AD NUMBER

ADB102654

LIMITATION CHANGES

TO:

Approved for public release; distribution is unlimited.

FROM:

Distribution authorized to DoD and DoD contractors only; Critical Technology; SEP 1985. Other requests shall be referred to Air Force Armament Laboratory, Eglin AFB, FL 32542-5434. This document contains export-controlled technical data.

AUTHORITY

AFSC ltr dtd 13 Feb 1992

AD NUMBER

B102654

CLASSIFICATION CHANGES

TO

FROM

AUTHORITY

AFSC (WRIGHT LAB / MNOI AT EGLIN AFB, FL)

LTR DTD 13 FEB 92

AFATL-TR-85-93

Common Ada® Missile Packages (CAMP)

Volume I: Overview and Commonality Study Results

Daniel G. McNicholl
Constance Palmer, et al.

McDONNELL DOUGLAS ASTRONAUTICS COMPANY
POST OFFICE BOX 516
ST. LOUIS, MISSOURI 63166

MAY 1986

FINAL REPORT FOR PERIOD SEPTEMBER 1984 – SEPTEMBER 1985

® Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

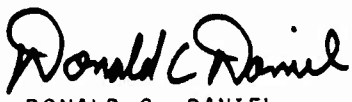
AIR FORCE ARMAMENT LABORATORY
Air Force Systems Command*United States Air Force*Eglin Air Force Base, Florida

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely related Government procurement operation, the United States Government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the Government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any potential invention that may in any way be related thereto.

This technical report has been reviewed and is approved for publication.

FOR THE COMMANDER



DONALD C. DANIEL
Chief, Aeromechanics Division

Even though this report may contain special release rights held by the controlling office, please do not request copies from the Air Force Armament Laboratory. If you qualify as a recipient, release approval will be obtained from the originating activity by DTIC. Address your request for additional copies to:

Defense Technical Information Center
Cameron Station
Alexandria, Virginia 22314

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization, please notify AFATL/FXG, Eglin AFB, FL 32542

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

The following notice applies to any unclassified (including originally classified and now declassified) technical reports released to "qualified U.S. contractors" under the provisions of DoD Directive 5230.25, Withholding of Unclassified Technical Data From Public Disclosure.

NOTICE TO ACCOMPANY THE DISSEMINATION OF EXPORT-CONTROLLED TECHNICAL DATA

1. Export of information contained herein, which includes, in some circumstances, release to foreign nationals within the United States, without first obtaining approval or license from the Department of State for items controlled by the International Traffic in Arms Regulations (ITAR), or the Department of Commerce for items controlled by the Export Administration Regulations (EAR), may constitute a violation of law.

2. Under 22 U.S.C. 2778 the penalty for unlawful export of items or information controlled under the ITAR is up to two years imprisonment, or a fine of $100,000, or both. Under 50 U.S.C., Appendix 2410, the penalty for unlawful export of items or information controlled under the EAR is a fine of up to $1,000,000, or five times the value of the exports, whichever is greater; or for an individual, imprisonment of up to 10 years, or a fine of up to $250,000, or both.

3. In accordance with your certification that establishes you as a "qualified U.S. Contractor", unauthorized dissemination of this information is prohibited and may result in disqualification as a qualified U.S. contractor, and may be considered in determining your eligibility for future contracts with the Department of Defense.

4. The U.S. Government assumes no liability for direct patent infringement, or contributory patent infringement or misuse of technical data.

5. The U.S. Government does not warrant the adequacy, accuracy, currency, or completeness of the technical data.

6. The U.S. Government assumes no liability for loss, damage, or injury resulting from manufacture or use for any purpose of any product, article, system, or material involving reliance upon any or all technical data furnished in response to the request for technical data.

7. If the technical data furnished by the Government will be used for commercial manufacturing or other profit potential, a license for such use may be necessary. Any payments made in support of the request for data do not include or involve any license rights.

8. A copy of this notice shall be provided with any partial or complete reproduction of these data that are provided to qualified U.S. contractors.

D E S T R U C T I O N N O T I C E

For classified documents, follow the procedures in DoD 5200.22-M, Industrial Security Manual, Section II-19 or DoD 5200.1-R, Information Security Program Regulation, Chapter IX. For unclassified, limited documents, destroy by any method that will prevent disclosure of contents or reconstruction of the document.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Distribution limited to DOD and DOD Contractors only; this report documents test and evaluation; distribution limitation applied September 1985.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S): AFATL-TR- 85-93
6a. NAME OF PERFORMING ORGANIZATION: McDonnell Douglas Astronautics Company; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: Aeromechanics Division, Air Force Armament Laboratory
6c. ADDRESS (City, State and ZIP Code): P. O. Box 516, St. Louis, MO 63166	7b. ADDRESS (City, State and ZIP Code): Eglin AFB, FL 32542-5434
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: STARS Joint Program Office; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: F08635-84-C-0280
8c. ADDRESS (City, State and ZIP Code): Room 3D139 (1211 Fern St.), The Pentagon, Washington, D.C. 20301-3081	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
63756A			

11. TITLE (Include Security Classification): Common Ada® Missile Packages (CAMP), Volume I:

12. PERSONAL AUTHOR(S): McNicholl, Daniel G., Palmer, Constance, Cohen, Sanford G., Whitford, William H., Goeke, Gerard O.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
FINAL	FROM Sep 84 TO Sep 85	May 1986	177

16. SUPPLEMENTARY NOTATION: SUBJECT TO EXPORT CONTROL LAWS. Availability of this report is specified on verso of front cover.

17. COSATI CODES		
FIELD	GROUP	SUB GR.

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number): Reusable Software, Missile Software, Software Generators, Ada, Parts Composition, Systems, Software Parts.

19. ABSTRACT (Continue on reverse if necessary and identify by block number): The objective of the CAMP program is to demonstrate the feasibility of reusable Ada software parts in a real-time embedded application area; the domain chosen for the demonstration was that of missile flight software systems. This required that the existence of commonality within that domain be verified (in order to justify the development of parts for that domain), and that software parts be designed which address those areas identified. An associated parts cataloging scheme and parts composition system were developed to support parts usage.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS ☒	21. ABSTRACT SECURITY CLASSIFICATION: Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
Christine Anderson	(904) 882-2961	AFATL/ FXG

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

11. TITLE (CONCLUDED)

Overview and Commonality Study Results.

3. DISTRIBUTION/AVAILABILITY OF REPORT. (CONCLUDED)

Other requests shall be referred to the Air Force Armament Laboratory (FXG) Eglin Air Force Base, Florida 32542-5434.

EXECUTIVE SUMMARY

The overall goal of the Common Ada Missile Packages (CAMP) program is to demonstrate to the DOD software engineering community that software parts which are reusable, efficient, and simple to use can be developed for embedded real-time software applications. The application area in which CAMP will demonstrate the feasibility and value of software parts is that of missile flight software systems. Like most embedded real-time applications, missile flight software is severely constrained in terms of memory and time. If software parts can be developed which meet the stringent requirements of this application area, then the utility of a parts approach to software engineering for other real-time embedded applications will have been established.

The past decade has seen the initiation of several major DOD efforts designed to reduce the cost of developing and maintaining mission critical software systems while at the same time increasing the reliability of this software. One component of most of these initiations is a move towards decreasing the amount of software that needs to be developed for new applications. An obvious way to decrease the amount of software is to develop software parts which can be reused. However, until recently it has been commonly believed that software parts were not feasible for embedded real-time applications because they could not be built efficiently enough to meet the needs of these applications. This belief has been based on the assumption that software parts must be general to be useful and that generality invariably leads to inefficiency.

With the advent of Ada, many professional software engineers believe that the time has arrived when software parts can be built which are both general enough to have wide applicability and efficient enough to meet the needs of the real-time embedded applications. The first phase of the CAMP program was designed to determine the validity of this belief.

Phase 1 of the CAMP program was a 12-month feasibility study which began in September 1984. This contract was sponsored by the United States Air Force Armament Laboratory (AFATL) and performed by the McDonnell Douglas

Astronautics Company in St. Louis (MDAC-STL). The Department of Defense (DOD) STARS (Software Technology for Adaptable, Reliable Systems) program provided partial funding for this contract.

The primary objectives of the first phase of the CAMP program were (1) to determine the feasibility of reusable missile software parts written in Ada, and (2) to determine the feasibility of an automated software parts composition system (i.e., a tool which would automate or semi-automate the process of building a new software system using existing software parts). If the parts and/or the parts composition system proved feasible, then they were to be designed during CAMP. In addition to these primary objectives, the following tasks were performed in Phase 1 of CAMP: (1) the development of a missile software parts cataloging scheme; (2) an evaluation of the utility of expert systems in the software parts engineering process; (3) an evaluation of the Japanese software reusability efforts; and (4) an evaluation of the STARS data collection forms.

The work performed, the results obtained, and the conclusions reached during the first phase of CAMP are described in detail in Volumes I through III of this technical report. Some of the major conclusions reached during the CAMP-1 contract are discussed in the subsequent paragraphs.

Sufficient commonality does exist (albeit well disguised) within missile software systems to justify the development of reusable software parts. During the CAMP domain analysis over 200 reusable missile software parts were identified. These parts range from straightforward mathematical routines to relatively complex parts for navigation and guidance functions.

Ada is well suited for building software parts which are reusable, simple to use, and protected against misuse. While Ada's advanced features (e.g., packages, overloading, etc.) are complex to master, they give the software part developer the tools he needs to develop truly reusable parts which are simple to use. All the complexity is hidden from the user behind a simple interface. The use of strong data typing allows the construction of parts which are protected against misuse.

Parts can be developed in Ada which are efficient enough for real-time embedded applications while still being reusable and simple to use. The issue of efficiency was of prime importance during CAMP as the parts designed had to be useful in an area where memory and time are both severely constrained. While the mainframe software developer might be willing to sacrifice a degree of efficiency in order to achieve an increase in software development productivity, the real-time embedded programmer cannot afford this sacrifice--he wants both efficiency and productivity. During CAMP, a method of using the advanced features of Ada was developed that would allow the construction of reusable software parts which were as efficient as equivalent application-specific Ada components.

There are significant benefits associated with reusable software parts. During CAMP an analysis was performed to determine the potential benefit of reusing software. This analysis demonstrated that relative small degrees of reuse can result in significant productivity improvements and that the availability of tools to decrease the cost of reusing parts would significantly increase software productivity.

Developing good reusable software parts is more expensive than developing customized software and required special expertise. A software part which must be reusable, efficient, and simple to use requires a great deal of Ada expertise to build. In addition, it requires insight into the application area. Experience on CAMP indicated that it would be unrealistic to expect mainstream projects to develop parts within the project because of the need for additional resources (expertise and effort). Likewise, it would be unrealistic to expect an isolated parts group to develop good parts. The ideal parts development group would be a separate team, but, would be heavily influenced and directed by ongoing projects. Although reusable parts are more expensive to develop than one-shot code, the benefits of reuse justify their development.

Automated tools can significantly increase the productivity gains achieved by reusable software parts. Part of the aforementioned benefit analysis dealt with the cost of reusing software. it was shown that small decreases in the cost of using the parts can result in large increases in

productivity. Automated tools can significantly decrease the cost of using parts. Three areas can benefit from the use of automated tools: Parts Identification, Parts Cataloging, and Component Construction.

The identification of existing software parts is a process which can, and should, be automated. The software parts identification function allows the missile software engineer to identify reusable software parts which are appropriate for his new missile application. What distinguishes this function from a simpler parts catalog (which was also developed on CAMP) is that the identification process can take place at a very early point in the missile development process. This early identification of applicable software parts is needed to facilitate missile design trade-offs, cost analyses, and sizing and timing analysis. In effect, this function allows the user to map missile requirements onto a set of parts. The implementation of this function is facilitated by the use of Artificial Intelligence techniques.

The construction of application-specific software components from parts which capture recurring patterns of logic can, and should, be automated. During CAMP-1, the feasibility of providing the user with the ability to automatically generate components of a new missile software system was demonstrated. The generation of these application-specific software components is accomplished through the use of schematic parts. A schematic part is a part which captures a recurring pattern of logic which cannot be implemented directly in Ada. In effect, a schematic part is a blueprint together with a set of construction rules which specify how the part is to be constructed in light of a given set of requirements. The CAMP parts consist of three types of parts: simple, generic, and schematic.

Expert Systems have great potential in automating the software parts engineering process. An expert system is an Artificial Intelligence system which emulates the manner in which human experts solve problems. During CAMP we constructed a proof-of-concept implementation of a schematic part constructor using the Automated Reasoning Tool (ART) which is a commercial expert system tool. We also verified that it can be used to construct the software parts catalog and the software parts identification function

previously described. By using an expert system, later phases of CAMP will be able to construct a software parts composition system which is intelligent and flexible and hence will promote the use of the CAMP parts and the best use of the software engineer's time. The automated software parts composition system that was designed during CAMP was called the Ada Missile Parts Engineering Expert (AMPEE) system. This system provides a complete set of software parts engineering functions in one tool.

(The reverse of this page is blank)

PREFACE

This report describes the work performed, the results obtained, and the conclusions reached during the Common Ada Missile Packages (CAMP) contract (F08635-84-C-0280). This work was performed by the Computer Systems & Software Engineering Department of the McDonnell Douglas Astronautics Company, St. Louis, Missouri (MDAC-STL), and was sponsored by the United States Air Force Armament Laboratory (FXG) at Eglin Air Force Base, Florida. This contract was performed between September 1984 and September 1985.

The MDAC-STL CAMP program manager was Dr. Daniel G. McNicholl (McDonnell Douglas Astronautics Company, Computer Systems and Software Engineering Department, P.O. Box 516, St. Louis, Mo. 63166); and the AFATL CAMP program manager was Christine M. Anderson (Air Force Armament Laboratory, Aeromechanics Division, Guidance and Control Branch, Eglin Air Force Base, Florida 32542).

This report consists of three volumes. Volume I contains overview material and the results of the CAMP commonality study. Volume II contains the results from the CAMP automated parts engineering study. Volume III contains the rationale for the CAMP parts.

Commercial hardware and software products mentioned in this report are sometimes identified by manufacturer or brand name. Such mention is necessary for an understanding of the R & D effort, but does not constitute endorsement of these items by the U.S. Government

ACKNOWLEDGEMENT

Special Thanks to the Armament Division Standardization Office and to the Software Technology for Adaptable, Reliable Systems (STARS) Joint Program Office for their support of this project.

(The reverse of this page is blank)

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Concluded)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Concluded)

ACRONYMS

ACM	Association for Computing Machinery
ADL	Ada Design Language
ADRAT	Air Data Rationale
AFATL	Air Force Armament Laboratory
AGM	Air-to-Ground Missile
AI	Artificial Intelligence
AIAA	American Institute of Aeronautics and Astronautics
AMPEE	Ada Missile Parts Engineering Expert (System)
ANSI	American National Standards Institute
APRAT	Autopilot Rationale
ART™	Automated Reasoning Tool
AXE™	Archetype Xenoclause Embedding (Language)
CACM	Communications of the ACM
CAINS	Carrier Aircraft Inertial Navigation System
CAMP	Common Ada Missile Packages
CIRIS	Completely Integrated Reference Instrumentation System
COBOL	Common Business-Oriented Language
COCOMO	Constructive Cost Model
CPU	Central Processing Unit
CSC	Computer Software Components
CSCI	Computer Software Configuration Item
CWCS	Common Weapons Control System
DACS	Data and Analysis Center for Software
DARTS™	Development Arts for Real-Time Systems
DB	Database
DBMS	Database Management System
DDD	Detailed Design Document
DE	Domain Engineer
DEC	Digital Equipment Corporation
DID	Data Item Descriptor
DIS	Digital Integrating Subsystem
DOD	Department of Defense

ACRONYMS (Continued)

DSMAC	Digital Scene Matching Area Correlation
EROS	Embedded Real-Time Operating System
FCRAT	Fuel Control Rationale
FIFO	First-In-First-Out
FIPS	Federal Information Processing Standards
FORTRAN	Formula Translation
FSM	Finite State Machine
GAN	Guidance and Navigation
GANP	Guidance and Navigation Program
GPS	Global Positioning System
HOL	High Order Language
HOS	Higher Order Software, Inc.
IAF	Interactive Application Facility
IBM	International Business Machines
ICP	Interface Control Panel
IEEE	Institute for Electrical and Electronic Engineers
IIR	Infrared Imaging Radar
IMSL	International Math and Statistical Library
ISA	Inertial Sensor Assembly
ISO	International Standards Organization
JCM	Joint Cruise Missile
JCMPO	Joint Cruise Missile Program Office
JDA	Japanese Defense Agency
JOVIAL	Jules' Own Version of the International Algebraic Language
KB	Knowledge Base
KBMS	Knowledge-Based Management System
KBSA	Knowledge-Based Software Assistant
KFRAT	Kalman Filter Rationale
LADDER	Language Access to Distributed Data with Error Recovery

ACRONYMS (Continued)

LCIGS	Low Cost Inertial Guidance System
LISP	List Processing Language
LLCSC	Lower Level Computer Sosftware Component
LOC	Lines of Code
MDAC	McDonnell Douglas Astronautics Company
MDAC-STL	McDonnell Douglas Astronautics Company - St. Louis
MDC	McDonnell Douglas Corporation
MGD	Midcourse Guidance Demonstration
MIL-STD	Military Standard
MIT	Massachusetts Institute of Technology
MODEL	Module Description Language
MRASM	Medium Range Air to Surface Missile
NAG	Numerical Algorighms Group, Inc.
NISO	National Information Standards Organization
NL	Natural Language
NTIS	National Technical Information Service
PAM	Process Abstraction Method
PCE	Parts Construction Expert
PDL	Program Design Language
PIE	Parts Identification Expert
PSL	Problem Statement Language
(PSL/)PSA	Problem Statement Analyzer
QA	Quality Assurance
RSL	Requirements Specification Language
RTP	Requirement and Test Procedure
SAFE	Skills Acquisition from Experts
SDS	Software Development Specification

ACRONYMS (Concluded)

SE	Software Engineer
SED	System Engineering Documentation
SEP	Software Engineering Practice
SGS	Software Generation System
SINP	Strapdown Inertial Navigation Program
SL	Specification Language
SOW	Statement of Work
SQL	Structured Query Language
SRAM	Short Range Attack Missile
SREM	Software Requirements Engineering Methodology
SRS	Space Relay System
SRS	Software Requirements Specification
STARS	Software Technology for Adaptable Reliable Systems
TASM	Tomahawk Anti-Ship Cruise Missile
TERCOM	TERrain COrrelation Matching
TLAM	Tomahawk Land Attack Missile
TLCSC	Top Level Computer Software Component
TLDD	Top Level Design Document
USC	University of Southern California
UTG	Unaided Tactical Guidance
UTGV	Unaided Tactical Guidance Validation
VHOL	Very High Order Language
WHC	Warhead Control Commonality
WPSRAT	Waypoint Steering Rationale
WSMR	White Sands Missile Range

SECTION I
INTRODUCTION

1. PURPOSE

This report contains a description of the work performed on the first phase of the Common Ada Missile Packages (CAMP) program. This was a 12-month feasibility study sponsored by the United States Air Force Armament Laboratory (AFATL) and performed by the McDonnell Douglas Astronautics Company in St. Louis (MDAC-STL). The CAMP program was performed during the period of September 1984 through September 1985. The CAMP program was partially funded by the Department of Defense (DOD) STARS (Software Technology for Adaptable, Reliable Systems) program.

The overall goal of the CAMP program is to demonstrate to the DOD software engineering community that reusable software parts can be practical in embedded real-time software applications such as missile software systems. The first phase of CAMP had two specific objectives: to determine the feasibility of reusable missile software parts written in Ada; and, to determine the feasibility of an automated software parts composition system.

In addition to these primary objectives, the CAMP program involved the performance of the following tasks.

a. The development of a missile software parts cataloging scheme.
b. An evaluation of the utility of expert systems in the software parts engineering process.
c. An evaluation of the Japanese software reusability efforts.
d. An evaluation of the STARS data collection forms.

2. BACKGROUND

During the past 10 years, DOD and the military services have become increas- ingly sensitive to the critical role that software plays in mission critical systems. These organizations have been forced to accept the existence of a software crisis by the realization that software has been a major source of problems in defense systems. The software crisis is characterized by: (1) rapidly escalating software development and maintenance costs, (2) delays in deployment of new defense systems due to ever expanding software development schedules, (3) restrictions on the number of programs which can be developed concurrently due to a shortage of software expertise, and (4) reliability problems with deployed defense systems due to poor software quality.

The basic cause of the software crisis is the tremendous increase in the demand for software, and the explosive growth in the size, complexity, and critical nature of modern software systems. This crisis has resulted in a situation where existing software methods and tools are antiquated, software engineers are under-trained and over-committed, and software project managers are too busy handling the crisis to develop sound software management procedures. Obviously such a complex problem has no one solution, but concrete initiatives do exist which, if taken, will alleviate the current problems. While it is beyond the scope of this report to discuss all the initiatives which are being proposed, there is one common factor which can be found in most of these initiatives--the reuse of software parts. The CAMP program was motivated by the potential of reusable software.

3. SOFTWARE REUSABILITY

Simply stated, software parts are pre-built software components which have been explicitly designed to be used (reused) in multiple applications within a given application area (see Figure 1). Although one typically thinks of only code as being reusable, this is not the case. Later in this report it shall be shown that a software part can be any software entity which is capable of reuse (e.g., a software design component, a series of software tests, etc.).

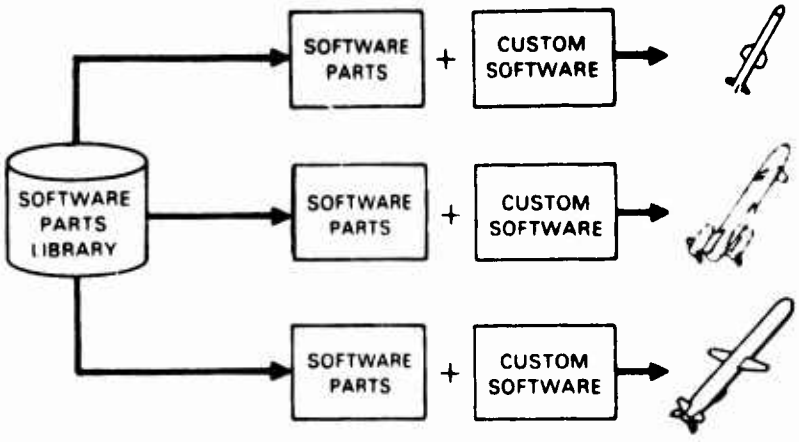


Figure 1. The Use of Software Parts in Missile Software Systems

a. The Benefits of Software Reuse

The potential benefits of a parts approach to software engineering are quite significant. Software reuse can:

(1) Increase software development and maintenance productivity,
(2) Lead to more reliable software, and
(3) Help conserve critical software engineering expertise.

The most obvious benefit of reusing software is that less code needs to be developed and therefore less time and money is required for the development of new software systems. Even a modest degree of software reuse can result in significant cost savings. Figure 2 depicts the effect of software reuse on software development productivity.

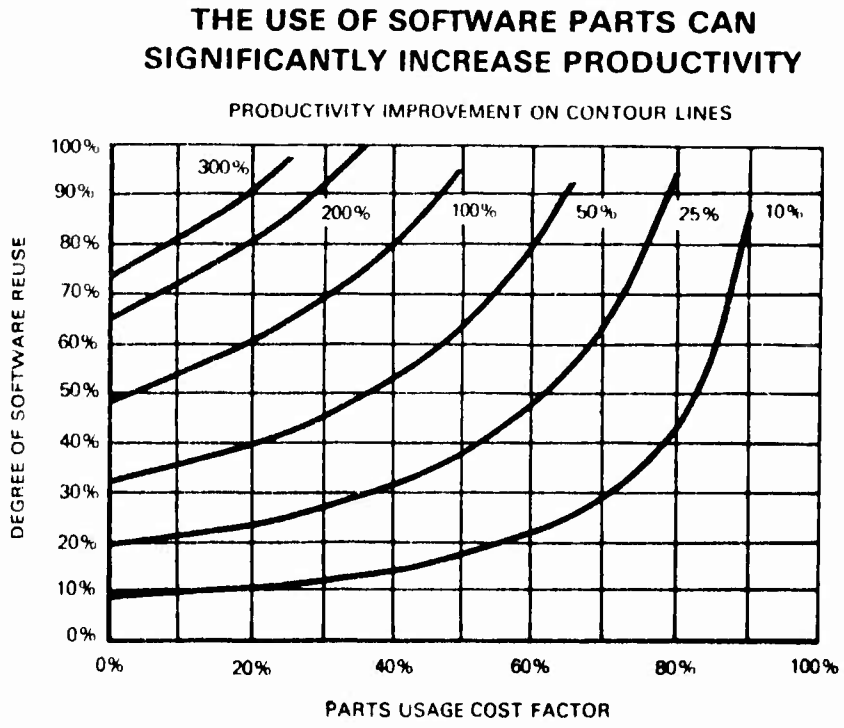


Figure 2. The Effect of Software Reuse on Development Productivity

It can be seen from Figure 2 that the relationship between the amount of software reused and the resulting productivity improvement is dependent upon a parts usage cost factor. This cost factor captures the cost of finding the right parts, analyzing the parts, and developing the application software to interface with the parts as a percentage of the cost to develop the software parts from scratch. Figure 2 shows that if one could build 40 percent of a new software system using available parts, and the cost factor was 20 percent (i.e., reuse of parts costs 20 percent more than non-reuse), then a 50 percent increase in development productivity could be achieved.

Obviously there has to be a limit to the amount of productivity one can gain from using parts. Figure 3 depicts the amount of productivity improvement (as a function of the cost factor) that can be expected if 100 percent of a software system was constructed from parts.

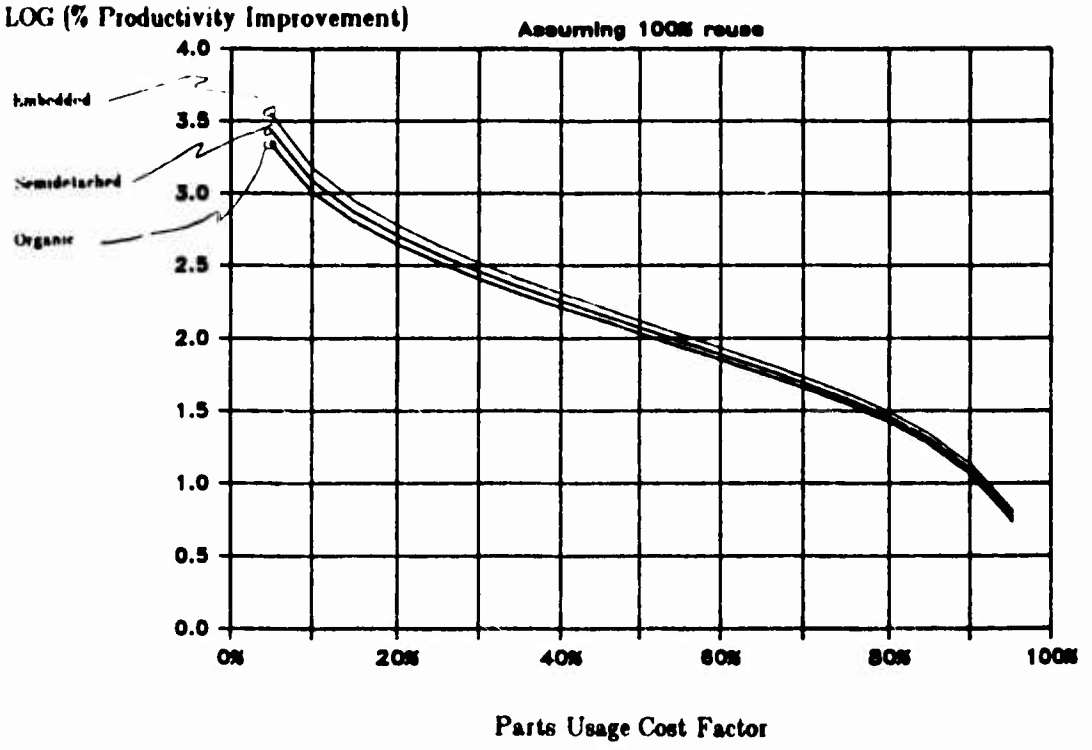


Figure 3. The Productivity Gained With 100 Percent Software Reuse

According to Figure 3, if an embedded software system was completely constructed from software parts (i.e., 100 percent reuse), and the cost factor was 55 percent then a productivity improvement of 100 percent could be expected. The rationale behind Figures 2 and 3 is discussed in Section V. Since maintenance costs (i.e., the cost to correct software errors, modify the software to a new environment, and expand the capabilities of the software) often greatly outweigh the development costs of software, a reduction in the amount of code to be maintained can result in drastically lower product life cycle costs. As with any software, software parts will need to be maintained. There are two factors which indicate that the maintenance of software parts will be much cheaper than the maintenance of custom written software: the cost of maintaining the parts is amortized across multiple projects; and, errors in software parts will be found much earlier than errors in customized software since the parts are receiving more use. The earlier an error is detected the cheaper it is to correct that error.

If software parts are rigorously tested before they are cataloged for later reuse, then the reliability of the new software systems will be increased. This is especially important since the impact of software errors in parts is much higher than for customized software since the errors will affect more programs. Without vigorous testing, parts developed by different people at different times, with differing degrees of reliability will not be trusted.

The required staffing for a software development and/or maintenance project will be decreased, assuming a significant level of software reusability. Given the shortage of software engineers that exists throughout the industry, this is a major advantage. If the parts include functions which typically require a high degree of application expertise (e.g., Guidance & Control), then a project can perform the same development with fewer software engineers experienced in the application area.

b. The Barriers to Software Reuse

Given all the aforementioned benefits, it is only natural to wonder why software has not been reused in the past. The answer to this question is that it has, but only to a very limited degree. Almost all software systems have incorporated certain types of software parts. The most common type of part has been the mathematical part (i.e., a routine from a math library). Yet if this type of low-level reusability was all that could be hoped for, the benefits discussed earlier would not be fully achievable.

There are several reasons why the software engineering community has not been able to achieve a meaningful level of software reuse in the past.

(1) Our programming languages have not had the facilities to support software reusability.

(2) We have not invested the time and effort to identify the commonality in software systems.

(3) It has not been demonstrated that software reuse can work in embedded, real-time software applications.

(4) We have not encouraged and/or required our software developers to reuse software parts.

With the advent of Ada, there is now a computer programming language which was explicitly designed with the goal of software reusability in mind. Specifically, Ada possesses facilities for: transporting programs across machine and operating system boundaries; enforcing the design and construction of autonomous software units with clean, well designed interfaces; and, developing software parts which are generic in nature and which can be tailored, using the Ada language itself, for a particular application.

One of the major barriers to an effective software reusability program is the need to conduct an in-depth domain analysis of the application area in which the software is to be reused. A domain analysis is an examination of a specific application area which seeks to identify common operations, objects, and structures which are candidates for software parts. Domain analyses are not cheap to perform. They require an intensive examination of existing software systems within the application area being studied, and personnel skilled both in modern software development techniques and in the application area. Yet, to attempt to start a software reusability program without adequately performing this analysis is as foolish as attempting to design a software system without performing an analysis of the software requirements.

It has long been a belief of the real-time, embedded software engineering community that reusable software parts could not work in their application area. This belief has been based on the assumption that software parts, by necessity, must be general enough to allow broad use. The implication of this observation is that efficiency is lost when generality is sought. In the past this implication has been true, however this no longer need be true. There are two factors which have changed the picture: the advent of modern compilers with advanced optimization features; and the advent of Ada which allows generality without high losses in efficiency.

Modern compilers can now detect and correct inefficiencies introduced in source programs in order to obtain generality. For example, most modern compilers will allow a subroutine to be expanded inline thereby allowing the subroutine to be reusable without incurring the overhead of a subroutine call. The Ada generic facility allows the creation of tailorable software parts which provide a good degree of generality while still being efficient. For example, a single generic package can be built which would be capable of buffering any data object without any loss in runtime efficiency.

One of the thorniest issues which arises in every reusable software effort and which can cause a total failure of the effort is the need to enforce the reuse of parts. Without reuse, the development of reusable software parts becomes an exercise in futility and any additional cost incurred to develop these parts (as opposed to one-shot code) cannot be amortized. Simply stated, programmers do not like to reuse software because: programmers feel that reusing parts lessens their creative role in the development of software systems; they have little faith in the correctness of reusable parts; they are not aware of the existence of reusable parts; they find the software parts difficult to understand and/or reuse in comparison to developing new software, and they feel that they can build a better part.

The key factors in overcoming the reluctance of programmers to reuse pre-built software parts are discipline, knowledge, tools, and management commitment (see Figure 4).

