCost-Ada	0 out of 2	
	PRICE-S	0 out of 3	
	SYSTEM-3	0 out of 3	
	SPQR/20	0 out of 3	
Model Consistency on Tools/Environment Applications	COSTMODL	0 out of 1	
	SoftCost-Ada	0 out of 2	
	SASET	0 out of 3	
	PRICE S	0 out of 3	
	SYSTEM-3	0 out of 3	
	SPQR/20	0 out of 3	

* COSTMODL is an automated implementation of Ada COCOMO, IOC version.

Schedule Duration

For assessing the expected accuracy of software scheduling techniques, especially for the cost model scheduling algorithms, two studies are noted. The Blalock study (Air Force Institute of Technology Thesis: 1988), which focussed on five cost models, showed that COCOMO was the least accurate of the five and had an error of greater than 50%. The other four models: PRICE S, SYSTEM-3, SPQR/20, and SOFTCOST-R were accurate within 20% of the actual schedule. There was one project targeted in the study; therefore, general conclusions could not be drawn about the models studied [FERENS 1989].

A subsequent study compared schedule duration for eight completed Ada projects to the durations that were estimated by the six models studied. Table 7 shows results based upon a comparison of estimated to actual schedule duration. Nominal run results for schedule duration are provided in Table 8. It is interesting to note that the estimates for scheduled duration correlated more closely to the actual schedule in the majority of cases when a minimum set of data was used as input to the models.

6.2.1 Accounting for Reusable Components In a New Cost Estimate

Software cost models typically take the form of a set of equations which relate size, effort, and calendar time, and hence allow the prediction of effort and time given size as an input parameter. The effort and time predictions are then modified by a number of additional parameters which reflect conditions specific to the project and developing organization. In a recent review of six cost estimation models [IITRI 1989], all of them require the size and language of reusable components that are to be incorporated into a new product. What follows will be based on the two models whose underlying equations are nonproprietary, Ada COCOMO and the Navy's Software Architecture, Sizing, and Estimating Tool (SASET). In these models reused components are accounted for in the base estimate for software size. The number of instructions of new code to be developed and number of instructions that are to be adapted are combined into an estimate for the "equivalent delivered source instructions." The following discussions of the COCOMO Reuse Model and SASET's Direct Input mode for software sizing illustrate how these models account for reusable components in new developments.

TABLE 7. TEST CASE STUDY RESULTS [IITRI 1989]: SCHEDULE DURATION

Model	Performance (Within 30%)	Range
SYSTEM-3	4 of 8	-27% to - 7%
PRICE S	3 of 8	3% to 18%
SASET	3 of 8	-24% to 6%
SPQR/20	3 of 8	-16% to 26%
COSTMODL	2 of 6	-28% to -25%
SoftCost-Ada	2 of 7	-26% to - 4%

After Application of the Means:

Model	Applied Mean	Performance (Within 30%)	Range
SYSTEM-3	1.38	5 of 8	0% to 28%
PRICE S	.69	5 of 8	-29% to 28%
SPQR/20	.85	5 of 8	-29% to 29%
SASET	1.63	4 of 8	-30% to 23%
SoftCost-Ada	.90	3 of 7	-13% to 24%
COSTMODL	1.93	1 of 6	-11%

* COSTMODL is an automated implementation of Ada COCOMO, IOC version.

TABLE 8. TEST CASE STUDY NOMINAL RUN RESULTS [IITRI 1989]: SCHEDULE DURATION

Model	Performance (Within 30%)	Range
SPQR/20	6 of 8	-23% to 28%
PRICE S	4 of 8	-26% to 21%
SoftCost-Ada	2 of 7	- 6% to 5%
SYSTEM-3	2 of 8	-23% to 5%
COSTMODL	1 of 6	-26%

After Application of the Means:

Model	Applied Mean	Performance (Within 30%)	Range
SPQR/20	.94	6 of 8	-28% to 20%
PRICE S	.97	5 of 8	-28% to 29%
SoftCost-Ada	1.89	2 of 7	0% to 14%
SYSTEM-3	2.05	3 of 8	-25% to 19%
COSTMODL	2.1	2 of 6	-27% to -23%

* COSTMODL is an automated implementation of Ada COCOMO, IOC version.

6.2.1.1 COCOMO Reuse Model

The basis of the adaptation equations used in COCOMO is that reused code is not counted in the same way as newly developed software. Reuse of existing code may require additional effort in the following ways [Boehm 1981]:

- redesigning adapted software to meet the objective of the new product
- reworking portions of the source code to accommodate changes in the new product's environment (hardware, operating system, etc.)
- integrating the adapted software into the new product environment.

The COCOMO Reuse Model uses the following equations to determine equivalent d elivered source instructions (EDSI):

$$EDSI = (ADSI) [.4 (DM) + .3 (CM) + .3 (IM)] / 100$$

where

EDSI	=	Equivalent delivered source instructions
ADSI	=	Number of adapted or reused instructions
DM	=	Percentage of adapted software design modified
CM	=	Percentage of adapted code modified
IM	=	Percentage of integration required for modified software as compared to the normal amount of integration and testing effort required for software of comparable size.

The coefficients (Design: 40%, Code: 30%, and Integration and Test: 30%) are determined from the average amount of effort devoted to each corresponding phase of the life-cycle. An installation whose phase distributions are considerably different might consider an alternate formula -- for example, for small embedded-mode jobs where less effort is spent in integration and testing:

$$EDSI = (ADSI) [.4 (DM) + .4 (CM) + .2 (IM)] / 100$$

The EDSI value is added to the number of new source instructions to be developed. The combined size value is then used in the nominal estimating COCOMO equations to predict effort and schedule.

6.2.1.2 SASET Method for Calculating Equivalent New HOL

The equivalent new high order language line of code value is computed for each software function before the sizes are aggregated to represent the total size of the software that is being estimated. To calculate "equivalent new higher order language" lines of code, the analyst must account for the condition of the code. Given a lines of code estimate for a software function, the analyst must determine what percent of the code is new, modified, or rehosted. The code conditions are briefly defined below [Silver 1988]:

> **New Code:** This constitutes software code that is to be developed from scratch. Software requirements must be determined, a design established, the design must be coded and units tested. Regardless of the software type, 100 % of this LOC value is used in the new higher order language equivalent calculations.
>
> **Modified Code:** This constitutes software code which is already partially complete and which can be utilized in the software program under consideration. Generally, modified software at the very least needs to be retested, and some redesign and recoding efforts are required. The new higher order language equivalent calculations use 73 % of the modified LOC value.
>
> **Rehosted Code:** This consists of completed and tested software code which is to be transferred from one computer system to another. Generally, the code requires no requirements definition, little or no design definition, and only partial testing. The new higher order language equivalent calculations use 10 % of the rehosted LOC value.

The following example illustrates how equivalent new higher order language lines of code are calculated to account for reuse [Silver 1988]:

> The condition of 9,900 lines of code estimated as the size of a software function is distributed as follows:

Condition	Fraction of Total	Lines of Code
New	1/3 (33.3 %)	3,300
Modified	1/3 (33.3 %)	3,300
Rehosted	1/3 (33.3 %)	3,300
	100 %	9,900

The condition of the code is considered in the computation for equivalent DSI as follows:

Code Size				Equivalent DSI
3,300 (New)	*	100 %	=	3,300
3,300 (Modified)	*	73 %	=	2,409
3,300 (Rehosted)	*	10 %	=	330
		Total HOL Equivalent:		6,039

6.2.2 Estimating the Development of Reusable Code

Only three of the six models (PRICE S, Ada COCOMO, and SoftCost-Ada) reviewed in the above-mentioned Ada costing study [IITRI 1989] took into account the issue of developing reusable software. Although two of the approaches are proprietary, PRICE S and SoftCost-Ada, the following paragraphs provide an overview on how model developers view this issue.

6.2.2.1 PRICE S

PRICE S differentiates between requirements to produce reusable software at the system level and at the module or component level [Park 1989]. At the component level, estimators can view requirements to produce reusable code as either

- complications to the development process, or
- an increase in the application's difficulty.

The first view is appropriate when the developer uses time, rather than adding new people, to meet reusability requirements. The second view is appropriate when both resources and time will be used to achieve reusability. These views increment the complexity and hence the cost of the affected code rather than distribute the increased effort over the entire product. System-level requirements for reusability, on the other hand, affect the design, documentation, and testing of the full product [Park 1989].

6.2.2.2 ADA COCOMO

Development of reusable code is accounted for in the Degree of Reuse (RUSE) parameter. TABLE 9 provides the RUSE cost driver ratings and associated effort multipliers. With this input, the estimator enters the degree of reusability for which the software is being built. The ratings indicate that the development of reusable software will increase cost anywhere from 10% (reuse within a single mission) to as much as 50% (reuse in any application).

TABLE 9. ADA COCOMO DEGREE OF REUSE PARAMETER

Rating	Rating Description	Multiplier
Nominal	Not for Reuse Elsewhere	1.0
High	Reuse Within Single Mission	1.10
Very High	Reuse Across Single Product	1.30
Extra High	Reuse in Any Application	1.50

6.2.2.3 SOFTCOST-ADA

SoftCost-Ada differs from the other models in that development of reusable components in Ada is viewed differently than development of reusable components in other languages. Its input parameter, Reuse Costs, specifies how the technical and managerial costs associated with reuse are handled. Rating descriptions are provided in TABLE 10 [RCI 1989]. The main philosophy behind this parameter is that Ada has specific features, generics, which have been included in the language to make developing reusable components easier. Further, once the developer becomes more proficient in the language, reusable software will be even easier to develop.

