

DTIC FILE COPY

2

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of Management and Budget, Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE 10 MAY 89	3. REPORT TYPE AND DATES COVERED Final Report	
4. TITLE AND SUBTITLE A Program Manager's Guide to Generic Architectures			5. FUNDING NUMBERS C- F33600-87-D-0337	
8. AUTHOR(S) Richard B. Quanrud			6. PERFORMING ORGANIZATION REPORT NUMBER 3451-4-214/1	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) SoftTech, Inc. 3100 Presidential Drive Dayton, OH				
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Aeronautical Systems Division SCOL/ Building 676 Area B Wright Patterson AFB, OH 45433			10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION/AVAILABILITY STATEMENT UNLIMITED			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This report is an introduction to the use of generic architectures for managers of DoD software acquisition programs. Generic architectures are an approach to the development and use of reusable software components. They extend the level of software reuse within a system through the development of reusable components designed for use in a specific family of applications and within a specific design context. As such, they may be used to support functions that are not easily supported with more traditional reusable software libraries.				
14. SUBJECT TERMS Reusable Software, Command & Control Software, Ada			15. NUMBER OF PAGES 30	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED			16. PRICE CODE	
18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED		19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED		20. LIMITATION OF ABSTRACT UNLIMITED

NSN 7540-01-280-5500

Standard Form 298, (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-01DTIC
ELECTE
SEP 27 1990
S E D

AD-A227 074

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.

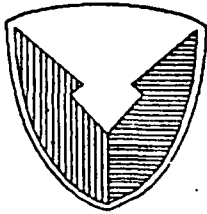
Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

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.



CECOM

CENTER FOR SOFTWARE ENGINEERING
ADVANCED SOFTWARE TECHNOLOGY

Subject: **A PROGRAM MANAGER'S GUIDE TO
GENERIC ARCHITECTURES**

Final Report

CIN: C04-029NN-0001-00

10 MAY 1989

A PROGRAM MANAGER'S GUIDE TO GENERIC ARCHITECTURES:

Final Report

PREPARED FOR:

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

PREPARED BY:

SofTech Inc.
460 Totten Pond Road
Waltham, MA 02254



10 MAY 1989

Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Final Report, May 10, 1989

Copyright © SofTech, Inc., May 1989

This material may be reproduced by or for the U.S. Government pursuant to the copyright license under the clause at DFARS 252.227-7013 (April 1988).

Printed in the U.S.A.

The reproduction of this material is strictly prohibited. For copy information, contact the U.S. Army CECOM, Ft. Monmouth, NJ.

The information in this document is subject to change without notice.

Apple and Macintosh are trademarks of Apple Computer, Inc.

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02154-1960

TABLE OF CONTENTS

<u>Section</u>		<u>Page</u>
	A PROGRAM MANAGER'S GUIDE TO GENERIC ARCHITECTURE	vii
1	REUSABLE COMPONENT LIBRARIES	1
2	GENERIC ARCHITECTURES	3
3	RELATIONSHIP TO LIBRARIES OF REUSABLE COMPONENTS	6
4	ADA SUPPORT FOR GENERIC ARCHITECTURES	8
	4.1 Ada Packages as Reusable Components	8
	4.2 Non-Intrusive Modifications to Reusable Components	9
	4.3 Object-Oriented Development	9
5	BENEFITS	11
6	DEVELOPMENT OF A GENERIC ARCHITECTURE	13
	6.1 Requirements Analysis	13
	6.2 Design	14
	6.3 Coding and Testing	15
7	ACQUISITION CONSIDERATIONS	16
	7.1 Project Domain	16
	7.2 Contractor Selection	17
	7.3 Development	18
8	SUMMARY AND CONCLUSIONS	20
9	FURTHER READING	22

LIST OF FIGURES

<u>Figure</u>		<u>Page</u>
1	The Reuse of Components in Applications with Different Designs	2
2	The Reuse of Components in Systems with a Common Design	3
3	A Workstation Network Applications Environment	5

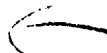
A PROGRAM MANAGER'S GUIDE TO GENERIC ARCHITECTURES

↓
The reuse of existing software components can significantly improve productivity on software development projects. A generic architecture provides a way to increase the level of reuse beyond that possible with traditional approaches.)