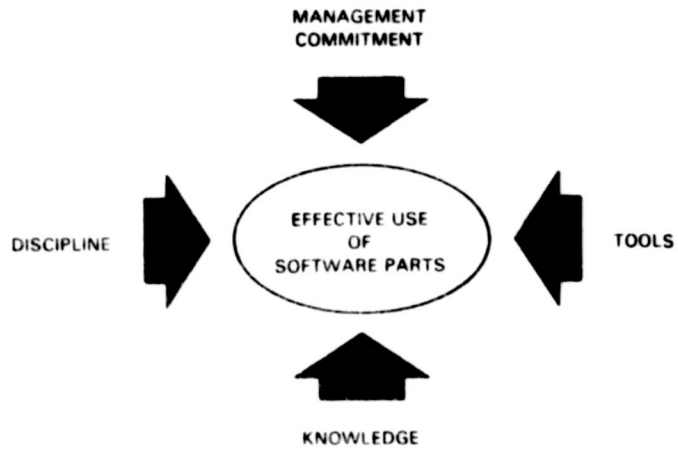


Figure 4. The Keys to Overcoming Programmer Reluctance to Reuse Software

Discipline: A successful software parts program must involve the imposition of a high degree of parts usage discipline within the organization. This discipline must be enforced by reviews and audits.

Knowledge: Programmers must also have the knowledge that parts exist and that they have been validated. Just as hardware designers are expected to know which parts are available, professional software engineers should be expected to know about software parts.

Tools: Automated tools are an essential aspect of a software reusability program. They free the software engineer from mundane, mechanical chores associated with using parts, thus increasing productivity. These tools should facilitate the retrieval of appropriate parts, the generation of new parts, the composition of software systems with existing parts, and a wide variety of other functions relating to parts usage.

Management Commitment: Managers will support a software reusability program only when it becomes apparent that reuse will cut costs and increase profits; DOD reuse incentives and/or the removal of reuse disincentives to contractors could significantly impact management commitment.

c. The Challenge

The reuse of software parts offers the promise of dramatic increases in software development and maintenance productivity. Yet this promise can only be achieved if an organization is working in an application area which has a significant degree of commonality, sufficient time and effort are invested to exploit this commonality by developing parts, and the tools and methods needed to enforce a software parts engineering discipline are put in place.

4. ASSUMPTIONS

The CAMP program was based on several assumptions which are believed to be true but cannot be proven true at this time.

a. Ada will be the primary language used to develop future missile software systems.
b. Ada compilers will produce object code sufficiently efficient for missile software systems.
c. Since commonality exists in current missile software systems, then it will exist in future missile software systems.

Although the use of Ada on mission critical system has been mandated by DOD directive, the reality of the situation is that Ada will only be used if compilers and associated tools are available and efficient. At the current time, great progress is being made on efficient Ada compilers and support tools are in development. The compilers which are being developed compare very well to the best compilers for more established Higher Order Languages such as FORTRAN.

Missiles, like most other products, tend to evolve in generations, (see Figure 5). As will be discussed in subsequent parts of this report, the CAMP project examined current missile software systems and to some degree the trends which were known to exist which might affect future missile software systems. However, the implication that if commonality exists in current missiles then commonality will exist in future missile cannot be proven.

5. UTILITY GOALS

Because of the relative immaturity of the software engineering discipline, the sofiware industry has been plagued with tools and methods which are less than useful. Frequently it seems as if people are busy developing solutions for theoretical rather than real problems. In order to avoid the occurrence of this situation in the CAMP program, a number of goals were established for both the parts development task and the automatic tool definition tasks.

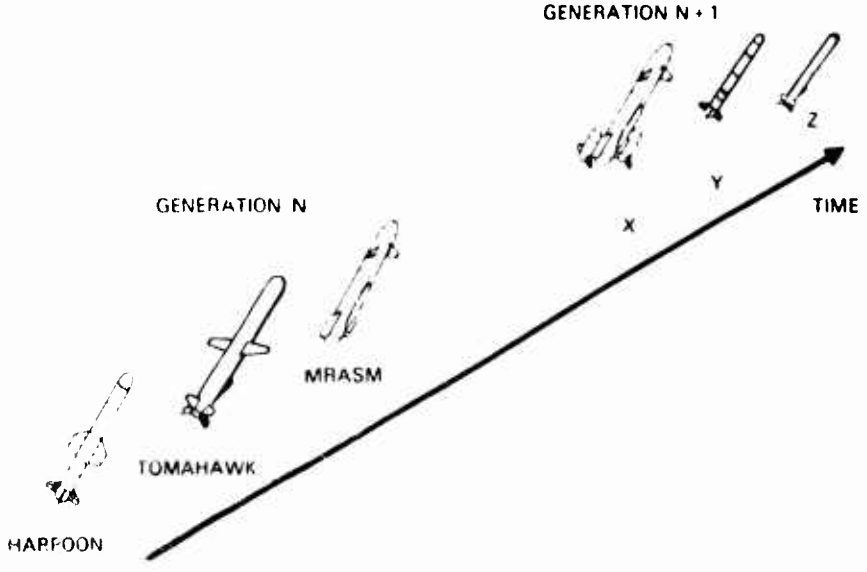


Figure 5. The Evolution of Missile Software Systems

a. Parts Utility

The following goals were set to establish the utility of software parts identified in the CAMP project.

(1) A software part must provide a useful function to more than one potential application.
(2) The use of parts should result in little, if any, loss of runtime efficiency.
(3) The use of parts should be simple.
(4) The use of parts should allow flexibility.

In the CAMP context, a software part implicitly implies that it is intended for reuse. However, great care must be taken to avoid developing parts which are not useful to a sizable application area. For example, a generic part which would perform the function X = A + B - C for any A, B, or C might have the potential of high use, but would not be useful. On the other hand, developing parts which have little possibility of being used more than once is also futile.

The use of a Higher Order Language such as Ada implies that some (hopefully a small) amount of runtime efficiency is given up in order to achieve higher software development and maintenance productivity. In embedded, real-time applications it would not be acceptable to give up another significant degree of efficiency to obtain reusability. For this reason the goal was set that any software part developed on CAMP must be as efficient as a custom written Ada component developed for a particular application by an expert Ada programmer.

One of the pitfalls in developing reusable software parts is to strive so hard for generality that the part requires too much information from the user and is therefore difficult to use. On CAMP it was established that simplicity would not be sacrificed for power. Fortunately, through the appropriate use of defaults in Ada (e.g., forcing Ada to look at the current context for an overloaded operator), the capability of developing a part which is both simple and powerful exists. However, the difference between making a part which is simple or powerful and making a part which is simple and powerful is a substantial amount of extra work.

Another common pitfall in the development of reusable parts is to build into the parts a bias towards one way of solving a problem. CAMP established the goal of multiple levels of part usage. This goal states that if in order to meet future requirements, it is envisioned that the operations or objects that a part provides can be arranged differently (but the arrangements are unknown at the current time), then the subparts of that part should be directly usable by the end-user (see Figure 6).

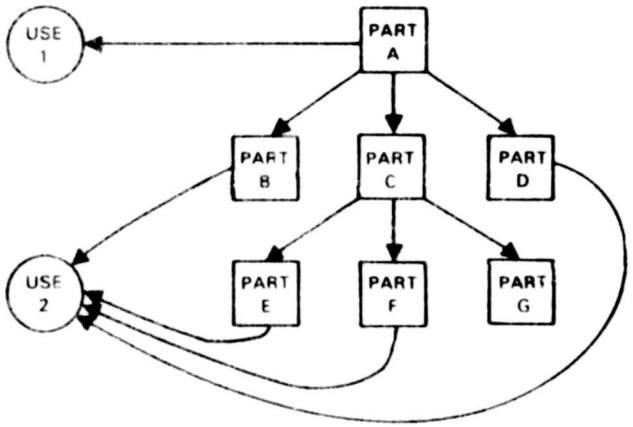


Figure 6. Multiple Level Part Usage

b. Automation Tool Utility

The following goals were set to establish the utility of any automated tool proposed in the CAMP project.

(1) Automation tools should be based on feasibility and value.
(2) Automation tools should be usable by the average software engineer and in some cases by the domain engineers (i.e. a non-software engineer).
(3) The use of esoteric technologies should be avoided when proven technologies can solve the problem as efficiently and as effectively.
(4) Automation tools should simplify the process of using complex Ada features.

CAMP was chartered to examine existing and emerging technologies which could be used to automate or semi-automate the process of developing, using, and managing software parts. One of the pitfalls in this type of work is to develop a tool which while feasible, is not practical. The goal was set on CAMP that the proposed tool must be capable of being constructed so it could be used by the average software engineer to perform actions which needed to be performed and perform these actions more efficiently and/or effectively than the software engineer could.

Researchers typically like to use the newest gadget to solve problems. On CAMP it was established that proven technologies would be used wherever possible. For example, although a software parts catalog could be established using an expert system, it could also be developed using classical data base management systems. On the other hand, when the requirements are to provide advanced cataloging features, and integrate the catalog with a software generation system, an expert system might be needed.

The fact that Ada is a very powerful language is an advantage and a disadvantage. Ada allows programmers to do things which they could not do before, but at the cost of having to learn how to properly use quite complex Ada facilities (e.g., tasking, generics, etc.). CAMP established the goal of developing tools which would allow average missile software engineers to exploit the power of Ada's more complex facilities, without having to master all their intricacies.

6. ORGANIZATION OF THE REPORT

Because of the amount of material to be covered, this report has been divided into three volumes. Volume I contains overview material and details on the CAMP commonality study. Volume II contains the detailed result of the CAMP software parts composition system study. Volume III contains details on the justification for the parts identified in the CAMP study.

In addition to this final technical report, a number of other documents were prepared during the CAMP program.

a. A software requirement specification for the CAMP parts.
b. A software requirement specification for the Ada Missile Parts Engineering Expert (AMPEE) system.
c. A top-level design document for the CAMP parts.
d. A top-level design document for the AMPEE system.
e. A draft detailed design document for a representative sample of the CAMP parts.

These documents were all prepared in accordance with the Defense System Software Development Standard (DOD-STD-2167). Since this program is one of the first to produce documents in accordance with this new DOD standard, comments regarding its use are provided in Appendix D.

SECTION II
THE DOMAIN ANALYSIS

1. INTRODUCTION

The first step in determining whether reusable missile software parts were feasible was to conduct a domain analysis. A domain analysis is an investigation of a specific application area which seeks to identify the operations, objects, and structures which commonly occur in software systems within this area. Note that unlike a classical systems analysis, a domain analysis is not limited to one system, rather, it examines all systems of a certain type--which in the case of the CAMP project are missile flight software systems.

The concept of a domain analysis is not new. It is the cornerstone of the Draco system (References 1 and 2) and has been acknowledged by leading software engineers as the most difficult part of establishing a software reusability program. This task has been described by L. Belady of the IBM Watson Research Center as:

> "... tediously studying complex and often structurally obsolete software - not considered a pleasant or even respectable activity today" (Reference 3).

It is important to note that the existence of commonality within the missile flight software application area could not be taken for granted. Although significant degrees of commonality have been found in other application areas, there was no guarantee that such high levels of

commonality existed in the realm of missile software. The Missile Systems Division of the Raytheon Company has had a good deal of success with reusing software. However, the application domain in which this work was performed was not missile software; rather, it was business-oriented information processing systems written in COBOL. They report (Reference 4) that 40 percent to 60 percent of their programs' code was found to be repeated in more than one application, and they have standard software parts to take advantage of this fact.

Few organizations or researchers have conducted an in-depth domain analysis of embedded real-time software systems. The primary reason for this seems to be that investigators were put off both by the complexity of the software and the application expertise required. For example, S. Sundfor (References 5 and 6) started to evaluate the applicability of Draco (a particular software generation system) for real-time software systems, but instead, ended up using it on a graphic-based application. The point to be made here is that a domain analysis of embedded real-time software systems must involve both software engineering expertise and application domain expertise. The domain analysis for CAMP met these criteria because it involved knowledgeable software engineers as well as expert engineers from other missile systems areas (e.g., guidance and control, etc.).

Domain analyses are not cheap to perform. They require an intensive examina- tion of existing software systems within the application area being studied, and personnel skilled both in modern software development techniques and in the application area. Yet, to attempt to start a software reusability program without adequately performing this analysis is as foolish as attempting to design a software system without performing an analysis of the software requirements (unfortunately both these situations occur more often then we like to think).

There are three steps involved in the performance of a domain analysis:

a. The Domain Definition,
b. The Domain Representation, and
c. The Commonality Study.

Each of these activities are discussed in paragraphs 2 through 4. Paragraph 5 discusses various classifications of parts and types of commonality. Paragraph 6 lists several general observations made during the CAMP domain analysis. Paragraph 7 lists the parts identified during the CAMP domain analysis.

2. THE DOMAIN DEFINITION

Domain definition is the process of determining the scope of the domain analysis (i.e., what application area will be examined). Although this appears to be a straight-forward task, there are several factors which introduce complexity.

a. Fuzzy domain boundaries
b. Domain Overlaps
c. Domain Intersections

Domains (like so many other things) do not have rigid boundaries. What one person might firmly believe to be within a domain another person would not. It is extremely important that, early in the domain analysis, time and effort be expended to identify these areas of contention and that a clear definition of functional areas be defined for the domain in question. In the case of CAMP, the AFATL established a general definition of the domain which was then refined by MDAC-STL.

The problem of overlapping domain has to do with the fact that some functional areas within the domain under examination (e.g., missile flight software systems) might also be legitimately within another domain (e.g., avionic flight software). The presence of the overlap by itself is not a problem (in fact, it is a benefit since the work done for one domain can be applied at no additional cost to the other domain). The problem is a matter of losing sight of the original domain (i.e., being sidetracked into the overlapped domain). The solution to this problem is to establish safeguards (e.g., periodic reviews) which will ensure that the domain analysis is addressing issues in the domain of interest.

A similar problem occurs when a functional area within the domain under examination applies to a wide spectrum of domains. Figure 7 depicts this situation.

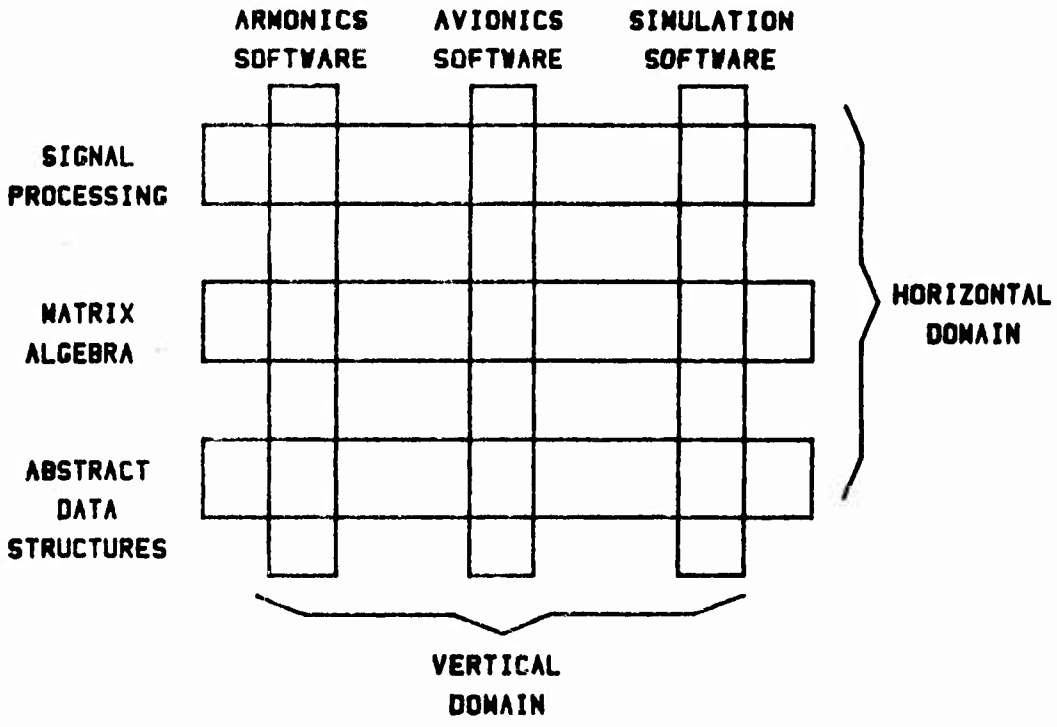


Figure 7. Vertical and Horizontal Domains

A Vertical Domain is an application-oriented grouping of software systems. A Horizontal Domain is an application-independent grouping of software. CAMP was a vertical domain, namely armonics (armament electronics), which overlapped with quite a number of horizontal domains (e.g., signal processing, matrix algebra, etc.). The problem which arises is the same as the previous overlap situation--being sidetracked. It is easy to be seduced into identifying software parts not because they are needed within the domain under examination, but because they are a natural part within the horizontal domain. In a situation where unlimited resources are available for a domain analysis this might not be viewed as a problem. If it is a problem, the answer is to establish managerial controls (e.g., reviews).

3. THE DOMAIN REPRESENTATION

Domain representation involves the selection of a set of applications which will serve to characterize the domain under investigation. In the ideal (but not too realistic) situation, all applications which were classified as being in the domain would be in this set. However, practical constraints on time and effort and the limitation of humans to deal with too much data demand a subset of the universe of discourse.
There are several factors which influence this application selection process.

a. The availability of information concerning the applications.
b. The quality of the information.
c. The representativeness of the applications.

In the case of software systems, the availability of application information pertains to software documents. In other words, can the domain analysts get access to the requirements specifications, design specifications, and code of the systems? In addition to the availability of these documents, their quality is a major issue. Many old software systems were not documented well. As such, they are not good candidates for the domain application set unless the people who developed the system are still around and willing to spend some time with the domain analyst.

It is extremely important that the set of applications which is intended to represent the domain is indeed representative of that domain. For example, it would not make sense to select only anti-ship missile applications if the domain is all missiles.

The CAMP contract required the selection of ten missile software systems for which sufficient documentation existed to perform the commonality study. This set of missile software systems also had to cover at least two of the following types of missiles: air-to-air, air-to-surface, surface-to-air, and surface-to-surface. In addition, the set of missile software systems had to include the functional areas listed in Figure 8.

Area	Subarea	Function
Navigation	Optimal Estimation	Covariance propagation
		Coupled/Uncoupled
		Kalman filters
	Strapdown Navigation	Quaternion Processing
Guidance	Guidance Laws	Pursuit Guidance
		Proportional Guidance
		Optimal Guidance
Control	Autopilot	Digital Filters
	Engine Management	Pulse Motor Logic
	Antenna Control	
Signal Processing	Spectral Analysis	Fast Fourier Transforms
Fuzing	Optical	Optimized Time
	Active	Delay
	Semi-Active	
Weapon/Aircraft		
Avionics Interface	Weapon Initialization	Inertial Systems
		Transfer Alignment

Figure 8. The CAMP Functional Areas

The set of missile software systems chosen by the CAMP project is shown in Figure 9. These missile software systems are described in more detail in Appendix A. References 7 through 29 list the software documentation that was used.

(1) Flight software for the Medium Range Air to Surface Missile (AGM-109H)
(2) Flight software for the Medium Range Air to Surface Missile (AGM-109L)
(3) Unaided Tactical Guidance Project Strapdown inertial navigation program
(4) Guidance and navigation program for the Midcourse Guidance Demonstration
(5) Flight software for the Tomahawk Land Attack Missile (BGM-109A)
(6) Flight software for the Tomahawk Anti Ship Missile (BGM-109B)
(7) Flight software for the Tomahawk Land Attack Missile (BGM-109C)
(8) Flight software for the Tomahawk Land Attack Missile (BGM-109G)
(9) Flight software for the Harpoon Missile (Block 1C)
(10) Safeguard Spartan missile

Figure 9. The CAMP Domain Application Set

4. THE COMMONALITY STUDY

The commonality study is the portion of the domain analysis concerned with identifying common operations, objects, and structures which are candidates for construction as reusable software parts. This analysis is very similar to the analysis performed in the construction of an expert system. Both these analyses are attempts to formalize a body of knowledge. In the case of an expert system the knowledge is needed to construct an artificial expert. In the case of a reusability domain analysis the knowledge is needed to distillate abstract requirements from concrete instances of requirements.

The raw data for this analysis comes from software documents. This includes requirements specifications, design specification, and, in the worst case, code listings. On the surface it would appear that only requirement specifications would be needed. After all, these specifications document what the application is to do. While a large number of the software parts are able to be identified from these requirements, the design specification also plays an important role. In addition to providing more details about the system (which is useful in understanding the requirement) a significant number of parts cannot be identified from the requirements

alone. These type of parts provide internal functions which are not directly mappable back to the external requirements of the system. Code listings usually are not a useful data source because of their level of detail. However, they frequently are needed to help understand the requirements and design specification.

The actual performance of the commonality study is best performed by means of what we call the functional strip method. In this method, the analyst examines a particular application function across all the applications in the domain representation set. For example, the analyst would investigate how the strapdown computations were implemented in all the missiles. The more narrow these functional strips are, the easier it is to detect commonality. However, if the strips are too narrow, then some higher level commonality might be overlooked. We found in CAMP that data flow modeling was a good technique for representing the functions of each of the missiles in a manner which would highlight their similarities. Two barriers which need to be overcome in this analysis are arbitrary differences and notational differences.

In some cases, two systems will perform an operation differently for no valid reason. In others words, they could have used the same operation. The determination that an operational difference is arbitrary is not a conclusion which should be reached casually. It is very easy to mistakenly call a difference arbitrary because of a lack of understanding. If the difference is really arbitrary, one (or both) operations can be candidate for parts.

A major problem with detecting commonality in embedded real-time system is the fact that different applications will use different notation and/or different algebraic formulation for the same operation. To the software engineer not well versed in the application two names or equations which are almost the same present a dilemma--are they the same or not.

5. PARTS CLASSIFICATION

During the CAMP commonality study, it was recognized that commonality existed at several different levels. Three such levels are depicted in Figure 10.

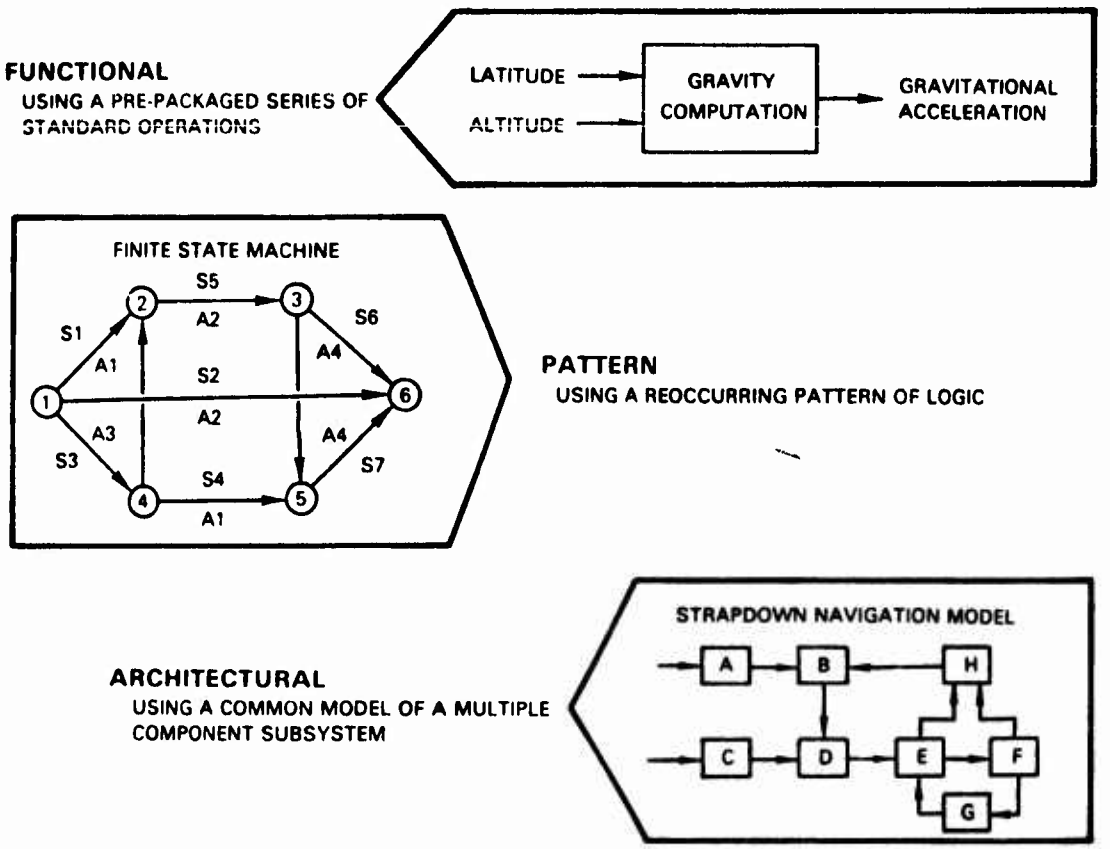


Figure 10. Three Levels of Commonality

Functional commonality involves a black box view of common operations (i.e., packing a series of standards operations). This is the type of commonality classically associated with software reusability. The equations for computing gravitational acceleration are an example of functional commonality.

Pattern Commonality involves the recognition that there are common patterns of logic that recur time and time again in missile software systems which are more complex than simple black box functions. A finite state machine is an example of pattern commonality. This pattern appears in missile software systems in the platform caging function, the launch platform interface function, and many other functional areas.

Architectural Commonality involves the recognition that there exists common models of a major component of a missile software system. This type of commonality is similar to pattern commonality, except that it is on a larger scale. A Strapdown Navigation subsystem, or a pitch autopilot is an example of architectural commonality. At an appropriately high level of abstraction there exists a common model for both of these subsystems. One clue that architectural commonality exists is the existence of standard pictures of a process in text books on the subject in question.

The presence of multiple levels of commonality has a major impact on the way in which a domain analysis is performed. If domain analysts look only for similar equations and operations at a low level, they are likely to overlook the existence of commonality at the higher levels. For this reason, the analysts must be sensitive not only to the details of the systems they are studying, but also to abstractions of which the systems under examination are instances.

During the CAMP domain analysis we recognized that two classes of parts were being identified--domain dependent parts and domain independent parts. **Domain dependent** parts provide operations and objects which are applicable only to the domain being investigated (and perhaps to other very closely related area). In the CAMP domain analysis, domain dependent parts are applicable to missile software systems (in some cases these parts could also be useful to the closely related avionics area). **Domain independent** parts provide operations and objects which, while highly relevant to the domain under examination, are also useful to other application areas. The math parts which were identified during CAMP are examples of domain independent parts. They are needed in the missile software domain, but they are also useful in other real-time embedded software systems.

Given the large number of parts typically identified during any domain analysis, it is often useful to develop some type of Software Parts Taxonomy. A software parts taxonomy is a means of classifying parts which can help the domain analysts organize their work. The taxonomy developed during CAMP (which was a very dynamic entity) is shown in Figure 11.

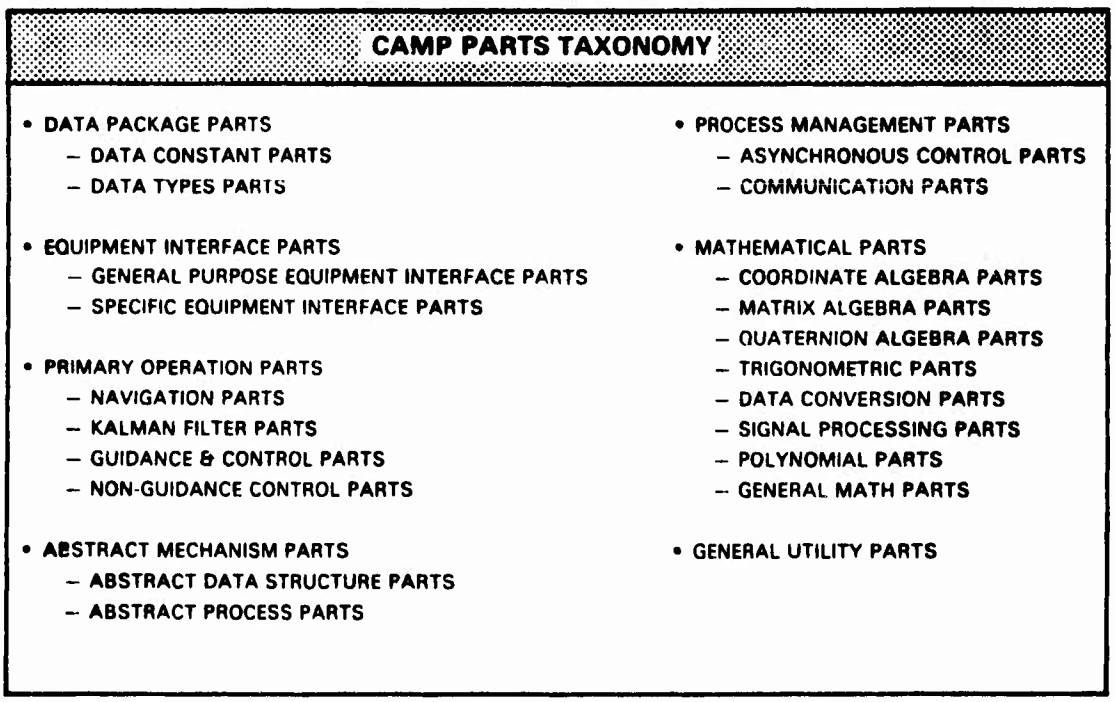


Figure 11. The CAMP Software Parts Taxonomy

Appendix B describes the taxons (i.e., the individual classes) within this taxonomy.

Early in the domain analysis, it was recognized that we were identifying three types of parts based on how the parts would need to be constructed (see Figure 12).

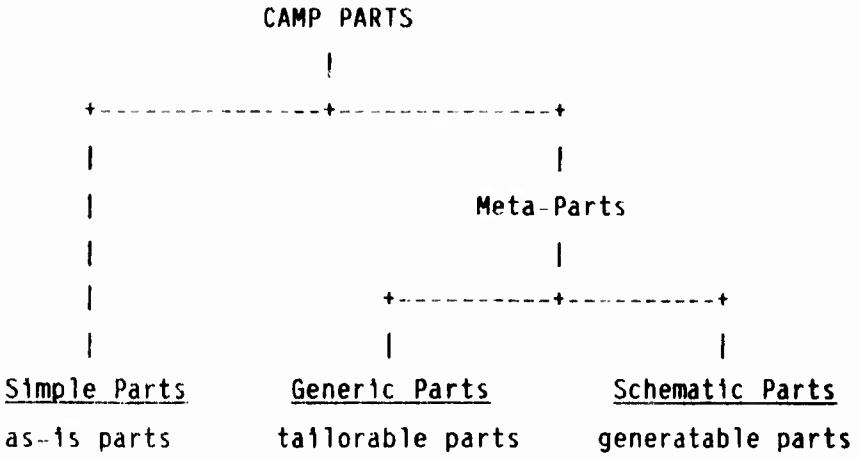


Figure 12. The Three Types of Software Parts

A Simple part is a software part which is capable of being reused as is. In other words, these parts could be used without any tailoring by the user.

The idea of a meta-part is that sometimes we want a family of parts, but do not want to build or maintain each member of that family. Unlike a simple part, meta-parts cannot be used as they exist. Rather, they must be customized to a particular application. In Volume II the use of a software generation system to perform this customization will be discussed. Two types of meta-parts have been identified; they are distinguished by how a particular member of the family of parts described by the meta-part is obtained.

A generic part is a template from which a number of specific parts can be obtained by means of the Ada generic facilities. These are parts in which the parameterization of the part conforms to the capabilities of an Ada generic unit. One example of this level of part would be an abstract data structure such as a generalized First-In-First-Out (FIFO) queue in which the type of the data objects to be queued would be supplied and a specific FIFO queue part would be instantiated for that situation.

A **schematic part** consists of a template and a set of construction rules which are used to generate a number of specific components. Schematic parts differ from generic parts in two important aspects: (1) the generation of specific parts from a schematic part cannot be achieved by means of the Ada generic facilities; and (2) there is no code to look at until a specific part is built. A relatively simple example of a schematic part would be a finite automaton which requires the association of a actions with state transitions (these types of finite automata are usually referred to as Mealy machines). The requirement that actions be associated with state changes cannot be realized in Ada even with its generic facilities because Ada does not have a variable procedure data type. However, the structure of such a part is straightforward. Therefore, the schematic construction rules would be used to build an Ada unit which meets the needs of the user.

The idea of a schematic part is not a new concept (although our use of the term schematic is new). Many researchers have talked about standardizing software designs. Raytheon (Reference 4) has standardized code fragments which represent the major logic structures within their application area. These code fragments serve as templates which are reused between applications. Other researchers (References 30, 31, 32, and 33) have also recognized the benefits of reusable software designs.

Our use of the term 'schematic' as opposed to 'design' is to imply that given the requirements of the part, an expert programmer would know how to build the part, and this knowledge forms a schematic of the family of parts. A standard design, unlike a schematic part, does not imply an associated set of construction rules.

6. GENERAL OBSERVATIONS

One of the general goals of the domain analysis was to identify the characteristics of missile flight software which distinguish it from other software. These characteristics would then be used to drive the detailed analysis needed to identify parts and meta-parts. Figure 13 lists the characteristics identified in this analysis. Each of the characteristics has an effect on the detailed domain analysis as discussed in the following paragraphs.