TABLE 10. SOFTCOST-ADA REUSE COST RATINGS

Rating	Rating Description
Low	Limited packaging for future reuse
Nominal	< 10% of software packaged for future reuse
High	< 20% of software packaged for future reuse
Very High	> 20% of software packaged for future reuse

6.2.3 Deficiencies In Software Cost Models

While current cost models are including the basic reuse of code into their algorithms, they do not account for the more complex issues of reuse. For example, to incorporate a reusable component into an application, the designer may have to do an extensive search of several component libraries, evaluate the candidate components, understand how to use the components, and integrate them into the application. SASET does not account for this potentially costly process. And COCOMO only factors in the integration cost. No cost model anticipates a domain analysis, or the development of a generic architecture,nor do they consider the expense of CASE tools or constructors. Clearly there are deficiencies in current costing models. These tools can be useful, however, in the cost estimation of non-reusable code. Such an estimate would provide a baseline for comparison in the monitoring of the efforts of reuse on a development. In other words, a developer would use a costing tool to estimate the development cost of the system, ignoring reuse. The developer would then track actual expenses (that include reuse) and compare them with the estimated cost. Thus, the developer can monitor the economic effects reuse had on the project.

6.3 ESTIMATING THE COST/BENEFITS OF REUSE

There are two methods to evaluate the economics of reuse across the software life-cycle:

- factor adjustments to cost models
- empirical estimation.

6.3.1 Factor Adjustments to Cost Models

While there are no explicit metrics for reuse, there are software cost models that estimate a normal software development. These cost models can provide an initial estimate. This estimate can then be adjusted by those costs that are incurred by a specific reuse approach. This adjustment should include both the anticipated additional costs, summarized in Table 11, and the areas of potential cost savings, summarized in Table 12.

The major driver to reuse elements during software development is economic. However, for any economics gain to be realized additional costs will undoubtedly be incurred. For elements to be developed for reuse they will be more general purpose, less application-specific, than in a standard development. This may result in increased documentation, more accommodating design, and code which stresses a simplified flow of control. Achieving these attributes will require more effort, therefore more money. If previously developed components are to be used, a source of components must be located and the available elements studied to identify candidate components. These components must be adequately understood to be fitted into the system currently under development. Understanding the specific candidate components, as well as understanding the reuse of components will not be efficient or optimized. Other factors which may add to the cost of reuse include designing with external design constraints (either designing reusable components or designing a system to incorporate reusable components), implementing a component library, performing a domain analysis, and acquiring support tools. Not all of these additional costs will be experienced in every instance of reuse; however, they must be considered when estimating the cost of system development when reuse is involved.

For reuse to make economic sense, areas of cost savings must exceed these additional costs. One area of potential savings is in system development. Reusing documentation, design, or code may cost less than developing the components. There are additional areas with even greater potential savings. The validation process for reused software should be significantly less expensive. If a machine interface is reused, training costs should be significantly less. The greatest potential for savings is typically the area of greatest expense; for software intensive systems this is the maintenance activity. Maintenance activities are anticipated to be less expensive due to increased familiarly with reused software components by postdeployment support personnel and by the anticipated increase in reliability. Furthermore, the maintenance phase benefits when modules are loosely coupled and highly cohesive.

6.3.2 Empirical Estimation

Another method of evaluating the cost/benefits of reuse is through an empirical investigation. To employ this method one must collect data from other developments that used similar reuse approaches. By comparing the cost data from those other projects and evaluating their respective scopes, one could empirically derive a cost estimate. The accuracy of this approach is dependent upon how much data is collected and how similar the current development is to those about which the data was collected.

TABLE 11. ANTICIPATED ADDITIONAL COSTS.

ANTICIPATED ADDITIONAL COSTS
Additional labor required to develop reusable modules
Cost of obtaining a reusable component
Learning curve
Design constraints imposed by reusable units
Startup cost for a component library
Domain Analysis
Support tools

TABLE 12. AREAS OF POTENTIAL COST SAVINGS.

AREAS OF POTENTIAL COST SAVINGS	
Development	Validation
Training	
Maintenance	

7.0 REUSE METRICS

A quantitative evaluation of reusability can be valuable in the selection of components for use in an application or for a component's acceptance in a repository. Unfortunately, there exists no single metric that can provide a comprehensive assessment of reusability. There are some metrics, however, that can provide a first approximation of a component's reuse potential by measuring selected software engineering attributes that affect reusability. Examples of attributes that affect reusability include complexity, independence, modularity, simplicity, and data bindings. Researchers and tool vendors have identified hundreds of metrics that apply to these attributes; we will mention only a few to provide some insight into the current state of the practice.

Examples of metrics relevant to software complexity include number of statements per software module, number of subprogram calls per module, number of logical paths through a program, number of levels of control in a program, and so on. Independence metrics can be based on the numbers of accesses to I/O types and packages, system dependent services, compiler dependent services, and tasks. Examples of modularity metrics include those that measure information hiding and the degree of coupling between modules: use of private and limited private types; proportion of operators, objects and types in the module bodies or the private part of package specifications; proportion of blocks which do not contain bodies of packages, tasks, procedures, or functions; use of variable declarations in package specifications; and others.

Fortunately for software developers and evaluators, a number of automated tools are already available for the specific purpose of measuring software attributes such as those listed above. Some of these tools are being used or considered for use by repository managers in evaluating software for insertion into repositories. For instance, ADAMAT and LOGISCOPE, described below, have been used by staff at RAPID to determine the characteristics of modules inserted into RAPID's library.

The remainder of this section will briefly describe some popular tools and collections of metrics that can, in a limited fashion, determine the reusability of a software unit.

RADC Software Quality Attributes Worksheets

The RADC worksheet method is a non-language specific technique based on the manual application of generalized worksheets. The worksheets have been automated as part of QUEF. To effectively apply the worksheets, several early steps must be taken, specifically:

1) Identification of quality goals

2) Tailoring of the questionnaires to adequately reflect the project under assessment and the defined goals

3) Tailoring of the equations for factor scores to reflect the defined goals.

The RADC worksheet method is defined as a hierarchy of factors (13), criteria (29), metrics (73), and metric elements (>300). Questionnaire worksheets are tailored to the project being assessed and then completed. Yes, No, and Numeric answers are transcribed to worksheets where affirmative answers are equated to a value of one and negative answers are equated to a value of zero. Numeric responses are already in the zero-to-one range by the nature of the question. Scores are then aggregated up the hierarchy by averaging related metric elements into a metric and related metrics into a criterion. Criteria scores are then used in tailored equations to determine the factor score. The worksheets should be applied throughout the development life-cycle requirements specification through testing and delivery.

QUEF

QUEF is an automated tool being developed by Software Productivity Solutions, Melbourne, Florida. This tool is expected to be completed by mid-1990. The tool is an automation of the RADC worksheet method. However, significant emphasis is being placed on the development of an extremely friendly user interface. Another important difference will be the development of an Ada parser. The parser only analyzes code with the intent to complete the questionnaires. No analysis specific to the Ada language is performed.

ADAMAT

ADAMAT is a language specific automated tool developed by Dynamics Research Corporation. The ADAMAT tool operates by examining compilable Ada source code. The technique used by the tool is the counting of significant language features that are considered to promote or detract from the quality of the product. These counts are the metric elements. Metric element scores are shown as a ratio of the number of opportunities to comply with the preferred quality practice versus the number of actual compliances. The metric scores are then aggregated to a criterion level and then to a factor level. The factors evaluated by the tool are reliability, portability and maintainability. Six criteria are evaluated: anomaly management, independence, modularity, self-descriptiveness, simplicity, and system clarity. Criteria scores are derived from 153 metric values. The tool provides the capability to tailor the metrics gathered and to tailor the aggregation process; that is, the user has the ability to selectively omit metric elements

and metrics. Weights can also be set to give greater importance to one metric over another or one criterion over another in the score calculations. Results can be viewed at any level in the hierarchy, or reports can be triggered by user-specified thresholds. Using thresholds, the user would indicate minimal acceptable scores and a report would be generated only if the scores were below the threshold.

LOGISCOPE

LOGISCOPE is an automated and mostly language-specific tool. It operates by analyzing module source code and producing graphs (kiviat diagrams, control graphs, and call graphs and histograms) to provide information about that module. The LOGISCOPE tool was developed by Verilog in Toulouse, France, and is used and marketed by AMS Software. More than 40 high-level languages can be analyzed, including assembly, COBOL, Ada, C, FORTRAN, and Pascal.

The intent of the LOGISCOPE tool is to assess the complexity, efficiency, and structural integrity of the module by investigating the number of paths, the level of required "decision making" through those paths, the overall size and the depth of the calling hierarchy, and the structure and readability/understandability of the code. Approximately 22 metrics are taken to make this assessment and used in a standard metric, criterion, factor hierarchy. LOGISCOPE analysis can begin as early as development and applied periodically throughout the life-cycle. Information is available from a static and a dynamic analyzer. The static analyzer provides measures specific to syntax, text elements, logical structure and architecture levels. The dynamic analyzer measures path coverage by inserting proves into the source code through the use of CASE tools.

Static Analyzer

Kiviat Diagrams are used to identify whether the metrics are within acceptable ranges. The diagram is actually a plot of metrics radially along a set of spokes. The minimum and maximum values are shown as concentric circles through which these spokes pass. If the corresponding points are outside the inner circle and within the outer circle, then the measure is acceptable.

Control graphs plot the flow of control through the module. That is, from a starting point at the left of the diagram, an arrow directs control to the next point (such as a call or decision) and shows the paths and looping possible through the module. From these diagrams the developer can assess the structure of the module -- whether it is nicely structured with minimal looping, backtracking, and decision making, or whether it is excessively complex or poorly designed.

Dynamic Analyzer

The LOGISCOPE dynamic analyzer also measures the unit and integrated testing coverage through control graphs. Special control graphs can be generated to identify the location of paths that have not been covered through testing and the number of lines in that path. This allows modification of the test plan to increase test coverage before final testing and delivery.