Typically, reusable components are available from software libraries that provide them for use on a wide variety of applications. However, with a generic architecture, the components are developed for use in the applications of a specific project or organization and are designed to meet the specific requirements of those applications. This change in focus makes it possible to take advantage of similarities in the requirements and design of those applications so as to develop reusable components for operations that are difficult to support with more general purpose library components.)

In general, the reuse of software components contributes to software productivity by reducing the number of lines of new code that must be designed, coded, tested, and maintained. However, the benefits to be derived from a generic architecture are not limited to greater productivity, but include greater reliability, enhanced interoperability, improved operator performance, and lower training costs as well. Generic architectures also make software more adaptable to changing requirements, environments, and technology and provide strong support for the rapid prototyping of related applications.

(KR)



Section 1

REUSABLE COMPONENT LIBRARIES

The reuse of software components is common in most projects. It may be as simple as the exchange of source code among programmers who recognize common requirements and opportunities for code reuse. However, in such cases, the code is often maintained separately in each application in which it is used. Although some time is saved in the design and coding of the component, the other benefits of common testing and maintenance are lost.

A more organized approach is to place reusable components in a common library. Each component is tested and documented before it enters the library. Copies of the code and documentation are distributed to users on request. Problems with a component are reported to the library where they can be fixed and the changes made available to all users.

Reusable component libraries typically serve a number of projects in an effort to maximize the opportunities for the reuse of the components. Libraries also collect components from a variety of sources to increase the likelihood that they will have the components needed to meet the specific requirements of their client projects. At some point, the number of components may become large enough to require the use of classification systems and retrieval software to aid in the identification of the most appropriate component for a particular use.

The typical library component is designed to perform some well defined function that is likely to be required in a number of different software systems. They are particularly useful when they encapsulate the expertise required to perform a complex operation that would be difficult or expensive to program. Device drivers, window managers, and complex mathematical functions are good examples. They can include major subsystems which provide a significant part of the software environment in which an application program will operate. In most cases, they are designed to be independent of the design of the application in which they are used.

This element of design independence is illustrated in Figure 1. The figure provides a conceptual model of three applications, of very different design, which share several reusable components. Letters have been assigned to the components for identification. A different set of components has been used in each application and the components have no direct interfaces with each other. The interface code that invokes the operations of each of the reusable components is generally not reusable and is developed as new code in each of the applications.

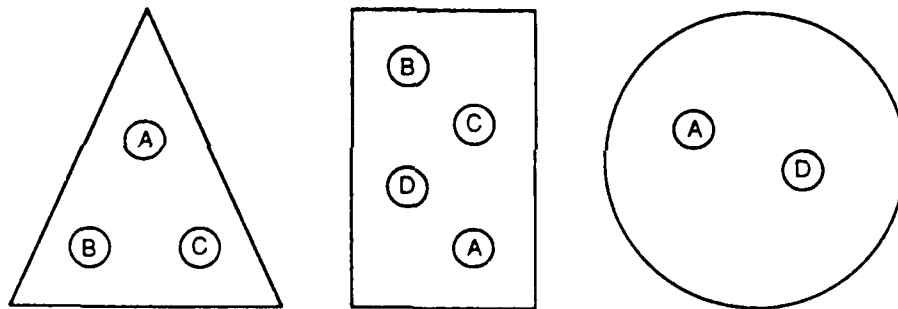


Figure 1. The Reuse of Components in Applications with Different Designs

Section 2

GENERIC ARCHITECTURES

In the development of related applications, design independence is not the central issue. For these applications it is often possible to use a common high-level design and to develop additional reusable components that are based on that design. The common design and design-dependent reusable components are the essential elements of a generic architecture.

The use of a generic architecture is illustrated in Figure 2. Here the high-level design of the two applications is the same. Nominally, each application has a similar set of components, and those components have direct interfaces with each other. Many of the components can be reused without change. Differences in the requirements of the two applications are handled by modifying or replacing only those components affected by those requirements. In Figure 2, components "C" and "E" are different in the two systems, but they have the same interface with the other components in both systems. This illustrates the generic architecture approach to reusable software.