- Very high degree of data flow inter-connectivity
- Complex decision making
- Large number of mathematical data transformations
- Little data movement
- Relatively simple data structures
- External interfaces with special purpose equipment
- Processes that have rigid temporal relationships
- High use of intermediate results of calculations
- Time-Driven processes

Figure 13. Characteristics of Missile Flight Software Systems

Missile software systems tend to have a large number of independent data objects which must be communicated between functions. The ramification of this fact is that the use of non-parameter procedure communication is essential. As such, in a system with asynchronous processes, data update protection must be provided so that if one process is updating a piece of common data it cannot be accessed by any other process (either to read or write). This protection would only be needed for data objects which are larger than the unit (typically a word) which is automatically protected by the computer. Without this protection, the problem of data integrity will arise. The impact of this fact on the CAMP investigation was to cause us to design data package parts and communication mechanisms to support this behavior.

Early in the investigation we realized that a large percentage of missile functions are devoted to decision making processes (i.e., should an action be taken in light of the current situation). This caused us to take a macroscopic view of the missile functions, looking for recurring patterns of decision making. The result of this was the identification of several abstract processes which provide these logic patterns.

The fact that missile software systems use a large number of mathematical transformations is certainly not new, but, we were driven by this fact to look for more powerful mathematical primitives. Examples of these types of parts would be interpolation tables, change accumulators, etc. While these parts are still quite primitive, they capture some of the standard mathematical operations frequently used within missile systems.

One area which the CAMP team did not have to devote much time to was the process of data movement (e.g., sorts, ordered lists, etc.). While some of these operations and structures were needed (e.g., the maintenance of mission data, etc.) they do not play a major role in this type of software. In a similar vein, missile software does not normally use complex data structures such as multi-threaded linked list, etc. which are typically found in other software systems.

Because of the need for missile software to interface with a variety of special purpose hardware peripherals, we were driven to examine the possibility of developing standard interfaces for classes of equipment such as inertial sensor assemblies, radar altimeters, etc.

One of the surprising results of the initial examination of the CAMP missile set was the realization that, to a large degree, they did not need exotic tasking capabilities. We observed that a large percentage of the asynchronous behavior of these systems could be captured by one level of tasking. In other words, deeply embedded tasking is not needed. In addition, these tasks fell neatly into one of a small number of categories.

a. Aperiodic processes
b. Continuous processes
c. Periodic processes
d. Data-driven processes
e. Interrupt-driven processes

Because of this observation we were driven to look at a way to standardize the construction of the asynchronous structure of missile software systems. The result of this was the development of process controller part and the task shell parts.

A very important observation from the initial investigation was that missile software systems make heavy use of intermediate results. For example, in the area of the primary navigation functions, not only are the end results such as velocity and position needed, but the results of intermediate calculations used to achieve those results are needed by other functions such as the Kalman filter. The major impact of this observation was to cause us to design the parts so that the parts' intermediate results

were kept in a separate package from the calculations, see Figure 14. In this fashion, the intermediate results are available both to the function which calculated them and to other functions.

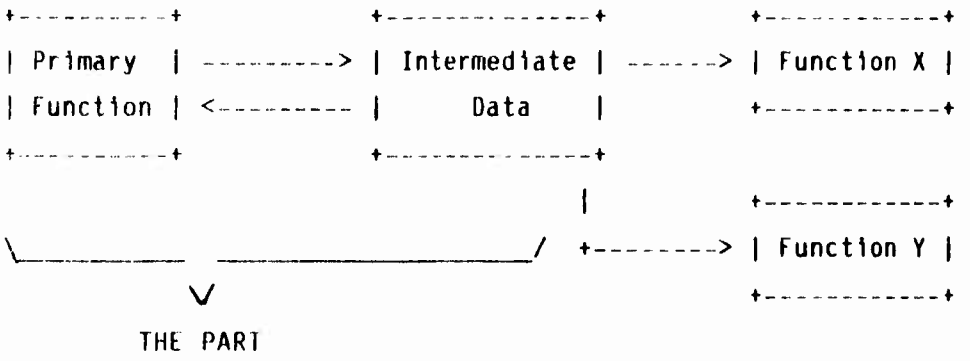


Figure 14. The External Packaging of Data

The final observation from the initial analysis was that there were many actions which were time-driven. As such we looked for standard parts to capture the pattern of these paradigms. The result was the domain independent process sequencers.

7. PARTS IDENTIFICATION

Over 200 parts were identified during the CAMP domain analysis. Appendix C lists these parts which are discussed in much greater detail in the Parts Software Requirements Specification for CAMP and in Volume III of this report.

One of the difficulties we encountered during the domain analysis and later in the design phase of CAMP, was what to call a part. For example, in an Ada package which provides a large number of subprograms which are related (but do not have information strength) are the individual exported subprograms each called a part. Likewise, is a schematic part (which unlike a generic part doesn't exist as Ada code until after a specific instance is generated) a part? During CAMP we used common sense to guide us in delineating what were parts and what were not. No definitive procedure was established.

SECTION III
THE SPECIFICATION OF THE PARTS

1. INTRODUCTION

The specification of CAMP parts must address a number of different aspects of reusable software. The specification must inform the part user of what the part accomplishes functionally and also provide performance information about the part. The specifications must be documented in accordance with the Software Requirements Specification (SRS) DID of DOD-STD-2167, and must facilitate communication of the part's characteristics. A final requirement of the specification is to provide an environment for the part, describing the dependencies which exist between parts.

The CAMP specification technique utilizes both textual and graphical documentation in accomplishing these goals. The textual form is based on the SRS and provides a verbal description of the requirements for each part, following the input-processing-output format. The graphical form provides data flow information about each part and serves as a mechanism for communicating part requirements to missile system engineers. This section will present the specification technique and will document the rationale behind it.

2. ISSUES

The specification of requirements for reusable software necessitates a different approach from that used for single-user software. The Defense System Software Development Standard (DOD-STD-2167) establishes the concept of a Computer System Configuration Item (CSCI) as the basis for software development and defines a CSCI as implementing a complete software subsystem. The standard specifies that requirements shall be derived from the system requirements as defined in the system/segment specification for each CSCI and shall be documented in a Software Requirements Specification. This Specification shall provide interface and data requirements for the CSCI

plus detailed functional and performance requirements for each functional or structural component within the CSCI.

There are three main issues surrounding the specification of parts within the CAMP domain: what constitutes a part, how the part will be used, and how to specify a part so that it is reusable. The first issue will emerge from the process of decomposing system specifications into parts. The second issue will emerge from creating Ada design methods. The third issue will depend on the choice of a specification technique. These three issues are discussed below.

The decomposition of the CAMP CSCI into software requirements is a major issue of the CAMP study. For CAMP parts, the CSCI consists of the requirements derived from the Commonality Study. In essence, CAMP will be developing a CSCI to meet the system requirements of the entire missile group, and it is this missile group which constitutes the system/segment. This CSCI will not meet the entire software requirements of any one missile and will not implement a complete software subsystem. Rather, the CAMP CSCI must provide parts which meet requirements which are common to a number of missiles in the group. Together with custom software, the parts will provide the capability of implementing a software subsystem.

A second issue is the development of requirements which must not only establish the functional nature of the part, but must also establish a means of using the part. The functional nature of a part can be derived from the domain analysis, i.e., the requirement of a reusable part will be based on requirements for specific missile functions. But use of a part must take into account: incorporating the part into a design which is utilizing reusable software, and making use of other reusable parts in order to remain consistent with the rest of the CSCI.

While these issues would generally be considered design considerations for single use software, they play an important role in establishing the requirements for reusable parts. In fact, the data constant and data type parts exist solely to support the design of functional parts which can serve as part of a larger design.

The requirements for single use software establish a single, complete software subsystem. The definition of this unique subsystem will provide the software interfaces to other software components as well as to hardware components. The functional and performance requirements of the entire system will form the basis for choosing these interface requirements.

The specification of reusable software parts must allow for their integration into a variety of different contexts. This need arises because reusable parts must address a wide range of system requirements, rather than requirements of a single system, and must facilitate composition into larger software entities. While inputs to and outputs from a given function will form the basis of interfaces to a part, the requirements must also address the problem of integrating lower level functions into higher level operations or subsystems.

Because the CAMP parts must be specified to meet requirements of an 11th missile, (i.e., a missile that was not used in the commonality study), the final issue is specifying parts which can be composed into systems which have not yet been defined. The interfaces between the larger entities which go into this new missile must also be considered at the time the parts are specified. For this reason, the CAMP specifications must identify parts which are capable of integration into more complex functions, and which are flexible in their application.

3. THE SOFTWARE REQUIREMENTS SPECIFICATION

The CAMP program effectively applied the new DOD standards for the specification of CAMP parts. The program developed a Software Requirements Specification following the tasks described under section 5.1 of DOD-STD-2167. The Specification is in accordance with DI-MCCR-80025, Software Requirements Specification. The structure of the DID requires some tailoring to address the issues discussed above. The remainder of this section summarizes the changes made by the CAMP group to tailor the DID for the description of reusable software parts.

The new standard is especially effective in clearly separating the requirements and design activities into three distinct areas: (1) software requirements analysis, (2) preliminary design, and (3) detailed design. Previous standards such as DOD-STD-1679 and MIL-STD-483 do not make this distinction and preliminary design is subsumed under requirements analysis and detailed design. This clear delineation also lends itself to establishing the manner in which Ada should be applied to the documentation process, as is discussed below.

a. The Use of Ada in the Specifications

The CAMP development effort demonstrated that the SRS should be prepared without reference to a specific Ada architecture. While other studies have attempted to use Ada as a software requirements language, the CAMP approach yielded more generalized requirements specifications, essential for developing reusable Ada software. The development of the requirements must, of course, take cognizance of Ada, to assure that the capabilities of Ada are fully exploited and to establish requirements which are conducive to Ada implementation. Furthermore, Ada terminology (e.g., tasks, packages, generics) is applied wherever appropriate. Nonetheless, keeping the requirements free of Ada constructs improved the readability and utility of the parts specifications.

In addition to documenting specifications, the SRS also establishes design guidelines. The distinction between requirements and design allows CAMP to create an architectural design based on Ada packages. The top-level, or architectural, design defines these packages, their interfaces, and major architectural issues in the design of the package bodies. The lower-level, or detailed, design shows the algorithmic design for subprograms defined in the top-level design and also defines the subprogram and data structures which are encapsulated within the common package bodies. These design issues are fully discussed in Section IV.

b. The CAMP Graphical Technique

The CAMP Graphical Technique offers a powerful yet simple means of communicating the requirements and data flow of CAMP parts. Figure 15 shows the data flow of a typical Ada part. There are three major components to these diagrams:

(1) the rounded rectangles represent functions which the part must provide

(2) the rectangles represent data stores, and

(3) the arcs represent data flows.

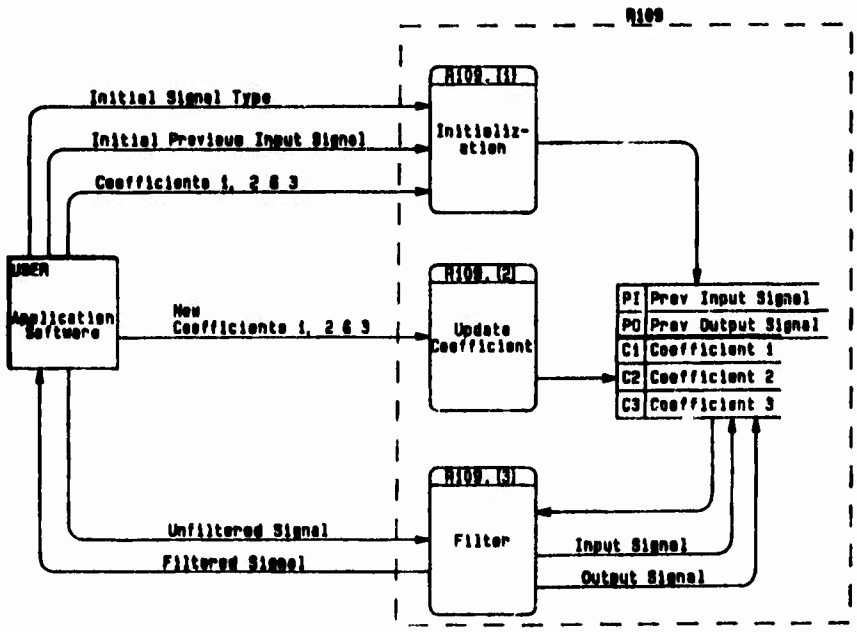


Figure 15. Data Flow of a Typical Part

This graphic technique also captures the flavor of Ada specifications, without establishing a specific Ada architecture. Thus, the functions can be implemented as Ada procedures or functions; the data flows can be accomplished through parameter passing or through shared data; and the data stores can be implemented as data structures in Ada.

c. The Input-Processing-Output Description

The input, processing, and output sections document the processing which must be accomplished by the functions of a part. The requirements specification must define inputs to the part, as illustrated in the data flows, and their data type and meaning. The requirements do not, in general, establish precise Ada data types and they do not define the data control, i.e., parameter data, common data, or local data. This precision is supplied in the design.

The SRS does establish the data type in a form easily recognized by domain experts. This generic definition is well suited to parts which are to be designed as generic Ada parts, where the actual data type will be assigned by the part user. In those cases where a precise data type is specified in the requirements, an Ada generic would not be suitable for the design.

Ada operations are used as a PDL in defining the processing requirements of a part. While this PDL definition is not a complete algorithmic description in most cases, it does show the logical and arithmetic operations which a part will accomplish. The exact nature of these operations will be derived from the domain analysis.

The definition of output from a part will be similar to that of input. The data flows show the output but do not define the data structures. The part specification defines the outputs but does not establish data structures or control. The design of the part will define these features.

d. The Environment Description

The CAMP parts are not intended for stand-alone use. They must be combined into larger contexts for effective utilization. Examples of this combining of parts is the specification of requirements for Basic Data Types, and for Earth Constants. Neither of these packages is useful by itself, but they are essential for the design of other higher-level parts. Another example, illustrated in Figure 16, is the grouping of packages into a Wander Angle or North Pointing navigation system.

However, the specific requirements for a part cannot assume the existence of the lower level packages. A user may wish to use the CAMP navigation packages, but not want to use the CAMP data types. The statement of a part environment will establish the context of a given part, independent of other CAMP parts. The environment may include a description of constants or of data types which are required by a part. For those parts requiring special processing, the environment may indicate external subprograms which must exist for the part to function. Finally, the environment may establish the data which must be supplied for initialization of the state of a given part, indicating that the part may be designed as a generic.

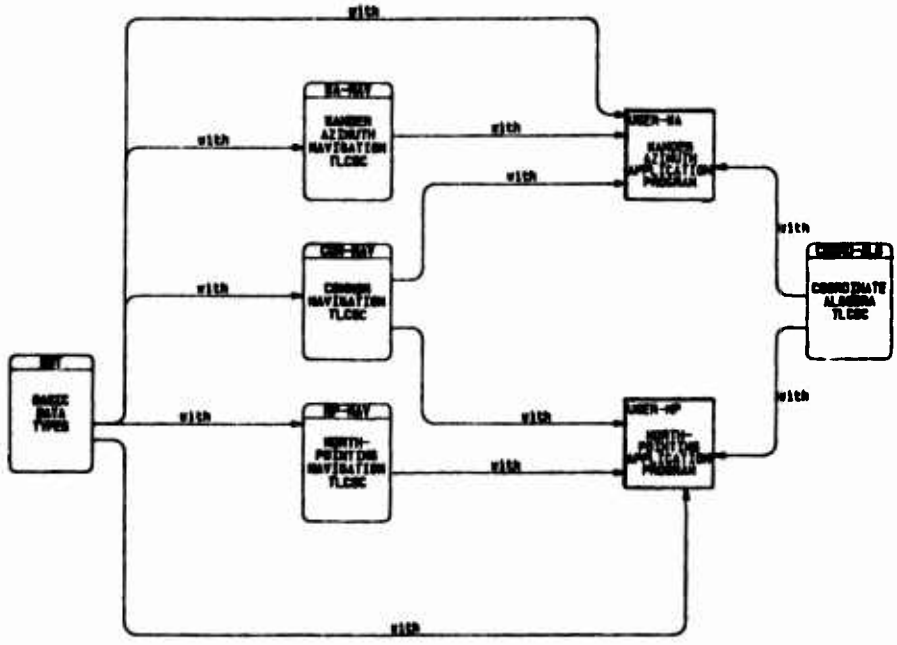


Figure 16. Creating an Environment Through Combining Parts

SECTION IV
THE DESIGN OF CAMP PARTS

1. INTRODUCTION

A design methodology for reusable parts must address several problems arising from conflicting design requirements. To be reusable, parts must have well-defined interfaces reflecting user requirements. The data types used to define these interfaces must be strongly typed to minimize inappropriate use of a part (e.g., using an object of a velocity type for a parameter which should be of a distance type). The design methodology must support the creation of parts which provide a large variety of arithmetic, trigonometric, matrix and other mathematical functions capable of operating on strongly typed data. Finally, the method chosen must ease the job of a part user, possibly at the expense of a significant increase in the development costs of the part. These design requirements (summarized in Figure 17) are necessary to provide parts which are powerful in functionality, yet easy to use.

The CAMP program has produced a top-level design methodology for parts which can meet user typing requirements and which can support parts which are well-tested and may be used off-the-shelf. The methodology utilizes

- Reusable parts must have well-defined interfaces.
- Data must be strongly typed to minimize inappropriate use of a part.
- Design must produce parts which provide a variety of mathematical operations on different data types.
- Design must ease the job of the user, providing simple yet flexible parts.

Figure 17 Issues Affecting the Design of Reusable Parts

several of Ada's special programming features, including derived types and subprograms, generic instantiation and subprogram overloading. The CAMP team believes that this design approach will be equally appropriate for non-missile applications outside of the CAMP domain.

2. METHODS

The CAMP program has considered six design methods which attempt to resolve these design conflicts and achieve reusable parts. These methods are called: Typeless, Overloaded, Generic, State Machine, Abstract Data Type, and Skeletal. They are illustrated in Figure 18.

The processing required to perform the Compute Earth Relative Horizontal Velocities function (SRS R001) will serve as an example in applying these methods. The inputs to this computation are:

Nominal East Velocity (VEL_{NE})
Nominal North Velocity (VEL_{NN})
Wander Angle (WA)

The function performs the following operations:

$VEL_E := VEL_{NE} * \cos(WA) - VEL_{NN} * \sin(WA)$
$VEL_N := VEL_{NE} * \sin(WA) - VEL_{NN} * \cos(WA)$

where

VEL_E is velocity in the true east direction
VEL_N is velocity in the true north direction

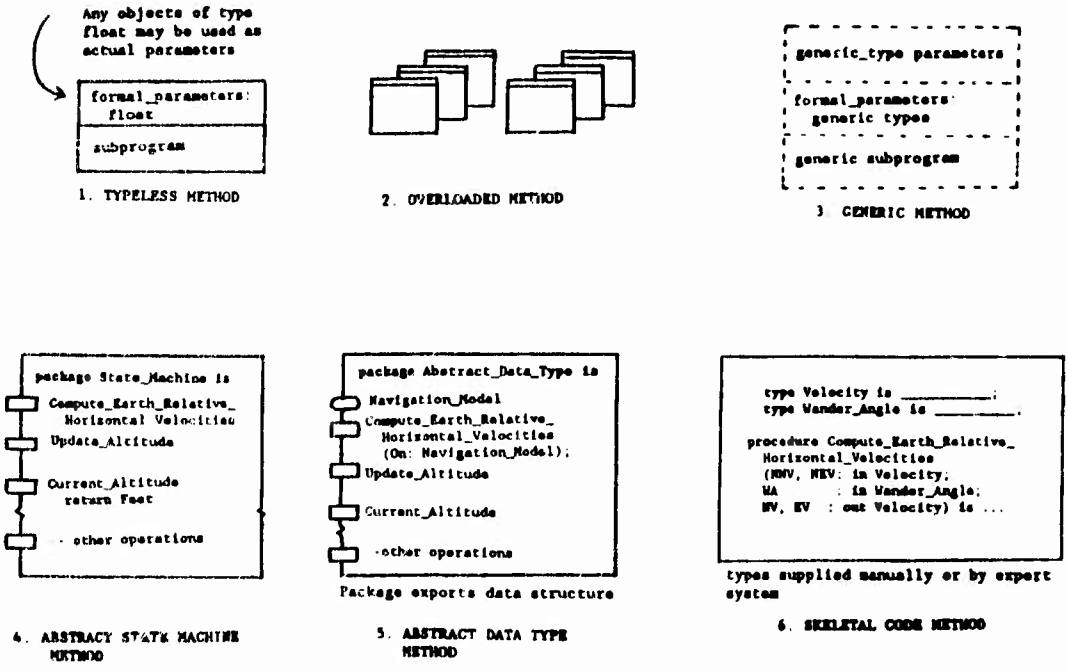


Figure 18. Design Methods for Reusable Parts

a. Typeless Method

The Typeless method assumes that all data objects and actual parameters will be of type float. The benefit of this approach is that it alleviates the need for special mathematical operators and functions, since they are all defined for float in standard packages. The severe disadvantage of this method is that the compiler and runtime system cannot perform type checking to prevent misuse of a part, because all objects are of the same type.

The recent failure of a laser tracking experiment on the Space Shuttle illustrates an error which occurred from mixing data types. The failure was caused by entering data for transmitter elevation in units of feet, while the Shuttle's computer expected it in nautical miles. A data type and object declared as follows:

```
type Ground_Elevation is new Nautical_Miles
  range -1.0 .. 6.0;
Transmitter_Elevation: Ground_Elevation;
```

would solve the problem by restricting the input of Transmitter_Elevation to values which are allowed by the data type. Without this restriction, any value would be allowable for a ground elevation, even 9994 nautical miles, as was input for the experiment which failed.

The Typeless method will result in a procedure specification for the Compute Earth Relative Horizontal Velocities as shown in Figure 19. The program using this procedure could pass any objects of the type float as actual parameters. The compiler could not perform type checking to prevent type mismatching and there could be no runtime checking to assure correct ranges for the actual parameters. This method produces a simple part design, but because there is no error checking it is not a robust design.

```
function Compute_Earth_Relative_Horizontal_Velocities
   (Nominal_East_Velocity,
   Nominal_North_Velocity,
   Wander_Angle_          : in float;
   East_Velocity,
   North_Velocity         : out float);
```

Figure 19. Specification of a Function Using the Typeless Method

b. Overloaded Method

The Overloaded method provides the user a separate version of each part to allow for the different combination of data types which the part user might require. Figure 20 illustrates the application of the overloaded method to the Compute Earth Relative Horizontal Velocities where the velocities are of data types Feet_Per_Second and Meters_Per_Second and the wander angle is in Radians. Other overloaded subprograms would allow wander angle in degrees and semicircles. This method is the same as that used by Ada packages such as Standard and Calendar to provide operations on their data types.

The major advantage of the overloaded method is the simplicity it offers in designing and using the parts. The designer will decide what combinations of data types will be allowed for each part. He will explicitly

declare the parameter interfaces for these overloaded subprograms and can design all of the mathematical parts which the subprograms will use. In the example in Figure 20, the mathematical parts which he must provide are sine and cosine of radians, and multiplication of velocity by a trigonometric ratio returning a velocity. Strict type checking will assure that actual and formal parameters match and that the values of the actual parameters fall within ranges allowed by the type.

Because Ada supports this overloading of subprogram definitions, the user need not call a version of a part specific to a given combination of data

```
package Overloaded_Method is
   function Compute_Earth_Relative_Horizontal_Velocities
   (Nominal_East_Velocity,
   Nominal_North_Velocity:  in Feet_Per_Second;
   Wander_Angle_          :  in Radians;
   East_Velocity,
   North_Velocity         :  out Feet_Per_Second);
   function Compute_Earth_Relative_Horizontal_Velocities
   (Nominal_East_Velocity,
   Nominal_North_Velocity:  in Feet_Per_Second;
   Wander_Angle_          :  in Radians;
   East_Velocity,
   North_Velocity         :  out Feet_Per_Second);
   -- other overloaded functions
   -- . . .
end Overloaded_Method;
```

Figure 20. Specification of a Package Using the Overloaded Method

types; the Ada disambiguation feature will resolve the call. In fact, should user requirements change, and a different combination of data types result, the call need not be changed, the Ada language will resolve the new reference.

The major disadvantage of this method is the large number of parts which must be declared at the architectural level. In the example cited above, the Compute Earth Relative Horizontal Velocities function would require six subprograms to accommodate the different data types. The complete

navigation packages encapsulating these overloaded subprograms could grow to 100 or more subprograms.

c. Generic Method

The Generic method uses Ada generics to provide parts which are tailorable to user-defined types. Figure 21 shows the generic Earth

```
generic
   type Velocity is digits <>;
   type Wander_Angle is digits <>;
   with function "*"
        (Left:Velocity; Right: Trig.Trig_Value)
        return Velocity is <>;
   with function sin (Angle: Wander_Angle)
        return Trig.Trig_Value is <>;
   with function cos (Angle: Wander_Angle)
        return Trig.Trig_Value is <>;
function Compute_Earth_Relative_Horizontal_Velocities
   (Nominal_East_Velocity,
   Nominal_North_Velocity:  in Velocity;
   Wander_Angle_          :  in Wander_Angle;
   East_Velocity,
   North_Velocity         :  out Velocity);
```

Figure 21. Function Specification Using the Generic Method

Relative Horizontal Velocities using generic types for velocity and wander angle. The specific types required by an application would be supplied when the generic function is instantiated.

The Ada generic facility allows functions and operators to be passed as parameters. These generic subprogram parameters are used within individual parts to specify mathematical functions which are needed for generic data types. For example, the Compute Earth Relative Horizontal Velocities generic will require sine and cosine functions on the generic wander angle, and multiplication between the generic velocity and a

trigonometric ratio. This large number of generic parameters could place an enormous burden on the part user, requiring him to specify all of the actual generic parameters (both types and subprograms) needed by the instantiation. In addition, he is required to create functions for the actual generic subprogram parameters. The generic function which is illustrated will require a total of five actual generic parameters and for each combination of data types, the creation of three functions.

A method which uses default parameters could alleviate some of the user's problems. If the design provides functions for typical combinations of data types, then the instantiation will, by default, use these functions as actual subprogram parameters. Using the same example, the design could support trigonometric functions on radians, then if radians is used for the angle generic, the instantiation would default to these predefined trigonometric functions for the generic sine and cosine operations. The main advantage of this method is to permit the user great flexibility in the use of data types, yet provide a simple method for using the parts.

d. State Machine Method

A state machine is an object consisting of data and operations which may be performed to change the state of that data. The unique feature of a state machine is the fact that the structure of the data is not known outside of the machine. The ability to maintain a single set of external interfaces, while changing the internal structure of the state data is a powerful feature of this method. Because data structures are not available externally, there are also operations which permit a user of the state machine to examine the current state of the machine by retrieving the values of data. This definition of a state machine is similar to a black box with procedures to change the state, and functions to read the state, but no ability to see inside the box.

The State Machine implementation of a navigation system would provide all of the operations needed to perform the navigation functions plus operations to obtain the value of those data which define the navigation model. Figure 22 illustrates part of a state machine implementation of a navigation system.

This approach would alleviate the data typing and mathematical problems of the other methods, because the part designer has chosen all internal data types and defined the required mathematical operations. The user of the part would not be able to access any data values stored within the navigation model as shown in Figure 22 except through the interfaces provided by the designer. However, there would be unacceptable inefficiencies in this approach, with the need to convert all data to the part's internal format.

```
with Basic_Data_Types;
generic
   type Velocity is digits <>;
   type Altitude is range <>;
package Navigation_State_Machine is
   procedure Compute_Earth_Relative_Horizontal_Velocities;
   procedure Update_Altitude;
-- Operations to provide state information
   function Current_Altitude return Altitude;
end Navigation_State_Machine;
```

Figure 22. Package Specification Using the State Machine Method

For example, the abstract data type may assume that all angular measurements are in radians. If the user system measures in degrees, all such measurements would require conversion before use by the abstract data type. The part would include the routines to convert to and from the internal form, but this requirement would add an additional subprogram call to each operation.

An additional advantage to this method is the ability to create more than one body for a single specification. Because all data is controlled within the body, a part user may use only the specification and write his own body, defining data according to his own choices. Similarly, the parts designer may provide multiple bodies for a single specification, thus alleviating the efficiency issues by creating bodies which are efficient for a particular situation. This, of course, increases the cost of creating parts and produces a design with less use of commonality, yet it is an

effective method when the choice of a data structure cannot, for reasons of efficiency or simplicity, be established in the package specification. This method would be appropriate for Kalman Filter operations, where the operations are well defined, but the structure of the data and the specific operations on that data will vary from application to application.

e. Abstract Data Type Method

The abstract data type method can establish a single data structure at specification time and thus is completely reusable between applications. An Ada package defines this abstract data type, providing a type declaration plus allowable operations on the type. An abstract data type differs from other data types in that the Ada package hides details about the structure of the type. The word hides describes the fact that the data structure is controlled by the package; the user of the package knows the structure of the data, and may make design choices based on that structure, but he cannot access data directly from the structure.

Information hiding of this type is accomplished through the Ada packaging construct which restricts access to objects of the abstract type to only those operations defined by the package specification. In contrast to the Abstract State Machine, the data structure of an abstract data type is part of the specification, and a package body must operate on that unique structure. If a part user wishes to use the operations of the Abstract Data Type but use a different data structure, then he must not only rewrite the body which will operate on the data structure, but he must also rewrite the specification which defines the structure. This method is useful for such data abstractions as vectors and matrices, stacks and queues, but is not appropriate for more complex data structures such as those used by a navigation system or Kalman Filter operation.

The Ada design of a typical abstract data type package is shown in Figure 23. Note that this example shows a queue type as an entity exported by the package. The private part of the package will define the internal structure of this type. A user of the part can then decide if this structure is suited for his application. For example, if the queue is structured as a linked list, then the efficiency of dynamic allocation of data and access of

data will be design issues which the part user must consider. Had this package been defined as an abstract state machine, the user would have no information from the specification about the nature of the data structure.

```
generic
      type Queue_Element is private;
      Queue_Size: natural;
package Abstract_Queue is
      type Queue is private;
-- User of this package will manipulate objects of the
-- Queue type through the following operations
     procedure Add (Element: in Queue_Element;
                        To_Queue: in out Queue);
-- Other operations . . .
private
      type Queue is . . .
end Abstract_Queue;
```

Figure 23. Package Specification of an Abstract Data Type

To further contrast this method with the state machine approach, an Ada package can define an abstract navigation data type, supported by the generic and private type facilities of Ada. This package would define a navigation type and all the operations needed to perform the navigation functions. The details of implementation of this type would be given in the private part of the package. This method is illustrated in Figure 24.

```
with Basic_Data_Types;
generic
      type Velocity is digits <>;
      type Altitude is range <>;
package Abstract_Navigation_Type is
      type Navigation_Model is private;
      procedure Compute_Earth_Relative_Horizontal_
            Velocities
            (Updating: in out Navigation_Model);
--    Conversion routines
      function Current_Altitude
            (Based_On: Navigation_Model)
            return Altitude;
private
      type Navigation_Model is
            record
               Missile_Velocity:  Velocity;
               Missile_Altitude:  Altitude;
         . . .
            end record
end Abstract_Navigation_Type;
```

Figure 24. Package Specification Using the Abstract Data Type

This method offers essentially the same advantages as the state machine method in regards to data typing and operations. Because the part designer chooses the data structures, the designer will also create all operations needed to support these structures. The chief disadvantage is, again, the need to perform data conversion each time a Navigation Model object is updated. Unlike the state machine, the Abstract Data Type user must use the data structure which the part designer created. If for efficiency reasons, the user does not like this structure he must rewrite the package specification and the package body.

f. Skeletal Code Method

The Skeletal approach gives the user of a part great flexibility. The raw input to this method is not complete Ada code, as in the other methods, but is a template which may be manipulated manually by a template-driven editor, or automatically by an expert system. A sample of a template is shown in Figure 25.

```
function Compute_Earth_Relative_Horizontal_Velocities
   (Nominal_East_Velocity,
   Nominal_North_Velocity,
   Wander_Angle_          :  in _____;
   East_Velocity,
   North_Velocity         :  out _____);
```

Figure 25. Specification of a Function Using the Skeletal Method

When used in a manual mode, the user edits the skeletal code into his existing design and inserts data types, operators, and other identifiers as required. When used in conjunction with an expert system, the expert system will prompt the user for information it needs to fill in the blanks, information which will again include specific data types and operations. While the manual approach would require the user to complete much of the environment for the skeletal part, rules programmed into the expert system allow the system to complete the Ada code, filling in types, operators, and any additional subprograms not in the skeletal version.