Call graphs give a pictorial view of the system architecture. A call graph is a hierarchical graph of the calling sequence from the main program to the lowest level routines. Other outputs include metrics histograms (e.g., the number of statements in each module), quality factor histograms (a combination of metrics compared to standards), and test coverage histograms (e.g., percent of coverage for each module).

Other Analytical Methods

Several studies have been performed that attempt to simply define what metrics are important or, with respect to Ada, how language features affect some reusability factors. Some examples of these types of studies are listed below:

- An analysis of the physical properties of software (size) as they affect reusability and design guidelines to enhance reusability [HESS 1987]
- A description of the metric developed during the foundation phase of Army WWMCCS Information System (AWIS) [DELANEY 1988]
- Methods for goal setting, data collection and analysis for complexity, quality and cost [BASILI 1983].

Other studies attempt to define metrics applicable to reuse. These methods are primarily non-automated and most require source code analysis. This combination often results in the method being impractical for application to any but the smallest of projects. Examples of these studies are listed below:

- Measures of Ada complexity through an extension of the McCabe's Cyclomatic Complexity Metric [TAUSON-CONTE 1988]
- Analysis of complexity with respect to understandability, testability, and maintainability through an examination of the relationship between program slices and module cohesion [OTT 1989]
- Ada reusability as measured through an analysis of data bindings between modules [BASILI]
- Defect density as a measure of reliability and maintainability [VALETT 1989].

8.0 SUMMARY AND CONCLUSIONS

Most software engineers today practice some form of reuse. This reuse, however, is restricted to the individual developer. What this report has described are state-of-the-art methods to formalize reuse. Formalization will extend the benefits of reuse to the project or system level.

Most current reuse methods concentrate on reusing code. These methods can usually extrapolate a certain amount of reuse to other elements in the life-cycle. A generic architecture approach, for example, could reuse requirements, design, code, test procedures, documentation and training. Component reuse could reuse code, verification, and validation. Thus, when evaluating the costs and benefits of reuse methods, it is important to consider the entire software life-cycle.

One interesting side-effect of reuse is maintainability. Most of the software engineering factors that promote reusability also apply to maintainability. There is a difference, however, in the relative importance of each factor. For example, the presentation factors are much more critical for maintainability than they are for reusability. It is likely, nevertheless, that writing or incorporating reusable code will produce a more maintainable product. This implies a considerable cost savings in the maintenance phase of the software life-cycle.

TABLE 13 offers general guidance in selecting reuse strategies.

TABLE 13. SELECTION OF REUSE STRATEGIES.

	Production	Consumption
Generic Architectures	• Developer must be able to identify/define several similar applications. • All applications should be funded through the same source. • Anticipated domain in which these applications fall should not revolve around a rapidly growing technology.	Should always be used on second through Nth application for which the architecture was designed.
Structural Models	• Developer should have a requirement for several similar applications without a clear definition of the applications.	Should be used on all related applications where the patterns apply.
Constructors	• Developer should be able to identify many applications for the desired components. • Project must be large enough to absorb the overhead of creating/maintaining a component library. • Constructors are only feasible on long term projects. • Related components must be similar enough that they can be derived from the same generic package or blueprint. • Related components must differ enough to make the overhead of building the constructor worthwhile.	
Domain Specific Libraries/Repositories	• Function includes creating, managing, and populating with reusable components. • Strategy is appropriate for large programs that are servicing multiple applications in a tight domain.	Many factors need to be considered: library quality, library access, components cost, retrieval methods used in finding the appropriate components, level of documentation, test routines for the components, examples of their use.
Domain Independent Libraries/Repositories	• Strategy is useful when servicing applications in multiple domains.	Many factors need to be considered: library quality, library access, components cost, retrieval methods used in finding the appropriate components, level of documentation, test routines for the components, examples of their use.

TABLE 13. SELECTION OF REUSE STRATEGIES (CONT.).

	Production	Consumption
Commercial Components	N/A	• Always a good idea to survey market and evaluate the applicable components. • Most should be fairly efficient and reliable. • Tend to be much cheaper than writing the components from scratch. • Consideration should be given to the cost of integrating these components into the design. Commercial components are likely to impose design constraints on the application and may require a special interface to be developed. • Could also be costly to understand the function/use of the component. • Factors also apply to public domain components.
Object Oriented Development	• Always worth consideration because it can augment any other reuse method. • Can also be used as the sole reuse method in a project. • Appropriate in both large and small applications.	

BIBLIOGRAPHY

[ARM] ANSI/MIL-STD-1815A. Department of Defense, Ada Joint Program Office. Reference Manual for the Ada Programming Language. Washington: Government Printing Office, 1983.

[Bailey 1989] Bailey, S., Laird, J., Falacara, G., Angevine, M. "GENESYS: Embedded Software Tailorability." Proceedings of the 7th National Conference on Ada Technology, March 1989, pp 13-24.

[Basili 1988] Basili, V.R. "Towards a Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment." College Park: Institute for Advanced Computer Studies, December 1988.

[Boehm 1981] Boehm, Barry. Software Engineering Economics. Englewood Cliffs: Prentice Hall, 1981.

[Booch 1987A] Booch, Grady. Software Components with Ada: Structures, Tools, and Subsystems. Menlo Park: Benjamin/Cummings, 1987.

[Booch 1987B] Booch, Grady. Software Engineering with Ada, 2nd Ed. Menlo Park: Benjamin/Cummings, 1987.

[Brown 1988] Brown, Gerald R. and Quanrud, Richard B. "The Generic Architecture Approach to Reusable Software." Proceedings of the Sixth National Conference on Ada Technology, March 1988, pp. 390-394.

[Bunch 1988] Bunch, J. "Rapid Search and Retrieval of Reusable Components." STARS Foundations Workshop, November 1988.

[Ferens 1989] Daniel V. Ferens, Defense System Software Project Management, Air Force Institute of Technology, Draft edition: 11 August 1989.

[Freeman 1987] Freeman, Peter. Software Reusability. Washington: Computer Society Press, 1987.

[Gargaro 1988] Gargaro, Anthony. "Analysis of the Impact of the Ada Runtime Environment on Software Reuse." Final Technical Report to Center for Software Engineering, CECOM, December 1988.

[Guerrieri 1988] Guerrieri, E. "Searching for Reusable Software Components with the RAPID Center Library System." Proceedings of the 6th National Conference on Ada Technology, March 1988, pp 395-405.

[Hocking 1988] Hocking, D. E. "The Next Level." Proceedings of the 6th National Conference on Ada Technology, March 1988, pp 407-410.

[Horowitz 1984] Horowitz, E. and Munson, J. "An Expansive View of Reusable Software." Transactions on Software Engineering, Vol. SE-10, September 1984.

[IITRI 1987] IIT Research Institute, U. S. Army Cost and Economic Analysis Center (USACEAC) Software Cost Model Research Paper, September 1987.

[IITRI 1989] IIT Research Institute. Test Case Study: Estimating the Cost of Ada Software Development. April 1989.

[Kaiser 1987] Kaiser, Gail E. and Garlan, David. "Melding Software Systems from Reusable Building Blocks." IEEE Software, July 1987: 17 - 24.

[Lanergan 1979] Lanergan, R. and Poynton, B. "Reusable Code: The Application Development Technique of the Future." Proceedings of the IBM SHARE/GUIDE Software Symposium, IBM, Monterey, CA, October 1979.

[Lee 1988A] Lee, Kenneth J., Rissman, Michael S., D'Ippolito, Richard, Plinta, Charles, and Van Scoy, Roger. "An OOD Paradigm for Flight Simulators." 2nd Ed. Pittsburgh: Software Engineering Institute, September 1988.

[Lee 1988B] Lee, Kenneth, Plinta, Chuck, and Rissman, Mike. "Application of Domain Specific Software Architectures." Pittsburgh: Software Engineering Institute, December 1988.

[Levy 1989] Levy, P., Ripken, K. "Experience in Constructing Ada Programs form Non-Trivial Reusable Modules." Proceedings of the Ada-Europe International Conference, Stockholm, May 1989, pp 100-112.

[Lin 1988] Lin, Dar-Biau. "A Knowledge-Structure of a Reusing Software Component in LIL." Proceedings of the Sixth National Conference on Ada Technology. March 1988, pp. 377-380

[Lubars 1987] Lubars, M. "Wide-Spectrum Support for Software Reusability." Proceedings of the Workshop on Software Reusability and Maintainability, National Institute of Software Quality and Productivity, October 1987.

[McCain 1985] McCain, R. "A Software Development Methodology for Reusable Components." Proceedings of the 1985 Hawaii International Conference on Systems Science, January 1985.

[McNicholl 1988] McNicholl, D. G. et al. "Common Ada Missile Package - Phase 2." Air Force Armament Laboratory, Eglin AFB, Florida, 1988.

[National 1989] The National Institute for Software Quality and Productivity. National Conference: Software Reusability, July 1989.

[Neighbors 1984] Neighbors, J. "The Draco Approach to Constructing Software from Reusable Components." Transactions on Software Engineering, Vol. SE-10, September 1984.

[Park 1989] Park, Robert E., "Ada Estimating - A PRICE S Profile." January 1989.

[Prieto-Diaz 1987] Prieto-Diaz, R. "Domain Analysis for Reusability." Proceedings of COMPSAC '87, 1987.

[RCI 1989] Reifer Consultants, Inc. Softcost-Ada User Guide Software Version 2.0. January 1989.

[Randall 1988] Randall, William D. Jr. Software Reusability: A Decision Tree Model, 1 June 1988.

[Rice 1981] Rice, J.G. Build Program Technique: A Practical Approach for the Development of Automatic Software Generation Systems. New York: Wiley, 1981.

[Ross 1986] Ross, Donald L. "Classifying Ada Packages." Ada Letters, Vol. 6, No. 4, July/August 1986.

[Silver 1988] Silver, Aaron, et. al. SASET User's Guide, July 1988.