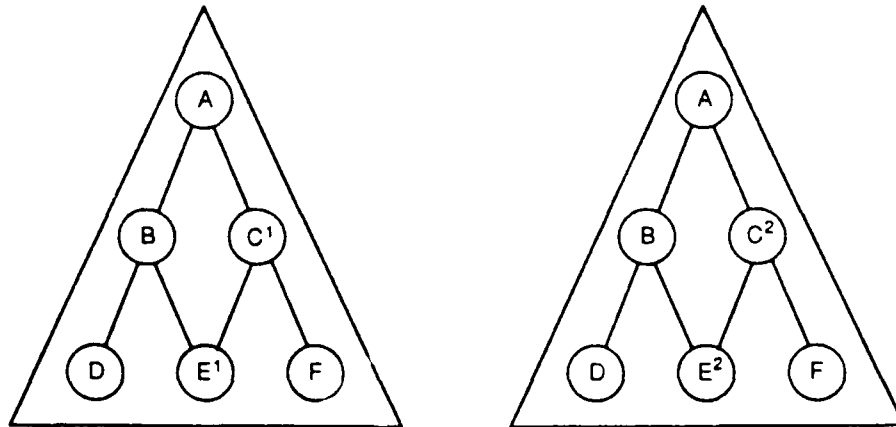


Figure 2. The Reuse of Components in Systems with a Common Design

A generic architecture provides a high-level design for a family of related applications and a set of reusable components that are intended for use in those applications. The use of a common high-level design allows code that is dependent on that design to be included in the reusable components. A good example of design-dependent code is code that is contained in one component and invokes a function contained in another component. Such a design dependency limits the use of the first component to designs in which the second component is also used. Acceptance of design dependencies allows more of the code of the application to be included in the reusable components. However, the use of those components is limited to applications in which the components have the same design relationship.

Under what circumstances will the designs used in a family of applications be similar enough to allow the use of a generic architecture? The answer to this question often depends more on the operating context or environment of a set of applications than on the applications themselves.

For example, applications that are executed on networked workstations, such as those shown in Figure 3, are likely to be good candidates for the use of a generic architecture. This is because those applications are likely to have a large number of common support requirements. These might include support for an interactive dialog with the user, the ability to print reports, or the ability to send and receive messages using standard protocols. Although the displays, reports, and messages will be different for each application, much of the supporting code required to produce displays, print reports, or format messages may be the same. Moreover, the basic control mechanisms are likely to be similar in most of the applications.

It is usually not difficult, in such situations, to place the code that is truly application specific in components that are separate from those supporting the more generic elements of an operation. This allows the application specific components to be replaced or modified without changing the remaining supporting components.

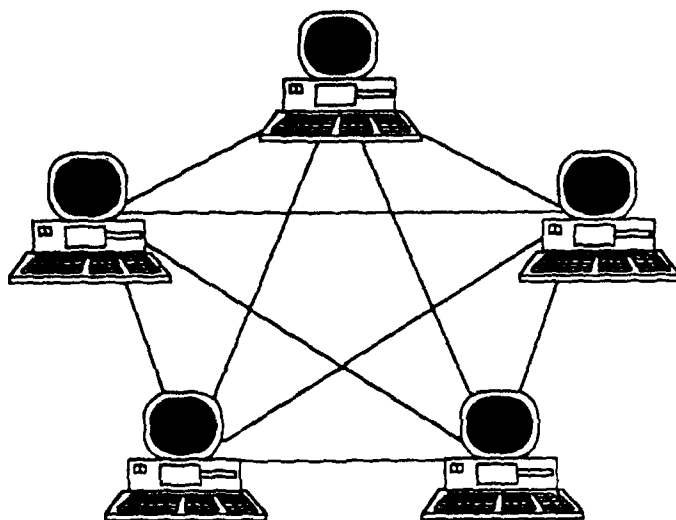


Figure 3. A Workstation Network Applications Environment

The amount of code that can be reused in similar applications can be surprisingly large. Magnavox was able to reuse 75% of the code developed for the Army's Advanced Field Artillery Tactical Data System (AFATDS) on another project, the Elevated Target Acquisition System. SofTech was able to reuse 68% of the code developed for the applications of the Air Force Rapid Emergency Reconstitution (RAPIER) project across the nine applications of that project. It has been estimated that over 70% of the code developed for Apple Macintosh applications could be reused across those applications.

The recent trend toward the procurement of large numbers of commercial desk-top computers and workstations for general use throughout a command or program has created ideal conditions for the development and use of generic architectures and for the use of reusable software in general. Within such an environment, there are likely to be numerous opportunities to exploit similarities among related applications.