While the manual approach seems flexible and simple to use, the difficulty of this method is the need to create an environment for each Ada part built from the skeletal code. If two or more designers are working on similar parts, they may choose different values for completing the template, requiring duplication of environmental data. There would also be a tendency to avoid strong data typing because of the overhead involved in creating the functions and operators for strongly typed data. The expert system concept offers the long-term solution to these difficulties, by building the environment as a by-product of the user dialogue. As the user specifies types, for example, the expert system can define the functions and operators

for those types. The expert system can also inform the user of existing parts which may serve his design.

3. SELECTED DESIGN APPROACH

The CAMP program has developed a design approach based on the generic and overloaded methods which permit reuse of compiled parts from a CAMP Program Library as well as the reuse of source code from a CAMP Text Library. These two modes of usage are referred to as the bundled and unbundled forms and are illustrated in Figure 26.

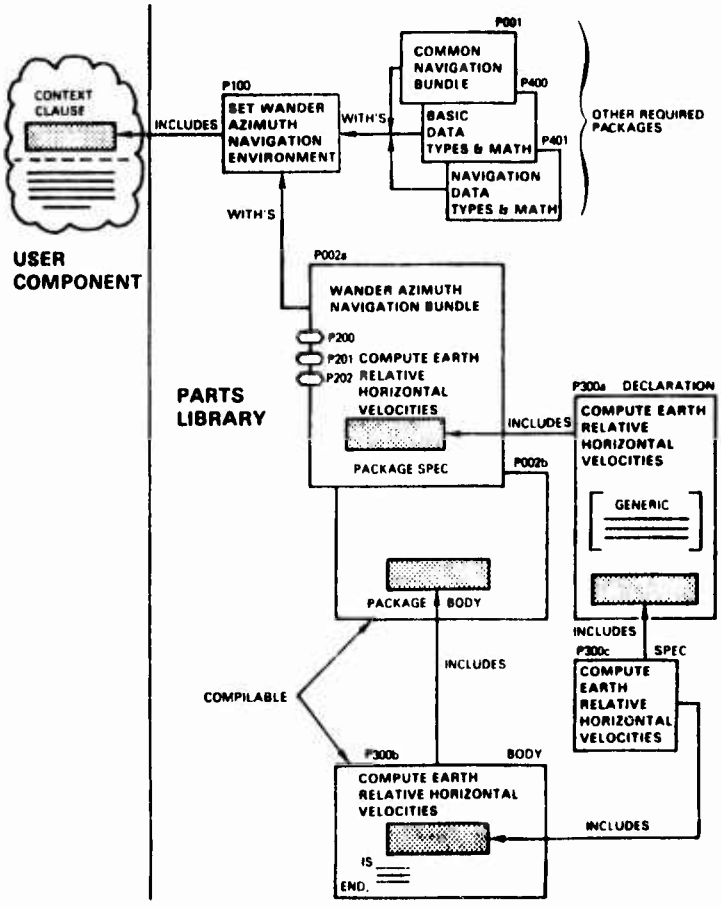


Figure 26. Modes of Using CAMP Parts

In the bundled form, parts are provided to the user in packages. The packages are withed by the user and the user calls the parts through the package mechanism. For packages which contain generic subprograms, a package is withed and generic parts are instantiated for future use. In the unbundled form, the user can access parts as code segments, which are included in his own text files. These code segments may consist of context clauses, packages, generic parts, subprogram specifications, or subprogram bodies.

The library technique specified by the Ada standard (MIL-STD-1815A) will support the CAMP Program Library. The CAMP top-level design specifies parts in the bundled form which have been compiled and are accessed via the Ada context clause. The development and use of the CAMP Text Library will require an include facility, via a pragma, to permit the reuse of source code. This facility was part of earlier Ada standards and is a feature supported by many of the commercially developed Ada Programming Support Environments.

a. Support Provided from the Program Library

The CAMP Ada Library will provide flexibility through the use of generic parts as well as ease of use through packages of typical instantiations of these parts. Generic units offer the user a high degree of flexibility by allowing the part user to use predefined data types or to create his own. Should he choose to do the instantiation using predefined types, the design will provide him with packages of operators and predefined functions using these data types. The instantiation can then use these operators and functions as defaults, easing the user's responsibilities. If the user defines his own types for use by the generic functions, he must also create his own operators and functions and use these in the instantiation of the generic subprograms. He should be cautioned that use of float for all types will negate type checking.

The CAMP program library will also provide packages of typical instantiations of the generic parts. The criteria used for selecting these instantiations will be sensible combinations of data types. This application of the overloaded method provides protection against misuse of a part, supported by the compiler, through strong data type checking of parameters. A package of navigation parts using metric linear measurements and radians for angular measurements would be an example of such a combination. Figure 27 illustrates the methods of using parts which the Program Library will support.

The CAMP program assumes that the expert system will not be a part of the initial Ada parts system, and therefore the program library must provide the user with all the parts he might potentially need. The support required of the CAMP program library increases the work of the original designer of a part, and also increases the overall number of parts, but it is the only method that can address all of the user requirements, without an expert system.

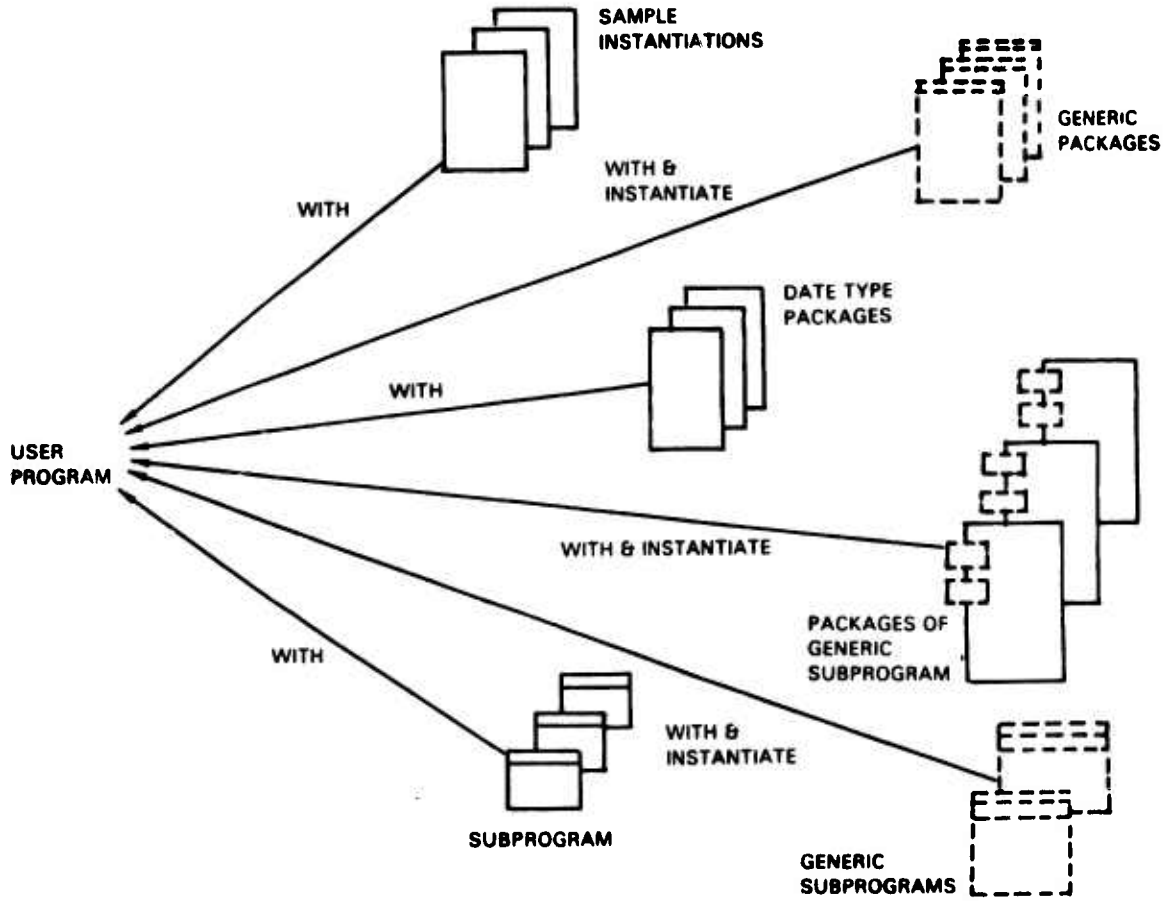


Figure 27. Support Provided by the Program Library

b. Support Provided from the Text Library

The CAMP text library will support the construction of source code from text segments. These fragments will include subprogram specifications and bodies, generic parts, context clauses, data structure declarations, and other Ada program fragments. The Ada instruction "pragma include (Text File)" will permit the user to insert these textual segments into program source code. (This pragma was part of the original Ada standard but was then dropped.) The support provided by the pragma include (if present) will permit the part user to reuse code at the lowest part level, obviating the need for establishing a context of complete packages where he only needs a small number of entities from each package. Figure 28 illustrates the methods of accessing text from the Text Library.

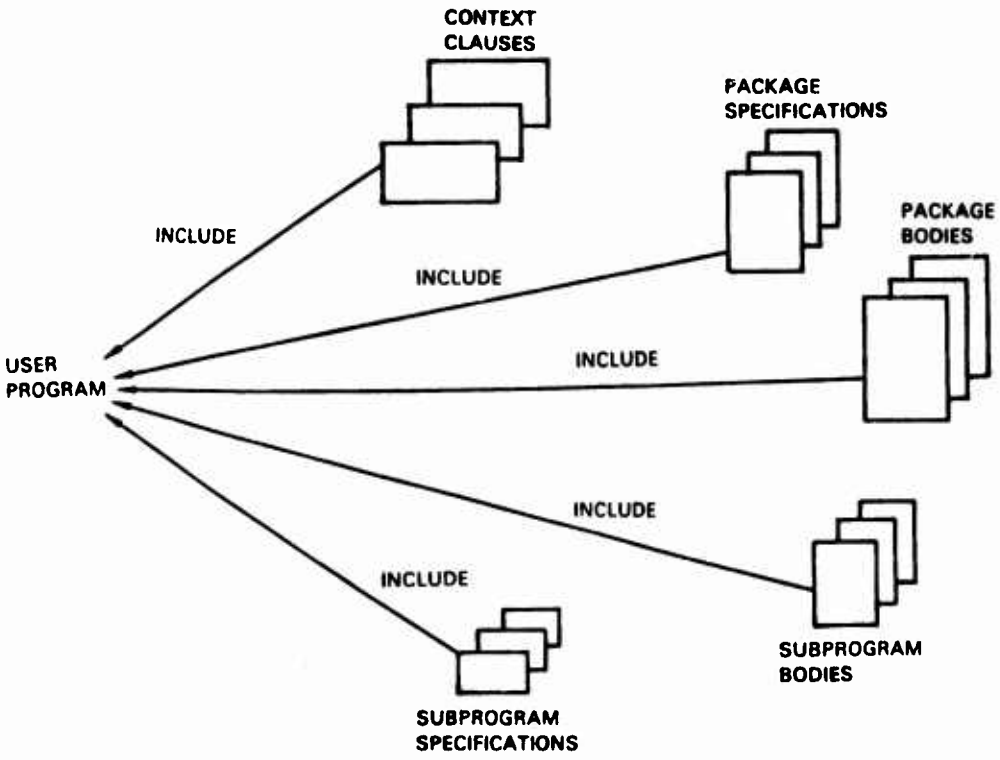


Figure 28. Support Provided by the Text Library

c. Method of Accessing Parts

The CAMP design, both the bundled and unbundled modes, provides a variety of different ways to access parts; Figure 29 illustrates these methods. At the center of the figure is the specification of the Part Package (Item a) containing specifications of generic subprograms. These subprogram specifications are themselves text which has been included from other source files, labeled Generic Subprogram Specifications (Item b). The bodies of the subprograms are individual source files labeled Generic Subprogram Bodies (Item c). The generic subprogram specifications and bodies are included as text in the Part Package Body (Item d). The packages labeled Sample Instantiations (Item e) are packages which contain instantiations of generic subprograms from the Part Package Specification. The user (Item f) has the following choices:

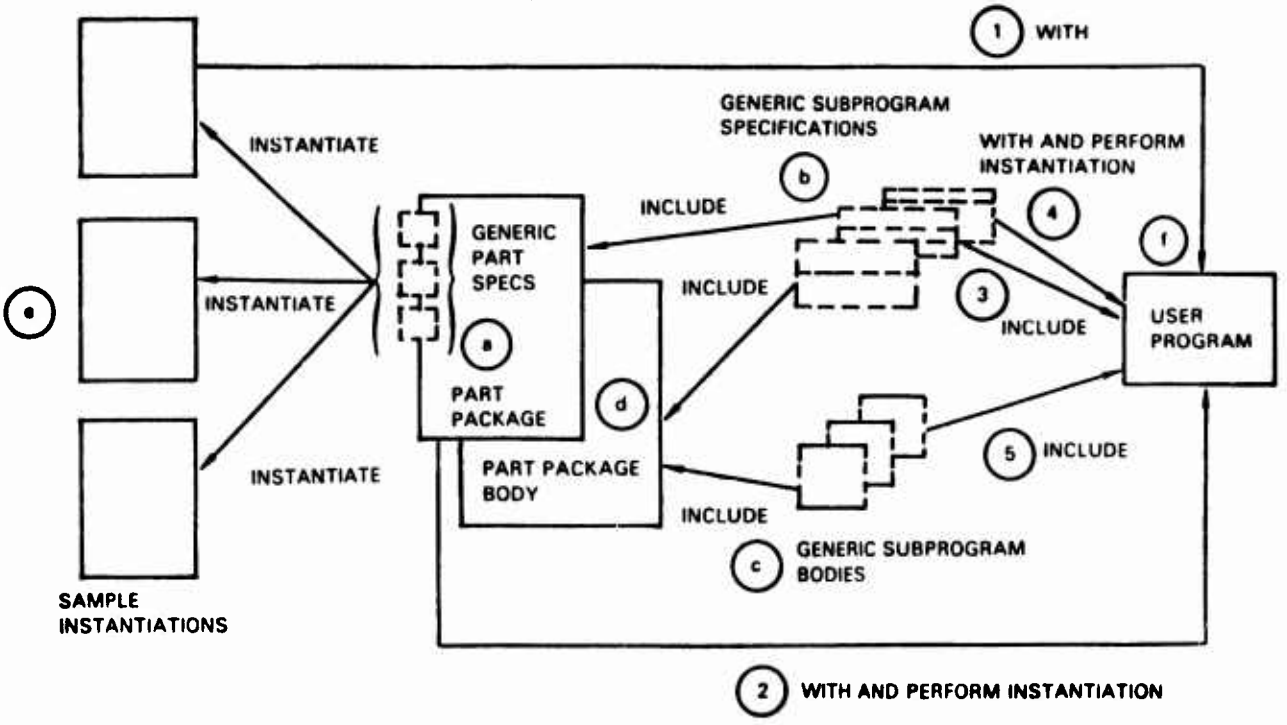


Figure 29. Methods of Accessing Parts

(1) "With" one or more Sample Instantiation packages which will provide subprograms.

(2) "With" the Part Package and perform instantiations for the subprograms which are needed.

(3) "Include" files from the Generic Subprogram Specifications and create instantiations from these generics.

(4) "With" one or more Generic Subprograms and perform instantiations on these individual generics.

(5) "Include" files from the Generic Subprogram Bodies into a user package or program.

d. Efficiency of Parts

The efficiency of individual parts is a major requirement of the CAMP design. The goal of this requirement is to assure that reusable parts will be as efficient as parts which have been custom-designed. The issue of efficiency is most apparent in the design of mathematical functions which are widely used by the primary missile functions.

The development of reusable parts for vector operations illustrates the design of reusable parts which realize the goal of efficiency; Figure 30 shows a portion of this design. Operations on a vector may be performed in a generalized mode, without restrictions on the size of the vectors or matrices. This method is referred to in Figure 30 as the casual user. However, many of the missile operations which need these parts are dealing with three dimensional coordinate axes. The CAMP design includes a special package of coordinate vector and matrix operations to handle this requirement. The actual implementations of these operations are made efficient by unfolding the computations, recognizing that loop operations are needed for generality, but are not needed for coordinate data structures, where there are always three elements. Use of this technique, referred to as the user concerned with efficiency, will utilize three computations to perform a vector operation rather than nine as required by the general looping technique. This approach shows the gains which can be made in terms of the flexibility and utility of the parts but at some increased cost in design.

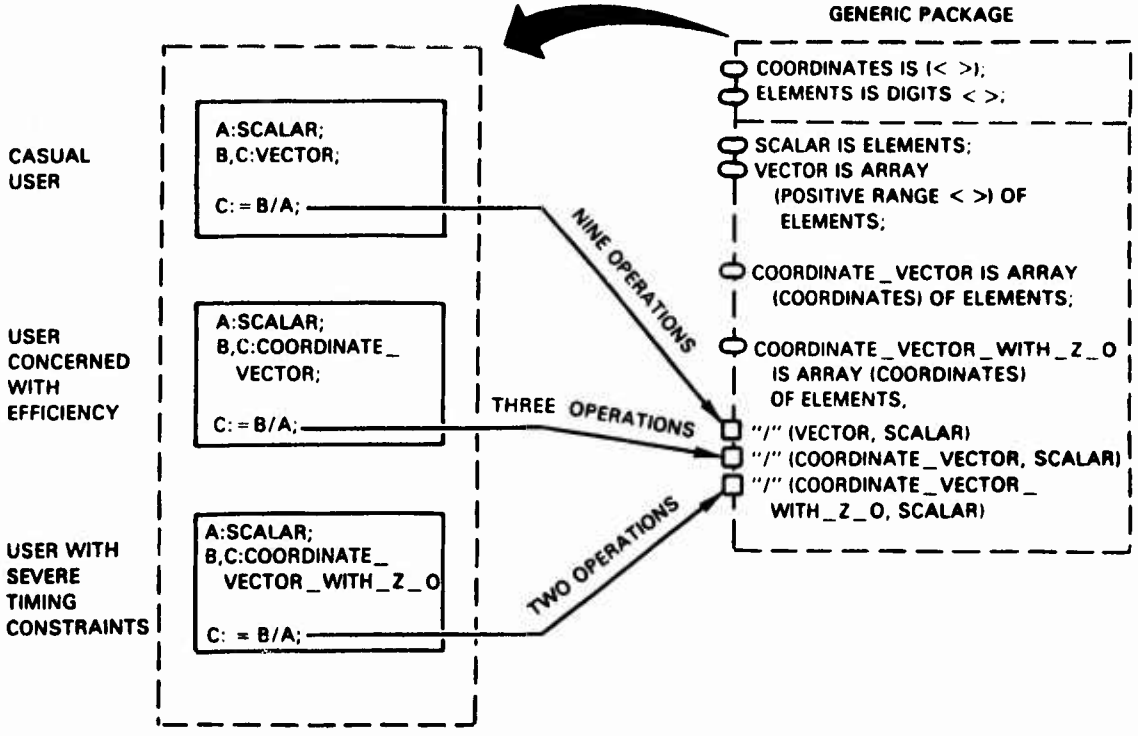


Figure 30. Addressing Efficiency Through Reusable Parts

Many of the primary missile functions operate on coordinate data where one or two dimensions do not apply. For example, angular velocity will exist in only two dimensions and gravity in only one. The CAMP design makes effective reuse of parts by providing sparse vector operations, which make use of the fact that there will be zeroes in the vector. The navigation functions which must operate on vectors of these types can reuse parts from the coordinate vector packages. Sparse matrix operations are used by the user with severe timing constraints in Figure 30.

A significant factor in the design of these parts is that they are accomplished using overloading. The data type chosen by the user to represent his vector will determine the exact operation; it need not be explicitly stated by the user program. For example, if the user defines his vector as a Coordinate_Vector the vector part performing three operations will be automatically invoked.

The CAMP design for trigonometric operations also illustrates the addressing of efficiency issues. The design includes a standard trigonometric package exporting trigonometric operations which are required by the missile functions. The design also includes a polynomial package which exports functions which may provide greater precision, speed, or memory economy, if those are requirements. Computational short cuts to maximize the reuse of intermediate values are another part of the design. An example of this method is the SINE-COSINE function which returns both the sine and cosine of an angle where the algorithm uses the same intermediate terms in computing each value.

4. DOCUMENTATION APPROACH

The documentation of the Top-Level design for the CAMP parts must describe the architecture of part packages and detail the interfaces between packages. This will require TLCSC's which address the issues shown in Figure 31. These requirements must be met both in the TLDD and in the header of the design code itself. Figure 32 illustrates the structure for this information both in the design code header and in the TLDD.

The DOD-STD-2167 Data Item Descriptor for the Software Top-Level Design Document (DI-MCCR-80012) does not adequately cover these issues. The DID seems to be directed towards a design which features data passing through shared data, rather than parameter passing, and parameterless subroutines employed for structural reasons, rather than functional or object-oriented decomposition. This architecture for a TLCSC is not compatible with the object-oriented

- The package context (the list of external packages which are needed)
- The decomposition of the TLCSC into LLCSC's
- Ada Design of the specification of the TLCSC and its LLCSC's
- Major entities which are local to the package body
- Externally callable entries (where tasking is used)
- Requirements for instantiation and other use of a part
- Global processing and output

Figure 31. Issues Which Must Be Addressed in the TLDD

```
-- TLCSC Name
-- ID Number
-- Purpose
-- Requirements Trace
-- Context
-- Exported Entities
-- Local Entities
-- Additional coding information
--
--
package XYZ is
    function . . .
    function . . .
end XYX;
```

DESIGN CODE HEADER

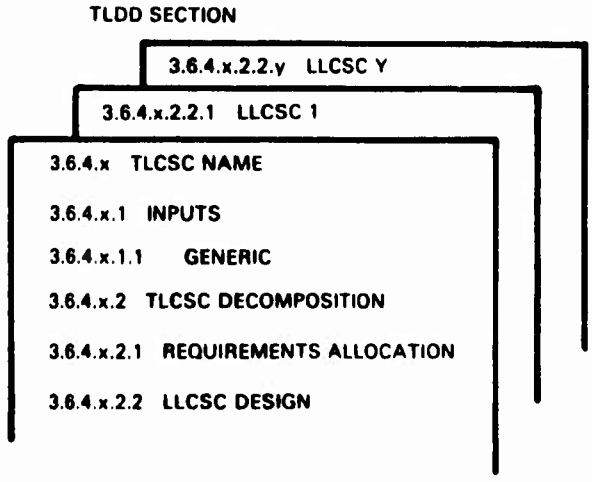


Figure 32. Structure for Design Header and TLDD

nature of an Ada package specification. Therefore, the TLDD is not sufficient for our documentation needs.

Much of the information that properly belongs to a TLCSC which has been designed using Ada has been placed in the Software Detail Design Document (e.g., the TLCSC decomposition, and LLCSC interfacing). The CAMP project has determined that this information must appear in the top-level design description. This will require that the DID for top-level design be modified to include architectural information highlighting the structure of the TLCSC down to the unit level, where units are externally callable. It should also include structural information which is required for the detailed design of these external interfaces. In Ada terms, the TLDD will document the Ada specification plus major data structures and processing needs of the package body. Figure 33 illustrates the top-level design of a typical TLCSC.

The documentation of the top-level design, will in general refer back to the Software Requirements Specification for the top-level algorithmic description. This should be sufficient information for the needs of detailed design, to avoid redundancy between the SRS and the TLDD. For algorithms which have not been adequately documented in the SRS, especially those in the domain independent area, the TLDD must include input-processing-output information.

The Detailed Design Document will describe the implementation of all of the top-level design requirements, for both the bundled version of parts and the unbundled version. The DDD must contain the full package body for all TLCSC's plus those source code segments which are used to build the Ada design code. The actual code for the DDD will consist of a series of pragma include (text file) statements. The text that has been included using this method will be expanded for documentation purposes. The DDD will include the design code for individual parts, the CAMP library structure and the CAMP source text structure.

```
with Basic_Data_Types;

package North_Pointing_Navigation_Parts is

    package BDT  renames Basic_Data_Types;
    package Trig renames BDT.Trig;


    generic
      type Velocity_Vector      is private;
      type Rotation_Rate_Vector is private;
      type Acceleration_Vector  is private;
      with function "+" (Left, Right : Rotation_Rate_Vector)
                        return Rotation_Rate_Vector is <>;
      with function Cross_Product (Left  : Rotation_Rate_Vector;
                                   Right : Velocity_Vector)
                                  return Acceleration_Vector is <>;
    function Generic_Compute_Coriolis_Acceleration
                                  (Velocity_in : in Velocity_Vector;
                                   Rho_in      : in Rotation_Rate_Vector;
                                   Omega_in    : in Rotation_Rate_Vector)
                                  return Acceleration_Vector;


    generic
      type Earth_Position  is digits <>;
      type Distance        is digits <>;
      type Distance_Vector is array (BDT.North_Pointing_Axes)
         of Distance;
      with function "/" (Left, Right : Distance)
                        return Trig.Trig_Value is <>;
      with function "/" (Left : Distance; Right : Trig.Trig_Value)
                        return Distance is <>;
      with function Sin (Angle : Earth_Position)
                        return Trig.Trig_Value is <>;
      with function Cos (Angle : Earth_Position)
                        return Trig.Trig_Value is <>;
    function Generic_Compute_Radii_of_Curvature
        (Latitude : in Earth_Position;
        Altitude : in Distance) return Distance_Vector;

end North_Pointing_Navigation_Parts;
```

Figure 33. Top-Level ADL Design of a Typical TLCSC

SECTION V
THE BENEFITS OF SOFTWARE REUSE

1. PURPOSE

The objective of this analysis was to determine the effect that differing degrees of software reusability would have on software development productivity.

2. DEFINITIONS

While various metrics have been proposed to measure software development productivity, experience has shown that the traditional Lines Of Code per Man Month (LOC/MM) metric is highly correlated to most proposed alternatives, and that the average software engineering manager finds it simpler to grasp than the proposed alternatives. While one might decry the use of such a metric for aesthetic reasons, no viable alternative has yet passed the test of time. For these reasons, productivity (P) was defined, in terms of the total size (S) of the software system (as measured in LOC) and the effort (E) required to develop the code (as measured in man-months), as follows.

$$P = S / E \qquad (1)$$

Since this analysis is concerned with measuring the change in productivity (the dependent variable) as opposed to absolute productivity measurements, the following definition will be used for the degree of productivity improve ment (P%) when comparing the productivity of a project which did not use parts (P_{NP}) and the productivity of a project, developing the same software, using parts (P_P).

$$P\% = (P_P - P_{NP}) / P_{NP} \quad (2)$$

This equation can be simplified as follows.

$$P\% = (P_P / P_{NP}) - 1 \quad (3)$$

For example, if project A and B are developing the same software system, and project A achieves 100 LOC/MM using parts while project B achieves 80 LOC/MM not using parts then the degree of productivity improvement is 25 percent.

$$P\% = (100 / 80) - 1 = 0.25$$

The independent variable in this analysis is the degree of software reuse (R%). This will be defined in terms of the amount of code developed (S_D) and the total size of the software system (S_T) as follows.

$$R\% = (S_T - S_D) / S_T \quad (4)$$

Which can be simplified as follows.

$$R\% = 1 - (S_D / S_T) \quad (5)$$

For example, if project A developed 10,000 lines of code and reuses a number of software parts which provided another 3,000 lines of code, then its degree of software reuse is 23 percent.

$$R\% = 1 - (10{,}000 / 13{,}000) = 0.23$$

3. APPROACH

Determining the effect of reusing software parts on software development productivity involves a comparison between a classical (i.e., no software reuse) project and a project developing the same software using parts. While the use of empirical data, collected during a controlled experiment, would lend credence to any results obtained, such experiments have proven impractical. Few organizations have the resources to fund parallel efforts on real applications, and experiments based on contrived applications often do not generalize to real applications.

For this reason, an analytical approach for evaluating the effect of software reuse was used. In other words, a predictive mathematical model (M) of software cost would be used to determine the effort required to develop software as a function of the software's size and other cost drivers. The software size and other cost drivers would be varied to account for the differences in using and not using parts.

While the exact results obtained using this approach could be questioned (e.g., Was an appropriate software cost model used? Were appropriate parameter values selected?), there is sufficient empirical evidence that the model and the parameter values chosen are close enough to the real world to make the general nature of the results valid.

4. PREDICTIVE MODEL

The Construction Cost Model (COCOMO) developed by Barry Boehm was used as the analytic model in this analysis. This model was chosen based on its wide acceptance by the aerospace software engineering community and the fact that it is one of the few models which have been rigorously defined and documented (see Reference 34). Although COCOMO exists in several versions (each version requiring more information from the estimator and in turn providing more accurate estimates) the Basic COCOMO version was sufficient for this analysis. The following equations define the basic COCOMO model.

$$AAF = (0.4)DMF + (0.3)CMF + (0.3)IMF \quad (6)$$

$$S = C_N + C_A(AAF / 100) \quad (7)$$

$$E = a (S^b) \quad (8)$$

Where

a = 2.4 and b = 1.05 if mode = organic

a = 3.0 and b = 1.12 if mode = semidetached

a = 3.6 and b = 1.20 if mode = embedded

Figure 34 defines the COCOMO modes and Figure 35 defines the variables in these equations.

Organic	This type of software is typically developed by small teams, has relatively relaxed design constraints, negotiable requirements, and usually does not involve the concurrent development of hardware/procedures.
Semidetached	This type of software is a midpoint between embedded and organic.
Embedded	This type of software typically has severe requirements and design constraints, is tightly coupled to complex hardware, software and procedure interfaces, and often requires innovative algorithms.

Figure 34. COCOMO Modes

[AAF]: The Adaptation Adjustment Factor is a percentage which weights the existing code which is to be adapted by the amount of modification needed to use it in the new software

[DMF]: The Design Modification Factor is the percentage of the design of the adapted software which has to be modified to suit the new project

[CMF]: The Code Modification Factor is the percentage of the code of the adapted software which has to be modified to suit the new project

[IMF]: The Integration Modification Factor is the percentage of additional effort required to integrate the adapted software into the new software as compared to the effort normally required to integrate software of a comparable size

[S] This is the total size of the software system as measured in KLOC (thousands of lines of code)

[C_N] This is the amount of new code developed for the project as measured in KLOC (thousands of lines of code)

[C_A] This is the amount of existing code adapted for use in the project as measured in KLOC (thousands of lines of code)

[E] This is the software development efforts (design through testing) as measured in KLOC (thousands of lines of code)

Figure 35. COCOMO Equation Variables

An example (not based on parts reuse) will serve to illustrate the use of this model. The TOMEX (Totally Made-Up Example) project has determined that it will develop 10,000 lines of code and that it can adapt another 3,000 lines of code from the already completed ANMEX (Another Made-Up Example) project. The TOMEX project involves the developed of real-time embedded software with tight constraints on interfaces. The TOMEX software is being developed at the same time as the hardware devices with which it must interface.

From an analysis of the 3,000 lines of code of the ANMEX project which is targeted for reuse, it has been determined that, on average:

- 20 percent of the design of the adapted code will have to be modified,

- 50 percent of the code will have to be modified (those portions which were written in assembly language will need recoding since TOMEX is using a different target processor than ANMEX), and

- the integration of the adapted code into the TOMEX software system will require 30 percent more effort than if we had developed new code for the equivalent functions (i.e. it will be slightly more difficult to integrate the existing parts than to integrate new code because of the need to work with someone else's interfaces).

This description leads to the following values for the COCOMO input parameters.

C_N = 10 KLOC
C_A = 3 KLOC
DMF = 20 %
CMF = 50 %
IMF = 30 %
MODE = Embedded

We can now perform the following calculations.

AAF = (0.4)20 + (0.3)50 + (0.3)30 = 32%
S = 10 + 3(32/100) = 10.96 KLOC
E = 3.6 $(10.96^{1.2})$ = 63.7 manmonths

5. ANALYSIS

Using the COCOMO equation for effort, the equation for productivity (Eq 1) can be reformulated as follows.

$$P = S / aS^{b} \tag{9}$$

This equation can be simplified as follows.

$$P = 1 / aS^{b-1} \tag{10}$$

If one assumes (for the moment) that there is no development effort expended when parts are reused, then the productivity of the parts approach (P_p) can be stated in terms of the size of the total software (S_T) and the effort required to develop the new code as follows.

$$P_p = S_T / a\,(S_D)^{b} \tag{11}$$

Equation 5 can be reformulated to provide a definition of the size of the newly developed code in terms of the degree of software reuse as follows.

$$S_D = (1-R\%)S_T \tag{12}$$

Using equation 12 with equation 11 the following equation is obtained.

$$P_p = S_T / a\,[(1-R\%)S_T]^{b} \tag{13}$$

Then using a transformation similar to that used to develop equation 10, we can simplify this as follows.

$$P_p = 1 / a(1-R\%)^{b}(S_T)^{b-1} \tag{14}$$

The productivity improvement equation (EQ 3) can now be written as follows.

$$P\% = \frac{1 \,/\, a(1-R\%)^{b}(S_T)^{b-1}}{1 \,/\, a(S_T)^{b-1}} - 1 \qquad (15)$$

Which can be simplified as follows.

$$P\% = (1-R\%)^{-b} - 1 \qquad (16)$$

Figure 36 graphically depicts equation 16.