[Solderitsch 1989] Solderitsch, J. J., Wallnau, K.C., Thalhamer, J. A. "Constructing Domain-Specific Ada Reuse Libraries." Proceedings of the 7th Annual Conference on Ada Technology, March 1989, pp 419-433.

[Tracz 1988] Tracz, Will. Software Reuse: Emerging Technology. Washington: Computer Society Press, 1988.

[US Army 1989] U.S. Army Institute for Research in Management Information, Communications, and Computer Science. Proceedings: Ada Reuse and Metrics Workshop, June 1989.

APPENDICES

APPENDIX A. EXAMPLES OF PACKAGE TYPES

A.1 Example of Open Package

This example is a dynamic array manager. All arrays in Ada are static. A package must be written to handle variable length arrays. The reason this package was implemented as an open package was to allow component selection and slices on objects of the type. The package specification is as follows:

```
GENERIC
  TYPE elements       IS PRIVATE;
  TYPE ranges         IS RANGE <>;
  TYPE static_ranges  IS RANGE <>;
  TYPE static_arrays  IS array(static_ranges RANGE <>) of elements;
  WITH FUNCTION  index_of (element : IN     static_arrays)
                                     RETURN ranges;
  WITH FUNCTION  array_of (index   : IN     ranges)
                                     RETURN static_arrays;
  null_element          : IN     elements;
  maximum_array         : IN     ranges;
  maximum_index_length  : IN     ranges;
PACKAGE dynamic_array_manager IS

  TYPE dynamic_arrays IS array(1 .. maximum_array +
                                    maximum_index_length) of elements;

  SUBTYPE lengths     IS ranges RANGE 0 .. maximum_array;
  SUBTYPE indices     IS ranges RANGE 1 .. maximum_array;
  SUBTYPE counts      IS ranges RANGE 1 .. maximum_array + 1;



  overflow       : EXCEPTION;
  out_of_bounds  : EXCEPTION;
  index_overflow : EXCEPTION;


--CONVERSIONS

  FUNCTION  static_array_of  (dynamic_array : IN     dynamic_arrays)
                                            RETURN static_arrays;
  -- Converts a dynamic to a static array.

  FUNCTION  dynamic_array_of (static_array  : IN     static_arrays)
                                            RETURN dynamic_arrays;
    -- Converts a static to a dynamic array.
    -- Raises index_overflow if array is not large enough to hold length value.

  FUNCTION  length_of        (dynamic_array : IN     dynamic_arrays)
```

```
                                                RETURN lengths;
    -- Returns the current length of a array.

  FUNCTION  null_array                          RETURN dynamic_arrays;
    -- Returns a null dynamic array.


--SEARCHING

  FUNCTION previous_index_of (element  : IN     elements;
                              within   : IN     dynamic_arrays;
                              before   : IN     counts)
                                        RETURN lengths;
    -- Finds the index of the previous element specified.
    -- If element is not found, returns 0.
    -- Raises out_of_bounds if specified index is longer than array.

  FUNCTION next_index_of     (element  : IN     elements;
                              within   : IN     dynamic_arrays;
                              after    : IN     lengths := lengths'first)
                                        RETURN lengths;
    -- Finds the index of the next element specified.
    -- If default value of after is taken, searches from beginning of array.
    -- If element is not found, returns 0.
    -- Raises out_of_bounds if specified index is longer than array.

  FUNCTION previous_index_of (subarray : IN     static_arrays;
                              within   : IN     dynamic_arrays;
                              before   : IN     counts)
                                        RETURN lengths;
    -- Finds the index at the end of the previous array specified.
    -- If array is not found, returns 0.
    -- Raises out_of_bounds if specified index is longer than array.


  FUNCTION next_index_of     (subarray : IN     static_arrays;
                              within   : IN     dynamic_arrays;
                              after    : IN     lengths := lengths'first)
                                        RETURN lengths;
    -- Finds the index at the beginning of the next array specified.
    -- If default value of after is taken, searches from beginning of array.
    -- If array is not found, returns 0.
    -- Raises out_of_bounds if specified index is longer than array.


--DYNAMIC OPERATIONS

  FUNCTION  "&"          (left           : IN     dynamic_arrays;
                          right          : IN     dynamic_arrays)
                                        RETURN dynamic_arrays;
    -- Concatenates two arrays together.
    -- This operation must be used in place of predefined "&" in order to
    -- set the length of the new array correctly.
```

```
    -- Raises overflow if resultant array exceeds maximum_array.
    -- Raises index_overflow if array is not large enough to hold length value.


  PROCEDURE append      (dynamic_array : IN     dynamic_arrays;
                         to            : IN OUT dynamic_arrays);
    -- Appends one array to the end of another.
    -- Raises overflow if resultant array exceeds maximum_array.
    -- Raises index_overflow if array is not large enough to hold length value.


  PROCEDURE insert      (dynamic_array : IN     dynamic_arrays;
                         into          : IN OUT dynamic_arrays;
                         starting_at   : IN     counts);
    -- Inserts one array into the middle of another.
    -- Raises overflow if resultant array exceeds maximum_array.
    -- Raises out_of_bounds if specified index is longer than array.
    -- Raises index_overflow if array is not large enough to hold length value.


  PROCEDURE remove      (dynamic_array :    OUT dynamic_arrays;
                         from          : IN OUT dynamic_arrays;
                         starting_at   : IN     indices;
                         stopping_at   : IN     indices);
    -- Removes a subarray from a array.
    -- The array from which the subarray is removed is changed.
    -- Raises out_of_bounds if either specified index is longer than array.
    -- Raises index_overflow if array is not large enough to hold length value.


  PROCEDURE foreshorten (dynamic_array : IN OUT dynamic_arrays;
                         to            : IN     counts);
    -- Foreshortens a array from the beginning.
    -- Raises out_of_bounds if specified index is longer than array.
    -- Raises index_overflow if array is not large enough to hold length value.


  PROCEDURE truncate    (dynamic_array : IN OUT dynamic_arrays;
                         to            : IN     lengths);
    -- Truncates a array from the end.
    -- Raises out_of_bounds if specified index is longer than array.
    -- Raises index_overflow if array is not large enough to hold length value.


END dynamic_array_manager;
```

There are several things to notice about this example:

- The package is built around type Dynamic_Arrays. It is implemented as an open type This means that the user has access to its structure.

- Several of the generic formal parameters are needed to define the class: Elements, Ranges, Maximum_Array, and Maximum_Index_Length.
- Types Static_Ranges and Static_Arrays are needed to allow conversions between static and dynamic arrays.
- The other generic parameters provide values and operations needed by class.
- The subprograms in the visible part of the package specification are operations exported by the class.
- Subtypes Lengths, Indices, and Counts are needed by several of these subprograms.
- The three exceptions are further declarations needed by a user of the package.

A.2 Example of Private Package

This example is a date managing package. The package specification is as follows:

```
PACKAGE date_manager IS

  TYPE dates           IS  PRIVATE;

  maximum_year          : CONSTANT := 10_000;
  maximum_days_per_year : CONSTANT := 366;

  TYPE years           IS  RANGE -maximum_year ..  maximum_year;
    -- Positive years are CE years.  Negative years are BCE years.
    -- There is no year 0.
  TYPE months          IS (january  , february , march    , april    ,
                           may      , june     , july     , august   ,
                           september, october  , november , december );
  TYPE days            IS  RANGE 1 ..  31;
  TYPE numbers_of_days IS  RANGE -maximum_year * maximum_days_per_year ..
                                  maximum_year * maximum_days_per_year;

  null_date     : CONSTANT dates;

  date_overflow : EXCEPTION;
  day_overflow  : EXCEPTION;
  invalid_date  : EXCEPTION;


--CONVERSIONS

  FUNCTION  date_of (year  : IN     years;
                     month : IN     months;
                     day   : IN     days)
                           RETURN dates;
    -- Converts from years, months, and days to dates.
    -- Raises invalid_date if year, month, and day do not represent a
```

```
    -- possible date.

  PROCEDURE split   (date  : IN     dates;
                     year  :    OUT years;
                     month :    OUT months;
                     day   :    OUT days);
    -- Converts from dates to years, months, and days.


--CURRENT DATE

  FUNCTION  current_date RETURN dates;
    -- Returns the current date.


--ARITHMETIC OPERATIONS

  FUNCTION  "+" (left  : IN     dates;
                 right : IN     numbers_of_days)
                         RETURN dates;
    -- Adds a number of numbers_of_days to a date.
    -- Raises date_overflow if resultant date is out of range.

  FUNCTION  "+" (left  : IN     numbers_of_days;
                 right : IN     dates)
                         RETURN dates;
    -- Adds a number of numbers_of_days to a date.
    -- Raises date_overflow if resultant date is out of range.

  FUNCTION  "-" (left  : IN     dates;
                 right : IN     numbers_of_days)
                         RETURN dates;
    -- Subtracts a number of numbers_of_days from a date.
    -- Raises date_overflow if resultant date is out of range.

  FUNCTION  "-" (left  : IN     dates;
                 right : IN     dates)
                         RETURN numbers_of_days;
    -- Subtracts one date from another and returns the number of
    -- numbers_of_days.
    -- Raises day_overflow if number of days is out of range.


--COMPARISONS

  FUNCTION  "<"  (left  : IN     dates;
                  right : IN     dates)
                          RETURN boolean;
    -- Returns whether left is less than right.

  FUNCTION  "<=" (left  : IN     dates;
                  right : IN     dates)
                          RETURN boolean;
```

```
    -- Returns whether left is less than or equal to right.

  FUNCTION  ">"  (left  : IN    dates;
                  right : IN    dates)
                          RETURN boolean;
    -- Returns whether left is greater than right.

  FUNCTION  ">=" (left  : IN    dates;
                  right : IN    dates)
                          RETURN boolean;
    -- Returns whether left is greater than or equal to right.

PRIVATE
  TYPE dates IS  RANGE -maximum_year * maximum_days_per_year ..
                       maximum_year * maximum_days_per_year;

  null_date : CONSTANT dates := 0;
    -- 1 represents January 1, 1 CE.  -1 represents December 31, 1 BCE.
END date_manager;
```

There are several things to notice about this example:

- The package is built around type Dates, which is implemented as a private type. This means that predefined assignment and equality are available to the user of the package.
- The subprograms in the visible part of the package specification are operations exported by the class.
- Types Years, Months, Days, and Numbers_Of_Days are needed by several of these subprograms.
- The constant Null_Date and the three exceptions are further declarations needed by a user of the package.

