

DTIC FILE COPY

1

AD-A225 209



DTIC
ELECTE
JUL 3 1 1990
S D^{CS} D

CAMP

DISTRIBUTION STATEMENT A

Approved for public release
Distribution Unlimited

Developing And Using Ada Parts
In

Real-Time Embedded Applications



AJPO

**MCDONNELL
DOUGLAS**

96 07 1 080



REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
1. AGENCY USE ONLY (Leave blank)				
2. REPORT DATE 27 April 90		3. REPORT TYPE AND DATES COVERED Final 20 Jul 88 - 27 Apr 90		
4. TITLE AND SUBTITLE Developing Ada Using Ada Parts In Real-Time Embedded Applications		5. FUNDING NUMBERS C: F08635-88-C-0002 PE: 64740F PR: 3672 TA: 01 WU: 01		
6. AUTHOR(S) Constance Palmer AFATL Program Manager: James W. Lund (AFATL/FXG)				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) McDonnell Douglas Missile Systems Company P.O. Box 516 St Louis MO 63166		8. PERFORMING ORGANIZATION REPORT NUMBER		
9. SPONSORING MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Armament Laboratory Aeromechanics Division Guidance and Control Branch AFATL/FXG Eglin AFB FL 32542-5434		10. SPONSORING MONITORING AGENCY REPORT NUMBER AFATL-TR-90-67		
11. SUPPLEMENTARY NOTES This report was not edited in the Technical Reports Section.				
12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution is unlimited		12b. DISTRIBUTION CODE		
13. ABSTRACT (Maximum 200 words) This manual provides practical guidance in the development and use of reusable Ada software for mission critical real-time embedded applications. It covers the reusable software development lifecycle, as well as the impact of software reuse on the development of real-time embedded applications. The manual takes a broad approach to the issues encountered during the entire lifecycle of parts development and use.				
14. SUBJECT TERMS Reusable Software, Missile Software, Software Generators, Ada Parts, Software Parts		15. NUMBER 225		
17. SECURITY CLASSIFICATION OF REPORT Unclassified		18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified		19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified
				20. LIMITATION OF ABSTRACT SAR

DEVELOPING AND USING ADA PARTS IN REAL-TIME EMBEDDED APPLICATIONS

Common Ada Missile Packages — Phase 3 (CAMP-3)
CONTRACT NO. F08635-88-C-0002
CDRL SEQUENCE NO. A008

27 April 1990

Accession For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

Prepared for:
Air Force Armament Laboratory/FXG
Munition Systems Division
Guidance and Control Branch
Eglin Air Force Base, Florida 32542



Prepared by:
McDonnell Douglas Missile Systems Company
P.O. Box 516
St. Louis, MO 63166

90 07 11 080

PREFACE

This manual provides practical guidance in the development and use of reusable Ada software for mission critical real-time embedded (RTE) applications. It covers the reusable software development lifecycle, as well as the impact of software reuse on the development of RTE applications. This manual is not intended as a set of coding guidelines for writing reusable software, rather, it takes a broader approach to the issues encountered during the entire lifecycle of parts development and use.

Two major challenges were faced in developing this manual: (1) addressing the diverse needs of a broad spectrum of readers; and (2) presenting practical information that has immediate applicability. The reader is assumed to have some familiarity with Ada in order to understand the examples and the Ada-specific issues that are presented.

This manual was developed as part of the Common Ada Missile Packages Phase 3 (CAMP-3) program. It was developed by the Computer and Software Technology Center of the McDonnell Douglas Missile Systems Company, St. Louis, Mo., and was sponsored by the United States Air Force Armament Laboratory (FXG) at Eglin Air Force Base, Florida.

ACKNOWLEDGEMENT

Special thanks to MSD/YB Air-to-Surface Ballistic Weapons SPO; Computer Resource Management Technology (CRMT) Program, ESD/AVSE; and AJPO/ATIP for their support of this project; and to Chris Herr for developing many of the examples and case studies; to the reviewers for their valuable comments and suggestions; and to the CAMP-1 and CAMP-2 staff upon whose work much of this current effort draws.

Table of Contents

INTRODUCTION

1. INTRODUCTION	3
1.1 Motivation for Software Reuse	3
1.2 Barriers to Software Reuse	4
1.3 Reuse in the RTE Community	7
1.4 Summary	7
1.5 About the Manual	7

SECTION I: FOUNDATIONS OF SOFTWARE REUSE

2. PARTS DEVELOPMENT OVERVIEW	11
3. DOMAIN ANALYSIS	13
3.1 Domain Definition	14
3.2 Domain Representation	16
3.3 Domain Exploration	17
3.3.1 Analysis of Existing Applications	18
3.3.2 Application Characteristics	19
3.4 Domain Modeling	19
3.5 Component Specification	20
3.6 Component Development Verification	21
3.7 Classification of Parts	21
3.7.1 Faceted Classification Schemes	22
3.7.2 Enumerative Classification Schemes	22
3.7.2.1 Abstraction Level	22
3.7.2.2 Domain Relevance	24
3.7.2.3 Descriptive Attributes	24
3.7.2.4 Instantiation Techniques	24
3.7.3 Domain-Based Classification	27
3.8 Tools	28
3.9 Management of the Domain Analysis Process	29
3.10 A Case Study: The CAMP Kalman Filter Parts	29
3.10.1 The Domain Analysis	29
3.10.2 Architecture of the Kalman Filter Parts	31
3.10.3 Implementation	32
3.10.4 Parts Use and Refinement	33
3.11 Summary	33
4. REQUIREMENTS DEFINITION	35
4.1 Requirements Development	35
4.2 Requirements Definition	36
4.3 Requirements Review	38
4.4 Definition of a Part	38
5. ARCHITECTURAL DESIGN of SOFTWARE PARTS	41
5.1 Overview of the CAMP Semi-Abstract Data Type Method	41
5.2 Alternative Design Methods	42
5.3 Using the Generic Method to Design Parts	44
5.4 Interaction Between Parts	47
5.4.1 Parts Build on Other Parts	48
5.4.2 Parts Work Together	49
5.4.3 Parts Facilitate Use of Other Parts	50

5.5 Design Considerations	51
5.6 Design Guidelines	56
5.7 Case Studies	70
5.7.1 Case Study 1: Development of the CAMP Coordinate Vector/Matrix Algebra Package	70
5.7.1.1 Domain Analysis	70
5.7.1.2 Design	71
5.7.1.3 Maintenance	75
5.7.1.4 Further Enhancements	78
5.7.2 Case Study 2: Data Typing	79
5.7.2.1 Weakly Typed Parts	79
5.7.2.2 Strongly Typed Parts	82
5.8 Summary	89
6. DETAILED DESIGN and CODING of SOFTWARE PARTS	91
6.1 Merging Detailed Design and Coding	91
6.2 Identification of Additional Parts	92
6.3 Design Reviews	92
6.4 Design for Efficiency	95
6.5 Coding Guidelines	95
6.6 Documentation	97
7. TESTING of REUSABLE SOFTWARE PARTS	101
7.1 Unit and Package Testing	102
7.1.1 Unit Test Approach	103
7.2 Summary	103
8. OTHER TOPICS in PARTS DEVELOPMENT	105
8.1 Management Issues	105
8.1.1 Management Support	105
8.1.2 Process Management	106
8.2 Scope of Reuse	106
8.3 Parts Development Organizations	106
8.4 Configuration Management	107
8.5 Maintenance	108
8.5.1 Modification of Parts	109
8.5.2 Proliferation of Part Variants	110
8.6 Ada Language Considerations	110
8.7 Tools	111

SECTION II: APPLICATION OF REUSABLE SOFTWARE

9. PARTS USE OVERVIEW	115
10. PLANNING for REUSE-BASED DEVELOPMENT	117
10.1 Cost Factors in Reuse-Based Software Development	117
10.1.1 Factors Increasing Cost	117
10.1.2 Factors Decreasing Cost	118
10.1.3 Weighing the Factors	118
10.1.4 Payback from Reusable Software	118
10.2 Management Support	119
10.3 Additional Lifecycle Activities	121
10.3.1 Refinement of the Domain Analysis	122
10.3.2 Project-Specific Commonality	122
10.3.3 Application of Reusable Components	122
10.4 Pitfalls in Software Reuse	124

10.5 The Role of a Reuse Support Organization	124
10.6 Tools for Reuse	125
10.7 Compilers for Reuse	126
10.8 Maintenance and Configuration Management	127
11. REQUIREMENTS ANALYSIS	129
11.1 Parts Identification	129
11.1.1 Tracking Parts Use	130
11.1.2 Tool Support	130
11.2 Compiler Analysis	131
11.3 Documentation	131
11.4 Summary	131
12. PRELIMINARY DESIGN of PARTS-BASED APPLICATIONS	133
12.1 Parts Analysis	134
12.1.1 Closeness of Fit	134
12.1.2 Design Differences	135
12.1.3 Test Support	136
12.1.4 Development Standards	136
12.2 Impact on Design	136
12.2.1 Levels of Abstraction	137
12.2.2 Data Typing	137
12.2.3 Project-Specific Reuse	137
12.3 Reviews	138
12.4 Summary	139
13. DETAILED DESIGN and CODING of REUSE-BASED APPLICATIONS	141
13.1 Impact of Reuse	141
13.2 Tracking Parts Use	142
13.3 Parts Modification	144
13.4 Reviews	145
13.5 Case Study: Incorporating a Reusable Kalman Filter	145
13.5.1 Using the Kalman Filter Parts "As Is"	146
13.5.2 Using Custom Code with the Kalman Filter Parts	146
13.5.3 Modifying the Data Types Package	149
13.6 Summary	151
14. TESTING of REUSE-BASED APPLICATIONS	153
14.1 Test-Related Activities	154
14.2 Summary	155
 SECTION III: MAXIMIZING SOFTWARE REUSE 	
15. MAXIMIZING SOFTWARE REUSE OVERVIEW	159
15.1 Language Features	159
15.2 Standardization	160
15.3 Training	160
15.4 Attitude/Culture	160
15.5 Reuse of Non-Code Software Entities	161
15.6 Tools	161
15.6.1 A Parts Engineering System for Reuse-Based Application Development	162
15.7 Reuse Support Group	163
15.8 Summary	164

16. ADA LANGUAGE CONSIDERATIONS	165
16.1 Efficiency and Effectiveness	165
16.2 Compilers	166
16.3 Chapter 13 Features	168
16.4 Ada 9X	169
17. COMPONENT LIBRARIES	171
17.1 Scope	172
17.2 Library Users	173
17.3 Library Entities	173
17.4 Catalog Technologies	173
17.5 Screening of Parts	174
17.6 Entry of Parts	174
17.7 User Support	175
17.8 Part Attributes	175
17.8.1 Classifying Parts	176
17.9 Populating Libraries	176
17.10 Library Evaluation Criteria	177
17.11 Case Study: The CAMP Catalog	179
18. APPLICATION-BASED SEARCH and RETRIEVAL	183
19. SOFTWARE CONSTRUCTION TOOLS	187
19.1 Case Study: The CAMP Constructors	187
19.1.1 An Example: The CAMP Kalman Filter Constructor	189
19.1.1.1 Kalman Filter Matrices	190
19.1.1.2 Tailoring	190
19.2 Future Directions	191

APPENDICES

I OVERVIEW of the CAMP PROGRAM	195
I.1 Phase 1: Feasibility Study	195
I.2 Phase 2: Technology Demonstration	195
I.3 Phase 3: Technology Transfer and Refinement	196
II SUMMARY of the CAMP 11TH MISSILE APPLICATION	199
II.1 What Is the 11th Missile?	199
II.2 Requirements	200
II.3 Results	201
III PARTS DESIGN ALTERNATIVES	205
III.1 Typeless Method	206
III.2 Overloaded Method	207
III.3 Generic Method	208
III.4 Abstract State Machine Method	209
III.5 Abstract Data Type Method	210
III.6 Skeletal Code Method	212
III.7 Summary	212
References	215

List of Figures

Figure 1-1:	Software Parts Use in Missile Software Systems	4
Figure 3-1:	Model of the DA Process	15
Figure 3-2:	Vertical and Horizontal Domains Overlap	16
Figure 3-3:	Domain Exploration	18
Figure 3-4:	Using a Canonical Domain Structure in a DA	20
Figure 3-5:	CAMP Kalman Filter Parts Domain Utility Matrix	21
Figure 3-6:	Three Levels of Commonality	23
Figure 3-7:	CAMP Parts Taxonomy	25
Figure 3-8:	Types of Parts	27
Figure 3-9:	High-Level View of Missile Kalman Filter Operations	30
Figure 3-10:	CAMP Kalman Filter Common Architecture	30
Figure 3-11:	Software Parts Use in Missile Software Systems	32
Figure 4-1:	Requirements Specification for a Reusable Part	37
Figure 4-2:	A Generic Package Can Be a Part	39
Figure 4-3:	Generic Packages and Subprograms Can Be Parts	39
Figure 4-4:	Non-Generic Units Can Be Parts	40
Figure 5-1:	Reusable Parts Design Methods	43
Figure 5-2:	Comparison of the Six Reusable Parts Design Methods	45
Figure 5-3:	User Substitution of Low-Level Parts	46
Figure 5-4:	Example of Use of Ada Generics in Design of Reusable Parts	46
Figure 5-5:	Commonality Captured in the Generic Part Body	47
Figure 5-6:	Assembling a North-Pointing Navigation System	48
Figure 5-7:	Some Parts Build on Other Parts	49
Figure 5-8:	Parts Work Together	50
Figure 5-9:	Parts Facilitate Use of Other Parts	51
Figure 5-10:	Required Operations Obtained Through Use of Generic Formal Parameters	53
Figure 5-11:	Sample Instantiations of Geometric_Operations Parts Using Default Routines	54
Figure 5-12:	Sample Instantiations of Geometric_Operations Parts Using Specialized Sin_Cos Procedure	55
Figure 5-13:	Distance_to_Current_Waypoint Function (Version 1)	58
Figure 5-14:	Distance_to_Current_Waypoint Function (Version 2)	58
Figure 5-15:	Abstract Data Structures Hierarchy	61
Figure 5-16:	Alternate Abstract Data Structures Hierarchy	61
Figure 5-17:	Updated Generic Package	66
Figure 5-18:	Predefined Instantiations of Generic Package	67
Figure 5-19:	Use of Instantiated Package with Renaming	67
Figure 5-20:	Use of Instantiated Package without Renaming	67
Figure 5-21:	Top-Level Design of Coordinate_Vector_Matrix_Algebra Package (Prerelease Version) ..	73
Figure 5-22:	Top-Level Design of Coordinate_Vector_Matrix_Algebra Package (Release Version 1.0)	74
Figure 5-23-a:	Top-Level Design of Coordinate_Vector_Matrix_Algebra Package (Release Version 1.1) (Part 1 of 2)	76
Figure 5-23-b:	Top-Level Design of Coordinate_Vector_Matrix_Algebra Package (Release Version 1.1) (Part 2 of 2)	77
Figure 5-24:	Modified Top-Level Design of Vector_Operations Package	78
Figure 5-25:	Weakly Data Typed Package (Float)	79
Figure 5-26:	Use of Weakly Typed Non-Generic Package	80
Figure 5-27:	Weakly Data Typed Package (Generic)	81
Figure 5-28:	Strongly Data Typed Package (Inflexible)	82
Figure 5-29:	Use of Multiple Trig Packages	84
Figure 5-30:	Strongly Data Typed Package (Flexible)	86

Figure 5-31: Use of Multiple Flexible Trig Packages	88
Figure 6-1: For High-Level Parts, Detailed Design is Code	93
Figure 6-2: Simple Parts Require Few Comments	93
Figure 6-3: Complicated Parts Require More Comments	94
Figure 6-4: Parts Can Be Designed for Various Degrees of Efficiency	96
Figure 6-5: Complicated Parts with No Comments	98
Figure 7-1: Parts Testing Cycle	102
Figure 7-2: Package Alignment_Measurements	104
Figure 7-3: Procedure Cancel_Measurement	104
Figure 8-1: Alternative Parts Development Organizations	108
Figure 10-1: Software Lifecycle Activities	121
Figure 10-2: Parts Use and Development Cycle	123
Figure 10-3: The Role of a Software Library in the Development Lifecycle	125
Figure 11-1: Reuse and Requirements Analysis	132
Figure 12-1: Use of Parts at Architectural Design Time	137
Figure 12-2: Software Parts Use in Missile Software Systems	138
Figure 12-3: Reuse and Preliminary Design	140
Figure 13-1: Kalman Filter Parts Bundle	147
Figure 13-2: Using the Kalman Filter Parts "As Is"	148
Figure 13-3: Kalman Filter Parts Instantiate Other Parts	149
Figure 13-4: Replacing Parts with Custom Code	150
Figure 13-5: Reuse and Detailed Design/Coding	152
Figure 14-1: Test Approach	154
Figure 15-1: Summary of Tools for Software Reuse	162
Figure 15-2: The CAMP Parts Composition System	163
Figure 16-1: FORTRAN in Ada: Ada Language Features Do Not Force Good Design	166
Figure 16-2: Utilization of Ada Language Features	167
Figure 17-1: CAMP Catalog Parts Attributes	180
Figure 17-2: CAMP Parts Taxonomy	181
Figure 18-1: CAMP Parts Exploration Approaches	184
Figure 18-2: CAMP Missile Software Model	185
Figure 18-3: CAMP Missile Model Walkthrough Example	185
Figure 19-1: CAMP Schematic Reuse	188
Figure 19-2: High-Level View of Missile Kalman Filter	189
Figure 19-3: CAMP Kalman Filter Parts Hierarchy	189
Figure I-1: CAMP Products	197
Figure II-1: 11th Missile Hardware Design	200
Figure II-2: Overloaded Operator Caused Problems for Compiler	202
Figure II-3: Compilers Had Problems Finding Default Subprograms	203
Figure III-1: Reusable Parts Design Methods	205
Figure III-2: Strong Data Typing Example	206
Figure III-3: Typeless Method Example	207
Figure III-4: Overloaded Method Example	207
Figure III-5: Generic Method Example	208
Figure III-6: Tunneling of Parameters	209
Figure III-7: Abstract State Machine Method Example	210
Figure III-8: Abstract Data Type Method Example	211
Figure III-9: Skeletal Code Template Method Example	212

List of Tables

Table 1-1:	Barriers to Software Reuse	5
Table 3-1:	The CAMP Domain Representation Set	17
Table 3-2:	CAMP Functional Areas	17
Table 3-3:	CAMP Parts Taxonomy	26
Table 3-4:	Initially Identified Kalman Filter Parts	31
Table 3-5:	Steps in a Domain Analysis	33
Table 3-6:	Major Products of a Domain Analysis	34
Table 5-1:	Data Typing Comparisons	57
Table 5-2:	Various Degrees of Data Typing	59
Table 5-3:	Bundling by Type versus Class	62
Table 5-4:	Operations Implemented Via Multiple Packages	65
Table 5-5:	General versus Coordinate Matrix Operation	71
Table 5-6:	Interface to Weakly Typed Non-Generic Package	80
Table 5-7:	Use of Weakly Typed Generic Package in Weakly Typed Application	81
Table 5-8:	Use of Weakly Typed Generic Package in Strongly Typed Application	81
Table 5-9:	Interface to Weakly Typed Generic Package	82
Table 5-10:	Strongly Typed Part in Weakly Typed Application	83
Table 5-11:	Interface to Multiple, Inflexible Trig Packages	85
Table 5-12:	Use of Flexible Package in Multiple Applications	87
Table 6-1:	CAMP Code Header Information (DOD-STD-2167)	100
Table 7-1:	Unit- and Package-Level Test Decision Matrix	103
Table 8-1:	Items Under Configuration Control	108
Table 12-1:	Software Design Review Questions	139
Table 13-1:	Summary of CAMP Parts Usage in the 11th Missile Application	143
Table 13-2:	Summary of CAMP Parts Used in the 11th Missile Application	143
Table 13-3:	CAMP 11th Missile Parts Usage	144
Table 16-1:	Use of Chapter 13 Ada Features by the CAMP 11th Missile	169
Table II-1:	Processors and Their Functions	200
Table II-2:	11th Missile Effort	204
Table II-3:	11th Missile Size - Parts Method	204
Table II-4:	11th Missile Productivity - Parts Method	204
Table II-5:	Effect of Parts on 11th Missile Effort	204

List of Acronyms

ACM	Association of Computing Machinery
ACVC	Ada Compiler Validation Capability
ADL	Ada Design Language
AFATL	Air Force Armament Laboratory
AFB	Air Force Base
AIAA	American Institute of Aeronautics and Astronautics
AMPEE	Ada Missile Parts Engineering Expert (system)
ANSI	American National Standards Institute
ART TM	Automated Reasoning Tool (Inference, Corp.)
BDT	Basic_Data_Types (CAMP)
CAMP	Common Ada Missile Packages
CASE	Computer-Aided Software Engineering
CCB	Configuration Control Board
CDRL	Contractual Data Requirements List
CPDS	Computer Program Development Specification
CPPS	Computer Program Product Specification
CPU	Central Processing Unit
CSC	Computer Software Component
CSCI	Computer Software Configuration Item
CSU	Computer Software Unit
CVMA	Coordinate_Vector_Matrix_Algebra (CAMP)
DA	Domain Analysis
DoD	Department of Defense
DOD-STD	Department of Defense Standard
DSMAC	Digital Scene Matching Area Correction
DTIC	Defense Technical Information Center
ERA	Entity-Relationship-Attribute
FIFO	First-In First-Out
FORTTRAN	FORmula TRANslation
FQT	Formal Qualification Testing
FSED	Full-Scale Engineering Development
FSM	Finite State Machine
GPMath	General_Purpose_Math (CAMP)
GPS	Global Positioning System

GRACE™	Generic Reusable Ada Components for Engineering
HIL	Hardware-in-the-Loop
IDD	Interface Design Document
IRS	Interface Requirements Specification
ISA	Inertial Sensor Assembly
ISO	International Standards Organization
JOVIAL	Jules Own Version of International Algebraic Language
KFDT	Kalman_Filter_Data_Types (CAMP)
LCO	Lifecycle Object
LISP	List Processing (language)
LOC	Lines of Code
LRM	Language Reference Manual
MDMSC	McDonnell Douglas Missile Systems Company
MIL-STD	Military Standard
MRASM	Medium Range Air-to-Surface Missile
NDS	Non-Developmental Software
NIH	Not Invented Here
PCS	Parts Composition System
PDL	Program Design Language
RAPID	Reusable Ada Packages for Information Systems Development
RC	Reusable Component
ROM	Read Only Memory
RTE	Real-Time Embedded
SCP	Software Change Proposal
SDD	Software Design Document
SDF	Software Development File
SDP	Software Development Plan
SDR	Software Discrepancy Report
SIGAda	(ACM) Special Interest Group for Ada
SRS	Software Requirements Specification
STARS	Software Technology for Adaptable, Reliable Systems
STD	Software Test Description
STLDD	Software Top-Level Design Document
STP	Software Test Plan
TERCOM	Terrain Correlation Mapping

TLCSC

Top-Level CSC

UDF

Unit Development File

INTRODUCTION

CHAPTER 1 INTRODUCTION

Software has increasingly become a limiting factor in the deployment of new systems. While hardware technology has advanced rapidly and experienced declining costs, software costs, as a percentage of project development, have increased. Greater demands are being made of the software that is embedded in advanced systems. For example, the F-4 fighters flown during the Vietnam era contained no software, the F-16 is estimated to have 236,000 lines of code, and the B-1 Bomber is estimated to have 1.2 million lines of code (Reference [2]). Not only is the sheer quantity of software increasing, but it is playing an increasingly critical role. These increasing demands have led to schedule delays, cost overruns, product quality and reliability problems, escalating maintenance costs, and a shortage of adequately trained software engineers. This situation is commonly referred to as the "software crisis." This "crisis," first identified in the mid-70s, is still with us today.

There is no panacea for a problem of this magnitude: dramatic changes are needed in software development processes, but these changes will also raise new issues that must be addressed before the solution can effectively be applied. Programming language standardization and software reuse are two means of effecting at least some of the gains that are needed. Ada has become the standard language for mission critical systems (Reference [14]), but software reuse is still not a widespread reality. Many of the issues related to standardization on Ada have been addressed, but this is not the case with software reuse: there are many new issues facing an organization that wants to develop and use reusable software components. Many of these will be addressed in the remainder of this manual.

1.1 Motivation for Software Reuse

The potential benefits of a parts-based approach to software engineering are quite significant, but they are predicated on the assumptions that a given application area exhibits significant and continuing levels of commonality that can be exploited through the development and application of reusable software components, and that methods (and tools) are available for facilitating/enforcing a parts-based approach to software development. In order to maximize potential benefits, reuse needs to be planned rather than allowed to occur in an ad hoc manner; there must be a systematic approach to parts development and use. Ad hoc reuse (e.g., people working in close proximity sharing code) has been practiced for many years, but in order to realize the productivity gains that are needed to overcome the software crisis, this process must be formalized at more than the code-sharing level.

Software reuse can increase software development and maintenance productivity; lead to higher quality, more reliable software; and conserve and preserve software engineering expertise. Although increased software development productivity is often cited as a key reason to practice software reuse, some of the real gains may come from increased reliability and lower maintenance and enhancement costs. Maintenance costs may be cut dramatically because parts will have been thoroughly and reliably tested prior to release. Rigorous testing of software parts is critical because the impact of errors in software parts can be much greater, due to their wide dissemination, than it is for custom software. Parts sets with varying degrees of reliability will not be trusted and will, most certainly, not be used. Reusable software also provides an organization with a way to capture and conserve its critical software engineering expertise.

There are many views on the level at which reuse should take place: there is reuse of code, design, requirements, analyses, etc. Research has been done in all of these areas. The area with the highest potential for near-term payoff is that of code reuse, including deliverable code, test code, simulation code, etc. Thus, most of the issues discussed in this manual pertain to reuse of software code components, or parts. Reusable code components can exist at many levels of abstraction; they can range from low-level mathematical operators to complex architectural structures.

Reusable software parts can be defined as pre-built software components which have been explicitly designed to be used in multiple applications, generally within a given application area (see Figure 1-1). It is important to note that there are other definitions of reusable software — some consider any software lifecycle object (LCO) that is used, or capable of being used, more than once, regardless of whether it was explicitly designed for reuse or not, to be reusable software. Reuse of non-code objects from the software development process is sometimes referred to as *software engineering reuse* (Reference [19]).

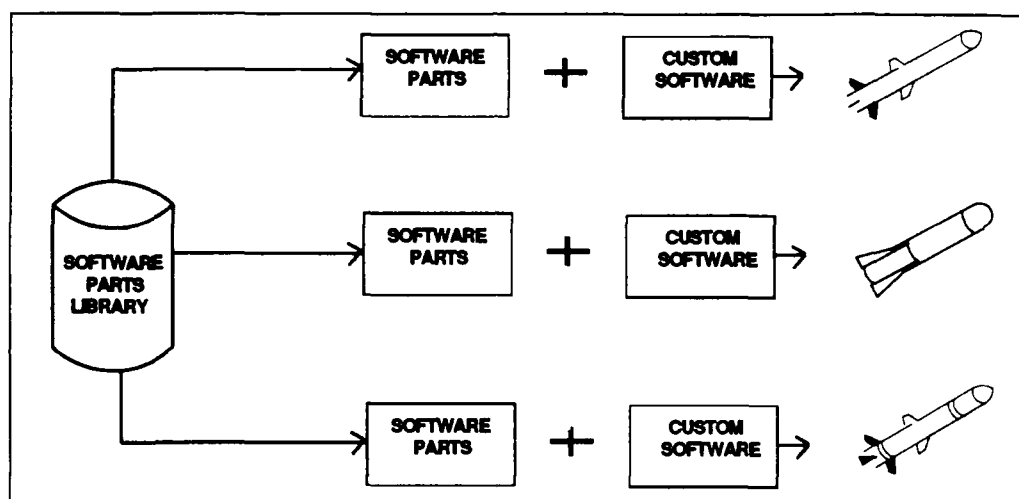


Figure 1-1: Software Parts Use in Missile Software Systems

Reuse of code is a good starting point for a software reuse program, but there are limits to the gains from code reuse alone. Typically, the coding and corresponding unit testing activity consumes only 20% of the resources of a software development effort, thus, even if 100% code reuse were achieved, the productivity improvement would never actually reach 20% (because of the effort involved in actually reusing software). Although most reuse efforts stress reuse of associated lifecycle objects, such as requirements specifications and design documents, reuse of more abstract designs and requirements remains, for the most part, a research issue.

Despite the widespread interest and research in software reuse, there is still not a major software components industry, and much of the software in new systems is still custom-crafted. The U.S. DoD software engineering community is not alone in its pursuit of software reuse; there are many commercial efforts, as well as European and Japanese efforts underway.

1.2 Barriers to Software Reuse

Software reuse will not just happen — there are significant psychological, sociological, managerial, and technical hurdles that must be overcome. Effective reuse requires planning! Some of the major factors that have impeded meaningful levels of software reuse from being achieved are summarized in Table 1-1.

Table 1-1: Barriers to Software Reuse

- Programming Language Features and Capabilities
- Domain Analysis
- *Not Invented Here* (NIH) Syndrome
- Lack of Training in Software Reuse
- Few Support Tools
- Lack of Knowledge about Available Parts
- Locating Parts
- Integration of Reusable and Custom Code
- Documentation
- Proprietary Rights Issues
- Contractual/Product Liability Issues
- Organizational Profit Maximization In-the-Small
- No Requirement to Reuse
- Lack of Convincing Data on Benefits

Although software reuse is not programming language dependent, certain language features can facilitate or hinder it. A language, such as Ada, which incorporates features that facilitate reuse and modern software engineering practices, brings meaningful levels of software reuse within reach. Ada possesses facilities for enforcing the design and construction of autonomous software units with clean, well-defined interfaces, and developing software parts that are generic in nature and that can be easily tailored for a particular application using features present in the language itself. Reuse has occurred in other languages but not at the level that is possible with Ada (e.g., FORTRAN math/statistics routines have been very successful, but reuse in general has not taken place in FORTRAN applications). Programming language features can facilitate software reuse, but they do not guarantee it (e.g., an Ada generic package is not necessarily a reusable package). Thus, reuse is not restricted to applications developed in any given language.

The need to conduct an in-depth domain analysis of the application area for which parts are to be developed is a major barrier to implementing an effective software reuse program. Domain analysis, which is discussed in detail in Chapter 3, 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 require intensive examination of existing software systems within the application area, and personnel skilled both in modern software development techniques and in the application area.

The psychological barriers to software reuse should not be underestimated. Software engineers frequently bypass reusable software in favor of creating their own solution both because it is more "fun" to develop software from scratch, and because they lack knowledge of and confidence in software that is available for reuse. Additionally, some software engineers feel that reuse will lessen their creativity. It can, in reality, free them from coding details and allow them to spend more time in the creative design process. Software engineers also generally lack training in software reuse; they are taught to reinvent it rather than reuse it. As Jean Sammet pointed out, *"Ever since the second square root routine was written, the programming field has lost adequate control of reusability"* (Reference [36]).

The need to integrate reusable software components into new applications poses another barrier to effective software reuse. Much of the software that is currently available was not developed specifically for reuse and can be particularly difficult to understand and fit into a new application. Without adequate information about the components, it may indeed be more cost-effective to redevelop software than to determine what is required to fit it into a new application. As reuse becomes more widespread, developers will face the problem not only of

integration of reusable parts with custom code, but also the integration of parts sets from different developers. These parts sets may have been developed under very different sets of development standards, and hence have structures that are difficult to integrate. It is important to know the structure and standards of the parts being used.

Reusers also face more mundane problems, such as documentation of software that includes reusable parts and how to test that software. For instance, does the reusable software need to be completely retested at the unit level or will integration testing suffice?

Protection of proprietary rights poses another obstacle to industrywide adoption of parts sets, but should not pose a problem for company or program implementation of a software reuse approach. Incorporation of reusable software in a deliverable product also raises contractual and product liability issues that have yet to be universally resolved. There are often contractual restrictions on the use of non-developmental software or on the reuse of software developed under contract.

Organizations are often structured as a collection of many smaller components, each with its own set of goals. This structure makes it difficult to promote the sharing of resources, including software, and makes it difficult to put a software reuse program in place. To be effective, everyone affected must be convinced of the benefits of software reuse, including company management, software development staff, and project teams. They must be convinced that software reuse is in their best interests (Reference [7]). Reuse at the corporate level takes not only a significant effort investment, but also a leap of faith on the part of the managers. The payback from starting a reuse program is not immediate. This is an investment in the future, but once the benefits of software reuse become apparent and the technology matures, software reuse should be self-motivating, i.e., the government should not need to provide contractual incentives, but it will need to remove contractual barriers or disincentives. Reuse can be practiced at virtually any organizational level, thus a reuse program could be put in place at a project or program level first, and then expanded to include more of the organization.

Tool support for software reuse can facilitate the introduction and acceptance of parts-based software development. Tools can free the engineer from mundane, mechanical chores and facilitate identification, retrieval, evaluation, and incorporation of reusable software components in new applications. They can enhance productivity gains by decreasing the cost of reuse. Tool support can range from a simple keyword retrieval mechanism to a complete computer-aided software engineering (CASE) environment that supports all aspects of software engineering with parts. The Japanese have shown that even simple keyword retrieval can be used successfully in identification and retrieval of available components. As collections grow, it may become increasingly difficult to identify the best match for a particular operation, thus, identification and retrieval techniques are a major area of research within the reuse community.

Given all of these barriers to software reuse, it is not surprising that software reuse is not widespread. Ad hoc reuse has been practiced for some time, but widespread, systematic reuse will require that adequate incentives be provided to software engineers and their managers. These incentives can take the form of a combination of rewards and mandates. Regardless of the form, a high-degree of discipline is required for an organization to successfully establish and maintain a parts-based approach to software development. Software reviews and audits are one way to enforce a software reuse policy.

There have been relatively few concrete demonstrations of successful software reuse within real-time embedded applications, and because of this, it is difficult to demonstrate the savings that can accrue from reuse. Much of

the savings may come during the maintenance phase, and is not readily apparent to the manager starting a new development effort. For the most part, hard data to support the belief that significant cost savings will accrue in later stages of the lifecycle is still lacking.

1.3 Reuse in the RTE Community

Despite the apparent advantages of software reuse, the real-time embedded (RTE) software engineering community has been particularly skeptical of the practicality of software reuse within its domains. The RTE community has generally thought that software parts (i.e., components specifically written to be reused) are not practical in real-time embedded applications because reusable software parts must be general enough to allow broad use and generality has historically implied an intolerable loss of efficiency. Within non-RTE applications, it is generally possible to trade run-time efficiency for significant increases in software quality and productivity, but within RTE applications, this luxury can generally not be afforded.

RTE software is typically targeted at micro-processors embedded in products such as aircraft, missiles, and satellites. Additional memory or processor upgrades are not a simple matter since these embedded processors must comply with severe limitations on weight, power, and volume, and must also conform to military specifications. The demand for greater functionality has more than accounted for the added capabilities provided by advances in memory and processor technologies.

1.4 Summary

This manual is motivated by the fact that RTE community needs are different, and that development of reusable software is different from development of one-shot or custom software. Reusable software has additional documentation needs, testing may have to be more stringent, and there are procedural factors that must be considered. These additional factors result in a higher cost to develop reusable code than to develop one-shot code, but with reuse, that cost can be amortized across several projects. To be effective, reuse must be planned and considered throughout the software lifecycle, and processes and tools that support a lifecycle approach to software reuse are needed.

1.5 About the Manual

This document draws heavily on the Ada parts development and use experience gained on the Common Ada Missile Packages (CAMP) project performed for the Air Force Armament Laboratory at Eglin AFB. This program is described in Appendices I and II. Although the manual uses DOD-STD-2167A (Defense System Software Development, Reference [16]) terminology, the content is applicable to non-DoD projects, as well.

The remainder of this manual is organized into three sections plus appendices.

- **Foundations of Software Reuse:** This section deals with the development of reusable software and the special considerations of this type of software.
- **Application of Reusable Software:** This section deals with the development of applications that incorporate reusable software components. It covers how to identify parts that can be used, how to locate those parts, how to incorporate the parts, the special documentation needs, etc.
- **Maximizing Software Reuse:** This section covers special considerations in developing and reusing Ada software parts, such as compiler maturity, software catalogs, etc.
- **Appendices:** There are three appendices which provide an overview of the CAMP program; an overview of the CAMP parts use testbed effort, known as the 11th Missile Application; and a description of the parts design method alternatives that were investigated prior to CAMP parts development.

SECTION I

FOUNDATIONS OF SOFTWARE REUSE: DEVELOPMENT OF REUSABLE COMPONENTS

CHAPTER 2

PARTS DEVELOPMENT OVERVIEW

Many objects are generated during the software development lifecycle — requirements, design, code, documentation, test products (i.e., plans, procedures, and code). Each of these objects can be reused, but the reuse process may differ depending on the representation of each of these objects and on the emphasis of the reuse effort (e.g., some reuse programs emphasize the reuse of code, while others place the primary emphasis on reuse of requirements and/or design, with reuse of code resulting from reuse of these other software lifecycle products). Reuse of non-code software entities is an issue of finding or developing an adequate representation scheme for those entities that will both permit and facilitate their reuse. The representation problem is one of the primary reasons for the current emphasis on reuse of code-level components and their associated documentation — the implementation language provides a rigorous and well-understood representation of the reusable object. Although there are definite benefits to code-level reuse, coding generally comprises only 20% of the project resources, thus, reuse that concentrates only on code reuse, results in less productivity gain than reuse throughout the lifecycle.

In this section we will discuss the foundations of a software reuse program that is concerned primarily with code-level reuse, but recognizes the need to reuse related software lifecycle objects (LCOs). The discussion will range from domain analysis and component identification to the development of effective reusable software components for RTE applications concerned with *efficiency, accuracy, maintainability, and understandability*. We will also discuss some of the characteristics of good components, such as generality, tailorability, and ease of use; the importance of quality; the level of documentation that is needed for reusable components and how this differs from custom components; the need for strict adherence to programming standards and good software engineering practices; and the importance of reviews involving domain experts. Although the process of developing good reusable software is similar to that of developing good software in general, there are some differences, e.g., the type of documentation; the level of testing; the need to anticipate future uses; and the need to include domain experts and/or typical users in reviews throughout the lifecycle. Sound software engineering is a necessary but not sufficient condition for producing reusable software.

When beginning a software reuse program, a systematic approach to identifying and developing reusable software components is needed. This usually takes the form of a domain analysis (DA). During a domain analysis, a domain is studied in depth, including an examination of a representative set of applications (if they exist). The domain analysis results in development of a domain model that provides the framework for development of reusable software components (i.e., it leads to the identification of common objects, operations, and structures). Domain analysis is a challenging, time-consuming task.

Reusable components can also be identified by projects as the need or opportunity arises. This method of identification is generally most useful once a core set of components has been identified for a particular domain. It is a good way to enrich an existing parts set, but it is generally inadequate as a means of establishing an initial parts base. Because of their involvement in the domain, project personnel are often able to identify parts that would not be identified by an independent software reuse group.

A software reuse group can "harvest" or "scavenge" parts from projects, i.e., they can attend project reviews with the specific intent of identifying areas (and software) that show potential for reuse. The software obtained in this manner will generally not be directly reusable, i.e., a parts group will have to modify or restructure the

components for reuse, perform additional testing, and produce the documentation that is needed for reusable components. These methods for component identification are discussed further in Chapter 8.

Reusable code components should be designed with the following goals in mind:

1. A part should provide a useful object or function to more than one potential application. Parts should not be developed that have no future or are not really useful (e.g., parts that do not provide a significant benefit over developing the software from scratch, such as a part that adds any two numbers).
2. Use of parts should result in little, if any, loss of run-time efficiency.
3. Parts should be easy to use and flexible.
4. If it is possible that operations and/or objects that a part provides can be arranged differently (i.e., in some way that is not currently envisioned), then the subparts should be directly usable by the end-user.

Adherence to standards is particularly important in the development of reusable software. Software developed in compliance with a reasonable set of standards is easier to understand, maintain, and integrate. Standards should be applied to all areas of software development — coding, documentation, naming, testing, etc.

This section is organized by functional area within the software development lifecycle, rather than by lifecycle phase (e.g., although test plan development generally takes place during architectural design, and test procedure development takes place during detailed design, these activities are discussed under *Testing*). Domain analysis and requirements definition are presented in separate sections, but they are not performed in a strictly sequential manner. There is significant overlap in activities, and activities (or groups of activities) can be performed recursively. As previously mentioned, the emphasis here is on the reuse of Ada components and their associated lifecycle objects.

The remainder of this section is organized as follows:

- **Domain Analysis:** This chapter covers the principles and performance of a domain analysis. It explains what a domain analysis is, provides some guidance in how it should be performed, and identifies a number of issues that need to be addressed.
- **Requirements Definition:** This chapter discusses requirements engineering for reusable software, and provides a definition of a reusable Ada component.
- **Architectural Design:** This chapter covers alternative design approaches and describes in detail the approach used in the development of the CAMP Ada parts.
- **Detailed Design and Coding:** This chapter covers the aspects of detailed design and coding that are unique to parts development.
- **Testing:** This chapter covers the special testing needs of reusable software.
- **Other Topics:** This chapter includes a discussion of Ada language issues, management issues, configuration management, tools, standards, and maintenance issues that are specific to parts development.

CHAPTER 3 DOMAIN ANALYSIS

The first step in developing reusable components within a domain is to conduct a *domain commonality analysis*, or simply, a *domain analysis (DA)*. In this chapter, the activities involved in performing such an analysis are discussed: domain exploration, commonality specification, component development verification, DA management, etc.

A *domain analysis* is an investigation of a specific domain or application area which seeks to identify a common "generic" paradigm for the domain. This is the primary means to identify candidate reusable software components when initiating a software reuse effort. A domain analysis is similar to a systems analysis, but is much broader in scope. It is not limited to a single system — it involves the examination of all, or a representative set, of systems of a certain type (e.g., the CAMP domain analysis examined the operational flight software from a set of 10 missiles), and may also include an examination of the underlying domain theory and future direction of the domain. It results in the development of a domain model that represents a common view of the domain. This model provides the framework for the identification of objects, operations, and structures that can be captured as reusable software components.

A domain analysis is perhaps the single most important step in starting a software reuse program. Although it is not the only way to identify reusable components, it is the best way to begin. Domain analysis has the advantage that it is a systematic approach to uncovering commonality. Once a software reuse program is in place, additional reusable components may be identified through other means which are discussed in Chapter 8.

Performing a domain analysis is one of the most difficult tasks in a software reuse program, particularly for real-time embedded applications. This is because of the complexity of the applications, the efficiency requirements, and the tedium of poring over documentation and code produced on other projects. Domain analyses are also relatively expensive to perform, requiring intensive examination of existing software systems within the domain, study of the underlying domain theory, and the expertise of both well-trained software engineers and domain experts.

It is important to point out that there is not, as yet, a standard procedure for performing a DA. Despite this, the concept of domain analysis is not new. Domain analyses can be performed for reasons other than the development of reusable software components. For example, a domain analysis might be performed in order to develop a model of an application area for instructional purposes, in order to facilitate database design, for documentation purposes, or to facilitate software maintenance or verification. Domain analysis and domain modeling are both areas of current research interest. Neighbors performed early research into domain analysis in support of reuse of analysis and design (see Reference [32]). As interest has grown in the area, there have been an increasing number of studies undertaken to develop a standard process for performing domain analyses.

The existence of domain commonality that is appropriate for capture as reusable software components cannot be taken for granted. Although significant levels of commonality have been found in some application areas, there is no guarantee that high levels will be found in all domains. For example, a high degree of commonality has been found within various business-oriented systems (e.g., Raytheon Missile Systems Company, Reference [24]), but this cannot be taken as an indication that this same level of commonality will be found within any or

all RTE application domains. Real-time embedded applications have special needs which must be taken into consideration when developing reusable software — typically, there are hard real-time constraints, limited computer memory and data storage available, and extremely high reliability requirements.

Domain selection may arise as an issue if parts are developed by a group that has no ties to a specific domain (i.e., they are an independent parts development group rather than a project-directed group), or that has ties to several domains. This has been addressed during Neighbor's work on Draco (see References [19, 32]). To be a viable candidate for domain analysis and parts development, the domain must have both a future and a past. It would not be a cost-effective use of resources to perform a domain analysis in an obsolete area, but there should be past applications in the same or related areas to draw on in order to perform the domain analysis. Mature, stable domains should prove to be the most cost-effective in which to work. The set of applications available for examination (i.e., the domain representation set) must be sufficiently broad to adequately cover the domain under investigation.

The model of the DA process presented here is based on experience on the CAMP program and a survey of other industrial and academic software reuse efforts (References [32], [34], [22]). Figure 3-1 depicts a model of the DA process. It consists of six major activities; each is discussed in subsequent paragraphs.

- Domain Definition: The process of defining or bounding the subject domain
- Domain Representation: The process of selecting an adequate representation of the domain; this includes domain theory, as well as a set of existing applications
- Domain Exploration: The process of studying the domain, including the theory, current state, future state, and representative applications, in order to develop a domain model and identify common operations, objects, and structures
- Domain Modeling: The process of developing a domain model that can be used as a framework in the identification of reusable software components
- Component Specification: The process of specifying the common objects, operations, structures, etc. for encapsulation as reusable software components
- Component Development Verification: The process of verifying that candidate reusable software components should be built

3.1 Domain Definition

Domain definition is the process of clearly bounding the subject domain. It is an iterative process that begins with a definition of the domain that delineates the scope of the domain analysis, and proceeds to the development of a domain model that serves as the framework for developing reusable components.

The domain must be defined in sufficient detail and with sufficient clarity so that both the customer and the domain analysts have a common understanding of its scope. As the domain analysis proceeds and the analyst gains greater understanding of the domain, the domain definition, or boundaries will undergo successive refinement. The definition may start out as a natural language statement which seeks to define the domain (e.g., "The domain is missile operational flight software."), and proceed to a highly structured model by the time the analysis is completed. The domain model is an important product of the domain analysis process because it provides a framework for the identification and development of reusable components.

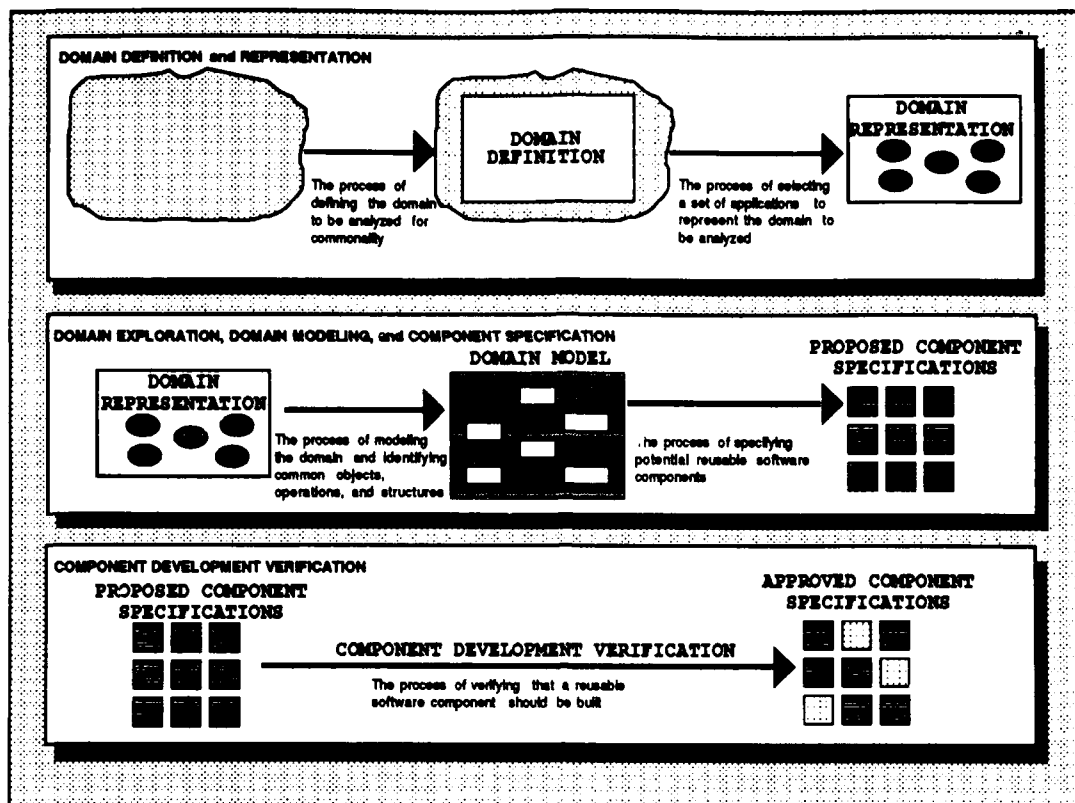


Figure 3-1: Model of the DA Process

There are several complicating factors in the domain definition process:

- Fuzzy domain boundaries
- Overlapping domains (vertical domains)
- Intersecting domains (horizontal domains)

Domains do not always have clear-cut boundaries, thus the bounding process is, at times, subjective. Despite this, the analyst should attempt to bound the domain as precisely as possible (within reason and given normal project constraints on time and budget) in order to focus the analysis effort and prevent wasted effort during later DA activities.

The fact that domains overlap is not in and of itself a problem. A problem arises if this overlap causes the domain analysts to be sidetracked. A *vertical domain* is an application-oriented grouping, whereas a *horizontal domain* is an application-independent grouping. Examples of vertical domains include armament electronics software, avionics software, and C³I software. Armonics software is a vertical domain which overlaps a number of horizontal domains, including signal processing, matrix algebra, and abstract data structures. Figure 3-2 depicts overlapping vertical and horizontal domains.

It is easy to be sidetracked into identifying commonality, and hence candidate reusable software components, not because they are needed within the domain under examination, but because they are within the horizontal domain. Periodic reviews can help ensure that the domain analyst is addressing the correct domain.

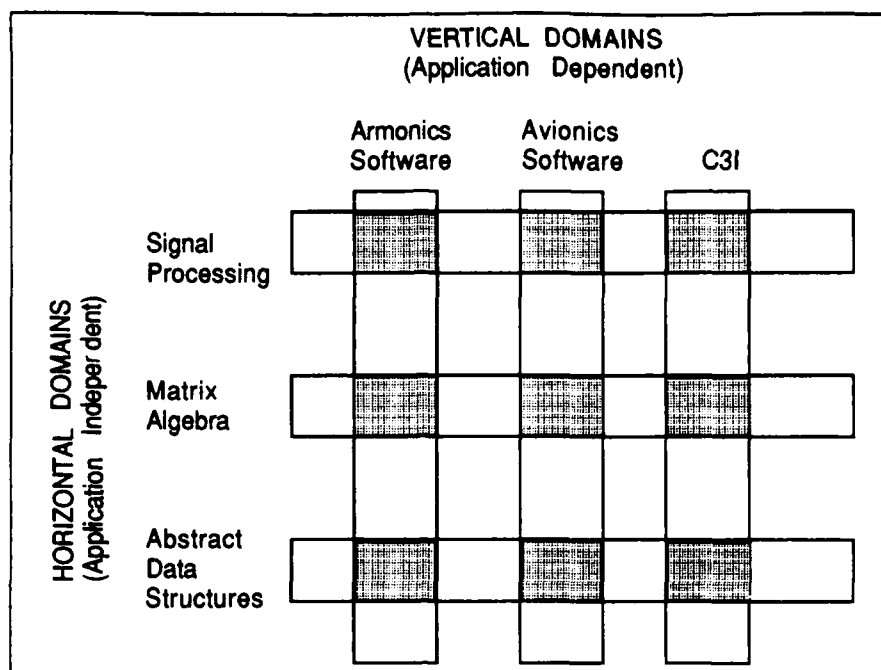


Figure 3-2: Vertical and Horizontal Domains Overlap

3.2 Domain Representation

A domain is represented not only by the applications that have been developed in that area, but also by the underlying theory of the domain and by the collective experience of domain experts. These are all things that have to be taken into consideration during domain exploration.

The *domain representation set* is a set of applications which serves to characterize the domain under investigation. Although practical constraints prevent the examination of every possible application in a domain, the goal is to select a set of applications that meets at least the following criteria.