Section 3

RELATIONSHIP TO LIBRARIES OF REUSABLE COMPONENTS

The development of a generic architecture is not an alternative to the use of a library of software components in the development of computer software. Instead, it compliments the use of library components by allowing additional components of an application to become reusable. The development of a generic architecture should take full advantage of any components that can be supplied from an existing library.

The important difference between the components that might be supplied by a library and the additional components that would be developed for a generic architecture is that library components usually stand alone. They do not normally depend on operations performed by other library components although they may require the services of the underlying operating system.

The additional components developed for a generic architecture generally do not stand alone. They must operate in the same design context and in conjunction with other components on which they depend for support. They can be reused across applications only to the degree that those applications share a common design.

Any stand-alone components that might be developed for a generic architecture are good candidates for inclusion in a reusable component library. The remaining design-dependent components might also be included, but they raise a number of special problems. The interdependencies among these components must be clearly identified for the users of the library so that all of the necessary components are included in any application in which they are used. It is also useful for a library to provide some way to identify all of the components that are used in a given generic architecture so that they can be examined as a group for possible use in other applications.

Indeed, other projects which have the same operating environment may be able to reuse a generic architecture. If not, they may be able to

adapt large parts of it to a new generic architecture. Existing generic architectures can also serve as useful examples to later developers. The latter consideration alone probably justifies their inclusion in libraries of reusable components.

It may also be desirable to include a generic architecture in a library for long term maintenance, particularly if it is likely to be reused, in whole or in part, by other projects.

However, in most cases, the organization that operates a library is probably not the ideal organization to develop a generic architecture. The development of a generic architecture requires a fundamentally different point of view. The ability to achieve high levels of software reuse within the applications based on a generic architecture depends to some degree on resisting the temptation to over-generalize, i.e., to design the components to meet requirements that go beyond those of the project that plans to use the architecture. By its nature, a library organization tries to make components useful to a wide range of users and is probably more likely to over-generalize than a project organization.

A generic architecture is inherently project specific. Ideally, its development should be sponsored by the organization responsible for the project that will be supported by the architecture. No one else has adequate information on the requirements of the applications. No one else is as likely to benefit from its development.

Section 4

ADA SUPPORT FOR GENERIC ARCHITECTURES

The ability to develop an inventory of reusable "software parts" was a major consideration in the design of the Ada language. It should not be surprising that Ada provides strong support for generic architectures and for reusable components in general.

Reusable components should be treated as "black boxes" which perform well defined operations, but keep the details of the implementation of those operations hidden from the user. The object here is not secrecy, but rather the ability to use a component without having to be concerned about the details of its implementation. If this is not the case, it can be expected that the user will soon modify the component and it will no longer be reusable. A component is truly reusable only when it can be used again without changing the program code of the component.

4.1 Ada Packages As Reusable Components

The Ada language is well suited to the development of black box components. Most of the operations in an Ada program are grouped into "packages" along with the data used in those operations. A reusable component in Ada is usually implemented as an Ada package.

The interface to an Ada package is defined in an Ada coded package "specification". That interface defines the form of all program calls to the operations of the package as well as any data that may be accessed from outside the package.

The code that implements the operations of a package is contained elsewhere in a package "body". Here one would find the code that carries out the operations defined in the interface specification as well as any local data that is used within the package.

4.2 Non-Intrusive Modifications To Reusable Components

Ada also has several features which allow the programmer to modify the operations of a package without changing the program code of the package. The two most important are "generics" and "separate subunits".

A generic package differs from a normal package in that it allows selected data and operations in the package to be redefined at the time that the package is used in a program. The package specification identifies the data and operations that can be changed in this manner. The remainder of the package is unchanged, and no permanent changes are made to the program code of the package.

A separate subunit is the implementation of an operation outside of the package in which it is defined. Its interface is still defined in the package specification, but its program code is separate from the package body. In its place, the body contains a statement that indicates that the code for the operation is contained in a separate subunit. This means that the program code for that operation can be replaced without changing the code of the package body.

Of course, it is always possible to replace a component that cannot be suitably modified. Indeed, most of the code for the replacement component may be taken from the original. The penalty is that the component is now unique to the application in which it is used and must be maintained separately as long as it is used.

4.3 Object-Oriented Development

Ada is considered by most authorities to be an "object-oriented" language. This means that it allows programmers to implement software components using an object-oriented programming style. An object-oriented software component contains all of the data used to represent some real-world object as well as the code for all operations on that object.