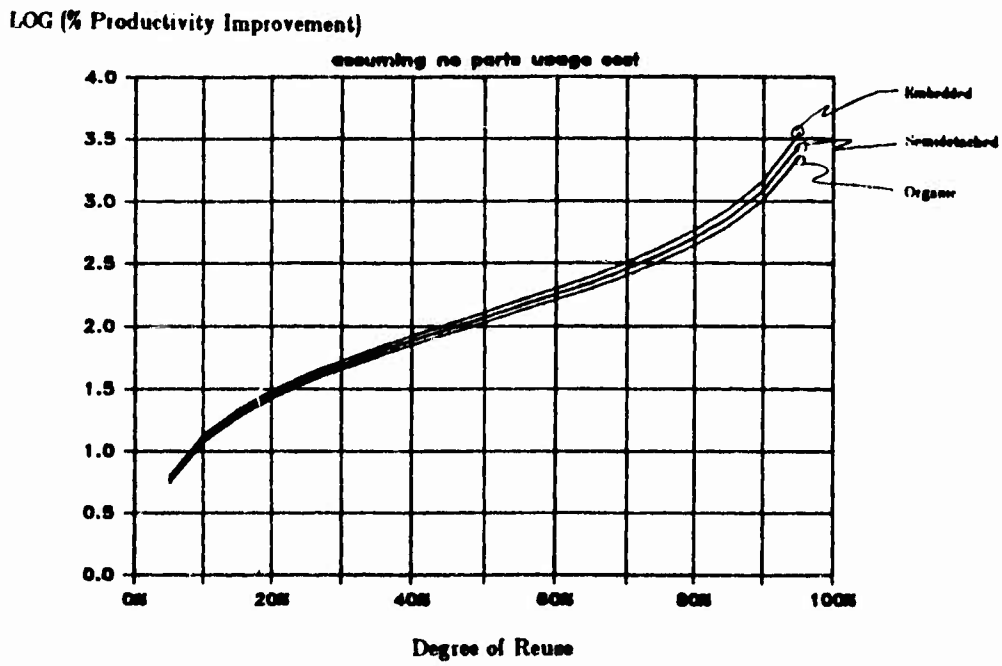


Figure 36. The Effect of Software Reuse on Productivity Assuming no Usage Cost

It can be seen that Figure 36 presents an unrealistic view of using parts. As the degree of reuse approaches 100 percent, the degree of productivity improvement approaches infinity. The missing factor is that there is a developmental cost to reuse parts. The COCOMO Adaptation Adjustment Factor (AAF), see Equation 6, offers a good method of including this parts usage cost. It is assumed that when parts are being reused (as opposed to non-parts reusability) that the parts will be used without any

significant redesigning or recoding. For the purpose of this analysis, it was estimated that 5 percent of the design, and 10 percent of the code of the parts would have to be modified be in a typical application. While it is preferable that parts should be designed to be reused as they exist (i.e., 0 percent design and code modification), this is an ideal goal which will seldom be achievable in real applications.

It is also estimated (based on CAMP experience) that the cost to integrate the parts into the new software will require 10 percent additional effort as compared to the effort to integrate new, customized code which would perform the same functions as the parts used. This cost includes such factors as identifying the appropriate parts, analyzing the parts for suitability, and writing the newly developed software in such a manner that it complies with the interfaces of the parts. These assumptions yield the following AAF.

$$AAF = (0.4)5 + (0.3)10 + (0.3)10 = 8\%$$

The AAF value will be used to define the parts usage cost factor (U) as follows.

$$U = AAF / 100$$

The productivity of the parts approach (P_p) can now be stated as follows using equation 7 and 9.

$$P_P = S_T / a \left[(1-R\%)S_T + (R\%)S_T(U) \right]^b \tag{17}$$

The productivity improvement using parts can be stated as follows.

$$P\% = \frac{S_T / a \left[(1-R\%)S_T + (R\%)S_T(U) \right]^b}{S_T / a \left[S_T\right]^b} - 1 \tag{18}$$

This equation can be simplified as follows.

$$P\% = [1-R\%(1-U)]^{-b}-1 \qquad (19)$$

Figures 37, 38, and 39 depict the behavior of this equation for differing values of U, R% and P. For example, from Figure 37 it can be seen that when the degree of reuse is 50 percent and the cost factor is 10 percent, the productivity gained is 100 percent ($\log_{10}[100] = 2$).

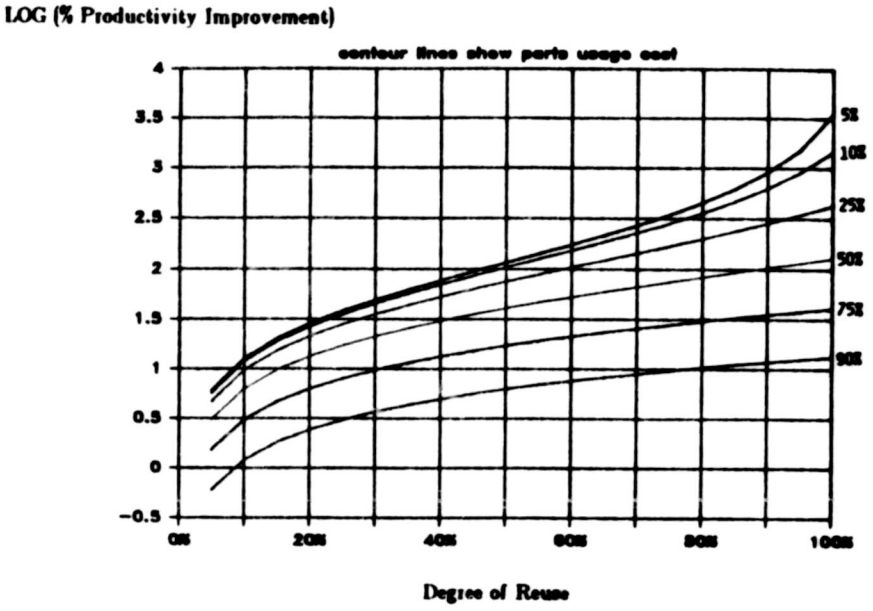


Figure 37. The Effect of Software Reuse on Productivity (Embedded Software)

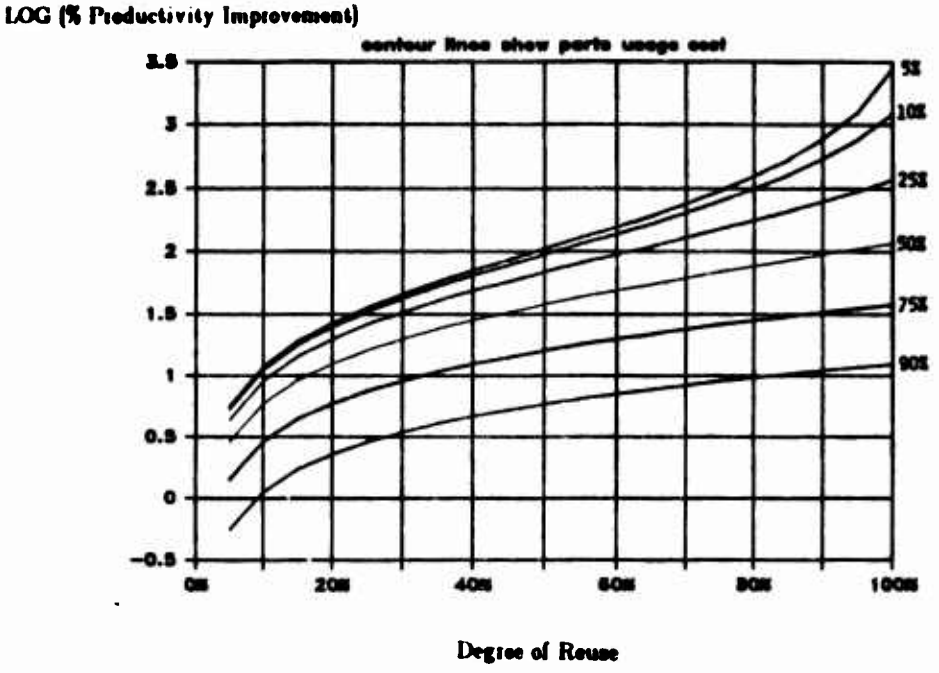


Figure 38. The Effect of Software Reuse on Productivity (Semidetached Software)

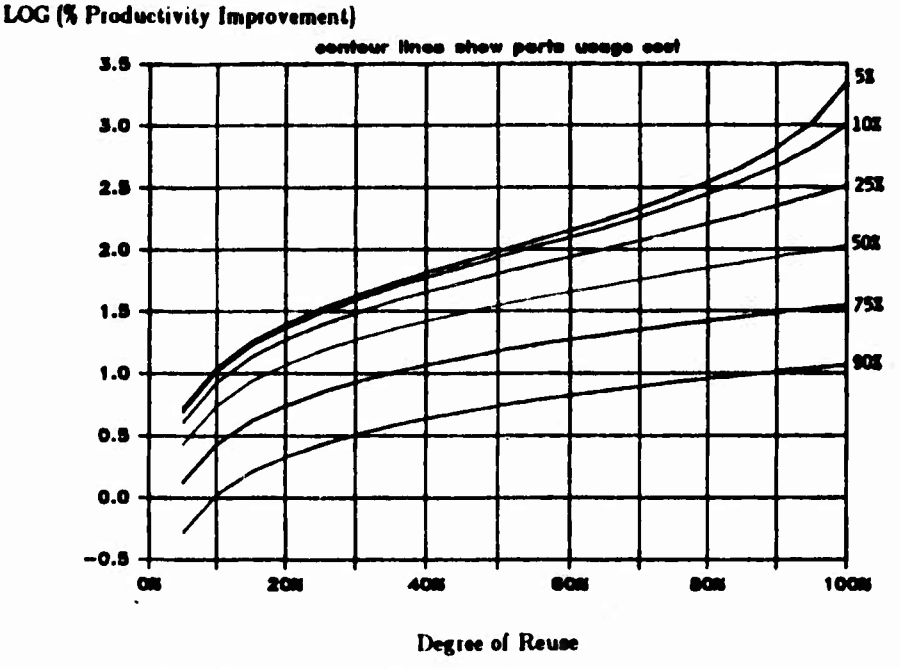


Figure 39. The Effect of Software Reuse on Productivity (Organic Software)

Figure 40 depicts a slightly different view of the same equation. In this figure, the factors controllable by an organization are on the axis, and the result (i.e. the productivity improvement is represented by contour lines).

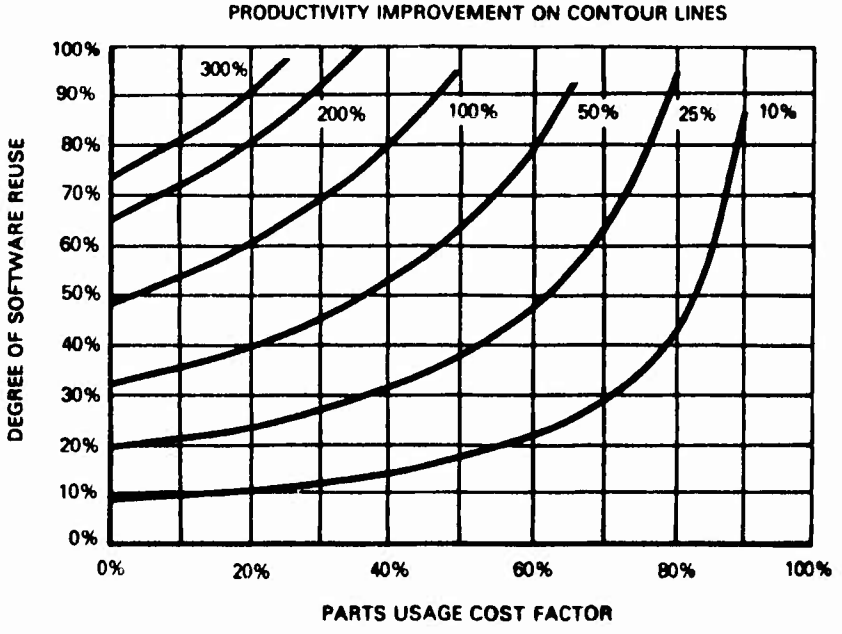


Figure 40. The Effect of Software Reuse and the Cost of Using the Parts on Productivity for Embedded Software

When the degree of reusability is 100 percent (i.e., R=1), equation 19 transforms to the following.

$$P\% = U^{-b} - 1 \qquad (20)$$

Figure 41 depicts the behavior of this equation for differing values of U. For example, if the usage cost is 30 percent, then the maximum productivity improvement for an organic software project (at 100 percent reuse) will be 254 percent ($\log_{10}[254] = 2.4$).

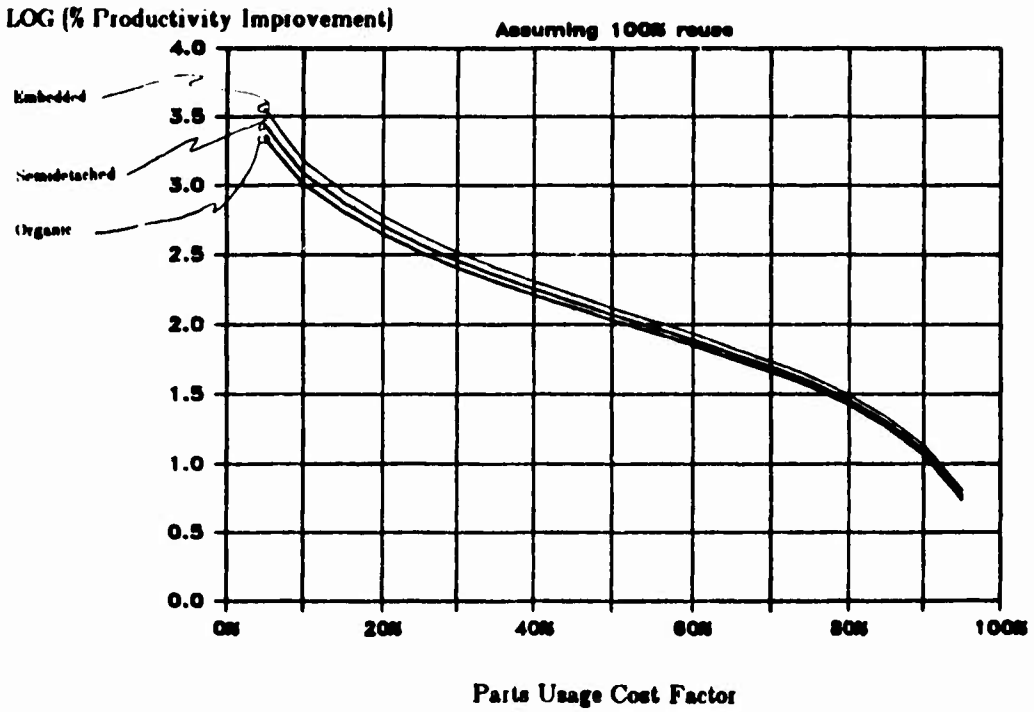


Figure 41. The Effect of the Cost of Using Parts on Productivity

Figure 42 depicts the curve of equation 20 for a usage cost of 8 percent (see previous rationale). At recent STARS meetings, there have been statements that 30 percent is a good estimate for the degree of software reuse that can be expected in the near term. Using these two factors we arrive at the conclusion that the near term productivity improvement should be 47 percent.

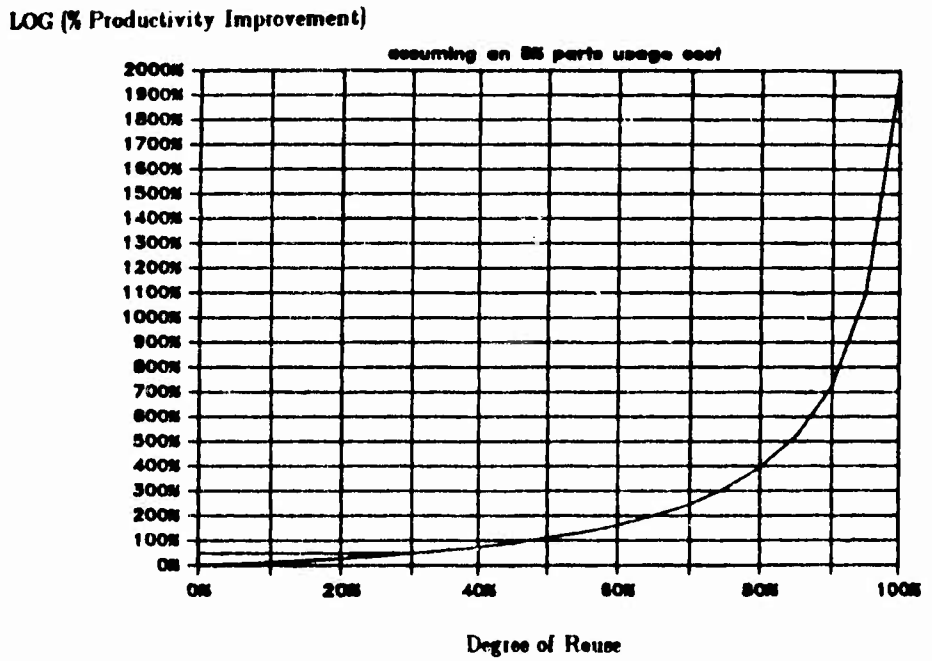


Figure 42. The Effect of Software Reuse on Productivity Assuming an 8% Parts Usage Cost Factor

SECTION VI
EVALUATION OF THE JAPANESE SOFTWARE REUSE EFFORTS

1. INTRODUCTION

During the week of 21 January 1985, three members of the CAMP team visited the six Japanese companies listed in Figure 43 for the purpose of evaluating the software reusability programs, tools, and techniques used by those companies. Although this trip was specifically designed to examine the area of software reusability, our discussions with the Japanese ranged far beyond this one area. This trip was conducted in response to a contractual requirement of the CAMP program.

Toshiba Corporation
Yokosuka Electrical Communication Laboratory, NTT
Nippon Electric Company (NEC) Corporation
Fujitsu, Ltd.
Hitachi Software Engineering
Company X (requested anonymity)

Figure 43. Companies Visited

The general observations made by the authors are discussed in paragraph 2. Detailed descriptions of our visit to each company are contained in the paragraphs 3 through 8.

It should be noted that in most cases the authors met with senior managers and software engineers from application areas other than missile software systems. In many cases, the individuals we talked with were not working directly on real-time, embedded applications although they appeared to be knowledgeable of their companies' efforts in this type of application.

2. GENERAL OBSERVATIONS

a. Software Reusability

Software reusability efforts found within the Japanese companies visited appeared to be equivalent to that found within most American companies, which is to say that reuse of software is in the early stages of implementation. In the past, most of the Japanese reusability efforts have been centered around two approaches--the tailoring of entire systems and the use of application generators; in contrast, the CAMP project is concerned mainly with a parts approach to reusability.

The process of tailoring an existing system to meet a new set of requirements was one that we observed in most of the companies we visited. This approach, which we refer to as system reusability, is based on the premise that within an application area most systems tend to have a common structure and that it is cost effective to start the development of a new system from a baselined version of a similar system which can then be modified to achieve the new set of objectives.

For a number of years the Japanese have been claiming extraordinarily high software development productivity figures. It turns out that these numbers are mainly a result of using application generators within very narrow and well-defined application areas such as banking and man-machine interface construction. An applications generator is a software system that can produce new software systems when provided with the requirements of the new application by the user.

Although all the companies we visited were examining the feasibility of a parts approach to software reusability, only one of them seemed to have progressed beyond the initial study phases. Yet, even this one company had not implemented a full-scale program.

b. Tools & Techniques

During the past several years there have been several articles in the American software literature which have given the impression that Japanese companies have made significant and widespread use of state-of-the-art software engineering tools and techniques. In many cases the authors of these articles have not explicitly made this observation, but, the American software community has drawn this conclusion. Although we saw and heard about a number of tools and techniques during our visit, there are several factors which lead us to conclude that the popular impression is false.

Like most American companies, the Japanese companies have software technology groups or laboratories which develop new tools and techniques. We were told by the companies we visited that many of the tools we saw and heard about were still in the laboratory and not in widespread use throughout their organizations.

In the opinion of the authors, many of the tools we saw were not very technologically innovative. They performed relatively straight-forward tasks using classical techniques. The major reason for this seems to be because the Japanese utilize a large number of non-degreed personnel, thus their tools are designed for the novice software engineer. It is our opinion that many of the tools we saw would be frustrating to American software engineers. These tools typically provide only one path to any objective, which is by means of a rigidly controlled, step-by-step procedure. There is no way to abbreviate this procedure even if the user is proficient with the process and knows what steps are not needed and therefore can be skipped.

Many of the tools we saw would have limited utility in American companies because their major goal was to help Japanese programmers overcome the problems that arise from using conventional programming languages such as Ada, Fortran, COBOL, and 'C'. These languages consist of English language commands, and what is often overlooked by members of the American software

community is that the commands within these programming languages are used as is by the Japanese, (i.e., there is not a Japanese version of COBOL or Fortran or Ada). Although Japanese high school students are taught English, the average programmer has difficulty thinking in this language. For this reason, many of the Japanese tools are designed to give their programmers facilities to write Japanese statements and have those statements converted automatically to the English language commands of the programming language being used.

c. Staffing

One surprising observation made by the authors was the extensive use made of non-degreed personnel in the companies we visited. Although the numbers varied among companies, between 40 percent and 60 percent of the personnel involved in applications software development within these companies had only a high school education.

These employees receive between 6 and 12 months of training within the company and are used initially as software technicians (i.e. performing coding and unit testing functions). Much of the training they receive is on-the-job training as opposed to formal course work. Once proficient, they have the same opportunities to advance within the company as degreed personnel.

One factor that makes this type of staffing feasible is the low turn over found within most Japanese companies. This allows the investment in training to be amortized over the many years of employment.

d. Facilities

The authors were somewhat surprised to observe the software development facilities used by most of the companies visited. In many cases the old bull-pen was still in widespread use. We were also surprised by the use of centralized terminal areas as opposed to terminals at the programmers' work site. We were informed by several companies that plans called for upgrading these facilities in the near future.

e. Ada

Almost all of the companies visited were extremely interested in Ada and were evaluating it for future use. Some of the companies had performed significant work in this area, including the development of Ada compilers and tools to be used with Ada (e.g., symbolic debuggers).

Most of the companies visited perform work for the Japanese Defense Agency (JDA) and were closely tracking JDA's study of Ada. Since our visit, we have learned that JDA did mandate the use of Ada.

f. Expert Systems

The authors were somewhat surprised by the low level of work being performed by most of the companies visited in the application of expert system technology to software engineering problems. Although most of the companies had some efforts underway in this area, they seemed to be mostly research projects.

3. TOSHIBA CORPORATION

Toshiba Corporation was visited on Monday, 21 January 1985. This meeting took place at the new Toshiba Headquarters building located in Tokyo. Figure 44 lists the representatives from Toshiba who were present at this meeting. Figure 45 shows the agenda for the meeting.

Koshi Asakawa, Senior Manager
Defense Market Planning, Defense Products Division
Akira Ito, General Manager
Systems & Software Engineering Division
Ikumune Takahashi, Senior Manager
Software Engineering Department, Systems & Software Engineering Division
Dr. Hideo Nakamura, Assistant Specialist
Systems & Software Engineering Division
Yutaka Ohfude, Manager
Systems & Software Engineering Division
H. Morioka, Senior Manager
Advanced Programs Engineering, Komukai-Works
M. Ikeda, Deputy Manager
Advanced Systems Engineering, Avionics Engineering, Komukai-Works
S. Kinoshita, Senior Specialist
Basic Software Design & Development, Ohme-Works
T. Tanaka, Specialist
Heavy Apparatus Engineering Laboratory, Heavy Apparatus Group
S. Yamamoto, Manager
Eng. Administration & Factory, Information Systems, Fuchu-Works
Ms. M. O'oe, Engineer
Power Generation Control Systems, Fuchu-Works

Figure 44. Toshiba Contacts

10:00 - 10:15	Greetings
10:15 - 11:00	Toshiba Introduction
11:00 - 11:30	Toshiba "Missile Programs & it's Software"
11:30 - 12:00	MDAC Presentation on CAMP
13:45 - 14:30	Toshiba "Software Technology & Activities"
14:30 - 15:30	Discussion

Figure 45. Toshiba Meeting Agenda

a. Products

Toshiba develops software for many different types of commercial and military products, some of which are listed in Figure 46.

Office Automation	computers, data processing systems, word processors, optical page readers, plain-paper copiers, facsimile machines, etc.
Labor Saving ..... Equipment	letter & package sorting M/C, automatic fare collection systems, banking equipment, etc.
Medical Equipment	computed tomography scanners, ultrasound and X-ray diagnostic equipment, nuclear medicine equipment, etc.
Telecommunication .... Equipment	broadcasting and telecommunications, UHF radio relay systems, telephone switchboards
Air Port Equipment ...	air traffic control systems, runway lighting systems, etc.
Space Programs	broadcasting satellite, engineering test satellite, etc.
Defense Programs	missiles, etc.

Figure 46. Toshiba Product Lines

b. Languages

Toshiba's past missile software applications have been written primarily in assembly language. This has been due to the severe time and storage constraints. However, they are studying the use of Ada and other Higher Order Languages because the Japan Defense Agency (JDA) is doing so. Like most of the Japanese firms doing business with the JDA, Toshiba will be greatly affected by any decision the JDA makes with respect to the use of a given HOL.

c. Reusability

Toshiba's main approach to reusability is the reuse of systems, rather than individual software components within the system. An entire software package is standardized, so that in the development of future similar applications, this system serves as a baseline from which changes can be made. Their reported productivity using this approach is 2,800 lines of code per man-month based on the use of assembly language.

They are just now starting to work on software parts reusability as opposed to system reusability. There has been no major reuse of missile software yet. Reusability at the Fuchu Works (non-missile applications) is much further along. Reusability is promoted by rewarding software reuse. If a software engineer develops and registers a reusable components some type of monetary reward is given to him. When a reusable software component is identified, it is registered at a software engineering center and entered into the software database. Software can be registered at either the local level or at the head office.

A parts catalog is not yet part of the Software Work Bench (SWB) at the Fuchu Works. Application specific reusable parts are collected into a textual catalog of sorts. Toshiba believes that consideration of reuse of software must begin at the design level. They also have identified the reuse of software components as a major goal in the future.

d. Tools

Fuchu Works has the SWB (Software Work Bench) which consists of components to support software design, implementation, testing, and maintenance, project management, and software quality control. PROSUS (Process Oriented System Support Software) provides data management system, I/O service system, system support, system monitoring system. IMAP (Integrated Management and Production System) is intended to provide a software CAD approach.

Toshiba is working on graphical techniques (standardized display patterns) and estimates that these techniques should be available for software development in approximately 2 years.

e. Staffing

Within Toshiba, 45 percent of employees are engineers and technical workers (3500); of those, 25 percent perform software related functions. Sixty percent of their software workers are non-degreed, but undergo a one-year internal software training course.

There is some overlap in the types of jobs performed by degreed and non-degreed personnel. The basic functions that must be performed are as follow: requirements specification, system design, program design, coding, and testing. Generally, 1 to 5 years is spent coding; even degreed personnel start in this area. After coding, personnel move on to requirements and system design. Non degreed personnel do have an opportunity for advancement.

f. Enforcement of Methodologies

Enforcement of methodologies is generally by consensus which may take many years to reach; this can vary somewhat by project.

g. Knowledge Engineering

Toshiba is currently working on a knowledge engineering approach to software retrieval.

h. Miscellaneous

Quality Assurance R&D at Toshiba has produced a system referred to as ESQUT (Evaluation of Software Quality from the User's Viewpoint). They have started to use this system in office automation applications.

4. YOKOSUKA ELECTRICAL COMMUNICATION LABORATORY, NTT

Yokosuka Labs was visited on Tuesday, 22 January 1985. This meeting took place at their Kanagawa facility. Figure 47 lists the representatives from Yokosuka Labs who were present at this meeting. There was no explicit agenda for the meeting, but after a brief introduction, the authors were given a number of demonstrations of software tools. These were followed by an MDAC presentation on CAMP and an informal discussion on a wide range of software related topics.

Sadahiro Isoda
Senior Staff Engineer
Ruoichi Hosoya
Chief of Processing Programs Section, Data Processing Division
Dr. Kimio Ibuki
Director, Ibuki's Research Section
Dr. Akihiro Hashimoto
Deputy Director, Data Processing Development Division

Figure 47. Yokosuka Labs Contacts

a. Products

The Electrical Communications Lab is similar to Bell Labs in that it is the research lab for Nippon Telegraph and Telephone Public Corporation (NTT). Some of the areas of research and development that are pursued at the lab are summarized in Figure 48.

b. Languages

Yokosuka makes use of CHILL, assembly languages, and is currently getting involved with Ada. Their interest in Ada stems from an interest in

Transmission	digital networks, optical fiber transmission, radio transmission, mobile communication, satellite communication
Information Processing	data communication network, information processing hardware and software, production of software, intelligent processing
Customer Equipment	telephone sets, data terminals, speech processing, character recognition
Visual Communication	facsimile communication, facsimile equipment, picture processing
Integrated Communications ...	business communication systems, document communications communication processing systems

Figure 48. Yokosuka ECL R&D Areas

portability of software among machines.

They are developing Ada interfaces to a library of existing routines that are written in assembly language; there are currently approximately 140 of these software parts. Yokosuka has an Ada (subset) compiler and associated tools under development (DIPS - Dendenkosha Information Processing System - Ada). They are also working on a syntax-directed editor. A 3-year Ada development project calls for the involvement of approximately 50 people. Yokosuka also has plans to develop cross-compilers for micros (8086, Z80, 68000). Their long-range plans call for the development of an Ada machine.

c. Software Reusability

Yokosuka Laboratories has only recently begun to study software reusability; the Ada interfaces to assembler routines are a step in that direction. A form has been developed to capture the necessary information for a parts catalog. Retrieval of parts will be by keyword.

d. Tools

The engineers at Yokosuka expressed an interest in developing a software parts composition system, but work has not yet begun in this area. They, like the CAMP team, have rejected a universal software generation system approach. An attempt is being made to develop common software development tools for CHILL and Ada.

HCP (Hierarchical and Compact Description Chart) provides standards for graphical description of data structures and program structure, and for the layout on paper of the data flow and program description. HD (HCP Chart Designing system) is a prototype software CAD system.

WAVE (Widely Available Versatile Environment) is intended to be an integrated software development environment that facilitates the collection and transfer of software development knowledge.

SL (Superb Data Oriented Language) consists of 5 sublanguages that allow for the automatic generation of COBOL application programs from program specifications written in SL.

ADAM (Ada Parts Management Tool) is intended to perform three primary functions: (1) register and retrieve software parts and display parts specification, (2) consistency checking, (3) notification of need to recompile due to program modifications.

VIPS is a visual debugger that displays dynamic flow of control and changes to data.

e. Staffing

The employees within the lab itself were highly educated: 10 percent have PhDs, 70 percent have Masters degrees, and 20 percent have Bachelors degrees. The applications programmers at Yokosuka are largely high school graduates who are hired upon graduation and put through an internal training program. The management at Yokosuka made the observation that it was easier to train young people in Ada and CHILL than it was to re-train their older programmers.

f. Enforcement of Methodologies

The discussion of enforcement of methodologies centered around the topic of reusable software. See paragraph c for coverage of this topic.

g. Knowledge Engineering

Yokosuka Labs is not yet using a knowledge engineering approach to software retrieval.

5. NIPPON ELECTRIC COMPANY

NEC was visited on Wednesday and Thursday, 23 & 24 January 1985. The Wednesday meeting took place at the Fuchu plant; the Thursday meeting was held at the Tokyo office. Figure 49 lists the representatives of NEC who were present at the meetings.

The meeting at Fuchu consisted of presentations by NEC personnel, and an overview of the CAMP program by MDAC. This was followed by a general discussion and a tour of the NEC facilities. Thursday's meeting in Tokyo began with an overview of software engineering technology at NEC, and a brief presentation on CAMP by MDAC. This was followed by a tour and demonstrations of a number of software tools. The meeting ended with a general discussion.

a. Products

NEC concentrates on developing hardware and software for the communications area. It provides products for both the private and government sectors, and has customers both in Japan and abroad. The product areas are summarized in Figure 50.