- Sufficient data (e.g., specifications) must exist and be readily available for each selected application. For example, an application for which there is only a code listing would not be an ideal candidate. Ideally, the analyst should have access to requirements and design documents, as well as to code. Access to the developers of the applications is a definite advantage. In an ideal world, sufficient information about an application would be available from its documentation, but, this is often not the case. Quality of the documents is an important consideration when selecting representative applications, and can sometimes be traded-off with access to the original developers.
- All areas within the domain of interest must be covered by one or more applications. For example, if the domain of interest is missile flight software, then data should be included on anti-ship missiles, air-air missiles, etc. The CAMP domain analysis examined 10 missile software systems (see Table 3-1), and was required to include at least 2 of the following types: air-to-air, air-to-surface, surface-to-air, and surface-to-surface. Additionally, the set had to include the functional areas enumerated in Table 3-2. A concise domain definition will aid in the selection of a representation set.
- Selected applications must not be based on antiquated or obsolete technology. Since the purpose is to extrapolate commonality into the future, the use of applications which are out-dated will result in parts that have no users.

- The domain representation set must be large enough to establish commonality, but must not be so large that the analysts are overwhelmed with data. In general, there should be a minimum of 3 applications in the set; it is unlikely that an adequate domain analysis could be performed on fewer than three applications (Reference [3]).

When a domain analysis is performed in a new application area, there may not be existing applications to examine. In this case, the domain will be represented by the domain theory and closely related applications. Although an application may appear to be radically different from existing applications, further analysis will undoubtedly reveal sub-areas that are closely allied with existing applications. It is rarely the case that a new application is entirely different from everything that has preceded it; progress is generally evolutionary.

Table 3-1: The CAMP Domain Representation Set

- Flight Software for the Medium Range Air-to-Surface Missile (AGM-109H)
- Flight Software for the Medium Range Air-to-Surface Missile (AGM-109L)
- Strapdown Inertial Navigation Program for the Unaided Tactical Guidance Project
- Guidance and Navigation Program for the Midcourse Guidance Demonstration
- Flight Software for the Tomahawk Land Attack Missile (BGM-109A)
- Flight Software for the Tomahawk Anti-Ship Missile (BGM-109B)
- Flight Software for the Tomahawk Land Attack Missile (BGM-109C)
- Flight Software for the Tomahawk Land Attack Missile (BGM-109G)
- Flight Software for the Harpoon Missile (Block 1C)
- Safeguard Spartan Missile

Table 3-2: CAMP Functional Areas

AREA	SUB-AREA	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 Engine Management Antenna Control	Digital Filters Pulse Motor Logic
Signal Processing Fuzing	Spectral Analysis Optical Active Semi-Active	Fast Fourier Transforms Optimized Time Delay
Weapon/Aircraft Avionics Interface	Weapon Initialization	Inertial Systems Transfer Alignment

3.3 Domain Exploration

Domain exploration is the portion of the domain analysis concerned with identifying commonality within the domain. It involves the study of domain theory; past, current, and future states of the domain; and representative domain applications in order to identify commonality that can be captured as reusable operations, objects, and structures. This analysis is similar to the knowledge engineering process performed during the construction of an expert system — both are in-depth studies of a given domain that attempt to formalize a body of knowledge. In domain analysis, this knowledge is needed to distill abstract requirements from existing knowledge and concrete instances of requirements. Domain exploration leads to the refinement of the domain

definition, and aids in the development of a domain model. Once again, the domain model provides a framework for the identification and development of reusable software components.

3.3.1 Analysis of Existing Applications

Much of the raw data for the domain exploration comes from software documents, including requirements specifications, design documents, and code listings (see Figure 3-3). Although a great deal of information can be gleaned from the requirements documents alone, the design documents often provide information that is useful in understanding the requirements. The design documents can be useful in identifying internal functionality that cannot be directly mapped to the external requirements of the system. Although code listings are not generally useful by themselves, they can aid the analyst in understanding the requirements and design specification.

It is likely that more parts will be identified once implementation of the originally identified parts set is begun. There are two main reasons for this: (1) supporting parts will undoubtedly be identified as the previously identified parts are implemented, and (2) domain analysis is iterative. For example, during the CAMP domain analysis, approximately 250 parts were identified, but the final count when implementation was completed was about 450. Additional parts were needed to support the ones previously identified, and in some cases variants of previously identified parts were developed.

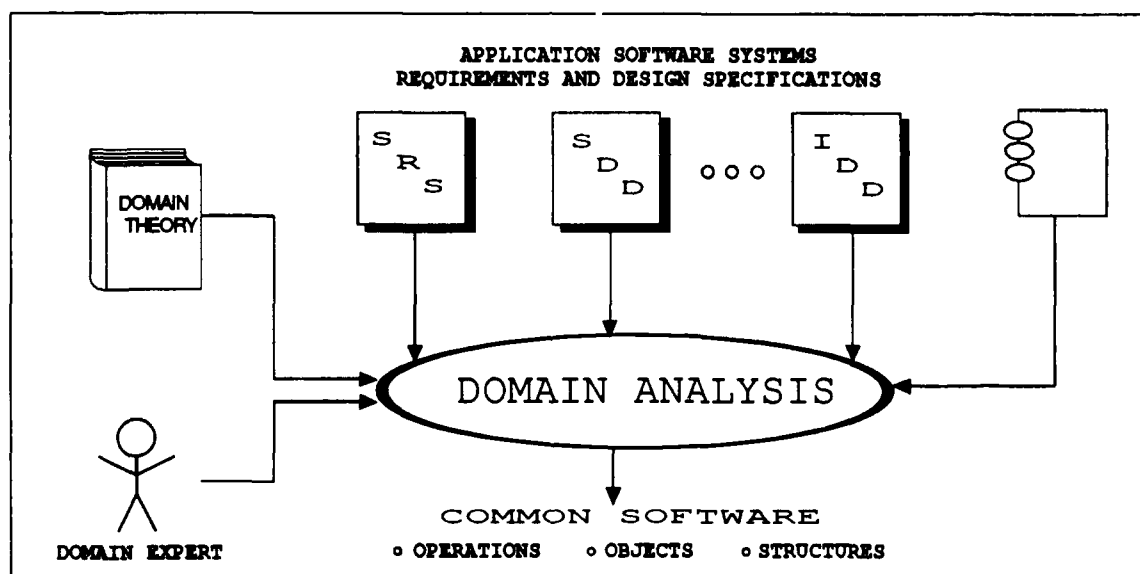


Figure 3-3: Domain Exploration

Analysis of existing applications can be a tedious process. Some of the difficulties are discussed below.

- **Language differences:** The applications that comprise a representation set may be in different programming languages, and not all of the languages may be familiar to the domain analyst.
- **Style:** Even when programming style standards are enforced within a project, the domain analyst will have to deal with different styles across applications. Style can be a major impediment in analysis of code.
- **Conventions:** Within each field there are different notational (as well as other) conventions with which the domain analyst must be familiar in order to fully understand the code and documentation.

- **Arbitrary differences:** Arbitrary differences can lead an analyst to incorrectly conclude that commonality does not exist when, in fact, it may. In some cases, two systems will perform an operation differently for no apparent reason, i.e., they could have used the same operation with no change in operational capabilities. This is very difficult to determine because lack of domain understanding can often make arbitrary differences seem like substantive differences, and make substantive differences seem arbitrary. Notational differences can lead an analyst to conclude that two operations are different when really they are the same. This is one instance where the assistance of a domain expert is important; he may more readily be able to determine significant versus insignificant differences between applications.
- **Levels of commonality:** Another difficulty in performing a domain exploration is that commonality will be at different levels of abstraction. That is, the analyst cannot just look for common low-level subroutines; there may also be commonality at much higher levels, as well, e.g., subsystems, portions of subsystems, architectures, etc.

3.3.2 Application Characteristics

One of the early goals in the domain analysis process is to identify the characteristics of the domain which distinguish it from other domains. This information can aid in directing the detailed analysis. For example, in the CAMP domain, it was noted that within the existing applications there was a high degree of data interconnectivity and that heavy use was made of intermediate results. These observations led to the development of components that separated intermediate results from the components that performed the calculations, so that these intermediate results would be available to other functions.

3.4 Domain Modeling

There are several techniques that can be used in performing a commonality study. One technique that was used in the CAMP domain analysis was the *functional strip* method (see Reference [27]). This involved an analyst looking at a particular application function across all members of the domain representation set. For example, within the CAMP domain representation set, an analyst would investigate how strapdown computations were implemented in all of the missile software systems. The narrower the functional strips, the easier it is to detect commonality, although if the strips are too narrow, higher levels of commonality may be overlooked.

An object-oriented analysis of the domain and each member of the domain representation set can also lead to the identification of commonality across applications. The intersection of the analyses can be used as a good starting point for the development of one or more domain models and associated reusable software components.

One approach to analysis and model development is referred to as a *Canonical Domain Structure* approach (see Figure 3-4). In this approach, an analyst attempts to develop a canonical structure for the domain being analyzed. Development of this structure helps the analyst determine if an entity is common across multiple applications and helps the analyst find areas in which other common components should exist. The existence of a common diagrammatic representation in a textbook for the application is one indicator of the existence of a common architecture for an application area or domain. For example, the existence of a diagram of the structure of an autopilot in a textbook on the subject would be a good indication of a common architecture that could be represented via software parts.

Other modeling techniques can also be used in performing a commonality study, e.g., entity-relation modeling. Knowledge engineers in the artificial intelligence field are trained to work with domain experts to help them formalize their knowledge for later encapsulation within some type of knowledge-based system; many of the

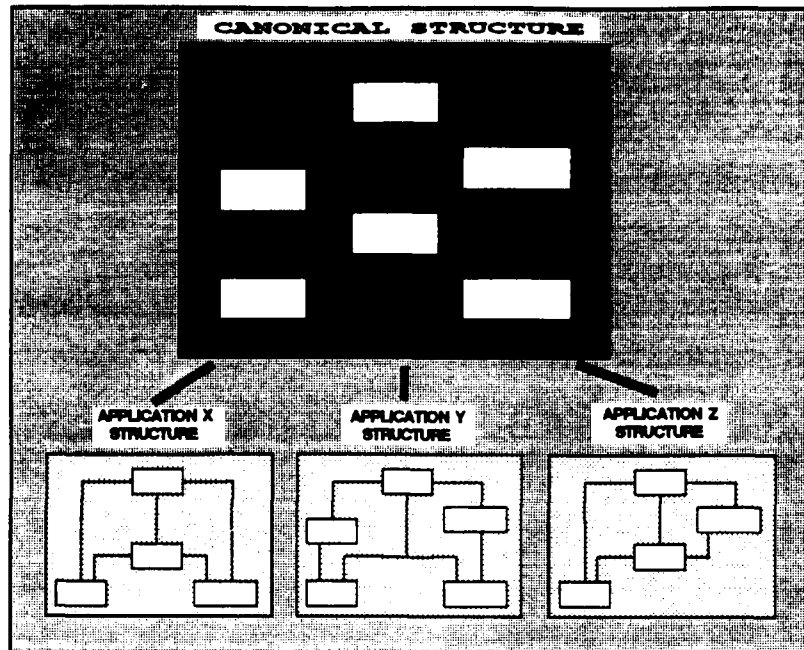


Figure 3-4: Using a Canonical Domain Structure in a DA

knowledge engineer's techniques may be applicable to performing a domain analysis, as there is significant overlap in the goals of both efforts.

The analysis techniques and model development are closely allied. The model representation is dependent on the analysis. For example, if object-oriented analysis techniques are used, an object-oriented model would, most likely, be developed, whereas if a functional approach were taken in the analysis, the model would, most likely, represent the functional structure of the domain.

3.5 Component Specification

When a common operation, object, or structure is found during domain exploration, it needs to be specified in a rigorous form so that the part designer can understand the requirements. One of the most important aspects of the specification is a clear definition of the variability that an object, operation, or structure must capture. The variability should accommodate future, as well as current, applications. Obviously, this is a challenging goal. Even with careful consideration and analysis, it will not be possible to anticipate all future uses of a component, and hence it will not be possible to incorporate all of the variability that may be needed.

When a part is designed, it must be clear which portions must be tailorable; this information must be captured in a preliminary software requirements specification for the parts. For example, if a part can be used with different types of matrix storage representation, then this must be recorded. A domain expert is often needed for this portion of the domain analysis, as well as for the earlier steps. In fact, it may prove useful to consult more than one expert in order to get a different perspective.

In many cases, a textual specification will be insufficient to capture the specification of a reusable component. Data flow analysis or an Ada-based specification can supplement, or in some cases replace, a textual description. Some work has been done on extending Ada into a specification language (for example, see Reference [7]). The specification technique must be able to accommodate components at different levels of abstraction.

3.6 Component Development Verification

The fact that an operation, object, or structure is found to be common during the domain exploration stage of a domain analysis is a necessary, but not sufficient, reason to build a part. It could be that the operation will never be needed again or that there will not be enough future applications which will use the part to justify its development. A project must establish both technical and business Go/No_Go criteria that will be used to decide whether or not to build a part. The criteria should be established prior to this verification activity, and should be developed in conjunction with the customer and domain experts. The criteria may depend, at least in part, upon available resources (i.e., there may be less strict criteria if greater resources are available). The Go/No_Go criteria can include factors such as those enumerated below.

- Number of uses a component would have seen in the domain representation set had such a part been available
- Relationship of the proposed component to past and current practices, and to anticipated future practices within the domain
- Complexity of the proposed component, i.e., a complex component that is expensive to develop for each application but that will be used relatively infrequently, may still be worth implementing
- Anticipated future use of the component

The number of uses a component would have seen in the domain representation set had such a part been available can be assessed by constructing a *domain utility matrix* that maps the common parts to the members of the domain representation set that would have been able to use that part. This can identify the extent of commonality of a "common" entity within a domain representation set. During the CAMP domain analysis, domain utility matrices were constructed for various categories of parts (Reference [28]). Figure 3-5 shows the domain utility matrix that was constructed for the CAMP Kalman filter parts.

MISSILE KF PARTS	A	B	C	D	E	F	G	H	I	J
	X	X	X	X		X		X	X	

Figure 3-5: CAMP Kalman Filter Parts Domain Utility Matrix

This type of mechanism can assist in verifying parts for inclusion in a parts set (and library). The technique also highlights the importance of selecting an adequate domain representation set. If the set does not include an adequate representation of the domain (including future trends), then sub-optimal decisions regarding which parts to construct may be made.

Domain experts play an important role not only in identifying commonality, but also in validating proposed parts. They should be cognizant of future trends and directions within their domains, and should use this information in evaluating the validity of a proposed reusable part.

3.7 Classification of Parts

Given the large number of parts that are usually identified during a domain analysis, it is imperative that a classification scheme be developed that will cover the domain. The classification scheme is one of the products of the domain analysis. The domain model and the classification scheme are generally closely related, with the model imposing a natural order on the set of reusable components.

A classification structure aids not only the domain analyst, but also helps in the retrieval and use of parts by end-users. A collection of reusable components does not have to be limited to one classification scheme. In fact, multiple classification schemes will provide the parts user with more than one perspective in identifying relevant parts.

Classification schemes can be categorized as faceted, enumerative, or domain-based. Each of these types of classification is discussed in more detail in subsequent paragraphs.

3.7.1 Faceted Classification Schemes

Faceted schemes are constructive in nature — each facet or category contains terms which can be used to describe an object. The classification of an object is *constructed* from terms in available facets (Reference [35]). When a user is searching a collection that has a faceted classification scheme, he specifies terms for each of the facets of interest. He does not need to specify terms for all of the facets, and generally, the order of specification of the terms does not matter. This approach was first applied in library science, and has been applied to several collections of reusable software components.

Faceted schemes provide a good deal of flexibility, but can be more difficult to develop and implement than an enumerative scheme. They are easily extensible, and, thus should be well-suited for use with large and expanding collections (Reference [35]). Experience to-date has indicated that they work best on collections of closely related items (References [13], [9]).

3.7.2 Enumerative Classification Schemes

Enumerative schemes are generally hierarchical and seek to enumerate all possible categories in the classification scheme. These schemes are generally straightforward to implement, but less flexible than faceted schemes, i.e., they are more difficult to expand (other than at leaf nodes) than a faceted scheme. In order to retrieve items from a set which has been classified with an enumerative scheme, the user must have a fairly good idea of the type of item he is looking for. For example, if a set of components were classified by keyword, the user must be able to select the appropriate keywords in order to find applicable reusable components. Depending on how the scheme was actually implemented (e.g., did the system include a feature to retrieve synonyms or to test for conceptual closeness of stored components), there could be a significant number of "misses" when searching, yet, this would not necessarily mean that applicable components were not available.

Enumerative schemes for classifying software can be based on any number of features of the collection of components. Some of these features are enumerated below, and discussed in the following paragraphs.

- Abstraction Level
- Domain Relevance (domain dependent, domain independent)
- Descriptive Attributes (e.g., keywords, developer, etc.)
- Instantiation techniques (i.e., the means of obtaining specific instances of a reusable component for use in an application)

3.7.2.1 Abstraction Level

Software components can exist at different levels of abstraction. The abstraction level can be used to categorize components. Three levels that have been identified are described below and depicted in Figure 3-6.

- **Functional:** Functional commonality involves a *black-box* view of common operations. This is the type of commonality that has traditionally been associated with software reuse. For example, a code segment that contains the equations for computing gravitational acceleration.
- **Pattern:** This involves the recognition that there are common patterns of logic that recur throughout families of software systems, and are more complex than simple black-box functions. For example, finite state machines occur in missile software systems in the platform caging function, the launch platform interface function, and many other functional areas.
- **Architectural:** Architectural commonality involves the recognition that there are common models of major components within software systems. This type of commonality is similar to pattern commonality, but is on a larger scale. Examples of architectural commonality within the missile operational flight software domain include a strapdown navigation subsystem or a pitch autopilot. At an appropriately high level of abstraction, there is a common model for each of these subsystems.

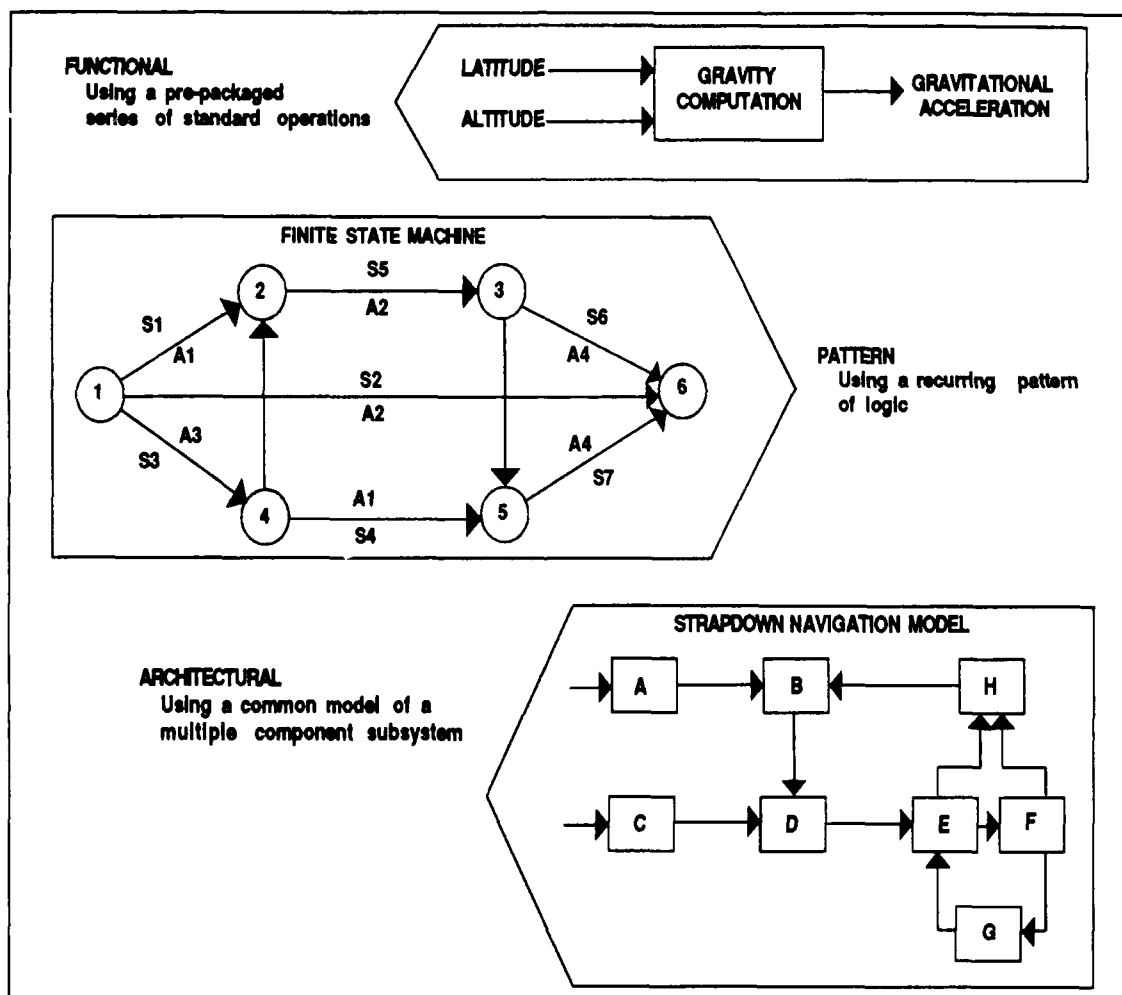


Figure 3-6: Three Levels of Commonality

The presence of multiple levels of commonality has a major impact on the performance of a domain analysis. It means that the domain analyst must look beyond common functions and low-level operations, and be able to develop abstractions of system objects and operations. This ability requires greater domain knowledge on the part of the analyst.

3.7.2.2 Domain Relevance

Parts can be classified as to their relevance to a particular domain. Domain-dependent parts provide objects and operations that are applicable primarily to the domain under investigation. The parts are said to belong to a specific vertical domain. Domain-independent parts provide objects and operations that are highly relevant to the domain under examination, but are also useful to other domains, e.g., math parts will be applicable to a broad spectrum of problem domains. These types of parts are said to belong to a horizontal domain. See Paragraph 3.1 for a further discussion of horizontal and vertical domains.

3.7.2.3 Descriptive Attributes

Software parts can be described by a number of attributes, e.g., keywords, developer, hardware dependencies, project for which they were developed, etc. These attribute values can be used in the classification process to categorize the parts. Each component in the collection can be described by a set of attributes. Providing access to reusable components via these attributes provides the user with different views of the available resources.

One attribute that is often used to classify collections of reusable software is an attribute that captures the area of functionality of the component. The areas can be arranged hierarchically to form a *taxonomy*. The CAMP parts taxonomy is shown in Figure 3-7; Table 3-3 describes each of the categories in the CAMP parts taxonomy.

3.7.2.4 Instantiation Techniques

In addition to classifying parts by abstraction level, domain relevance, and descriptive attributes, a reusable component can be classified by the way specific instances of the part are generated. Three categories of parts that are based on instantiation method are *simple*, *generic* (both simple and complex), and *schematic* (see Figure 3-8). *Simple* parts are parts that can be used "as is" with no tailoring (i.e., what you see is what you get). Although reuse of simple parts may be of benefit, the real productivity gain will come from reusing more complex structures. The problem with this is that complex structures are generally more difficult for the software developer to incorporate into a new application. The key is to provide him with a means of reducing their perceived complexity (e.g., a system model of the parts) and facilitating their use (see Reference [7]).

Generic parts are parts that take advantage of the Ada generic facility for tailoring the part for a particular application. A *simple generic part* is a software part that can be instantiated in a straightforward manner with data type and subroutine information. This is a fairly low-level part with little or no dependence on other parts. For example, a generalized first-in-first-out (FIFO) queue part could be developed in which the type of the data object to be queued would be supplied and a specific FIFO queue component would be instantiated for that type. In languages other than Ada, routines can sometimes be developed that perform something akin to generic instantiation.

Meta-parts allow a *family* of components to be generated from a single part. This is advantageous because it may not be feasible from a catalog size or user perspective, or cost-effective to have all family members in a reusable parts library. Within the Ada realm, there are two types of meta-parts: *complex generic parts* and *schematic parts* — they are distinguished by how specific instances of a family are obtained. A *complex generic part* is a part from which specific instances can be obtained via the Ada generic facility. These instantiations are more complex than for simple generic parts; there may be multiple layers of generics with complex interactions (e.g., the CAMP Kalman filter or autopilot parts).

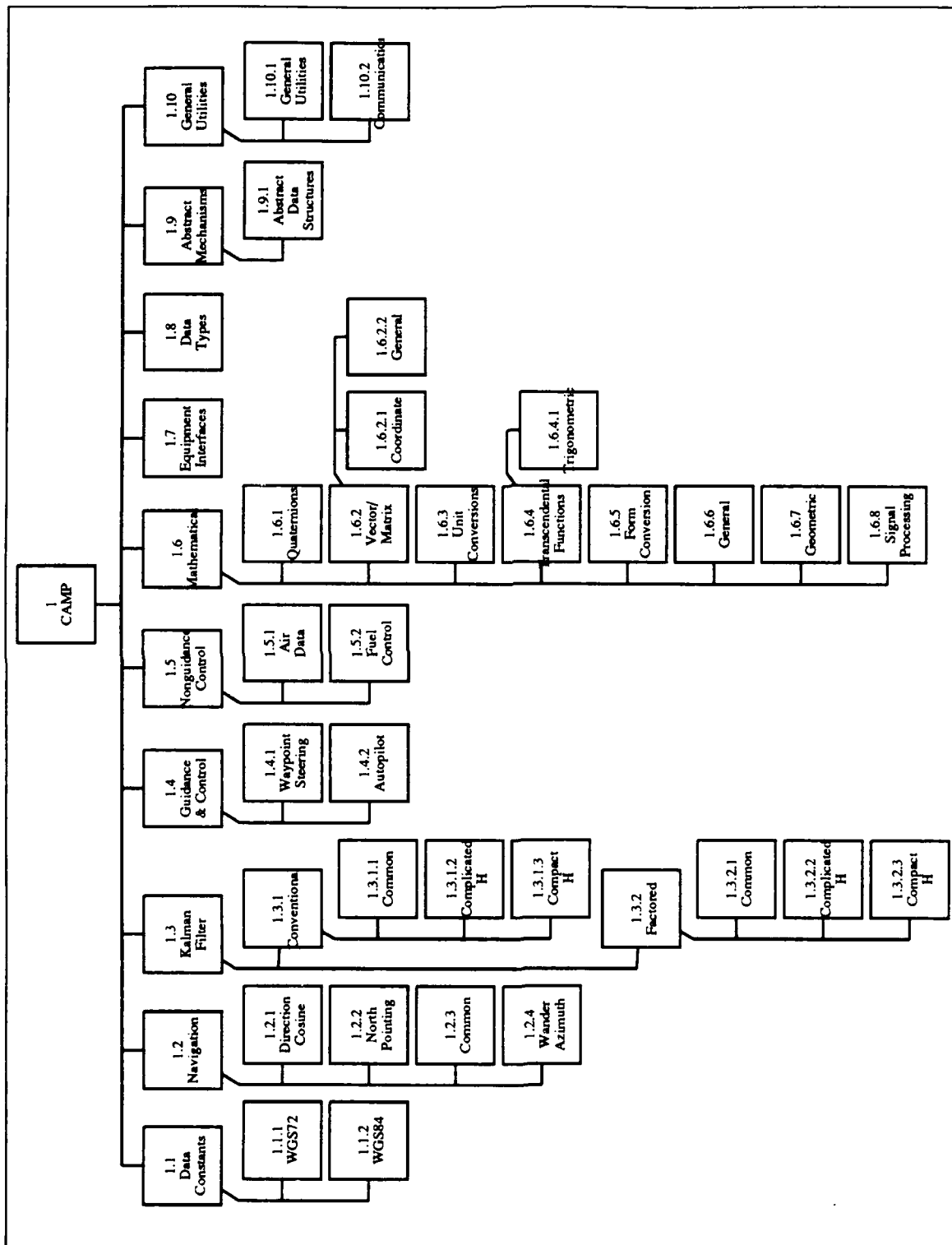


Figure 3-7: CAMP Parts Taxonomy

Table 3-3: CAMP Parts Taxonomy

CATEGORY	NAME	DESCRIPTION
Data Constants	WGS72_Ellipsoid_Engineering_Data WGS72_Ellipsoid_Metric_Data WGS72_Ellipsoid_Unitless_Data WGS84_Ellipsoid_Engineering_Data WGS84_Ellipsoid_Metric_Data WGS84_Ellipsoid_Unitless_Data Universal_Constants Conversion_Factors	Parts which provide data constants used in a typical missile application
Navigation	Common_Navigation_Parts Wander_Azimuth_Navigation_Parts North_Pointing_Navigation_Parts Direction_Cosine_Matrix_Operations	Parts which provide the basic functionality of a navigation subsystem
Kalman Filter	Kalman_Filter_Common_Parts Kalman_Filter_Compact_H_Parts Kalman_Filter_Complicated_H_Parts Factored_Kalman_Common_Parts Factored_Kalman_Filter_Compact_H_Parts Factored_Kalman_Filter_Complicated_H_Parts	Parts which provide common Kalman filter functions
Guidance and Control	Waypoint_Steering Autopilot	Parts which provide the basic functionality of a guidance and control subsystem
Nonguidance Control	Air_Data_Parts TLCSC Fuel_Control_Parts TLCSC	Parts which provide the basic functionality of a control subsystem for operations outside of the guidance area
Mathematical	Coordinate_Vector_Matrix_Algebra General_Vector_Matrix_Algebra Standard_Trig Geometric_Operations Signal_Processing Polynomials General_Purpose_Math Unit_Conversions External_Form_Conversion_Twos_Complement Quaternion_Operations	Parts which provide a variety of useful mathematical functions such as coordinate and matrix algebra, trigonometric, and signal processing functions
Equipment Interfaces	Missile_Radar_Altimeter Missile_Radar_Altimeter_with_Autopower_On Clock_Handler	Parts which provide standard interfaces to specific hardware components or to general classes of hardware
Data Types	Basic_Data_Types Kalman_Filter_Data_Types Autopilot_Data_Types	Parts which provide data types used in other parts or in a user application
Abstract Mechanisms	Abstract_Data_Structures	Parts which provide abstract data structures and processes
General Utilities	General_Uilities Communication_Parts	Parts which provide other functions needed for missile or other weapons system operation

Schematic parts capture commonality that cannot be captured directly in the implementation language. A schematic part consists of a blueprint together with construction rules for building a specific instance of the part. Schematic parts differ from Ada generic parts in two important aspects:

- The generation of specific parts from a schematic part cannot be achieved via an existing Ada facility, and thus, is not language-specific, i.e., it can be accomplished in languages other than Ada.
- There is no complete, compilable code to view until a specific component is generated from a schematic part. For example, a finite state machine (FSM) may have actions associated with state

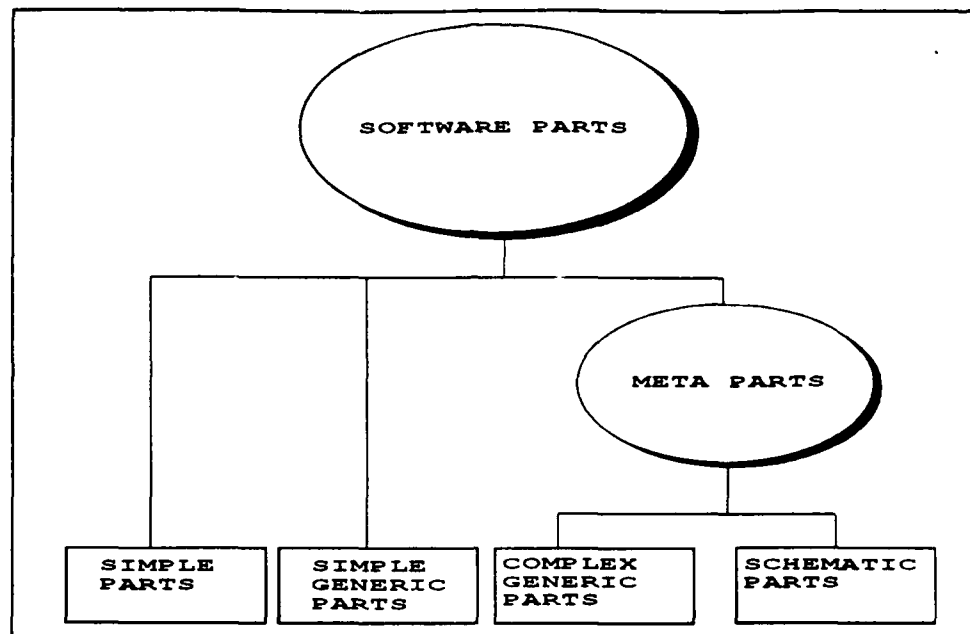


Figure 3-8: Types of Parts

transitions. Although most software engineers would readily know how to implement an FSM, the commonality cannot be captured using Ada alone because Ada does not provide for procedure types that would allow the association of actions with states or transitions. Because the structure of such a part is well-known, it is fairly straightforward to capture the construction rules and generate customized components based on user inputs for states, transitions, and actions.

These types of parts are referred to as *schematic* in order to convey that they capture both the design of the part and the construction rules. Schematic parts can sometimes be identified by the fact that there is some common understanding among "experts" of the implementation process, yet there is not a straightforward way to parameterize the implementation in the given implementation language. Schematic parts are discussed in more detail in Chapter 19.

The introduction of schematic parts greatly increases the range of variability that can be built into reusable components, i.e., reusable parts are no longer limited to software whose variability can be expressed using the implementation language features alone. In order to maximize the benefits of schematic parts, an automated means of translating the blueprint into an application-specific component is needed. The CAMP program prototyped a tool referred to as a *component constructor* that generated specific instances of a schematic part based on user-supplied requirements. Although a number of component constructors for generating application-specific Ada components from schematic parts were developed during the CAMP prototype tool development effort, much more work needs to be done in this area. The method employed in the initial prototype required a significant effort to implement and maintain. Other researchers have also investigated composition/generation techniques.

3.7.3 Domain-Based Classification

Domain-based or application-based classification allows the user to search a collection of components based on domain or application knowledge rather than on attribute values or some other previously discussed criterion. This requires the existence of an underlying domain model and application knowledge. In the case of domain-based search and retrieval, available components can be associated with each node of a domain model, allow-

ing the user to traverse the model and obtain information about relevant parts at each node. An application-based search scheme associates reusable components with portions of the application area.

This type of facility was also prototyped on the CAMP program in the AMPEE System Parts Identification functions (Reference [30]). These functions provide the user with a means of identifying potentially applicable reusable components based on knowledge the user has about the domain or application, rather than on specific information about the available components. They provide high-level access into the parts catalog, and can be used early in the software development lifecycle to identify potentially applicable software parts for the user. The selection of parts is based on information that the user provides about his application or based on a model of the application domain.

This type of facility can facilitate cost estimates, timing and sizing estimates, and help maximize reuse by heightening awareness of available parts early in the software development lifecycle. It can be used during proposal (or even pre-proposal) preparation, as well as during system requirements and design activities to provide early identification of potentially applicable parts. Once the user has identified candidate components, he can use a parts catalog to obtain specific information on such things as performance, environmental requirements, restrictions, etc. Although this type of facility was demonstrated on the CAMP program in the missile operational flight software domain, it has broad applicability to other domains as well. These techniques are discussed in more detail in Chapter 18.

3.8 Tools

Domain analysis has not been part of the traditional software development process, thus there is no standard set of tools to support this activity. Existing tools which can aid during the DA process can be drawn from the following areas.

- Software Requirements Engineering: Tools in this area are typically used to specify, analyze, and document software requirements.
- Artificial Intelligence Knowledge Engineering: Tools in this area are typically used to collect, organize, and analyze knowledge within a domain of interest.
- Library Sciences: Tools in this area are used to classify and organize information coming from many diverse sources.

Some of the types of tools that may support domain analysis are identified below.

- System analysis tools to facilitate the analysis of existing applications in the domain
- Tools that help the domain analyst find commonality (for example, some type of abstraction and pattern matching tool set)
- Tools that help the analyst handle the large amounts of raw data inherently involved in a DA (e.g., databases, editors, data screening or filtering tools, and categorization tools)
- Modeling tools for use in the development of a domain model. For example, entity-relationship-attribute (ERA) modeling may be used to model the domain, thus tools that support ERA model development can also support domain analysis.
- Specification tools that help the domain analyst specify commonality (e.g., domain modeling tools, requirements specification tools)
- Knowledge engineering tools for use in acquiring and processing domain information.

Because there is still considerable research being performed in the area of domain analysis, the question of optimal tool support is largely unresolved. This should not be a significant barrier to local reuse efforts because, for smaller domains or for a single project, many of the domain analysis tasks can be performed using commonly available editors, database management systems, drawing tools, etc. The lack of tool support does become a significant problem as the domain is broadened and the quantity of data to be processed increases.

3.9 Management of the Domain Analysis Process

The domain analysis team should consist of a senior software engineer trained in modern software engineering methodologies, and one or more domain experts. A domain expert is needed to provide insight into the domain that the software engineer may not have. Domain expert involvement is critical! Lack of domain expert involvement can lead to the development of reusable components that bear little relationship to what is really needed by domain application developers.

Periodic reviews during the domain exploration process provide valuable input to the domain analysts, make the customer more comfortable with the process, and provide feedback to management on the progress of the domain analysis. These reviews help to ensure that the correct domain is being addressed. Although the domain analysis should not be too narrowly constrained — some "wandering" can lead to the identification of additional commonality within the domain of interest — reviews can prevent the wandering from going too far afield. Reviews should be held at different levels, but ultimately should include the domain analyst(s), domain experts, the customer, and management. Domain expert involvement in the review process is critical in validating the results of the commonality study.

3.10 A Case Study: The CAMP Kalman Filter Parts

The following case study illustrates the process of identification, development, and deployment of reusable software components in a real-time embedded application area.

Kalman filters play a significant role in missile operational flight software, and thus are represented in the CAMP parts set. Basically, a Kalman filter for missile flight operations combines external position or velocity measurements with internal position or velocity measurements, taking into consideration the uncertainty of the measurements when calculating an optimal estimate of missile position and velocity. The Kalman filter also produces an optimal estimate of inertial sensor errors. Figure 3-9 presents a high-level view of missile Kalman filter operations.

3.10.1 The Domain Analysis

During the CAMP domain analysis (Reference [28]), a common architecture of Kalman filter operations was found in the representation set (i.e., in the set of 10 missile software systems that were examined); it is depicted in Figure 3-10. In all, 13 parts were identified to facilitate the implementation of a Kalman filter for missile operations. These parts are identified in Table 3-4 and discussed in detail in Reference [28].

One of the products of a domain analysis (in addition to the identification of common objects, operations, and structures) is the identification and specification of the variability that must be provided (see Paragraph 3.5). In the Kalman filter parts, variability was found in the following items:

- The number of states in the Kalman filter
- The number of measurements in the Kalman filter

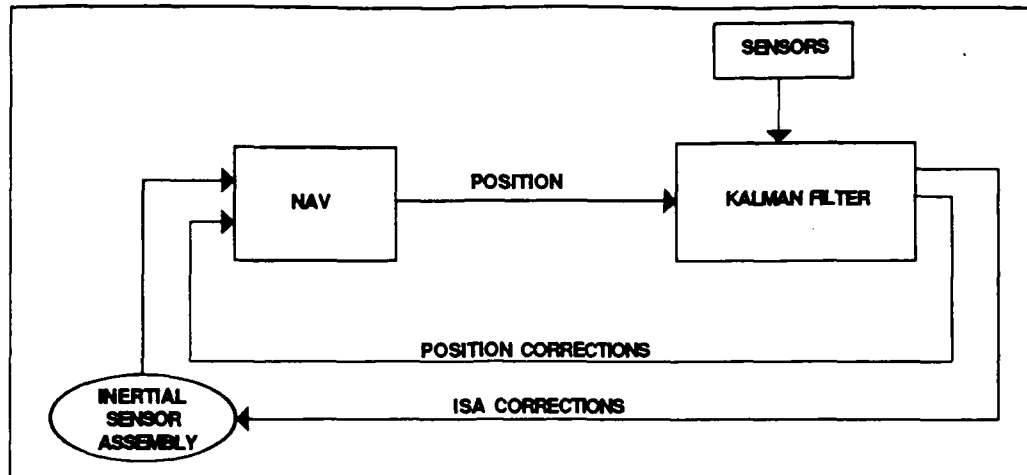


Figure 3-9: High-Level View of Missile Kalman Filter Operations

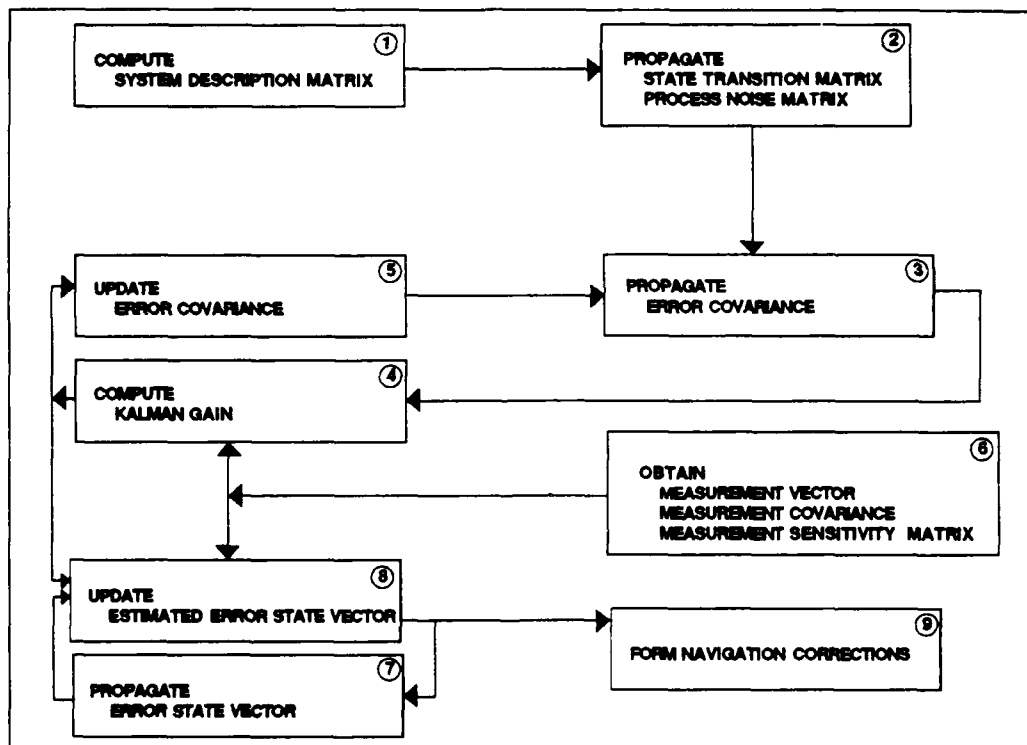


Figure 3-10: CAMP Kalman Filter Common Architecture

- The data type of the elements in the Kalman filter matrices
- The form of the measurement sensitivity matrix (referred to as "H"). This matrix could take two different forms: "Compact_H" and "Complicated_H". The Complicated_H parts assume that the measurement sensitivity matrix is sparse, which allows for a more complicated relationship between a measurement component and the components of the state vector. In the so-called "Compact_H" parts, it is assumed that in any given row of the measurement sensitivity matrix, there is only a single "1", with the other elements being zero (this implies that any measurement component is a direct measurement of a single component of the state vector). Ten of the Kalman filter parts provide an alternative representation for the measurement sensitivity matrix.

Table 3-4: Initially Identified Kalman Filter Parts

- Propagate State Transition and Process Noise Matrices
- Propagate Error Covariance Matrix Manager
- Kalman Update
- Kalman Update (Complicated_H Version)
- Propagate State Transition Matrix Manager
- Propagate State Transition Matrix
- Compute Kalman Gain
- Compute Kalman Gain (Complicated_H Version)
- Update Error Covariance Matrix
- Update Error Covariance Matrix (Complicated_H Version)
- Update State Vector
- Update State Vector (Complicated_H Version)
- Sequentially Update Covariance Matrix and State Vector
- Sequentially Update Covariance Matrix and State Vector (Complicated_H Version)

(Note: The measurement sensitivity matrix (H) relates the external measurement vector (Z) to the state vector (X) as follows: $Z=H \cdot X$. External measurements can be obtained from radar or barometric altimeters, TERCOM, DSMAC, GPS, or some other external system. The state vector contains estimates of errors in navigation parameters and in inertial sensor data). Reference [28].

Domain analysis also revealed that Kalman filter developers use matrix representations that cannot be captured as Ada generic units. Although the CAMP parts set provides several matrix representations intended to provide greater efficiency than a full storage representation (e.g., half-storage, diagonal, dynamically sparse), these would not meet the requirements of missile Kalman filter operations. Kalman filter developers use a *static sparse* representation for some of the larger, heavily used matrices in Kalman filter operations. A *static sparse* representation requires that the developer know in advance the elements of the matrix that will be non-zero. The representation and operations then capture and operate only on those non-zero elements, saving storage (because only the non-zero elements are stored) and execution time (because the operations are performed only on the non-zero elements of the matrix); program size can increase because program loop constructs are removed in favor of operating on individual elements of the matrix. Because of the size of the matrices, development of the code to perform the required matrix operations is a tedious task at best; the software developer must manually determine which elements of the matrices will need to be multiplied together and then produce the code, hoping he does not make any errors with indexing. In principle this process is straightforward, thus, the generation of the static sparse matrix representation and the associated operations can be captured in a schematic meta-part and used as the basis for a *component constructor*. Meta-parts and component constructors are discussed in more detail in Chapter 19 in Section III, *Maximizing Software Reuse*.

3.10.2 Architecture of the Kalman Filter Parts

One of the primary design goals of reusable software parts is flexibility that will allow the parts to be tailored to fit the needs of a wide range of applications. To this end, the Kalman filter parts were designed in such a way that the software developer does not need to use all of the CAMP parts, i.e., he can replace any of the CAMP Kalman filter parts with his own or other parts as long as the interfaces are satisfied. The structure remains the same regardless of whether the user incorporates "compact" or "complicated" parts. Figure 3-11 depicts the structure of the CAMP Kalman filter parts. This structure has, in fact, been maintained as the CAMP Kalman filter parts set has been extended.

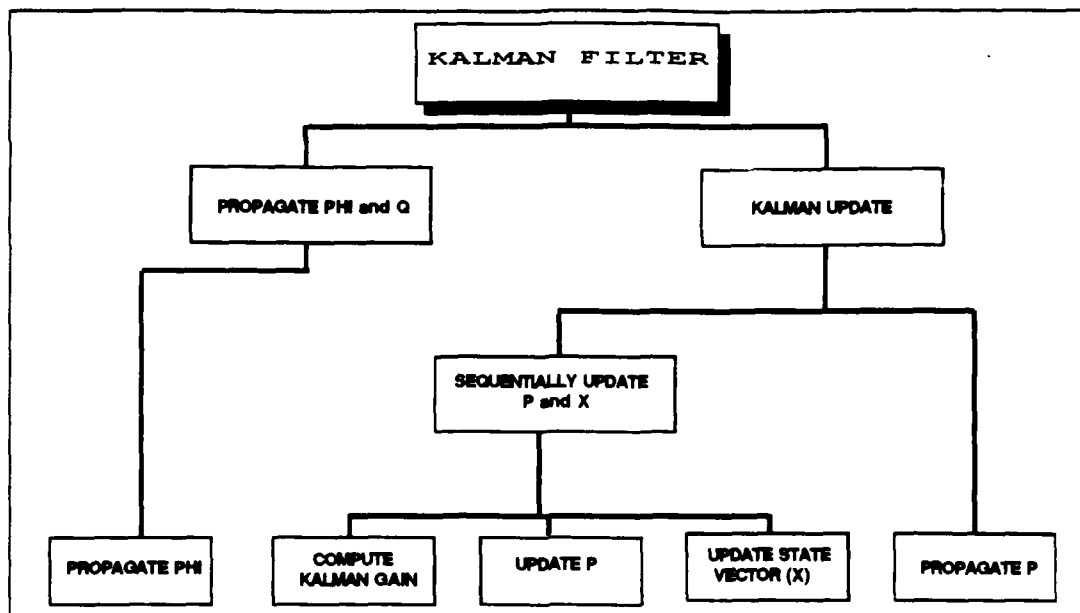


Figure 3-11: Software Parts Use in Missile Software Systems

An analysis of the CAMP domain representation set revealed that, with the exception of the Complicated_H parts, all of the CAMP Kalman filter parts could have been used in 7 out of 10 of the missile software systems represented. This can be represented in a domain utility matrix as it was originally presented in the CAMP Final Technical Report (see Reference [28]). A similar matrix is depicted in Figure 3-5.

3.10.3 Implementation

The Kalman filter parts identified by the original CAMP domain analysis were implemented, tested, and then used on a demonstration program. This experience highlighted a number of pitfalls of parts development.

During development of the CAMP Kalman filter parts, there were repeated reminders of the importance of domain expert involvement in parts development for such a complex domain. Kalman filter development is generally performed by guidance and control engineers with several years of experience (one is not considered an expert in the domain for many years). Because of the severe size and speed constraints imposed on missile flight operational software, efficiency is of the utmost concern to these developers.

Although a domain expert assisted with the domain analysis, the parts development team still encountered problems with requirements. The main problem was one of communication. Because of the English-like nature of Ada, the domain expert, who was presented with a design developed in Ada, did not always realize that he lacked a full understanding of the material presented to him. Conversely, the software engineer developing the Ada design thought he had a better understanding of the domain than he did, and hence, made unwarranted assumptions.

These are very common problems in parts development. The apparent simplicity of Ada (i.e., its English-like syntax) can be deceptive. Additionally, it is fairly common for someone to fancy himself a domain expert after relatively short exposure to a topic. In an area as complex as Kalman filters, this can only cause problems. Although there is no sure-fire way to eliminate these problems, awareness that they can occur, combined with on-going, in-depth reviews by software engineers and domain experts can reduce their occurrence.

The problem is not so much one of errors in the code, as it is that the parts do not have the functionality needed to support the domain for which they are built. In the case of the CAMP Kalman filter parts, the parts underwent requirements specification, top-level and detailed design, and testing, with no realization that the parts would not meet the needs of a Kalman filter developer. The catalyst for this discovery was a request for additional test data from the domain expert. In working through the test data, errors were uncovered in the parts that had been implemented. This highlights the need for domain expert review throughout the parts development activity. Discovery of errors this late in the development resulted in the need for significant changes to the parts and to a tool that was under development to facilitate the use of the parts.

3.10.4 Parts Use and Refinement

The CAMP 11th Missile demonstration program used the CAMP parts in an actual missile application (see Appendix II). As a result of their use on this program, the Kalman filter parts underwent modification and extension. During the CAMP parts maintenance effort, additional Kalman filter parts were added to the CAMP parts set; these provide an alternative approach to that implemented in the original parts set. Development of those parts required additional mathematical parts as well. Because of the flexibility inherent in the original design, the additional type of Kalman filter was able to take advantage of the structure and many of the existing parts.

The case study in Chapter 13, Paragraph 13.5 illustrates the use of the Kalman filter parts in a new application.

3.11 Summary

The main steps in a domain analysis are identified in Table 3-5. Parts developers do not generally proceed sequentially through all of these steps. Once a significant portion of the domain has been analyzed, parts development can begin. Generally, additional parts (i.e., common objects, operations, and structures) will be identified after the domain analysis has been "completed" and parts implementation has begun. For example, during the initial CAMP domain analysis, approximately 250 parts were identified, but by the time the initial parts set was delivered, it consisted of over 450 parts. Additional parts were identified that were needed to support previously identified parts. Use of the parts can result in the identification of a need for both part variants and additional parts.

Table 3-5: Steps in a Domain Analysis

- Select the domain
- Define or bound the domain
- Select a representation set
- Explore the domain
- Identify common objects, operations, and structures
- Develop a domain model
- Develop a classification scheme
- Classify the parts
- Identify a requirements specification technique and design methodology
- Develop preliminary software requirements for parts
- Verify that the identified parts should be developed
- Conduct in-process reviews by expert domain review team

It may prove advantageous to structure the parts development group so that the domain analysts and the requirements/design engineers can operate in parallel. Although the structure of the group can vary quite

significantly, the composition is critical to the success of the endeavor. This is not a task that can be performed well by inexperienced software engineers. Highly skilled, creative people are needed; they must have experience with modern software engineering methodologies, techniques, and tools. Inputs from domain experts are critical.

Table 3-6 summarizes the major products of a domain analysis.