For example, a message might be such an object. An object-oriented message component or package would contain all of the data required to

describe one or more messages. It would also include the program code for all of the operations that might be performed on messages. Examples are operations which create, send, receive, and display messages.

Object-oriented development provides a logical basis for grouping operations and data into components. In the process it reduces the amount of data that needs to be accessed from other components. This can substantially reduce the complexity of software systems. It also simplifies the interfaces among components and makes them inherently more reusable.

It is sometimes argued that Ada is not truly an object-oriented language because it does not support "inheritance". Inheritance in an object-oriented programming language allows a new component to "inherit" all of the data and operations of some existing component. The new component may contain additional data and operations for the object of the original component. No changes are made to the original component and no special provisions need be made in advance to allow its code to be inherited by a later component. In effect, this allows extensions to a reusable component without changing the part that is being reused in other programs.

This would be a convenient way to modify reusable components in a generic architecture. However, most of the effect of inheritance can be achieved through the use of generics and separate subunits.

Section 5

BENEFITS

The use of a generic architecture has a number of important benefits.

The most apparent is the potential for a significant reduction in both the time and effort required to develop and maintain the applications which use the architecture. The use of the established, tested design of the architecture can streamline the design phase in the development of later applications. For each application, there is less new code to be developed and less application-unique code to be maintained. The reliability of each application is enhanced through greater reliance on components that have been used and tested in other applications. The availability of an existing design and components should also reduce the time required to develop applications based on the architecture.

However, the benefits to be derived from the use of a generic architecture are not limited to streamlined development and lower maintenance costs.

In the workstation network example discussed earlier, the use of the same components to implement message protocols and other standards assures a degree of interoperability among the applications that would be difficult to achieve among independently developed applications.

The use of common support routines in the interactive user interface will produce a common "look and feel" to that interface across applications. This can reduce training costs and contribute to improved operator performance. It can also produce a degree of consistency in the man-machine interface that makes it easier to reassign operators to other applications which share the same generic architecture.

The need to confine application-dependent code to a limited number of components tends to make applications based on a generic architecture more adaptable to requirements which may change from user to user. For

example, changes required to adapt a system to different organizations are likely to be confined to those application-dependent components.

Hardware and operating system dependencies tend to be concentrated in the reusable components of a generic architecture. This can be an important advantage when the applications are moved to a new hardware or software environment. In most cases, modification of the reusable components containing the hardware dependencies can be done once for all of the applications using those components. Without a generic architecture, it is more likely that changes would have to be made to a larger number of application-unique components.

The rapid prototyping of new applications is easier if there is an available generic architecture for similar applications. The design and supporting code are already integrated, tested, and in place. Only the application-specific elements of the prototype need be added. The convenience of such a prototyping "platform" makes it easier to experiment with different versions of the prototype as the requirements for an application are refined.

Finally, the components of a generic architecture are designed to be adaptable in order to meet the requirements of the different applications that will use the architecture. This characteristic also makes them more adaptable to future changes in requirements or technology.

Section 6

DEVELOPMENT OF A GENERIC ARCHITECTURE

With some small changes in emphasis, the development of a generic architecture and its applications conforms well to the development process defined by DOD-STD-2167A. The applications covered by the generic architecture might be identified as separate Computer Program Configuration Items (CSCIs) in the System/Segment Specification. The generic architecture itself would constitute an additional CSCI.

The use of a generic architecture forces the software developer to give more attention to requirements analysis and design than with less coordinated approaches to software development. The level of software reuse obtained from a generic architecture is highly dependent on the amount of time and effort given to these critical early phases of the development effort.

6.1 Requirements Analysis

A thorough analysis of the requirements of a family of applications will increase the likelihood that those requirements will be met with reusable components rather than with application-specific code. Draft versions of the Software Requirements Specifications (SRSs) should be completed for each of the applications prior to the development of an SRS for the generic architecture. The application SRSs can then be analyzed to identify the common requirements that should be included in the SRS for the generic architecture.

This selection of the requirements for the generic architecture may be more complex than it might appear. Few of the requirements are likely to be stated in a way which separates the unique requirements of the individual applications from the more generic support that might be implied by those requirements. The SRS for the architecture must define the generic support requirements that are often hidden beneath the surface of the specific requirements of the applications.

It should be recognized that it may not be practical to define a single generic architecture for all of the applications. Some applications may be sufficiently different to justify a separate generic architecture or unique enough to be developed independently.