Dr. M. Yamaguchi, Assistant General Manager
Radio Application Division

T. Kato, Supervisor
Microcomputer Software Dept., Software Product Engineering Laboratory

Y. Ishida, Senior Program Mgr
Special Integrated System Division

T. Saya, Manager
2nd Common Software Development Dept., Basic Software Development Dept.

T. Ogou, Engineering Manager
2nd Common-Software Development Dept., Basic Software Development Dept.

N. Ukita, Senior Program Mgr
Dept. of System Management, NEC Aerospace Systems Ltd.

H. Shirakura, Manager
Special Systems Dept., NEC Aerospace Systems Ltd.

K. Miwa, Senior Program Manager
International Government Sales Group

H. Nagasawa, Senior Program Mgr
Maritime Defense Systems, 2nd JDA Sales Division

O. Shigo, Supervisor
Software Engineering Department, Software Engineering Product Laboratory

S. Misaki, Manager
Interface Architecture Department, Software Product Engineering Laboratory

Figure 49. NEC Contacts

Switching	Transmission/Terminals
Radio	Computers/Industrial Electronic Systems
Electron Devices	Home Electronics
Research & Development	

Figure 50. NEC Product Line

b. Languages

The 'C' programming language is used quite extensively, as is COBOL for business applications. NEC, like many of the companies we visited, is awaiting a decision by the JDA on the use of Ada. They are developing an Ada compiler on their DIPS computer targeted to the DIPS; the compiler is written in CPL (Common Programming Language), a PL/1 subset.

NEC has identified several issues surrounding the introduction of Ada in Japan, such as should there be domestic development of Ada compilers and tools or should they be imported from the U.S., and how can support for Ada be developed?

c. Software Reusability

The management at NEC thinks that reuse of software is one solution to the problem of limited resources and increasing demand for software. NEC has not yet implemented the concept of reusability; they are currently solving the problem with increased man-power. Several research projects are underway to prove the concept of reusable software parts. One project entails having an engineer identify reusable parts in order to build and organize a parts library.

d. Tools

The observation was made by a representative from NEC that formal specification languages are good for analysis but not good for people (i.e., they make it difficult to communicate between many different types of people). NEC would like to develop graphical software development environment.

SEA/1 (Software Engineering Architecture) is a COBOL-based parts definition language. Work is currently proceeding on prototype programs (i.e., templates or skeletons for software parts) mainly in the business and aerospace areas. They have not yet incorporated an expert system approach. Retrieval of parts is by keyword.

SDMS (Software Development and Maintenance System) is a design system using a graphical technique; it utilizes underlying knowledge bases.

STEPS is a tool for producing standardized documentation. SPD (Structured Programming Diagram) is a technique used to design well structured programs; it has been proposed for an ISO standard.

e. Staffing

NEC hires many new graduates (approximately 1000 a year; most are engineers). Because of the shortage of engineers, NEC informed the authors that they also employ large numbers of women as technicians and coders, and cross-train large numbers of non-engineering graduates. NEC management informed the authors that, to a great extent, they view women as a temporary labor source. They reported that most women remain in the work force for only about 5 years after graduation from high school. Both groups are provided with in-house training that consists of 3 months of collective training, and then on-the-job training.

f. Enforcement of Methodologies

Discussions of enforcement centered around enforcement of software reuse. NEC management seemed to feel that users can't be forced to reuse parts.

g. Knowledge Engineering

NEC is working on automatic program generation in the banking domain. They are making use of a frame-driven system that incorporates both forward and backward chaining.

h. Miscellaneous

Sixteen percent of NEC's sales are to NTT and the government, 50 percent are to the private sector; 34 percent are overseas. Purchases from NEC amount to 12.5 percent of total JDA expenditures.

NEC has developed the NEDIP (Nippon Electric Dataflow Image Processing System) computer, a non-Von Neumann machine that operates at about the speed of a Cray.

6. FUJITSU

Fujitsu was visited on Thursday, 24 January 1985. Figure 51 lists the representatives of Fujitsu who were present at the meeting. Figure 52 presents the meeting agenda.

Masakatsu Sugimoto, Deputy Manager
Artificial Intelligence Lab, Fujitsu Laboratories Ltd.
Kuniaki Mori, Section Manager
Software Development Planning Group, Development Planning Office
Dr. Toru Nakagawa, Manager
IIAS-SIS, Fujitsu Limited
Mr. Haraguchi
Development Department, Software Division, Computer Systems
Tomoharu Mohri, Researcher
Artificial Intelligence Lab, Fujitsu Laboratories,Ltd.
Mr. Sugimoto, Deputy Manager
Artificial Intelligence Lab, Fujitsu Laboratories Ltd.
Makoto Tsujigadoh, General Manager
Research Management Division, IIAS-SIS, Fujitsu Limited
Sanya Uenara
Tools and Methodologies Section, Software Lab, Fujitsu Laboratories Ltd.

Figure 51. Fujitsu Contacts

Introduction
Software Development Environments in Fujitsu
Software Components Technology in Fujitsu
Introduction to PARADIGM
Approaches of Fujitsu Laboratory
Research Direction
MDAC Presentation on CAMP
Technical Exchange on Ada
General Discussion

Figure 52. Fujitsu Meeting Agenda

a. Products

Fujitsu is involved in telecommunications, data processing, semiconductors, and components. Figure 53 summarizes the major product lines.

Man machine Interface ..	CAD System, intelligent robots, optical sensing and recognition, speech processing, Kanji recognition, printer and facsimile equipment, displays, medical equipment
Telecommunications	communications networks, optical communications systems, radio communications systems
Information Processing .	intelligent information processing system, file memory
Electronic Devices	high electron mobility transistor, silicon LSIs, Josephson junctions, device coding techniques, fine pattern technology, semiconductor material and processing techniques, optical semiconductor devices, infrared devices, optical circuit components

Figure 53. Fujitsu Product Lines

b. Languages

COBOL is used extensively in their business applications. They are currently investigating the feasibility of Ada by using NYU/Ada Ed on a VAX 11/780. They are interested in obtaining other (better) Ada compilers; they do not currently have plans to develop a compiler in-house.

c. Software Reusability

Fujitsu personnel made the observation that reuse of software by itself is not enough; three approaches (new languages, development methodology, and reusable software) must be combined into a software parts approach. Software must be designed for reuse. The first step is to find commonality so that parts can be developed. Engineers at Fujitsu have identified three types of software parts: black box parts, patterns (which provide user with skeletal algorithms and the user is shown where modifications are needed), and good examples. This taxonomy of parts closely matches the CAMP simple, generic, and schematic parts.

Fujitsu is trying to develop a componentized software approach (i.e., develop a database of software parts with unified interfaces). They, like others, think that reusable components must be registered.

d. Tools

PARADIGM is a collection of standard programs and data. The programs consist of program patterns and parts (i.e., actual code segments); the data consists of data screen and report format source code. There are two types of paradigms: standard (i.e., those supplied by Fujitsu) and non standard (i.e., those created by users). PARADIGM is for medium-sized business applications. The developers expect that its use can cut software development time in half by the second or third use of PARADIGM (i.e., as users move up their learning curve).

PDAS (Programming and Design Assist System) is designed to computerize and standardize documents with the eventual goal of automatically generating code from those documents.

SDL (Functional Specification Description Language) provides an environment for developing electronic switching system software. SDL makes use of knowledge engineering. Both PDAS and SDL are in the prototyping stage.

SDT (Software Design Technique) is a structured design technique that can be used as an alternative to HIPO.

e. Staffing

Staffing was not discussed at this meeting.

f. Enforcement of Methodologies

The discussion in this area centered on software reusability.

g. Knowledge Engineering

Fujitsu has an Artificial Intelligence Lab, but the technology does not seem to be applied at a significant level in the applications area.

7. HITACHI SOFTWARE ENGINEERING CO., LTD.

Hitachi Software Engineering was visited on Friday, 25 January 1985. This meeting took place at the Hitachi facility located in Yokohama. Figure 54 lists the representatives from Hitachi who were present at this meeting. Figure 55 shows the agenda for the meeting.

Tomoo Matsubara, Chief Engineer
Masahiro Hirai, Manager, Test and Assurance Department
Masafumi Shimoda, Manager, Overseas Project
Shohei Tomit, Overseas Project

Figure 54. Hitachi Software Engineering Co., Ltd. Contacts

09:00 - 09:15	Greeting and Introduction
09:15 - 10:15	MDAC Presentation on CAMP
10:15 - 10:45	Hitachi Presentation on Productivity (SKIPS)
10:45 - 11:15	Hitachi presentation on Quality Control
11:45 - 12:30	Discussion

Figure 55. Hitachi Meeting Agenda

a. Products

Hitachi Software Engineering Co. is concerned primarily with the development of software products for Hitachi computers. The Hitachi software product line is summarized in Figure 56.

b. Languages

The use of assembly languages is decreasing, while the use of COBOL, PL/1, and HPL (a PL/1-type language) is increasing. The variety of languages makes it difficult to impose standards and reuse. Work is proceeding on implementation of an Ada compiler for the Hitachi M series (this is similar to IBM 3081).

On-line Application Systems: Banking, Stock Information Systems, Taxation Systems, Patent Information System, Computer-Aided Traffic Control System
Minicomputer Applications: Automated Warehouse System, Hotel Reservation and Account System, Cash Dispenser Control System, Telex Exchange System
Graphics and Numerical Control
Operating Systems
Programming Languages and Compilers: Fortran, PL/1, COBOL, RPG, special-purpose languages
Database Management Systems: For Hitachi M series of computers
Electronic Switching Systems
Microprocessor-applied Hardware Systems: Chinese character data entry system, banking teller machine

Figure 56. Hitachi Software Engineering Co. Product Lines

c. Software Reusability

Hitachi has not yet been successful at reuse of software in embedded systems, but the use of standard program formats in banking applications has been fairly successful. They think this is due to the high degree of commonality found in banking applications. When starting a project, they not only decide on a methodology and tools, they also decide on what parts (i.e., standard formats) will be reused.

d. Tools

Tools are developed in part to overcome the language problem. STAMPS (Standardized Modular Programming System) makes use of a combination of a business oriented language and semi-parametric programming to generate COBOL source code. This product is actually being exported. Hitachi makes use of a variety of tools to aid in the testing of software.

e. Staffing

Sixty percent of Hitachi's software engineers are university graduates. The company invests much money in lifetime training of employees; this is feasible because of the low turn-over in personnel.

f. Enforcement of Methodologies

The discussion centered on enforcement of software reusability.

g. Knowledge Engineering

There was not sufficient time to explore this matter in detail with Hitachi personnel.

h. Miscellaneous

There is a great deal of concern with QA at Hitachi Software Engineering. A number of programs have been implemented to increase the quality of the software products produced.

8. COMPANY X

Company X was visited on Friday, 25 January 1985. There was no formal agenda for this meeting. At the end of the meeting, this company's chief spokesman requested that his company's name not be mentioned in any external reports.

a. Products

Company X is a defense contractor, working on various missile developments.

b. Languages

Company X makes use of assembly language for missile applications. They also use 'C' and Pascal for non-missile applications. Company X, like most of the other companies visited, has not yet made a decision on the applicability of Ada to their problem domain. They are awaiting guidance from JDA.

c. Software Reusability

This company is moving towards reuse of software modules (i.e., they think software reuse is necessary), but has not yet implemented a coherent approach to reusability. Language and hardware differences have been major obstacles to software reuse.

d. Tools

No one set of tools or methodologies is used across all projects at Company X, but little by little, standard practices are being implemented.

e. Staffing

All software engineers at Company X are college graduates; most have degrees in electrical or electronics engineering. College graduates experience 1-2 years of on the job training in software design and development. Company X feels that it takes 5-6 years of experience for engineers to be able to properly perform the design task. The first 3 years of employment are similar to an apprenticeship. Company X encourages their engineers to keep current with technology via self-study.

f. Enforcement of Methodologies

Enforcement of methodologies is via audits and walkthroughs.

g. Knowledge Engineering

Company X is not performing any known work in this area.

SECTION VII
EVALUATION OF THE STARS SOFTWARE MEASUREMENT FORMS

1. GENERAL COMMENTS

Part of the CAMP project involved the completion of applicable STARS Baseline Software Measurement Data Item Descriptions. The following six forms were completed:

- Resource Expenditure Detailed Report
- Software Development Environment Summary Report
- Software Characteristics Detailed Report
- Software Change/Error Document
- Software Evaluation Report (REQ)
- Software Evaluation Report (PRELIM)

MDAC-STL supports the STARS effort to obtain software cost data (we have had a similar effort underway for over a year). However, we found several problems with the forms.

a. Cost

A significant amount of time is required to perform the analysis necessary to complete the forms. We estimate that it will take approximately 50 hours per 1000 lines of code to properly complete the forms (from requirements through testing and integration). For large projects, it is possible that the full-time services of one engineer would be required to perform only this analysis. The amount of detail required by the forms is unreasonable, and the cost of providing the detail far outweighs the benefits obtained.

b. Applicability to CAMP

Since the CAMP software is a collection of reusable parts rather than a complete software system, a majority of the items requested were not applicable. The rationale for requiring the CAMP project to complete the Software Measurement forms was to measure various aspects of reusability in order to measure a component's reuse potential. However, only one of the six forms makes any mention of reuse potential.

In addition, the forms assume that formal configuration control exists. Since CAMP is a research project, this type of environment was not practical.

c. Instructions

The directions for completion of the forms are generally quite poor. Often the directions for an item simply restate the question rather than explain what is wanted. For example, in the Completeness section of the Software Evaluation Report (Prelim), the question is asked: "How many data references are identified?" Instead of explaining what is wanted, the directions state: "Enter the number of data references that are identified."

In addition, only two of the six forms have a glossary, and only one has a list of acronyms. The inclusion of these two items, and possibly a sample completed form, would be a useful addition to the instructions.

d. Exposure of Contractor's Proprietary Data

Certain requested items (in particular, cost data) could expose a contractor's proprietary data. An example of such a request is found in Resource Expenditure Detailed Report, which asks for both labor hours and labor cost, which indirectly exposes a contractor's labor rate. Such questions should be removed from the form or assurances should be given to contractors that this data will be secure in the STARS database.

e. Subjectivity of Requested Data

The data requested by the forms is often of a subjective nature. The usefulness of this type of data is highly questionable.

2. COMMENTS ON SPECIFIC FORMS

The following paragraphs contain comments specific to each of the six forms.

a. Resource Expenditure Detailed Report

This form details a contractor's incurred resource expenditure for software development in a particular project. It is completed at the beginning and end of each software development activity. On the CAMP project this translated to four submittals:

- Start of Requirements Analysis,
- End of Requirements Analysis,
- Start of Preliminary Design, and
- End of Preliminary Design.

The information requested by this form is objective, quantitative information and the format is relatively clear. As mentioned earlier, labor hours and labor dollars are requested, thus exposing a company's labor rate. In addition, one of the items requested is computer cost. This is not applicable for organizations which set up a pool for buying equipment, since a cost per CPU time unit is not computed.

b. Software Development Environment Summary Report

This form describes the environment in which the software is being developed and is completed at the beginning of each software development activity. It requests information about the project's software configuration, development site, design methods, personnel, and communication techniques. The CAMP project submitted this form at the beginning of Requirements Analysis and at the beginning of Preliminary Design.

No problems were found with this form. The information requested was objective and easily obtained.

c. Software Characteristics Detailed Report

This form requests detailed software characteristic information at every level (CSCI, TLCSC, Unit). The CAMP project submitted this form at the end of Requirements Analysis and at the end of Preliminary Design.

This form required the largest amount of time to complete because of the detail requested. Almost 20 hours was spent completing the two submissions required on the CAMP project; it is staggering to think of the amount of time which will be required on a large project. The following paragraphs contain comments on some of the sections.

The second question of Section M, which requests unit characteristics, requires the use of a Halstead Software Science Metrics tool, but this is not mentioned anywhere in the documentation. The cost of developing such a tool and completing the analysis for each unit would be a significant load on a project.

Section F, Quality Factor Requirements, gives three choices, "High", "Medium", and "Low", for a series of questions. A fourth choice is needed between "High" and "Medium" to describe the situation in which a quality factor is highly emphasized but not quantified, since many of the factors described are not quantifiable when developing reusable software. In addition, this section is highly subjective.

d. Software Change / Error Document

This form requests detailed information on the changes and errors that occur in a CSCI. The CAMP program submitted this form at the end of Requirements Analysis, but we challenge its applicability in our case. Since we did not baseline our requirements until the end of requirements analysis, it is incorrect to say that we had any errors. In addition, since CAMP is a research project, we did not exercise formal configuration control, which is something that this form assumes.

This form is most applicable to a system that is already deployed, or in the test and integration phase.

e. Software Evaluation Report (Req) and Software Evaluation Report (Prelim)

These forms request an evaluation of the software at the end of Software Requirements Analysis and Preliminary Design, respectively. They collect data in 28 software-oriented classification areas. The CAMP project submitted Software Evaluation Report (Req) at the end of Requirements Analysis and Software Evaluation Report (Prelim) at the end of Top-Level Design.

Many of the forms strayed from objectivity, but these two forms were the worst offenders, requesting vague and subjective information. For example, one of the questions in Req is: "Are there requirements to organize and distribute information within the CSCI?"

In addition, most of the questions were not applicable to CAMP, since they referred to implementation dependent details. For example, Prelim asks for processing time, memory space used, etc. Only a few sections of these forms will yield valid data on the reusability of CAMP parts.

In addition, these forms use the word "all" too often in questions. For example, one of the questions in Prelim is: "Does the design provide interface compatibility among all processors, communication links, memory devices, and peripherals?" This requirement would be unattainable, and therefore the answer to a question such as the above could not be "Yes".

SECTION VIII
REUSABLE SOFTWARE PARTS CONCLUSIONS

This section discusses some of the conclusions reached during CAMP as they relate to the issues of commonality and parts. Conclusions from the CAMP Software Parts Composition study are discussed in Volume II.

Sufficient commonality does exist (albeit well disguised) within missile software systems to justify the development of reusable software parts. CAMP identified over 200 reusable software parts for the missile software domain. The parts identified range from very simple mathematical functions to complex processes and structures. The existence of these parts demonstrates that commonality does exist. It is our belief that most embedded real-time software systems have an equivalent amount of commonality.

The proper performance of a domain analysis is the single most important step in a software reusability program. The identification of meaningful reusable software parts is highly dependent upon a detailed analysis of the target domain by investigators skilled in the design of software parts and in the application area. It is especially critical in real-time embedded software systems to avoid the temptation to propose software parts without conducting this type of analysis or the parts that are developed will likely not be reusable and/or efficient.

Commonality exists at many levels. The identification of functional commonality is typically not difficult. However, the team conducting the domain analysis must be sensitive to the existence of the higher levels of commonality (i.e., pattern and architectural commonality) in order to maximize the number and value of parts.

There is some commonality which cannot be captured in Ada. Although the generic facility in Ada is a mechanism which is invaluable in developing reusable software parts, by itself it is not enough. There exist significant types of commonality which cannot be implemented by means of generics. What is needed is some type of system (e.g. an expert system) in which design

paradigms and rules for building components from these paradigms can be expressed.

Reusable software parts are feasible for the missile software domain. The feasibility of developing software parts for the missile domain was established by showing that Ada could be used to develop parts which were reusable across a wide spectrum of missile applications, and sufficiently efficient for embedded real-time applications. The CAMP parts were designed so that they were as efficient as an application-specific Ada software component written by an Ada expert.

Reusable parts can be built which are both simple to use and powerful. Classically, software engineers have had to compromise between making parts which were simple to use or which were powerful (i.e., allowed the user a high level of control). The design of the CAMP parts demonstrated that when the advanced features of Ada are carefully used, parts can be built which appear simple to the user. In addition, the parts can be used by many different levels of user. The casual user can allow the part to use defaults (unseen to him), the stringent user can override defaults to achieve a level of control over the behavior of the parts.

Developing good reusable parts will cost more (approximately 10 percent - 15 percent) than developing an equivalent amount of application-specific code. The effort required to develop parts which are reusable, efficient, and simple to use is higher than the effort required to develop one-shot software. Although parts can be developed cheaply, good parts (i.e., parts which have a high degree of reusability, efficiency, and simplicity) require special expertise and extra attention.

Software parts have significant potential for increasing software development productivity. Section V of this report established the beneficial impact that software parts can have on software productivity. When amortized across a non-trivial number of applications, the cost to develop and maintain software parts should be greatly outweighed by the cost savings associated with using the parts.

Software parts should be strongly data typed. On the surface, the use of strong data typing in reusable software parts for embedded real-time software systems does not appear practical--but this is not the case. If a non-generic subprogram's parameters are strongly typed (e.g., typing a distance parameters for a computation only in feet), then the generality of the part is lost (e.g., the part cannot be used with parameters in meters). If a generic approach is used, then all internal computation must be generic functional parameters to allow for different typing of the data (e.g., if a part needs to multiply a speed parameter by a time parameter and we wish the user to have the flexibility of specifying the units of the speed and time parameters, then the multiply operator will need to be a generic functional parameter). The requirement to supply a large number of such generic functional parameters can become excessive. Yet, having strong data typing provides valuable protection against misusing the part (e.g., if a part requires a parameter in feet it will not accept a parameter of type meters). One solution which was developed on CAMP was to use the generic approach, but, to have many of the generic functional parameters default to operations provided by the same packages which supplied the data types. In other words, a package which defines speed in feet per second and time in seconds would also overload the multiplication operator to return a distance type in feet. All of this is invisible to the user, unless he wants to override the defaulting mechanism.

Software reuse is not an all or nothing process. Although the ideal goal in developing reusable software parts is to have parts which can always be used exactly as they exist (i.e., with no modification), the value of software reusability is not dependent upon achieving this goal completely. If a user has to modify the parts slightly, there is still a great benefit in using the parts. Another way of saying the same thing is that a part which is close to what a user wants is better that starting from scratch. We should not become such purist that we lose sight of the pragmatic value of such parts.

Software parts should be developed by a project-directed parts group. Figure 57 depicts three approaches to developing software parts. The autonomous parts group approach has the disadvantage that it is easy for the

parts group to develop parts which are not needed or are not usable by the projects. The disadvantage of the project spin-off approach is that projects seldom, if ever, have the additional resources needed to develop good parts (remember that it will cost more to develop a good reusable part). The ideal approach to institutionalizing the development of parts appears to be the project-directed parts group. In this approach, the parts group receives part suggestions or even draft parts from projects and they develop the part. Concurrently, they study specified domains (which have been selected by the projects as being of high value) and the work of the parts group is reviewed by the projects.

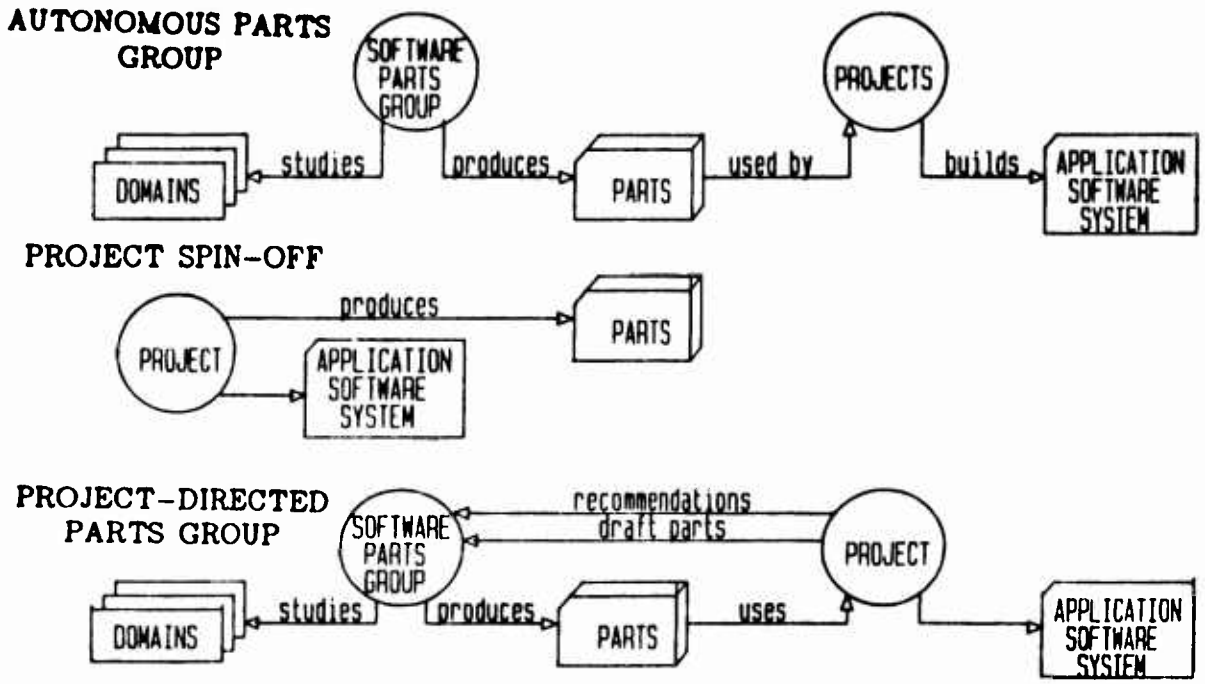


Figure 57: Approaches to Developing Software Parts

The ultimate success of a parts approach to software engineering is dependent upon the establishment of a software parts engineering discipline. The use of software parts will not come naturally to most organizations. To overcome the initial reluctance to use parts and to maximize the use of parts, four things are needed.

a. Management commitment to spur the introduction of parts in the software engineering process.

b. Tools to automate the mechanical chores associated with the use of parts.

c. Knowledge about what parts are available and how they should be used.

d. The enforcement of a discipline which will ensure that parts are used when possible, and are used correctly.

APPENDIX A

CAMP MISSILE SOFTWARE SET

APPENDIX A
CAMP MISSILE SOFTWARE SET

This appendix contains brief descriptions of the ten missile software systems used in the CAMP domain analysis.

ID AGM-109H
Name ..... Medium Range Air-to-Surface Missile
Launch Platform ... Surface Ship and Submarine
Target Type ..... Extended Ground Targets
Warhead ..... Cu ntional, Multiple
Sponsor ..... Air Force
Host Computer ..... Class II Digital Integrating Subsystem (DIS) Computer

---

ID AGM-109L
Name ..... Medium Range Air-to-Surface Missile
Launch Platform ... Aircraft
Target Type ..... Land and Sea Targets
Warhead ..... Conventional, Unitary
Sponsor ..... Air Force
Host Computer ..... Class II Digital Integrating Subsystem (DIS) Computer

---

ID ..... UTG-SINP
Name .....
Unaided Tactical Guidance Strapdown Inertial Navigation Program
Launch Platform ... N/A
Target Type ..... N/A
Warhead ..... N/A
Sponser ..... Air Force
Host Computer ..... MDAC Model 771 processor

ID ..... MGD-GANP

Name ..... Midcourse Guidance Demonstration Guidance and Navigation Program

Launch Platform ... N/A

Target Type ..... N/A

Warhead ..... N/A

Sponsor ..... Air Force

Host Computer ..... Class I Digital Integrating Subsystem (DIS) Computer

---

ID BGM-109A

Name ..... Tomahawk Land Attack Missile

Launch Platform ... Surface Ship or Submarine

Target Type ..... Ground Based

Warhead ..... Nuclear

Sponser ..... JCMPO

Host Computer ..... Litton LC4516C

ID BGM-109B

Name ..... Tomahawk Anti-Ship Missile

Launch Platform ... Surface Ship or Submarine

Target Type ..... Surface Ship

Warhead ..... Conventional, Unitary

Sponsor ..... JCMPO

Host Computer ..... IBM SP-0

ID BGM-109C

Name ..... Tomahawk Land Attack Missile

Launch Platform ... Surface Ship or Submarine

Target Type ..... Ground Based

Warhead ..... Nuclear

Sponsor ..... JCMPO

Host Computer ..... Litton LC4516C

ID BGM-109G

Name ..... Tomahawk Land Attack Missile

Launch Platform ... Ground Based CWCW

Target Type ..... Ground Based

Warhead ..... Nuclear

Sponsor ..... JCMPO

Host Computer ..... Litton LC4516C

Name ..... Harpoon Missile (IC)

Launch Platform ... Surface Ship, Submarine, Aircraft

Target Type ..... Surface Ship

Warhead ..... Conventional, Unitary

Sponsor ..... Navy

Host Computer ..... Midecourse Guidance Unit

Name ..... SPARTAN

Launch Platform ... Ground Based

Target Type ..... Ballistic Missile

Sponsor ..... Army

APPENDIX B

THE CAMP SOFTWARE PARTS TAXONOMY

APPENDIX B

THE CAMP SOFTWARE PARTS TAXONOMY

Figure B-1 depicts the CAMP software parts taxonomy. Each of the taxa (i.e., classes) in this taxonomy are described in the following subsections.

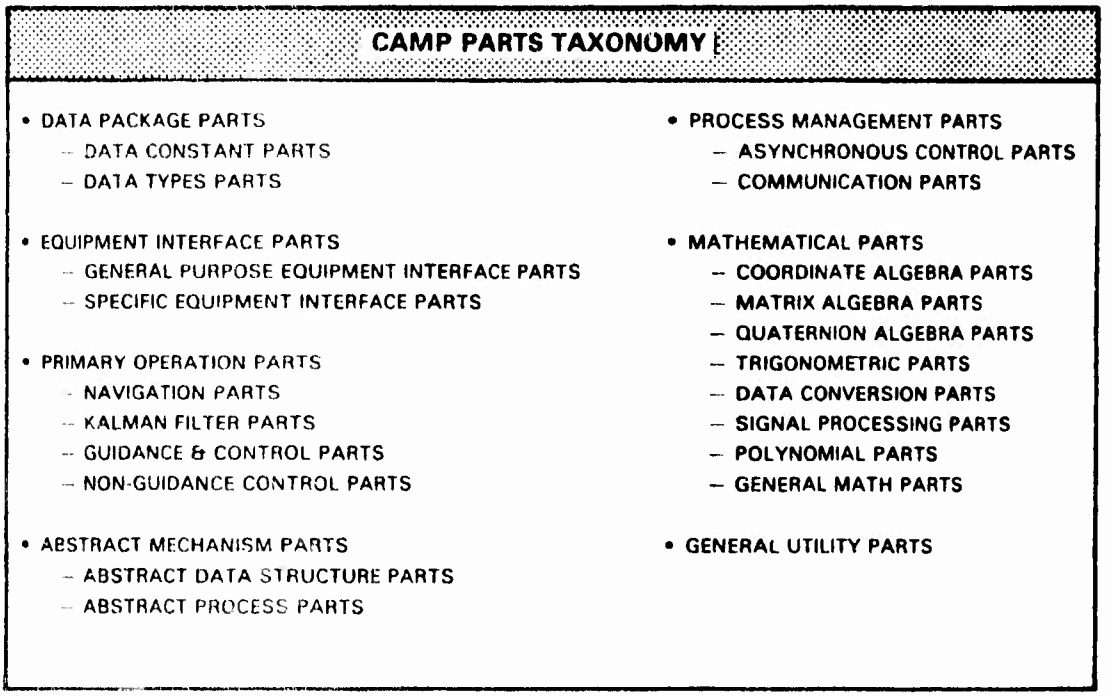


Figure B-1. CAMP Software Parts Taxonomy

1. DATA PACKAGE PARTS

These are parts which encapsulate related data items.

1.1 Data Type Parts

These are parts which provide cohesive sets of Ada data types (i.e. the characterization of data values and applicable operations) to the user. A package containing the data types used within a specific coordinate system would be an example of this type of part.

1.2 Data Constant Parts

These are similar to data type packages except they provide cohesive sets of Ada data constants (i.e., data objects with fixed values) to the user. A package containing constant earth data, such as earth radius and gravity coefficient, would be an example of this type of part.

2. EQUIPMENT INTERFACES PARTS

These are parts which provide standard interfaces for peripheral equipment.

2.1 General Purpose Equipment Interfaces parts

These are parts which provide interface to general classes of equipment. Some examples of this type of part would be an interface to a generic radar altimeter or an interface to a generic data bus.

2.2 Specific Equipment Interfaces Parts

These are parts which provide interfaces to specific hardware devices, such as an interface to a MIL-STD-1553B data bus.

3. PRIMARY OPERATION PARTS

These are parts which provide the operational requirements capabilities.

3.1 Navigation Parts

These are parts which provide the functions required to navigate a missile (i.e., determine its position and velocity).

3.2 Kalman Filter Parts

These are parts which provide the functions required to implement Kalman Filters.

3.3 Guidance & Control Parts

These are parts which provide functions for guiding and controlling the missile (e.g., lateral guidance, vertical guidance, autopilot, etc.).

3.4 Non-Guidance Control Parts

These are parts which provide control functions for non-guidance areas such as fuel control, air data control, seeker control, etc.

4. ABSTRACT MECHANISM PARTS

These are parts which provide mechanisms for implementing various abstract data structures and processes.

4.1 Abstract Data Structures Parts

These are parts which provide mechanisms for implementing high level data objects. For example, various types of queues such as FIFO and LIFO queues, would be implemented by this type of part.

4.2 Abstract Process Parts

These are parts which provide abstract processes such as event sequencers, finite state machines, etc.

5 PROCESS MANAGEMENT PARTS

These are parts which provide function for managing the asynchronous behavior of a missile software system and the communication between tasks.