Table 3-6: Major Products of a Domain Analysis

- | |
|--|
| <ol style="list-style-type: none">1. Domain model2. Classification scheme3. Preliminary specifications for the reusable components |
|--|

CHAPTER 4

REQUIREMENTS DEFINITION

Requirements definition is a critical, but historically problematic step in the software development process. Incorrect requirements are frequently the source of problems that become apparent later in system development. Much effort has been expended over the years in developing methods, techniques, and tools to reduce errors in requirements definition and specification, yet, there is still no ideal solution.

Requirements engineering for reusable software introduces another element of complexity to an already complex process. Reusable parts must address a widerange of system requirements, rather than the requirements of a single system. Thus, their specification must allow for their integration into a variety of different contexts, and facilitate composition into larger software entities. While inputs and outputs will form the basis of interfaces to a part, the requirements must also address the issue of integrating lower level parts into higher level parts or subsystems.

The requirements must be abstracted to such a level as to allow functionality to be apparent. Application-specific details should not introduced into the requirements for reusable components, i.e., the requirements should reflect the wide applicability of the part. The parts will be incorporated into systems that have not yet been defined, thus, the end-user must be able to integrate these parts into more complex operations. Obviously, the parts must be flexible in their application. The specification should identify environmental dependencies and requirements. While these issues would normally be considered even for single-use software, they are particularly important for reusable code.

4.1 Requirements Development

Parts identification and development can be project-driven or under the control of an independent parts development group; it is generally best if *parts identification* is project-driven and *parts development* is performed by a specialized parts development group. In this environment, the part developer may have little idea of how a part will ultimately be used. This is a compound problem — reusable software developers generally do not know how the reusable components will be used, and the developers generally lack the degree of domain knowledge held by the domain expert. Because of this, it is very easy to develop incorrect requirements. Thus, it is critical to have the involvement of a domain expert in the requirements review process. This involvement is necessary to ensure both correctness and that truly usable parts are developed.

The separation of domain expertise and implementation expertise was identified as a source of several errors during parts development on the CAMP project. Although a domain expert was involved in the domain analysis and some of the requirements specification, most of the requirements were developed by software engineers in parallel with or after the domain analysis. When the requirements were developed by software engineers, they were reviewed for correctness by domain experts. After the review, the requirements were turned over to software engineers who, in many cases, did not have domain expertise. Two problems arose from this situation:

- In some cases, the software engineer's lack of domain expertise caused him to misinterpret the requirements.
- In other cases, the domain expert thought that he understood the requirements and/or design that were presented to him for review, but his lack of expertise in the specific programming language caused him to misinterpret them.

In at least one case, the software engineer thought he understood the application area, parts were developed

for a set of complex generic parts, a software tool was developed to support the use of these complex generics, and when test data was sought from the domain expert, the domain expert identified a deficiency in the original requirements. In this case, 10 packages were affected —

- | | |
|-------------------------------------|--------------------------------------|
| – General_Vector_Matrix_Algebra | – Common_Navigation_Parts |
| – Kalman_Filter_Data_Types | – North_Pointing_Navigation_Parts |
| – Kalman_Filter_Common_Parts | – Wander_Azimuth_Navigation_Parts |
| – Kalman_Filter_Complicated_H_Parts | – General_Purpose_Math |
| – Kalman_Filter_Compact_H_Parts | – Direction_Cosine_Matrix_Operations |

These parts had already been designed, coded, and tested. Obviously, errors found at this late stage in the development cycle are much more critical and costly to correct than errors found at a requirements walkthrough.

Although the need for stable, well-defined requirements is universal, the problems raised by poor requirements are exacerbated in a software reuse environment. In many cases the software parts may be developed by software engineers who lack specific application domain expertise, making it particularly important that the requirements for reusable software be complete, with greater attention to detail, than requirements for custom code. While this problem might not occur if the reusable software were being developed by the same groups that were going to use it, it is a problem that is likely to occur whenever one group is developing reusable parts for other groups.

The requirements need to be developed with an eye toward generality. Both the developer and the reviewers need to evaluate whether the requirements are for a specific implementation of a given function or equation, or for the general case. There is nothing inherently wrong with implementing a specific case as long as this is recognized and intended.

When developing the requirements for an operation, extra attention should be paid to identifying any portions of that operation which can be further decomposed and developed as separate reusable parts. For example, a vector length calculation should not be built into the requirements for another part, but rather the requirements for a vector length should be specified separately and then the vector length part can be used when specifying other requirements.

4.2 Requirements Definition

A requirements definition for a part should contain functional, performance, and environmental requirements. It should identify the objects and operators that are encapsulated or exported by the part — it must capture both the algorithm and the data flow. It should facilitate communication of a part's characteristics and provide data on the environment required by the part, including dependencies which exist between parts. For example, during the specification of the CAMP parts, a textual and a graphical technique were used. The textual technique consisted of a written description of the part; the graphical technique consisted of a data flow diagram. Figure 4-1 contains an example of a CAMP Software Requirements Specification for a reusable part; the requirements were documented in accordance with DOD-STD-2167. This level of detail for the specification of simple parts may not be required, i.e., a textual requirements specification would probably suffice for simple parts. More complex parts did benefit from the use of data flow diagrams.

Some researchers have advocated the use of formal specifications for reusable parts. Although there are some advantages to the use of a formal notation, at this time the benefits do not appear to outweigh the costs. One

3.4.3 Navigation Parts

3.4.3.1 [R185] Compute East Velocity

This part shall calculate the East velocity component of a missile when a local level, wander azimuth coordinate system is being used for navigation. Figure 3.4.3-1 depicts the data flow nature of this part.

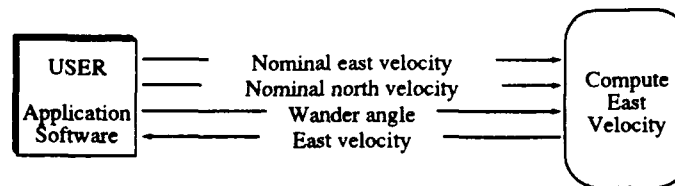


Figure 3.4.3-1. Data Flow of R185

3.4.3.1.1 Inputs

The meaning and typing of the input parameters for this part are given below.

- (a) Nominal East Velocity (VEL_{NE}): This is the velocity in the direction which would be East if the wander angle is zero. It is of type *Velocity*.
- (b) Nominal North Velocity (VEL_{NN}): This is the velocity in the direction which would be North if the wander angle is zero. It is of type *Velocity*.
- (c) Wander Angle (WA): This is the value of the wander azimuth angle. It is of type *Angle*.

3.4.3.1.2 Processing

The processing performed by this part is described in the following equations.

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

3.4.3.1.3 Outputs

The meaning and typing of the output parameters for this part are given below.

- (a) East Velocity (VEL_E): This is the calculated velocity in the East direction. It is of type *Velocity*.

Figure 4-1: Requirements Specification for a Reusable Part

suggested benefit of formal specifications is that they provide a basis for code generation via some transformation system. While this may be true, transformational systems for code generation are still, for the most part, in the research stage, and the material presented in this manual emphasizes techniques that can be used now. Another benefit of formal requirements is that they allow for the precise definition of a part, and thus, may facilitate the mapping of user requirements to available parts. The cost to the user of formal notation is generally quite high, as it requires him to learn another language above and beyond the implementation language.

Extensions to Ada have been proposed in order to enrich the language into a requirements language. This is usually proposed in conjunction with a tool that would perform analysis of the requirements and then go on to produce at least the skeletal code (e.g., Reference [7]).

4.3 Requirements Review

Requirements developed for reusable software should be subjected to rigorous review. With custom software, there would generally be no reason for outside groups to review requirements specifications for another project, but in the case of reusable software, it may be highly desirable, if not necessary, in order to establish that the parts will meet the needs of future developers. External review teams should include both domain experts and potential users of the parts.

Although it may not be necessary for a domain expert to actually develop the requirements, he must be involved in a review of those requirements. The implication of this is that the requirements must be specified in a form that is comprehensible to the domain expert. Graphical techniques may prove to be particularly useful.

4.4 Definition of a Part

There is no standard definition of a software component or part, but a project or organization involved in software reuse will need to develop or adopt a definition. This will facilitate the collection of metrics associated with software reuse. Metrics cannot be collected about reusable components unless a reusable component can be identified. An issue to be considered in determining what constitutes a part is whether each software lifecycle object (LCO) associated with a reusable object, operation, or structure should be considered an individual part, or whether the entire bundle of LCOs should be considered as one reusable component. Eventually, the software community will need to standardize on a definition so that data about software reuse can be meaningfully exchanged and compared.

There are many alternative definitions. The set of criteria established on the CAMP program for identifying Ada code parts is enumerated below; associated LCOs were not treated as separate parts, i.e., only Ada code parts were counted. Obviously, other types of reusable software components would require a different set of criteria.

1. A part is an Ada package, subprogram, or task. A part can be any level of software unit (e.g., a CSC or CSU in DOD-STD-2167A terminology).
2. A part must be usable in a standalone fashion.
 - It may *with* other parts.
 - It does not depend on other packages, subprograms, or tasks encapsulated with it to perform a single function.

Figure 4-2 shows an example of a higher level generic part — `Clock_Handler`. The `Clock_Handler` package maintains a clock. Even though a single application may not require all of the routines in `Clock_Handler`, the routines could not logically exist alone: it would make little sense to reset a clock that is never read. Therefore, the entire package is considered a part.

Figure 4-3 shows an example of a lower-level generic part — `Latitude_Integration`. This package maintains a latitude. It is designated as a part because, although the `Integrate` function could exist on its own, the `Reinitialize` procedure could not.

Figure 4-3 also shows an example of a generic procedure, `Compute_Coriolis_Acceleration`, which is a part. Figure 4-4 shows an example of a generic package, `Vector_Operations`, which contains several subroutines, each of which is a part. In these cases, the procedures, rather than any encapsulating packages, are designated as parts since the procedures can logically exist on their own.

- A part may require types or objects that have been encapsulated with it. The subroutines shown in Figure 4-4 are parts even though they require the data type, `Vectors`, defined by the `Vector_Operations` package.

3. Organizational packages are not parts; and package bodies are never parts, even if they have processing within them. Organizational parts are parts that group logically related parts together, such as the CAMP General_Vector_Matrix_Algebra package that contains a collection of packages that define general-purpose matrix types and operations.

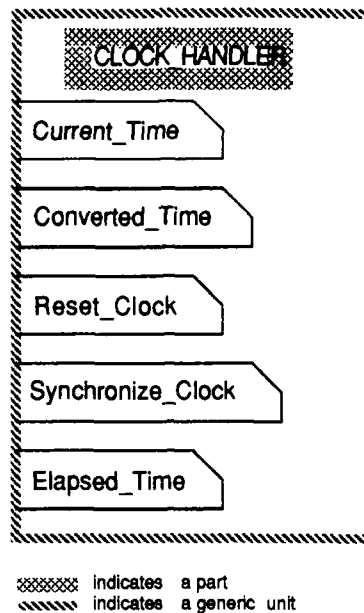


Figure 4-2: A Generic Package Can Be a Part

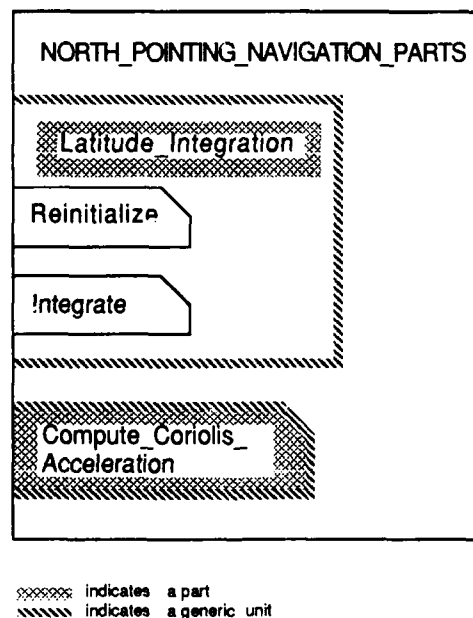


Figure 4-3: Generic Packages and Subprograms Can Be Parts

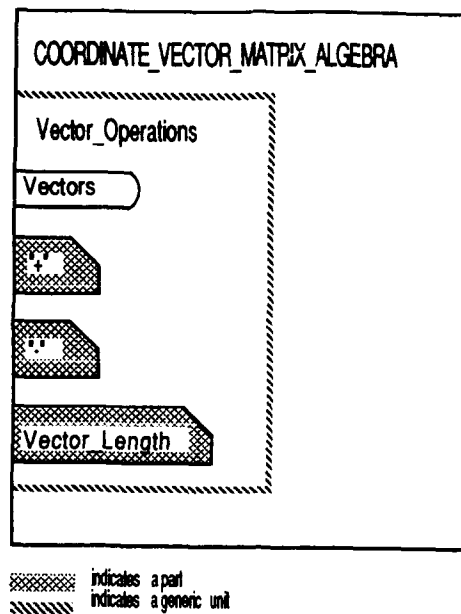


Figure 4-4: Non-Generic Units Can Be Parts

CHAPTER 5

ARCHITECTURAL DESIGN of SOFTWARE PARTS

Reusable components must be designed with reuse in mind. It is generally not enough to simply use good design principles and expect that another project will be able to lift packages, procedures, and functions out of existing code for use in their application. Code designed and written without reuse in mind (even Ada generic units) may be too tightly coupled to its original environment to be reusable.

Development of reusable software presents the designer with a conflicting set of design goals. In addition to being reusable on a number of different real-time embedded applications, the design of reusable parts must address flexibility, efficiency, reusability, ease of use, and protection against misuse.

In this chapter, the design methodology developed and used on the CAMP program for Ada parts development will be discussed. Six alternative design methodologies were explored before one — the semi-abstract method — was selected.

5.1 Overview of the CAMP Semi-Abstract Data Type Method

The CAMP methodology centers on what is referred to as the *semi-abstract data type*. Many researchers, particularly those not well-versed in the special needs of RTE applications, have advocated the use of pure abstract data types. This is a conceptually clean approach, but in practice may not be suitable for RTE applications. The abstract data type approach limits accessibility of the data structures, whereas the *semi-abstract data type* approach allows the user to directly manipulate them. Although this may not generally be a good practice, it is often essential in RTE applications, and when done cautiously, is acceptable. This direct access provides the user greater control in fine-tuning the efficiency of his application. The CAMP methodology makes use of strong data typing in order to provide the user with protection against misuse of the part by enabling the compiler to perform checking on the types provided; the burden of strong data typing is eased by providing defaults for generic parameters when this is feasible.

This design methodology addresses the conflicting goals that arise in the design of reusable software, and produces viable reusable parts for real-time embedded applications. A set of reusable part goals — flexibility, efficiency, reusability, ease of use, and protection against misuse — forms the basis of this method. *Flexibility* is the extent to which parts can be modified or tailored to the specific needs of an individual application. Although parts may be reusable, if they are not flexible and easily tailored, then the cost of using a part may be prohibitively large — it may be less expensive to develop a new part than to tailor or modify an existing part. It has been suggested that if a developer perceives the cost to develop from scratch is 70% or less of the cost to find and use a part, then he will develop the part from scratch (Reference [43]). The issue of *efficiency* is one which has long plagued software reuse efforts. The contention has been, and in many arenas still is, that parts which are reusable can never attain the required efficiency for use in real-time embedded applications. Parts must also be truly *reusable*, i.e., they must be suitable for use in a variety of applications, and they must be forward looking rather than backward looking if they are to actually see use. The parts must also be *easy to use* because difficulty of use increases the cost of reuse and may mean that the part will never be reused. *Protection against misuse* refers to providing the user with protection from choosing the wrong part for a given requirement or using a part inappropriately.

The CAMP design methodology meets the reusability goals and supports the development of parts which are well-tested and may be used off-the-shelf. These seemingly conflicting design goals are achieved by exploiting

Ada language features, such as generic units with default formal parameters (both objects and subprograms), strong data typing, derived types and subprograms, and subprogram overloading. This design approach was originally developed to address the needs of parts for missile operational flight software, but it is equally appropriate for RTE applications in other domains.

- Generic units: The primary facility Ada provides which promotes reusability is the generic unit.
- Strong data typing: The ability to strongly type data is a key feature in Ada. However, the use of strong data typing makes the design of generic packages and subprograms more complex; and the interaction of Ada typing rules with other Ada features, such as generic units, is non-trivial. This is why some software developers have developed Ada parts in a typeless fashion. Parts which are *typeless* are very prone to misuse. If parts are intended for long-term use, then the effort should be expended to build them in the best possible way.
- Generic object and subprogram parameter defaults: The use of strong data typing causes generic units to be more complex. Specifically, the generic packages and subprograms must now import many operations and functions which would otherwise be visible to them implicitly through the scoping rules of Ada. Ada allows this barrier to be overcome by the specification of defaults for generic object and subprogram parameters. The use of generic units for part design is further discussed in Section 5.3.

These conflicting requirements make development of viable RTE parts more difficult, and have led some to the conclusion that "since the generality needed for flexibility and portability will increase software overhead and, consequently, decrease the software's efficiency . . . it is very difficult to construct reusable missile software that is still viable." (Reference [4], p 105).

5.2 Alternative Design Methods

Six methods for the design of reusable parts were considered early in the CAMP program. Figure 5-1 illustrates these methods; they are summarized here and discussed in detail in Appendix III.

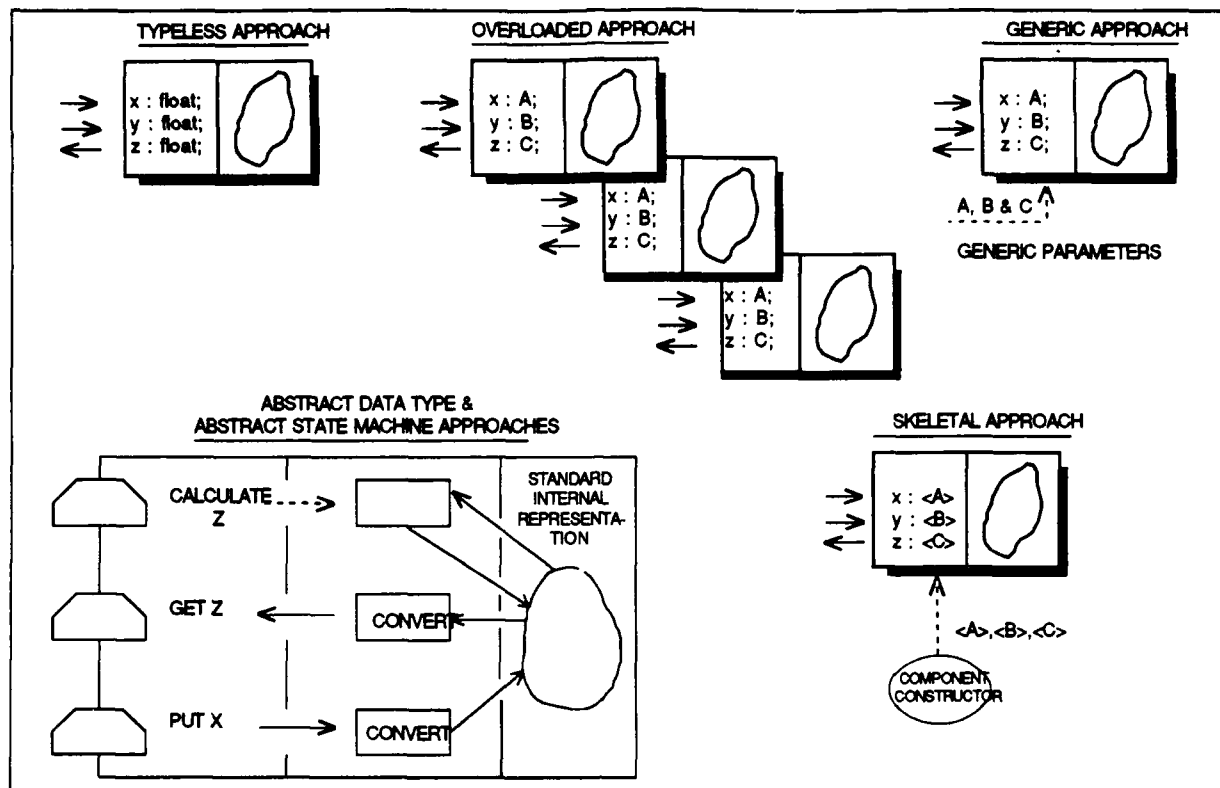


Figure 5-1: Reusable Parts Design Methods

- **Typeless:** All objects are declared as some basic type, e.g., float, integer, so there is no error checking on data types. Typeless parts are simple to design and use, but are not robust.
- **Overloaded:** There is a separate version of each part to allow for the different combinations of data types that the user might want; this is the method used in the Ada predefined packages *Standard* and *Calendar*. Parts are simple to design and use — the designer decides the combinations of data types that will be allowed for each part and explicitly declares parameter interfaces for these overloaded subprograms; the Ada disambiguation facility resolves calls so the user does not have to specify which version of a part he wants. Strict type checking ensures that actual and formal parameters match and that the values of the actual parameters fall within the ranges allowed by the type definitions. The major disadvantage is the large number of parts that have to be declared at the architectural level.
- **Generic:** This method uses Ada generic units to provide parts which are tailorable to user-defined types. The major disadvantage is that generally the user would need to supply a large number of generic parameters. This burden can be alleviated by the judicious use of default parameters, i.e., the designer provides for typical combinations of data types, and the instantiation will use these functions as actual subprogram parameters. This method provides flexibility while simplifying use. Parts can be set up for "tunnelling" of operators, i.e., types and operators are predefined, and the types can then be used to instantiate the generics and the operators will get pulled along. The major advantage of this approach is that it incorporates strong typing and is flexible.
- **Abstract State Machine:** This is a "black box" approach. Problems with data typing and mathematical operators are alleviated because these are all fixed by the designer. The user is provided with a high-level interface to the parts. There is no direct access to the data structures themselves — all access is through the operators provided in the interface. Efficiency problems can arise because of the need to perform data type conversions so that data will fit the internal representation. This could be overcome by creating multiple bodies that are efficient for a given situation, but would increase the cost of parts development. It can be effective when the data structure

cannot be established in the package specification (e.g., Kalman filter operations).

- **Abstract Data Type:** The data structure is declared in the private section in a package specification, thus the user is "stuck" with the data structure provided by the designer. To change it, he must change both the specification where the data structure is defined and the body. This method differs from the Abstract State Machine approach in that the interface consists of both the predefined set of operators and the data structure itself.
- **Skeletal Code:** This approach provides great flexibility — the user is provided with a template to be filled in. Problems may arise when more than one person is working a project — the template may be filled in differently by different engineers resulting in duplication of effort in producing an environment for such a part. There might also be a tendency to avoid strong data typing because of the overhead in creating functions and operators for strongly typed data. This approach would benefit from automated tools.

A thorough analysis of each method was conducted. Figure 5-2 summarizes the results of this analysis and identifies the advantages and disadvantages of the methods. The analysis focused on identifying the best methods for support of complex operators inside the body of parts and for simplifying the use of parts developed using the generic method. The generic method has the greatest potential for the design of "good" reusable parts. Prior Data Sciences, a Canadian firm specializing in the development of reusable, real-time software, has summarized the difficulties in developing reusable software based on generic units and of employing parts created using generic units.

- *"Library generic units are very difficult to write . . . the effort required to properly generalize them is usually significant."*
- *"Generic units are also difficult to use, especially when they have many interrelated parameters. The parameter matching rules can be very subtle." (Reference [25], p 70)*

Although generic units add complexity to the interfacing mechanism, the flexibility and protection against misuse which they afford weigh heavily in their favor. Generic units also provide flexibility for tailoring to the requirements of a specific application. The CAMP approach for the design of reusable parts utilizes the generic method.

5.3 Using the Generic Method to Design Parts

Effective use of generic units for the creation of reusable parts requires reconciliation between the complexity of the generic specification and the desired ease of use of the part. The description of the generic method in Appendix III discusses the conflict. In fact, the conflict entails the same trade-offs as those required to create reusable software: generality vs. efficiency and ease of use.

The Ada generic facility can be exploited in the development of reusable parts. Low-level parts can be designed as generic packages or subprograms. Higher-level parts can then be built from multiple levels of these generic units. The user supplies actual parameters to instantiate the generic parts and tailor them to his application. Multiple layers of generic units provide the part user with a broad choice in his selection of parts for an application: he may use low-level parts to implement low-level features of the individual objects of his design, or choose high-level parts which themselves serve as objects in his design. He can also replace low-level parts with his own code and still make use of the higher level parts, Figure 5-3 illustrates this.

A generic part uses its *generic formal parameters* for tailoring the part to a specific application. For example, the CAMP Compute_Earth_Relative_Horizontal_Velocities part (see Figure 5-4 for inputs, processing, and outputs) may be tailored for velocity type (feet per second, meters per second, miles per hour, knots) and for angle

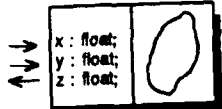
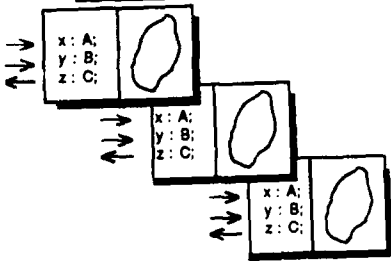
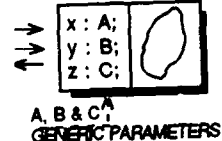
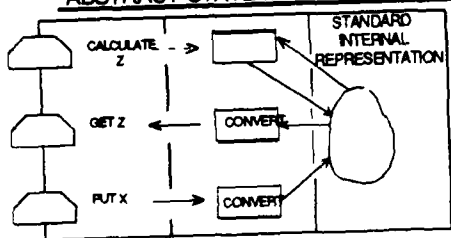
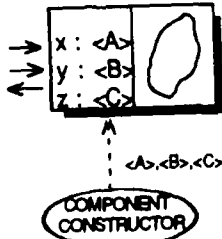
METHODS	ADVANTAGES	DISADVANTAGES
<u>TYPELESS APPROACH</u> 	No need to define new operators Simple interface	No protection against misuse
<u>OVERLOADED APPROACH</u> 	All operators provided Flexible Protected against misuse	Too many parts Maintenance nightmare
<u>GENERIC APPROACH</u> 	Flexible for new data types Protected against misuse	User must provide all operators Complex interface (generics)
<u>ABSTRACT DATA TYPE & ABSTRACT STATE MACHINE APPROACHES</u> 	Protected against misuse	"All or nothing at all" approach Inefficient Complex interface (instantiate entire package even if only one subprogram needed)
<u>SKELETAL APPROACH</u> 	Flexible	User must create data environment User must build in own protection Requires automated assistance for productive use

Figure 5-2: Comparison of the Six Reusable Parts Design Methods

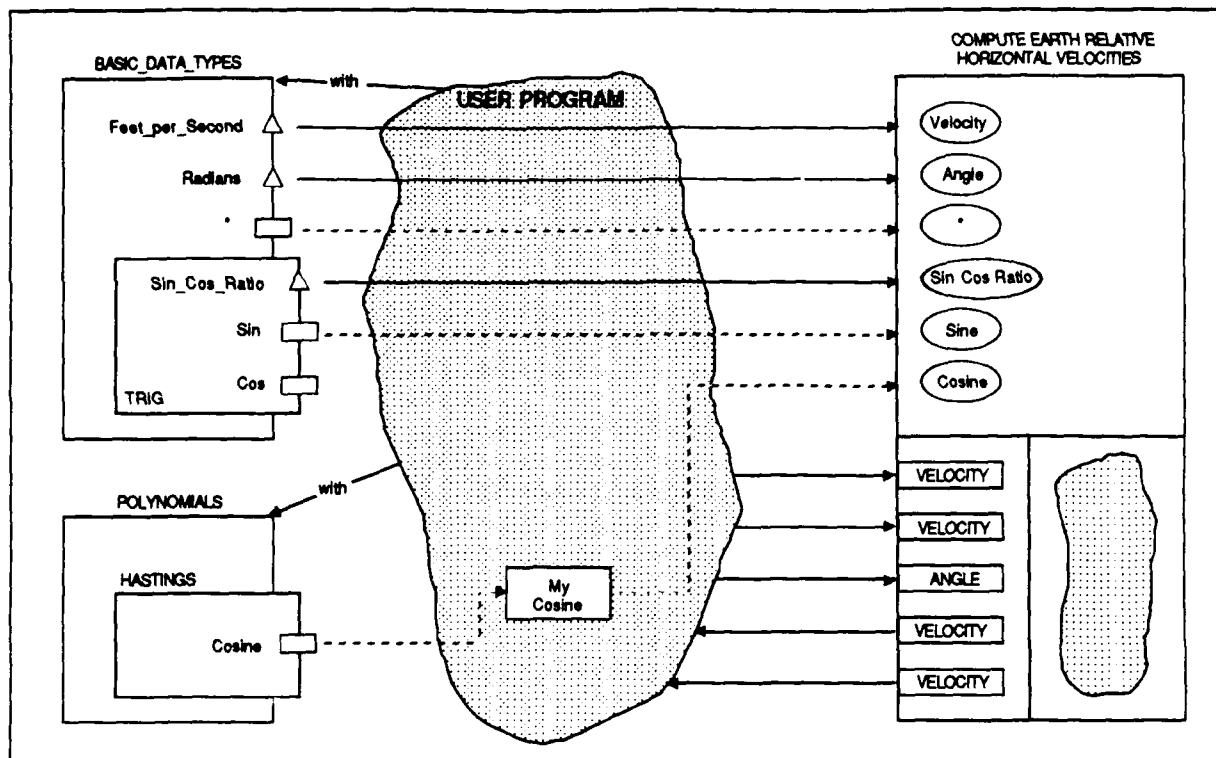


Figure 5-3: User Substitution of Low-Level Parts

type (radians, degrees, semicircles). In addition, the tailoring can extend to the return type of a sine or cosine operator.

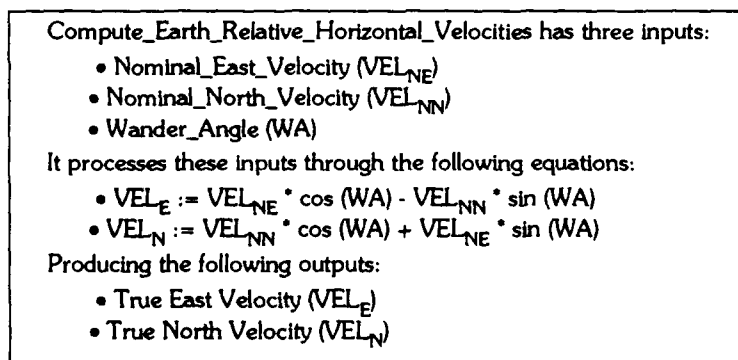


Figure 5-4: Example of Use of Ada Generics in Design of Reusable Parts

In order to complete the tailoring, the part must also allow tailoring for operators essential for the enforcement of strong data typing. Generally, operators are merely overloads of predefined operators ("+", "-", "*", "/"). For more complex operations, the user must create his own subprograms, such as sine and cosine, filters, matrix operations, etc. For these user-created operations, there are no language-defined constructs and the generic specification cannot fully describe the required operation. Only the part's documentation and the user's familiarity with the part's internal design can support creation of actual parameters to match the formal generic. Those features of a part which are truly common between applications, and are captured in the body of the part, include:

- The generic data types
- The sequence of operations
- Data types and operations not parameterized through the generic

Figure 5-5 shows the use of generic plus non-generic features of a part body. The formal data types and Sin and Cos operations are generic and, hence, tailorable. The multiplication operator is also generic. The subtraction and addition operations are not generic. Of course, the sequence of operations to calculate the output velocities is also non-generic.

```

procedure Compute_Earth_Relative_Horizontal_Velocities
  (Nominal_East_Velocity : in    Velocities;
   Nominal_North_Velocity : in   Velocities;
   Wander_Angle         : in    Angles;
   East_Velocity         : out  Velocities;
   North_Velocity        : out  Velocities) is

  Sin_W_A : Sin_Cos_Ratio;
  Cos_W_A : Sin_Cos_Ratio;

begin

  Sin_W_A := Sin (Current_Wander_Angle);
  Cos_W_A := Cos (Current_Wander_Angle);

  East_Velocity := Nominal_East_Velocity * Cos_W_A -
                    Nominal_North_Velocity * Sin_W_A;

  North_Velocity := Nominal_North_Velocity * Cos_W_A +
                    Nominal_East_Velocity * Sin_W_A;

end Compute_Earth_Relative_Horizontal_Velocities;

```

Figure 5-5: Commonality Captured in the Generic Part Body

5.4 Interaction Between Parts

An important aspect of the CAMP design approach is that parts can be designed to build on other parts, work together, and facilitate use of other parts. Figure 5-6 shows how these relationships come into play when developing a small portion of a north-pointing navigation system (Compute_Coriolis_Acceleration, Radius_of_Curvature, and Latitude_Integration). In order to instantiate these parts for use in the navigation system, the following must occur:

1. Ten packages must be compiled into the user's library. The user himself requires six of these (indicated by the arrows going into the user application). These six require an additional four.
2. Before instantiating the navigation parts the user must do the following:
 - Instantiate four versions of the square root package (GPMath.Square_Root) using data types and operators supplied by the basic data types (BDT) package.
 - Instantiate four versions of the vector operations package (CVMA.Vector_Opns) using data types and operators supplied by BDT and the square root functions contained in the packages previously instantiated by the user.
 - Instantiate a cross product function using scalar data types and operations supplied by BDT, along with vector data types and operations obtained from three separate instantiations of CVMA.Vector_Opns.

3. The three navigation parts can then be instantiated using:

- Scalar data types and operators supplied by BDT.
- Scalar data types and trigonometric functions supplied by an instantiation of the standard trig package contained in BDT (BDT.Trig).
- Vector types and operations supplied by the four instantiations of CVMA.Vector_Opns.
- Data constants supplied by the WGS72 ellipsoid metric data package (WGS72) and the WGS72 ellipsoid unitless data package (WGS72U).
- User-defined data types and objects.

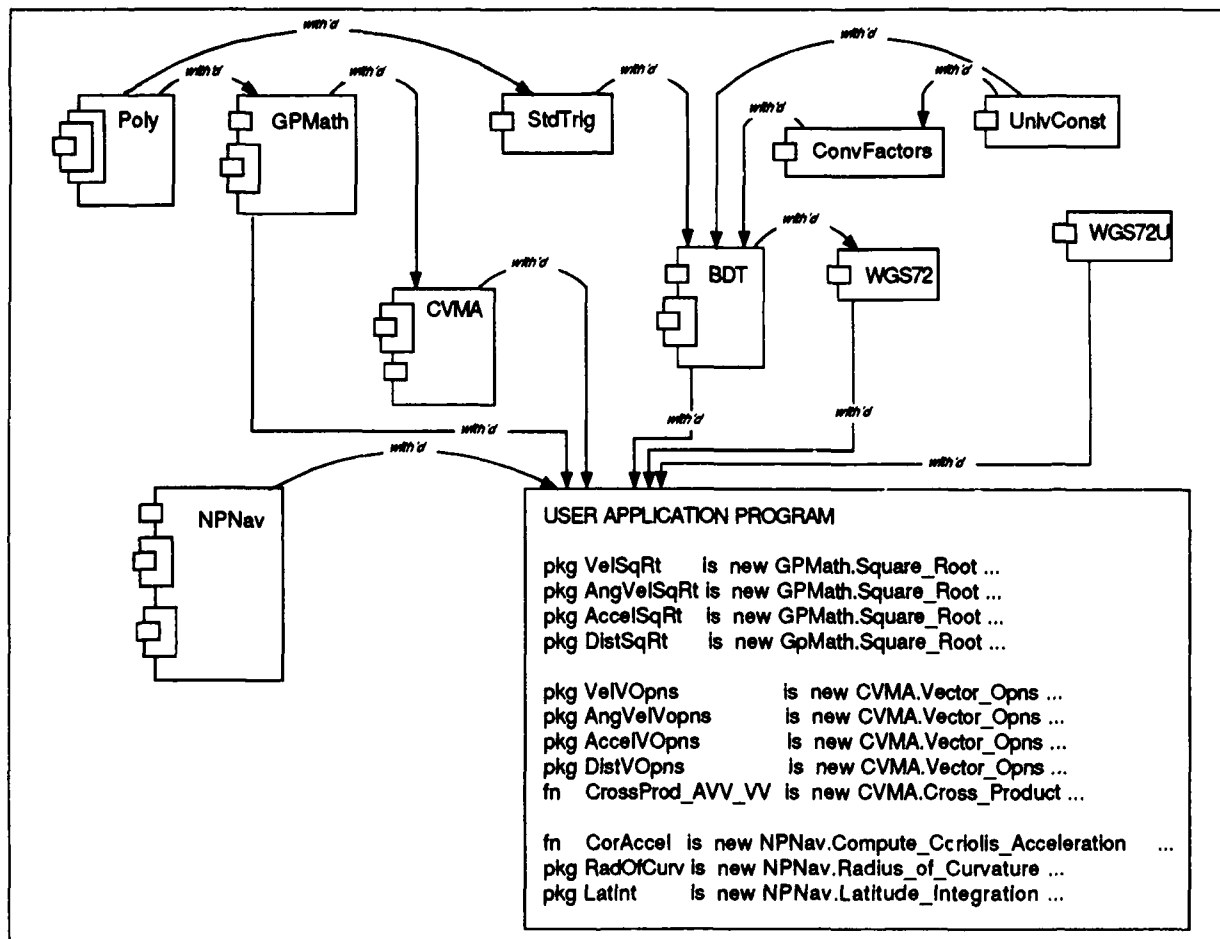


Figure 5-6: Assembling a North-Pointing Navigation System

5.4.1 Parts Build on Other Parts

Software parts should be designed to facilitate the composition of more complex parts from simpler parts. For example, consider the CAMP Polynomials, Standard_Trig, and Basic_Data_Types parts illustrated in Figure 5-7. The Polynomials part lies at the bottom of the build and provides an extensive set of polynomial solutions to various transcendental functions. The generic unit Standard_Trig forms the second layer by exporting trigonometric data types and operations. Standard_Trig uses the Polynomials package to obtain the required polynomial solutions to its exported transcendental functions. The Basic_Data_Types part provides the final layer. In addition to providing a set of data types and operations typical of a navigation implementation, Basic_Data_Types instantiates the Standard_Trig package. In this example, a user need only import Basic_Data_

Types in order to obtain a full set of navigation data types (such as various forms of distances, velocities, accelerations, etc.), operators upon these types, trigonometric data types (such as radians, degrees, etc.), and a full set of trigonometric functions.

This design approach offers several advantages, and is an important feature of the CAMP design methodology. Some of the characteristics of this approach include the following:

- Minimal functionality is added from one step to the next.
- Users of the higher level packages, such as *Basic_Data_Types*, frequently will not need to reference the lower level packages, such as *Polynomials*.
- Combining the parts saves work for the user.

It is one means of providing flexibility — the user can replace CAMP parts at any level with his own versions.

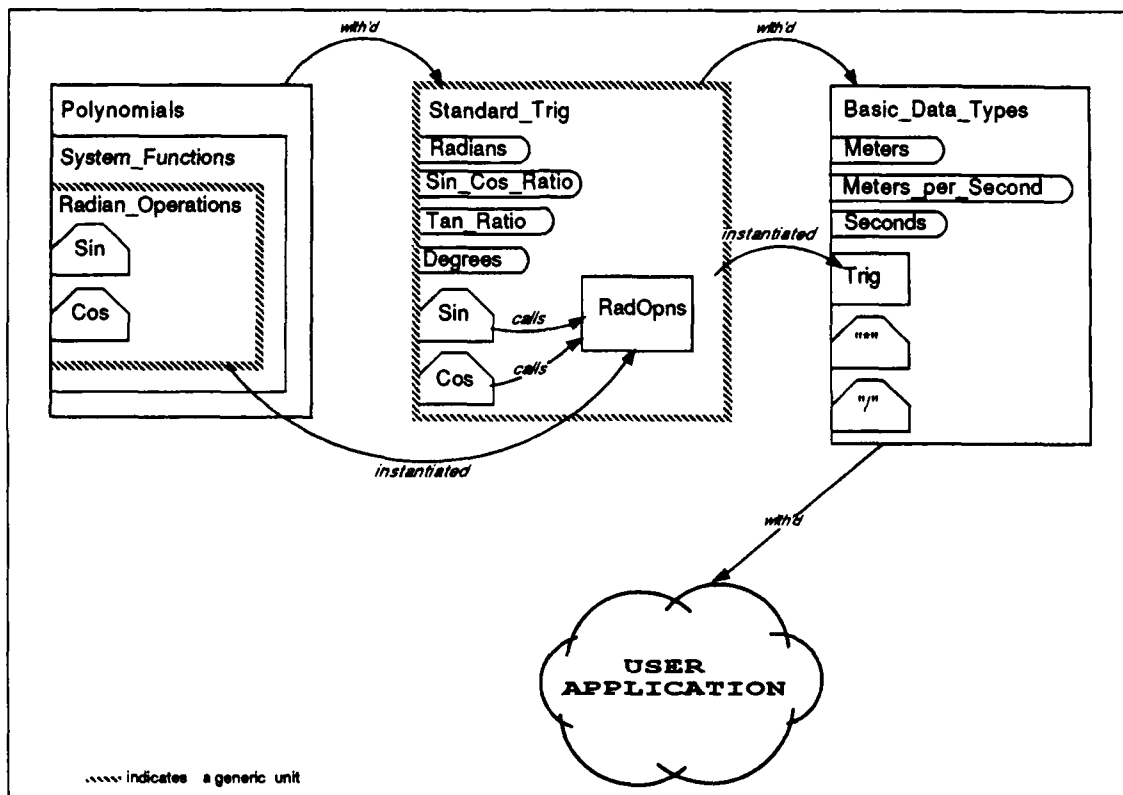


Figure 5-7: Some Parts Build on Other Parts

5.4.2 Parts Work Together

Parts can also be designed to work together, using low-level parts to support more complex operations. This design approach differs from the approach previously discussed in that functionality is added with each step and the lower parts are frequently required directly by the user. For example, consider the *Geometric_Operations* and *Waypoint_Steering* parts shown in Figure 5-8. The *Waypoint_Steering* part exports the *Steering_Vector_Operations* package which handles the initialization and updating of waypoint steering vectors. In order to perform its operations, the *Steering_Vector_Operations* package instantiates two subprograms from the *Geometric_Operations* package which are designed to calculate unit radial vectors, unit normal vectors, and course segments. This design methodology has several benefits:

- Because the lower level parts are not placed in the body of the higher level package, they are also available to the user. In this case, the geometric operations are not put in the body of the Waypoint_Steering package, so they can be used for other applications as well.
- Maintainability is improved because the code for the lower level part (in this case Geometric_Operations) is not duplicated within the higher level part (in this case the Waypoint_Steering package).
- The parts are easier to use because the instantiations of the lower level parts (e.g., Geometric_Operations) are performed within the higher level parts (e.g., Steering_Vector_Operations LLCSC) instead of bringing them in as generic subroutines. This saves the user the effort of locating and instantiating additional parts.

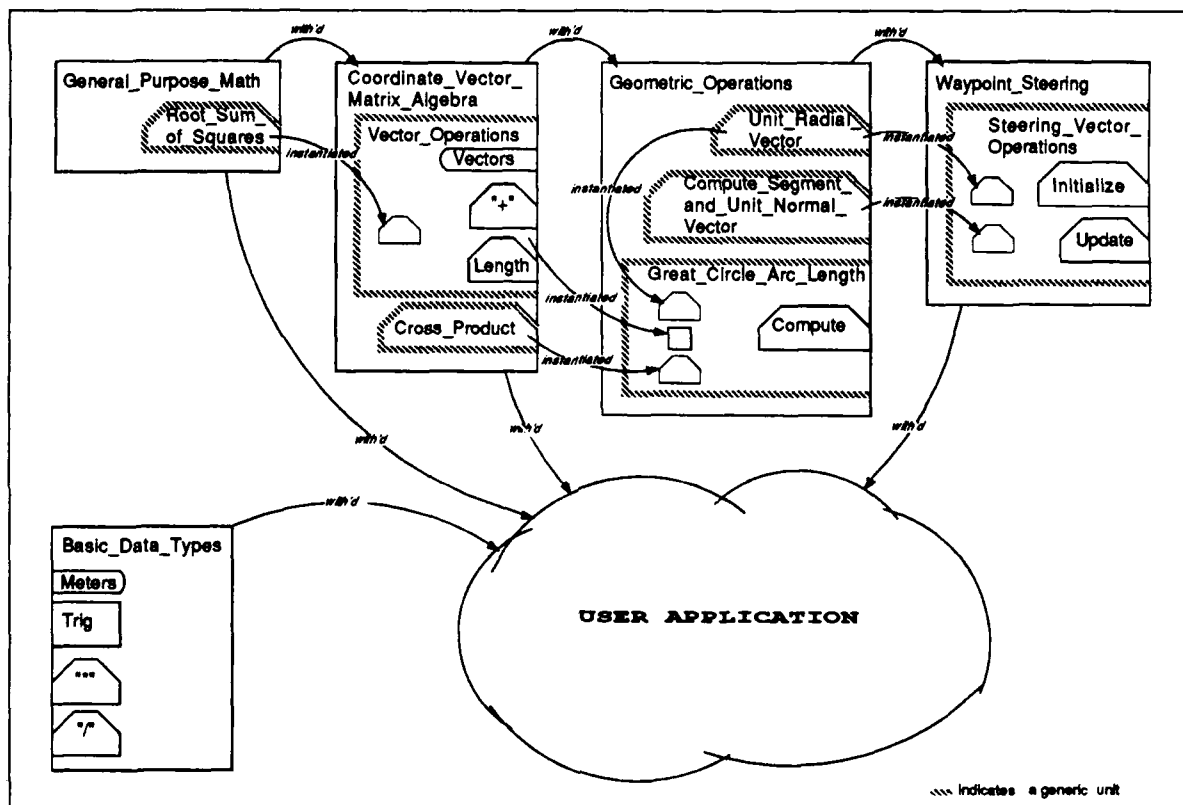


Figure 5-8: Parts Work Together

5.4.3 Parts Facilitate Use of Other Parts

Parts can be designed to facilitate the use of other parts by providing the requisite generic actual parameters. For example, consider Figure 5-9. In order to instantiate the generic **Compute_Segment_and_Unit_Normal_Vector** procedure, the user need only define a discrete type for *Indices*. The remaining scalar types can be obtained from the **Basic_Data_Types** package, along with the multiplication and division operators; the vector type and operations on that type (i.e., **Vector_Length** and **Cross_Product**) can be obtained by instantiating the **Vector_Operations** package in the **Coordinate_Vector_Matrix_Algebra** CSC; and a value for the radius of the Earth can be found in the **WGS72_Ellipsoid_Engineering_Data** CSC. This kind of support can be found in most of the CAMP parts, and is an example of how the *ease of use* goal can be met. Parts developers should anticipate the need for supporting parts and provide them in the parts set.

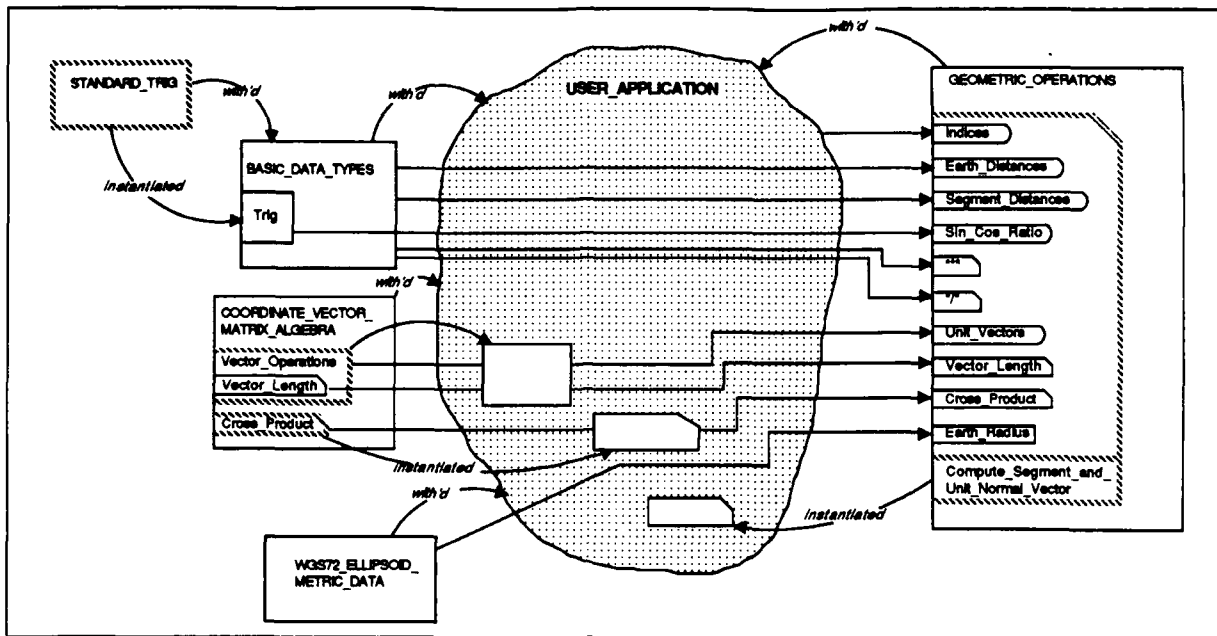


Figure 5-9: Parts Facilitate Use of Other Parts

5.5 Design Considerations

A primary consideration in the design approach presented here was how to provide low-level operations, such as linear algebra and transcendental functions, to more complex routines. There were several options:

- In-line the required operations directly into the higher level routine: This option was rejected because it would cause the parts to become excessively large. Also, in-lining would increase testing time, and has the potential for creating a maintenance nightmare.
- Place the required code in subroutines located in package bodies: This option, while an improvement over in-lining, would also increase the size of the parts, lengthen testing time, and increase maintenance difficulties.
- Instantiate a required operation from another part: In a few cases this option may be desirable. This method may be desirable if: (1) only one method exists for implementing the required operation; or (2) the instantiating part is a very high-level part, such as a Kalman filter update package, designed to provide one possible solution to a problem by bringing together one possible combination of lower level parts. It is also acceptable if the required operation is a very basic one, such as a trigonometric function, and it is not possible to know ahead of time which algorithm will provide optimal performance.
- Bring the required operations in via generic parameters: This option is the one of choice in the vast majority of cases.

The use of generic formal subprograms to import required operations is an important design feature. It has the advantage of providing great flexibility to the user by allowing parts to be developed to supply low level operations or allowing the user to define his own, as shown in the following examples.

• Example I:

In this example, assume the user wants to instantiate both of the parts contained in the Geometric_Operations CSC shown in Figure 5-10. Each part requires a sine/cosine procedure as a generic parameter. If the user has imported the Basic_Data_Types (BDT) package, he already has access to the sine/cosine procedure provided indirectly by BDT's instantiation of Standard_Trig (Trig). If this procedure is satisfactory, he does not need to specify it in his instan-

tiation since the BDT version will be selected by default. If, however, the user's calculations require more accuracy or speed, he may construct a different sine/cosine procedure by building one from the over 25 sine functions provided by the Polynomials CSC or by writing his own. This new sine/cosine procedure may then be used in one of the following ways:

- If he wants to use this new procedure throughout his application for all sine/cosine calculations, the procedure can be specified in such a way as to hide the sine function contained in BDT.Trig. He can let the generic actual subroutines default to this new procedure. This is illustrated in Figure 5-11.
- If the newly created sine/cosine procedure is to be used only for certain calculations, it can be designed in such a way as to not hide the one contained in BDT.Trig. In this case, the special procedure would have to be explicitly specified in instantiations where it was to be used. Using this method, it is possible for the user to create multiple sine/cosine procedures — a fast one, a highly accurate one, and a general purpose one — to meet his needs. This is illustrated in Figure 5-12.