It is at this point that a decision should be made on the feasibility of a generic architecture for the applications under consideration. That decision should be based on an analysis of the anticipated benefits from its use on those applications.

6.2 Design

The design activity defines the components of each CSCI, the interfaces, and the control relationships among those components. The results are contained in Software Design Documents (SDDs) for each of the CSCIs.

The design provided by the generic architecture should be based on an analysis of the requirements of each application that separates those that might be met with reusable components from those which are unique to the application. It should then isolate application-unique functions in a limited number of non-reusable components or subunits that can be replaced from one application to the next. Requirements for functions that operate on different types of data may be met with generic components. Components of the generic architecture that are likely to be replaced by application-specific components should be designed to support the later integration testing of the architecture.

The design process is likely to be iterative as tentative designs for the architecture are proposed and revised in response to the reactions of those designing the dependent applications.

Although the design of the generic architecture and its applications can proceed in parallel, the design of the architecture should be completed first. It is then possible to describe the application designs in the context provided by the design of the generic architecture. The application designs should cover only those requirements that are not covered by the architecture and should reference the design of the architecture for the remainder.

6.3 Coding And Testing

The coding and most of the testing of the generic architecture should take place before the coding and testing of the applications. This will allow the architecture to be used to support the testing of the applications and will serve as an effective system test of the architecture as well.

Section 7

ACQUISITION CONSIDERATIONS

To this point, this paper has focused on an understanding of generic architectures and the technical issues surrounding the use of a generic architecture on a project. However, there are other issues of acquisition strategy that must be addressed in any serious attempt to implement a generic architecture.

7.1 Project Domain

The first issue is the domain, in project terms, of the architecture. It is not unusual to find that the applications of several different projects will share the same operating environment. There may even be strong requirements for the interoperability of the applications on one project with the applications of the others. This would seem to be the ideal situation for the use of a single generic architecture across all of the projects.

However, there may be a number of problems with such an approach. Organizational and funding responsibility for the different projects may reside in distinctly separate organizations. The implementation schedules for the projects may not coincide. Separate acquisition efforts may be planned which could result in the selection of different contractors for each of the projects. Finally, it may be that no single organization has the background necessary to assume the responsibility for the development of a generic architecture that would be used by all of the projects.

The domain of a generic architecture should be large enough to provide adequate opportunities for the reuse of software components within the domain. As long as that requirement is met, there are some advantages to limiting the domain to a single, well-defined project. For example, the schedule for that project would not be affected by delays experienced on other projects. Control would remain within a single organization.

Issues with respect to the requirements or design would not have to be negotiated among projects with potentially different approaches to those matters.

Such a strategy could still be of benefit to other projects. The development of a generic architecture for one project would provide a prototype or model for the subsequent development of generic architectures on other projects. Each successive architecture would be a refinement of the design and components used in earlier architectures. It is likely that many of the components would be reused in any event.

7.2 Contractor Selection

Care must be taken to structure a competitive acquisition so that it results in the selection of a contractor with the commitment and qualifications necessary to successfully implement a generic architecture for the project. Most of the benefits described earlier are important to the government, but may not be as important to the contractor responsible for the initial development effort. A contractor may be able to implement the same applications at a competitive price without a generic architecture using traditional programming practices and less qualified personnel. The problems with such an approach might become apparent only in the longer term operation and maintenance of the applications.

The key technical requirement to be included in the RFP is for extensive reuse of software components across the applications of the project. The contractor's proposal should describe how the reuse of software will be maximized. Beyond a point, greater reuse of software components can only be achieved through the use of a common high-level design, essentially the generic architecture approach. The proposal should also describe the contractor's past experience with the development of reusable components and provide evidence of a strong commitment to software reuse within the contractor's organization.

7.3 Development

The results achieved through the use of a generic architecture are directly related to the thoroughness of the requirements analysis and design activities. Although these activities are critical to the success of any software effort, they are particularly important when defining and developing components that are intended for reuse across different applications. Typically, additional time spent in requirements analysis and design avoids delays in the subsequent implementation of the software.

Unfortunately, pressure to show early results often leads to less than adequate attention to these critical activities. The early scheduling of requirements and design reviews leads to the submission of documents that are often incomplete and poorly understood. This may be followed by a cursory review of those documents that does not uncover deficiencies in the statement of the requirements and conceptual problems with the resulting design. Such practices will certainly reduce the effectiveness of the generic architecture as a source of reusable components for the applications.