A.3 Example of Limited Package

This example is a standard doubly linked list manager. The package specification is as follows:

```
GENERIC
  TYPE items IS LIMITED PRIVATE;
  WITH PROCEDURE assign (item  : IN     items;
                         to    :    OUT items)  IS <>;
  WITH FUNCTION  "="    (left  : IN     items;
                         right : IN     items)
                                RETURN boolean IS <>;
  WITH FUNCTION  "<"    (left  : IN     items;
                         right : IN     items)
                                RETURN boolean IS <>;
PACKAGE doubly_linked_list_manager IS

  TYPE doubly_linked_lists IS LIMITED PRIVATE;

  TYPE directions          IS (forward, backward);

  overflow      : EXCEPTION;
  no_list       : EXCEPTION;
  no_item       : EXCEPTION;
  out_of_bounds : EXCEPTION;


--STATE

  PROCEDURE view                (list   : IN     doubly_linked_lists;
                                 as     :    OUT doubly_linked_lists);
    -- Creates another view of the same list.

  PROCEDURE copy                (list   : IN     doubly_linked_lists;
                                 to     :    OUT doubly_linked_lists);
    -- Makes a copy of the list.
    -- Raises overflow if storage is exceeded.

  PROCEDURE create              (list   :    OUT doubly_linked_lists);
    -- Allocates head and tail of list.
    -- Sets current item to head.
    -- Raises overflow if storage is exceeded.

  PROCEDURE destroy             (list   : IN OUT doubly_linked_lists);
    -- Deallocates list.

  FUNCTION  is_null             (list   : IN     doubly_linked_lists)
                                  RETURN boolean;
    -- Returns true if list does not exist.

  FUNCTION  is_empty            (list   : IN     doubly_linked_lists)
                                  RETURN boolean;
```

```
    -- Returns true if list has no items in it.
    -- Raises no_list if list does not exist.


--WRITE

  PROCEDURE insert              (item   : IN     items;
                                 into   : IN OUT doubly_linked_lists;
                                 going  : IN     directions := forward);
    -- Inserts item in order in list.
    -- If at least one identical item already exists in list and direction
    -- is forward, inserts new item after last identical item.
    -- If at least one identical item already exists in list and direction
    -- is backward, inserts new item before first identical item.
    -- Sets current item to item inserted.
    -- Raises no_list if list does not exist.
    -- Raises overflow if storage is exceeded.

  PROCEDURE modify_current_item (within : IN OUT doubly_linked_lists;
                                 to     : IN     items;
                                 going  : IN     directions := forward);
    -- Modifies current item in list with specified values.
    -- If modification necessitates relocation of item in list, moves item.
    -- If at least one identical item already exists in list and direction
    -- is forward, inserts new item after last identical item.
    -- If at least one identical item already exists in list and direction
    -- is backward, inserts new item before first identical item.
    -- Raises no_list if list does not exist.
    -- Raises no_item if there is no item at current location.

  PROCEDURE delete_current_item (from   : IN OUT doubly_linked_lists;
                                 going  : IN     directions := forward);
    -- If specified direction is forward and item is only item in list,
    -- sets current item to tail.
    -- Otherwise, sets current item to next item.
    -- If specified direction is backward and item is only item in list,
    -- sets current item to head.
    -- Otherwise, sets current item to previous item.
    -- Raises no_list if list does not exist.


--READ

  PROCEDURE locate              (item   : IN     items;
                                 found  :    OUT boolean;
                                 within : IN OUT doubly_linked_lists;
                                 going  : IN     directions := forward;
                                 again  : IN     boolean    := false);
    -- Sets current item to item.
    -- If item is not found, within does not change.
    -- If specified direction is forward, searches from head of list.
    -- If specified direction is backward, searches from tail of list.
    -- If again is true, searches for next item in same direction.
```

```
    -- Raises no_list if list does not exist.

  FUNCTION  current_item_in     (list    : IN     doubly_linked_lists)
                                           RETURN items;
    -- Returns current item in list.
    -- Raises no_list if list does not exist.
    -- Raises no_item if there is no item at current location.


--TRAVERSE

  PROCEDURE set_to_head         (list    : IN OUT doubly_linked_lists);
    -- Sets current item to head.
    -- Raises no_list if list does not exist.

  PROCEDURE set_to_tail         (list    : IN OUT doubly_linked_lists);
    -- Sets current item to tail.
    -- Raises no_list if list does not exist.

  FUNCTION  at_head_of          (list    : IN     doubly_linked_lists)
                                           RETURN boolean;
    -- Returns true if previous node is head of list.
    -- Raises no_list if list does not exist.

  FUNCTION  at_tail_of          (list    : IN     doubly_linked_lists)
                                           RETURN boolean;
    -- Returns true if next node is tail of list.
    -- Raises no_list if list does not exist.

  PROCEDURE step_forward_in     (list    : IN OUT doubly_linked_lists);
    -- Sets current item to next item in list.
    -- Raises no_list if list does not exist.
    -- Raises out_of_bounds if next position is tail of list.

  PROCEDURE step_backward_in    (list    : IN OUT doubly_linked_lists);
    -- Sets current item to previous item in list.
    -- Raises no_list if list does not exist.
    -- Raises out_of_bounds if previous position is head of list.

  GENERIC
    TYPE inputs  IS LIMITED PRIVATE;
    TYPE outputs IS LIMITED PRIVATE;
    WITH PROCEDURE process (data     : IN OUT items;
                            using    : IN     inputs;
                            updating : IN OUT outputs;
                            again    :    OUT boolean);
  PROCEDURE traverse_forward_in (list     : IN OUT doubly_linked_lists;
                                 using    : IN     inputs;
                                 updating : IN OUT outputs);
    -- Iterates forward over each item in list.
    -- Raises no_list if list does not exist.

  GENERIC
```

```
    TYPE inputs  IS LIMITED PRIVATE;
    TYPE outputs IS LIMITED PRIVATE;
    WITH PROCEDURE process (data     : IN OUT items;
                            using    : IN     inputs;
                            updating : IN OUT outputs;
                            again    :    OUT boolean);
  PROCEDURE traverse_backward_in (list     : IN OUT doubly_linked_lists;
                                  using    : IN     inputs;
                                  updating : IN OUT outputs);
    -- Iterates backward over each item in list.
    -- Raises no_list if list does not exist.

PRIVATE
  TYPE node_kinds IS (head_node, component, tail_node);
  TYPE contents (node : node_kinds := component);
  TYPE content_links IS ACCESS contents;

  TYPE contents (node : node_kinds := component) IS RECORD
    CASE node IS
      WHEN head_node =>
        first    : content_links := NULL;
      WHEN component =>
        previous : content_links := NULL;
        item     : items;
        next     : content_links := NULL;
      WHEN tail_node =>
        last     : content_links := NULL;
    END CASE;
  END RECORD;

  TYPE doubly_linked_lists IS RECORD
    head    : content_links := NULL;
    current : content_links := NULL;
    tail    : content_links := NULL;
  END RECORD;

  null_list : CONSTANT doubly_linked_lists := (NULL, NULL, NULL);
END doubly_linked_list_manager;
```

There are several things to notice about this example:

- The package is built around type Doubly_Linked_Lists, which is declared as a limited private type.
- The package is parameterized by the type Items, which it imports. The operations it also imports in the generic formal part are needed to support manipulation of objects of type Items. This is because Items is imported as a limited private type to allow instantiation with any type.
- The subprograms in the visible part of the package specification are operations exported by the class.

- Type Directions is needed by several of these subprograms.
- The package contains two iterators, Traverse_Forward_In and Traverse_Backward_In. These are generic procedures instantiated with the types to be input and output and a procedure Process to be executed at each node of the linked list. Process allows the data at each node to be updated or read, passes in and out information to be used or acquired at each node, and contains an out parameter Again to signal the iterator to abort the traversal if it is set to False.
- The constant Null_List and the four exceptions are further declarations needed by a user of the package.