• Example II:

If the user wants to construct a Kalman filter that incorporates a complicated-H matrix, he can use the `Kalman_Filter_Data_Types` package and all the data types it provides, and all generic formal subroutines required by instantiations of any of the parts contained in the `Kalman_Filter_Common_Parts` and `Kalman_Filter_Complicated_H_Part` CSCs will properly default. If however, he wants to reduce storage by not using full storage matrices, he can define his own data types and operations, and still use the Kalman filter parts without making any modifications to the parts themselves.

```

generic
  type Angle      is digits <>;
  type Trig_Ratio is digits <>;
package Standard_Trig is

  type Radians      is new Angle;
  type Sin_Cos_Ratio is new Trig_Ratio range -1.0..1.0;

  procedure Sin_Cos (Input      : in    Radians;
                     Sin_Result : out Sin_Cos_Ratio;
                     Cos_Result : out Sin_Cos_Ratio);

end Standard_Trig;

-----

with SYSTEM;
with Standard_Trig;
package Basic_Data_Types is

  type Real      is digits SYSTEM.MAX_DIGITS;
  type Meters     is digits SYSTEM.MAX_DIGITS;

  package Trig is new Standard_Trig
    (Angle      => Real,
     Trig_Ratio => Real);

  type Earth_Position_Radians is new Trig.Radians;

  function "" (Left  : Meters;
               Right : Trig.Sin_Cos_Ratio)
    return Meters;

end Basic_Data_Types;

package Geometric_Operations is

  generic
    type Indices      is (<>);
    type Earth_Positions is digits <>;
    type Sin_Cos_Ratio is digits <>;
    type Unit_Vectors is array (Indices)
      of Sin_Cos_Ratio;

    X : in Indices      := Indices'FIRST;
    Y : in Indices      := Indices'SUCC(X);
    Z : in Indices      := Indices'LAST;
    with procedure Sin_Cos
      (Input      : in    Earth_Positions;
       Sine       : out Sin_Cos_Ratio;
       Cosine     : out Sin_Cos_Ratio)
      is <>;

  function Unit_Radial_Vector
    (Lat_of_Point : Earth_Positions;
     Long_of_Point : Earth_Positions)
    return Unit_Vectors;

  generic
    type Earth_Distances is digits <>;
    type Earth_Positions is digits <>;
    type Segment_Distances is digits <>;
    type Sin_Cos_Ratio is digits <>;
    Earth_Radius      : in Earth_Distances;
    with function "" (Left  : Earth_Distances;
                     Right : Sin_Cos_Ratio)
      return Segment_Distances is <>;
    with function Sqrt (Input : Sin_Cos_Ratio)
      return Sin_Cos_Ratio is <>;
    with procedure Sin_Cos
      (Input      : in    Earth_Positions;
       Sine       : out Sin_Cos_Ratio;
       Cosine     : out Sin_Cos_Ratio)
      is <>;

  package Great_Circle_Arc_Length is

    function Compute
      (Latitude_A : Earth_Positions;
       Latitude_B : Earth_Positions;
       Longitude_A : Earth_Positions;
       Longitude_B : Earth_Positions)
      return Segment_Distances;

  end Great_Circle_Arc_Length;

end Geometric_Operations;

```

Figure 5-10: Required Operations Obtained Through Use of Generic Formal Parameters

```

with Basic_Data_Types;
with Geometric_Operations;
with WGS72_Ellipsoid_Metric_Data;
procedure User_Application is

    use Basic_Data_Types;

    package BDT renames Basic_Data_Types;
    package GEO renames Geometric_Operations;
    package WGS72 renames WGS72_Ellipsoid_Metric_Data;

    type Indices is (X, Y, Z);

    type Unit_Vectors is array (Indices) of BDT.Trig.Sin_Cos_Ratio;

    function Sqrt (Input : BDT.Trig.Sin_Cos_Ratio)
        return BDT.Trig.Sin_Cos_Ratio;

    - --new Sin_Cos procedure to override that provided by BDT.Trig
    procedure Sin_Cos (Input : in      BDT.Earth_Position_Radians;
                      Sine   : out    BDT.Trig.Sin_Cos_Ratio;
                      Cosine : out    BDT.Trig.Sin_Cos_Ratio);

    function U_Radial_Vector is new GEO.Unit_Radial_Vector
        (Indices      => Indices,
         Earth_Positions => BDT.Earth_Position_Radians,
         Sin_Cos_Ratio  => BDT.Trig.Sin_Cos_Ratio,
         Unit_Vectors   => Unit_Vectors);

    - --Sin_Cos defaults to new Sin_Cos procedure

    package Great_Circle_Arc_Len is new GEO.Great_Circle_Arc_Length
        (Earth_Distances  => BDT.Meters,
         Earth_Positions  => BDT.Earth_Position_Radians,
         Segment_Distances => BDT.Meters,
         Earth_Radius     => WGS72.Earth_Equatorial_Radius,
         Sin_Cos_Ratio    => BDT.Trig.Sin_Cos_Ratio);

    - --Sin_Cos defaults to new Sin_Cos procedure

begin
    ...
end User_Application;

```

Figure 5-11: Sample Instantiations of Geometric_Operations Parts Using Default Routines

```

with Basic_Data_Types;
with Geometric_Operations;
with WGS72_Ellipsoid_Metric_Data;
procedure User_Application is

    use Basic_Data_Types;

    package BDT    renames Basic_Data_Types;
    package GEO    renames Geometric_Operations;
    package WGS72  renames WGS72_Ellipsoid_Metric_Data;

    type Indices is (X, Y, Z);

    type Unit_Vectors is array (Indices) of BDT.Trig.Sin_Cos_Ratio;

    function Sqrt (Input : BDT.Trig.Sin_Cos_Ratio)
        return BDT.Trig.Sin_Cos_Ratio;

    -- additional Sin_Cos procedure
    procedure Fast_Sin_Cos (Input : in    BDT.Earth_Position_Radians;
                           Sine   : out BDT.Trig.Sin_Cos_Ratio;
                           Cosine : out BDT.Trig.Sin_Cos_Ratio);

    function U_Radial_Vector is new GEO.Unit_Radial_Vector
        (Indices      => Indices,
         Earth_Positions => BDT.Earth_Position_Radians,
         Sin_Cos_Ratio  => BDT.Trig.Sin_Cos_Ratio,
         Unit_Vectors   => Unit_Vectors,
         Sin_Cos        => Fast_Sin_Cos);

    package Great_Circle_Arc_Len is new GEO.Great_Circle_Arc_Length
        (Earth_Distances  => BDT.Meters,
         Earth_Positions  => BDT.Earth_Position_Radians,
         Segment_Distances => BDT.Meters,
         Earth_Radius     => WGS72.Earth_Equatorial_Radius,
         Sin_Cos_Ratio    => BDT.Trig.Sin_Cos_Ratio);

    -- Sin_Cos defaults to BDT.Trig.Sin_Cos

begin
    ...
end User_Application;

```

Figure 5-12: Sample Instantiations of Geometric_Operations Parts Using Specialized Sin_Cos Procedure

5.6 Design Guidelines

The following paragraphs contain guidelines which should be considered when developing reusable Ada components. All of these guidelines arise from the fact that reusable parts must be tailorable and flexible, and that it is impossible for the parts developer to anticipate all future uses of the parts. All design and implementation decisions should address the questions, "What might the user want to do with this part?" and "What would be legitimate and safe for him to do with the part?".

The guidelines address how various design decisions can positively and negatively impact the user. For example, one of the guidelines shows how decisions regarding strength of data typing can affect the effort required to integrate a part with the user's application. Some guidelines are given for recognizing areas appropriate for project-specific reusable parts, and some suggestions are offered to make it easier for the user to incorporate the reusable code into his application.

Guideline #1: Reusable parts should be designed to support, but not force, strong data typing.

One of the benefits of Ada is its strong data typing facility. Strong data typing allows many errors to be caught during compilation rather than during testing or execution. Reusable parts should be designed to permit and promote strong data typing, but they should also be designed to allow the user to make the final decision regarding the strength of data typing.

Consider the examples in Table 5-1 which show the package specifications for four implementations of a trigonometric package. The first two examples in this table, TrigOpns1a and TrigOpns1b, show a weakly data typed package using the same data type for both input and output values. TrigOpns1b is an improvement over TrigOpns1a because it allows the user to define the data type being used. The use of either of these packages in a strongly typed application would require the user to either perform type conversions with each call or write an interface package that performs these conversions for him. The last two examples in Table 5-1, TrigOpns2a and TrigOpns2b, contain strongly typed packages. TrigOpns2a not only allows strong typing, but attempts to force strong data typing on the user by defining the data types in the package specification. This forces the user to either use these data types, perform type conversions with every call, or write an interface package. TrigOpns2b has the advantage that it allows the user to determine the strength of the data typing because all of the data types are imported. A more detailed discussion of these implementations can be found in the case study in Paragraph 5.7.2.

Table 5-1: Data Typing Comparisons

CODE EXAMPLE	COMMENTS
Weak data typing that gives no control to the user: <pre> package TrigOps1a is -- --expects inputs in units of radians function Sin (Angle : FLOAT) return FLOAT; function Cos (Angle : FLOAT) return FLOAT; function Tan (Angle : FLOAT) return FLOAT; end TrigOps1a; </pre>	User has no control over the data types in this package because the Ada predefined data type 'FLOAT' is used. If the user's application does not use the data type 'FLOAT', he will be forced to perform type conversions with each call or write an interface package to go between his code and this package.
Weak data typing that allows user to define digits of precision: <pre> generic type Values is digits <>; package TrigOps1b is -- --expects inputs in units of radians function Sin (Angle : Values) return Values; function Cos (Angle : Values) return Values; function Tan (Angle : Values) return Values; end TrigOps1b; </pre>	User now has control over the digits of precision of the data type used by this package, but inputs and outputs are still of the same type. If the user's application employs stronger data typing, he will be forced to perform type conversions with each call or write an interface package to go between his code and this package.
Strong data typing that is forced on the user: <pre> generic type Input Values is digits <>; type Output Values is digits <>; package TrigOps2a is type Radians is new Input Values; type Degrees is new Input Values; type SC_Ratios is new Output Values; type T_Ratios is new Output Values; function Sin (Angle : Radians) return SC_Ratios; function Cos (Angle : Radians) return SC_Ratios; function Tan (Angle : Radians) return T_Ratios; function Sin (Angle : Degrees) return SC_Ratios; function Cos (Angle : Degrees) return SC_Ratios; function Tan (Angle : Degrees) return T_Ratios; end TrigOps2a; </pre>	User now has control over the digits of precision, and the inputs and outputs are of different data types. He is, however, forced to use not only strong data typing, but also the data types defined by this package. If he doesn't, he will have to perform type conversions with each call or write an interface package to go between his code and this package.
Facilitates strong data typing, but allows user to decide how strong: <pre> generic type Radians is digits <>; type Degrees is digits <>; type SC_Ratios is digits <>; type T_Ratios is digits <>; package TTrigOps2b is function Sin (Angle : Radians) return SC_Ratios; function Cos (Angle : Radians) return SC_Ratios; function Tan (Angle : Radians) return T_Ratios; function Sin (Angle : Degrees) return SC_Ratios; function Cos (Angle : Degrees) return SC_Ratios; function Tan (Angle : Degrees) return T_Ratios; end TrigOps2b; </pre>	User now has control over the digits of precision and the strength of data typing. He can use strong typing by specifying a different data type for each of the generic formal parameters. Alternatively, he can use weak data typing by specifying the same data type for Radians, SC_Ratios, and T_Ratios if his application required radian calculations or by specifying the same data type for Degrees, SC_Ratios, and T_Ratios if his application required degree calculations. (Radians and Degrees require separate data types to disambiguate the overloaded functions.)

When considering data typing in the design of reusable Ada components, the question arises, "How strong is strong enough?". At a minimum, parts should be designed to support different units of measure, e.g., separate data types should be provided for distances, velocities, accelerations, sine/cosine ratios, pressure, etc. Consider the example shown in Figure 5-13 which contains the specification for a generic function to compute the distance to the current waypoint. It imports 4 data types, 1 object, and 2 subprograms via its generic formal parameters.

```

generic
  type Unit_Vectors      is private;
  type Sin_Cos_Ratio     is digits <>;
  type Distances         is digits <>;
  Earth_Radius           : in Distances;
  with Function Dot_Product (Left : Unit_Vectors;
                             Right : Unit_Vectors)
    return Sin_Cos_Ratio is <>;
  with function "*" (Left : Sin_Cos_Ratio;
                    Right : Distances)
    return Distances is <>;
  function Distance_to_Current_Waypoint
    (Unit_Radial_M : Unit_Vectors;
     Unit_Tangent_B : Unit_Vectors) return Distances;

  function Distance_to_Current_Waypoint
    (Unit_Radial_M : Unit_Vectors;
     Unit_Tangent_B : Unit_Vectors) return Distances is
    Dot_Prod_Result : Sin_Cos_Ratio;
    Answer           : Distances;
  begin
    Dot_Prod_Result := Dot_Product (Left => Unit_Radial_M,
                                    Right => Unit_Tangent_B);
    Answer := Dot_Prod_Result * Earth_Radius;
  end Distance_to_Current_Waypoint;

```

Figure 5-13: Distance_to_Current_Waypoint Function (Version 1)

While it may not be obvious from looking at this specification, it makes an assumption regarding data typing — it assumes that the precisions for all objects of a particular unit of measure are the same. In particular, it assumes that the distance returned by the function is of the same data type as the distance used to define the radius of the Earth. This assumption is not always true and can be removed. A revised specification is shown in Figure 5-14.

```

generic
  type Unit_Vectors      is private;
  type Sin_Cos_Ratio     is digits <>;
  type Earth_Distances   is digits <>;
  type Segment_Distances is digits <>;
  Earth_Radius           : in Earth_Distances;
  with Function Dot_Product (Left : Unit_Vectors;
                             Right : Unit_Vectors)
    return Sin_Cos_Ratio is <>;
  with function "*" (Left : Sin_Cos_Ratio;
                    Right : Earth_Distances)
    return Segment_Distances is <>;
  function Distance_to_Current_Waypoint
    (Unit_Radial_M : Unit_Vectors;
     Unit_Tangent_B : Unit_Vectors) return Segment_Distances;

```

Figure 5-14: Distance_to_Current_Waypoint Function (Version 2)

It is necessary to decide just how flexible a part should be with respect to data typing. Table 5-2 shows various generic specifications for a simple dot product function. The complexity of the specifications range from a very simple version which assumes that the input and output data types are the same, to the most complex one which allows input, output, and intermediate values to be of different data types. As can be seen from these and earlier examples, the complexity of developing and using the code increases as the number of assumptions decreases.

Table 5-2: Various Degrees of Data Typing

CODE EXAMPLE	COMMENTS
Weak data typing: <pre> generic type Elements is digits <>; type Indices is (<>); type Vectors is array (Indices) of Elements; function Dot_Product (Left : Vectors; Right : Vectors) return Elements; function Dot_Product (Left : Vectors; Right : Vectors) return Elements is Answer : Elements := 0.0; begin for I in Indices loop Answer := Left(I) * Right(I) + Answer; end loop; return Answer; end Dot_Product; </pre>	This shows a very basic dot product function. It assumes the data types of both the input vectors are the same and the data type of the return value is the same as the vector elements.
Mild data typing: <pre> generic type Input_Elements is digits <>; type Output_Elements is digits <>; type Indices is (<>); type Vectors is array (Indices) of Input_Elements; with function "*" (Left : Input_Elements; Right : Input_Elements) return Output_Elements is <>; function Dot_Product (Left : Vectors; Right : Vectors) return Output_Elements; </pre>	This function no longer assumes that the input and output values have the same data type. It does, however, continue to assume that the input vectors are of the same data type.
Strong data typing: <pre> generic type Left_Elements is digits <>; type Right_Elements is digits <>; type Output_Elements is digits <>; type Left_Indices is (<>); type Right_Indices is (<>); type Left_Vectors is array (Left_Indices) of Left_Elements; type Right_Vectors is array (Right_Indices) of Right_Elements; with function "*" (Left : Left_Elements; Right : Right_Elements) return Output_Elements is <>; function Dot_Product (Left : Left_Vectors; Right : Right_Vectors) return Output_Elements; </pre>	This function no longer assumes that all the vectors are of the same data type. It does, however, assume that the intermediate values are of the same data type as the output value.
Stronger data typing: <pre> generic type Left_Elements is digits <>; type Right_Elements is digits <>; type Intermediate_Values is digits <>; type Output_Elements is digits <>; type Left_Indices is (<>); type Right_Indices is (<>); type Left_Vectors is array (Left_Indices) of Left_Elements; type Right_Vectors is array (Right_Indices) of Right_Elements; with function "*" (Left : Left_Elements; Right : Right_Elements) return Intermediate_Values is <>; with function "+" (Left : Intermediate_Values; Right : Intermediate_Values) return Output_Elements is <>; function Dot_Product (Left : Left_Vectors; Right : Right_Vectors) return Output_Elements; </pre>	This function no longer assumes that the intermediate values are of the same data type as the output value. It does, however, assume that all the intermediate values are of the same data type. Without knowing the size of the vectors ahead of time, it would not be possible to provide separate intermediate data types.

Table 5-2 should not be taken as a recommendation to always provide maximum flexibility, but rather to consider the amount of flexibility required in a part. At a minimum, a part should provide different data types for each unit of measure used by the subroutine. If the units of measure of intermediate types are different from the input and output types, they should also be provided so that the generic formal subroutines can more closely reflect those being used by the application developer. For example, if inputs are in units of accelerations and time, outputs are in units of distances, and intermediate values are in units of velocities, data types and operators should be imported to reflect the actual operation of the function in order to give the greatest amount of control to the user, even if this means importing data types not required for the input and output types.

Guideline #2: Bundling can facilitate locating and retrieving parts, but the bundling scheme should be chosen carefully.

Parts bundling is the process of placing multiple, related code parts in the same package. For example, the CAMP parts set includes a Wander_Azimuth_Navigation package that contains all operations specific to wander azimuth navigation, a North_Point_Navigation package that contains all operations specific to north pointing navigation, a Common_Navigation package that contains all operations which are common to north pointing and wander azimuth navigation, a Geometric_Operations package that provides the geometric calculations required for guidance, a Clock_Handler package which contains the operations required to maintain a system clock, etc. Bundling aids the user in locating relevant parts: by placing all parts dealing with a specific area in a single package, the user simply has to locate the single package dealing with the area of interest and he is assured that either the part he is looking for is in that package or it doesn't exist in that parts set.

Parts may be bundled according to types or classes of operations or objects. For the purposes of this discussion, a type is considered more restrictive than a class. For example, types of operations would include north-pointing navigation, wander azimuth navigation, and common navigation operations; the class of operations for this example would be navigation. Examples of types of objects would be stacks, queues, and buffers; the corresponding class of objects would be abstract data structures. Bundles do not necessarily consist of homogeneous types of parts; e.g., they can consist of a high-level type of operation, such as navigation or Kalman filtering, and all of the supporting parts needed to use that operation.

Figures 5-15 and 5-16 show a set of operations on abstract data structures packaged according to each of these bundling schemes. The first packaging structure, shown in Figure 5-15, has bundled the operations by class — the class being abstract data structures. This bundling scheme results in the creation of a single top-level package which exports six separate subpackages. Each of these subpackages defines a different type of object (i.e., stacks, queues, etc.) and operations on that object. Figure 5-16 shows the same set of operations bundled according to object type rather than object class. The result is six separate packages, each of which defines its own object type and operations on that type.

Table 5-3 summarizes the advantages and disadvantages of bundling by class and type.

ABSTRACT DATA STRUCTURES

- | | | |
|--|---|---|
| <ul style="list-style-type: none"> • Bounded_FIFO_Buffer <ul style="list-style-type: none"> • Clear_Buffer • Add_Element • Retrieve_Element • Peek • Buffer_Status • Buffer_Length • Unbounded_FIFO_Buffer <ul style="list-style-type: none"> • Initialize_Buffer • Clear_Buffer • Free_Memory • Add_Element • Retrieve_Element • Peek • Buffer_Status • Buffer_Length | <ul style="list-style-type: none"> • Nonblocking_Circular_Buffer <ul style="list-style-type: none"> • Clear_Buffer • Add_Element • Retrieve_Element • Peek • Buffer_Status • Buffer_Length • Unbounded_Priority_Queue <ul style="list-style-type: none"> • Initialize • Clear_Queue • Free_Memory • Add_Element • Retrieve_Element • Peek • Queue_Status • Queue_Length | <ul style="list-style-type: none"> • Bounded_Stack <ul style="list-style-type: none"> • Clear_Stack • Add_Element • Retrieve_Element • Peek • Stack_Status • Stack_Length • Unbounded_Stack <ul style="list-style-type: none"> • Initialize • Clear_Stack • Free_Memory • Add_Element • Retrieve_Element • Peek • Stack_Status • Stack_Length |
|--|---|---|

Figure 5-15: Abstract Data Structures Hierarchy**BOUNDED_FIFO_BUFFER**

- Clear_Buffer
- Add_Element
- Retrieve_Element
- Peek
- Buffer_Status
- Buffer_Length

NONBLOCKING_CIRCULAR_BUFFER

- Clear_Buffer
- Add_Element
- Retrieve_Element
- Peek
- Buffer_Status
- Buffer_Length

BOUNDED_STACK

- Clear_Stack
- Add_Element
- Retrieve_Element
- Peek
- Stack_Status
- Stack_Length

UNBOUNDED_FIFO_BUFFER

- Initialize_Buffer
- Clear_Buffer
- Free_Memory
- Add_Element
- Retrieve_Element
- Peek
- Buffer_Status
- Buffer_Length

UNBOUNDED_PRIORITY_QUEUE

- Initialize
- Clear_Queue
- Free_Memory
- Add_Element
- Retrieve_Element
- Peek
- Queue_Status
- Queue_Length

UNBOUNDED_STACK

- Initialize
- Clear_Stack
- Free_Memory
- Add_Element
- Retrieve_Element
- Peek
- Stack_Status
- Stack_Length

Figure 5-16: Alternate Abstract Data Structures Hierarchy

Table 5-3: Bundling by Type versus Class

AREA AFFECTED	BUNDLING BY CLASS	BUNDLING BY TYPE
Maintainability	Minimizes the number of packages since all the operations dealing with a class of objects will be placed in the same package. For example, a single abstract data structures package would contain multiple subpackages each dealing with a single type of abstract data structure (e.g., bounded stack, unbounded stack, bounded queue, unbounded queue).	Increases the number of packages since a separate package is created for each type of object.
	Packages can become very large. For example, the CAMP General_Vector_Matrix_Algebra package specification is 11,812 lines long, including headers, comments, blank lines, and 1135 lines of code. This makes the packages more difficult to maintain.	Since the operations are split among multiple packages, the individual packages which must be maintained are smaller than those created when parts are bundled by class. Consequently, maintenance of individual packages or subroutines is easier, decreasing the likelihood of introducing an error during maintenance.
	Minimizes the number of files to be maintained because the number of packages is minimized.	Increases the number of packages and files to maintain because multiple packages are created.
Compilation	Minimizes the number of packages which must be compiled into the Ada library.	Increases the number of packages which must be compiled into the Ada library.
	If an application needs to use any of the subroutines in the package, all of the subroutines in all of the subpackages must be compiled. For example, if an application requires unbounded stack operations, the code implementing the unbounded stack must be compiled along with the code implementing all the other abstract data structures defined by the package.	The user must compile only those packages which he requires for his application since they are no longer packaged with all the other operations in the class.
	If a subroutine's specification is modified or if a new subroutine or package is added, the resultant recompilation of the specification means that all the bodies for all the subpackages and subroutines also have to be recompiled.	Compilation time following revision of a part's specification is decreased. Since the operations for the different object types are in separate packages, a revision to a function specification affects only one package specification and, consequently, far fewer subroutines. This decreases the number of units requiring recompilation. The addition of a package would not affect any of the existing packages if it is designed as a separate package.
Usability	The user needs to look at only one package in order to locate a function.	By splitting the operations among multiple packages, the user can no longer look at only one package in an attempt to locate a function. However, when looking at one of the individual packages, it may be much easier for the user to determine whether or not the function is there.
	A large package may be more difficult to understand.	By splitting the operations among multiple, smaller packages, each individual package is easier to understand since it can be looked at in isolation.

When deciding how to package a set of objects or operations, the advantages and disadvantages of each of the bundling schemes need to be evaluated in light of the needs of the intended users. The primary advantage of bundling by type is that the resulting packages are smaller, making them easier to develop, maintain, and understand. The primary advantage of bundling by class is that it minimizes the search space for candidate software parts. This was the primary consideration when developing the CAMP parts, so parts were frequently bundled by class. This may not be the primary consideration for other reuse development efforts. Additionally, a parts catalog can alleviate some of the burden of searching through a large collection of parts.

Guideline #3-a: Avoid duplication of data types packages.

Guideline #3-b: Minimize variant proliferation. Concentrate on developing a tailorable component and providing tailoring guidelines.

Duplicate packages and variant proliferation pose problems for both parts users and developers, and may indicate that an area of tailorability had not been previously considered. Duplication increases the number of candidate parts a user must consider, thus increasing the difficulty of selecting parts for reuse. It can also create configuration management problems since more parts must be maintained and tracked. If an error is discovered in one part, the existence of variants may require the corrections to be made in several places, thus increasing the likelihood that additional errors will be introduced. There is also the possibility that not all the variant parts will be corrected or that they may not all be corrected in the same way.

As an example of part proliferation, consider the package specifications and bodies shown in Table 5-4; they are representative of an early design approach taken on an actual software reuse effort. As part of this effort, a set of unit-specific data type packages was to be developed, along with multiple packages for each set of operations implemented. The data type packages would contain the numeric types defining the units, as well as any vector/matrix operations required — duplicate packages would be developed for metric (meters and radians) and English (feet and degrees) units.

Several versions of each of the operation sets were implemented in an effort to increase reuse by providing something for everyone. The metric and English packages would provide parts that users could simply *with* into their application — no instantiations would be required. The package using the Ada predefined data types was provided for long-time FORTRAN programmers who might resist Ada's strong data typing. The generic package was provided for users who wanted to define their own data types rather than use those in the data type packages that was to be developed in conjunction with the parts. Table 5-4, while not containing the actual code developed as part of this effort, shows how this multiple package approach would have been applied to a simplistic example.

Although the goals of this approach were noble, there are a number of problems with it:

- There was too much duplication in the data types packages. Duplicate vector/matrix operations were to be implemented for both English and metric units. At the very least, these could be placed in a separate generic package that could be instantiated in each of the data types packages. Additionally, many, if not all, of these operations could be obtained from an existing parts set. Thus, not only should duplication be avoided in parts under construction, but care should be taken to ensure that new parts do not duplicate existing parts.
- *You can't be everything to everyone.* An attempt had been made to provide a stand-alone package that would satisfy all of the desires of all engineers — a package had been provided for engineers who wanted to use meters and radians, another for engineers who wanted feet and degrees, another for engineers who wanted units of feet and degrees but preferred their code to look like FORTRAN, and a generic package for engineers who wanted units of feet and degrees but preferred to supply their own data types. But what about engineers who wanted feet and radians? Or the engineers who wanted to provide their own data types, but wanted units of feet and degrees? Or the engineers who wanted to use the predefined data types with metric units? The list could go on and on. It simply is not practical to provide a separate set of code for each situation.

- Development, testing, configuration management, and maintenance of duplicate/variant packages can quickly become a nightmare. For each set of operations implemented in the example, at least four packages were planned. The personnel and time resources needed to support this type of development increases quickly, adding significantly to the cost of parts development.

Even given the resources needed for implementation of all of the variants, configuration management can quickly get out of hand. In this example, the algorithms in all the packages were identical, thus an error in the algorithm would have resulted in an error in every variant. Upon detection of the error, all of the variants would need to undergo modification, retesting, and rebaselining. This is an error-prone process since care needs to be taken to ensure that identical modifications are made to each and every variant.

- The generic package shown in Table 5-4 was too restrictive and made too many assumptions. Since these assumptions were not obvious from the code (i.e., the data types being brought in were named 'Distances' and 'Intervals2_per_Distance' rather than 'Feet' and 'Intervals2_per_Feet'), it would not be obvious from looking at an instantiation of this, or similar packages, that the use of 'Meters' and 'Radians' would be incorrect.

One approach to overcoming the limitations of the approach presented in this example is to develop a single, flexible generic package that could accommodate all the desired permutations. Careful design of this package would allow sufficient flexibility to provide all the operations of the individual packages shown in Table 5-4. This revised generic package is shown in Figure 5-17.

Table 5-4: Operations Implemented Via Multiple Packages

CODE EXAMPLE	COMMENTS
<pre> with BDT; package Metric_Opns is function Updated (X : BDT.Meters_per_Second) return BDT.Meters; end Metric_Opns; package body Metric_Opns is function Updated (X : BDT.Meters_per_Second) return BDT.Meters is A : constant BDT.Seconds2_per_Meter := 4.567; B : constant BDT.Seconds := 1.456; C : constant BDT.Meters := 0.123; Answer : BDT.Meters; begin Answer := A*X*X + B*X + C; return Answer; end Updated; end Metric_Opns; </pre>	Implementation in metric units of meters and radians. All required data types are obtained from a separate metric data types package. This package does not make use of the code in any of the other packages.
<pre> with BDT; package English_Opns is function Updated (X : BDT.Feet_per_Second) return BDT.Feet; end English_Opns; package body English_Opns is function Updated (X : BDT.Feet_per_Second) return BDT.Feet is A : constant BDT.Seconds2_per_Foot := 1.392; B : constant BDT.Seconds := 1.456; C : constant BDT.Feet := 0.403; Answer : BDT.Feet; begin Answer := A*X*X + B*X + C; return Answer; end Updated; end English_Opns; </pre>	Implementation in English units of feet and degrees. All required data types are obtained from a separate English data types package. This package does not make use of the code in any of the other packages.
<pre> package Float_Opns is function Updated (X : FLOAT) return FLOAT; end Float_Opns; package body Float_Opns is function Updated (X : FLOAT) return FLOAT is A : constant FLOAT := 1.392; B : constant FLOAT := 1.456; C : constant FLOAT := 0.403; Answer : FLOAT; begin Answer := A*X*X + B*X + C; return Answer; end Updated; end Float_Opns; </pre>	Implementation in English units of feet and degrees. All objects are declared using the predefined data types 'FLOAT' and 'INTEGER'.
<pre> generic type Distances is digits <>; type Intervals is digits <>; type Intervals2_per_Distance is digits <>; type Velocities is digits <>; function "*" (L : Intervals2_per_Distance; R : Velocities) return Intervals is <>; function "*" (L : Seconds; R : Velocities) return Distances is <>; package Generic_Opns is function Updated (X : Velocities) return Distances; end Generic_Opns; package body Generic_Opns is function Updated (X : Velocities) return Distances is A : constant Intervals2_per_Distance := 1.392; B : constant Intervals := 1.456; C : constant Distances := 0.403; Answer : Distances; begin Answer := A*X*X + B*X + C; return Answer; end Updated; end Generic_Opns; </pre>	Operations implemented in English units of feet and degrees. All the data types are obtained through generic formal parameters.

```

generic
  type Distances          is digits <>;
  type Intervals          is digits <>;
  type Intervals2_per_Distance is digits <>;
  type Velocities         is digits <>;
  A : in Intervals2_per_Distance;
  B : in Intervals;
  C : in Distances;
  function "*" (L : Intervals2_per_Distance;
               R : Velocities)
    return Intervals is <>;
  function "*" (L : Seconds;
               R : Velocities)
    return Distances is <>;
package Unitless_Generic_Opns is
  function Updated (X : Velocities) return Distances;
end Unitless_Generic_Opns;

package body Unitless_Generic_Opns is
  function Updated (X : Velocities) return Distances is
    Answer : Distances;
  begin
    Answer := A*X*X + B*X + C;
    return Answer;
  end Updated;
end Unitless_Generic_Opns;

```

Figure 5-17: Updated Generic Package

Guideline #4-a: *Provide pre-instantiated Ada generic parts where feasible.*

Guideline #4-b: *Use the Ada 'renames' facility to rename subroutines exported by instantiated packages, making it transparent to the user's code that routines are being obtained from an instantiated package rather than from a non-generic package.*

When developing reusable Ada generic parts, it may be desirable to *pre-instantiate* some of the parts rather than require the end-users to instantiate them. This can make the parts easier to use (the parts are immediately usable because the user does not have to decipher what is required to perform the instantiation), and can be accomplished by providing packages that instantiate a given generic unit. If the generic unit is a package, a set of packages may be provided which both instantiate the generic package and rename all the exported subroutines. Renaming is useful because the exported subroutine names can be shortened, making them easier to use. Using this technique, it is transparent to the application code that the routines are being obtained from an instantiated package rather than from a non-generic package, i.e., from a package that has been *with'd* in.

For example, consider the generic package presented in Figure 5-17. Figure 5-18 shows how a set of packages can be developed which instantiate the original generic package and then rename all of the exported routines.

```

with BDT;
package Metric_Opns is
  package Instantiated_Metric_Opns is new Unitless_Generic_Opns
    (Distances      => BDT.Meters,
     Intervals       => BDT.Seconds,
     Intervals2_per_Distance => BDT.Seconds2_per_Meter,
     Velocities      => BDT.Meters_per_Second,
     A               => 4.567,
     B               => 1.456,
     C               => 0.123);
  function Updated (X : BDT.Meters_per_Second) return BDT.Meters
    renames Instantiated_Metric_Opns.Updated;
end Metric_Opns;

with BDT;
package English_Opns is
  package Instantiated_English_Opns is new Unitless_Generic_Opns
    (Distances      => BDT.Feet,
     Intervals       => BDT.Seconds,
     Intervals2_per_Distance => BDT.Seconds2_per_Foot,
     Velocities      => BDT.Feet_per_Second,
     A               => 1.392,
     B               => 1.456,
     C               => 0.403);
  function Updated (X : BDT.Feet_per_Second) return BDT.Feet
    renames Instantiated_English_Opns.Updated;
end English_Opns;

package Float_Opns is
  package Instantiated_Float_Opns is new Unitless_Generic_Opns
    (Distances      => FLOAT,
     Intervals       => FLOAT,
     Intervals2_per_Distance => FLOAT,
     Velocities      => FLOAT,
     A               => 1.392,
     B               => 1.456,
     C               => 0.403);
  function Updated (X : FLOAT) return FLOAT
    renames Instantiated_Float_Opns;
end Instantiated_Float_Opns;

```

Figure 5-18: Predefined Instantiations of Generic Package

Figure 5-19 shows an example of the code required to reference the 'Updated' function contained in the implementation of 'Metric_Opns' shown in Table 5-4. This is identical to the code that would be required to access the 'Updated' function that is renamed in and exported by the 'Metric_Opns' package shown in Figure 5-18. Figure 5-20 shows the code that would be required if the function was not renamed.

```

with BDT;
with Metric_Opns;
procedure User_Application is
  X : BDT.Meters_per_Second;
  Y : BDT.Meters;
begin
  Y := Metric_Opns.Updated(X);
end User_Application;

```

Figure 5-19: Use of Instantiated Package with Renaming

```

with BDT;
with Metric_Opns;
procedure User_Application is
  X : BDT.Meters_per_Second;
  Y : BDT.Meters;
begin
  Y := Metric_Opns.Instantiated_Metric_Opns.Updated(X);
end User_Application;

```

Figure 5-20: Use of Instantiated Package without Renaming

Guideline #5: Minimize duplicate application-specific code.

Development of reusable code parts can be cost-effective even within a single project, thus during application development, portions of the application which are identified as reusable should be designed and coded for reuse. These parts may be very application- and project-specific and may never see reuse on projects other than the one for which they were developed, but if they are particularly complex or heavily used, the project can still benefit from decreased testing time, increased reliability, ease of modifications, etc. These parts are sometimes readily identified during design, or earlier, but other times, the commonality is not recognized until detailed design or coding.

While not every segment of identical or similar code should be replaced by a generic unit, the following situations should signal that the creation of a reusable component should be considered:

- Code is identical except for the data types being operated on.
- Code is identical except for the procedure or group of procedures called at a particular point in the algorithm.
- Significant or complicated sections of code are duplicated in multiple routines.
- A new routine or section of a routine can be written by performing a global search and replace on an existing piece of code.

While these situations seem obvious to the reader, they are not always obvious to the software developer. This is true even if the developer has experience with reusable software. While it may be readily apparent that code segments that deal with identical data types and operations should be developed as separate, reusable sub-routines, it may not initially be apparent when code should be developed as a generic unit. This is because many developers tend to shy away from the use of Ada generics, thinking them more appropriate for broader-based reuse efforts rather than for use within a single project.

Guideline #6: Generic units should be given names that do not conflict with their likely instantiated names.

The naming of reusable generic units can affect their ease-of-use. For example, in the CAMP parts set, names of generic units fall into two categories:

- The first includes names of generic functions that were destined to become operators (e.g., "*", "/", "+"). Section 12.1(4) of the Ada language reference manual (Reference [15]) specifies that the "designator of a generic subprogram must be an identifier"; therefore, these units had to be given English-like identifiers such as `Diagonal_Full_Matrix_Addition` or `Matrix_Matrix_Multiply`. These functions could be given operator symbols when instantiated by the user (e.g., function "*" is new `Matrix_Matrix_Multiply`).
- The second includes generic units which would continue to have English-like identifiers after instantiation (e.g., `Calculate_East_Velocity`, `Matrix_Matrix_Transpose`, `Cross_Product`), although not necessarily the same name as the generic unit.

Some of the generic units in the second category were inconvenient to use because during development they were given names that would be appropriate for use in an application rather than names that indicated that

they were generic. If the end-user of the part wanted the instantiated unit to have a different name than the generic unit, there was no inconvenience because he could write:

```
function Transpose is new Matrix_Matrix_Transpose ..
```

or

```
function East_Velocity is new Calculate_East_Velocity ...
```

The inconvenience arose when the user either wanted his instantiated routine to have the same name as the generic unit or needed it to be the same to allow generic formal subprograms to default elsewhere. This was because Section 8.3(5,22) of the Ada language reference manual (Reference [15]) prevented him from doing the following because "a reference to an identifier within its own declaration is illegal":

```
function Cross_Product is new Cross_Product ...
```

Consequently, the following is required for applications which require the instantiated name to be the same as the name of the generic unit:

```
function Cross_Prod is new
  CVHA.Cross_Product
  (Axes      => Navigation_Axes,
   Left_Elements => Radians_per_Sec_EFF,
   Right_Elements => Feet_per_Sec_EFF,
   Result_Elements => Feet_per_Sec_per_Sec_EFF,
   Left_Vectors  => Angular_Vel_Vectors,
   Right_Vectors  => Velocity_Vectors,
   Result_Vectors => Acceleration_Vectors);

function Cross_Product (Left  : Angular_Vel_Vectors;
                        Right : Velocity_Vectors)
  return Acceleration_Vectors
renames Cross_Prod;
```

While this is not a major problem and in no way affects the functionality or usefulness of a part, it does affect ease-of-use and user-acceptance, particularly if this has to be done frequently.

An addition to an organization's parts development standards that requires that all generic procedures and functions, and possibly packages, be named in such a way as to indicate they are generic (e.g., Generic_Cross_Product or Cross_Product_Template) would easily prevent this situation from occurring. This would allow the user to select a name without being concerned about whether or not it was the same as that of the generic unit.

Guideline #7: Compilation order should be included with composite Ada parts.

A very useful piece of information that should be included with every reusable code component is the compilation order. This will facilitate use of higher-level components because use of these components will likely require compilation of multiple packages. Determining compilation order can be a time-consuming, trial-and-error process, thus providing this information with a reusable part will make the part easier to use, thus decreasing the cost of reuse.

5.7 Case Studies

In the following paragraphs two case studies are presented that illustrate the process of designing reusable software components for real-time embedded applications. The first case study illustrates that the development of flexible, reusable parts is a recursive process, and that both review and actual use are vital components of the development cycle. The CAMP part, `Coordinate_Vector_Matrix_Algebra`, is used as an example in this case study. The second case study returns to the topic of data typing that was first raised in the design guidelines in Paragraph 5.6.

5.7.1 Case Study 1: Development of the CAMP Coordinate Vector/Matrix Algebra Package

This case study follows the development of the CAMP `Coordinate_Vector_Matrix_Algebra` (CVMA) package from domain analysis through maintenance. It also discusses additional enhancements which are possible. The purpose of this case study is to show that design of reusable parts is a recursive process, and that both reviews and actual use are integral in the parts development lifecycle.

5.7.1.1 Domain Analysis

During the CAMP domain analysis, the need for both general and specialized vector/matrix operations was identified. The general operations would operate on vectors and matrices of various sizes as specified by the end-users; the specialized operations would operate on 3-element vectors and 3x3 matrices. These coordinate data structures could be used to store guidance and navigation data. The general vector/matrix operations could also be used for this purpose, but execution speed could be maximized by designing a package that took advantage of the fixed size of the data structure. This difference in the code is illustrated in Table 5-5.

Table 5-5: General versus Coordinate Matrix Operation

Code for general matrix addition operation that makes no assumptions about the size of the matrix:	Code for coordinate matrix addition operation that assumes a 3x3 matrix:
<pre> generic type Rows is (<>); type Columns is (<>); type Elements is digits <>; package Matrix_Operations is type Matrices (Rows, Columns) of Elements; function "+" (L : Matrices; R : Matrices) return Matrices; end Matrix_Operations; package body Matrix_Operations is function "+" (L : Matrices; R : Matrices) return Matrices is separate; end Matrix_Operations; separate (Matrix_Operations) function "+" (L : Matrices; R : Matrices) return Matrices is Answer : Matrices; begin Go_Down_Rows: For Row in Rows loop Go_Across_Columns: For Col in Columns loop Answer(Row,Col) := L(Row,Col) + R(Row,Col); end loop Go_Across_Columns; end loop Go_Down_Rows; end "+"; </pre>	<pre> generic type Rows is (<>); type Columns is (<>); type Elements is digits <>; RX : in Rows := Rows'FIRST; RY : in Rows := Rows'SUCC(RX); RZ : in Rows := Rows'LAST; CX : in Columns := Columns'FIRST; CY : in Columns := Columns'SUCC(CX); CZ : in Columns := Columns'LAST; package Matrix_Operations is type Matrices (Rows, Columns) of Elements; function "+" (L : Matrices; R : Matrices) return Matrices; end Matrix_Operations; package body Matrix_Operations is function "+" (L : Matrices; R : Matrices) return Matrices is separate; end Matrix_Operations; separate (Matrix_Operations) function "+" (L : Matrices; R : Matrices) return Matrices is Answer : Matrices; begin -- --Row 1 Answer(RX,CX) := L(RX,CX) + R(RX,CX); Answer(RX,CY) := L(RX,CY) + R(RX,CY); Answer(RX,CZ) := L(RX,CZ) + R(RX,CZ); -- --Row 2 Answer(RY,CX) := L(RY,CX) + R(RY,CX); Answer(RY,CY) := L(RY,CY) + R(RY,CY); Answer(RY,CZ) := L(RY,CZ) + R(RY,CZ); -- --Row 3 Answer(RZ,CX) := L(RZ,CX) + R(RZ,CX); Answer(RZ,CY) := L(RZ,CY) + R(RZ,CY); Answer(RZ,CZ) := L(RZ,CZ) + R(RZ,CZ); end "+"; </pre>

5.7.1.2 Design

The Coordinate_Vector_Matrix_Algebra (CVMA) package was designed to satisfy the requirements for coordinate operations. The following design decisions were incorporated in this package:

- All operations would be grouped in a single package to facilitate identification of parts by end-users.
- The Coordinate_Vector_Matrix_Algebra package would contain two subpackages which exported a vector and matrix type, respectively, and general operations on those data types (e.g., addition, subtraction, set-to-zero).
- When possible, the names of the subroutines and the name and order of parameters would be the same as the corresponding routines located in the general vector/matrix algebra package. This would have several benefits:
 - It would allow the user to use either package as long as the routines he was using were defined in both of them. This switch could be effected by changing only the instantiation of the generic package — it would not be necessary to change the actual calls to the subroutines since their names, along with the names and order of their parameters, would be the same.
 - It would be possible to design higher-level routines, such as those dealing with navigation and guidance, that could accept routines from either package as their generic actual parameters.

- The data types would not be private types. This would allow users full access to the data types, if they required it, and is representative of the CAMP *semi-abstract data type* approach. It would also allow higher-level routines which required access to the data type to obtain it without incurring the overhead of a procedure call.
- The remaining operations (i.e., those not located in the subpackages which exported the data types) would be placed in the CVMA package and would be designed as generic units that accepted the data types exported by the matrix and vector operations packages.

Figure 5-21 contains a portion of an early version of the specification for this package. The design was subsequently revised to correct the unacceptable limitations in flexibility which are identified below.

- Vector_Operations: The generic data types did not provide for the intermediate results to be of a different data type. It also did not allow the square root of a data type to differ from the data type. When the Vector_Length performs its operations, it must first square each component of the input record and then take the square root of the sum of these squares. On a project employing strong data typing, a square root function would take in a squared data type (e.g., Feet_Squared, Seconds_Squared, Meters_Squared_per_Second_Squared) and return a non-squared data type (e.g., Feet, Seconds, Meters_per_Second). The design of the Vector_Operations package shown in Figure 5-21 did not support this.

The package was modified to include a third data type, *Elements_Squared*, that would be the result of multiplying two objects of type *Elements*, and a multiplication operator to define that operation.

- Cross_Product: This function incorrectly assumed that all vectors involved in a cross-product operation were of the same data type. For example, in the CAMP 11th Missile Application, the coriolis acceleration was calculated using the following formula:

$$\vec{a} = \vec{\omega} \times \vec{v}$$

where: $\vec{a}$ is the acceleration vector whose elements have units of $\frac{ft}{sec^2}$

$\vec{\omega}$ is the angular velocity vector whose elements have units of $\frac{radians}{sec}$

$\vec{v}$ is the velocity vector whose elements have units of $\frac{ft}{sec}$

$\times$ is the cross-product operator

The design of the Cross_Product function shown in Figure 5-21 did not support this operation because it required the data types of the elements of all the vectors to be the same. This function needed to be modified to allow the vectors to contain elements of different data types.

- Matrix_Matrix_Multiply: This function assumed that all matrices involved in the operation were of the same data type. This assumption was too restrictive, thus it was modified to allow for matrices with different element data types.

Figure 5-22 shows the revised partial package specification for the Coordinate_Vector_Matrix_Algebra package; it incorporates all of the changes identified above.

```

package Coordinate_Vector_Matrix_Algebra is

  generic
    type Axes      is (<>);
    type Elements  is digits <>;
    with function Sqrt (Input : Elements) return Elements is <>;
  package Vector_Operations is

    type Vectors is array(Axes) of Elements;

    function "+" (Left  : Vectors;
                  Right : Vectors) return Vectors;
    function "-" (Left  : Vectors;
                  Right : Vectors) return Vectors;
    function Vector_Length (Vector : Vectors) return Elements;
    function Dot_Product (Vector1 : Vectors;
                         Vector2 : Vectors) return Elements;
    function Sparse_Right_Z_Add (Left  : Vectors;
                                Right : Vectors) return Vectors;
    function Sparse_Right_X_Add (Left  : Vectors;
                                Right : Vectors) return Vectors;
    function Sparse_Right_XY_Subtract (Left  : Vectors;
                                       Right : Vectors) return Vectors;
    function Set_to_Zero_Vector return Vectors;

  end Vector_Operations;

  generic
    type Axes      is (<>);
    type Elements  is digits <>;
  package Matrix_Operations is

    type Matrices is array (Axes, Axes) of Elements;

    function "+" (Left  : Matrices;
                  Right : Matrices) return Matrices;
    function "-" (Left  : Matrices;
                  Right : Matrices) return Matrices;
    function "+" (Matrix : Matrices;
                  Addend : Elements) return Matrices;
    function "-" (Matrix : Matrices;
                  Subtrahend : Elements) return Matrices;
    function Set_to_Identity_Matrix return Matrices;
    function Set_to_Zero_Matrix return Matrices;

  end Matrix_Operations;

  generic
    type Axes      is (<>);
    type Elements  is digits <>;
    type Vectors  is array(Axes) of Left_Elements;
    function Cross_Product (Left  : Vectors;
                           Right : Vectors) return Vectors;

  generic
    type Axes      is (<>);
    type Elements  is digits <>;
    type Matrices is array (Axes, Axes) of Elements;
    function Matrix_Matrix_Multiply
      (Matrix1 : Matrices;
       Matrix2 : Matrices) return Matrices;

  end Coordinate_Vector_Matrix_Algebra;

```

Figure 5-21: Top-Level Design of Coordinate_Vector_Matrix_Algebra Package
(Prerelease Version)

```

package Coordinate_Vector_Matrix_Algebra is

  generic
    type Axes          is (<>);
    type Elements      is digits <>;
    type Elements_Squared is digits <>;
    with function "*" (Left : Elements;
                      Right : Elements) return Elements_Squared is <>;
    with function Sqrt (Input : Elements_Squared) return Elements is <>;
  package Vector_Operations is

    type Vectors is array(Axes) of Elements;

    function "+" (Left : Vectors;
                  Right : Vectors) return Vectors;
    function "-" (Left : Vectors;
                  Right : Vectors) return Vectors;
    function Vector_Length (Vector : Vectors) return Elements;
    function Dot_Product (Vector1 : Vectors;
                          Vector2 : Vectors) return Elements_Squared;
    function Sparse_Right_Z_Add (Left : Vectors;
                                 Right : Vectors) return Vectors;
    function Sparse_Right_X_Add (Left : Vectors;
                                 Right : Vectors) return Vectors;
    function Sparse_Right_XY_Subtract (Left : Vectors;
                                       Right : Vectors) return Vectors;
    function Set_to_Zero_Vector return Vectors;

  end Vector_Operations;

  generic
    type Axes          is (<>);
    type Elements      is digits <>;
  package Matrix_Operations is

    type Matrices is array (Axes, Axes) of Elements;

    function "+" (Left : Matrices;
                  Right : Matrices) return Matrices;
    function "-" (Left : Matrices;
                  Right : Matrices) return Matrices;
    function "+" (Matrix : Matrices;
                  Addend : Elements) return Matrices;
    function "-" (Matrix : Matrices;
                  Subtrahend : Elements) return Matrices;
    function Set_to_Identity_Matrix return Matrices;
    function Set_to_Zero_Matrix return Matrices;

  end Matrix_Operations;

  generic
    type Axes          is (<>);
    type Left_Elements is digits <>;
    type Right_Elements is digits <>;
    type Result_Elements is digits <>;
    type Left_Vectors   is array(Axes) of Left_Elements;
    type Right_Vectors  is array(Axes) of Right_Elements;
    type Result_Vectors is array(Axes) of Result_Elements;
    with function "*" (Left : Left_Elements;
                      Right : Right_Elements) return Result_Elements is <>;
    function Cross_Product (Left : Left_Vectors;
                           Right : Right_Vectors) return Result_Vectors;

  generic
    type Axes          is (<>);
    type Left_Elements is digits <>;
    type Right_Elements is digits <>;
    type Result_Elements is digits <>;
    type Left_Matrices is array (Axes, Axes) of Left_Elements;
    type Right_Matrices is array (Axes, Axes) of Right_Elements;
    type Result_Matrices is array (Axes, Axes) of Result_Elements;
    with function "*" (Left : Left_Elements;
                      Right : Right_Elements) return Result_Elements is <>;
    function Matrix_Matrix_Multiply
      (Matrix1 : Left_Matrices;
       Matrix2 : Right_Matrices) return Result_Matrices;

  end Coordinate_Vector_Matrix_Algebra;

```

Figure 5-22: Top-Level Design of Coordinate_Vector_Matrix_Algebra Package
(Release Version 1.0)

5.7.1.3 Maintenance

The CAMP 11th Missile effort identified a limitation in the CVMA parts that prevented them from being used for many navigation applications: in the initially released version of CVMA, the vectors and matrices were either 3x1 or 3x3, and it was assumed that the indices were the same on both axes of all the data structures involved in the operation. For example, in the `Matrix_Matrix_Multiply` function shown in Figure 5-22 three matrix data types were brought in via generic formal parameters. These three matrix types contained different element types, but both axes of all the matrices were dimensioned by the same data type. Consequently, the package could be used to do the following:

```

type Nav_Indices      is (X, Y, Z);
type Meters           is digits 9;
type Seconds          is digits 9;
type Meters_per_Second is digits 9;
type Meters_Matrices  is array (Nav_Indices, Nav_Indices) of Meters;
type Meters_per_Sec_Matrices is array (Nav_Indices, Nav_Indices) of Meters_per_Second;
type Seconds_Matrices is array (Nav_Indices, Nav_Indices) of Seconds;
...
Meters_Matrix      : Meters_Matrices;
Meters_per_Sec_Matrix : Meters_per_Sec_Matrices;
Seconds_Matrix     : Seconds_Matrices;
begin
...
Meters_Matrix := Meters_per_Sec_Matrix * Seconds_Matrix;
...
end;
```

but not the following because the x- and y-axes of all the matrices had to be the same:

```

type Earth_Indices      is (Greenwich, Right, Polar);
type Navigation_Indices is (X, Y, Z);
type Body_Indices       is (Nose, Right_Wing, Belly);
type Sin_Cos_Ratio      is digits 9;
type CNE_Matrices is array (Earth_Indices,
                             Navigation_Axes) of Sin_Cos_Ratio;
type CBN_Matrices is array (Navigation_Axes,
                             Body_Indices) of Sin_Cos_Ratio_FP;
type CBE_Matrices is array (Earth_Indices,
                             Body_Indices) of Sin_Cos_Ratio;
...
CNE_Matrix : CNE_Matrices;
CBN_Matrix : CBN_Matrices;
CBE_Matrix : CBE_Matrices;
begin
...
CBE_Matrix := CNE_Matrix * CBN_Matrix;
...
end;
```

Yet, this was exactly what the 11th Missile required as part of its earth-to-body transformations. The inability to specify different indices for the matrices was unacceptable for this application and resulted in the package not being used for 11th Missile matrix operations, although it was used for the coordinate vector operations. The general matrix package was used for these matrix operations, but the speed of these operations was suboptimal because the general package could make no assumptions about the size of the data structures.