5.1 Asynchronous Control Parts

These are parts which provide the facilities needed to control the asynchronous behavior of the software processes. In essence, they give the programmer the capability of specifying the asynchronous aspects of missile flight software separate from its functional aspects and at a much higher level of abstraction.

5.2 Communication Parts

These are parts which provide the facilities needed to control data based communications between and within processes. A part which provides mutual update exclusion protection between data which is shared by two or more asynchronous processes would be an example of this type of part.

6. MATHEMATICAL PARTS

These are parts which provide a wide variety of relatively standard mathematical operations.

6.1 Coordinate Algebra Parts

These are parts which provide operations on 3 dimensional coordinate vectors.

6.2 Matrix Algebra Parts

These are parts which provide standard matrix and vector operations (e.g., multiplication, cross-products, etc.).

6.3 Quaternion Algebra Parts

These are parts which provide standard quaternion operations.

6.4 Trigonometric Parts

These are parts which provide the standard trigonometric operations (e.g., sine, cosine, etc.).

6.5 Data Conversion Parts

These are parts which provide operations for converting data from one form/unit to another.

6.6 Signal Processing Parts

These are parts which provide signal processing functions such as filters and limiters.

6.7 Polynomial Parts

These are parts which provide polynomial functions which can be used for a variety of operations (e.g., building special purpose trigonometric functions, etc.)

6.8 General Math Parts

These are general purpose math parts useful in a variety of situations (e.g. interpolation tables, change accumulators, etc.).

7. GENERAL UTILITIES PARTS

These are parts which provide general facilities such as memory declassification, memory checksum, etc.

(The reverse of this page is blank)

APPENDIX C

CAMP PARTS LIST

APPENDIX C
CAMP PARTS LIST

This appendix contains a list of parts identified during the CAMP project. The parts are listed by category and subcategory. The part number is a unique number used for identification purposes. The column entitled Schematic Only refers to whether a part consists only of a part constructor or whether there is also an underlying part that is available separately from the constructor.

CATEGORY	SUBCATEGORY	PART #	NAME	Schematic Only
Data	Constants	R017	WGS72 Ellipsoid Data (Metric Version)	No
Data	Constants	R018	WGS72 Ellipsoid Data (Engineering Version)	No
Data	Constants	R019	WGS72 Unitless Ellipsoid Data	No
Data	Constants	R020	Universal Constants	No
Data	Constants	R158	Conversion Factors	No
Data	Types	R021	Basic Data Types	No
Data	Types	R023	Kalman Filter Data Types	No
Data	Types	R163	Autopilot Data Types	No
Data	Types	R236	Data Type Constructor	Yes
Equipment Interface	General Purpose	R189	Missile Radar Altimeter Handler	No
Equipment Interface	General Purpose	R042	Missile Radar Altimeter Handler with Auto Power On	No
Equipment Interface	General Purpose	R043	Bus Interface Constructor	Yes
Equipment Interface	General Purpose	R046	Clock Handler	No
Navigation	Wander Azimuth	R185	Compute East Velocity	No
Navigation	Wander Azimuth	R186	Compute North Velocity	No
Navigation	Wander Azimuth	R001	Compute Earth Relative Horizontal Velocities	No
Navigation	Common	R002	Altitude Integration	No
Navigation	Common	R003	Compute Ground Velocity	No
Navigation	Wander Azimuth	R004	Compute Total Angular Velocity	No
Navigation	Common	R005	Compute Gravitational Acceleration	No
Navigation	Common	R006	Compute Gravitational Acceleration from Sin LAT	No
Navigation	Wander Azimuth	R007	Compute Coriolis Acceleration	No
Navigation	Wander Azimuth	R187	Compute Coriolis Acceleration from Total Rates	No
Navigation	North Pointing	R008	Compute Coriolis Acceleration	No
Navigation	Common	R009	Compute Heading	No
Navigation	Common	R010	Update Velocity	No
Navigation	Common	R039	Compute Scalar Velocity	No
Navigation	Wander Azimuth	R035	Compute Radii of Curvature	No
Navigation	North Pointing	R036	Compute Radii of Curvature	No
Navigation	North Pointing	R012	Compute Total Platform Rotation Rates	No
Navigation	Wander Azimuth	R011	Compute Total Platform Rotation Rates	No
Navigation	Common	R157	Compute Rotational Increments	No
Navigation	North Pointing	R014	Compute Earth Rotation Rate	No
Navigation	Wander Azimuth	R013	Compute Earth Rotation Rate	No
Navigation	Wander Azimuth	R025	Compute Earth Relative Rotation Rates	No
Navigation	North Pointing	R026	Compute Earth Relative Rotation Rates	No
Navigation	North Pointing	R027	Latitude Integration	No

CATEGORY	SUBCATEGORY	PART #	NAME	Schematic Only
Navigation	Wander Azimuth	R029	Compute Latitude	No
Navigation	Wander Azimuth	R030	Compute Latitude using Arctangent	No
Navigation	North Pointing	R031	Longitude Integration	No
Navigation	Wander Azimuth	R033	Compute Longitude	No
Navigation	Wander Azimuth	R028	Compute Wander Azimuth Angle	No
Navigation	Common	R032	Initialize Direction Cosine Matrix	No
Navigation	Common	R034	Trapeziodal Integration	No
Navigation	North Pointing	R016	North Pointing Navigation Bundle	No
Navigation	Common	R188	Common Navigation Bundle	No
Navigation	Wander Azimuth	R015	Wander Angle Navigation Bundle	No
Navigation	Common	R237	Navigation Component Constructor	Yes
Navigation	Common	R238	Navigation Subsystem Constructor	Yes
Kalman Filter	General	R145	Propogate State Transition and Process Noise Matrices	No
Kalman Filter	General	R146	Propogate Error Covariance Matrix	No
Kalman Filter	General	R147	Kalman Update	No
Kalman Filter	General	R14[illegible]	Propogate State Transition Matrix	No
Kalman Filter	General	R149	Compute Kalman Gains	No
Kalman Filter	General	R150	Update Error Covariance Matrix	No
Kalman Filter	General	R151	Update State Vector	No
Kalman Filter	General	R152	Sequentially Update Covariance Matrix and State Vector	No
Kalman Filter	General	R153	Kalman Executive	No
Kalman Filter	Complicated H	R181	Kalman Update	No
Kalman Filter	Complicated H	R182	Compute Kalman Gain	No
Kalman Filter	Complicated H	R183	Update Error Covariance Matrix	No
Kalman Filter	Complicated H	R184	Update State Vector	No
Kalman Filter	Complicated H	R201	Sequentially Update Covariance Matrix and State Vector	No
Math	Coordinate Algebra	R047	Coordinate Algebra Bundle	No
Math	Coordinate Algebra	R050	Vector Vector Addition	No
Math	Coordinate Algebra	R205	Sparse Right X Vector-Vector Addition	No
Math	Coordinate Algebra	R206	Sparse Right Z Vector-Vector Addition	No
Math	Coordinate Algebra	R051	Vector Vector Subtraction	No
Math	Coordinate Algebra	R207	Sparse Right XY Vector-Vector Subtraction	No
Math	Coordinate Algebra	R052	Vector Vector Dot Product	No
Math	Coordinate Algebra	R208	Vector Length	No
Math	Coordinate Algebra	R053	Vector Vector Cross Product	No
Math	Coordinate Algebra	R054	Vector Scalar Multiplication	No
Math	Coordinate Algebra	R209	Sparse X Vector-Scalar Multiplication	No

CATEGORY	SUBCATEGORY	PART #	NAME	Schematic Only
Math	Coordinate Algebra	R055	Vector Scalar Division	No
Math	Coordinate Algebra	R049	Matrix Vector Multiplication	No
Math	Coordinate Algebra	R056	Matrix Scalar Multiplication	No
Math	Coordinate Algebra	R057	Matrix Scalar Division	No
Math	Coordinate Algebra	R060	Matrix Scalar Addition	No
Math	Coordinate Algebra	R067	Matrix Scalar Subtraction	No
Math	Coordinate Algebra	R068	Matrix Matrix Multiplication	No
Math	Coordinate Algebra	R070	Matrix Matrix Addition	No
Math	Coordinate Algebra	R071	Matrix Matrix Subtraction	No
Math	Coordinate Algebra	R072	Set to Identity Matrix	No
Math	Coordinate Algebra	R078	Set to Zero Matrix	No
Math	Matrix Algebra	R058	Matrix Algebra Package	No
Math	Matrix Algebra	R061	Vector Vector Addition	No
Math	Matrix Algebra	R062	Vector Vector Subtraction	No
Math	Matrix Algebra	R063	Vector Vector Dot Product	No
Math	Matrix Algebra	R104	Vector Length	No
Math	Matrix Algebra	R065	Vector Scalar Multiplication	No
Math	Matrix Algebra	R066	Vector Scalar Division	No
Math	Matrix Algebra	R069	Matrix Vector Multiplication	No
Math	Matrix Algebra	R073	Matrix Scalar Multiplication	No
Math	Matrix Algebra	R074	Matrix Scalar Division	No
Math	Matrix Algebra	R075	Matrix Scalar Addition	No
Math	Matrix Algebra	R076	Matrix Scalar Subtraction	No
Math	Matrix Algebra	R077	Matrix Matrix Multiplication	No
Math	Matrix Algebra	R079	Matrix Matrix Addition	No
Math	Matrix Algebra	R080	Matrix Matrix Subtraction	No
Math	Matrix Algebra	R155	Set to Identity Matrix	No
Math	Matrix Algebra	R156	Set to Zero Matrix	No
Math	Matrix Algebra	R210	Statically Sparse Matrix Constructor	Yes
Math	Matrix Algebra	R226	Define Dynamically Sparse Matrix	No
Math	Matrix Algebra	R211	Define Symmetric (Half Storage) Matrix	No
Math	Matrix Algebra	R227	Define Symmetric (Full Storage) Matrix	No
Math	Matrix Algebra	R212	Define Diagonal Matrix	No
Math	Matrix Algebra	R235	General Purpose Matrix Constructor	Yes
Math	Geometric	R081	Geometric Package	No
Math	Geometric	R082	Compute Angle Between Headings	No
Math	Trigonometric	R083	Radian Trigonometric Package	No

CATEGORY	SUBCATEGORY	PART #	NAME	Schematic Only
Math	Trigonometric	R084	Degree Trigonometric Package	No
Math	Trigonometric	R085	Semicircle Trigonometric Package	No
Math	Trigonometric	R086	Sine	No
Math	Trigonometric	R087	Cosine	No
Math	Trigonometric	R088	Tangent	No
Math	Trigonometric	R089	Arcsine	No
Math	Trigonometric	R090	Arccosine	No
Math	Trigonometric	R091	Arctangent	No
Math	Trigonometric	R092	Sine	No
Math	Trigonometric	R093	Cosine	No
Math	Trigonometric	R094	Tangent	No
Math	Trigonometric	R095	Arcsine	No
Math	Trigonometric	R096	Arccosine	No
Math	Trigonometric	R097	Arctangent	No
Math	Trigonometric	R098	Sine	No
Math	Trigonometric	R099	Cosine	No
Math	Trigonometric	R100	Tangent	No
Math	Trigonometric	R101	Arcsine	No
Math	Trigonometric	R102	Arccosine	No
Math	Trigonometric	R103	Arctangent	No
Math	Data Conversion	R105	Unit Conversion	No
Math	Data Conversion	R106	External Form Conversion	No
Math	Signal Processing	R107	Signal Processing Package	No
Math	Signal Processing	R108	Limiter	No
Math	Signal Processing	R037	Limiter	No
Math	Signal Processing	R038	Limiter	No
Math	Signal Processing	R160	Limiter	No
Math	Signal Processing	R109	General 1st Order Filter	No
Math	Signal Processing	R161	Tustin Lead-Lag Filter	No
Math	Signal Processing	R110	Second Order Filter	No
Math	Signal Processing	R111	Filter Coefficient	No
Math	Signal Processing	R162	Tustin Lag Filter	No
Math	Signal Processing	R202	Absolute Limiter with Flag	No
Math	Signal Processing	R203	Tustin Integrator with Limit	No
Math	General Purpose	R112	General Math Package	No
Math	General Purpose	R113	Change Calculator	No
Math	General Purpose	R114	Accumulator	No

CATEGORY	SUBCATEGORY	PART #	NAME	Schematic Only
Math	General Purpose	R115	Change Accumulator	No
Math	General Purpose	R116	Interpolation	No
Math	General Purpose	R117	Extrapolation	No
Math	General Purpose	R118	Interpolation Table	No
Math	General Purpose	R119	Interpolation Table	No
Math	General Purpose	R120	Incrementor	No
Math	General Purpose	R121	Decrementor	No
Math	General Purpose	R122	Absolute Value	No
Math	General Purpose	R224	Sign	No
Math	General Purpose	R123	Square Root	No
Math	General Purpose	R124	Integrator	No
Math	General Purpose	R142	Running Average	No
Math	General Purpose	R143	Mean Absolute Difference	No
Math	General Purpose	R144	Mean Value of a Vector	No
Math	Polynomial	R213	Table Lookup	No
Math	Polynomial	R214	Chebyshev	No
Math	Polynomial	R215	Fike	No
Math	Polynomial	R216	Hart	No
Math	Polynomial	R217	Hastings	No
Math	Polynomial	R218	Least Squares	No
Math	Polynomial	R219	Legendre	No
Math	Polynomial	R220	Modified Newton-Raphson	No
Math	Polynomial	R221	Newton-Raphson	No
Math	Polynomial	R222	Taylor Series	No
Math	Polynomial	R223	System Functions	No
Abstract Mechanism	Data Structure	R125	FIFO Buffer	No
Abstract Mechanism	Data Structure	R126	Circular Buffer	No
Abstract Mechanism	Data Structure	R164	FIFO Buffer	No
Abstract Mechanism	Data Structure	R165	Priority Queue	No
Abstract Mechanism	Data Structure	R166	Stack	No
Abstract Mechanism	Data Structure	R167	Unbounded Stack	No
Abstract Mechanism	Process	R127	Finite State Machine Constructor	Yes
Abstract Mechanism	Process	R128	Mealy Machine Constructor	Yes
Abstract Mechanism	Process	R129	Event-Driven Sequencer Constructor	Yes
Abstract Mechanism	Process	R130	Time-Driven Sequencer Constructor	Yes
Abstract Mechanism	Process	R159	Sequence Controller Constructor	Yes
Process Management	Asynchronous Control	R131	Process Controller Constructor	Yes

CATEGORY	SUBCATEGORY	PART #	NAME	Schematic Only
Process Management	Asynchronous Control	R132	Aperiodic Task Shell Constructor	Yes
Process Management	Asynchronous Control	R133	Continuous Task Shell Constructor	Yes
Process Management	Asynchronous Control	R134	Periodic Task Shell Constructor	Yes
Process Management	Asynchronous Control	R135	Data Driven Task Shell Constructor	Yes
Process Management	Communication	R136	Message Checksum	No
Process Management	Communication	R137	Update Exclusion	No
General Utility		R138	Memory Checksum	No
General Utility		R139	Memory Checksum	No
General Utility		R140	Memory Declassification	No
General Utility		R141	Instruction Set Test	No
General Utility		R239	Generic Instantiation Constructor	Yes
Guidance & Control	Autopilot	R048	Integral Plus Proportional (IPP) Gain	No
Guidance & Control	Autopilot	R059	Pitch Autopilot	No
Guidance & Control	Autopilot	R064	Lateral/Directional Autopilot	No
Guidance & Control	Waypoint Steering	R168	Compute Unit Radial Vector	No
Guidance & Control	Waypoint Steering	R169	Compute Segment Unit Normal Vector	No
Guidance & Control	Waypoint Steering	R170	Initialize Steering Vectors	No
Guidance & Control	Waypoint Steering	R171	Update Steering Vectors	No
Guidance & Control	Waypoint Steering	R172	Compute Turn Angle and Direction	No
Guidance & Control	Waypoint Steering	R173	Compute Crosstrack and Heading Error when Turning	No
Guidance & Control	Waypoint Steering	R174	Compute Crosstrack and Heading Error	No
Guidance & Control	Waypoint Steering	R175	Compute Crosstrack and Heading Error when not Turning	No
Guidance & Control	Waypoint Steering	R176	Compute Distance to Current Waypoint	No
Guidance & Control	Waypoint Steering	R177	Compute Turning and Nonturning Distances	No
Guidance & Control	Waypoint Steering	R178	Perform Stop Turn Test	No
Guidance & Control	Waypoint Steering	R179	Perform Start Turn Test	No
Guidance & Control	Waypoint Steering	R180	Perform Start/Stop Turn Test	No
Non-Guidance Control	Air Data	R228	Compute Outside Air Temperature	No
Non-Guidance Control	Air Data	R229	Compute Pressure Ratio	No
Non-Guidance Control	Air Data	R230	Compute Mach	No
Non-Guidance Control	Air Data	R231	Compute Dynamic Pressure	No
Non-Guidance Control	Air Data	R232	Compute Speed of Sound	No
Non-Guidance Control	Air Data	R233	Compute Barometric Altitude	No
Non-Guidance Control	Fuel Consumption	R234	Mach Control	No
		219	<== TOTAL NUMBER OF PARTS	

APPENDIX D

EVALUATION OF DOD-STD-2167 AND ASSOCIATED DID'S

APPENDIX D
EVALUATION OF DOD-STD-2167 AND ASSOCIATED DID'S

1. INTRODUCTION

The standard for Defense System Software Development (DOD-STD-2167) successfully establishes a complete life-cycle process for developing and maintaining Mission-Critical Computer Software. It provides a uniform method for:

a. generating different types and levels of software and documentation;
b. applying development tools, approaches and methods; and,
c. planning and controlling projects.

The Data Item Descriptors (DID's) which accompany the standard provide adequate guidelines for recording and communicating the information generated during the software development process.

One of the major tasks of the CAMP effort was the application of MIL-STD-2167 to the development of a reusable parts library and an automated parts engineering system. The standard is not directed at the development of reusable software, expert systems, or Ada-based systems. The application of 2167 required the CAMP team to tailor the requirements of the standard to meet the particular needs of each of these three areas.

While the standard is adequate for defining the development process of some aspects of reusable software, the specification and design of CAMP parts according to 2167 posed some difficulties. The specification must inform the part designer and user of what the part accomplishes functionally and also provide performance information about the part. The fact that CAMP is concerned with reusable software does not significantly change this activity. The standard does not establish the exact meaning of a Computer Software Configuration Item (CSCI) for reusable software, nor does it define the meaning of a software part. In addition, the specification must provide an environment for the part, describing the dependencies which exist between parts. The top-level design must describe the architecture of the system and its decomposition into components, detailing the interfaces between components. Both the top-level and detailed design must describe global processing and output for each component as well as major entities which are local to the component. DOD-STD-2167 was not intended to address these issues for reusable software.

Following DOD-STD-2167 for the development of expert systems raised other issues during the CAMP study. The development process for these systems does not follow the traditional top-down approach. Major requirements for fundamental system functions of the expert system are not decomposed into subordinate units, except for general utilities, such as creating relations or updating a data base. These system functions rely on inference mechanisms built into an expert system, and are usually recursive Lisp-based operations. The process of developing the knowledge base, the set of rules for driving the expert system, is also not covered by 2167, but it is clearly a system requirement.

Software developed using Ada is also not addressed by the standard. The Joint Logistics Commanders Computer Software Management Subgroup is currently developing a revision to the standard to incorporate the use of Ada into the development process. The use of Ada on CAMP established several areas of the standard which the revision process must address. These include Ada constructs such as packages and generics, as well as the concept of data typing.

This Appendix will present an overview of DOD-STD-2167, comparing it to previous standards, and assessing its general applicability to the development of mission-critical software. It will also describe the application of the standard to the development of reusable software parts, the development of the expert system, and the use of Ada for the CAMP study. A section of the Appendix will describe each of the three DID's which apply to CAMP documents and their appropriateness to the documentation of CAMP software. The final section of the Appendix discusses the design and coding standards of DOD-STD-2167 and the top-level design and detailed design headers developed for CAMP Ada designs.

2. GENERAL APPLICABILITY OF DOD-STD-2167

Current and future Government software projects will require contractors to meet the new Defense System Software Development standards of DOD-STD-2167. The standard, issued on June 4, 1985, supersedes the previous standard for software development, DOD-STD-1679A. Associated with DOD-STD-2167 are several Data Item Descriptions (DID's) which define the documentation requirements for Government contracted software projects. These DID's supersede those associated with 1679A and they also supersede several DID's associated with other military standards, such as the A, B5, and C5 specifications of MIL-STD-490, and the Part I and Part II specifications of MIL-STD-483. The new standard makes some significant changes in the software development and documentation process, but leaves other areas relatively unchanged.

Perhaps the most important difference between 2167 and 1679A is that 2167 requires a formal Software Specification Review (SSR), whereas 1679A did not. The purpose of the SSR is to determine whether the Software Requirements Specification (SRS), the Interface Requirements Specifications (IRSs), and the Operational Concept Document (OCD) form a satisfactory basis for proceeding into preliminary software design. This extra formal review may have a substantial impact on software requirements costs. However, these costs could conceivably be made up during later phases because the SSR and associated baselining will tend to make the software requirements more stable. The new review will prevent the introduction of new requirements or

changes to old requirements emerging from a Preliminary Design Review and often not completed until the time of the Critical Design Review, forcing major changes late in a program.

The following is a list of other requirements in 2167 which are not mentioned in 1679A.

a. Use approved structured tools and/or techniques for requirements analysis.
b. Utilize a Program Design Language (PDL) or other top-level design tool for preliminary design.
c. Incorporate human factors engineering principles in preliminary and detailed design.
d. Employ a Program Design Language during detailed design.
e. Establish Software Development Files (SDFs) for all units.
f. Utilize approved coding standards which may be those of the contractor.
g. Use independent personnel for CSCI testing and quality evaluations.
h. Establish a Software Development Library (SDL).

The documentation required by the two standards is very similar. A list of the documentation for 2167 is shown in Figure D-1 and a comparison of the two sets of documents is shown in Figure D-2. One difference is that 2167 requires a Computer Resources Integrated Support Document (CRISD). This document describes all the resources that are required to support the deliverable software when it is in operation, including such things as hardware, operating systems, other support software, personnel required, and training programs. Another difference in the documentation is that Software Trouble Reports and Software Change/Software Enhancement Proposals are no longer required as separate documents, but their formats are to be included in the Software Configuration Management Plan.

Redundancy is a major weakness of the primary engineering DID's of MIL-STD-2167. The Software Requirements Specification, Top-Level Design Document, and Detailed Design Document, each restate all functional areas in terms of input-processing-output. The CAMP group determined, through applying

DOD-STD-2167 DEFENSE SYSTEM SOFTWARE DEVELOPMENT DIDs

REQUIREMENTS CODE	CATEGORY	NUMBER	ACRONYM	NAME
Required	Engineering	DI-MCCR-80008	SSS	System/Segment Specification
Required	Management	DI-MCCR-80030	SDP	Software Development Plan
Included	Management	DI-MCCR-80009	SCMP	Software Configuration Management Plan
Included	Management	DI-MCCR-80010	SQEP	Software Quality Evaluation Plan
Included	Management	DI-MCCR-80011	SSPM	Software Standards and Procedures Manual
Required	Engineering	DI-MCCR-80025	SRS	Software Requirements Specification
Included	Engineering	DI-MCCR-80026	IRS	Interface Requirements Specification
Required	Engineering	DI-MCCR-80012	STLDD	Software Top Level Design Document
Required	Engineering	DI-MCCR-80031	SDDD	Software Detailed Design Document
Included	Engineering	DI-MCCR-80027	IDD	Interface Design Document
Included	Engineering	DI-MCCR-80028	DBDD	Data Base Design Document
Required	Engineering	DI-MCCR-80029	SPS	Software Product Specification
Required	Engineering	DI-MCCR-80013	VDD	Version Description Document
Required	Test	DI-MCCR-80014	STP	Software Test Plan
Required	Test	DI-MCCR-80015	STD	Software Test Description
Required	Test	DI-MCCR-80016	STPR	Software Test Procedure
Required	Test	DI-MCCR-80017	STR	Software Test Report
Vendor	Operational	DI-MCCR-80018	CSOM	Computer System Operator's Manual
Vendor	Operational	DI-MCCR-80019	SUM	Software User's Manual
Vendor	Support	DI-MCCR-80020	CSDM	Computer System Diagnostic Manual
Vendor	Support	DI-MCCR-80021	SPM	Software Programmer's Manual
Vendor	Support	DI-MCCR-80022	FSM	Firmware Support Manual
Required	Operational	DI-MCCR-80023	OCD	Operational Concept Document
Required	Support	DI-MCCR-80024	CRISD	Computer Resources Integrated Support Document
Required	Engineering	DI-E-3128	ECP	Engineering Change Proposal
Required	Engineering	DI-E-3134	SCN	Specification Change Notice

REQUIREMENTS CODE:

CODE	MEANING
Required	Typically required
Included	May be included in the previous required document
Vendor	May be vendor supplied

Figure D-1. DOD-STD-2167 Defense System Software Development DID's

DOD-STD-2167

DEFENSE SYSTEM SOFTWARE DEVELOPMENT DIDs

SUPERSEDED DIDs

ACRONYM	NAME	DOD-STD-1679A NUMBER	MIL-STD-483 NUMBER	MIL-STD-490 NUMBER	OTHER NUMBERS
SSS	System/Segment Specification		DI-E-3101	A Spec	
			DI-E-3117		
SDP	Software Development Plan	DI-A-2176A	DI-S-30567A		
SCMP	Software Configuration Management Plan	DI-E-20358			
		DI-E-21758			
		DI-E-2177A			
		DI-E-2178A			
SQEP	Software Quality Evaluation Plan	DI-R-2174A			DI-R-3521
SSPM	Software Standards and Procedures Manual				
SRS	Software Requirements Specification	DI-E-21368	DI-E-31198	B5 Spec	
			DI-E-30113		
			DI-E-30130A		
			DI-E-30139		
IRS	Interface Requirements Specification	DI-E-2135A		B5 Spec	
STLDD	Software Top Level Design Document	DI-E-2138A		CS Spec	
SDDD	Software Detailed Design Document	DI-S-2139A		CS Spec	
IDD	Interface Design Document	DI-E-2135A		CS Spec	
DBDD	Data Base Design Document	DI-S-2140A	DI-E-30144	CS Spec	DI-E-30150
SPS	Software Product Specification	DI-S-2141A	DI-E-312 /B	CS Spec	DI-A-30022
			DI-E-30110		DI-E-30112
			DI-E-30111		DI-S-30568
			DI-E-30140		
			DI-E-30145		
VDD	Version Description Document		DI-E-3121		
			DI-E-3122		
			DI-E-3123		
STP	Software Test Plan	DI-T-2142A	DI-T-3703A		DI-T-30715
STD	Software Test Description	DI-T-2143A			
STPR	Software Test Procedure	DI-T-2144A			DI-T-30716
STR	Software Test Report	DI-T-2156A	DI-T-3717A		
CSOM	Computer System Operator's Manual	DI-M-2148A			
SUM	Software User's Manual	DI-M-2145A	DI-M-30421		DI-M-3410
					DI-M-30404
					DI-M-30419
CSDM	Computer System Diagnostic Manual		DI-M-30422		DI-M-30408
SPM	Software Programmer's Manual				DI-M-3411
FSM	Firmware Support Manual				
OCD	Operational Concept Document				
CRISD	Computer Resources Integrated Support Document				DI-S-30569
ECP	Engineering Change Proposal				
SCN	Specification Change Notice				

NOTES: 1. The 2167 Software Configuration Management Plan incorporates the 1679A Software Change/Software Enhancement Proposal (DI-E-2177A), and the Computer Software Trouble Report (DI-E-2178A).

2. The 2167 Interface Requirements Specification and Interface Design Document together supersede the contents of the 1679A Interface Design Specification (DI-E-2135A).

Figure D-2. Comparison of DID's from DOD-STD-2167 and other Standards

these DID's that redundancy could be avoided by stating input - processing - output requirements once in the Software Requirements Specification, referring to that source in the Top-Level Design Document, and establishing the details of the algorithmic and data requirements in the Ada design code of the Detailed Design Document. Use of this method also enforces maintenance of the requirements document, as the design is updated, and reduces the likelihood of introducing errors during the design process.

The new standard concentrates on the activities, products, reviews, and baselines that are required during each phase of software development. It presents very little information about how to accomplish these tasks. The 1679A standard is much less clear about just what is to be accomplished, but it presents more how-to information. For example,

- Part of 1679A amounts to a coding standard. No such information is present in the body of 2167, but a design and coding standard is included as an appendix.
- Details about how patches to code are identified are in 1679A. DOD-STD-2167 states only that the contractor must keep track of changes.
- Details about how to conduct stress testing are in 1679A. These are not present in 2167.
- In requirements analysis, 1679A requires diagrams of equipment and software relations using internal and external data flow. The new standard only requires that the contractor use structured requirements analysis tools and/or techniques.

Functional requirements are emphasized by 1679A. Functional, performance, interface, and qualification requirements are treated equally by 2167. The CAMP study concluded that this life-cycle approach was one of the strengths of 2167.

3. SPECIFIC APPLICABILITY TO REUSABLE SOFTWARE, EXPERT SYSTEMS AND ADA

The CAMP program was one of the first to apply DOD-STD-2167 to a complete software development process. CAMP is also applying Ada under 2167. In

addition, it is the only program which has simultaneously addressed issues raised by the application of the standard to the development of a library of reusable software and an expert system to support automated parts engineering. Each of these three areas raises specific issues about the applicability of 2167 to particular aspects of software development.

a. Application to Reusable Software

The development of a library of reusable software necessitates a different approach from that used for single-use software. The Defense System Software Development Standard establishes the concept of a Computer System Configuration Item (CSCI) as the basis for software development and defines a CSCI as implementing a complete software subsystem. The standard specifies that "requirements shall be derived from the system requirements as defined in the system/segment specification" for each CSCI and shall be documented in a Software Requirements Specification. This Specification shall provide interface and data requirements for the CSCI plus detailed functional and performance requirements for each functional or structural component within the CSCI.

For the development of a reusable parts library, the CSCI consists of the requirements derived from a domain analysis. The CSCI must meet the system requirements of the entire domain which, in essence, constitutes the system/segment. This CSCI will not meet the entire software requirements of any one system and will not implement a complete software subsystem. Rather, the CSCI must provide parts which meet requirements common to a number of systems in the domain and, together with custom software, provide the capability of implementing a software subsystem.

There are three main issues surrounding the development of the parts library within the CAMP domain: what constitutes a part, how the part will be used, and how to specify a part so that it is reusable. The first issue will emerge from the process of decomposing system specifications into parts. The second issue will emerge from creating Ada design methods. The third issue will depend on the choice of a specification technique. Each of these three issues is covered in Section III of Volume I of this report.

b. Application to Expert Systems

An expert system provides extensive mechanisms for facilitating complex decision making processes. The two primary mechanisms are a Knowledge Base containing the expertise for a particular domain, and the Inference Engine to draw inferences using the knowledge base and data supplied by the user. The knowledge base includes facts, rules and mechanisms for structuring this knowledge.

For the CAMP program, the expert system concept is applied to the process of automated parts engineering, i.e., using a knowledge-based system for constructing software. The expert system contains the knowledge base which together with user inputs can create new software, combine it with existing parts, and produce a new system. These concepts are completely elaborated in Volume II of this Report.

The CAMP program tailored DOD-STD-2167 for the development of this expert system. The concept of a CSCI for an expert system exists for system functions with regards to the support mechanisms performing data base operations. Other aspects of the system functions do not decompose into lower level components, in a traditional top-down approach. These functions rely entirely on the inference mechanisms built into an expert system; they are recursive routines which do not follow the input-processing-output model. The standard is also not currently intended for the development of the knowledge base which is at the heart of an expert system. Therefore, the development process consists of separating these two areas, following 2167 for the traditional CSCI aspects, and approaching the inference mechanisms and knowledge base as separate software requirements areas.

c. Application to Ada

The use of DOD STD-2167 posed some problems for the CAMP program, but they were expected because the new standard does not specifically address software development in Ada. The Joint Logistics Commanders initiated the development of DOD-STD-2167 at the time the original Ada standard was under review. The Commanders decided at that time that issues of software development which related to Ada would be deferred until 2167 was published and the revision process began. The incorporation of Ada would be a part of that revision process. This revision process is currently in progress.

The major problems in applying Ada are not in the development process as much as in the documentation. The CAMP program concluded that Ada should not be a part of the requirements activity, but that requirements should remain independent of a particular implementation. The implementation language is of course a major consideration in developing requirements, but the study concluded that use of Ada constructs would restrict the development process. Ada is extremely well suited to the software design process, and its major design element, the package, is central to the CAMP program. The major activities specified under 2167 therefore posed no problems.