A.4 Example of Opaque Package

This example is a network traversal simulator, which simulates the movement of any sort of object (e.g., trains, cars, messages, water) over any sort of network (e.g., tracks, roads, wires, pipes). The package specification is as follows:

```
WITH
  calendar;
GENERIC
  TYPE directions   IS (<>);
  TYPE states       IS (<>);
  TYPE datum_ids    IS RANGE <>;
  TYPE node_ids     IS RANGE <>;
  TYPE node_indices IS RANGE <>;
  TYPE lengths      IS RANGE <>;
  TYPE tolerances   IS RANGE <>;
  TYPE data         IS PRIVATE;

  WITH FUNCTION  backward                          RETURN directions;
  WITH FUNCTION  forward                           RETURN directions;
  WITH FUNCTION  neither                           RETURN directions;
  WITH FUNCTION  starting                          RETURN states;
  WITH FUNCTION  stopping                          RETURN states;
  WITH FUNCTION  normal                            RETURN states;
  WITH FUNCTION  deadlocked                        RETURN states;

  WITH FUNCTION  datum_id_of        (datum    : IN     data)
                                                   RETURN datum_ids;
  WITH FUNCTION  direction_of       (datum    : IN     data)
                                                   RETURN directions;
  WITH FUNCTION  length_of          (datum    : IN     data)
                                                   RETURN lengths;
  WITH FUNCTION  tolerance_of       (datum    : IN     data)
                                                   RETURN tolerances;
  WITH FUNCTION  next_node_index_of (datum    : IN     data)
                                                   RETURN node_indices;
```

```
  WITH FUNCTION  at_destination     (datum    : IN     data)
                                                RETURN boolean;
  WITH FUNCTION  exit_time_of       (datum    : IN     data)
                                                RETURN calendar.time;
  WITH PROCEDURE process            (datum    : IN OUT data;
                                     node_id  : IN     node_ids;
                                     state    : IN     states);

  null_datum      : IN     data;
  null_node_id    : IN     node_ids;
  null_node_index : IN     node_indices;

PACKAGE network_manager IS

  -- This package simulates the movement of any number of data items on a
  -- network of any configuration.  All data items are added to and removed
  -- from their nodes on a first-in-first-out basis.
  --
  -- Generic formal subprogram Process is called in the following situations:
  --
  --     o when a data item is added to a node,
  --
  --     o when a data item is transferred from one node to another,
  --
  --     o when a data item is removed from a node,
  --
  --     o when a data item cannot be transferred due to deadlock.
  --
  -- Process receives the current state of the node as either starting,
  -- stopping, normal, or deadlocked.  It should then take the action
  -- appropriate to the application.


  TYPE nodes       IS PRIVATE;

  TYPE node_groups IS ARRAY(node_ids RANGE <>) OF nodes;

  null_node : CONSTANT nodes;

  overflow  : EXCEPTION;
  no_node   : EXCEPTION;


  PROCEDURE create  (node           :    OUT nodes);
    -- Creates a new node.
    -- Raises Overflow if available storage is exceeded.

  PROCEDURE destroy (node           : IN OUT nodes);
    -- Destroys node.
```

```
  PROCEDURE connect (node           : IN     nodes;
                     named          : IN     node_ids;
                     of_length      : IN     lengths;
                     with_tolerance : IN     tolerances;
                     to_follow      : IN     node_groups;
                     to_precede     : IN     node_groups);

    -- Initializes node with a name, a length, and a tolerance.
    -- Connects node with others, both in front and behind.
    -- If there is no other node, it should be set to null.
    -- Raises No_Node if node does not exist.
    -- Raises Overflow if available storage is exceeded.

  PROCEDURE add     (datum          : IN     data;
                     to             : IN     nodes);
    -- Adds a new data item to node designated by to.
    -- Raises No_Node if node does not exist.
    -- Raises Overflow if available storage is exceeded.

  PROCEDURE enable  (node           : IN     nodes);
    -- Allows execution of node.
    -- Raises No_Node if node does not exist.

  PROCEDURE disable (node           : IN     nodes);
    -- Suspends execution of node.
    -- Raises No_Node if node does not exist.

  PROCEDURE start   (node           : IN     nodes);
    -- Initiates execution of node.
    -- Raises No_Node if node does not exist.

  PROCEDURE stop    (node           : IN     nodes);
    -- Terminates execution of node.
    -- Raises No_Node if node does not exist.

PRIVATE
  TYPE node_objects;
  TYPE nodes IS ACCESS node_objects;

  null_node : CONSTANT nodes := NULL;
END network_manager;
```

There are several things to notice about this example:

- The package is built around type Nodes. It is implemented as an opaque type.
- The package "withs" package Calendar. This is because Calendar is at a lower level of abstraction.

- Several of the generic formal type parameters are attributes of the class: Directions, Node_Ids, Lengths, Tolerances, and Data.
- The other generic formal type parameters are needed for the imported operations; functions Backward, Forward, Neither, Starting, Stopping, Normal, and Deadlocked are needed as values of generic formal types Directions and States; and the generic formal object parameters are other values needed by the class.
- The remaining generic formal subprograms are imported operations needed by the class.
- The subprograms in the visible part of the package specification are operations exported by the class.
- Type Node_Groups is needed by several of these subprograms.
- Constant Null_Node and the two exceptions are other declarations needed by a user of the package.

A.5 Example of Closed Package

This example is a package to log messages in a concurrent system. It is implemented as a closed package, the definition of the log type being hidden in the package body. Consequently, all that appears in the interface are operations. The package specification is as follows:

```
PACKAGE log_manager IS

  PROCEDURE create_log;
    -- Creates log in memory.

  PROCEDURE destroy_log;
    -- Removes log from memory.

  PROCEDURE log      (item : IN     string);
    -- Adds entry to log.

  PROCEDURE dump_log (to   : IN     string);
    -- Writes log to disk.

END log_manager;
```

There are several things to notice about this example:

- The type around which this package is built is hidden in the package body.
- The subprograms in the visible part of the package specification are operations exported by the class.

APPENDIX B. MODEL VENDORS/POINTS OF CONTACT (POC)

Each of the models included in this study are undergoing continual revision as developers receive feedback from their users. For additional information about a model or package, the designated vendor/point of contact listed in Table B-1 should be contacted.

TABLE B-1. MODEL VENDORS/POINTS OF CONTACT (POC)

MODEL	VENDOR/POC
Ada COCOMO	Mr. Bernie Roush NASA Johnson Space Center Mail Code FM 7 Houston, TX 77058 (713)483-9092
PRICE S	Dr. Robert E. Park PRICE Systems General Electric Company 300 Route 38, Bldg. 146 Moorestown, NJ 08057 1-800-GE-PRICE
SASET	Mr. Steve Gross Naval Center for Cost Analysis Department of the Navy Washington, DC 20350-1100 (202) 694-0173
SoftCost-Ada	Mr. Donald Reifer Reifer Consultants, Inc. 25550 Hawthorne Blvd, Suite 208 Torrance, CA 90505 (213) 373-8728

TABLE B-1. COST MODEL POINTS OF CONTACT (Continued)

SPQR/20	Mr. Wayne Hadlock Software Productivity Research, Inc. P.O. Box 1033 1972 Massachusetts Avenue Cambridge, MA 02140 (617) 495-0120
SYSTEM-3	Mr. Wayne Stanley Computer Economics, Inc. Suite 109 4560 Admiralty Way Marina del Rey, CA 90292-5424 (213) 827-7300
DoD:	Lt. Paul Marsey Wright-Patterson AFB (513) 255-6347

TABLE B-2. ADA COCOMO IMPLEMENTATIONS POINTS OF CONTACT (POC)

PACKAGE	POC
BMO*	Lt. Darrish Headquarters BMO-ACS Norton AFB, CA 92409-6468 (714) 382-4713 Autovon: 876-5836
COSTAR	Mr. Dan Ligett Softstar Systems 28 Ponemah Road Amherst, NH 03031 (603) 672-0987
COSTMODL	Mr. Bernie Roush NASA Johnson Space Center Mail Code FM 7 Houston, TX 77058 (713) 483-9092
GECOMO	Ms. Susan Boers GEC Software 1850 Centennial Park Drive, Suite 300 Reston, VA 22091 (703) 648-1551 Mr. Peter Sizer 132-135 Long Acre London WC2E England 44-1-240-7171

* Currently does not include Incremental Development.
Restricted use to Government only.

APPENDIX C. HARDWARE REQUIREMENTS

Table C-1 summarizes the hardware requirements for each of the models. All of the models are available on an IBM PC (or compatible). Additional details concerning hardware requirements are provided in the following text.

TABLE C-1. HARDWARE REQUIREMENTS

	IBM PC	ZENITH-248	PRIME	VAX	MODEM
COSTMODL	X				
PRICE S	X		X		X
SASET	X				
SoftCost-Ada	X			X	
SPQR/20	X				
SYSTEM-3	X	X			

X = Available to DoD and Commercial users

1. **COSTMODL**: COSTMODL runs on IBM PCs and compatibles. A hard disk and 640K bytes of memory are required. Any monitor may be used, but a color monitor is preferred since color is used to differentiate between different classes of data.

2. **PRICE S**: PRICE S runs on a PRIME minicomputer operating under PRIMOS. In addition, PRICE S can be accessed via a time-sharing system with an office terminal and standard modem.

3. **SASET**: SASET may be hosted on any IBM PC or compatible with a minimum of 512K bytes of memory, one disk drive, and an 8088/86, 80186, 80286, 80386 microprocessor running PC-DOS or MS-DOS, version 2.0 or higher. The model functions with either a color or monochrome monitor. A hard disk and printer are optional.

4. **SoftCost-Ada**: SoftCost-Ada runs on an IBM PC, PC/XT, PC/AT, PS/2 or compatible with a minimum of 256K bytes of memory and a color or monochrome display. The system requires PC-DOS or MS-DOS, version 2.0 or higher. A minimum of one floppy disk drive is required. A hard disk drive and printer are optional. SoftCost-Ada may also be hosted on the Digital MicroVax II or VAX 11/780 with VMS version 4.6 or higher.

5. **SPQR/20**: SPQR/20 runs on an IBM PC, XT, AT, or compatible with 512K bytes of memory and a color or monochrome display. Two floppy disk drives or a floppy disk drive and a hard disk drive are required.

6. **SYSTEM-3**: SYSTEM-3 runs on an IBM PC, XT, AT, Zenith 248 or compatibles with a minimum of 512K bytes of memory and a color or monochrome display. The system requires PC-DOS or MS-DOS, version 2.0 or higher. A minimum or one floppy disk drive is required.

APPENDIX D. CONTRACTUAL ARRANGEMENTS AND COSTS

Tables D-1 and D-2 summarize the availability and cost of models applied in the test case study. Table D-1 summarizes the availability of each model to DoD and commercial users. Availability through request means that a potential user can receive the model at no cost by contacting the model POC. The model is received on diskettes that are provided by the requesting agency. Table D-2 shows the DoD rates for each of the models. Separate commercial rates apply to PRICE S and System-3. Additional costs may be associated with user training, which is required for some of the models. Also, rates will vary depending upon the type of licensing agreement procured (annual, site, corporate, etc.).