The schedule for the requirements analysis and design activities should be based on industry experience with projects of similar size and complexity. Guidance may be found in sources such as Barry Boehm's "Software Engineering Economics" [1]. Additional time should be allowed for the editing of the requirements and design documents to meet military standards and for the review of the delivered documents by the program office.

The development of early prototypes of the architecture, and at least some of the applications, may absorb much of the pressure to show early results. The prototypes can be used to provide the users with a tentative response to their requirements. As such, they become vehicles for the refinement of the requirements and the development of an appreciation of critical design issues.

[1] Barry W. Boehm, Software Engineering Economics, Prentice-Hall, Englewood Cliffs, N.J., 1981.

Both the prototypes and the resulting requirements documents should give particular attention to those areas which experience has shown have the greatest potential for the types of reusable components provided by a generic architecture. These include:

- The interactive interface with the user,
- Reports produced by the applications,
- Data base requirements, and
- Communications standards and protocols.

The interactive interface should be specified in sufficient detail to completely support the development of user manuals and final qualification tests. Report specifications should be adequate to support a detailed review by the intended users of the applications. The data base requirements should be sufficiently precise and detailed to allow users to verify the definitions of all data elements and the logical relationships among those elements. The communications standards and protocols should address in detail the issues of interoperability among the applications and with other systems.

Section 8

SUMMARY AND CONCLUSIONS

The reuse of software components improves software development productivity by reducing the amount of program code that must be developed and maintained and by capturing expertise that might not be readily available on a project. The generic architecture approach to reusable software allows a significantly greater share of the code to be reused in applications which can share a common high-level design.

Generic architectures compliment the use of libraries of reusable components by expanding the use of reusable components to parts of an application which have traditionally been considered too design dependent for reusable software. Most of the components available from libraries are designed to stand alone while those developed for a generic architecture often require the services of other components developed for the architecture. The ability to achieve high levels of software reuse depends to some degree on limiting the scope of a generic architecture to the specific applications planned for that architecture.

The Ada language is well suited to the development of reusable components and generic architectures. Ada packages provide a well defined interface for a component and encapsulate the details of the implementation. Generic and separate subunit features of the language allow the operations of a component to be modified without changing the code of the component itself. The language supports an object-oriented development process which provides a logical and efficient basis for organizing operations and data into reusable components.

The benefits of a generic architecture include a significant reduction in software development and maintenance costs, greater reliability of the resulting software, improved interoperability among applications based on the same architecture, a common "look and feel" to the interface of those applications, and greater adaptability to changing requirements, environments, and technology.

The development of a project-oriented generic architecture and its applications fits well into the DOD-STD-2167A defined development process. The requirements for the architecture must be derived from the requirements for the applications supported by the architecture. The architecture may be designed in parallel with its applications, but its implementation should precede that of the applications so that it is available to supply components required for the integration and testing of the applications.

Several issues of acquisition strategy need to be addressed in the development of a generic architecture. The domain of the architecture needs to be broad enough to provide adequate opportunities for the reuse of the components provided by the architecture. It should be narrow enough to allow it to be successful on at least one project. The contractor selection process should be structured to ensure that the contractor is sufficiently qualified and committed to meet the software reuse goals of the project. The requirements analysis and design phases of the work must be carefully monitored to ensure that the results will be adequate to support the successful development of a generic architecture.

Section 9

FURTHER READING

The material in this paper is based on a study done by SofTech for the U.S. Army CECOM titled "Generic Architecture Study" dated January 22, 1988. Copies may be obtained from the Center for Software Engineering. Another paper based on the study and titled "The Generic Architecture Approach to Reusable Software" was presented at the Sixth National Conference on Ada Technology and is contained in the Proceedings, pages 390-394 (March 1988).

An excellent introduction to the use of Ada in object-oriented development is contained in an article by Grady Booch titled "Object-Oriented Development" that was published in the February 1986 issue of the IEEE Transactions on Software Engineering (volume SE-12, number 2, page 211).

An account of the use of this approach on a major project is contained in the "Phase I Final Technical Report for the Mobile Command and Control System (MCCS) Mission Support Segment (MSS)" that was prepared by SofTech for the U.S. Army (CECOM) and the U.S. Air Force Space Command/SWSC, dated February 28, 1989. It is also available at the CECOM Center for Software Engineering.