Applying Ada under the DID's of 2167 did create significant problems. A major aspect of Ada development in general, and reusable parts in particular, is the specification and design of data elements. These elements consist of types, constants, and objects and are not functional or interface requirements. They do not fit neatly into the Software or Requirements Specification documents. The concept of "Input - Processing - Output" does not apply because these data elements exist independent of one particular function. The design documents seem more suited to a Fortran implementation, with significant global data and minimal concern for the implementation at the design level. Use of Ada for design requires an essential concern for the implementation because the architectural design consists of compilable Ada packages, their specifications and the specification of exported entities such as subprograms and types. With considerable tailoring, the CAMP program was able to make use of the design DID's. These tailoring issues are covered below with specific reference to the use of Ada for specifying requirements, Ada for design, and Ada design and coding standards.

4. THE SOFTWARE REQUIREMENTS SPECIFICATION

The CAMP program effectively applied DOD-STD-2167 for the specification of CAMP parts and for the specification of most aspects of the parts engineering system. The program developed Software Requirements Specifications following the tasks described under section 5.1 of DOD-STD-2167. These Specifications are in accordance with DI-MCCR-80025, Software Requirements Specification. The structure of the DID required some tailoring to address the issues discussed above. The remainder of this section summarizes the changes made by the CAMP group to tailor the DID for use by CAMP.

The new standard is especially effective in clearly separating the requirements and design activities into three distinct areas: software requirements analysis, preliminary design, and detailed design. Previous standards such as DOD-STD-1679 and MIL-STD-483 do not make this distinction. and preliminary design is subsumed under requirements analysis and detailed design. This clear delineation, another of the strengths of 2167, also lends itself to establishing the manner in which Ada should be applied to the documentation process, as is discussed below.

a. The Use of Ada in the Specifications

The CAMP development effort demonstrated that the SRS should be prepared without reference to a specific Ada architecture. While other studies have attempted to use Ada as a software requirements language, the CAMP approach yielded more generalized requirements specifications, essential for developing reusable Ada software. The development of the requirements must, of course, take cognizance of Ada, to assure that the capabilities of Ada are fully exploited and to establish requirements which are conducive to Ada implementation. Furthermore, Ada terminology (e.g., tasks, packages, generics) is applied wherever appropriate. Nonetheless, keeping the requirements free of Ada constructs improved the readability and utility of the parts specifications.

In addition to documenting specifications, the SRS also establishes design guidelines. For the reusable parts library, designed using Ada, the

distinction between requirements and design allowed CAMP to create an architectural design based on Ada packages. The top-level, or architectural, design defines these packages, their interfaces, and major architectural issues in the design of the package bodies. The lower-level, or detailed, design shows the algorithmic design for subprograms defined in the top-level design and also defines the subprogram and data structures which are encapsulated within the common package bodies. These design issues are fully discussed in Section IV of Volume I of this report.

b. The Input-Processing-Output Description

Under DOD-STD-2167, the input, processing, and output sections document the processing which must be accomplished by the operations of a function. The requirements specification must define inputs to the part but they do not, in general, establish precise Ada data types and they do not define the data control, i.e., parameter data, common data, or local data. This precision is supplied in the design.

The SRS for the parts, and for the parts engineering system, document the data in a form easily recognized by the designers and users of these systems. This generic definition is well suited to parts which are to be designed as generic Ada parts, where the actual data type will be assigned by the part user. For the expert system, this definition can be easily incorporated into the design for the expert system.

Ada operations are used as a PDL in defining the processing requirements for all functions, both for the parts and for the parts engineering system. While this PDL definition is not a complete algorithmic description in most cases, it does show the logical and arithmetic operations which a function will accomplish. For the reusable parts the exact nature of these operations will be derived from the domain analysis. The parts engineering system operations come from the particular requirements of a knowledge based system.

The definition of output from a function will be similar to that of input. This data flow shows the output but does not define the data structures. The function's specification defines the outputs but does not establish data structures or control. The design of the part will define these features.

c. The Environment Description

The Software Requirements Specification DID does not address the relationships which must exist between parts in a reusable parts library. The CAMP parts are not intended for stand-alone use. They must be combined into larger contexts for effective utilization. However, the specific requirements for a part cannot assume the existence of the lower-level packages. A user may wish to use the CAMP navigation packages, but not want to use the CAMP data types. The statement of a part environment will establish the context of a given part, independent of other CAMP parts. The environment may include a description of constants or of data types which are required by a part. For those parts requiring special processing, the environment may indicate external subprograms which must exist for the part to function. Finally, the environment may establish the data which must be supplied for initialization of the state of a given part, indicating that the part may be designed as a generic.

5. THE SOFTWARE TOP-LEVEL DESIGN DOCUMENT

The DID for the Software Top-Level Design Document establishes this document as describing the structure and organization of a particular CSCI. The document establishes the Top Level Computer Software Components (TLCSC's) and the interfaces between these components. This document is sufficient for the top-level design of the parts engineering system by limiting the context of design to the support functions and by not employing Ada for design.

The document does not meet all the documentation needs of the reusable parts design. The documentation of the Top-Level design for the CAMP parts must describe the architecture of part packages and detail the interfaces between packages. This will require TLCSC's which address the following issues:

- The package context (the list of external packages which are needed)
- The decomposition of the TLCSC into LLCSC's
- Ada Design of the specification of the TLCSC and its LLCSC's
- Major entities which are local to the package body
- Externally callable entries (where tasking is used)
- Requirements for instantiation and other use of a part
- Global processing and output

These requirements must be met both in the TLDD and in the header of the design code itself.

The DID Software Top-Level Design Document does not adequately cover these issues. As described above, the DID seems to be directed towards a design which features data passing through shared data, rather than parameter passing, and parameterless subroutines employed for structural reasons, rather than functional or object-oriented decomposition. This architecture for a TLCSC is not compatible with the object-oriented nature of an Ada package specification. Therefore, the TLDD is not sufficient for our documentation needs.

Much of the information that properly belongs to a TLCSC designed using Ada has been placed in the Software Detail Design Document (e.g., the TLCSC decomposition, and LLCSC interfacing). The CAMP project has determined that this information must appear in the top-level design description. This will require that the DID for top-level design be modified to include architectural information highlighting the structure of the TLCSC down to the unit level, where units are externally callable. It should also include structural information which is required for the detailed design of these external interfaces. In Ada terms, the TLDD will document the Ada specification plus major data structures and processing needs of the package body.

The documentation of the top-level design, will in general refer back to the Software Requirements Specification for the top-level algorithmic description. This should be sufficient information for the needs of detailed design, to avoid redundancy between the SRS and the TLDD. For algorithms which have not been adequately documented in the SRS, especially those in the

domain independent area, the TLDD must include input-processing-output information.

6. THE SOFTWARE DETAILED DESIGN DOCUMENT

The DID for the Software Detailed Design Document establishes this document as describing in detail the organization and structure of a particular CSCI. This document contains data generated under the work task described by sections 5.3.1.2 and 5.3.2.3 of DOD-STD-2167, Defense System Software Development. CAMP produced a detailed design for the reusable parts only.

The CAMP program tailored the structure of this DID to meet the needs of reusable software components. The DID is intended for use in describing the design of a single application Computer Software Configuration Item (CSCI). The DID has been tailored for describing the design of parts used individually or on a group basis. In addition, elements of this DID were already incorporated into the paragraph structure of the Software Top Level Design Document as explained above.

The detailed design continued the application of Ada for design which began under top-level design. At the detailed level, the Ada consists of the representation in Ada of all algorithms specified in the SRS, plus any support processing which the top-level design requires. At this level, the directions of the DID are adequate for the detailed design process. The DID does not adequately address specific Ada features, such as the Ada package body, but tailoring the DID to satisfy these features is not a significant issue. This contrasts, of course with the major changes made in the top-level design.

7. CODING STANDARDS FOR ADA

Several problems exist in the coding standards developed under DOD-STD-2167 when applied to Ada. These problems are summarized in Figure D-3. The standard also establishes data to be included in code headers. Code headers developed by the CAMP program for Ada design used in the top-level and detailed design appear on the following pages.

PARA. #	DISCUSSION
All	This format is only applicable to task bodies, subprogram bodies, and the block containing a sequence of statements at the end of a package body. It does not adequately address Ada specification.
30.3.2	The for clause is not explicitly named. This paragraph could mention that the for clause is a variation of do while.
30.3.3.d	Unique names cannot apply where overloading is used.
30.3.3.e	This "standard format" precludes the use of declare blocks in Ada.
30.3.3.f	This cannot apply to task bodies if they are to be considered a unit.
30.3.5	Language "keywords" in Ada may be construed as including system-defined identifiers (e.g., those defined in package STANDARD). These identifiers must be overloadable.
30.3.6	Ada requires "mixed-mode" operations in some cases. (See package CALENDAR.) It allows other mixed-mode operations for arithmetic operators via overloading those defined in package STANDARD.
30.3.8	Nesting in Ada may include nested packages, tasks, etc. Does this nesting guideline apply to Ada units or to nesting of expressions (loop, if ... then, etc.) within a unit?

Figure D-3. Problems in Applying DOD-STD-2167 Coding Standards to Ada Design

a. Design Code Header Information for Top-Level Design

```
--  TLCSC Name
--  The name shall be descriptive of the processing performed by the
--  TLCSC.

--  TLCSC Identification Number
--  The design identification number used to identify the TLCSC for
--  configuration management

--  Detailed overview of TLCSC purpose.
--  For generic units this section shall also provide details of the
--  capabilities provided by generic parameters (analogous to states
--  of operation)

--  Requirements trace
--  Document SRS requirements met by the TLCSC.
--  May reference a block diagram to illustrate source of inputs and
--  destination of outputs of TLCSC.  Diagram should show allocation
--  of CSCI requirements.

--  Context of TLCSC
--  Describe context of TLCSC (packages which are withed, or are
--  otherwise visible and are referenced in the TLCSC).  Describe
--  what services of these packages (data types, objects, functions)
--  are used.  This will describe global data used by the TLCSC.

--  Exported Entities
--  Describe data objects, data types, subprograms and packages
--  defined by the TLCSC.  Summarize in tabular form to show services
--  exported by the TLCSC.  Also, describe in detail all exported
--  entities:

--    Data objects
--    Describe data objects exported by the TLCSC.  This shall
--    include:
--       • Name of object
--       • Type of data
--       • Value if a constant
--       • Brief description of data

--    Data types
--    Describe data types exported by the TLCSC.  This shall
--    include:
--       • Name of type
--       • Range of type
--       • Predefined operators
--       • Special operators
--       • Brief description of type

--    Subprograms
--    Describe the decomposition of the TLCSC into processing
--    entities which shall become Lower Level CSC's and units.  For
--    each LLCSC or unit defined by the decomposition provide the
--    following information:
```

```
--       • Name
--       • Abstract describing purpose of subprogram.  For
--         generic subprograms this shall include details of the
--         capabilities provided by generic parameters.
--       • Requirements trace
--       • Input data (parameters or global data)
--       • Processing algorithms
--       • Error conditions not handled immediately by the entity
--       • Outputs (parameters or global data)

--    Packages
--    Describe the decomposition of the TLCSC into packages which
--    shall become Lower-Level CSC's and units.  For each package
--    defined by this decomposition provide the following
--    information:
--       • Name
--       • Abstract describing purpose of subprogram.  For
--         generic subprograms this shall include details of the
--         capabilities provided by generic parameters.
--       • Requirements trace
--       • Entities exported

-- Local Entities
-- Describe the following entities which will be local to the TLCSC:

--    • Local data structures, encapsulated in the package body.
--    • Files or data bases used by the TLCSC and not by any other
--      TLCSC
--    • Data types defined local to the TLCSC and not used by any
--      other TLCSC.
--    • Generic subprograms or packages defined local to the TLCSC
--      and used by entities exported by the TLCSC.
-- Provide information describing the use of these local entities by
-- other entities within the TLCSC

-- Additional "coding" information
--    • Security level
--       None
--       Confidential
--       Secret

--    • Calling sequence

--    • History
--       Prepared by
--       Baseline data

--    • Revision History
--       Revised by
--       Revision date
--       Revision reason: brief description
```

b. Design Code Header for Detailed Design

```
-- LLCSC/Unit Name

-- LLCSC/Unit Number

-- Purpose

-- Requirements Trace

-- Utilization of other elements (Types, Procedures, Generics, from other
-- packages)

-- Data Specification

--   Generic Parameters

--     Name      Type      Mode      Description

--   Parameters

-     Name      Type      Mode      Description

--   Global Data

--     Name      Type      Source    Description

--   Local Data

-     Name      Type      Description

--   Local Entities (Instantiation or declaration of program units)

--     Name      Type      Source    Description

-- Exceptions

-- Security

-- Calling Sequence

-- History
--   Prepared By:
--   Baseline Date:

-- Revision History
--   Revised By:
--   Revision Date:
--   Revision Reason:
```

8. RECOMMENDATIONS

The CAMP program revealed a number of areas for improvements to DOD-STD-2167 and the accompanying DID's. The following list summarizes these areas:

a. Develop tailoring guidelines for applying DOD-STD-2167 to the development of reusable software libraries.
b. Develop tailoring guidelines for applying DOD-STD-2167 to the development of expert systems.
c. Revise Appendix III of DOD-STD-2167 to incorporate coding standards for Ada in accordance with Paragraph 7 of this Appendix.
d. Revise Software Requirements Specification DID to take data typing into account.
e. Revise Software Top-Level Design Document to include complete architectural breakdown of a system into Ada package specifications as described in Paragraph 5 of this Appendix.
f. Revise Software Top-Level Design Document to refer to the Software Requirements Document for input - processing - output information.

REFERENCES

[1] Neighbors, J.M., **Software Construction Using Components**, Ph.D. Dissertation, Department of Information and Computer Science, Univ. of California, Irvine, Technical Report 160, 1980.

[2] Neighbors, J.M., **DRACO User's Manual**, Version 1.3, Department of Information and Computer Science, University of California, Irvine, September 1984.

[3] Belady, L.A., "Evolved Software for the 80's" **IEEE COMPUTER**, February, 1979.

[4] Lanergan, R.G. and Grasso, C.A., "Software Engineering with Reusable Designs and Code" **IEEE Transactions on Software Engineering**, Vol. SE-10, No. 5, Sept. 1984.

[5] Sundfor, S., **DRACO Domain Analysis for Realtime Applications: The Analysis**, Reuse project RTP 015, Department of Information and Computer Science, Univ. of California, Irvine, June 1983.

[6] Sundfor, S., **DRACO Domain Analysis for Realtime Applications: Discussion of the Results**, Reuse project RTP 016, Department of Information and Computer Science, Univ. of California, Irvine, June 1983.

[7] McDonnell Douglas Astronautics Co., **Computer Program Performance Specification for the Cruise Missile Land Attack Guidance System (BGM-109C)**, Revision A, Prepared for the Joint Cruise Missile Program Office, JCM 1654, February 1984.

[8] McDonnell Douglas Astronautics Co., Computer Program Design Specification for the Cruise Missile Land Attack Guidance System (BGM-109C), Revision B, Prepared for the Joint Cruise Missile Program Office, JCM 1655, March 1984.

[9] McDonnell Douglas Astronautics Co., Computer Program Performance Specification for the Cruise Missile Land Attack Guidance System (BGM-109A), Revision B, Prepared for the Joint Cruise Missile Program Office, JCM 1652, October 1983.

[10] McDonnell Douglas Astronautics Co., Computer Program Performance Specification for the Cruise Missile Land Attack Guidance System (BGM-109G), Revision C, Prepared for the Joint Cruise Missile Program Office, JCM 1658, May 1984.

[11] McDonnell Douglas Astronautics Co., Computer Program Description Document - Kalman Data Processor Module (BGM-109G), Prepared for the Joint Cruise Missile Program Office, JCM 1624, August 1982.

[12] McDonnell Douglas Astronautics Co., Computer Program Description Document - Kalman Initialization Module (BGM-109G), Prepared for the Joint Cruise Missile Program Office, JCM 1621, Sept 1982.

[13] McDonnell Douglas Astronautics Co., Computer Program Description Document - Kalman Covariance Matrix Propagation Module (BGM-109G), Prepared for the Joint Cruise Missile Program Office, JCM 1627, November 1982.

[14] McDonnell Douglas Astronautics Co., Computer Program Description Document - Kalman System Update Module (BGM-109G), Prepared for the Joint Cruise Missile Program Office, JCM 1630, September 1982.

[15] McDonnell Douglas Astronautics Co., **Computer Program Development Specification for the Unaided Tactical Guidance Strapdown Inertial Navigation Program**, MDC A6216, Prepared for the U. S. Air Force Armament Laboratory, April 1981.

[16] McDonnell Douglas Astronautics Co., **Computer Program Performance Specification for the Guidance and Navigation Program of the AGM-109H Medium Range Air-to-Surface Missile**, Prepared for the Joint Cruise Missile Program Office, JCM 1023, August 1983.

[17] McDonnell Douglas Astronautics Co., **Computer Program Design Specification for the Guidance and Navigation Program of the AGM-109H Medium Range Air-to-Surface Missile**, Prepared for the Joint Cruise Missile Program Office, JCM 1024, March 1984.

[18] McDonnell Douglas Astronautics Co., **Data Base Design Document for the MRASM-H Guidance and Navigation Computer Guidance and Navigation Application Software**, Prepared for the Joint Cruise Missile Program Office, JCM 1025, October 1982.

[19] McDonnell Douglas Astronautics Co., **Computer Program Performance Specification for the Guidance and Navigation Computer - Operational Flight Software for the Tactical Land/Sea Attack Tomahawk Cruise Missile Model AGM-109L Medium Range Air-to-Surface Missile**, Prepared for the Joint Cruise Missile Program Office, JCM 1032, March 1982.

[20] General Dynamics Convair Division, **System Segment Specification for the Midcourse Guidance Demonstration Flight Software Phase III**, SS64-16700I, Prepared for the U. S. Air Force Armament Laboratory, June 1984.

[21] General Dynamics Convair Division, **Design Requirements Bulletin - Midcourse Guidance Demonstration (MGD)**, DRB-83-002, Revision A, Prepared for the U. S. Air Force Armament Laboratory, Nov 1983.

[22] General Dynamics Convair Division, **Computer Program Development Specification for the Midcourse Guidance Demonstration Guidance and Navigation (GAN) Program Phase III**, CS64-16740C, Prepared for the U. S. Air Force Armament Laboratory, February 1983.

[23] McDonnell Douglas Astronautics Co., **TASM Program Performance Specification (BGM-109B)**, Prepared for the Joint Cruise Missile Program Office, AS 4926, Revision C, SECRET, July 1984.

[24] McDonnell Douglas Astronautics Co., **TASM Program Design Specification (BGM-109B)**, Prepared for the Joint Cruise Missile Program Office, AS 4937, Revision B, SECRET, August 1984.

[25] McDonnell Douglas Astronautics Co., **Computer Program Development Specification for the Harpoon Midcourse Guidance Unit Computer Program**, Prepared for the Naval Air Systems Command, XAS 4099, CONFIDENTIAL, August 1984.

[26] McDonnell Douglas Astronautics Co., **Computer Program Performance Specification for the Cruise Missile Land Attack Guidance System (AGM-109, BGM-109A, and BGM-109C)**, 70C187010-1001, CONFIDENTIAL, February 1980.

[27] McDonnell Douglas Astronautics Co., **UpSTAGE Technology Report: Guidance Analysis and Simulation - Volume I**, MDC G3234, SECRET, September 1972.

[28] Bell Laboratories, **Program Design Specification for the M2 Terminal Guidance Routine**, D.3.0.70.A.5, CONFIDENTIAL, June 1982.

[29] Bell Laboratories, **WSMR - Guidance Program and Non-Fire Exercise Program Specification**, Revision 4, DAAH01-68-C-1237, CONFIDENTIAL, July 1972.

[30] Rich, C. and R.C. Waters, "Formalizing Reusable Software Components," Proceedings of the Workshop on Reusability in Programming, September 1983.

[31] Chandersekaran, C.S. and M.P. Perriens, "Towards an Assessment of Software Reusability," Proceedings of the Workshop on Reusability in Programming, September 1983.

[32] Soloway, E. and K. Ehrlich, "What DO Programmers Reuse? Theory and Experiment," Proceedings of the Workshop on Reusability in Programming, September 1983.

[33] Cavaliere, M.J. and P.J. Archambeault, "Reusable Code at the Hartford Insurance Group," Proceedings of the Workshop on Reusability in Programming, September 1983.

[34] Boehm, Barry W., Software Engineering Economics, Prentice-Hall, Inc., 1981.

INITIAL DISTRIBUTION

DTIC-DDAC	2	NASA (CODE RC)	1
AUL/LSE	1	NSA	1
FTD/SDNF	1	NTSC/CODE 251	1
HQ USAFE/INATW	1	AD/XRB	1
AFWAL/FIES/SURVIAC	1	LOCKHEED (DR SURY)	1
AFATL/DOIL	2	HUGHES (MR BARDIN)	1
AFATL/CC	1	IBM (MS VESPER)	1
AFCSA/SAMI	1	RAYTHEON (WILLMAN)	1
AFATL/CCN	1	SOFTECH INC (MS BRAUN)	1
AFATL/FXG	10	MITRE CORP (MR SYLVESTER)	1
ASD/RWX	1	AEROSPACE CORP (MR HOGAN)	1
SPAWAR (814AB NC #1)	1	TEXAS INSTRUMENTS (MR FOREMAN)	1
SPAWAR (CODE 613)	1	RATIONAL (MR BOOCH)	1
NRL (CODE 5150)	1	ROCKWELL INTERNATIONAL (MR GRIFFIN)	1
ASD/XRX	1	MITRE CORP (MS CLAPP)	1
AFWAL/AAA-2	1	NOSC (CODE 423)	1
HQ AFSC/PLR	3	GENERAL DYNAMICS (MR MURRAY)	1
INST OF DEFENSE ANALYSES	5	HONEYWELL INC (MS GIDDINGS)	1
STARS JOINT PROGRAM OFFICE	1	HONEYWELL INC (DR FRANKOWSKI)	1
USA MATERIEL CMD/AMCDE-SB	1	HUGHES DEFENSE SYS DIV (S.Y. WONG)	1
NAVAL SEA SYS CMD (SEA 61R2)	1	NOSC (MR WASILAUSKI)	1
NAVAL AIR DEVELOPMENT CTR (CODE 50C)	1	NAC (N)	1
BMDATC	1	TASC (MR SERNA)	1
NSWC/CODE N20	1	MARTIN MARIETTA (MR CUDDIE)	1
NAVAL UNDERSEA SYS CTR (CODE 3511)	1	TASC (DR CRAWFORD)	2
NOSC/CODE 423	1	SYSCON CORP (DR BRINTZINHOFF)	1
AFWAL/AAAF-2	1	ADA TRAINING SECTION	1
USA EPG/STEEP-MT-DA	1	COMPUTER SCI CORP (MR FRITZ)	1
AFSC/DLA	1	LINKABIT (MR SIMON)	1
USA MSL CMD/AMSMI-OAT	1	RAYTHEON (MR GINGERICH)	1
SPAWAR (CODE 06 NC #1)	1	UNIVERSITY OF COLORADO	1
ASD/EN	1	MCDONNELL AIRCRAFT (MR MCTIGUE)	1
AD/ENE	1	HUGHES AIRCRAFT (MR NOBLE)	1
SD/ALR	1	GENERAL DYNAMICS (MR PRZYBYLINSKI)	1
RADC/COEE	2	AJPO (MS CASTOR)	1
NRL/CODE 7590	1	NWC (CODE 3922)	1
AFSC/SDZD	1	NAVAIRSYSCOM HQS	1
HQ USAF/RDPV	1	ASD/ENASF	1
AD/ENSM	1	ASD/ENA	5
BMO/ENSE	1	UNIVERSITY OF TEXAS	1
ESD/ALS	1	MARTIN MARIETTA (MR SORONDO)	1
ESD/ALSE	2	MCDONNELL DOUGLAS (DR MCNICHOLL)	2
AJPO	1	GENERAL DYNAMICS (MR SCHNELKER)	1
AFATL/AS	1	TRW (MR SHUGERMAN)	1
AFATL/SA	1	RATIONAL (MR HAKE)	1
AD/XR	1	WESTINGHOUSE (MR GREGORY)	2
WIS SPMO/ADT	1	GENERAL DYNAMICS (DR TABER)	1
AFWAL/AAAF	1	BOEING COMMERCIAL AIRPLANE CO	1

INITIAL DISTRIBUTION (CONTINUED)

ROCKWELL INTERNATIONAL (MIKULSKI)	1	ROCKWELL (MS KIM)	1
BOEING AEROSPACE (MR HADLEY)	1	SCIENCE APPLICATIONS INTERNATIONAL	
IBM (MR MCCAIN)	1	(MR STUTZKE)	1
FSU (DR BAKER)	1	NSWC (U-33)	1
AEROSPACE CORP (MR LUBOFSKY)	1	COMPUTER SOFTWARE & SYSTEMS	1
DATA GENERAL (MR DAMASHEK)	1	GRUMMAN DATA SYSTEMS (MR MARKMAN)	1
SDC (MR HERMANN)	1	AFWAL/FIGL	1
WINTEC (MR CONNEL)	1	AFWAL/FIGX	1
NORTHROP (MR OHLSEN)	1	AFWAL/AARI-1	1
ARC (MR ROBERSON)	1	EMERSON ELECTRIC (MR BYRNES)	1
GRC (DR ALBRITTON)	1	TASC (MR JAZMINSKI)	1
AFWL/NTSAC	1	E-SYSTEMS (MR SNODGRASS)	1
USA MSL CMD/SCI INFO CTR	1	NTSC (CODE 742)	1
NSWC/TECH LIB	1	UNIVERSITY OF ALABAMA	1
NWC (CODE 343)	1	MCDONNELL DOUGLAS (MR VILLACHICA)	1
OO-NAVAL RESEARCH (CODE 784DL)	1	SANDERS ASSOCIATES (MR FRY)	1
NAVAL POSTGRAD SCHOOL (CODE 1424)	1	AFSC/PLR	1
DEFENSE COMMUNICATIONS AGENCY	1	AFSC/DLA	1
NASA AMES RESEARCH CTR (CHAPMAN)	1	WESTINGHOUSE (MR SQUIRE)	1
NASA LANGLEY RESEARCH CTR (MOTLEY)	2	NAVAIR SYSCOM (AIR 54662)	1
RAND CORP/AFELM	1	BOEING AEROSPACE (MR BOWEN)	1
AVCO SYS DIV/RESEARCH LIB	1	GOULD INC (DR ARORA)	1
AVCO-EVERETT RES LAB/TECH LIB	1	LOCKHEED (MR PINCUS)	1
FAIRCHILD IND/INFO CTR	1	INTERMETRICS (MR BROIDO)	1
GENERAL DYNAMICS, CONVAIR DIV	1	GENERAL ELECTRIC (MS MICKEL)	1
GENERAL DYNAMICS, FT WORTH DIV	1	HUGHES AIRCRAFT (DR HUANG)	1
HONEYWELL/TECH LIB	1	SYSTEMS DEVELOPMENT (MR SIMOS)	1
HUGHES AIRCRAFT/TECH LIB	1	GENERAL DYNAMICS (MR WARNER)	1
HUGHES AIRCRAFT/MSL SYS GP/TECH LIB	1	CARNEGIE-MELLON UNIVERSITY	1
LOCKHEED/TECH LIB	1	COMPUTER TECH ASSOCIATES (HEYLIGER)	1
LOCKHEED/TECH INFO CTR	1	NORTHROP CORP (MR SWAN)	1
MARTIN MARIETTA/TECH LIB	1	CNI SOFTWARE (SINGER KEARFOTT DIV)	1
MCDONNELL DOUGLAS/TECH LIB	1	LOCKHEED (MR COHEN)	1
MCDONNELL DOUGLAS/LIB SVCS	1	MCDONNELL DOUGLAS (SANDY COHEN)	2
NORTHROP CORP/AIRCRAFT DIV	1	DRAPER LABORATORY (MR DAVID)	1
RAYTHEON/TECH LIB	1	DRAPER LABORATORY (DR DEWOLF)	1
VOUGHT CORP/LIB	1	GOULD INC (MR WILLIS)	1
SDC (MR MILLAR)	1	ADVANCED TECHNOLOGY (MR COOPER)	1
INTERMETRICS (MR ZIMMERMANN)	1	HQ USAF/RDST	1
ROCKWELL/TECH INFO	1	AFPRO/EN	1
TRW (MR MUNGLE)	1	AFPRO/TRW/EPP	1
AT&T (MR MAY)	1	LOCKHEED (MR DORFMAN)	1
AVCO SYS TEXTRON (MR SOHN)	1	HONEYWELL (MR LANE)	1
BOEING ELECTRONICS (MR MEDAN)	1	SPERRY A&MG (MR ROSS)	1
USA CECOM/DRSEL-TCS-MCF	1	UNITED TECHNOLOGIES (STOLZENTHALER)	1
USA CECOM/DRSEL-TCS-ADA	1	LTV AEROSPACE & DEFENSE	1
UNIVERSITY OF WASHINGTON	1	EL66 B 4610 (MR FREEMAN)	1
AFATL/GRC	1	SYSTEMS DEVELOPMENT CORP (ATCHLEY)	1
USA-ARDC	1		

INITIAL DISTRIBUTION (CONCLUDED)

ROCKWELL (DILLHUNT)	1
IBM FED SYSTEMS DIV (MR ANGIER)	1
JET PROPULSION LAB (MR KRASNER)	1
MARTIN MARIETTA ORLANDO AEROSPACE	1
GENERAL ELECTRIC (MR DELLEN)	1
AFWAL/POFIC	1
LOCKHEED (READY)	1
BOEING AEROSPACE (MR DOE)	1
FORD AEROSPACE (DR FOX)	1
LOCKHEED SOFTWARE TECH CTR (LYONS)	1
IBM (MR BENESCH)	1
MITRE CORP (MR SHAPIRO)	1
MITRE CORP (MR MAGINNIS)	1
GRUMMAN AEROSPACE (MR POLOFSKY)	1
MCDONNELL DOUGLAS (KAREN L SAYLE)	1
MCDONNELL DOUGLAS (GLENN P LOVE)	1
ROME AIR DEVELOPMENT CTR (CHRUSCICKI)	1
RAYTHEON EQUIPMENT DIV (MS SILLERS)	1
AT&T BELL LABS (MS STETTER)	1

(The reverse of this page is blank)

SUPPLEMENTARY

INFORMATION

DEPARTMENT OF THE AIR FORCE
WRIGHT LABORATORY (AFSC)
EGLIN AIR FORCE BASE, FLORIDA, 32542-5434

ERRATA
AD-B102654

REPLY TO ATTN OF: MNOI

13 Feb 92

SUBJECT: Removal of Distribution Statement and Export-Control Warning Notices

TO: Defense Technical Information Center
ATTN: DTIC/HAR (Mr William Bush)
Bldg 5, Cameron Station
Alexandria, VA 22304-6145

1. The following technical reports have been approved for public release by the local Public Affairs Office (copy attached).

	Technical Report Number	AD Number
1.	88-18-Vol-4	ADB 120 251
2.	88-18-Vol-5	ADB 120 252
3.	88-18-Vol-6	ADB 120 253
4.	88-25-Vol-1	ADB 120 309
5.	88-25-Vol-2	ADB 120 310
6.	88-62-Vol-1	ADB 129 568
7.	88-62-Vol-2	ADB 129 569
8.	88-62-Vol-3	ADB 129-570
9.	85-93-Vol-1	ADB 102-654
10.	85-93-Vol-2	ADB 102-655
11.	85-93-Vol-3	ADB 102-656
12.	88-18-Vol-1	ADB 120 248
13.	88-18-Vol-2	ADB 120 249
14.	88-18-Vol-7	ADB 120 254
15.	88-18-Vol-8	ADB 120 255
16.	88-18-Vol-9	ADB 120 256
17.	88-18-Vol-10	ADB 120 257
18.	88-18-Vol-11	ADB 120 258
19.	88-18-Vol-12	ADB 120 259

2. If you have any questions regarding this request call me at DSN 872-4620.

LYNN S. WARGO
Chief, Scientific and Technical
Information Branch

1 Atch
AFDTC/PA Ltr, dtd 30 Jan 92

ERRATA

DEPARTMENT OF THE AIR FORCE
HEADQUARTERS AIR FORCE DEVELOPMENT TEST CENTER (AFSC)
EGLIN AIR FORCE BASE, FLORIDA 32542-5000

REPLY TO ATTN OF: PA (Jim Swinson, 882-3931) 30 January 1992

SUBJECT: Clearance for Public Release

TO: WL/MNA

The following technical reports have been reviewed and are approved for public release: AFATL-TR-88-18 (Volumes 1 & 2), AFATL-TR-88-18 (Volumes 4 thru 12), AFATL-TR-88-25 (Volumes 1 & 2), AFATL-TR-88-62 (Volumes 1 thru 3) and AFATL-TR-85-93 (Volumes 1 thru 3).

Virginia N. Pribyla

VIRGINIA N. PRIBYLA, Lt Col, USAF
Chief of Public Affairs

AFDTC/PA 92-039