During maintenance, the CVMA package was modified to allow all the axes of the vectors and/or matrices in an operation to be dimensioned by different data types. Figures 5-23-a and 5-23-b show a portion of the package specification that resulted from these enhancements.

```

package Coordinate_Vector_Matrix_Algebra is

  generic
    type Axes          is (<>);
    type Elements      is digits <>;
    type Elements_Squared is digits <>;
    X : in Axes := Axes'FIRST;
    Y : in Axes := Axes'SUCC(X);
    Z : in Axes := Axes'LAST;

    with function "*" (Left : Elements;
                      Right : Elements) return Elements_Squared is <>;
    with function Sqrt (Input : Elements_Squared) return Elements is <>;
  package Vector_Operations is

    type Vectors is array(Axes) of Elements;

    function "+" (Left : Vectors;
                 Right : Vectors) return Vectors;
    function "-" (Left : Vectors;
                 Right : Vectors) return Vectors;
    function Vector_Length (Vector : Vectors) return Elements;
    function Dot_Product (Vector1 : Vectors;
                        Vector2 : Vectors) return Elements_Squared;
    function Sparse_Right_Z_Add (Left : Vectors;
                                Right : Vectors) return Vectors;
    function Sparse_Right_X_Add (Left : Vectors;
                                Right : Vectors) return Vectors;
    function Sparse_Right_XY_Subtract (Left : Vectors;
                                       Right : Vectors) return Vectors;
    function Set_to_Zero_Vector return Vectors;

  end Vector_Operations;

  generic
    type M_Indices is (<>);
    type N_Indices is (<>);
    type Elements is digits <>;

    M_X : in M_Indices := M_Indices'FIRST;
    M_Y : in M_Indices := M_Indices'SUCC(M_X);
    M_Z : in M_Indices := M_Indices'LAST;
    N_X : in N_Indices := N_Indices'FIRST;
    N_Y : in N_Indices := N_Indices'SUCC(N_X);
    N_Z : in N_Indices := N_Indices'LAST;

  package Matrix_Operations is

    type Matrices is array (M_Indices, N_Indices) of Elements;

    function "+" (Left : Matrices;
                 Right : Matrices) return Matrices;
    function "-" (Left : Matrices;
                 Right : Matrices) return Matrices;
    function "+" (Matrix : Matrices;
                 Addend : Elements) return Matrices;
    function "-" (Matrix : Matrices;
                 Subtrahend : Elements) return Matrices;
    function Set_to_Identity_Matrix return Matrices;
    function Set_to_Zero_Matrix return Matrices;

  end Matrix_Operations;

```

Figure 5-23-a: Top-Level Design of Coordinate_Vector_Matrix_Algebra Package
(Release Version 1.1) (Part 1 of 2)

```

generic
  type Left_Axes      is (<>);
  type Right_Axes     is (<>);
  type Output_Axes    is (<>);
  type Left_Elements  is digits <>;
  type Right_Elements is digits <>;
  type Output_Elements is digits <>;
  type Left_Vectors   is array(Left_Axes) of Left_Elements;
  type Right_Vectors  is array(Right_Axes) of Right_Elements;
  type Output_Vectors is array(Output_Axes) of Output_Elements;
  L_X : in Left_Axes := Left_Axes'FIRST;
  L_Y : in Left_Axes := Left_Axes'SUCC(L_X);
  L_Z : in Left_Axes := Left_Axes'LAST;
  R_X : in Right_Axes := Right_Axes'FIRST;
  R_Y : in Right_Axes := Right_Axes'SUCC(R_X);
  R_Z : in Right_Axes := Right_Axes'LAST;
  O_X : in Output_Axes := Output_Axes'FIRST;
  O_Y : in Output_Axes := Output_Axes'SUCC(O_X);
  O_Z : in Output_Axes := Output_Axes'LAST;

  with function "*" (Left : Left_Elements;
                    Right : Right_Elements) return Output_Elements is <>;
function Cross_Product (Left : Left_Vectors;
                      Right : Right_Vectors) return Output_Vectors;

generic
  type Left_M_Indices  is (<>);
  type Left_N_Indices  is (<>);
  type Right_N_Indices is (<>);
  type Right_P_Indices is (<>);
  type Output_M_Indices is (<>);
  type Output_P_Indices is (<>);
  type Left_Elements    is digits <>;
  type Right_Elements    is digits <>;
  type Output_Elements   is digits <>;
  type Left_Matrices     is array (Left_M_Indices, Left_N_Indices) of Left_Elements;
  type Right_Matrices     is array (Right_N_Indices, Right_P_Indices) of Right_Elements;
  type Output_Matrices    is array (Output_M_Indices, Output_P_Indices) of Output_Elements;

  L_M_X : in Left_M_Indices := Left_M_Indices'FIRST;
  L_M_Y : in Left_M_Indices := Left_M_Indices'SUCC(L_M_X);
  L_M_Z : in Left_M_Indices := Left_M_Indices'LAST;
  L_N_X : in Left_N_Indices := Left_N_Indices'FIRST;
  L_N_Y : in Left_N_Indices := Left_N_Indices'SUCC(L_N_X);
  L_N_Z : in Left_N_Indices := Left_N_Indices'LAST;
  R_N_X : in Right_N_Indices := Right_N_Indices'FIRST;
  R_N_Y : in Right_N_Indices := Right_N_Indices'SUCC(R_N_X);
  R_N_Z : in Right_N_Indices := Right_N_Indices'LAST;
  R_P_X : in Right_P_Indices := Right_P_Indices'FIRST;
  R_P_Y : in Right_P_Indices := Right_P_Indices'SUCC(R_P_X);
  R_P_Z : in Right_P_Indices := Right_P_Indices'LAST;
  O_M_X : in Output_M_Indices := Output_M_Indices'FIRST;
  O_M_Y : in Output_M_Indices := Output_M_Indices'SUCC(O_M_X);
  O_M_Z : in Output_M_Indices := Output_M_Indices'LAST;
  O_P_X : in Output_P_Indices := Output_P_Indices'FIRST;
  O_P_Y : in Output_P_Indices := Output_P_Indices'SUCC(O_P_X);
  O_P_Z : in Output_P_Indices := Output_P_Indices'LAST;

  with function "*" (Left : Left_Elements;
                    Right : Right_Elements) return Output_Elements is <>;
function Matrix_Matrix_Multiply
  (Matrix1 : Left_Matrices;
   Matrix2 : Right_Matrices) return Output_Matrices;

end Coordinate_Vector_Matrix_Algebra;

```

Figure 5-23-b: Top-Level Design of Coordinate_Vector_Matrix_Algebra Package
(Release Version 1.1) (Part 2 of 2)

5.7.1.4 Further Enhancements

The package could have been further generalized by modifying the CVMA.Vector_Operations. The package, shown in Figure 5-23-a, brings in three data types and two functions. One of these data types (Elements_Squared) and one of the functions (Sqrt) are required for the Vector_Length function exported by the package. In order to instantiate this package, a square root function contained in another package must be instantiated even if the new application does not require the Vector_Length function.

Figure 5-24 shows an alternate version of the Vector_Operations package. In this version, the Vector_Length function and associated generic parameters have been reorganized into a generic function declared within the Vector_Operations package. This version eliminates the need to provide these parameters when the application does not require a length function, thereby making the package easier to use. The disadvantage is that projects requiring the length function are forced to perform an additional instantiation. The original design was chosen because the domain analysis indicated that coordinate vectors frequently required a length function.

```
package Coordinate_Vector_Matrix_Algebra is
  generic
    type Axes          is (<>);
    type Elements      is digits <>;
    X : in Axes := Axes'FIRST;
    Y : in Axes := Axes'SUCC(X);
    Z : in Axes := Axes'LAST;
    with function "*" (Left : Elements;
                      Right : Elements) return Elements_Squared is <>;
  package Vector_Operations is
    type Vectors is array(Axes) of Elements;

    function "+" (Left : Vectors;
                  Right : Vectors) return Vectors;
    function "-" (Left : Vectors;
                  Right : Vectors) return Vectors;
    function Dot_Product (Vector1 : Vectors;
                          Vector2 : Vectors) return Elements_Squared;
    function Sparse_Right_Z_Add (Left : Vectors;
                                 Right : Vectors) return Vectors;
    function Sparse_Right_X_Add (Left : Vectors;
                                 Right : Vectors) return Vectors;
    function Sparse_Right_XY_Subtract (Left : Vectors;
                                        Right : Vectors) return Vectors;
    function Set_to_Zero_Vector return Vectors;

    generic
      type Elements_Squared is digits <>;
      with function Sqrt (Input : Elements_Squared) return Elements is <>;
      with function Sqrt (Input : Elements_Squared) return Elements is <>;
      function Vector_Length (Vector : Vectors) return Elements;
    end Vector_Operations;
  end Coordinate_Vector_Matrix_Algebra;
```

Figure 5-24: Modified Top-Level Design of Vector_Operations Package

5.7.2 Case Study 2: Data Typing

This case study builds on the data typing guidelines from Section 5.6, using the trigonometric packages initially introduced there. In the following paragraphs, four different designs of a trigonometric package are discussed. Each has been designed to support different degrees of data typing and to provide varying degrees of control to the user. The purpose of this case study is to show how the design of these packages affects their incorporation into the user's application. For each package it is shown how it could be used (1) in a weakly typed applications using either the Ada predefined type `FLOAT` or a user-defined data type (e.g., type `My_Float` is digits 6;), and (2) in a strongly typed application.

One characteristic of a well-designed reusable part is the ease with which it can be incorporated into a user's application. While reusable components should be designed to permit and promote strong data typing, they should also allow the user to make the final decision regarding the strength of data typing. Consequently, a part should be designed to be easily incorporated into both weakly and strongly typed applications without requiring type conversions to be performed with every use.

5.7.2.1 Weakly Typed Parts

The two packages discussed in this paragraph employ weak data typing. One of these packages is generic, the other is not. The primary difference between these packages is the degree of control given to the user.

1. In the trigonometric package shown in Figure 5-25, the predefined data type `FLOAT` is used to define the input and output parameters, giving the user no control over the data types.

```
package TrigOpsla is
-- --expects inputs in units of radians
  function Sin (Angle : FLOAT) return FLOAT;
  function Cos (Angle : FLOAT) return FLOAT;
  function Tan (Angle : FLOAT) return FLOAT;
end TrigOpsla;
```

Figure 5-25: Weakly Data Typed Package (Float)

This version will fit into the user's application if that application uses the predefined data type `FLOAT`, but, if the user creates his own general data type (e.g., type `My_Float` is digits 6;) or employs stronger data typing, he will be forced to perform type conversions for all calls to routines in this package. The examples in Figure 5-26 show the use of this package by an application using the predefined data type `FLOAT` and stronger data typing than provided by this package.

```

with TrigOpsla;
procedure User_Application is
  Angle : FLOAT;
  Cos_of_Angle : FLOAT;
begin
  Cos_of_Angle := TrigOpsla.Sin(Angle);
end User_Application;

with TrigOpsla;
procedure User_Application is
  type Radians is digits 6;
  type Ratios is digits 6;
  Radian_Angle : Radians;
  Cos_of_Angle : Ratios;
begin
  Cos_of_Angle := Ratios (TrigOpsla.Sin (FLOAT (Radian_Angle)));
end User_Application;

```

Figure 5-26: Use of Weakly Typed Non-Generic Package

If the user's application employs stronger data typing than the reusable part, an interface package could be written rather than calling the trigonometric package directly and performing the necessary type conversions. This way, the floating point trigonometric package would appear to the rest of his application as a strongly typed package. This is illustrated in Table 5-6.

Use of an interface package successfully solves the problem of interfacing a strongly-typed application with weakly typed parts, but it imposes an additional burden on the user that can be avoided through more careful design.

Table 5-6: Interface to Weakly Typed Non-Generic Package

CODE EXAMPLE	COMMENTS
<pre> package Trig is type Radians is digits 6; type Ratios is digits 6; function Sin (Angle : Radians) return Ratios; function Cos (Angle : Radians) return Ratios; function Tan (Angle : Radians) return Ratios; end Trig; with TrigOpsla; package body Trig is function Sin (Angle : Radians) return Ratios is Answer : Ratios; begin Answer := Ratios (TrigOpsla.Sin (FLOAT (Angle))); return Answer; end Sin; ... end Trig; </pre>	This package acts as an interface to the TrigOpsla package.
<pre> with Trig; procedure User_Application is Radian_Angle : Trig.Radians; Cos_of_Angle : Trig.Ratios; begin Cos_of_Angle := Trig.Sin(Radian_Angle); end User_Application; </pre>	Strong data typing can be maintained by application code without required type conversions on each call through use of interface package.

2. In the trigonometric package shown in Figure 5-27, weak data typing is also employed, but in this example the package is generic and its single data type is brought in via a generic parameter.

```

generic
  type Values is digits <>;
package TrigOps1b is
  -- --expects inputs in units of radians
  function Sin (Angle : Values) return Values;
  function Cos (Angle : Values) return Values;
  function Tan (Angle : Values) return Values;
end TrigOps1b;

```

Figure 5-27: Weakly Data Typed Package (Generic)

The generic aspect of this package gives the user more control since he can either use the predefined data type `FLOAT` or define his own universal floating point type. The examples in Table 5-7 show how this package can be used in either situation.

Table 5-7: Use of Weakly Typed Generic Package in Weakly Typed Application

CODE EXAMPLE	COMMENTS
<pre> with TrigOps1b; procedure User_Application is package Trig is new TrigOps1b (Values => FLOAT); Angle : FLOAT; Cos_of_Angle : FLOAT; begin Cos_of_Angle := Trig.Sin(Angle); end User_Application; </pre>	Use of TrigOps1b package in application using the Ada predefined type <code>FLOAT</code> .
<pre> with TrigOps1b; procedure User_Application is type My_Float is digits 6; package Trig is new TrigOps1b (Values => My_Float); Angle : My_Float; Cos_of_Angle : My_Float; begin Cos_of_Angle := Trig.Sin(Angle); end User_Application; </pre>	Use of TrigOps1b package in an applications defining general floating point type.

The package shown in Figure 5-27 gives the user greater control than the one in Figure 5-25, but still forces the user to perform type conversions if he employs stronger data typing in his application than in the reusable parts.

The example in Table 5-8 shows the use of this package in an application employing strong data typing.

Table 5-8: Use of Weakly Typed Generic Package in Strongly Typed Application

CODE EXAMPLE	COMMENTS
<pre> with TrigOps1b; procedure User_Application is type My_Float is digits 6; type Radians is digits 6; type Ratios is digits 6; package Trig is new TrigOps1b (Values => My_Float); Radian_Angle : Radians; Cos_of_Angle : Ratios; begin Cos_of_Angle := Ratios(Trig.Sin(My_Float(Angle))); end User_Application; </pre>	Type conversions are required if the TrigOps1b package is called directly by an application employing strong data typing.

As with the previous package, an interface package could be written by the user to provide a clean interface between his strongly-typed application and the weakly typed trigonometric package. This interface package could also provide overloaded routines for angles with units other than radians. This is illustrated in Table 5-9.

Table 5-9: Interface to Weakly Typed Generic Package

CODE EXAMPLE	COMMENTS
<pre> package Trig is type Radians is digits 6; type Degrees is digits 6; type Ratios is digits 6; function Sin (Angle : Radians) return Ratios; function Cos (Angle : Radians) return Ratios; function Tan (Angle : Radians) return Ratios; function Sin (Angle : Degrees) return Ratios; function Cos (Angle : Degrees) return Ratios; function Tan (Angle : Degrees) return Ratios; end Trig; with TrigOps1b; package body Trig is package My_Trig is new TrigOps1b (Values => Radians); function Sin (Angle : Radians) return Ratios is Answer : Radians; begin Answer := My_Trig.Sin(Angle); return Ratios(Answer); end Sin; ... function Sin (Angle : Degrees) return Ratios is Answer : Radians; R_Angle : Radians; begin R_Angle := Convert to Radians(Angle); Answer := My_Trig.Sin(R_Angle); return Ratios(Answer); end Sin; ... end Trig; </pre>	This package acts as an interface to the TrigOps1a package.
<pre> with Trig; procedure User_Application is Radian_Angle : Trig.Radians; Cos_of_Angle : Trig.Ratios; begin Cos_of_Angle := Trig.Sin(Radian_Angle); end User_Application; </pre>	Use of the interface package, allows application code to maintain strong data typing without performing type conversions with every call.

5.7.2.2 Strongly Typed Parts

Both of the packages discussed in this paragraph employ strong data typing and are generic. As with the previous examples, these packages differ in the degree of control they offer the user.

1. The package shown in Figure 5-28 not only supports strong data typing, but actually forces the user to employ strong data typing or perform type conversions with each call. This is because the exported subroutines do not operate on the data types brought in via generic parameters, but rather on locally declared data types which are derived from the general formal types.

```

generic
  type Input_Values is digits <>;
  type Output_Values is digits <>;
package TrigOps2a is
  type Radians is new Input_Values;
  type Degrees is new Input_Values;
  type SC_Ratios is new Output_Values;
  type T_Ratios is new Output_Values;
  function Sin (Angle : Radians) return SC_Ratios;
  function Cos (Angle : Radians) return SC_Ratios;
  function Tan (Angle : Radians) return T_Ratios;
  function Sin (Angle : Degrees) return SC_Ratios;
  function Cos (Angle : Degrees) return SC_Ratios;
  function Tan (Angle : Degrees) return T_Ratios;
end TrigOps2a;

```

Figure 5-28: Strongly Data Typed Package (Inflexible)

Table 5-10 illustrates the use of this more strongly data typed package by an application using the predefined data type `FLOAT`.

Table 5-10: Strongly Typed Part in Weakly Typed Application

CODE EXAMPLE	COMMENTS
<pre> with TrigOps2a; procedure User_Application is package Trig is new TrigOps2a (Input Values => FLOAT, Output Values => FLOAT); Angle : FLOAT; Cos_of_Angle : FLOAT; begin Cos_of_Angle := FLOAT(Trig.Cos(Trig.Radians(Angle))); end User_Application; </pre>	The type conversions would be required with each call because the <code>TrigOps2a</code> package forces strong data typing.

This package is an improvement over the packages shown in Figures 5-25 and 5-27 in that it supports strong data typing, requires less work to instantiate than the package shown in Figure 5-30 because it has fewer generic parameters, and does more work for the user than the package in Figure 5-30 because it exports the data types. It has a disadvantage over the package in Figure 5-30 in that it forces the user to either perform type conversions with each call or write another layer of code between it and the application if the user wants or needs to declare his data types external to this package (e.g., in another trigonometric package that he is using); this makes the package subject to misuse.

Figure 5-29 shows an example of an application which uses two trigonometric packages, the one previously shown in Figure 5-28, and a new, faster package. Since both of these packages export data types, the user must either choose one of the sets of data types and perform type conversions when calling the other package (this is the alternative chosen in the following example), or declare his own data types and perform conversions on all calls. The requirement to perform type conversions increases the possibility of misuse of the package — if a data type is converted to the "wrong" type, the wrong routine will be called.

The problem of merging both of these trigonometric packages into one application could be resolved through the use of an interface package as shown in Table 5-11. This, however, imposes an unnecessary burden on the user that can be avoided through more careful design.

```

generic
  type Input Values is digits <>;
  type Output Values is digits <>;
package Really_Fast_TrigOps is
  type Radians is new Input Values;
  type Degrees is new Input Values;
  type SC_Ratios is new Output Values;
  type T_Ratios is new Output Values;
  function Sin (Angle : Radians) return SC_Ratios;
  function Cos (Angle : Radians) return SC_Ratios;
  function Tan (Angle : Radians) return T_Ratios;
  function Sin (Angle : Degrees) return SC_Ratios;
  function Cos (Angle : Degrees) return SC_Ratios;
  function Tan (Angle : Degrees) return T_Ratios;
end Really_Fast_TrigOps;

with Really_Fast_TrigOps;
with TrigOps2a;
procedure User_Application is
  type My_Input Values is digits 6;
  type My_Output Values is digits 6;
  package Fast_Trig is new Really_Fast_TrigOps
    (Input Values => My_Input Values,
     Output Values => My_Output Values);
  package Trig is new TrigOps2a
    (Input Values => My_Input Values,
     Output Values => My_Output Values);
  Angle : Fast_Trig.Radians;
  Cos_of_Angle : Fast_Trig.SC_Ratios;
begin
  -----
  -- --As long as the application only called routines in the Fast_Trig
  -- --package, no type conversions would be required
  -----
  Cos_of_Angle := Fast_Trig.Cos(Angle);
  -----
  -- --When the user wanted to call the other Trig package, type
  -- --conversions would be required which could lead to the following
  -- --error where a 'degree' cosine function is use to operate on an
  -- --angle with units of radians
  -----
  Cos_of_Angle := Fast_Trig.SC_Ratios(Trig.Cos(Trig.Degrees(Angle)));
end User_Application;

```

Figure 5-29: Use of Multiple Trig Packages

Table 5-11: Interface to Multiple, Inflexible Trig Packages

CODE EXAMPLE	COMMENTS
<pre> package Trig is type Radians is digits 6; type Degrees is digits 6; type Ratios is digits 6; function Sin (Angle : Radians) return Ratios; function Cos (Angle : Radians) return Ratios; function Tan (Angle : Radians) return Ratios; function Sin (Angle : Degrees) return Ratios; function Cos (Angle : Degrees) return Ratios; function Tan (Angle : Degrees) return Ratios; function Fast_Sin (Angle : Radians) return Ratios; function Fast_Cos (Angle : Radians) return Ratios; function Fast_Tan (Angle : Radians) return Ratios; function Fast_Sin (Angle : Degrees) return Ratios; function Fast_Cos (Angle : Degrees) return Ratios; function Fast_Tan (Angle : Degrees) return Ratios; end Trig; with TrigOpns2a; with Really_Fast_TrigOpns; package body Trig is package Reg_Trig is new TrigOpns2a (Input Values => Radians, Output Values => Ratios); package Fast_Trig is new Really_Fast_TrigOpns (Input Values => Radians, Output Values => Ratios); function Sin (Angle : Radians) return Ratios is Answer : Reg_Trig.Ratios; begin Answer := Reg_Trig.Sin(Reg_Trig.Radians(Angle)); return Ratios(Answer); end Sin; ... function Fast_Sin (Angle : Radians) return Ratios is Answer : Fast_Trig.Ratios; begin Answer := Fast_Trig.Sin(Fast_Trig.Radians(Angle)); return Ratios(Answer); end Fast_Sin; end Trig; </pre>	This package acts as an interface to the TrigOpns2a and Really_Fast_TrigOpns packages.
<pre> with Trig; procedure User_Applications is Angle : Trig.Radians; Cos_of_Angle : Trig.Ratios; begin Cos_of_Angle := Trig.Cos(Angle); Cos_of_Angle := Trig.Fast_Cos(Angle); end User_Applications; </pre>	User of an interface package allows the application program to call routines in both trigonometric packages without performing type conversions.

2. The package shown in Figure 5-30 has the advantage over the other trigonometric packages shown in Figures 5-25, 5-27, and 5-28 in that it allows the user to decide for himself how the data typing should be performed. It supports strong data typing, but does not require it; it also allows the user to declare the data types external to the package.

This package supports various levels of data typing as shown in Table 5-12. The design also facilitates the use of multiple trigonometric package as shown in Figure 5-31.

```
generic
  type Radians      is digits <>;
  type Degrees      is digits <>;
  type SC_Ratios     is digits <>;
  type T_Ratios      is digits <>;
package TrigOps2b is
  function Sin (Angle : Radians) return SC_Ratios;
  function Cos (Angle : Radians) return SC_Ratios;
  function Tan (Angle : Radians) return T_Ratios;
  function Sin (Angle : Degrees) return SC_Ratios;
  function Cos (Angle : Degrees) return SC_Ratios;
  function Tan (Angle : Degrees) return T_Ratios;
end TrigOps2b;
```

Figure 5-30: Strongly Data Typed Package (Flexible)

Table 5-12: Use of Flexible Package in Multiple Applications

CODE EXAMPLE	COMMENTS
<pre> with TrigOps2b; procedure Minimal_Radian_Data_Typing_Application is type Unused is new FLOAT; package Trig is new TrigOps2b (Radians => FLOAT, Degrees => Unused, SC_Ratios => FLOAT, T_Ratios => FLOAT); begin ... end Minimal_Radian_Data_Typing_Application; with TrigOps2b; procedure Minimal_Degree_Data_Typing_Application is type Unused is new FLOAT; package Trig is new TrigOps2b (Radians => Unused, Degrees => FLOAT, SC_Ratios => FLOAT, T_Ratios => FLOAT); begin ... end Minimal_Degree_Data_Typing_Application; </pre>	In this example, the TrigOps2b package is being using in an application which is using the predefined data type FLOAT. To use the generic package, the user simply instantiates it, specifying FLOAT for Radians, SC_Ratios, and T_Ratios if his application requires radian calculations or specifying the same data type for Degrees, SC_Ratios, and T_Ratios if his application requires degree calculations. (Radians and Degrees require separate data types to disambiguate the overloaded functions.)
<pre> with TrigOps2b; procedure Moderate_Radian_Data_Typing_Application is type Angles is digits 6; type Ratios is digits 6; type Unused is digits 6; package Trig is new TrigOps2b (Radians => Angles, Degrees => Unused, SC_Ratios => Ratios, T_Ratios => Ratios); begin ... end Moderate_Radian_Data_Typing_Application; with TrigOps2b; procedure Moderate_Degree_Data_Typing_Application is type Angles is digits 6; type Ratios is digits 6; type Unused is digits 6; package Trig is new TrigOps2b (Radians => Unused, Degrees => Angles, SC_Ratios => Ratios, T_Ratios => Ratios); begin ... end Moderate_Degree_Data_Typing_Application; </pre>	In this example, the user's application is employing slightly stronger data typing. While he does not need to specify the difference between a radian and an angle, and a sine/cosine ratio and a tangent ratio, he does need to specify the difference between the input and output values. He therefore instantiates the TrigOps2b package, providing separate types for the data types of the inputs (radians and degrees) and outputs (SC_Ratios and T_Ratios) when specifying the generic actual parameters.
<pre> with TrigOps2b; procedure Strong_Data_Typing_Application is type Radians is digits 6; type Degrees is digits 6; type SC_Ratios is digits 6; type T_Ratios is digits 6; package Trig is new TrigOps2b (Radians => Radians, Degrees => Degrees, SC_Ratios => SC_Ratios, T_Ratios => T_Ratios); begin ... end Strong_Data_Typing_Application; </pre>	In this example, the TrigOps2b package is being used in an application employing strong data typing. Here the user has specified different data types for each of the generic actual data types.

```

generic
  type Radians      is digits <>;
  type Degrees      is digits <>;
  type SC_Ratios     is digits <>;
  type T_Ratios      is digits <>;
package Really_Fast_TrigOps is
  function Sin (Angle : Radians) return SC_Ratios;
  function Cos (Angle : Radians) return SC_Ratios;
  function Tan (Angle : Radians) return T_Ratios;
  function Sin (Angle : Degrees) return SC_Ratios;
  function Cos (Angle : Degrees) return SC_Ratios;
  function Tan (Angle : Degrees) return T_Ratios;
end Really_Fast_TrigOps;

with TrigOps2b;
with Really_Fast_TrigOps;
procedure Strong_Data_Typing_Application is
  type Radians_ is digits 6;
  type Degrees_ is digits 6;
  type SC_Ratios_ is digits 6;
  type T_Ratios_ is digits 6;
  package Trig is new TrigOps2b
    (Radians_ => Radians,
     Degrees_ => Degrees,
     SC_Ratios_ => SC_Ratios,
     T_Ratios_ => T_Ratios);
  package Fast_Trig is new Really_Fast_TrigOps
    (Radians_ => Radians,
     Degrees_ => Degrees,
     SC_Ratios_ => SC_Ratios,
     T_Ratios_ => T_Ratios);
begin
  ...
end Strong_Data_Typing_Application;

```

Figure 5-31: Use of Multiple Flexible Trig Packages

5.8 Summary

The design approach presented here was used in development of the CAMP parts set. Its use in other domains will show the power of this approach in developing an integrated parts base. The ease with which parts can be fit into an application has shown the method to be extremely effective (see Reference [31]). The CAMP parts, which employ this method, have been easy to maintain, and the CAMP parts base has been extended as applications discover the need for additional parts.

Six alternative approaches were evaluated: typeless, overloaded, generic, abstract state machine, abstract data type, and skeletal code. The semi-abstract data type method is a hybrid approach based on the generic and overloaded methods. This was the approach was developed and used on the CAMP program. The method emphasizes the importance of efficiency. It utilizes derived types and subprograms, generic instantiation, and subprogram overloading. Multiple layers of generics provide the user with a broad selection of parts for an application. Predefined types and operators can be used to instantiate generic parts. This method is also characterized by what is referred to as a *part bundle*. A *bundle* is a collection of parts dealing with a particular type or class of object or operation, together with their environment. A key feature of the semi-abstract data type approach is the open architecture — the user can substitute his own parts for predefined parts anywhere in the part hierarchy. The semi-abstract data type is under the user's control.

CHAPTER 6

DETAILED DESIGN and CODING of SOFTWARE PARTS

In many respects, the activities during detailed design and coding are the same for both reusable software and custom code:

1. Completion of preliminary or top-level design
2. Allocation of requirements to lower level software components
3. Preparation of a software or unit development file (SDF/UDF)
4. Preliminary design walkthrough
5. Completion of detailed design
6. Preparation of test procedure
7. Detailed design walkthrough
8. Coding
9. Code walkthrough

In this chapter the differences that arise during development of reusable code versus custom code will be discussed, e.g., the special documentation needs of reusable software, the importance of coding guidelines and a few suggestions for developing these guidelines, and the impact of reuse on design reviews.

6.1 Merging Detailed Design and Coding

Detailed design and coding can often be merged through the use of Ada as the design language. The primary purpose of the program design language (PDL) representation developed during detailed design is to improve understanding of the software by providing additional information that is an appropriate level of abstraction above the code. The key here is that detailed design should be at a *higher* level of abstraction than the code. If it is not, then there may be excessive duplication of effort during detailed design and coding. Certain characteristics indicate when it is appropriate to skip development of PDL and go directly to code for development of parts.

- Low-level requirements: The requirements are specified at a very low level. For example, in the CAMP parts set, the algorithms for many of the math parts were completely specified during the requirements phase, therefore, there was no need to repeat these algorithmic requirements in the detailed design.
- Parts built from other parts: High-level parts can be designed to accept other parts as generic parameters. The highest level parts directly instantiate other parts that are required to perform lower level operations. The example in Figure 6-1 shows the detailed design/code for the procedure 'Kalman_Filter_Complicated_H_Parts.Sequentially_Updated_Covariance_Matrix_and_State_Vector.Update', and illustrates the instantiation of parts within other parts. The PDL for this procedure would be similar to the following:

```
for each measurement in the state vector loop
  compute K (Kalman gain)
  update P (error covariance matrix)
  update X (state vector)
end loop
```

The actual code for this procedure would be very similar to the PDL if, as in the CAMP parts, the calculations of a new K, P, and X consisted simply of calls to other routines. This similarity can be seen by comparing the PDL above with the actual code, shown in Figure 6-1, which contains nothing more than a loop and three subroutine calls.

- Parts are small: Parts tend to be small. The CAMP parts averaged less than 36 lines of code, and, therefore, were relatively simple. The need for high-level comments frequently decreases as the simplicity of the code increases. This can be seen in Figures 6-2 and 6-3 which contain simple and relatively complex routines, respectively. Figure 6-2 shows a piece of code which sets all elements of a symmetric, full-storage matrix to 0.0. The code for this procedure is quite simple and self-explanatory, and thus, contains no comments other than those in the code header. Figure 6-3, on the other hand, contains a more complicated piece of code which subtracts a symmetric, full-storage matrix from an identity matrix. Because of the complexity of this code, comments in addition to those contained in its code header are required. The ratio of comments to code for this piece of code is better than 1:2.

While merging of the detailed design and coding phases, hereafter referred to as detailed design, is not always appropriate, a part's developer should evaluate his situation to determine applicability. An additional consideration before making this decision is the development and documentation standards imposed by the customer (e.g., if PDL is required by the customer, then detailed design and coding cannot be merged without customer approval).

6.2 Identification of Additional Parts

Additional components may be identified during design and implementation. For instance, it may become apparent that variants of previously identified parts may be needed. Supporting parts may also be identified. In general, most of the parts identified during this time will be lower level parts. It is important that supporting or lower level parts identified during this time be developed as separate components, otherwise their functionality may be repeated across several other components.

6.3 Design Reviews

The design presented for walkthrough must be reviewed to ensure conformance with requirements, conformance with existing design and coding standards, consistency with other parts, completeness of documentation, and conformance of code headers to any documentation standards. Enforcement of standards in the development of reusable software is essential.

The primary differences between reusable code and one-shot code are as follow:

- Reusable code must be designed to accommodate anticipated future use.
- Reusable code may need to be more robust.
- Reusable code must minimize external dependencies that might be acceptable in custom code.

These factors must be taken into consideration during design reviews.

Although domain experts should be included in the domain analysis and should act as reviewers for both requirements and top-level design, they are not always needed as reviewers for the detailed design/code. Inclusion of domain experts in the detailed design review is dependent upon the complexity of the part.

- Assuming the requirements and top-level design have been reviewed by a domain expert, and that the algorithm is relatively straightforward, there may be little value in participation by a domain expert unless he is knowledgeable about the design/implementation language.

```

package body Kalman_Filter_Complicated_H_Parts is

  function Compute_Kalman_Gain ...
  procedure Update_Error_Covariance_Matrix ...
  procedure Update_State_Vector ...

  package body Sequentially_Update_Covariance_Matrix_and_State_Vector is

    K : K_Column_Vectors;

    function Compute_K is new Compute_Kalman_Gain ...
    procedure Update_P is new Update_Error_Covariance_Matrix ...
    procedure Update_X is new Update_State_Vector ...

    procedure Update (P
                      : in out P_Matrices;
                      X
                      : in out State_Vectors;
                      Z
                      : in   Measurement_Vectors;
                      Complicated_H
                      : in   H_Matrices;
                      Measurement_Variance : in   Measurement_Variance_Vector) is

    begin

      for Measurement_Number in Measurement_Indices loop

        K := Compute_K (P
                       => P,
                       Measurement_Number => Measurement_Number,
                       Complicated_H
                       => Complicated_H,
                       Measurement_Variance => Measurement_Variance);

        Update_P (P
                 => P,
                 Measurement_Number => Measurement_Number,
                 K
                 => K,
                 Complicated_H
                 => Complicated_H);

        Update_X (X
                 => X,
                 Z
                 => Z,
                 K
                 => K,
                 Measurement_Number => Measurement_Number,
                 Complicated_H
                 => Complicated_H);

      end loop;

    end Update;

  end Sequentially_Update_Covariance_Matrix_and_State_Vector;

end Kalman_Filter_Complicated_H_Parts;

```

Figure 6-1: For High-Level Parts, Detailed Design is Code

```

separate (General_Vector_Matrix_Algebra.
         Symmetric_Full_Storage_Matrix_Operations_Unconstrained)
procedure Set_To_Zero_Matrix (Matrix : out Matrices) is

begin

  Matrix := (others => (others => 0.0));

end Set_To_Zero_Matrix;

```

Figure 6-2: Simple Parts Require Few Comments

```

function Subtract_from_Identity (Input : Matrices) return Matrices is
-- -----
-- --declaration section
-- -----

    Answer      : Matrices(Input'RANGE(1), Input'RANGE(2));
    Col         : Col_Indices;
    Col_Count   : POSITIVE;
    Row         : Row_Indices;
    Row_Count   : POSITIVE;
    S_Col       : Col_Indices;
    S_Row       : Row_Indices;

-- -----
--begin function Subtract from_Identity
-- -----

begin
-- --make sure input matrix is a square matrix
if Input'LENGTH(1) = Input'LENGTH(2) then

-- --will subtract input matrix from an identity matrix by first
-- --subtracting all elements from 0.0 and then adding 1.0 to the
-- --diagonal elements;
-- --when doing the subtraction, will only calculate the remainder
-- --for the elements in the bottom half of the matrix and will simply
-- --do assignments for the symmetric elements in the top half of the
-- --matrix
    Row_Count := 1;

-- --S_Col will go across the columns as Row goes down the rows;
-- --Will mark column containing the diagonal element for this row
    Row := Input'FIRST(1);
    S_Col := Input'FIRST(2);
    Do_Every_Row:
    loop
        Col_Count := 1;

-- --S_Row will go down the rows as Col goes across the columns;
-- --when paired with S_Col will mark the symmetric counterpart
-- --to the element being referenced in the bottom half of the
-- --matrix
        Col := Input'FIRST(2);
        S_Row := Input'FIRST(1);
        Subtract_Elements_From_Zero:
        loop
-- --perform subtraction on element in bottom half of matrix
            Answer(Row,Col) := - Input(Row,Col);

-- --exit loop after diagonal element has been reached
            exit Subtract_Elements_From_Zero when Col_Count =
                Row_Count;

-- --assign values to symmetric elements in top half of matrix
-- --(done after check for diagonal, since diagonal elements
-- --don't have a symmetric counterpart)
            Answer(S_Row,S_Col) := Answer(Row,Col);

-- --increment variables
            Col_Count := Col_Count + 1;
            Col := Col_Indices'SUCC(Col);
            S_Row := Row_Indices'SUCC(S_Row);

        end loop Subtract_Elements_From_Zero;

-- --add one to the diagonal element
        Answer(Row, Col) := Answer(Row, S_Col) + 1.0;

        exit Do_Every_Row when Row_Count = Input'LENGTH(1);
        Row_Count := Row_Count + 1;
        Row := Row_Indices'SUCC(Row);
        S_Col := Col_Indices'SUCC(S_Col);

    end loop Do_Every_Row;

    else
        raise Dimension_Error;
    end if;

    return Answer;
end Subtract_from_Identity;

```

Figure 6-3: Complicated Parts Require More Comments

- As the complexity of algorithms increases, it becomes increasingly difficult for non-domain experts to understand them. A domain expert may more readily be able to verify the correctness of the code, whereas a non-domain expert must rely solely on tracing the requirements to the design.

The designer (who is not also a domain expert) may misinterpret the requirements, resulting in parts that do not meet the needs of future users. The domain expert reviewer is more likely to identify overall design flaws, whereas the software engineer reviewer is likely to identify only minor deviations from the requirements. Again, the value of domain expert review depends on the representation of the design, and on the familiarity the domain expert has with that representation.

Non-domain experts will generally require more information than a domain expert about complex algorithms before identifying implementation errors. Non-domain expert reviewers frequently assume that since the designer knows more about the algorithms, then he has probably written correct code.

6.4 Design for Efficiency

Real-time embedded applications present a particular challenge to the parts developer — efficiency is critical in these applications. One way to incorporate speed and storage efficiency in the design, is to develop special-purpose parts. For example, in the CAMP parts set, there is a collection of parts that deal with matrix data types and operators for matrices of unknown dimensions (i.e., general vector/matrix operations). Within this collection, there are parts for full storage, half storage, diagonal, and dynamically sparse matrix types and operators. There is also a collection of special-purpose parts that is specifically for coordinate vector operations (i.e., coordinate vector/matrix operations); these parts all make use of the fact that the vectors will be of length 3, and the matrices will be dimensioned by 3. This saves space and time. Use of these different vector routines is illustrated in Figure 6-4. Here, the casual user (i.e., a user who's first priority is not execution speed) might use a general routine that requires nine operations to perform the division: $3 \times (1 \text{ Divide} + 1 \text{ Increment_Pointer} + 1 \text{ Test_for_end_of_Array})$. Another user, knowing that he is working with coordinate vectors, may select the operator that knows about the data structure it is working with; the division would then require only 3 operations: one division for each element in the coordinate vector. Finally, a third user may know that one of the coordinate elements is always zero, and develop a special-purpose routine to take this into account; this division could then be accomplished with only 2 operations.

6.5 Coding Guidelines

Guidelines for coding reusable Ada parts are proliferating. These generally provide suggestions about the use of various Ada constructs to improve reusability (see References [38], [37], [8], and others). Although adherence to coding standards is important in the development of reusable software, standards alone do not make software reusable. Many of the guidelines have not been used in RTE applications and tend to adhere to idealistic goals. Some things to consider when evaluating guidelines for use in the development of reusable parts are enumerated below.

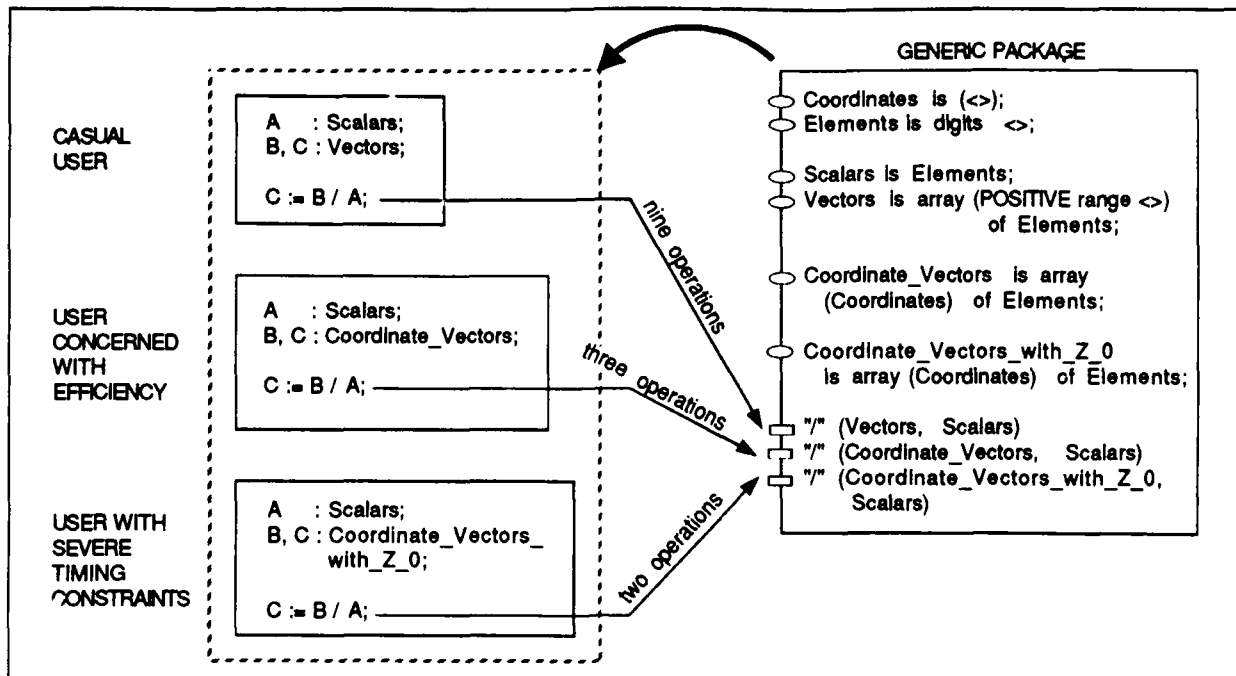


Figure 6-4: Parts Can Be Designed for Various Degrees of Efficiency

- Are the guidelines needed?
- Are they being used?
- Have they been used on a "real" project?
- Have they been used on a project similar to the one currently under development (i.e., one with the same types of constraints, etc.)?
- What were the results of their use?
- How do they differ from good coding practices?

The use of standards during development of reusable parts is important from the point of view of the developers, as well as the users. When users are assessing a set of parts for possible inclusion in their application, they need to learn what types of operations are available, how the parts interface with each other, how they are going to interface with their application, and what the parts do. In order to do this, they need to be able to easily understand the parts — this may require learning the style used to organize them, document them, create names for the packages and subroutines, create names for the data types and objects, etc. The user's job will be simplified if the development style for all the parts is the same. In order for the style to be the same across a given set, standards must be developed early in the parts' development lifecycle and enforced throughout the development process. Standards are also useful to parts developers, particularly to new developers brought onto the project midway through the lifecycle.

Standards for development of reusable code parts should address the following areas:

- **File Names:** How should the files containing the reusable code parts be named? Obviously, this is system dependent, but a consistent naming scheme should be developed in order to facilitate component retrieval and configuration management. For example, standards on the CAMP program required that the primary component of the file name be a two-part prefix, "xxx_yyy_". The first part of the prefix (i.e., "xxx") consisted of a top-level package identification number (e.g., 621 for Basic_Data_Types, 684 for Geometric_Operations, 001 for Common_Navigation); this

was used on all files pertaining to a particular top-level package, including test procedures, test plan, test results. The second part of the prefix (i.e., "yy") was used to indicate the level of nesting of the part contained in the file and was indicative of compilation order for the top-level package. For example, a user of the CAMP parts set could obtain a list of all files related to the Common_Navigation_Parts TLCSC by specifying "001*." for retrieval.

- Naming conventions:

- How should data types and objects be named? Should the names of all data types be plural and those of all objects singular?
- How should packages be named? Should they reflect the object being manipulated (e.g., Ordered_List, Balanced_Tree, Terminal) or should they reflect the operations on the objects being manipulated (e.g., Ordered_List_Operations, Balanced_Tree_Operations, Terminal_Operations)?
- How should generic units be named? Should they have a name similar to what would be used in the final application (e.g., Cross_Product, Compute_East_Velocity) or should they be coded to indicate they are generic (e.g., Cross_Product_Template, Compute_East_Velocity_Template) so that the end-user can select a name for the instantiation without being concerned about whether or not that name is the same as the name of the generic unit (see naming guidelines in Paragraph 5.6)?
- How should abbreviations be handled? Should all words in identifiers be spelled out? Are some abbreviations acceptable? Should certain abbreviations always be used instead of spelling the word out (e.g., should 'Opns' always be used instead of 'Operations')?

- Bundling: Should parts be bundled by types of operations and objects or by classes of operations and objects?

- Context clauses: Should the 'use' clause be permitted, and, if so, to what extent? Should it be permitted at anytime to allow items in other packages to be referenced without qualification? Should it only be used to allow access to operators (i.e., "+", "-", "*", etc.) with the "honor system" used to ensure developers use expanded names elsewhere? Should no 'use' clauses be allowed so that it is obvious what is being used from where (perhaps allowing operators to be renamed locally so that the infix notation could continue to be used)?

- Documentation: How extensive should code headers be? Should they be able to stand on their own (this will cause the headers to be large with a considerable amount of duplicate information)? Should they not repeat information that can be found in the code? Should they not repeat information that can be found in other headers? Should they tell the user how this part can be used with other parts in the set?

6.6 Documentation

Documentation of reusable parts is critical. Reusable code parts have somewhat different documentation requirements than custom code. The documentation for reusable code parts must include information on adaptation, sample usage of the part, and dependencies (e.g., environmental, hardware, compiler, and interpart dependencies). It also needs to include the standard items, such as purpose, input/output parameters, etc.

Although providing these additional items may impose a burden on the part developer, it eases the way for the part user. Consider the difficulty in understanding even a short segment of code when it is unfamiliar and undocumented. See Figures 6-3 and 6-5 to compare the difference that documentation makes. Although parts users may want to review the code itself, this does not negate the need for comments to explain use, dependencies, interactions, etc.

Documentation should take two forms — external documents (e.g., requirements, design, user's manual) and code headers. Code headers for reusable software will generally contain more usage information than one-shot

```

function Subtract_from_Identity (Input : Matrices) return Matrices is
-- -----
-- --declaration section
-- -----
    Answer : Matrices(Input'RANGE(1), Input'RANGE(2));
    Col     : Col_Indices;
    Col_Count : POSITIVE;
    Row     : Row_Indices;
    Row_Count : POSITIVE;
    S_Col   : Col_Indices;
    S_Row   : Row_Indices;

--begin function Subtract_from_Identity
-- -----
begin
    if Input'LENGTH(1) = Input'LENGTH(2) then
        Row_Count := 1;
        Row       := Input'FIRST(1);
        S_Col     := Input'FIRST(2);
        Do_Every_Row:
        loop
            Col_Count := 1;
            Col        := Input'FIRST(2);
            S_Row      := Input'FIRST(1);
            Subtract_Elements_From_Zero:
            loop
                Answer(Row,Col) := - Input(Row,Col);
                exit Subtract_Elements_From_Zero when Col_Count = Row_Count;
                Answer(S_Row,S_Col) := Answer(Row,Col);
                Col_Count := Col_Count + 1;
                Col        := Col_Indices'SUCC(Col);
                S_Row      := Row_Indices'SUCC(S_Row);
            end loop Subtract_Elements_From_Zero;
            Answer(Row, Col) := Answer(Row, S_Col) + 1.0;
            exit Do_Every_Row when Row_Count = Input'LENGTH(1);
            Row_Count := Row_Count + 1;
            Row       := Row_Indices'SUCC(Row);
            S_Col     := Col_Indices'SUCC(S_Col);
        end loop Do_Every_Row;
    else
        raise Dimension_Error;
    end if;
    return Answer;
end Subtract_from_Identity;

```

Figure 6-5: Complicated Parts with No Comments

code. The need for all of this documentation arises from the fact that parts users will not be familiar with the parts and hence, will need a fairly significant amount of information in order to effectively use them. Ideally, the documentation should also be available electronically so that the reuser can incorporate it into, or reference it in, the documentation for his new application.

Reusable Ada parts will, most likely, make extensive use of generic units, thus, a sample instantiation should be included in the documentation of generic parts. This should show not only how to instantiate the part, but also how other parts can be used to provide the required generic actual data types, objects, and/or subprograms. The sample usage section can show how the generic formal parameters can be used to tailor the part, e.g., how to tailor a matrix multiplication routine for use with dynamically sparse matrices. During part development, this portion of the documentation is time-consuming to produce and easily affected by modifications to the part, but it can be invaluable during part use. For example, during the CAMP 11th Missile Application development and during CAMP parts maintenance, the sample usage information turned out to be one of the more useful pieces of documentation for software engineers who were unfamiliar with the parts.

In many cases, a software developer is required to maintain a software development notebook (Reference [16]); it is generally a good idea regardless of the contractual requirement. A software development file, or notebook, should be maintained for each part (or group of parts) under development; it is a useful way to capture many of the decisions that are made in the course of software development, and can be useful as an additional source of information about reusable software parts. The SDF should be available electronically and should be used to maintain the information identified below.

1. Requirements
2. Preliminary design
3. Detailed design
4. Test plan/procedure, along with test code and test data
5. Test results
6. Problem reports and log: The software discrepancy reports (SDRs) and their disposition.
7. Change orders and log: Software enhancement proposal/software change proposal forms (SEP/SCP), along with their disposition
8. Walkthrough records, design decisions, notes, etc.

It is important to remember that the guidelines presented in this manual must be reconciled with any customer-required or contractual documentation standards.

One approach to documentation that can boost both productivity and increase documentation quality is to maintain the data needed for documentation in a central location and then automatically generate much of the documentation. This can eliminate redundancy (i.e., the information is entered into the central "repository" only once and can be extracted for any number of documents that require it), boost productivity (data is entered only once and changes are made in only one place), and increase product quality (fewer inconsistency because data is entered and modified in only one place). This is particularly useful for the end-users of the parts. For example, all of the design information for the CAMP parts is contained in their specification and body headers. This had two benefits: the information only had to be provided once, and it could be updated at the same time that the code was maintained. A software tool was developed to extract information from appropriate sections of the headers for placement in the design documents. Figure 6-1 identifies the type of information that is maintained in these headers, and indicates which of these sections are extracted for use in the top-level or detailed design documents (Note: CAMP parts were developed under DOD-STD-2167 not DOD-STD-2167A). When an updated document was required, the text merely had to be re-extracted.