1. COSTMODL: COSTMODL is available to all DoD and commercial users. Version 5.0 of COSTMODL is available by contacting Bernie Roush at NASA- Johnson Space Center. Version 5.0 implements the complete Ada COCOMO model and the Incremental Development model which were introduced by Dr. Boehm at the November 1987 COCOMO User's Group Conference. It does not include the enhancements to the Ada model that Dr. Boehm introduced at the 1988 COCOMO User's Group Conference. Enhancements will be incorporated in Version 6.0. Requests should be accompanied with three 360K 5.25" disks. Implementors are currently soliciting feedback on the package's user interface. After upgrades, COSTMODL will be available from

   NASA/COSMIC
   The University of Georgia
   Computer Services Annex
   Athens, GA 30602
   (404) 542-3265

   There is a nominal handling charge for the program.

2. PRICE S: PRICE S is part of the PRICE system of models that includes PRICE SZ for software sizing, and PRICE SL for software life-cycle costs (maintenance, enhancement, and growth activities associated with life-cycle support). Government users can use the PRICE S package on a time-sharing basis at $82 per hour (through 11/91) by contacting Lt. Ken Nelson of the Aeronautical Systems Division at Wright Patterson AFB. Commercial users can use the PRICE S package on a time-sharing basis at $15/hour of connect time and

TABLE D-1. CONTRACTUAL ARRANGEMENTS

	PURCHASE	LEASE	TIME SHARE	REQUEST
COSTMODL				X
PRICE S		X	X	
SASET				X
SoftCost-Ada		X		
SPQR/20	X			
SYSTEM-3		X		

TABLE D-2. LEASE/PURCHASE RATES (DoD)

	FIRST UNIT	EXTRA UNIT	TIME SHARE
COSTMODL	No Cost		
PRICE S			$82/Hour
SASET	No cost, but controlled access		
SoftCost-Ada	$8,000	$1,000/Copy	
SPQR/20	$5,000	Negotiable	
SYSTEM-3	$9,550/Year	$800/Copy	

$0.060/resource unit of CPU time by contacting PRICE Systems at Moorestown, New Jersey, but they must also pay an access fee of $40,000/year for one unit or $60,000/year for unlimited access. Commercial users can also lease the PRICE S package for installation on their own PRIME minicomputer at $60,000/year for one user at a time or $80,000/year for unlimited access. A one week training course is mandatory and costs $1,312.50 (through 11/91) for a government student or $1,750 for a commercial student. These costs include refresher training, manual updates, technical assistance, and newsletter at no additional charge.

3. **SASET**: SASET is presently being used by the U.S. Air Force Cost Center and the Naval Center for Cost Analysis. Availability to DoD users on a broader basis is an issue that will be decided by Mr. Steve Gross and the Naval Center for Cost Analysis. There are presently no plans to market the SASET model, but Martin Marietta Corporation may market a derivative model.

4. **SoftCost-Ada**: SoftCost-Ada is available for a monthly or annual licensing fee. The SoftCost-Ada PC version annual licensing fee for one unit costs $8,000. The price for additional copies is $1,000. A site license is $11,000. The SoftCost-Ada Vax version costs $8,000 for the first license (4 users) and $1,000 for each additional license (4 users). Prices include a telephone help line and system upgrades at no additional charge. SoftCost-Ada Vax version site licensing agreements are negotiable. A GSA contract is being negotiated which will result in a discount for DoD users.

5. **SPQR/20**: A SPQR/20 one time licensing fee for purchasing one unit costs $5,000. Site licenses and multi-volume purchases are negotiable.

6. **SYSTEM-3**: The System-3 annual rate for government users is $9,550/year for one unit; additional units (two and three) are $800/year and $600/year for four or more units. The System-3 annual licensing fee for commercial users is $12,500 for one unit; additional units (two through four) are $2,000/year. In addition, further price reductions are available for blocks of five, ten, 25 and 50 units. System-3 has a training course available. This course is strongly recommended and costs $790/person when given at the CEI facility. Training at the customer's facility (up to 20 persons per session) is $4500/session plus travel and living expenses for CEI personnel. Commercial training costs are $5,800/per session plus travel and living expenses. Prices include a telephone help line and system upgrades at no additional charge.

APPENDIX E. SCOPE OF COVERAGE: LIFE-CYCLE PHASES AND ACTIVITIES

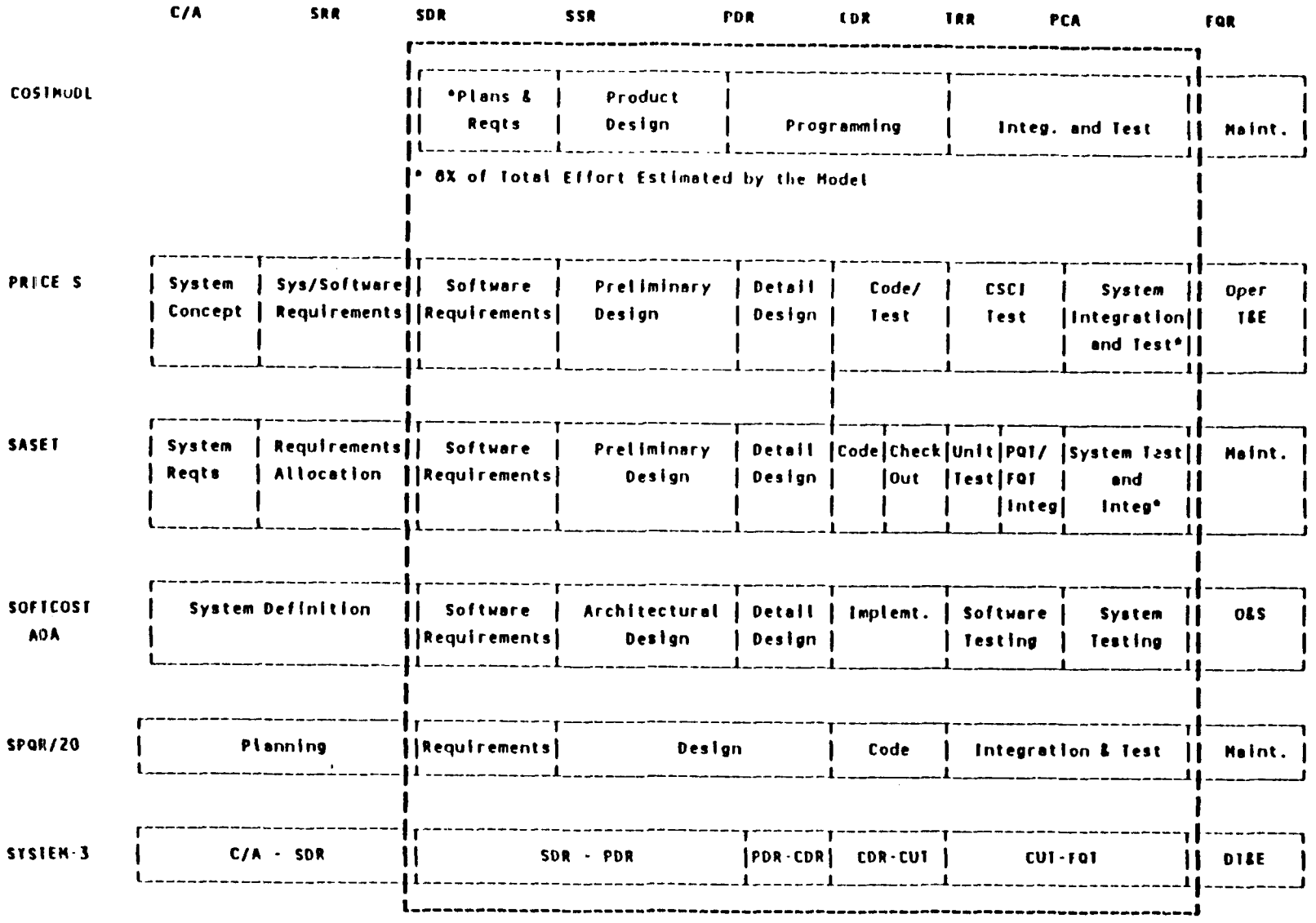
With the exception of SPQR/20, each model included in this Ada costing study generates an effort expenditure summary in terms of the software cost elements encompassed by the estimate and by life-cycle phase. SPQR/20 provides estimates only in terms of the software project activity. However, these activities can be mapped to life-cycle phases. Table E-2 shows the range of life-cycle phases covered by each model. Phases are mapped to technical reviews and audits [DOD-STD-2167A] in order to provide a basis for comparison of models in terms of life-cycle coverage. Blocks depicted in Table E-1 are labelled using the same phase terminology prescribed by each model. It is evident from the table that phases are not defined in a standard way across all models. All of the models cover the operational or maintenance phase in addition to development. Operational support, following successful completion of a software acceptance review (FQR), estimated for each model is provided in Table E-1.

A breakdown of software cost elements encompassed by the model estimates is provided in Table E-3. Activities are described using the exact terminology of the model. A separate estimate given in terms of person-months of effort is provided for each cost element.

TABLE E-1. OPERATIONAL SUPPORT ACTIVITIES

COSTMODL:	Annual maintenance
PRICE S:	Operational support for user-specified length
SASET:	Operational support for user-specified length
SoftCost-Ada:	Operational support for user-specified length
SPQR/20:	Up to 5 years of operational support
SYSTEM-3:	15 years of operational support

TABLE E-2. SCOPE OF COVERAGE: LIFE-CYCLE PHASES



* Resources expended during this phase are not included in the case study.

TABLE E-3. SOFTWARE COST ELEMENTS ENCOMPASSED BY MODEL ESTIMATES

MODEL	ACTIVITY
COSTMODL	. Requirements Analyses . Product Design . Programming . Test Planning . Verification and Validation . Project Office . Configuration Management/ Quality Assurance . Manuals
PRICE S	. Software Design . Programming . Documentation . Systems Engineering and Program Management . Quality Assurance . Configuration Management
SASET	. Software Engineering . Systems Engineering . Quality Assurance . Test Engineering
SoftCost-Ada	. Software Development . Software Management . Software Configuration Management . Software Quality Evaluation
SPQR/20	. Planning . Requirements . Design . Coding . Integration/Test . Documentation . Management
SYSTEM-3	. Systems Engineering . Project Management . Design . Programmers . Quality Assurance . Configuration Management . Test . Data Manipulation

APPENDIX F. GENERIC UNITS & TEMPLATE GENERATORS

Two techniques that directly support reuse are the use of generic units and the use of template generators. These two are similar in that each is used to reduce the amount of coding required by a programmer. In the case of a generic unit, a unit (e.g., a subprogram) is written with the use of parameters in place of data types or subprogram declarations. To instantiate the unit requires only that actual data types or declarations replace the parameters. Template generators provide the structure for a segment of code. The structure is based on patterns of program statements that repeat. In this case, the programmer fills in the portions of the template unique to a particular use. Generic units and template generators are each described in more detail in this appendix.

Generic Units

Generic units in Ada are templates that are filled in by the generic instantiation. Generic units can be either packages or subprograms. All generic units contain a generic formal part in which generic formal parameters are specified. The generic formal parameters are the "wildcards" in the template. Upon instantiation, matching actual parameters must be supplied for the generic formal parameters. These matching actual parameters are either supplied explicitly or by default.

There are three kinds of generic formal parameters:

- generic formal types
- generic formal objects
- generic formal subprograms.

Generic formal types allow a generic unit to be parameterized for a given type. Generic formal objects act as either constants or global variables, depending on their mode. A generic formal object of mode in will behave like a constant in the generic unit; a generic formal object of mode "in out" will behave like a global variable. Generic formal subprograms allow procedures and functions required by the generic unit to be imported by that unit. In general, generic formal

parameters allow entities to be passed to a generic unit from above, by the unit that "withs" and instantiates the generic unit, rather than from below, by "withing" a lower-level unit.

The following is a typical example of a generic unit:

```
generic
  type Items is limited private;
  with procedure Assign (Item  : in     Items;
                         To    :    out Items);
  with function  "="    (Left  : in     Items;
                         Right : in     Items)
                                return Boolean is <>;
  with function  "<"    (Left  : in     Items;
                         Right : in     Items)
                                return Boolean is <>;
package Doubly_Linked_List_Manager is

  -- package specification

end Doubly_Linked_List_Manager;
```

In this instance Doubly_Linked_List_Manager is parameterized by the type Items, which defines the components of the linked list. In addition, the package needs the subprograms Assign and "=" because type Items is limited private, and "<" because the items in the list need to be ordered. There are no generic formal objects in this example.

Because generic units are templates, multiple instances of them can be created by parameterizing them in various ways. The linked list above is a good example of this. One can create as many instances of a linked list for as many types of data as one likes by simply instantiating the generic unit with each type that one needs. Clearly, generics provide a tool which allows the programmer to create multiple instances of patterns detected in the application.

Template Generators

In Ada, generics were intended as a means of enhancing reuse among software components. A generic unit is a template for a package or subprogram. Generics work particularly well for abstract data types, such as linked lists, where a generic package can be instantiated with the type out of which the linked list is to be constructed. However, generics do not cover all cases in

which a template is required. This is particularly the case when the structure of various library units is the same but the content is different. As an example, consider the following two subprogram bodies:

```
procedure Update_Units (Using : in     Units) is
  Unit : Units;
begin
  Choose(Unit, From => Units_Table, Having => Using.Id);
  if Unit.Id = Null_Id then
    Insert(Using, Into => Units_Table);
  else
    raise Duplicate_Key;
  end if;
end Update_Units;

procedure Update_Personnel (Using : in     Personnel) is
  Person : Personnel;
begin
  Choose(Person, From => Personnel_Table, Having =>  Using.Name);
  if Person.Name = Null_Name then
    Insert(Using, Into => Personnel_Table);
  else
    raise Duplicate_Key;
  end if;
end Update_Personnel;
```

Notice that they both have a common structure. We might wish to take advantage of this commonality and construct a generic procedure that would embody the common flow of control, but which would be instantiated with the differences in content. Unfortunately, however, other than the flow of control, practically everything else would have to be passed as a generic parameter. This practically nullifies any advantage that would accrue from using a generic unit.

The interface would appear as follows:

```
generic
  type Records is private;
  type Keys    is private;
  Table    : in     Tables;
  Null_Key : in     Keys;
  with procedure Choose (Rec    :    out Records;
                         From   : in     Tables;
                         Having : in     Keys);
  with procedure Insert (Rec    : in     Records;
                         Into   : in     Tables);
  with function  Key_Of (Rec    : in     Records)
```

```
                                        return Keys;
procedure Update (Using : in     Records);
```

The generic formal parameters Records and Keys are necessary in order to make Update general for all records in the database. The constants Table and Null_Key and the procedures Choose and Insert are needed in the body of Update. The function Key_Of is necessary because Update does not know the structure of Records and hence has no way of extracting the key field from the record.

Given this information, the body would appear as follows:

```
procedure Update (Using : in     Records) is
  Rec : Records;
begin
  Choose(Rec, From => Table, Having => Key_Of(Using));
  if Key_Of(Using) = Null_Key then
    Insert(Using, Into => Table);
  else
    raise Duplicate_Key;
  end if;
end Update;
```

In order to instantiate Update, matching functions for Key_Of would have to be written:

```
function Id_Of (Unit : in     Units)
                      return Ids is
begin
  return Unit.Id;
end Id_Of;

function Name_Of (Person : in     Personnel)
                          return Names is
begin
  return Person.Name;
end Name_Of;
```

Assuming the other information is all available globally, Update would then be instantiated as follows for Units and Personnel:

```
procedure Update_Units is new Update(
      Records  => Units,
      Keys     => Ids,
```

```
        Table    => Units_Table,
        Null_Key => Null_Id,
        Choose   => Choose_Unit,
        Insert   => Insert_Unit,
        Key_Of   => Id_Of);


    procedure Update_Personnel is new Update(
        Records  => Personnel,
        Keys     => Names,
        Table    => Personnel_Table,
        Null_Key => Null_Name,
        Choose   => Choose_Person,
        Insert   => Insert_Person,
        Key_Of   => Name_Of);
```

If one has to go to this much trouble to make a library unit generic, it is probably not worth the effort. The conclusion is that payoffs from using generics do not arise unless a substantial portion of the content of the library unit is genericizable in addition to its structure. Nevertheless it would be helpful to have a means of taking advantage of a common structure. The fact that generics will not necessarily help in this regard does not mean that having some kind of templating mechanism is not a useful concept. It only highlights the necessity of having a tool which usefully provides templates for library units which share only a common structure.

Let us explore this in more detail. What is needed is a tool that would produce procedures Update_Units and Update_Personnel automatically. One approach that could be taken is to define the template by writing a procedure in which the variant items are indicated by means of symbols. A separate file of substitutions for these symbols could then be created. A tool could read both files, replacing the symbols in the first file by the substitution-instances in the second. In that way Update_Units and Update_Personnel could be generated automatically.

Below is an example of this technique. First a template for Update_Units and Update_Personnel might be constructed as follows:

```
procedure <procedure_name> (Using: in <type_1>) is
  <variable_1> : <type_1>;
begin
  Choose(<variable_1>, From => <table>, Having =>
         Using.<key>);
  if <variable_1>.<key> = <constant_1> then
    Insert <Using, Into => <table>);
  else
```

```
    raise Duplicate_Key;
  end if;
end <procedure_name>;
```

Then a file of substitution-instances would be created for each procedure to be instantiated.

The one for procedure Update_Units might look like this:

procedure_name	Update_Units
type_1	Units
variable_1	Unit
table	Units_Table
key	Id
constant	Null_Id

A tool could easily read this file, store the equivalencies in memory, and generate Update_Units by substituting the values in the right column for the symbols in the left.

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	July 1990	Final REport

4. TITLE AND SUBTITLE

Software Reuse Methods

5. FUNDING NUMBERS

MDA903-87-D-0056

6. AUTHOR(S)

Steve Goldstein

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

ITT Research Institute
4600 Forbes Boulevard
Lanham, MD 20706

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AJPO
3 E 114
The Pentagon
Washington, DC 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Given the increasing number of computerized, software-driven systems being designed and implemented throughout the Department of Defense (DOD) and industry, reusability of software has become a critical endeavor. To better prepare software engineers and computer programmers to address the challenge of software reuse, the U.S. Army Communications and Electronics Command (CECOM) has undertaken a program to investigate different software reuse methods. This effort will provide guidelines on reuse strategies for software developers. This report examines reusability characteristics, domain analysis, domain-independent approaches, domain-specific approaches, cost/benefit analysis for software reuse, and reuse metrics.

14. SUBJECT TERMS

SOFTWARE REUSE, REUSE METHODS, DOMAIN APPROACHES, COST BENEFIT ANALYSIS, REUSE METRICS.

15. NUMBER OF PAGES

117

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNLIMITED

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to **stay within the lines to meet optical scanning requirements.**

Block 1. Agency Use Only (Leave blank).

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	PR	- Project
G	- Grant	TA	- Task
PE	- Program Element	WU	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. (If known)

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD	- See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE	- See authorities.
NASA	- See Handbook NHB 2200.2.
NTIS	- Leave blank.

Block 12b. Distribution Code.

DOD	- DOD - Leave blank.
DOE	- DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
NASA	- NASA - Leave blank.
NTIS	- NTIS - Leave blank.

Block 13. Abstract. Include a brief (Maximum 200 words) factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code (NTIS only).

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.