This type of tool facilitates the production of extensive, high-quality documentation by eliminating tedious and often error-ridden duplication. Even with the elimination of duplicate documentation, maintaining documentation in a current state is not an easy task. Commercial software tools are also available that can be used to maintain a central data dictionary and generate DoD standard documentation.

Table 6-1: CAMP Code Header Information (DOD-STD-2167)

<u>HEADER CONTENTS</u>	<u>EXTRACTED FOR DESIGN DOCUMENT</u>
Name	.
Identification Number	.
Security Level	.
Purpose	.
Requirements trace	.
Context	.
Utilization of external elements	.
Packages	.
Subprograms and task entries	.
Exceptions	.
Data types	.
Data objects	.
Utilization of other elements in top-level component (DD)	.
Packages	.
Subprograms and task entries	.
Exceptions	.
Data types	.
Data objects	.
Input/output	.
Generic parameters	.
Data types	.
Data objects	.
Subprograms	.
Formal parameters	.
Local exceptions/types/objects (DD)	.
Exceptions	.
Data types	.
Data objects	.
Local entities (DD)	.
Exceptions raised (DD)	.
Calling sequence (DD)	.
Exported exceptions/types/objects (TLD)	.
Exceptions	.
Data types	.
Data objects	.
Exceptions raised (TLD)	.
Calling sequence/timing/priority (TLD)	.
Interrupt handling (TLD)	.
Sample usage (TLD)	.
Decomposition (TLD)	.
Local entities contained in package body (TLD)	.

CHAPTER 7

TESTING of REUSABLE SOFTWARE PARTS

Rigorous testing is essential to the success of any software reuse effort. One of the factors that has kept reuse levels low in the past has been skepticism about the quality of available parts. Virtually every software engineer feels that his code is better than anyone else's; consequently most engineers are reluctant to trust their project to someone else's code. Reuse can be jeopardized by reusable components of questionable quality; parts do not have to be error-ridden to cause problems. This is an extremely important point to keep in mind when evaluating software for inclusion in a reusable component library.

It is important to provide the test procedures, code, data, and expected test results along with the parts so that the end-user (i.e., the software engineer who is incorporating reusable software in a new application) can retest them if he wants. Tests should be repeatable, and test results should be reproducible. Providing the potential reuser with all of the test documentation (i.e., the test plan, test procedure, test code, and test results) will increase the likelihood that reusable software is actually reused. Until there is a well-established reusable software industry that has a widely accepted level of quality and documentation standards, and perhaps even software warranties, retesting of reusable software components will probably be required. The extent of the retesting will depend on the application (i.e., Is it mission critical? Does it have hard real-time constraints? Will integration testing suffice?).

Development of the test plan and procedures should be done in conjunction with a domain expert who would be in a position to determine both the adequacy of the testing and the expected test results. Testing of parts is much the same as testing of custom code. Reusable software components are designed to accommodate needed variation in future applications, thus, more test cases may be required for reusable software than for custom software.

Portability, which is sometimes considered a factor in reusability, can be tested by compiling the parts with more than one compiler on more than one platform. This will also permit the identification of any unintentional compiler and platform dependencies that may have crept into the code. Reusability among applications on the same type of platform can also produce savings. When higher-level components, such as requirements or design, are reused, the portability problem is diminished.

Test activities include the identification and organization of required tests, preparation of a test plan and procedure, preparation of test code and expected test results, actual testing of the part, and evaluation and reporting of the test results. The test cases should be included in the review at the detailed design walkthrough. Following completion of all design walkthroughs and implementation of walkthrough action items, parts should be given to a tester (preferably someone other than the designer) for development of the test code and actual testing.

Unit and integration testing of parts can be combined into a single phase if testing is performed in a bottom-up fashion, i.e., if all parts requiring other parts directly, or designed to use them through generic parameters are actually tested using the supporting parts which have already passed testing. This approach can shorten testing by eliminating the need to write code stubs and by eliminating the need to first test a part in isolation and then retest it using the parts themselves.

In a parts development effort there may be no software integration phase, or it may be quite limited, thus, there may well be no integration testing. Obviously, composite parts (i.e., parts that are composed from lower-level parts) would undergo some level of integration testing. There is also no system integration and testing for reusable parts because of the nature of reusable software.

Figure 7-1 depicts the software testing cycle. Parts may well require several passes through this cycle. A part should be put under configuration control after successfully passing all tests.

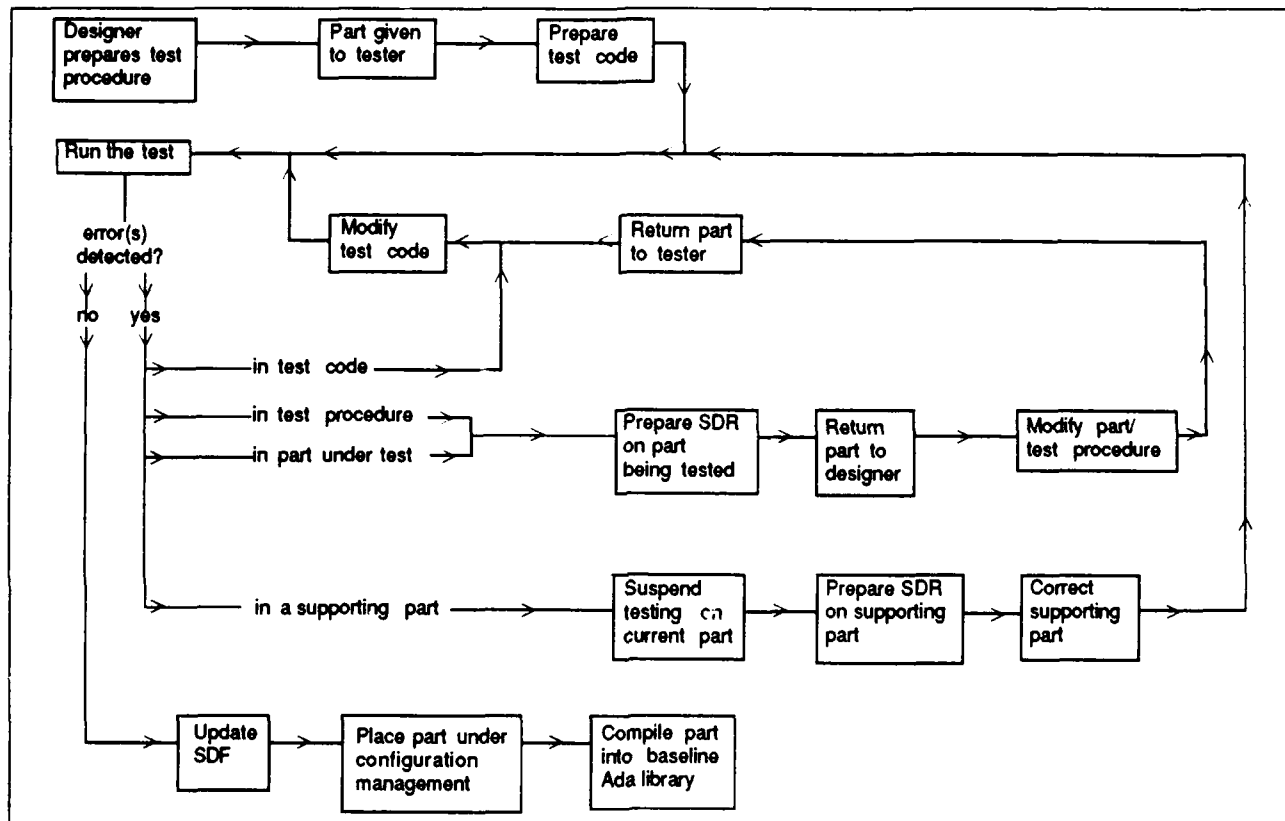


Figure 7-1: Parts Testing Cycle

7.1 Unit and Package Testing

In general, both subroutine, or unit-level, and package-level components are tested. Usually unit- and package-level tests are separate, but they can sometimes be combined or the package-level test can be skipped. Table 7-1 presents a decision matrix that can be used to determine the level of testing that is required.

Package-level testing is not required if there is no interaction among the units in the package. For example, a data types package that is just a collection of data type definitions and operators, with no data stored in the package body and units that do not invoke each other, would not require package-level testing (e.g., the CAMP Basic_Data_Types part). If the units and the interactions among them are both simple, the unit tests can be folded into the package-level test (e.g., Alignment_Measurements) or the unit tests can cover the package-level test requirements. If the units are complex, then unit-level tests is always required. If the unit interactions are simple, the package-level test requirements can be covered by the unit tests; if complex, a separate package-level test should be required.

Table 7-1: Unit- and Package-Level Test Decision Matrix

	CASES				
	1	2	3	4	5
CONDITION Unit	-	Simple	Simple	Complex	Complex
Interaction	None	Simple	Complex	Simple	Complex
TESTS REQUIRED Unit	Yes	Combine	Part of package test	Yes	Yes
Package	No	Combine	Yes	Part of unit tests	Yes

7.1.1 Unit Test Approach

Unit tests should cover both *white box* and *black box* viewpoints. A *white box* test is designed with knowledge of the unit's structure. The test cases are set up to exercise all paths through the unit and to invoke all branch conditions. A *black box* test is a functional test that assumes nothing about the unit's internal structure. It passes in a representative sample of input data and checks to see if the output is as expected.

These two approaches to unit testing can be illustrated using the CAMP Alignment_Measurements package. This package (see Figure 7-2) takes a sequence of integrated velocities from the navigator, keeps and corrects running sums of them, and periodically formats the sums into measurements and sends them to the Kalman_Filter package. Calls to control procedures (Initialize, Set_Measurement_Time, and Cancel_Measurement) initialize the package, specify when a measurement is to be sent to Kalman_Filter, and occasionally cancel a measurement. Integrate, Put_Reference_Velocity_Integrals, and Apply_Kalman_Position_Corrections receive data needed to compute or correct the integrated-velocity sums. Get_Integrals returns the current integrated-velocity sums.

A black box test would invoke the package with a sequence of control and data calls, verify that the current sums returned by Get_Integrals are correct, verify that the package sends correct measurements to Kalman_Filter at the correct times, and verify that a measurement is not sent if it has been canceled. The test designer would use the requirements specification and the Ada package specification to develop the tests.

The white box test designer would also use the package and procedure body listings to generate test cases that cover all the paths and exercise all the branch conditions. For example, the white box test of procedure Cancel_Measurement (see Figure 7-3) would call this procedure twice, once with Measurement_Pending true and once with it false. (Measurement_Pending is stored in the package body.)

7.2 Summary

Because software parts will see use in a variety of situations — some of them not foreseen by the original developer — testing must be comprehensive in order to ensure correctness and robustness. Testing should be carried out on a variety of platforms using various compilers; this will ensure that machine/compiler dependencies have not unintentionally crept into the code. Test plans, procedures, inputs, and results should be made available to the potential part user. This is one way to increase user confidence and facilitate reuse.

```

with Nav_Computer_Data_Types;
package Alignment_Measurements is

    package NCDT renames Nav_Computer_Data_Types;

    procedure Initialize (Initial_Time      : in NCDT.Seconds;
                          Reference_Altitude : in NCDT.Feet_FP;
                          Velocity          : in NCDT.Velocity_Vectors);

    procedure Integrate (Eff_Time_Of_Incr_Data : in NCDT.Seconds;
                        Velocity              : in NCDT.Velocity_Vectors;
                        Altitude              : in NCDT.Feet_EFP);

    procedure Put_Reference_Velocity_Integrals
        (X_Velocity_Integral : in NCDT.Feet_FP;
         Y_Velocity_Integral : in NCDT.Feet_FP);

    procedure Apply_Kalman_Position_Correction
        (Position_Error_X : in NCDT.Earth_Position_Radians;
         Position_Error_Y : in NCDT.Earth_Position_Radians);

    procedure Get_Integrals (Integrated_Vel_X : out NCDT.Feet_FP;
                            Integrated_Vel_Y : out NCDT.Feet_FP);

    procedure Set_Measurement_Time (Time : in NCDT.Seconds);

    procedure Cancel_Measurement;

end Alignment_Measurements;

```

Figure 7-2: Package Alignment_Measurements

```

with Environment;
separate (Alignment_Measurements)
procedure Cancel_Measurement is
begin

    if Measurement_Pending then

        -- --send invalid measurement to the Kalman
        Environment.Measurements.Here_is_Alignment_Measurement
            (Time_of_Alignm_Data => Measurement_Time,
             Alignm_Valid       => FALSE,
             MeasX               => 0.0,
             MeasY               => 0.0,
             MeasZ               => 0.0,
             VarX                => 0.0,
             VarY                => 0.0,
             VarZ                => 0.0);

        -- --cancel measurement pending flag
        Measurement_Pending := FALSE;

    end if;

end Cancel_Measurement;

```

Figure 7-3: Procedure Cancel_Measurement

CHAPTER 8

OTHER TOPICS in PARTS DEVELOPMENT

In this chapter the following topics will be discussed with respect to the development of reusable software components:

- Management issues
- Scope of software reuse
- Parts development organizations
- Configuration management
- Maintenance
- Ada language considerations
- Tools

8.1 Management Issues

Management issues can be broken into two categories: management support and management of the process. In the following paragraphs both issues will be discussed as they relate to the development of reusable software.

8.1.1 Management Support

Management support is needed in obtaining the resources to acquire (i.e., develop or purchase) reusable components and integrate reuse into the software development process. When a reuse program is begun, management must realize that they are making an investment in future software development efforts and that the payback may not be immediate. Management needs to stop thinking about software as a disposable good and start thinking about it as an investment. Reusable software provides a way to capture domain expertise and apply it to future application development.

In order for the development of reusable software components to be cost-effective, the additional cost associated with their development must be amortized over several uses. The additional cost results from the need to perform a domain analysis, the need to identify areas of variability for future software development efforts, and the need for additional testing (e.g., to cover the areas of variability) and documentation. It is extremely difficult for a project to begin a reuse program on its own because of the relatively high cost of parts development, and the realities of project time and cost constraints. It may be necessary, at least initially, for management to provide additional funding to projects to cover the start-up costs of software reuse.

It is sometimes difficult to establish widespread software reuse within an organization, even when management supports the cause. The manager of one software reuse group had management support, but still lacked project support although he had data to show that use of his parts could save projects time and money. In this particular case, his group developed common utilities that were broadly applicable to the types of software developed by his company. When a new project was initiated, this software reuse manager would meet with the project manager to discuss the availability and application of reusable components. The general arrangement was that the reuse group would bear the cost of component development and would assist in the use of the components. They would also provide maintenance as long as the components were not modified. Even with this type of support, projects were reluctant to "sign up." (Reference [40])

Management can promote reuse by a combination of mandate and incentives. One of the traditional barriers to software reuse is a "cultural" barrier — the "not invented here" syndrome plays a significant role in inhibiting reuse. At least initially, it may be necessary for management to require each project to at least examine and evaluate available components. A company's (or project's) software engineering practices can be modified to

make reuse part of the standard practices. Reviews can then be used to determine the extent to which reusable components are used in the object under review, as well as areas where reusable components could have been used but were not. Incentives can take a number of different forms. Software developers can be rewarded for the amount of software they reuse within a given period, or they can be rewarded for the amount of reuse that their software sees during a given period. Both approaches have been used.

8.1.2 Process Management

Reviews play a particularly important role in the development of reusable software, particularly during domain analysis and initial parts development. During domain analysis, reviews are needed to ensure that the domain analysts do not wander too far from the domain of interest. Domain expert review is critical during the entire process, with the possible exception of detailed design and coding activities.

Reusable component development requires close ties with the customer, regardless of whether the customer is internal (e.g., a project group), or external (e.g., a government agency). A parts development group must be responsive to customer needs, and must maintain good communications in order to ensure that the reusable software will actually meet the customer's needs.

When undertaking a software reuse program, management must ensure that the proper personnel are available to staff the effort. A parts development team needs to be highly skilled in modern software engineering principles, as well as in the implementation language. They need to have a high level of discipline so that software development and coding standards are followed throughout the parts lifecycle. And they need good communication skills in order to work effectively with the domain experts, the customer, and management.

8.2 Scope of Reuse

Reuse can be practiced at more than one organizational level, i.e., it can be practiced at the project level, program level, company level, corporate level, industry, etc. At the lowest level — the project level — a project group would identify portions of an application that are common across the project and develop those parts in such a way that they could be reused within the application. In this case, the project would both develop and use the reusable software. Once the project had developed and tested the parts, they should be evaluated for dissemination to other projects (i.e., they may find use on other applications) or for inclusion in a reuse library system. The chance that these components would be useful outside the project is not great, but the project may still see significant benefits from this type of reuse. One of the barriers to widespread use of project-developed software parts is that project personnel generally are not versed in the development of reusable software, thus their code may require significant effort to make it usable by other developers.

At another level, reusable software may be developed for a particular domain by a third party. In this case, the software would have broad applicability, e.g., math parts, user interface parts, database parts. Such parts could be used by a wide variety of projects, thus spreading the development cost.

In most of this manual, we assume that reuse is being practiced somewhere above the project level, e.g., parts are being developed at the company level for use by projects.

8.3 Parts Development Organizations

There are at least three alternative types of parts development organizations; they are depicted in Figure 8-1

- An independent or autonomous parts group, i.e., a group that develops reusable software independent of any particular project

- A project-level parts group, i.e., a group within a particular project or program that develops reusable software
- A project-directed parts group, i.e., an independent parts group that solicits inputs from and coordinates with projects about needed reusable software

Each approach has pros and cons. The first approach, i.e., an independent parts group, is able to concentrate reusable software development expertise in this group and thus, increase the quality of the components that are produced. The problem with this approach is that there may not be enough interaction with the projects who are the ultimate customers, thus, components may be produced that will find little use within the projects they are aimed at.

A project-level parts group will generally be able to produce components that are useful to and needed by the project on which they were developed, but they may find little use outside of that project. Additionally, this type of parts group may have little experience developing *reusable* software, and may lack the resources for parts development.

A project-directed independent parts group may be the best approach if the resources are available. With this approach, the group is able to develop expertise in reusable software production, while maintaining close ties with the projects who are the ultimate customers of the reusable software. This group, by virtue of its ties with the projects, will be able to develop a better sense of the reuse potential of candidate software parts. This should ultimately result in high quality, usable parts.

8.4 Configuration Management

Configuration management of reusable software is essential. It is critical that uncontrolled changes not be made to reusable software parts. Decisions to change parts that are under configuration control should rest with a "configuration control board." Errors should always be corrected. Enhancements/changes need to be evaluated on a case-by-case basis. Changes to parts and the reasons for those changes should be documented; this is often done in a software development file or notebook.

One way to facilitate configuration management is to place code libraries under the control of a parts librarian, granting read access to all developers/users, and granting update privileges only to the librarian. This will help eliminate the problem of "too many cooks in the kitchen." The librarian would then be responsible for baselining all components.

When a component is placed under configuration control, all files pertaining to that component (not just the code files) should be placed under configuration control. This includes the files listed in Table 8-1. The code component should then be compiled and the object code tracked.

If a part requires modification, the librarian should "reserve" the files needed to make the modifications. The files are "checked back in" when the modifications are complete, the part is successfully retested, and the source code, documentation, and SDF are updated. When rebaselining a modified component, the modified files should be placed back under configuration control; new files, if any, should be placed under configuration control; the modified part should be recompiled and the object code again placed under configuration control. Any other parts whose compilation is dependent upon the newly compiled part will also need to be recompiled.

Configuration control of reusable software is particularly important because of the widespread use that the software may see. There must be a mechanism for notifying users when errors are discovered in reusable

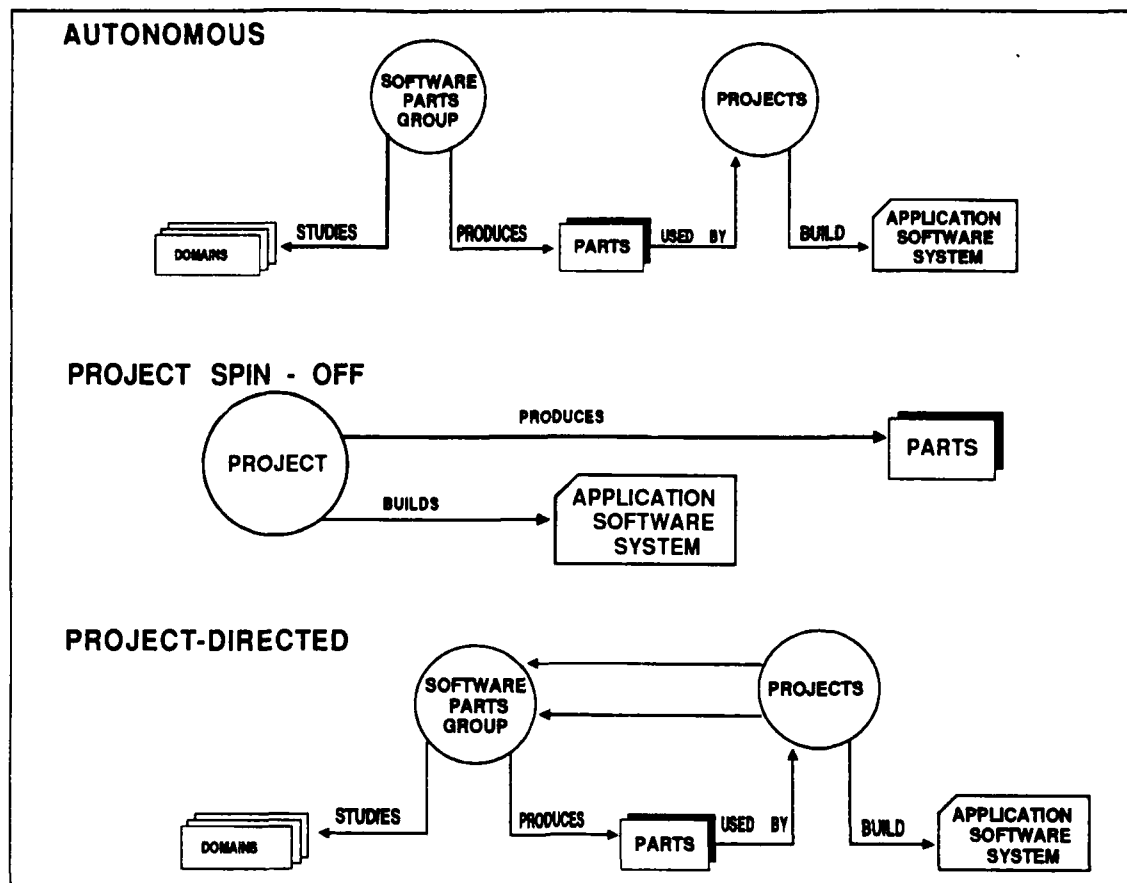


Figure 8-1: Alternative Parts Development Organizations

Table 8-1: Items Under Configuration Control

- All source code files for component
- Test procedure
- Test plan
- All test code files
- Input data for tests
- Expected results for tests
- Test results
- Test utility files

software components. Other than this, configuration control requirements for reusable software do not really differ from configuration control requirements for other types of software.

8.5 Maintenance

Maintenance of reusable components is of vital interest to those who may reuse software components. This was the number one concern voiced at a conference on software reuse (Reference [39]). The scope of a reuse effort has a direct bearing on the maintenance issue.

Maintenance can be performed by the parts development organization, by the parts user, or by a third party. The cost of reuse increases if parts users must also maintain the parts; maintenance will require them to become familiar with the inner workings of the reusable components.

If maintenance is provided outside of the using organization, there are special maintenance considerations that must be kept in mind:

- Current holders of the parts must be notified when changes are made to the parts set (particularly in the case of errors).
- There should be a feedback mechanism so that holders of the parts can provide inputs to the developers.

8.5.1 Modification of Parts

During the lifetime of a part, requests may surface for both enhancements and corrections to reusable components. Because it is anticipated that parts will see greater use than custom code, it is important that changes be performed in a controlled and orderly manner. As previously mentioned, changes should be controlled by a *configuration control board* (CCB). A CCB should contain representatives from all affected groups. For example, on the CAMP program, the CCB consisted of the program manager and the heads of the application task team and parts development team.

The purpose of a CCB is to review the requests for changes and determine the action to be taken. Requests for changes can be categorized as (1) unique to the requesting project, (2) narrow in scope, (3) broadly applicable, or (4) an error. The CCB's decision should be based on the following factors:

- The scope of the change: Is it a minor change or a major one? Is it specific to the requesting application or general enough to be relevant to other applications?
- Purpose of the change: Is it to correct an error (errors should always be corrected) or provide an enhancement?
- Resource constraints

There are several possible outcomes:

- The baselined parts can be modified. This course of action is appropriate if it is determined that a modification is appropriate not only for the current application, but also for other applications. For example, during the CAMP program, all the Kalman filter packages were modified because it was found that the generic parameters did not allow sufficient flexibility.
- Additional parts can be created, i.e., parts variants can be created.. The algorithms for some of the parts may make assumptions that are not appropriate for the current application. For example, the CAMP navigation parts take advantage of the fact that, for small angles, the sine of the angle is approximately equal to the angle itself. This assumption increases efficiency by eliminating the need to calculate an arcsine and produces satisfactory results for some missile applications. This assumption, however, was not appropriate for one particular application and potentially not for others as well. Consequently, new parts were created which used the arcsine instead of the approximation.
- The user can modify his own versions of the parts. In some cases, the required modifications may be specific to that application, and therefore, do not warrant modifications to the baselined parts. In these instances, the user can modify his own versions of the parts as required. This would be the required course of action if no maintenance were provided with the parts. Additionally, it has been advocated that an application development team should always take control of the version of the parts that they use. This has both pros and cons. On the positive side, it gives the development group control over changes to the parts. On the negative side, it can increase their costs by forcing them to become intimately familiar with the parts if a change is needed.

8.5.2 Proliferation of Part Variants

Proliferation of parts with minor variations should not be encouraged as it becomes difficult for users to determine which variant best fits his requirements. This is problematic because, in most cases, requirements specifications for the parts are not captured via a formalism that allows all of these variations to be accurately identified, thus it becomes difficult to distinguish between similar parts. Some researchers have suggested that families of parts be cataloged, and that a mechanism be provided that will generate the variants, rather than individually cataloging each of the variants (Reference [11]).

8.6 Ada Language Considerations

Reusability includes the ability to transport code between different machines and the ability to transport code between applications. In the case of Ada, standardization on a single language and prohibition of subsetting or supersetting of the language have largely achieved the goal of machine transportability. Complete Ada applications have been transported between widely disparate machines, with minimal changes to the source code. The success of this type of reusability is, of course, limited by machine dependencies of the code and the type of application involved. Reusability goes beyond transportability to encompass new applications rather than just new platforms.

The Ada language supports, but does not guarantee, software reuse. Ada was designed to support sound software engineering principles and includes features that facilitate the development of reusable components, such as generic units, separate compilation, and the package structure. There are portions of the current Ada standard that inhibit the ability to transport and reuse code between applications; some of these were discussed in Reference [31]. Not all of the problems are with the language itself; some are with the compilers and the validation process. For instance, most Ada compilers lack global optimization, and certain constructs tend to generate inefficient object code (e.g., generics, tasking). The compilation errors encountered with validated Ada compilers also suggests that the Ada Compiler Validation Capability needs to perform more stringent testing; it is important that all features of the language be adequately tested before a compiler is validated.

Several types of problems have been identified.

- There are instances in which the language standard leaves to the compiler implementor key decisions which can affect the ability of a compiler to handle code developed for reuse (e.g., object code generation of generic units).
- The Ada validation capability does not adequately test all of the standard Ada features which are required for implementation of reusable software (i.e., validated Ada compilers can generate incorrect object code or fail to produce object code).
- The Ada language standard lacks certain specific features which could further enhance reusability, especially for the design of special interfaces (e.g., the ability to pass procedures)

Suggested changes to the Ada language were presented in Reference [30]. These and other proposed changes are being addressed by the Ada revision process.

The complexity of the Ada language has been sited as an impediment to software reuse. It has been suggested that the wide range of choices provided the user results in difficulty in incorporation of reusable code in new applications. That is, it is difficult to predict the implementation approach that will be taken in any given application, and hence it is more difficult to develop reusable components that will meet a user's needs (Reference [17]). This problem can be at least partially ameliorated by getting developers to consider available reusable components early in the development process and by strict adherence to standards during both parts development and application development.

8.7 Tools

The correct tools can play a critical role in enhancing software development productivity. This is true in the development of both custom and reusable code. Tool support for the development of reusable software is basically the same as tool support for the development of custom software, except in the area of domain analysis; tool support for domain analysis was discussed in Chapter 3, Paragraph 3.8.

One area that bears special mention is the area of documentation. Automated document production tools that operate on a centralized data dictionary can facilitate the development of documentation for software systems that incorporate reusable components. If reusable components are used "as is", their documentation can generally be referenced rather than reproduced. If the components require modification, updated documentation will also be required. An automated documentation system operating off of a data dictionary can make this task much simpler than manual documentation production. The software developer would only have to enter the modifications into his version of the data dictionary in order to see them realized in the final documents.

SECTION II

APPLICATION OF REUSABLE SOFTWARE

CHAPTER 9

PARTS USE OVERVIEW

Software developers are increasingly faced with complex and changing requirements, staffing shortages, escalating software costs, projects that cannot be completed within schedule and budget, and products of questionable quality. There is no one solution to all of these problems, but software reuse has the potential to at least partially ameliorate them. The introduction of software reuse as part of the solution poses additional challenges to the software developer, i.e., it is a change from "business as usual".

Software reuse is feasible, but in order to maximize the benefits of reuse a systematic approach must be taken. Reuse must be planned rather than just allowed to happen in an ad hoc manner — ad hoc reuse has been practiced for many years without solving the so-called software crisis.

Reuse must be kept in mind from the very earliest stages of the software/system lifecycle. In fact, reuse should be considered during the proposal and pre-proposal stages of a project rather than waiting until development begins. It is generally too late to make widespread use of reusable software components if they are not considered until the coding phase of a project — by then, too many decisions will generally have been made that preclude the use of many available software components. The extent to which reusable components can be used in an application will be affected by decisions made during the design of both the reusable parts and the new application.

Although there are potentially significant benefits to software reuse, there are also many unresolved issues in parts-based application development. Developers need to be aware of the risks and issues associated with a parts-based approach to software engineering. Early identification of the issues can help a project avoid problems later (Reference [30]).

Potential reusers need to be aware that the use of certain language features alone does not make software reusable, e.g., generic code units are not necessarily reusable. The danger of this misperception is that an application developer may unwittingly reuse code that was not designed with reuse considerations in mind and have a very negative experience with "reuse". Other items developers need to consider are the following:

- The limitations/maturity of the compiler being used, i.e., can it handle the constructs used in reusable software component?
- The extent to which the available parts will actually meet the project's needs
- The efficiency of both the parts and the compiler with respect to the project's requirements
- The maintenance responsibility of the parts' supplier

Software parts can be used in a variety of applications: new full-scale engineering development (FSED), benchmark development, prototypes, or as examples for new custom code development. In the material that follows, we assume the parts are being used in an FSED project, although in reality there would be little difference in the required activities.

In the remainder of this section, each of the major software development lifecycle activities will be examined, and the impact of reuse-based software development on that activity will be discussed. The remainder of this section is organized as follows.

- **Planning for Reuse-Based Development:** This chapter discusses the impact of software reuse on project planning.
- **Requirements Analysis:** This chapter covers the effect of software reuse on requirements analysis for a new application.
- **Preliminary Design:** This chapter covers unique features of a preliminary software design effort that includes reusable software parts.
- **Detailed Design and Coding:** This chapter covers the aspects of detailed design and coding that are unique when software reuse is a part of the software development process.
- **Testing:** This chapter covers the special testing considerations of applications incorporating reusable software.

CHAPTER 10

PLANNING for REUSE-BASED DEVELOPMENT

Software reuse is not yet commonplace, thus not all of the reuse-related project planning issues can be definitively addressed. Some areas that require special consideration are identified below.

- There is always the possibility that parts anticipated for use won't work out. How can the developer account for this risk factor?
- How can the developer account for the cost of identifying, locating, analyzing, evaluating, and incorporating reusable components into the new application?
- What will it cost to train software engineers in effective software reuse? How long will it take them to "come up to speed"?
- Will the benefits of software reuse be realized on the first project? If not, how can the additional cost be justified to the customer? And to management?
- What additional planning is needed for effective software reuse?
- What additional tools are needed to facilitate and enhance software reuse?
- What are the contractual ramifications of software reuse?

10.1 Cost Factors in Reuse-Based Software Development

A reuse- or parts-based approach to software development can result in substantial savings, but there are also associated costs, thus, it must be considered during the cost estimation process. The real-time embedded software community currently lacks sufficient historical data for estimating the cost of reuse-based projects. Consequently, our existing cost estimating models may not be adequate. In costing a project that will incorporate reusable software parts, we need to consider not just the lines of code that will not have to be written, but also the cost of reuse. Assuming the availability of well-developed reusable components, we need to account for at least the factors listed below. Other factors may result in additional costs or savings.

- Cost to locate potentially applicable reusable component (RC) sets
- Cost to evaluate those components
- Cost to locate and retrieve specific reusable components
- Cost to modify and incorporate the RCs
- Savings of not developing the RC
- Savings of not having to document the RC and develop new test procedures, code, etc.
- Savings in integration
- Savings in maintenance

These factors are difficult to estimate, hence it is not straightforward to develop a cost estimate for a project that is planning to incorporate significant amounts of reusable software. Research is underway in this area at several organizations (e.g., see Reference [20]).

10.1.1 Factors Increasing Cost

If the reusable software components were not developed in accordance with rigorous documentation, coding, and testing standards, then additional costs may be incurred. For example, if it is necessary to document inadequately documented reusable software, the cost of reuse is increased. In mission critical applications, it will be necessary to test the reusable parts at some level even if they have previously been tested. The cost of

this activity will be lower if test procedures and code are available with the reusable parts. If it is necessary to develop test procedures and test code, the cost of reuse may go up sharply because the developer will have to determine what to test and how to test it. Differences in development standards (e.g., coding or design standards) between reusable parts and the new application can also increase the difficulty and cost of using parts in a new application. The parts may require modification, or additional code may be needed in order to interface the new application code with the parts. An investment is required to train software engineers in developing software that incorporates reusable parts.

10.1.2 Factors Decreasing Cost

Most of the factors that decrease the cost of applications development when a reuse-based approach is used have previously been discussed. They include items such as less code to develop, less testing and rework, less documentation, and lower maintenance costs resulting from the use of rigorously tested and previously used software.

10.1.3 Weighing the Factors

The benefits of reuse do not *always* outweigh the costs. If a significant effort is needed to modify a reusable component in order to obtain the required functionality or to bring a part up to project standards, it may not be cost-effective to reuse in that instance. The investment may be warranted if there are expected payoffs during maintenance.

The perceived cost of reuse is as important as the actual cost. A software developer must perceive the cost of reuse (whether it's code, code templates, design, etc.) to be less than the cost of reproducing the components himself (Reference [26]). Some researchers (Reference [43]) have postulated that if the perceived cost to reuse, including time to modify code that is not quite a fit, is greater than 70% of the perceived cost to develop from scratch, an engineer will develop custom code for the application. Software engineers tend to over-estimate the effort to reuse code and under-estimate the effort to develop code from scratch. Others (Reference [21]) claim that even if you break even in development time, you still come out ahead because of the increased quality of the software that results from using presumably well-tested software parts.

A means is needed to calculate the cost of reuse for a given part (in addition to determining project-level cost estimates for reuse-based application development). This will provide projects with data needed to make a decision whether to use a part and modify it to meet their application, or develop from scratch. There has been some work in this area (References [10] and [35]), but much remains to be done to develop effective and accurate estimation techniques. As yet, there is little firm data on the cost of reuse.

One thing is known — reuse is not free. For example, if 20% of the code for a new application is expected to come from reused parts, that does not mean that development costs for the project will decrease by 20%:

- Coding generally consumes only about 20% of a project's resources, thus, a 20% code reuse level will generally not result in 20% cost savings for the project.
- There are costs associated with reusing software, such as the cost of identifying, locating, and examining it, and the cost of testing and integrating it with the custom-built code.

10.1.4 Payback from Reusable Software

It is too early to determine exactly when the payback will occur in software reuse. There is some data available in commercial, business-type applications, and some estimates of the benefits of reuse for mission-critical applications have been made, but there is not yet sufficient data available for RTE applications to determine the

extent and timing of the payback for software reuse. Although there will be gains during software development, and greater gains will occur as support environments mature and the cost of reuse decreases, we must also look to the maintenance phase for payback. There we should see increased reliability in the products because of the use of extensively tested software parts. Tracz (Reference [41]) reports that "Maintenance cost reductions of up to 90% have been reported when reusable code, code templates, and application generators have been used to develop new systems." (Note: These results were not necessarily for RTE applications.)

We need to explore ways to maximize reuse and reduce its cost. Cost reductions can be accomplished through training and through the development and deployment of tools that support a parts-based approach to software development. Some of these issues were explored during the CAMP program, including prototype development of an integrated set of facilities to support parts-based software development (References [29, 30]). Other work has been done in these areas as well, e.g., STARS (Reference [23]), Army RAPID (Reference [42]). These issues are discussed in Section III, *Maximizing Software Reuse*.

10.2 Management Support

Management and customer support for software reuse is critical to its success. These two groups may have many of the same types of concerns about software reuse. Customers should realize that with software reuse there is the potential for higher quality products at lower cost and in a shorter delivery cycle than when the product is custom crafted. Management needs to realize two things:

- They can develop a competitive advantage by developing and using reusable software in their product lines (assuming that commonality exists and can be developed in a reasonably cost-effective manner).
- The industry as a whole can produce higher quality products if the industry as a whole develops and uses software parts.

Once the benefits of software reuse become apparent, it is not clear that the DoD or any other "umbrella" organization will need to provide incentives to encourage software reuse — the organizations will see that reuse is in their best interests. In the interim, it may be necessary to provide incentives to developers because of the increased cost of developing reusable software and the increased risk of incorporating it into new applications. The risk will diminish over time as the technology and methodologies that support reuse-based development emerge and mature.

The conversion to a parts-based approach to software development will require cultural changes as well as technology development. Software engineers are not taught to reuse software, they are taught to reinvent it. As Jean Sammet pointed out in Reference [36], *"Ever since the second square root routine was written, the programming field has lost adequate control of reusability."* The conversion of software developers to a reuse-based approach must begin with their coursework and be encouraged by management guidelines and practices.

The effort required to develop high quality software needs to be viewed as an investment rather than as an afterthought in the overall system development. Once the investment is made, the development costs can be amortized by reusing the software in additional applications.

Project managers need to have confidence that when they are asked to "sign up" to software reuse in their projects, that they have a reasonable chance of success. Once they are convinced of the value of software reuse, they need to be willing and able to make the investment to train their software engineers in both the

development and use of reusable software. Management needs to be willing to provide the resources needed to develop a support group that can assist projects in the transition to parts-based application development. Projects are constrained by their schedules and budgets and cannot be expected to fully embrace a technology that is still risky because of its early stage of development. Management and customers must be willing to share the risk with these projects.

Some of the typical risks and concerns of management are enumerated below.

- How do we know that the reusable components we think we can use will really work in our application (i.e., if we base our bid on the assumption that 20% of the software will be derivable from existing software components, what do we do if the parts don't actually work out?)
- What if there are errors in the parts? Who will maintain them? If the project has to maintain them, the cost of reuse is increased.
- What are the legal ramifications of reusing software developed outside of the project.

Confidence in software reuse technology and methodologies can only be gained from success stories. Paper studies and "toy" applications will not suffice for the projects that have tight schedules and requirements constraints. Realistic test programs can provide the type of environment where many of the issues or problems associated with software reuse can be worked out, and it can be demonstrated that it is viable, cost-effective, and ultimately beneficial. The DoD can facilitate the move to software reuse by funding these types of demonstration projects.

The CAMP 11th Missile application development is an example of this type of effort. The goal of this effort was to validate the concept of parts use in real-time embedded applications. The 11th Missile application entailed the redevelopment of missile guidance and navigation subsystems in Ada, using the CAMP parts. This effort was based on an existing application that was implemented in JOVIAL. It required the development of DoD standard documentation, as well as software development and hardware-in-the-loop (HIL) testing. It was targeted to a MIL-STD-1750A processor. The existence of the HIL test environment made this an attractive application to parallel. The development effort demonstrated that Ada was rich enough to allow implementation of virtually all of the required functionality (the application required only 21 lines of assembler code to accommodate existing system idiosyncrasies) and also demonstrated that reusable software parts could be incorporated into a new RTE development effort. The 11th Missile development effort also highlighted the risks associated with the application of new methodologies (i.e., software reuse in RTE applications). Although the 1750A-targeted Ada compiler was validated, many problems were encountered that would have devastated the schedule and budget of a production project. Most of these problems resulted from compiler immaturity; a few of the types of problems encountered are enumerated below.

- Many of the generics had to be *manually instantiated* because the compiler could not correctly process the complex generics used in the reusable software parts.
- The tasking overhead was prohibitive.
- The compiler was not able to generate object code that was efficient enough for the navigation subsystem to run in real time.

The problems encountered and the results obtained during the 11th Missile application development effort are detailed in Reference [31] and summarized in Appendix II. Although many valuable lessons were learned from this effort, it also highlights the importance of establishing testbed programs that can act as pathfinders and facilitate maturation of the required technology and methodologies.

10.3 Additional Lifecycle Activities

Figure 10-1 depicts the software development activities within the context of system development. Planning for software reuse needs to occur early in the software development process.

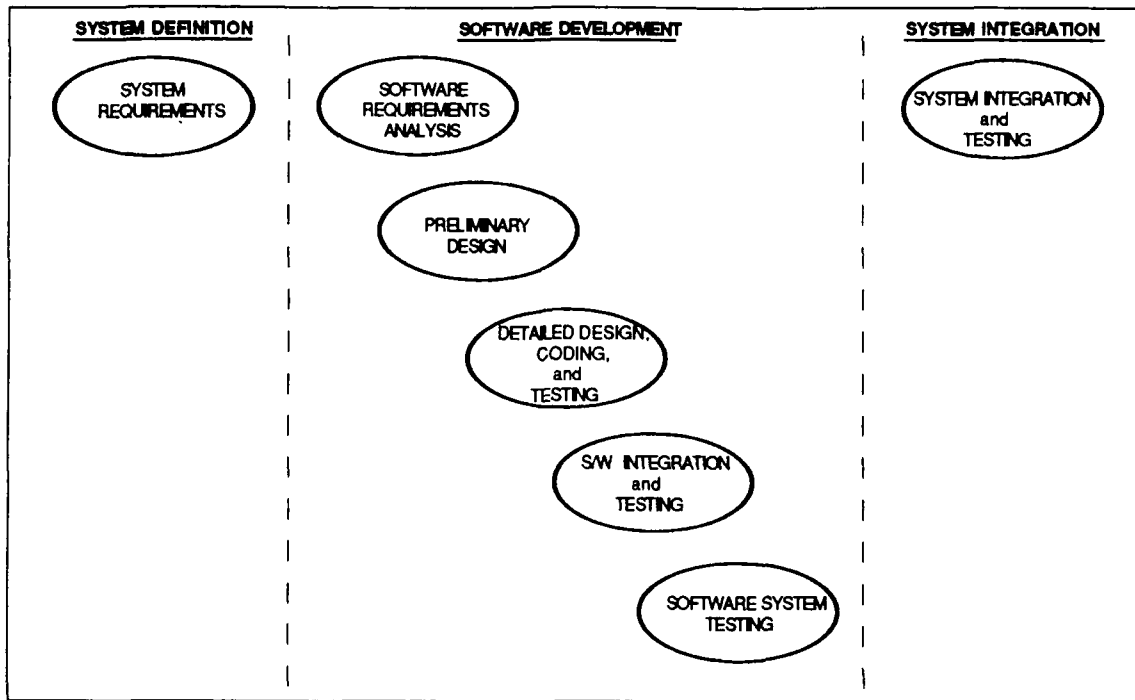


Figure 10-1: Software Lifecycle Activities

A software development plan should identify what the project is going to do with respect to software reuse. In fact, DOD-STD-2167A (*Defense System Software Development*, see Reference [16]) specifically requires the developer to call out the use of *non-developmental software* (NDS); this is any software, including *reusable software parts*, that is incorporated into the deliverable software but that is not developed on the project. Obviously, not all of the reusable software that will be used may be known at this early stage in the project, but the plan can outline a general approach to software reuse, such as the sources of components, what will be done to evaluate the components, and the target range of reusable components that will be incorporated into the new application (e.g., 10% or 50% or 90%). The fact that NDS is called out in the SDP highlights the importance of early planning for software reuse.

There are three major additional activities that occur during a reuse-based application development effort:

- Refinement of the domain analysis (if there was one)
- Project-specific commonality analysis and component development
- Application of existing reusable software components (this includes identification, evaluation, and incorporation of existing components)

10.3.1 Refinement of the Domain Analysis

Refinement of the domain analysis should include an analysis of the new application to determine if it fits the existing domain model. If it does not, refinements or additions may be needed to the model, as well as to the accompanying parts set. If there was no initial domain analysis and domain model development, a preliminary analysis and model can be developed based on the new application. As additional applications are developed, they can refine this "first-cut".

10.3.2 Project-Specific Commonality

During the course of development, projects need to examine the reusable components that are available, and use those that will fit reasonably well into their applications. They should also identify commonality within their application, as well as commonality between their application and others that may be developed within their organization. That is, essentially, they should revisit the domain analysis and try to identify additional candidates for reusable software components. Identification of commonality within the new application and development of components to "cover" this commonality are often overlooked in the planning stages, although they frequently take place on an ad hoc basis. The components developed to satisfy application-level commonality may be either locally or globally reusable, i.e., they may find use only within the current project or they may be reusable across a broad spectrum of projects. This was also discussed in the guidelines in Paragraph 5.6. Locally reusable components should be submitted to a project library for use by other project personnel. Reusable components with broader applicability should be submitted to a higher-level parts catalog or library for possible use by other projects.

Component libraries can and should be put in place at different levels within the software development community and within organizations. For example, a DoD-wide library may contain parts that are of general interest and usefulness (e.g., window and menu managers, plus domain-specific parts for domains such as avionics or harmonics), while a company library may have parts that contain proprietary algorithms and parts that are more specific to a particular company's product line. A project library may contain parts that are widely used within a particular project, but that may not be widely applicable outside of that particular project.

As a project or reuse group develops reusable software components, they need to be propagated up to the next higher level component library in order to get them into circulation and accessible by other projects. This should lead to the development of a rich and useful parts set that will meet the needs of future application developers.

Although projects are frequently the best source of reusable software (or, at least, of ideas for reusable software), they frequently do not have the resources to develop parts themselves. Ideally, a software reuse support group would be available to assist in the development of the newly identified common components, thus decreasing both the risk and cost to the project.

10.3.3 Application of Reusable Components

The first step in the application of existing software components to a new application effort is to identify parts sets that may be available in-house, commercially, or from other sources, and determine the domains they cover and which of them bear further investigation. For example, if avionics software were being developed and the only parts sets found dealt with banking applications, there would be a strong contra-indicator to parts use for that particular project, whereas if missile flight software parts were located, they would bear further investigation because of the overlap in the two domains. Once potentially applicable parts sets and parts are

located, additional information can be obtained on the developers, reason for development, the type of maintenance that will be provided, use restrictions (Are the parts copyrighted or in the public domain? Can they be used freely or are there other restrictions in their use?), and the cost, if any, of getting the parts. This, ideally, should be done prior to software development (i.e., during the proposal stage, or during system requirements development).

Figure 10-2 illustrates a parts use and development cycle. The TOMEX project, which is a new development effort, will initially go to an existing software library and obtain applicable components. During the development effort, certain portions of the application may lend themselves to development as reusable components. These components will initially reside in the project library, but may eventually be submitted to the next higher-level library (e.g., a company library) for use by other projects.

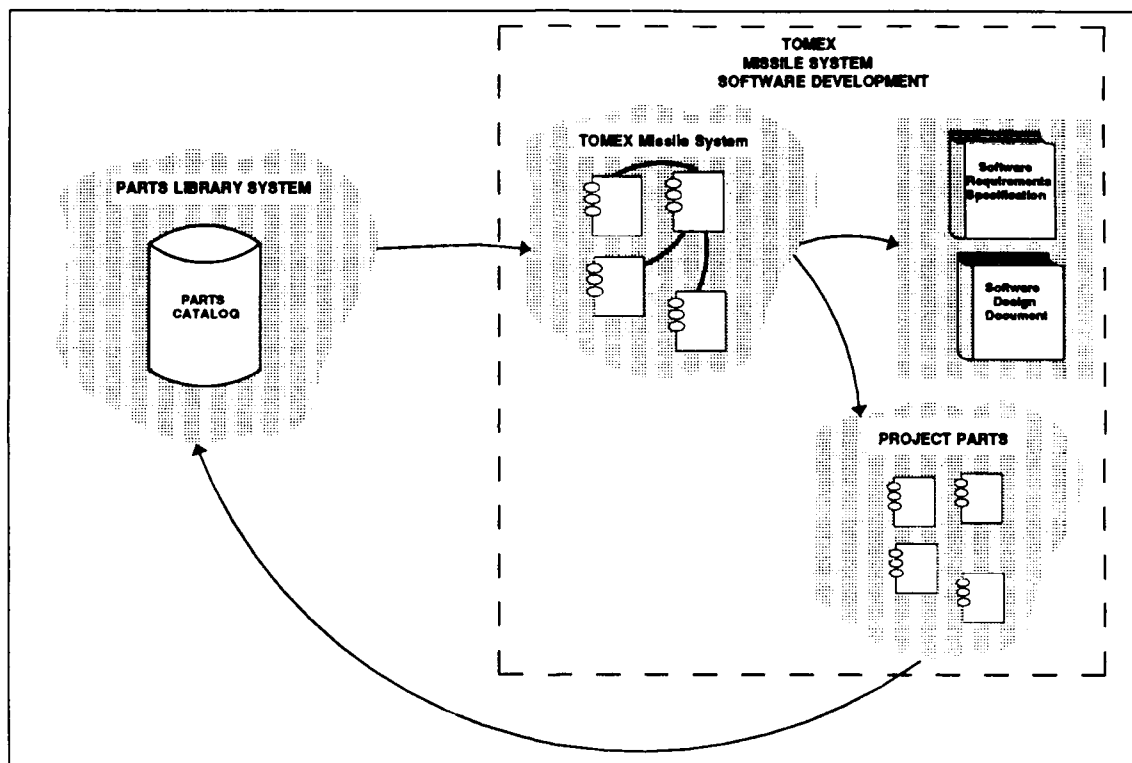


Figure 10-2: Parts Use and Development Cycle

Many issues may arise during parts use. For example, if parts from more than one source are used, do they need to be integrated? Do they have a compatible structure? Who is legally responsible for the correctness of the parts? Are there proprietary rights that need to be preserved when using the parts? Are the parts covered by distribution/export restrictions? How should reusable parts that are incorporated into a new application be documented? How should they be tested? Management may have some concerns about incorporating reusable software into a new application. For example, if the contract is "time and materials," they may think profits will be lowered if they incorporate reusable software because the development effort may not take as long, and may not require as many people. Once software reuse becomes established in the software development lifecycle, project development time will be shortened, allowing more projects to be undertaken.

Another management concern is that software reuse may seem risky, particularly initially. There may also be concerns about quality and the fact that the current developers are staking at least part of their reputations on someone else's software.

10.4 Pitfalls in Software Reuse

There are several pitfalls to be aware of in parts-based software development. Most of these issues are discussed in greater detail in subsequent chapters of this section.

- Immature compilers: Compilers used with parts-based application development must be able to properly handle the constructs used in the reusable components. This requires up-front analysis of the compiler before a project commits to software reuse. These compilers must also perform needed optimizations. For example, if the compiler does not optimize out unused code, an application may become too large for the target system. This is because reusable code components may have more operators, etc. than are needed by the current application, resulting in a significant portion of the reusable code being unused.
- Compiler differences between the development system and the target system: Again, a project needs to verify that both the development compiler and the target compiler will adequately handle the reusable software that is planned for inclusion in the new application.
- Reuse of code that is not really reusable: Certain programming language features can facilitate the development of reusable code, but there is no guarantee that all software that incorporates those features will be reusable. For example, a generic unit is not necessarily reusable; it may have too many environmental/application dependencies built in. This can lead to less than desirable results with code segments that developers think are examples of reusable software.
- Incompatible modifications made to parts in use: If the reusable components are accessed from a central library rather than put under project control, they could be modified in ways that would make them incompatible with the current application.
- Performance/functionality "surprises": If an adequate evaluation of candidate reusable components is not performed prior to use, there may be costly "surprises." For example, components that appeared to be close matches requiring only simple changes may, in fact, require major rework to fit them into the application.

10.5 The Role of a Reuse Support Organization

The role of a reuse support organization is dependent upon the scope of the reuse effort. It can range from someone simply entering component information into the library to a significant staff that not only checks code and documentation for compliance with standards, but also performs independent testing, assesses the value of parts, assists users in the use of parts throughout the lifecycle, develops new parts, and provides training. As the scope broadens, the need for, and demands on, a support staff grow. Such a staff could be invaluable in the technology and cultural transition from custom code development to widespread software reuse. They could provide the training and support needed to alleviate risk to projects that are considering a parts-based approach to software development.

In general, some of the basic functions that such a group can perform are as follow:

- Assist users in locating reusable software resources
- Assist in the use of reusable software
- Adapt project-developed software for reuse
- Maintain the reusable software (i.e., make error corrections, enhancements, etc.)
- Develop additional reusable software identified either by projects or by additional domain analysis

- Obtain feedback from users in order to fine-tune the software parts collection and the library
- Provide training in parts-based software development

10.6 Tools for Reuse

Tools that support reuse-based application development can decrease the cost of reuse and increase productivity. Although there are many types of tools that can support parts-based application development, a software repository, library, or catalog is an essential cornerstone in any software reuse effort. It provides a central location for information about available software components, and can be used in virtually all activities throughout the software development lifecycle (see Figure 10-3). Libraries are discussed further in Chapter 17.

A reusable software repository can facilitate —

- Identifying candidate software parts for use in a new application
- Retrieval of software parts
- Use of software parts
- Evaluation of candidate software parts. It may contain information about other projects that have used the parts and their experience with them, as well as performance data and operating restrictions.
- Notification of users about changes to the parts

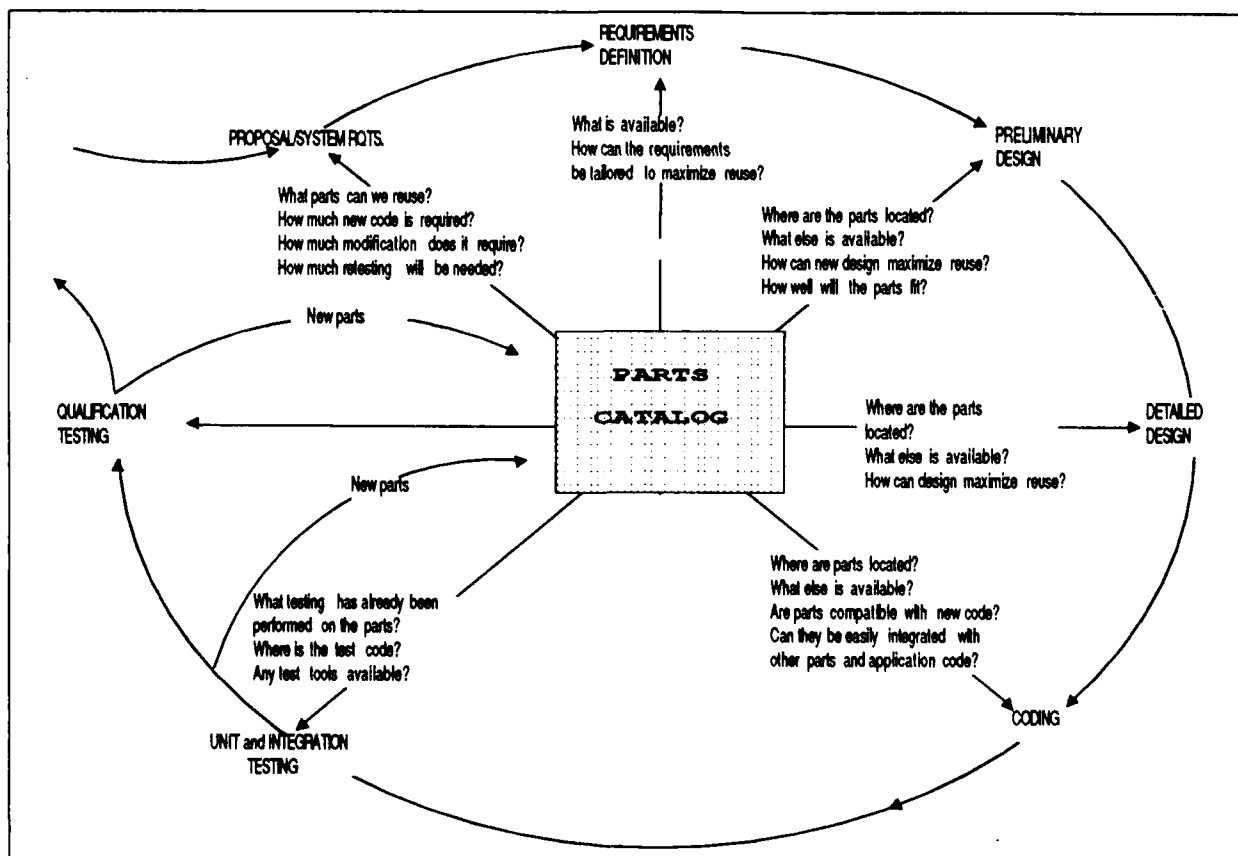


Figure 10-3: The Role of a Software Library in the Development Lifecycle

Tracz (Reference [41]) points out that special tools are *not* needed to get started in a reuse program; a basic catalog or database of parts will suffice. Almost all researchers have indicated that cataloging tools and techniques, and a means of describing software are essential to the success of software reuse from the end-user's point of view. Lubars (see Reference [26]) has said that "the most significant technical barrier to code reusability, is the problem of finding the desired piece of code." This problem decomposes into 3 steps: "(1) realizing that the code exists..., (2) finding the general location of the code ..., (3) finding the particular location ... of the code" The Japanese have had fairly good success even with simple keyword indexing of available parts. As component libraries increase in size, increasingly more complex mechanisms may be needed to locate and evaluate components. Parts cataloging and attribute development continue to be active research areas (see Reference [18] for examples of different approaches).

Tools that assist in the analysis and evaluation of reusable software components, and software composition tools that facilitate the integration of reusable components with custom code will also be useful in a parts-based application development effort. Tools that assist with documentation, particularly tools that eliminate redundancy in document production and automate the document production process will make software reuse more attractive. Detailed discussion of tool support for reuse-based application developed is deferred until Section III, *Maximizing Software Reuse*.

10.7 Compilers for Reuse

Although compilers are software tools, they warrant special attention. When a decision is made to systematically reuse software on an application, it is critical to determine if the selected compiler can properly handle the constructs used in the reusable software components. With Ada, compiler validation is necessary but it is not sufficient to ensure correct compilation of Ada source code. For example, on the CAMP parts usage demonstration, the development team used a validated Ada 1750A cross-compiler and found that the compiler was *not* able to correctly handle many of the CAMP Ada generic parts. These were parts that contained valid Ada syntax and that had been tested extensively on another validated compiler. This situation significantly increased the effort required to complete the task — many of the generic units had to be *manually instantiated*, test cases had to be developed and submitted to the compiler vendor, code had to be retested as new versions of the compiler were received from the vendor, and finally, code had to be modified when it became apparent that some of the problems were not going to go away. This seems to be primarily a maturity problem and will undoubtedly become less of a problem as Ada compilers mature. Projects with tight schedules and budgets would do well to benchmark their intended compilers **BEFORE** the project begins, and to develop a close working relationship with their compiler vendor.

Benchmarking for specific capabilities is a valuable way to identify compiler inadequacies with respect to reusable software, although benchmarks emphasizing the needs of software reuse in the RTE community are not prevalent. During the CAMP program, a benchmark suite was developed to test Ada compiler performance in the area of complex generics. It consisted of a series of compilations (based on CAMP packages) which require the compiler to process complex uses of Ada generic units (see Reference [12], as well as Chapter 16). These and similar tests are needed in order to verify that the compiler can at least correctly compile the types of reusable components that may be used.

10.8 Maintenance and Configuration Management

Maintenance of reuse-based applications differs from maintenance of custom software in that there may be some question as to who performs the maintenance. Once changes are made to reusable parts, the original developer is generally no longer responsible for the correct operation of the code. In fact, he may never have been responsible for maintenance.

If errors are uncovered in an application, it will first be necessary to determine if they occurred in the reused code or in the new code. If the error was in reused code and the code is supported, the trouble reports will go to the parts maintenance organization, otherwise the development group will need to provide maintenance. If the reusable software is supported, the following criteria can be used for deciding when to modify the part (i.e., take the code and change it yourself), build your own code, or see the parts development team and request a part enhancement:

- If the requested modification is specific to the application under development, then the application team should modify the part as needed or build a new part to meet their needs.
- If the proposed enhancement may have broad applicability, then the reuse support team should make the modification, and release the new version of the part for general use.

It is important to carefully document any project-specific changes that are made to reusable software parts because (1) if errors are uncovered during testing or deployment, this documentation can help pinpoint whether the problem resulted from the original part code or from tailoring; and (2) the application developer may want to suggest changes to the parts developer based on changes that were required for the application.

A major issue that arises in parts use is whether a project should obtain its own copies of the reusable software and put them under project configuration management, or if it should, if possible, compile out of the central software reuse library. Although there is conceptual appeal to working out of a central library and avoiding duplication of code, etc., the reality of the situation is that it is often best for a project to take control of their own copy of the reusable software that they plan to use. This provides them with the freedom to modify the parts as they see fit, relieves them of having to rely on a third party for updates, and protects them from "undesirable" modifications to the parts.

Two situations that need to be considered when deciding whether or not to "take control of the parts" are enumerated below.

- If a project has obtained its own version of the reusable software parts that are being used in its application and new versions of those parts are released by the supplier, how will the new versions be handled? What if the project has made changes to their versions, but the newly released versions contain desirable enhancements or error corrections?
- If a project has not obtained its own version of the parts, and "new and improved" versions are released that supercede the versions in use on the project and that are not completely compatible with the project's application, what will be done?

If automated tools are used to customize reusable software (e.g., component constructors), the application developer should generally go back to the tool for maintenance and modifications rather than making changes manually. If the changes are made manually, it may be cost-prohibitive to ever go back to the original tool to regenerate the code.

CHAPTER 11

REQUIREMENTS ANALYSIS

Software reuse can play a dual role during requirements analysis activities. Reusable software components can be used in the development of prototypes for verifying requirements, as well as in the development of the requirements for an FSED application. In this chapter, the concentration is on the impact of a reuse-based approach on application development rather than on prototype development.

During system requirements/design, system requirements are allocated to software, hardware, and firmware; preliminary software and interface requirements may be formulated at this time, as well. During software requirements analysis, those requirements should be finalized and documented. DOD-STD-2167A (Reference [16]) calls for the development of a *Software Requirements Specification* (SRS). The interfaces that are external to the major software components (in DOD-STD-2167A terms, this is a *Computer Software Configuration Item* (CSCI)), should also be identified and their requirements documented; the *Interface Requirements Specification* (IRS) is used for this. From the software development perspective, the output from the requirements analysis activity is the full set of engineering requirements for each CSCI in the system.

Requirements should be kept free from design decisions. Although this guideline should not be new to anyone, it is not always followed, and it is particularly important when a reuse-based approach to software development is used. Software reuse will be maximized by delaying design decisions until the requirements are finalized, and information can be gathered on available parts and be used to drive the design. Introduction of design decisions too early in the development process can result in a significant decrease in potential reuse for the project. This is because many design decisions will preclude the use of some parts, although the parts themselves may actually meet the requirements.

The major costs associated with software reuse during software requirements analysis activities are those of (1) locating parts sets, (2) identifying potentially applicable parts within those sets, and (3) identifying commonality within the application.

11.1 Parts Identification

It is appropriate (and potentially advantageous) to begin parts identification during the proposal stage, but it is not too late to start during requirements analysis. Parts identification cannot be delayed until coding! Although it may sometimes be possible to incorporate some reusable software during coding, software reuse will be maximized if it is considered much earlier. By the time a project reaches the coding stage, too many decisions have been made that can interfere with the incorporation of parts. If parts are identified early in the development process, the design can incorporate them from the beginning rather than requiring retrofitting to incorporate them later. In reality, parts identification will take place throughout requirements analysis, architectural design, detailed design, and coding.

During requirements analysis and definition, available parts sets should be identified and examined for potentially applicable parts. There is no single source of information on available parts sets. Parts sets may be available from commercial developers, from various government agencies, from private organizations, or from in-house parts developers or other projects. Additionally, standard algorithms for various functions have been documented in books and journals and can be implemented as utility parts sets. Examples of available parts sets include the EVB GRACETM parts, Booch parts, CAMP parts (for the missile operational flight software domain), and other software parts that have been developed as part of the STARS foundation area work (in-

cluding editors, window and menu managers, etc.). Once the parts sets are identified, parts within those sets should be identified for potential applicability to the current development effort. Early identification of parts is needed so that —

- Realistic cost estimates can be formulated
- Requirements can be developed that maximize reuse. For example, in the selection of navigation axes, a developer may decide to use an east-north-up frame of reference rather than an east-north-down if there are reusable software parts to support the former but not the latter.
- Information about available parts can be used to sway design in a reuse maximizing direction

There is some risk associated with early identification of *potentially* applicable parts — the potential may not be realized, thus requiring the development of custom code to meet the system requirements. This is a major reason why parts must be bundled with sufficient information about them to facilitate the formation of an informed decision early in the development process. The software developer cannot be expected to read every line of source code before determining whether a part can, or should, be used.

11.1.1 Tracking Parts Use

In examining parts sets for potentially applicable parts, it is necessary to correlate software requirements with available parts. This can be done either manually or automatically if the appropriate tools are available; conceptually there is no difference. Manually, this can be done by producing a correlation table or by marking up a copy of the software requirements document. This should be started during software requirements analysis, and will continue through preliminary and detailed design. The correlation table can be produced by going through relevant parts sets with the requirements document and determining if there are parts that may meet a requirement, either fully or partially. In a fully integrated CASE environment, the mapping could be automated so that as the user specified his requirements, the available parts sets would be searched for corresponding parts.

At this point, the correlation between requirements and parts is preliminary — the parts identified at this point may or may not actually work out. The amount of effort expended on this activity depends on the project schedule and budget. The more work done up-front, the greater the potential of maximizing reuse and minimizing risk. Although the list of parts produced may not be a final usage list, it will help in later phases when more information is needed on the parts.

11.1.2 Tool Support

Support tools can significantly decrease the cost and increase the success potential for would-be software reusers. In the ideal world, the mapping between requirements and software parts would be fully automated, so that as requirements were entered, they would be mapped to available software parts. The use of formal requirements specification languages has been advocated by some researchers as one means to facilitate automation of this mapping. Formal languages provide a more rigorous representation that would lend itself to automated query formulation. The major drawback is the high cost it imposes on the software developer — he must be proficient at the specification language, as well as understand the problem domain and know the implementation language.

Other forms of tool support can be provided that span the gap between a completely manual process and full automation. One of the major efforts during the CAMP program was a study of tool support for software reuse. This included the development of the concept of a *Parts Identification*, or *Parts Exploration* tool that provides the user with high-level access into a parts catalog; it is intended for early identification of potentially

relevant parts — it solicits requirements from the user and maps these requirements to available parts; it is discussed in more detail in Chapter 18.

11.2 Compiler Analysis

Projects must not only identify parts for reuse, they must also verify that their compilers can properly handle those parts. It is critical that this verification take place as early as possible in the project. One "discovery" made during the CAMP parts usage testbed effort (i.e., the 11th Missile development effort), was that a validated compiler is not necessarily the same as a correct compiler. The 11th Missile development team experienced compiler problems that would have devastated a "real" project with hard deadlines. Although validated Ada compilers were used (more than one was tried), these compilers were unable, in most cases, to handle the complex Ada generic units that were used in the reusable code. Many of the generics had to be *manually instantiated*, i.e., they were *un-genericized* and made into non-generic subroutines. In their generic form, the compiler either failed to compile them at all or generated incorrect code. The CAMP Benchmark Suite (see Reference [31]) contains compilation benchmarks that can be used to assess the adequacy of a compiler for handling generics which are essential for reusable Ada software. These tests involve the compilation and instantiation of a representative set of the CAMP parts.

Developers should also determine how unused code is handled by their compiler and linker/loader. It may frequently be the case that not all of the subroutines in a reusable code part are required in an application. In real-time embedded software systems it is undesirable and usually unacceptable to have unused code included in the object module. If the unused code is not removed automatically, the developer will have to remove it manually. This will increase the cost of reuse.

Much more work needs to be done in the area of benchmarking for reuse, as it is essential that a project be aware of the capabilities and limitations of a compiler before planning to incorporate reusable software. This is discussed in both Chapters 10 and 16.

11.3 Documentation

Although the reusable parts will be used to meet specific requirements that are documented in the software requirements document for the new application, the parts themselves must also be documented (either by the original supplier or by the project using them). If high-level parts are being reused and their requirements are documented in an project-acceptable format, it may be possible to reference the requirements document for the reusable part, rather than reproduce the requirement in the document for the new application. One of the benefits of reuse is that the software should already be well-tested and documented. The need to (re-)document reusable software components acts as an inhibitor to reuse.

11.4 Summary

During requirements analysis, the main questions that the developer needs to answer with respect to software reuse are as follow:

1. What parts sets are available?
2. What domains do these sets cover?
3. What functionality is provided by these parts?
4. Do they appear to meet project requirements?
5. Can the compiler handle these parts?
6. Is there commonality within the current application?

Figure 11-1 illustrates a typical scenario for requirements analysis activity under DOD-STD-2167A development. The developer generally will have a system specification document and a software development plan available at the beginning of software requirements activity. He may also have preliminary software and interface requirements documents. During the requirements analysis activity, the developer must produce a full set of engineering requirements for the software. This will occur regardless of whether the development is parts-based or not. The additional tasks that will be required to accommodate reuse include (1) locating available reusable software, (2) initiating a preliminary analysis for applicability and cost to reuse, (3) benchmarking compilers to determine if they can adequately handle the types of reusable software that are being considered, and (4) identifying commonality within the current application.

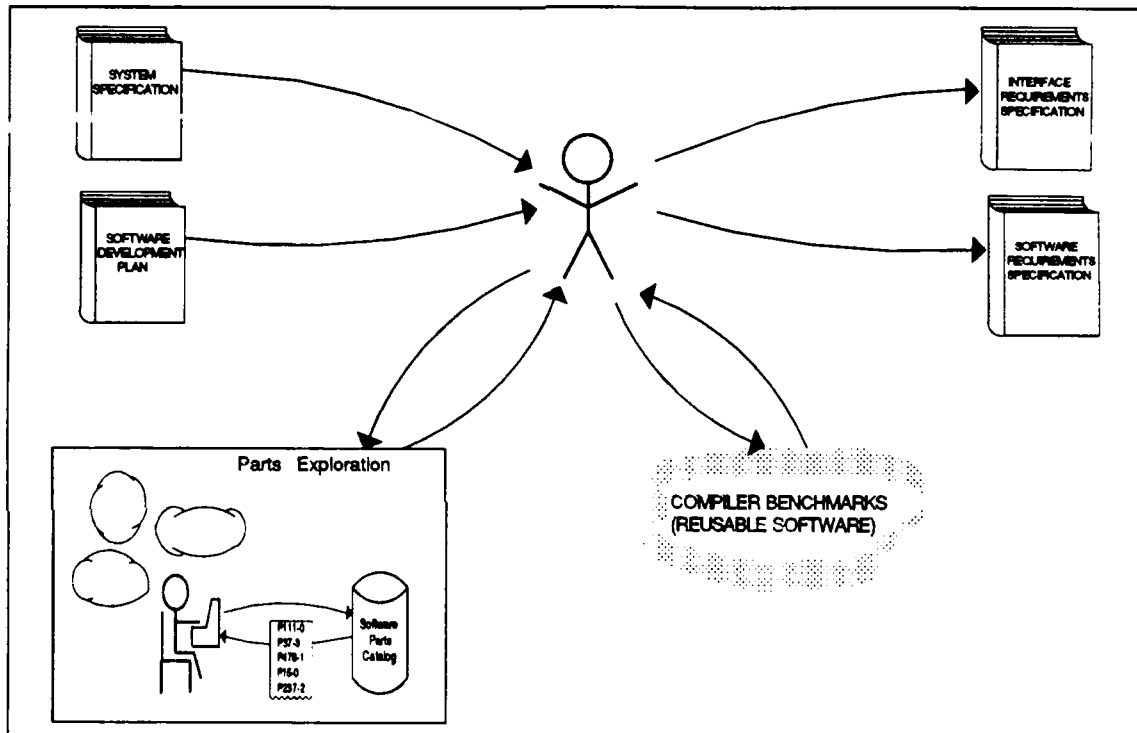


Figure 11-1: Reuse and Requirements Analysis

CHAPTER 12

PRELIMINARY DESIGN of PARTS-BASED APPLICATIONS

During preliminary design, the requirements documents are used as the basis for allocating requirements to actual software components. In terms of DOD-STD-2167A (Reference [16]), the top-level design is developed for each CSC (a CSC is comprised of CSUs, or Computer Software Units) and documented in a *Software Design Document* (SDD), and the top-level design of the external interfaces is documented in the *Interface Design Document* (IDD). During the design process, the integration test requirements for the top-level components should be developed and documented in their respective software development files (SDF). Formal qualification tests (to verify compliance with the requirements) should also be identified and documented. This would be documented in a *Software Test Plan* (STP) under DOD-STD-2167A.

The extent to which reusable components can be incorporated into the design of a new application is, in part, dependent on the architecture of the parts themselves. As previously discussed, parts that are designed for ease of use, protection from misuse, tailorability, and flexibility will generally find more use than parts developed without these types of goals in mind. For example, one design approach that has been used to meet these goals is the layering of Ada generics — as long as the interfaces are maintained, the user can replace virtually any part with custom code that he has implemented. This makes the part easy to use (if the user accepts the part "as is", there is little data that he has to supply), tailorable (he can replace portions of the reusable software with his own custom implementation while still benefitting from reuse of the higher level structure), and flexible (the parts can be used in multiple situations, not all of which may have been anticipated by the component developer).

Identification of potentially applicable reusable software components, which was begun during requirements analysis, will continue during preliminary design. The designer must verify that the parts that have been identified can indeed fit into the application or that the architectural design can be changed (without changing the requirements) to be compatible with the available parts. It is during this time that the *potentially* applicable parts need to be examined in-depth to determine if that potential will be realized.

Tools can facilitate this examination and analysis. Some types of software tools that may prove useful include the following:

- A catalog for obtaining detailed information about the parts
- A source code library so that code components can be examined
- Parts benchmarking software
- Other analysis aids (e.g., complexity analyzers)

The parts source code may be available directly through a catalog, otherwise an alternative means must be provided for the software developer to obtain the source code. Ideally, parts developers should supply either benchmark results or benchmarks so that sizing and timing data can be obtained. Analysis tools will facilitate determining whether the available components meet sizing, timing, and complexity requirements. Reuse support tools are discussed in greater detail in Section III, *Maximizing Software Reuse*.

12.1 Parts Analysis

During preliminary design activities, there should be an in-depth analysis of the parts previously identified, i.e., the analysis should go beyond an examination of catalog entry information. In Ada software development efforts, the package specifications are often considered the top-level design, thus, analysis of parts at this level often equates to analysis of Ada package specifications and associated documentation. Parts should be examined with respect to the following items:

- Closeness of fit between the parts and the requirements of the new application. The parts should be examined to determine what, if any, modifications are needed to make the parts work in the new application.
- Project coding, documentation, and quality standards
- Availability of test procedures, code, and data
- Reputation of the part developers
- Previous users' experience with the parts
- The architecture of the parts versus the architecture of the application under development

12.1.1 Closeness of Fit

The application designer needs to examine the reusable components to determine how well they will fit into his design. For example, are the functional requirements met by the reusable component? If so, does the design fit into the new application? Are the performance requirements met?

The designer must determine if he will be able to use the parts "as is" or if he will have to modify them. If a part does not fully meet a project requirement, the designer will need to determine the capability that will need to be added, deleted, or modified. Modification can range from a several line code change to full replacement of one or more package bodies. If modifications are required, the designer must determine the point at which to switch from modifying the existing part to providing his own package that acts as an interface between the reusable component and the custom code, or starting from scratch. If the existing part does not fit well into his design, but meets at least some of his requirements, he will need to evaluate the alternative costs of "scavenging" what he can from it or making it fit into the new application.

Examination of closeness of fit between parts and project requirements should include analysis of performance requirements, such as speed/space efficiency and accuracy of generated results, and a determination of whether these requirements are satisfied by available components. Several approaches may be taken:

- Benchmarking
- Inspection of the parts code itself
- Rapid prototyping

Benchmarking, or the review of benchmark results, can play an important role in the evaluation and selection of reusable parts for inclusion in an application. A number of parts developers/researchers have recommended that benchmark results be included in a parts catalog. This adds significantly to the cost of *developing* the parts, but can reduce the cost of *using* the parts because the application developer may be less likely to select an inappropriate part.

If benchmark data is not available with a part, the potential reuser can benchmark the part himself. This obviously increases the cost of reuse, but it may be necessary if benchmark data is considered critical in part

selection and is not available for the parts under consideration. Even if the parts have to be "tweaked" in order to use them in a new application, it is generally worthwhile to use reusable code because of the increase in reliability and the decrease in effort over the lifecycle to produce the final product. Reuse is not an all or nothing proposition!

Code inspection of the parts can identify differences in architectural approach between the parts and the new application. Prototyping affords an excellent opportunity for parts evaluation, but is often not an option for designers. In an ideal world, parts could be used in prototypes to determine exactly how they work before they need to be used in an actual application. Code analyzers and standards checkers can be used to check conformance or compatibility of candidate components to project design standards.

During this analysis process, the perceived cost (vs. the real cost) of reuse plays an important role. Although the true cost of modifying the reusable component may not be any higher and may, in fact, be lower than developing code from scratch, it is difficult to quantify the costs precisely. Developers generally under-estimate the cost of custom development, and over-estimate the cost of modifying reusable components.

There has been research into the development of techniques for automating the evaluation process and assisting in costing the use of a given part (see References [10] and [35]). Factors such as closeness of project and parts' requirements, and amount of modification needed to get the part to fit, play a role in the cost estimate. This was discussed briefly in Chapter 11. When more fully developed, these types of techniques should prove valuable during parts analysis.

12.1.2 Design Differences

As candidate parts are identified for use in an application, it is necessary to check for compatibility between the parts and the new design, and between parts from different parts sets. The impact of design differences between reusable components and the new application is dependent upon both the type of application and where within the application the components are to be used. If parts from various parts set do not need to interface with each other, or if the reusable parts are generally isolated from the custom portions, design differences may have little impact. If there is a need for close interaction among software components from these various sources and there are significant design differences, interfacing may be problematic. It is possible that, although parts from more than one parts set will be used, they may not need to interface with each other and, thus, integration of parts from these various sets becomes a non-issue. For example, if only parts from Parts Set A will be used in Subsystem X, and parts from Parts Set B will be used in Subsystem Y, there may be no need for concern about design differences between Parts Sets A and B. On the other hand, if parts from both A and B will be used in the same subsystem, it may be necessary to determine how well these parts will interface.

Differences in parts that may need to be considered include the following:

- Some parts (or custom code) may be designed with data types defined throughout the code, whereas others (such as the CAMP parts) may have the types defined in "data type" packages that are then *with'd* into other packages as needed.
- If an abstract data type approach has been taken in the parts development, and the application requires direct access to the data structures, some restructuring of the parts may be necessary.
- Strength of data typing between the new application and the parts may need to be resolved. Basically, strongly typed parts can be used with minimal effort by either weakly or strongly typed applications; more effort is required if the application is more strongly typed than the parts. See Paragraphs 5.6 and 5.7.2 for more details on resolving component-application data type differences.

12.1.3 Test Support

Availability of test procedures, code, and data is an important consideration when selecting parts. It has a direct bearing on the cost of reuse and the confidence that the user may have in the parts. Test procedures and code should always be provided along with the source code for reusable parts. This will allow the parts user to rerun the tests in his own environment. Undoubtedly, software parts used in mission critical applications will require retesting of some type. Anticipated test results should also be provided with reusable components. Accompanying documentation should indicate any areas that may be particularly problematic and require special testing attention if modifications are performed on the reusable component.

12.1.4 Development Standards

Software parts can be viewed as falling into two categories: parts that meet project standards (i.e., software that was developed using the same or similar standards that are applicable to the new development effort) and parts that don't (i.e., software whose development standards differ significantly). Differences in development standards may be less problematic if the components are fully and well-documented.

In the former case, no changes are required to code headers or documentation for the reused software; the design documents for the current application can reference the existing design documents for the parts. In the latter case, the differences may first become apparent in documentation — documentation may be either non-existent or inadequate. If this is the case, project standard headers should be added to the component, and it should be fully documented along with the current development effort.

It may be cost prohibitive to enforce coding standards on reusable parts unless the parts were developed in-house. Additionally, changes to the components to bring them into conformance with project coding standards can be dangerous! Once the components are modified (whether for substantive changes or merely to add headers or enforce standards conformance), they are no longer the same as when they were received and may no longer qualify for support (if there was any). Enforcement of coding standards could result in dramatic changes being made to the code. Depending on the customer for the final product, the developer may be able to compromise on development/coding standards given the benefits of reuse.

Quality standards should not be compromised. This is not to imply that all reusable components must be exactly as if they had been custom developed in-house. It does mean that certain standards should be set and maintained in order to preserve overall project quality. The gains from using components of questionable quality will be marginal at best.

12.2 Impact on Design

There are several items to be considered during design of reuse-based applications:

- Reusable components can exist at different levels of abstraction. If the reusable components are generally isolated from other portions of the application, design differences may have little impact, whereas if reusable components are interspersed with custom code, the differences may, in some cases, be problematic.
- Differences in approaches to data typing can cause incompatibilities between components and custom code
- Project-specific commonality may benefit from the development of reusable components.

Each of these items is discussed in subsequent paragraphs.

12.2.1 Levels of Abstraction

Software reuse can impact the design process in several ways. Reusable parts occur at many different levels (e.g., unit, subsystem, system), thus parts can be incorporated into top-level design, as well as detailed design and code; Figure 12-1 illustrates this. The occurrence of multiple levels of reusable components is one of the factors that makes early investigation into available components necessary. The parts identification that was begun during requirements analysis should continue during preliminary design; the parts-requirements correlation table should also be maintained. The top-level design will probably undergo several iterations, with each iteration raising the possibility of greater levels of reuse.

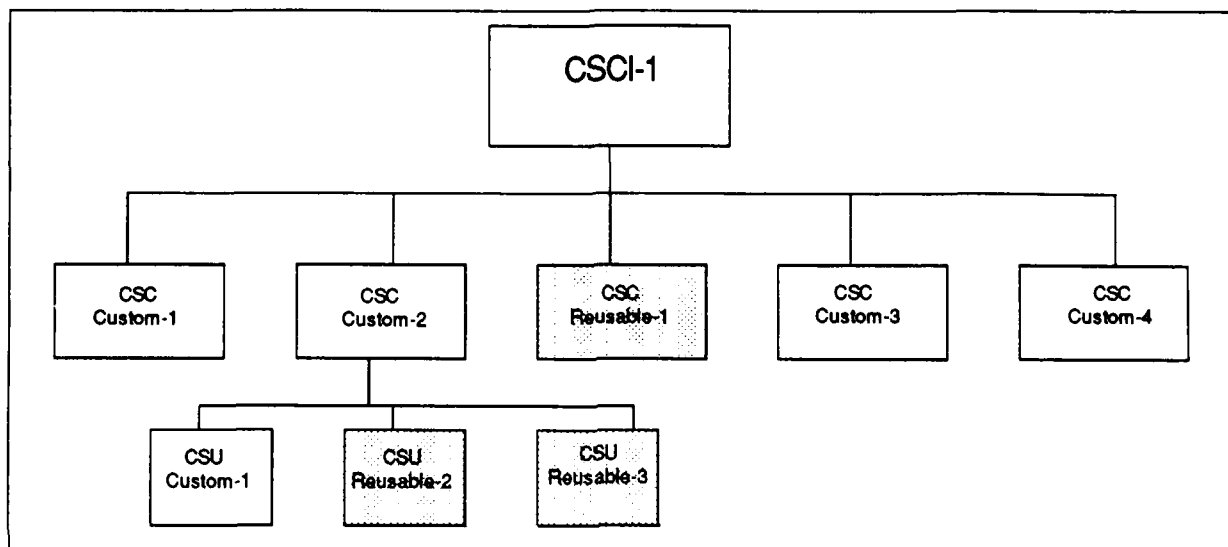


Figure 12-1: Use of Parts at Architectural Design Time

As an example, consider the CAMP Kalman filter parts which are comprised of components at different levels of abstraction. Figure 12-2 illustrates the hierarchical structure of this set of parts. The component at the highest level might first become a candidate for reuse during requirements analysis when the developer first identifies a need for a Kalman filter. Components at lower levels may then be identified as candidates during preliminary and detailed design.

12.2.2 Data Typing

One of the key features of Ada that prevent misuse of software is strong data typing. The issues that arise from the reuse of components that incorporate various degrees of data typing were discussed in detail in Chapter 5. Conversion of data types and development of interface packages between reusable code and new application code can become quite costly, thus, these matters are best given careful consideration during initial design.

12.2.3 Project-Specific Reuse

During the design of the new application, developers should identify portions of their system that lend themselves to development as reusable components. One clue is if they find themselves repeating design information. If the system is very large, this may not be readily apparent. This identification can be facilitated by cross-group reviews, i.e., software designers from different portions of the system should attend reviews in order to help identify commonality within the application. Once the commonality is identified, development should follow the guidelines in Chapter 5, *Architectural Design of Software Parts*.

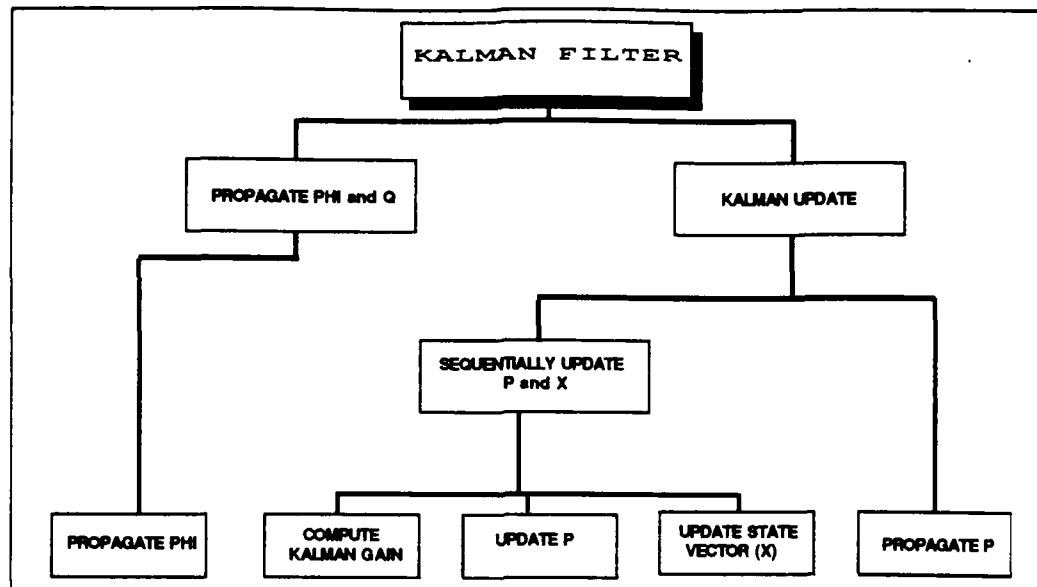


Figure 12-2: Software Parts Use in Missile Software Systems

12.3 Reviews

Architectural designs may undergo several reviews (both internal and external) depending on the size of the project. These may include project walkthroughs of the design of various portions of the application, as well as more formal reviews of the entire architectural design. Design reviews for applications incorporating reusable software need to cover both the reusable software and the custom software. The reviews of the designs for the reusable portions need to concentrate on different issues than those for custom software. Some of the key questions to be considered during reviews are enumerated in Table 12-1. As an example of the difference in emphasis, consider coding standards: reviewers would want to check custom code for conformance, but a decision may have been made that reusable code will be exempt from those standards because of the danger of modifying code and the additional cost it imposes.

During design walkthroughs, reviewers should verify that the design element will indeed satisfy the requirement(s) to which it is correlated. If the parts have previously undergone testing and use, the walkthroughs should be straightforward. Any deviations from project standards should be noted and, if necessary, corrected. During design walkthroughs of reusable software, generally only the portions that will actually be used need to be walked through. Application developers may want to "comment out" portions of the Ada specifications of the parts that will not be used. This will facilitate review, and will eliminate concerns about compilers not optimizing out unused code. On the other hand, the code which is commented out may be needed at a later date, thus requiring additional modification to the reusable components.

The reviewers should pay particular attention to whether reuse is being maximized, and should query the designers about the extent of reuse. This requires some familiarity on the part of the reviewers with available reusable parts sets. If reuse support personnel are available, they should be part of the review team because they, presumably, will be familiar with available parts and should be able to determine if parts are being used to the maximum extent possible. Project personnel are still needed to verify that the project's requirements are being met. If there are customer reviews of the top-level design, developers should be prepared to discuss reuse and defend their parts usage decisions. Care needs to be exercised when reviewing a design that incorporates

reusable software because there may be a tendency to make assumptions about the correctness and appropriateness of reusable software that would not be made about custom software.

Table 12-1: Software Design Review Questions

- Does the expected functionality of the code meet the requirements?
- Will it work correctly?
- Does it conform to project coding standards?
- Is it documented adequately?
- Is it documented according to project standards?
- Does it meet project quality standards?
- Does the algorithm satisfy the requirements?

12.4 Summary

Figure 12-3 illustrates a typical scenario for preliminary design activities and shows the additional activities when a reuse-based approach is taken. Although the documents identified in the figure are DOD-STD-2167A documents, the impact of software reuse on the preliminary design activity is basically the same regardless of the development standard used. During preliminary design, the developer will map requirements to actual software entities. Additional tasks that result from a reuse-based approach to software development are enumerated below.

- In-depth analysis of available parts
 - Coding, documentation, quality standards
 - Availability of test procedures, code, data
 - Reputation of part developers
 - Previous users' experience with the parts
 - Architecture of the parts
 - Closeness of fit between the parts and new application
- Benchmark the parts
- Integrate top-level reusable components into the design
- Identify top-level portions of the new application that can be built as reusable software components

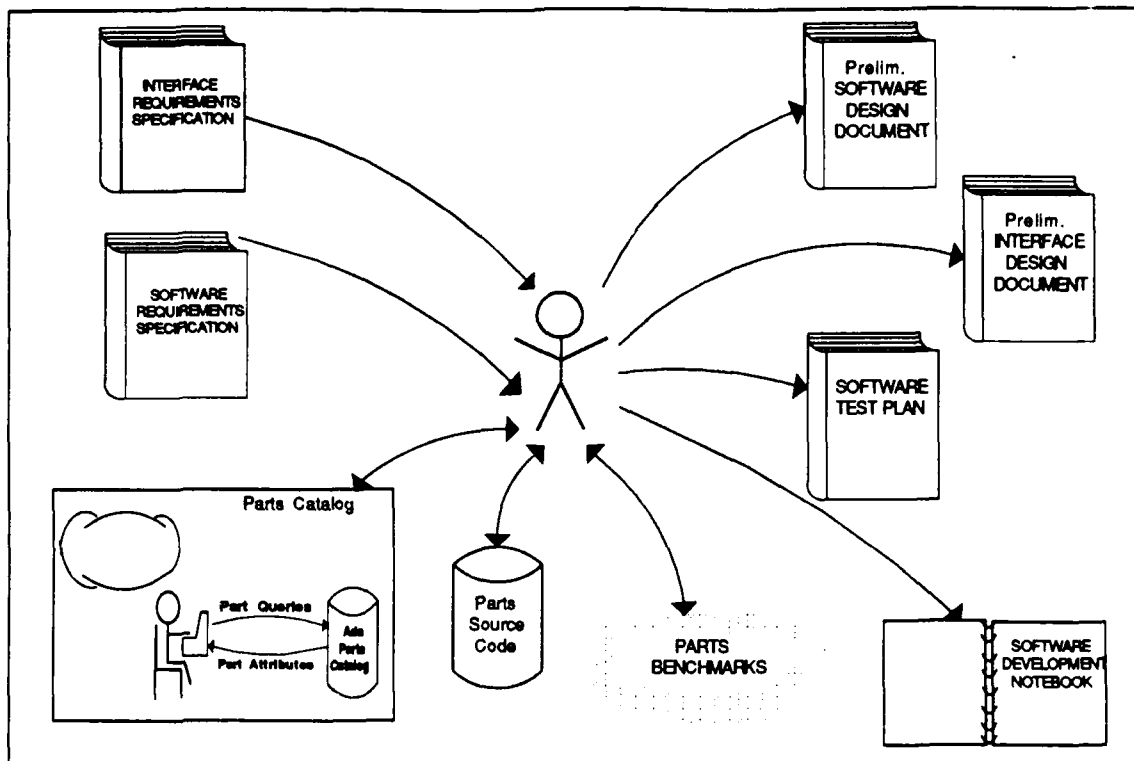


Figure 12-3: Reuse and Preliminary Design

CHAPTER 13

DETAILED DESIGN and CODING of REUSE-BASED APPLICATIONS

During detailed design and coding, the detailed design for each top-level software component (or CSC, in DOD-STD-2167A terms) will be completed. The software design should be documented in some type of software design document. Under DOD-STD-2167A, the external interfaces are documented in an *Interface Design Document*; this should also be completed at this time. Test cases, test schedules, and test procedures for the CSC integration and testing should be identified and documented in the software development file or notebook for that CSC. Test requirements, cases, schedules, and procedures for testing lower level software components (or Computer Software Units (CSU), in DOD-STD-2167A terms) should also be identified and documented in the software development file for the appropriate CSU. The test cases for formal qualification testing (FQT) should also be identified and described. Under DOD-STD-2167A, this would be documented in the *Software Test Description* (STD) for the CSCI. The actual source code is also produced to satisfy any requirements or design constraints that cannot be met with parts and to interface reusable components with custom software.

Ada lends itself well to the development of smaller modules, thus, it is often possible to combine detailed design and coding. If a module is sufficiently complex, then the detailed design and coding activities should be separated. If they are separated, the detailed design should be at a higher level of abstraction than the code. The detailed design/coding activities are a continuation of the preliminary or top-level design process. When detailed design and coding are combined into one activity, the result is the production of the Ada bodies of the previously specified packages.

13.1 Impact of Reuse

Reuse-based software projects may need to take a pragmatic approach and be willing to settle for an implementation that may not be *exactly* as their custom implementation would have been, as long as it meets all of the functional and performance requirements, as well as the quality standards set by the project. No two people will design a system in exactly the same way, thus, it is inevitable that the parts considered for reuse will not be exactly as a project would develop from scratch. It is easy for the designer's first reaction to be, "This won't work." It is necessary for him to get past this and on to the task of determining if the parts are suitable for his application. He can then address the issues previously discussed (i.e., Does it work as is? Does it come close enough so that the required modifications are cost effective? Does it need to be reworked? Does it need a complete overhaul? Does an interface to it need to be built? Does it need to be completely scrapped?). Even if the code is accepted "as is," documentation may still be needed to be brought up to project standards.

When developing a parts-based application, it is important to realize that the reusable parts may contain some code that is not used by the new application. An optimizing compiler should eliminate unused code and data, but if it does not, the user should be aware that he may need to manually remove it in order to achieve the efficiency requirements typical of RTE applications. Even if the unused code is eliminated by an optimizing compiler, the application developer may still want to remove or comment it out to avoid confusion when examining the code, and to eliminate the need to document and test it. It is sometimes helpful to develop a special commenting convention for unused code, such as "-- NOT_USED" to begin each line of a code segment that is unused. This is preferable to actual removal of the unused code because, as the design proceeds, some previously unused code may be needed. It has the additional advantage that a pre-processor could be developed to strip out that unused code. Marking unused code will facilitate review and maintenance because it

will be apparent which routines are being used from a package. There is always a danger in modifying code, even if it is only to comment it out — comment markers could be added to the wrong lines, lines could accidentally be deleted, etc. Even this type of modification can nullify maintenance agreements with parts suppliers.

13.2 Tracking Parts Use

Tracking of parts generally begins during requirements analysis with the development of some form of requirements-parts correlation table. This identifies potentially applicable software components. As application development continues, additional data can be tracked. It may be useful to keep track of where parts are used within an application, and what percentage of the new application is reused software as opposed to custom code. Tracking where within an application parts can be/are used can serve to identify areas for future parts development. Data on lines of code for both the reused code and custom code can provide an interesting comparison with the parts data. It may also be useful to keep track of whether parts are used "as is" or if they require modification. The reason for modification may be significant in later analysis, so this should be recorded as well. Modification may be necessary if a part does not provide the exact implementation that is needed either from a precision, efficiency, or functional point of view. Additionally, some parts may be "skeletal" rather than complete parts. Others may contain "excess" code. Ideally, much of this usage information can be fed back to a software reuse group and can facilitate the analysis of the parts by both the parts support personnel (if they exist) and future application developers.

As an example of tracking parts use on a new development effort, consider the CAMP 11th Missile development effort. The developers kept track of the number of parts that were used (see Table 13-1), as well as where and how many times each part was used (see Table 13-2). The usage status with respect to the 11th Missile effort of each part within the CAMP parts set was also tracked. Each part was assigned one of five codes to indicate its status in the 11th Missile Application. The codes used were follow:

- **U (Used):** The part was used without modification.
- **M (Modified):** The part was used with modification.
- **DF (Duplicate Function):** The part implements the same function as a part that was used by the 11th Missile.
- **NA (Not Applicable):** The part implements a function not required by the 11th Missile.
- **NC (Not Compatible):** The part performs a function required by the 11th Missile, but was not used.

This showed very clearly which parts were used and if they were used "as is" or with modification. It also provided information on why parts were *not* used. Of the 454 CAMP parts that were available at the time, 112 of the parts were used either "as is" or with modifications. The parts that were modified were changed either because they were not intended for direct use "as is," i.e., they were more like templates (e.g., the Kalman filter data types package which contained more data types than an application was likely to need, and data types of a more general nature than would probably be needed), or because they were almost, but not quite, a fit.

By keeping track of parts use, developers are able to determine where in the new application parts usage is concentrated. For example, the CAMP 11th Missile developers found parts usage concentrated in navigation, guidance, and Kalman filter operations. Tracking parts use can facilitate an analysis of why available parts are not used. Again, on the 11th Missile Application, developers found that of the 342 CAMP parts that were not used, 142 were not applicable, i.e., they implemented a function that was not needed by the 11th Missile; 179 parts duplicated the functionality of CAMP parts that were used, e.g., there were two different navigation

Table 13-1: Summary of CAMP Parts Usage in the 11th Missile Application

Parts Used	96
Parts Used with Modification	16
Parts Not used	<u>341</u>
	454
Used 24.7% of Parts	
Used 23.1% of Parts Lines-of-Code	

Table 13-2: Summary of CAMP Parts Used in the 11th Missile Application

Part	Used by LLCSC	Number Used
Basic Data Types	Navigation_Operations	1
Abstract Processes	Guidance_Operations	
Standard Trig and Polynomials	All tasks	1
	Navigation_Operations	22
	Guidance_Operations	
Coordinate Vector Matrix Algebra	Navigation_Operations	11
and General Purpose Math	Guidance_Operations	
WGS72 Ellipsoid	Navigation_Operations	3
	Guidance_Operations	
Common & Wander-Azimuth Navigation	Navigation_Operations	14
Direction Cosine Matrix Operations	Navigation_Operations	14
General Vector-Matrix Algebra	Kalman_Filter	11
	Navigation_Operations	
Kalman Filter	Kalman_Filter	8
Abstract Data Structures	Memory_Manager (N&G)	3
	Kalman_Filter	
Quaternion_Operations	Quaternions	2
Waypoint Steering	Guidance_Operations	8
and Geometric Operations		
Signal Processing	Guidance_Operations	5
Clock Handler	System_Time (N&G)	1
	Local_Time	
Universal Constants	Navigation_Operations	1
	Guidance_Operations	
	Message_Manager (N&G)	
Conversion Factors	Message_Manager (N&G)	5
and Unit Conversions		
Bus Interface	BIM_Interface (N&G)	1
External Form Conversion	Barometric_Altimeter	1
		<u>112</u>
"Used By" is the LLCSC which instantiates or imports the part.		

bundles, but only one was needed in the 11th Missile; and 20 of the parts were incompatible with the 11th Missile Application, i.e., they provided a function that was required by the 11th Missile, but they provided it in a way that was not appropriate to the 11th Missile implementation. Some of the parts were not used because of efficiency concerns about invoking a subroutine.

Data on lines of code for new and reused software should also be tracked. On the CAMP 11th Missile development effort this information was tracked for each of the major subsystems within the application; a sample of this data is shown in Table 13-3.

Table 13-3: CAMP 11th Missile Parts Usage

LLCSC	PACKAGE	LOC PARTS	LOC NEW	% PARTS
Environment LLCSC	Baro_Altimeter	10	52	16%
	OCU	0	133	0%
	TLM	128	409	24%
	ISA	0	254	0%
	SCP	0	42	0%
	IBB	0	79	0%
	Measurements	0	119	0%
	Internal_Bus	0	761	0%
	System_Clock	56	0	100%
	Local_Clock	56	0	100%
	CPU	0	38	0%
	Message_Proc	136	4053	3%
	Mem_Manager	124	52	70%
Navigation LLCSC	Nav_System	0	503	0%
	Quaternions	50	191	21%
	Nav_Ops	1727	2500	41%
	Align_Meas	0	213	0%
	Kalman Filter	1522	4714	28%
Guidance LLCSC	Internal Bus	0	740	0%
	Message Proc	141	975	13%
	Master_Time	56	0	100%
	CPU	0	38	0%
	Guid_Ops	1298	1739	43%
	System Op Data	0	34	0%
	Guid_Computer	0	49	0%
	Mem_Manager	52	124	30%

13.3 Parts Modification

A project may need to modify a reusable component for reasons other than errors, e.g., a reusable component may not quite meet the project's requirements. If the parts are supported, the decision to modify a part should generally be made in consultation with the support group so that it can be determined who will be responsible for the modification and future maintenance. If changes are specific to the application, the application group should make the change. This does not necessarily negate the benefits of reuse because the existing part can be used as a guide to developing the modified version. Additionally, much of the test procedure, code, etc., will probably already exist and just need to be modified. Reuse is *not* all or nothing. In fact, the modifications may be minor changes to enhance performance or functionality and, thus, will be significantly easier than starting from scratch. There are significant benefits to be realized even when parts are not used "as is." If the modification will have broad applicability, the support group should make the modification, and re-release the part.

13.4 Reviews

Although in an ideal situation, it would be possible to treat reusable parts as "black boxes," in reality, particularly with real-time embedded mission critical applications, it may well be necessary to walkthrough all design/code, regardless of whether it is new or reused. Generally, there should be at least one walkthrough for each Ada package; larger packages may be broken down and undergo several walkthroughs. As with walkthroughs of new code, these walkthroughs seek to ensure that the code meets the requirements, interfaces properly with other code, and conforms to project standards. Enforcement of project coding standards on the reused parts is generally not feasible, although the lack of, or inadequate, documentation should be remedied.

13.5 Case Study: Incorporating a Reusable Kalman Filter

This case study builds on the one presented in Chapter 3. Where that one began with domain analysis and went on to discuss some of the problems encountered in parts specification and design, this one discusses the incorporation of the reusable CAMP Kalman filter components in an application.

As previously discussed, Kalman filters play a significant role in missile operational flight software. Basically, a Kalman filter for missile flight operations combines external position or velocity measurements with internal position or velocity measurements, taking into consideration the uncertainty of the measurements when calculating an optimal estimate of missile position and velocity. The Kalman filter also produces an optimal estimate of inertial sensor errors. This is illustrated in Figure 3-9.

The CAMP domain analysis identified two forms of the measurement sensitivity matrix, "H": "Compact_H" and "Complicated_H". The Complicated_H parts assume that the measurement sensitivity matrix is sparse, which allows for a more complex relationship between a measurement component and the components of the state vector. In the so-called Compact_H parts, it is assumed that in any given row of the measurement sensitivity matrix there is only a single "1", with the other elements being zero (this implies that any measurement component is a direct measurement of a single component of the state vector). Five of the Kalman filter parts provide an alternative representation for the measurement sensitivity matrix. (Note: The measurement sensitivity matrix (H) relates the external measurement vector (Z) to the state vector (X) as follows: $Z=H \cdot X$. External measurements can be obtained from radar or barometric altimeters, TERCOM, DSMAC, GPS, or some other external system. The state vector contains estimates of errors in navigation parameters and in inertial sensor data.)

Figure 13-1 contains a portion of the CAMP parts used in performing Kalman filter Complicated_H operations. This collection of components is referred to as a *bundle* for the Kalman filter operations. It includes the Kalman filter parts, as well as the other parts that are needed to support Kalman filter operations, i.e., the "environment". The bundle includes the following generic packages:

- The Kalman_Filter_Complicated_H (KFComplicated) package which exports operations specific to a Kalman filter using a complicated H matrix
- The Kalman_Filter_Common (KFCommon) package which exports required operations common to Kalman filters using either a complicated or compact H matrix
- The General_Vector_Matrix_Algebra (GVMA) package which contains generic packages and sub-routines which export the matrix and vector data types and operations required to implement a Kalman filter
- The Polynomials package which provides various solutions to transcendental functions

- The `General_Purpose_Math` (GPMath) package which exports the square root operator required to instantiate the vector operations packages contained in the `General_Vector_Matrix_Algebra` package: It obtains this square root operation by instantiating one of the square root operations exported by the `Polynomials` package.
- The `Kalman_Filter_Data_Types` (KFDT) package which provides all the data type definitions and subroutines required to instantiate the generic Kalman filter routines: Most of these data type definitions and subroutines are obtained by instantiating packages and subroutines contained in the `General_Vector_Matrix_Algebra` package.

The user implementing a Complicated H Kalman filter from the CAMP parts has several options:

- Use the Kalman filter parts bundle "as is." This would involve *with'ing* the `Kalman_Filter_Data_Types` package into his application and using the data types and operations exported by it to instantiate the routines he requires in the `Kalman_Filter_Complicated_H` package.
- Plug in some of his own code for one or more of the routines contained in the Kalman filter collection. This could be done if the existing subroutines did not meet the user's functional or performance requirements.
- Modify the `Kalman_Filter_Data_Types` package to instantiate different packages and subroutines in the `General_Vector_Matrix_Algebra` package.

Each of these alternative is discussed in the following paragraphs.

13.5.1 Using the Kalman Filter Parts "As Is"

"As is" use of the Kalman filter bundle is illustrated in Figure 13-1. This approach can be taken when the user is satisfied with the data types and operations which are supplied by the `Kalman_Filter_Data_Types` (KFDT) package. It allows the user to get a Kalman filter up and running quickly (e.g., in order to test alternative design decisions, such as the number of states), even if the resulting filter does not meet all of the efficiency requirements of the final application. Use of this approach does not preclude the user from revisiting the code at a later time to increase its space and/or time efficiency.

When constructing a Kalman filter in this manner, the user creates his own Kalman filter implementation package (`Kalman_Filter_Application`) which does the following:

- *With's* the `Kalman_Filter_Data_Types` (KFDT) and `Kalman_Filter_Complicated_H` (KFComplicated) packages
- Uses the data types and operations exported by the KFDT package to instantiate the routines required in the KFComplicated package.

The remainder of the application code can obtain access to the Kalman filter data types and operations by *with'ing* in the KFDT and `Kalman_Filter_Application` packages. This approach is illustrated in Figure 13-2.

13.5.2 Using Custom Code with the Kalman Filter Parts

If the user initially implements his Kalman filter using the "as is" approach described in Paragraph 13.5.1, he may, at a later time, decide to replace some of this code with his own. This may be required to increase efficiency of his application or because the parts do not fully satisfy his requirements. If the modifications do not require altering the interfaces, the part code can be replaced with custom code simply by modifying the subroutine bodies.

When customizing the Kalman filter environment, the user may choose to modify some of the low-level utilities, modify the bodies of some of the Kalman filter routines, or choose to not use one or more of the Kalman filter

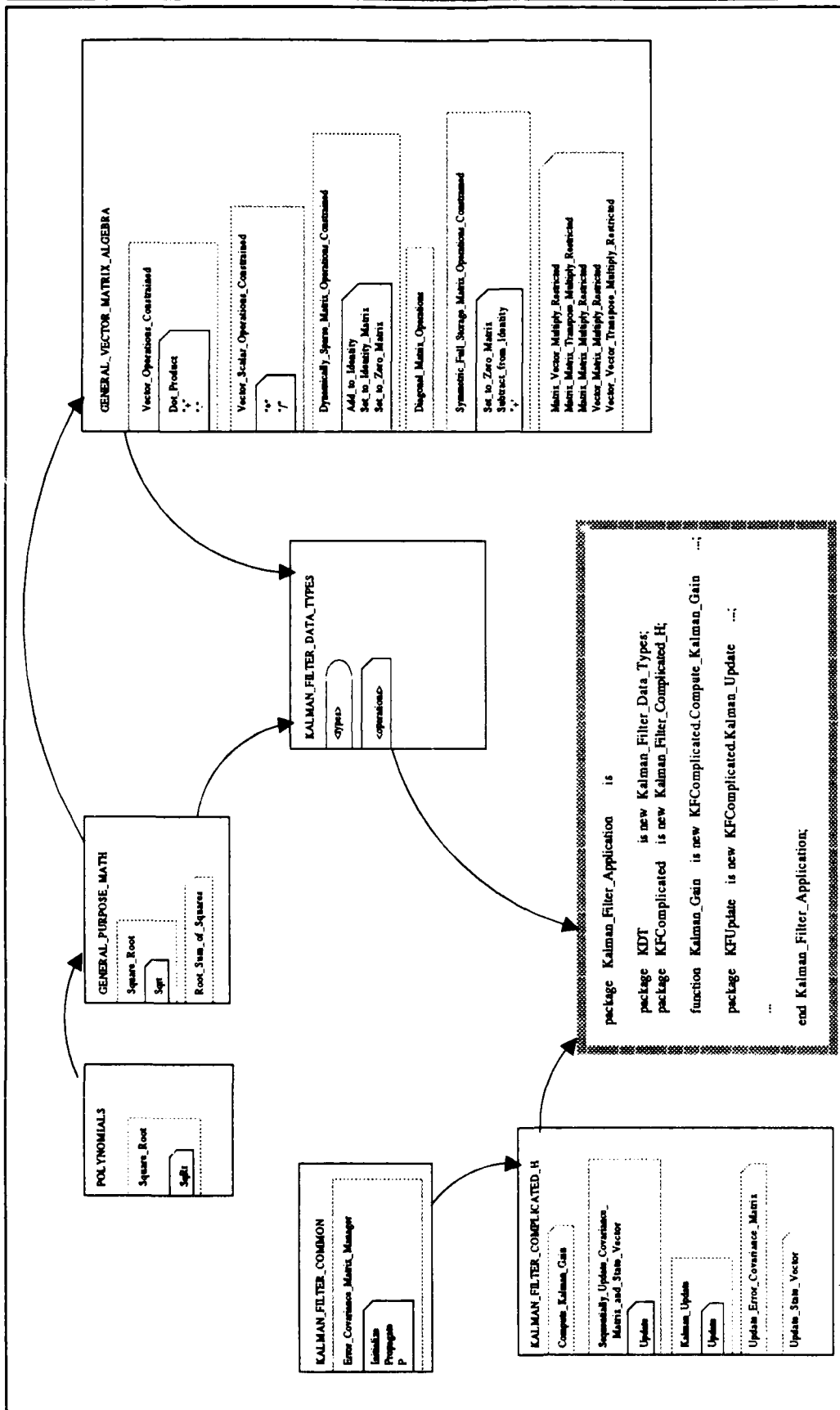


Figure 13-1: Kalman Filter Parts Bundle

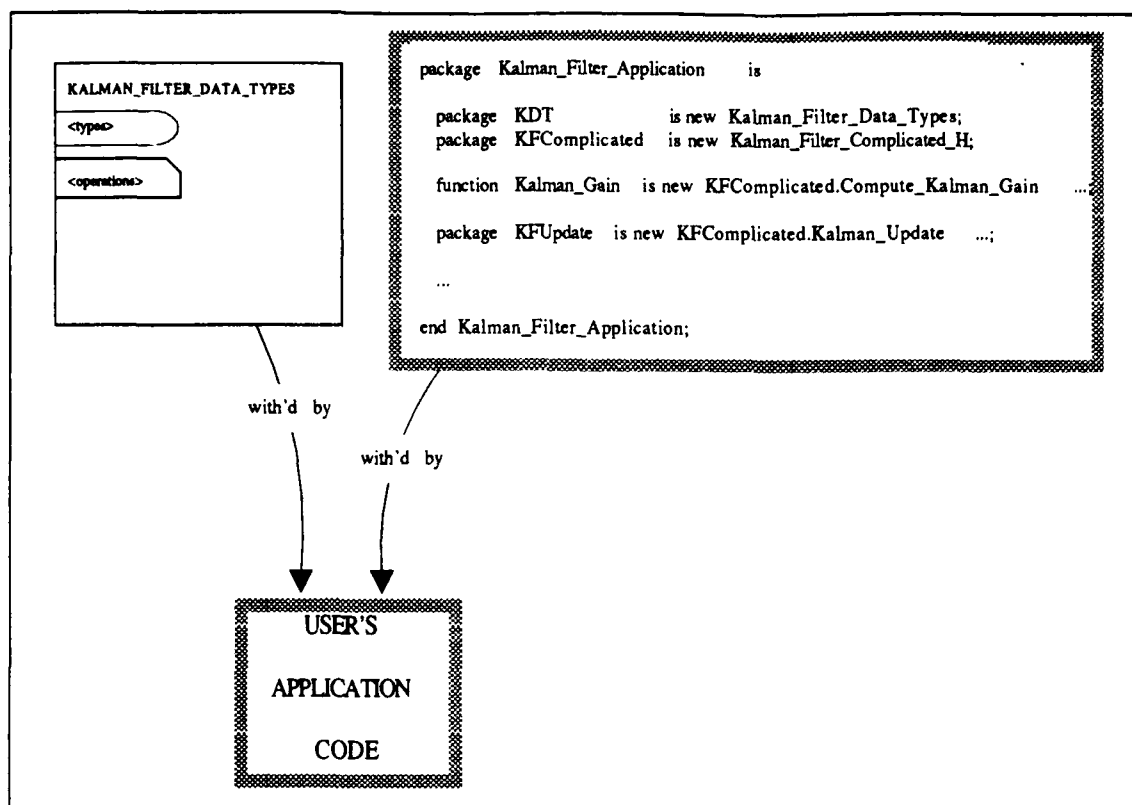


Figure 13-2: Using the Kalman Filter Parts "As Is"

routines and place the custom code in his own `Kalman_Filter_Application` package. These options are discussed in the following paragraphs.

One of the utilities in the Kalman filter environment is the square root function which is exported by the `General_Purpose_Math` (GPMath) package. If this square root function is not appropriate for the user's Kalman filter application he has two choices. If it was inappropriate for his entire application, he could modify the square root function in `Polynomials` which is instantiated by GPMath, he could modify GPMath to instantiate a different square root function in the `Polynomials` package, or he could add a new square root function directly to the GPMath package. If the square root function was appropriate for portions of his application, but not for his Kalman filter, he could modify the `Kalman_Filter_Data_Types` package to use a different square root function when instantiating the vector operations package in GVMA.

The user also has the option of modifying how his Kalman filter is built. As can be seen from Figure 13-3, some of the Kalman filter parts instantiate other Kalman filter parts. For example, the `KFComplicated.Kalman_Update` package instantiates the `KFCommon.Error_Covariance_Matrix_Manager` and `KFComplicated.Sequentially_Update_Covariance_Matrix_and_State_Vector` packages. The `KFComplicated.Sequentially_Update_Covariance_Matrix_and_State_Vector`, in turn, instantiates the `Compute_Kalman_Gain`, `Update_Error_Covariance_Matrix`, and `Update_State_Vector` subroutines also contained in the `KFComplicated` package. Because of this, the user can customize his Kalman update operation by modifying any of the parts being instantiated. Alternatively, rather than instantiating the `Kalman_Update` package, he can implement an update directly in his `Kalman_Filter_Application` package by instantiating alternative Kalman filter parts and, optionally, perform some of the operations with custom code.

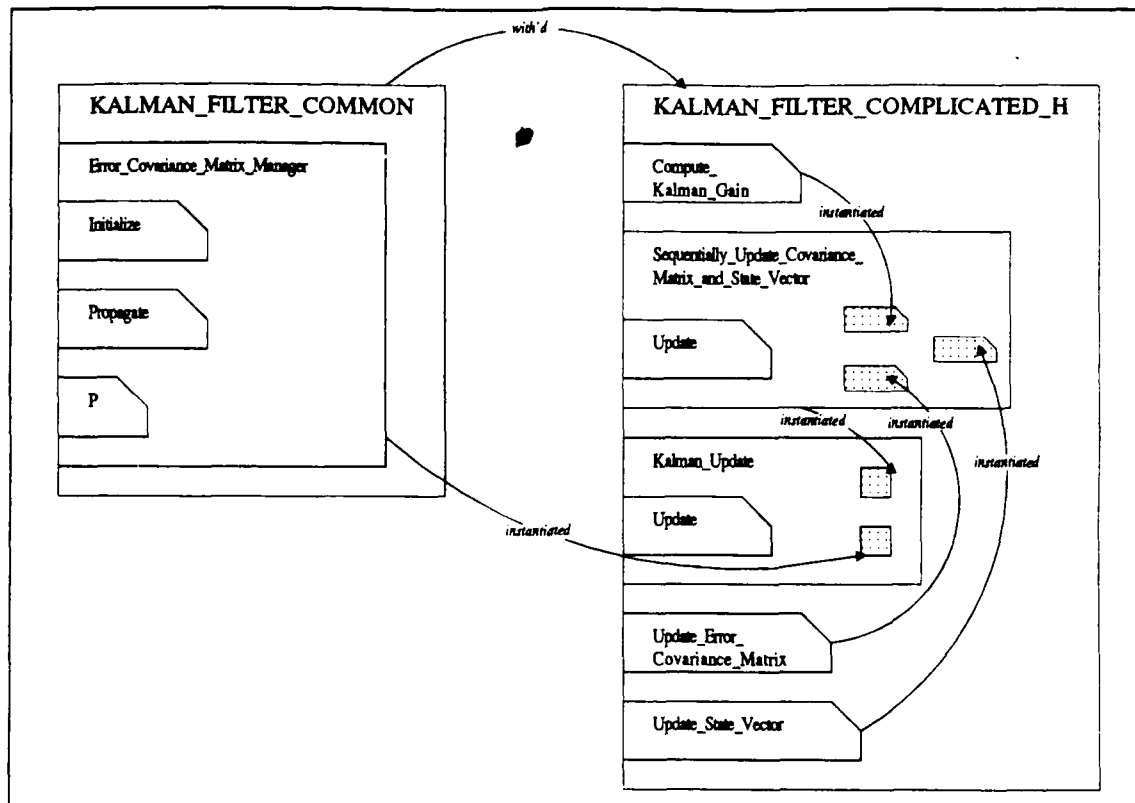


Figure 13-3: Kalman Filter Parts Instantiate Other Parts

Figure 13-4 represents the Kalman filter bundle after the user has modified the bodies for the Polynomials, Square_Root package, KFCommon.Error_Covariance_Matrix_Manager package, and KFComplicated.Update_State_Vector subroutine, and modified his Kalman_Filter_Application to do its own Kalman filter update operation rather than instantiating the update package in the KFComplicated package.

As long as the interfaces are not changed when the custom code is introduced, these modifications will not affect the rest of the user's application. Consequently, the user could start development of his application using the parts "as is" in order to get a Kalman filter up and running, and then go back and fine-tune the Kalman filter without affecting the code for the rest of his system.

13.5.3 Modifying the Data Types Package

In addition to modifying subroutine bodies, the user can also modify the Kalman_Filter_Data_Types package. This is particularly useful when greater efficiency is needed than is afforded through the standard CAMP Kalman filter data types package. The CAMP KFDT package was developed to facilitate testing of the other Kalman filter parts and to provide users with a set of data types and operations that would allow a quick implementation of a Kalman filter; it also provides the user with a template data types package that can be modified as needed. It was not developed with efficiency as the primary objective.

Modifications can range from simply instantiating different parts from the GVMA package, to replacing some of the instantiations with custom code. Some of the more likely changes to the KFDT package are identified below.

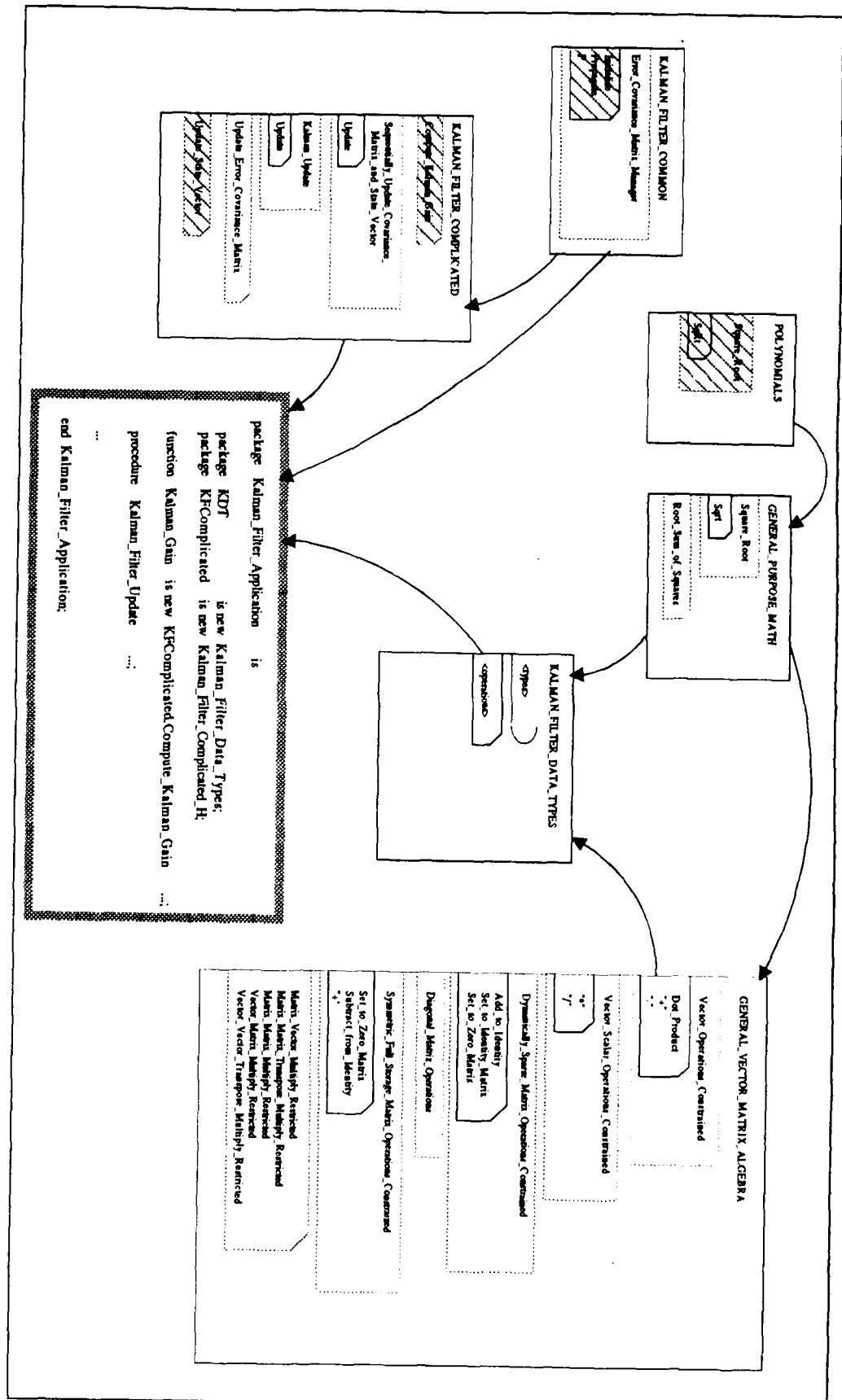


Figure 13-4: Replacing Parts with Custom Code

- Use of the `GVMA.Symmetric_Full_Storage_Matrix_Operations_Constrained` package. This package implements a symmetric matrix by storing all of the elements, but performing operations only on one of the symmetric elements and then assigning this value to both symmetric elements. This approach saves execution time over a full-storage full-operation matrix, but does not save storage space. In order to also save storage space, the user could change the KFDT package to instantiate the `GVMA.Symmetric_Half_Storage_Matrix_Operations_Constrained` package.
- Use of the `GVMA.Dynamically_Sparse_Matrix_Operations` package. This package exports a two-dimensional matrix whose elements are sometimes equal to 0. In order to save computation time, a zero-test is performed before each element operation; this saves the overhead of operating with zero-elements. This package provides an appropriate implementation for the Kalman filter matrices which require all the elements to be stored, but which may have a significant number of zero-elements.
- Use of a custom-built static sparse matrix. This type of matrix stores only the non-zero elements and thus, performs operations only on these non-zero elements. This improves on the execution time of the `GVMA.Dynamically_Sparse_Matrix_Operations` by eliminating the need to perform zero-testing. Because all of the operations are unfolded (i.e., the loops are removed) this type of operation may require more memory than one that contains loops. This is the trade-off between memory and execution speed.
- Use of operations on mixed data types — multiplying symmetric matrices and vectors, multiplying a sparse matrix by the transpose of another sparse matrix in order to obtain a symmetric matrix, multiplying a sparse matrix by a symmetric matrix to obtain a sparse matrix, multiplying two sparse matrices to obtain a symmetric matrix, multiplying sparse matrices, etc. The CAMP KFDT package obtains these operations by instantiating the appropriate generic subroutine in the GVMA package. If the user modifies the data types of one or more of the matrices involved in these operations, he also needs to modify KFDT so that its exported operators operate on the correct matrix types; this may involve writing his own operations. For example, if KFDT is modified so that some of the sparse matrices were represented as records containing only the non-zero elements, any operations on this new data type would need to be custom-developed since GVMA does not support operations on sparse matrices represented in this manner.

13.6 Summary

Two types of tools that can be particularly useful during detailed design and coding are a catalog of reusable components and *component constructors*. A catalog should provide information on functionality and performance of the reusable components, as well as information on usage, restrictions, etc. Software construction tools can provide a (semi-)automated means of composing new software systems from reusable components and custom software. Both of these types of tools are discussed in Chapters 17 and 19.

Figure 13-5 illustrates a typical scenario for detailed design and coding activities, and shows the additional activities when a parts-based approach is taken. Although the documents identified in the figure are DOD-STD-2167A documents, the impact of software reuse on this activity is basically the same when other standards are used. During detailed design and coding, the developer will complete the preliminary design, as well as the lower level design. Additional tasks that result from a reuse-based approach to software development are enumerated below.

- Check for compatibility of parts with custom application code
- Check for compatibility of parts from different parts sets
- Integrate lower level parts into the design and implementation
- Design and implement portions of the new application for reusability

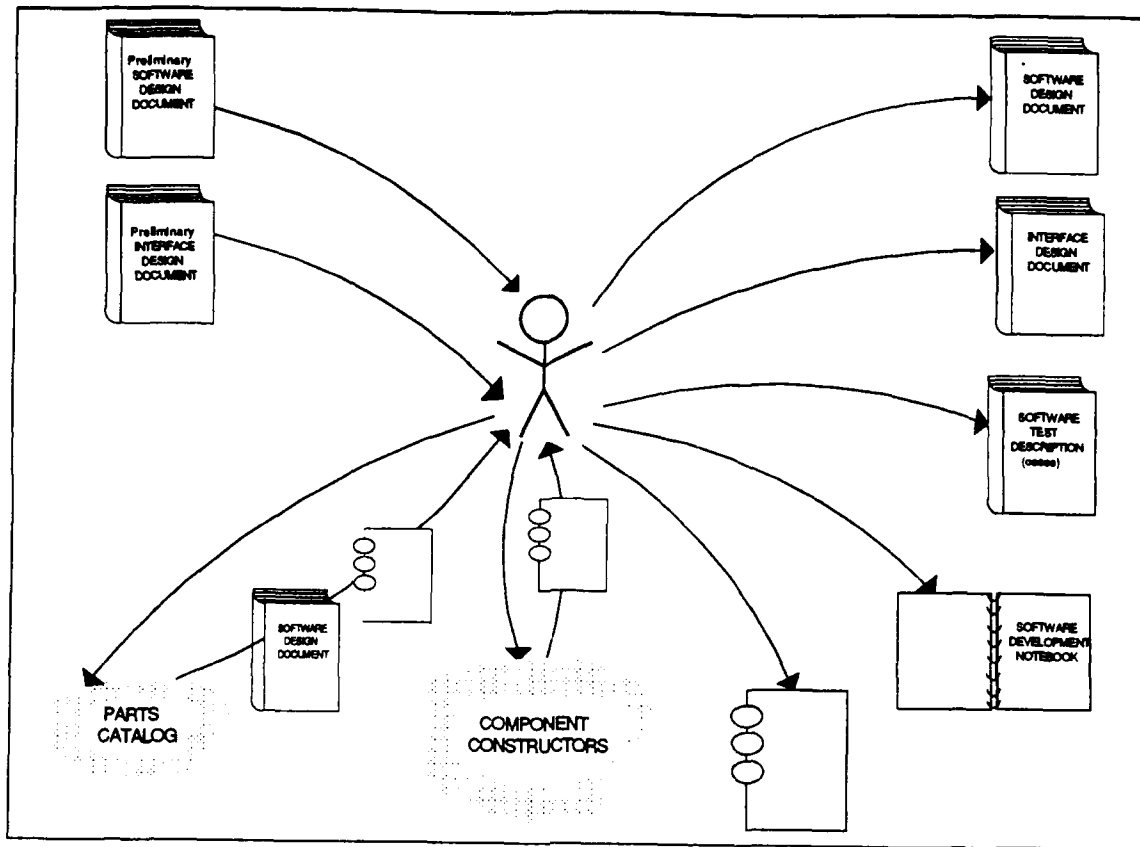


Figure 13-5: Reuse and Detailed Design/Coding

CHAPTER 14

TESTING of REUSE-BASED APPLICATIONS

One of the key issues in software reuse is confidence in the correctness of the reusable components. Because a developer of a new application is ultimately responsible for its correctness, regardless of the source of the software, he must somehow assure himself, as well as his management and customer, that the reused parts are indeed correct. Even if the parts have been thoroughly tested by the parts developer, the application developer must determine the level of testing he will perform on them. It is generally advisable to conduct some level of testing on the reused software, although it may not be necessary to subject it to unit testing.

The level of testing performed on reusable parts is dependent on the confidence that the developer has in the parts and the critical nature of the application. The developer may, in some cases, want to unit test all of the parts that will be incorporated into the new application. This may be true if the application is particularly critical or if modifications have been made to the reusable component. In some cases, unit testing may be by-passed in favor of integration testing. Bundling the test plan/procedure, test code, and expected results with a software part can provide the developer with an extra level of confidence in the parts — he can reperform the tests himself to verify that the parts work as advertised.

One approach to testing software that includes reusable components provides a three-tiered testing scheme. In this approach, no unit testing is done on any reusable software part developed by the same project team developing the new application since it will have already passed testing by that group. Other reusable software parts that are accessed directly by the new application undergo unit and integration testing. To minimize this effort, subroutines that are not used can be commented out. Some of the reused software parts will only be accessed through interface packages that are built on top of them. For example, a linked list package may only be accessed via an ordered list operations package. Packages falling into this category do not need to be unit tested, but they will undergo integration testing when the interface package is unit tested. If the reusable components come from a source other than the current project, it may be necessary to unit test them, particularly at the boundary conditions. This approach was taken on CAMP Parts Engineering System catalog system which incorporated reusable software from more than one parts set. A similar approach was taken on the CAMP 11th Missile Application: the CAMP parts were assumed to be correct and were not tested separately by the 11th Missile development team; they were tested indirectly as part of the units that invoked them.

A parts usage history can provide valuable information to a potential reuser when trying to determine how to handle testing of software that incorporates those parts. This information, which can be captured in a catalog or library, should indicate the experience that others have had in using the parts, and indicate the level and quality of testing they have undergone.

If the application is targeted at other than the development machine (as will generally be the case with RTE applications), it is often useful to first compile and perform unit testing on the development machine because the development machine will most likely have better debugging and code management tools than the compiler for an embedded computer. Once the units are debugged on the development machine, they can be tested with the cross-compiler's tools and then, if possible, with a simulator. In some cases, it may not be possible to meaningfully perform unit tests unless they were compiled on the target machine. During CAMP 11th Missile development effort (which was targeting a MIL-STD-1750A processor), most unit and package tests were first

executed in the development environment. Errors were found and corrected much faster in this environment than with the cross-compiler's tools. The units were then tested on a 1750A simulator or 1750A hardware. Since the code and the test drivers had been checked out, these tests served primarily to debug the 1750A cross-compiler, time the software, and check numerical accuracy. Figure 14-1 depicts this testing approach.

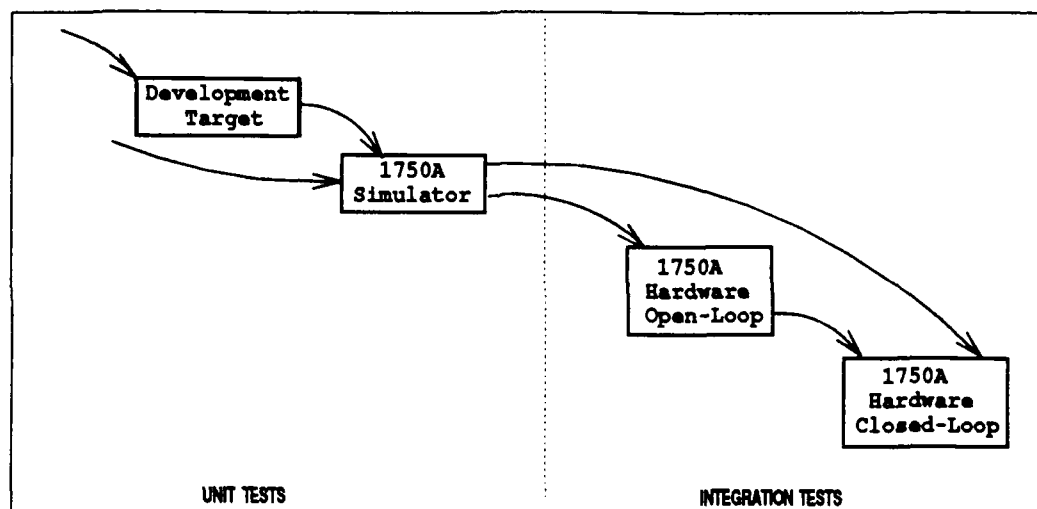


Figure 14-1: Test Approach

A parts catalog or library can support the testing of parts by providing information on level of testing and test results, and by including the test procedures, code, etc. The CAMP catalog contains pointers to test code for the reusable code parts; other catalogs may contain the actual test code and documentation.

14.1 Test-Related Activities

DOD-STD-2167A calls for the development of a test plan for each CSCI. Generally, reusable code components will not be at the CSCI level. They will be at the CSC or CSU level, thus, a project's test plan may not deal directly with reusable code components. The processing (i.e., evaluation, integration, testing, and documentation) of reusable software should be addressed in the software development plan that is developed prior to software development.

During detailed design and coding, the test cases, schedules, and procedures for integration testing should be identified and documented in the appropriate software development file; and the test requirements, cases, schedules, and procedures for CSU testing should be identified and documented in the SDF for the appropriate CSU. Testing may result in revisions to design documents or code, and hence retesting of various portions and levels of the application software. During testing, documentation and source code updates must be produced to reflect any changes that result from this testing activity.

Reuse has little direct impact during testing except in the following areas:

- Testing may go more smoothly due to the incorporation of well-tested, highly robust software parts.
- If any errors are identified during CSCI testing, it will have to be determined whether these errors resulted from the new code, the custom code, or the interfaces between them. Errors in the reusable code may be more difficult to locate. Remedies to errors in the parts are dependent upon the support structure that is in place to service the parts and the application developers.

14.2 Summary

Differences in testing applications comprised of all custom code versus those that incorporate reusable code arise mainly if errors are detected. Then the developer needs to determine where the error occurred and, if it occurred in a reusable component, he needs to know if the reusable software is supported and who is responsible for maintenance. If the reusable components are maintained by an outside party, turn-around time for error corrections is a critical factor. This type of information should be known long before the project actually reaches testing.

Reusable code should come with test documentation and test code. This will facilitate reuse by increasing user confidence and decreasing the cost to retest. Test products can be cataloged separately from the reusable code or can be a type of attribute for the code entry. How this is handled is dependent upon the particular catalog or library that is used.

SECTION III

MAXIMIZING SOFTWARE REUSE

CHAPTER 15

MAXIMIZING SOFTWARE REUSE OVERVIEW

Reuse frequently takes place on an ad hoc basis — programmers within the same group share code segments and tools they have developed, and as they move from project to project, they take their accumulated knowledge and software with them, reusing it on new projects. This type of reuse has been practiced for many years, but has not been sufficient to ameliorate the growing software crisis.

Software reuse will be maximized by (1) management support, (2) cultural changes, and (3) mature methodologies and tools. Management support is needed to obtain the additional resources required to integrate reuse into the software development process. Customer support and encouragement can have a significant impact on management support (i.e., it is unlikely that most production projects will be able to obtain the resources need to establish software reuse as a means of doing business if there is not customer support for the concept). Although there are technical impediments to software reuse, many of the barriers are cultural. For example, the "not invented here" syndrome plays a significant role in inhibiting software reuse. There are also contractual issues that inhibit reuse. Finally, although it is possible to establish and practice software reuse without the aid of sophisticated automated tools, reuse will be facilitated by software development environments that have built-in support for reuse, and by methodologies that specifically address reuse during the various software lifecycle activities. For example, a library system that supports code reuse and provides a mechanism for assisting in the integration of that code in the new application is more likely to encourage reuse than one that merely provides the location of the reusable code.

To be effective on a large scale, reuse needs to be broadly and systematically applied. Up-front planning for both the development and use of software components is needed. We have previously explored some of the costs associated with both development of reusable software components and a parts-based approach to software development. The cost to develop reusable software is higher than the cost to develop custom software due to special analysis design considerations for reusable components, additional documentation needs, etc. Costs associated with a parts-based software development approach include locating, identifying, evaluating, and incorporating available software. These are tasks which can be both time-consuming and, occasionally, tedious, particularly as the number of parts sets from diverse developers increase.

The remainder of this chapter provides an overview of issues that affect levels of software reuse, including language features, standardization, training, culture, reuse of higher levels of abstraction, and tools. The following chapters then expand on these topics.

15.1 Language Features

Throughout this manual we have assumed that reuse is taking place primarily at the code level and that the implementation language of the reusable components and applications is Ada. Reuse is not dependent upon a specific language, but it can be facilitated by language features that promote development of clean interfaces, information hiding, compile-time error checking (such as that available with strong data typing), and representation of user-defined entities (such as that available through user-defined data types). Use of a particular language or language feature does not imply that the resulting code is reusable, i.e., not every Ada generic unit is truly "reusable". Similarly, use of a particular language does not imply that the resulting software cannot be reusable, e.g., reusable FORTRAN routines can be written.

15.2 Standardization

Language standardization and standardization on a particular language for classes of applications promote software reuse. Lack of language standardization has been a hindrance to the development of widespread code reuse — the large number of languages has resulted in a situation where it generally was not cost-effective to develop a large parts set and supporting tools and environments for the range of languages that were being used. For example, FORTRAN mathematical subroutine libraries have been successful, but widespread use of higher level FORTRAN parts has not been achieved. This is, in part, due to the proliferation of FORTRAN dialects which impedes reuse. Obviously, if all parts were developed using the minimal subset of FORTRAN, this would not be a problem, but if the additional features that distinguish dialects were not in use, there would be no need for these dialects.

The DoD has addressed the language standardization issue by developing the Ada language and requiring that Ada compilers pass a series of tests for compliance with the language standard. This ensures that required language features are present, and provides the user with some level of confidence in the operational correctness of the compiler. Compiler validation does not imply that the compiler will correctly handle all valid Ada constructs, as it is impossible to check for all combinations and complexities of language use.

Standardization on a particular language for a class of applications can also promote reuse. It is difficult to reuse code components if one application is in Ada and another, similar application is in FORTRAN or JOVIAL, or some other language. Not only are there language barriers, but there may be knowledge barriers as well. For example, even if there is a way to interface the languages, the software engineer may not have enough familiarity with the language to ascertain if a component should be reused. The DoD has attempted to rectify this situation by mandating the use of Ada for mission-critical applications.

15.3 Training

Software engineers will require, and do not generally receive, training in order to successfully develop and use reusable software. At some Japanese companies, programmers are given monthly exercises in software development. The solutions are evaluated, in part, on how effectively reusable software is incorporated. Training in the identification of commonality within and across applications, and in the development of reusable software is also required. Software managers will also require training in software reuse. They will need to understand the special requirements of developing and using of reusable components.

Training should be introduced in the software engineering curriculum, as well as in the workplace. Software engineering students are generally not taught to reuse software; they are taught to develop applications from scratch. In fact, they are encouraged to "reinvent the wheel" by prohibitions on collaborative work and rewards for highly customized solutions. One-th-job training is needed for software engineers already in the workplace.

15.4 Attitude/Culture

Effective software reuse requires a change in culture or attitude. The "Not invented here" or "My way is better" syndrome can be a powerful inhibitor to software reuse. A combination of training and management incentives/mandates may be necessary to overcome this barrier. Once an organization begins to experience success in software reuse, the success stories may be the best means of overcoming attitude problems.

Managers also have attitudinal barriers that must be overcome. Too often software is viewed as a short-term asset; in order for reuse to succeed, software must be viewed as a capital investment. This is the difference between profit maximization in the large vs. profit maximization in the small. In some cases, this view is the

result of accounting practices (i.e., of how software can be handled on the balance sheet). This was also discussed in Chapter 10.

15.5 Reuse of Non-Code Software Entities

Virtually any software product can be reused, including requirements, design, code, documentation, test support documents, etc. The major problem is one of representation and support. Reuse of higher level entities (i.e., of higher levels of abstraction, such as designs or requirements) has intuitive appeal — fewer implementation dependencies would exist in the components, theoretically making them more reusable. The products of domain analysis capture canonical requirements and designs that can be reused. The representation of these and other entities continues to be a research issue, as does the mechanism for correlating related entities and representations within a library.

15.6 Tools

Tools and environments play an important role in the engineering and application of reusable software components. Although most of the tasks can be performed manually or with rudimentary tools, the tedium and time required to do so often makes this impractical. Tools that operate efficiently and reliably can facilitate the introduction and acceptance of software reuse techniques in the software development process, and provide additional productivity gains.

By and large, the tools needed for development of reusable software are the same ones needed in the development of custom software. The one major difference is in the area of tool support for domain analysis. Domain analysis has not been a part of the traditional software development process, thus there is no standard set of tools to support this activity. Some of the types of tools that may support domain analysis are identified below.

- System analysis tools to facilitate the analysis of existing applications in the domain
- Modeling tools for use in the development of a domain model. For example, entity-relationship-attribute (ERA) modeling may be used to model the domain, thus tools that support ERA model development can also support domain analysis.
- Knowledge engineering tools for use in acquiring and processing domain information

There is still considerable on-going research in the area of domain analysis, thus, the question of optimal tool support is largely unresolved. This should not inhibit the initiation of domain analysis, as many of the tasks can be performed with available tools.

Tool and environment support for reuse-based application development can ease the initial cost of reuse, facilitating acceptance of software reuse as a viable approach to software development and providing additional productivity gains. Figure 15-1 summarizes the types of tools that can be used for parts-based applications development.

Some type of component catalog or library is essential to the success of any software reuse effort. Software developers must have a means of identifying and locating potentially applicable parts. A catalog can range anywhere from a paper version to a full-blown, on-line library complete with distributed access and a high level of user support. A catalog can provide information on sizing and timing, on usage restrictions, on the location of documentation and source code, as well as test code and data, on how to use the parts, and on other user's experiences with the parts. Catalogs are discussed in more detail in Chapter 17.

Application-based search and retrieval facilities can be used early in the system/software development cycle to obtain information about potentially applicable software components long before detailed information is known

- COMPONENT CATALOG
- APPLICATION-BASED SEARCH/RETRIEVAL TOOLS
- GENERATION/COMPOSITION/INTEGRATION TOOLS
- DOCUMENTATION TOOLS
- TEST TOOLS
- EVALUATION SUPPORT TOOLS
 - BENCHMARKS FOR COMPILERS
 - BENCHMARKS FOR PARTS
 - ANALYSIS AIDS
- COMPILERS

Figure 15-1: Summary of Tools for Software Reuse

about the types of software components that might be needed. This type of facility can be used during proposal preparation and during software requirements and design to provide early identification of candidate reusable software components, allowing the user to obtain information about reusable components parts based on information about the application domain and specific application. More detailed information about the parts can then be obtained from a component catalog or library. Application-based search and retrieval facilities are discussed in more detail in Chapter 18.

Software composition/generation/integration tools can be used during FSED development and for prototyping during proposal preparation or during requirements development and verification. These types of tools can be used to obtain lower level parts for incorporation in an application (e.g., data type definitions together with their operators), as well as subsystem level parts for later integration. They can also facilitate the integration of custom and reusable software. Software composition/generation/integration tools are discussed in more detail in Chapter 19.

Benchmarks for both the parts and compilers are critical to the success of software reuse. Compilers need to be benchmarked early in a project in order to verify that they can properly and adequately handle the language features that are used in the parts. Parts need to be benchmarked to verify that they meet project functional and performance requirements. Parts that do not initially meet project requirements should not be rejected outright; slight modifications may make them acceptable, and the project can still benefit from the reuse. Benchmarks for reusable parts and compilers have been identified as an area that needs additional work. Some compiler benchmarks were developed during the CAMP program. These can be used to evaluate the correctness and effectiveness of the compiler; they are discussed in more detail in Reference [12]. Projects need to take responsibility for verifying the adequacy of their compilers, but compiler vendors need to provide compilers that support and promote Ada reuse.

Tools that analyze complexity and verify compliance with project development standards can also be useful in the evaluation of software components. Additionally, any support for estimating the closeness of fit between requirements and available components will also facilitate reuse. Tools that assist in the adaptation of components to new applications can also be useful for more complex components.

15.6.1 A Parts Engineering System for Reuse-Based Application Development

A parts engineering system should assist the user in a parts-based approach to software development. This can simplify the software development task and reduce the cost of parts reuse. Such a system should contain facilities to locate parts, assist the user in understanding and evaluating parts, and compose new software sys-

tems from parts. Ideally, much of the system that related to reuse would be transparent to the user. Although commercial tools are lacking in this area, a good deal of work is being performed.

During the CAMP program, a set of prototype tools was developed and integrated into what was referred to as a *parts composition, or engineering, system*. Although many more features could be incorporated into such a system, three major facilities were provided to support the user from pre-software development through coding and testing. Figure 15-2 depicts a high-level view of the system. The catalog served as the cornerstone of the system, providing detailed information about reusable resources. The Parts Exploration facility provided application-based search facilities and was linked with the catalog so that candidate components could be examined and evaluated. The component constructors provided a means to both tailor complex reusable components and generate portions of the application based on user requirements. A number of other facilities could be added to the system. For example, configuration management could be tied in with the catalog; requirements and design tools could be integrated and could act as a source of information for catalog attribute values, a central data dictionary could be integrated and used to generate documentation. The CAMP parts engineering system is used as an example throughout this section.

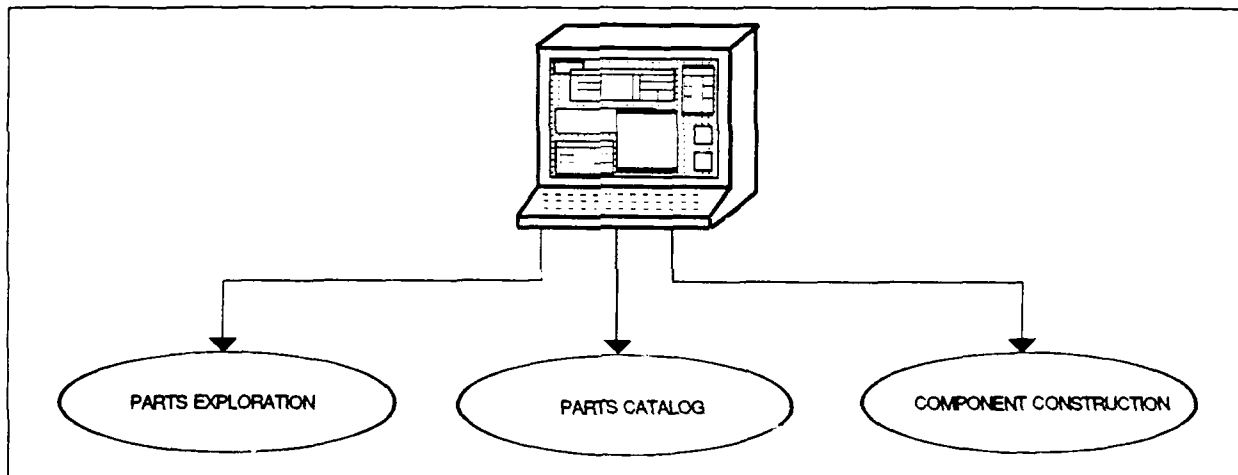


Figure 15-2: The CAMP Parts Composition System

15.7 Reuse Support Group

Reuse support can range from someone performing librarian duties and verifying that project standards have been met before a part is added to the catalog, to a significant staff that not only checks for compliance with standards, but also performs independent testing, assesses the value of parts, assists users in software reuse throughout the lifecycle, develops new parts, and provides training. As the scope broadens, the need for and demands on a support staff grow. Such a staff can be invaluable in the technology and cultural transition from custom code development to widespread software reuse. They can provide the training and support needed to alleviate risk to projects that are considering a parts-based approach to software development.

An organizationwide reuse support group that can assist projects in their reuse efforts can be a powerful means of overcoming technical and practical (i.e., time and cost) barriers to reuse. Production projects frequently cannot afford the time or resources it would take to begin a software reuse program. An organization-wide reuse group can provide the expertise needed to begin a reuse-based software development process and lessen risk of reuse, and can spread the cost of reuse over a number of projects.

15.8 Summary

The remainder of this section is organized as follows:

- **Ada Language Considerations:** This chapter includes a discussion of Ada language issues that impact software reuse.
- **Component Libraries:** The role of component libraries in a software reuse effort is discussed in this chapter. Desirable features, as well as ways to populate libraries are discussed.
- **Application-Based Search and Retrieval:** This chapter discusses the role of search and retrieval mechanisms that facilitate early planning in reuse-based software development efforts. These mechanisms basically provide access to the catalog at a higher level of abstraction than traditional keyword searches.
- **Software Construction Tools:** Software reuse can be facilitated by tools that assist in the tailoring and integration of reusable components. In this chapter, some of these types of tools are discussed. The CAMP prototype component constructors serve as an example.

CHAPTER 16

ADA LANGUAGE CONSIDERATIONS

Ada was developed specifically to address the problems of software development for large mission critical systems. It was intended to provide a standard language and promote the practice of sound software engineering practices, reducing the proliferation of implementation languages and lowering the cost of software production. At the time Ada was first conceived, the number of languages used on various DoD applications had proliferated into the 100s, and, in fact, even within a given project, multiple languages were in use (sometimes as many as 10 or more on a large project). This significantly increased the cost of development and maintenance because of the need to staff projects with personnel who know the sometimes obscure implementation languages as well as the domain. Not only is the code not very portable or reusable, but neither are the programmers. Costs can be reduced by using a single standard language for all DoD mission critical systems. Programmers will know the implementation language and be able to perform development and maintenance activities without having to learn both a new language and the application.

The Ada language embodies features that facilitate parts development and reuse. For example, the Ada package structure with the separation of interface (package specification) and implementation (package body) facilitates information hiding and the development of "clean" interfaces, thus promoting reuse. The generic unit allows development of a solution to a class of problems. For example, a generic queue package can be developed without worrying about the types of elements that will be handled in the queue. The user of that package then provides information on the type of elements he wants to queue, and the Ada generic facility will produce the appropriate object code to handle that case. Much has been written on Ada's support for sound software engineering practices, and the reader is referred to Reference [36] for more details on the features of Ada that provide this support.

Although certain Ada language features have been highlighted as promoting software reuse, Booch (Reference [6]) has pointed out that Ada does not really contain any features that have not been seen previously in other programming languages, but within Ada they are combined in a single language. Pascal forms much of the foundation of the Ada language, but lacks several features found in Ada, including private types, packages, exceptions, generic units, tasks, and representation specifications.

Ada alone is not THE solution to the software problems that have plagued the DoD software development community. Language features will not bring about a change in development and coding styles. There is certainly nothing inherent in the Ada language that would prevent someone from writing very FORTRAN-like applications, e.g., see Figure 16-1, although this would not be a good use of the available language features. Figure 16-2 shows the same segment of code rewritten into "good" Ada.

16.1 Efficiency and Effectiveness

The issues of language efficiency and effectiveness are distinct and must be addressed separately. Efficiency issues include concerns about compiler maturity and whether any language constructs are inherently inefficient. Effectiveness is concerned with whether the required operations can be implemented (easily) in the language.

The Ada language appears to be effective for RTE application. For example, the CAMP 11th Missile application, which consisted of approximately 21,000 lines of code, required only 21 lines of assembly language code to implement a complete missile navigation and guidance system. This demonstrated that it was possible to provide the required functionality almost entirely within the Ada language.

```
PACKAGE BODY NAVOPS IS

  TYPE FLVECTOR IS ARRAY(1..3) OF FLOAT;
  TYPE LFLVECTR IS ARRAY(1..3) OF LONG_FLOAT;

  ACCEL : LFLVECTR;
  VEL   : FLVECTOR;

  QUAT : ARRAY(1..4) OF FLOAT;

  BARALT : FLOAT;
  BARALTER : LONG_FLOAT;

  FUNCTION DOTPROD (LEFT, RIGHT : FLVECTOR) RETURN FLOAT IS
    ANSWER : FLOAT;
  BEGIN
    ANSWER := 0;
    FOR I IN 1..3 LOOP
      ANSWER := ANSWER + LEFT(I) * RIGHT(I);
    END LOOP;
    RETURN ANSWER;
  END DOTPROD;

  FUNCTION DOTPROD (LEFT, RIGHT : LFLVECTR) RETURN LONG_FLOAT IS
    ANSWER : FLOAT;
  BEGIN
    ANSWER := 0;
    FOR I IN 1..3 LOOP
      ANSWER := ANSWER + LEFT(I) * RIGHT(I);
    END LOOP;
    RETURN ANSWER;
  END DOTPROD;

END NAVOPS;
```

Figure 16-1: FORTRAN in Ada: Ada Language Features Do Not Force Good Design

Effectiveness should not be the sole criteria by which a language is judged. It must also be efficient if it is to be used in real-time applications. Efficiency is largely a compiler issue rather than a language issue and is often correlated with product maturity. This appears to be the case with Ada compilers. There do not appear to be any features of the Ada language that are inherently inefficient.

16.2 Compilers

Ada compilers undergo a *validation* process before they can be labeled "Ada." The Ada Compiler Validation Capability (ACVC) test suite is designed to ensure a certain level of quality and confidence in Ada compilers, and to a large extent, has succeeded. It is impossible for the ACVC to check for all valid combinations of all language features, thus validation ensures that required language features are present and operate individually as expected, but it does not ensure that complex variants or combinations of features work correctly. As a result, even validated compilers can produce erroneous code.

In order to avoid problems during development, a project should benchmark their compiler for desired features and combinations of features. For example, on a project that consists of all custom-built code, generics may not be used, thus the project would not need to concern itself with the correctness and efficiency of Ada generics. On the other hand, reusable Ada code will, most likely, make heavy use of Ada generics, so on a reuse-based Ada project, the compiler's ability to handle generics would be crucial.

```

generic
  type Elements is digits <>;
  type Indices is (<>);
package Vector_Operations is
  type Vectors is array (Indices) of Elements;
  function Dot_Product (Left : Vectors;
                        Right : Vectors) return Elements;
end Vector_Operations;

package body Vector_Operations is
  function Dot_Product (Left : Vectors;
                        Right : Vectors) return Elements is
    Answer : Elements := 0;
  begin
    Add_Up_Square_of_Elements:
    for I in Vectors'RANGE loop
      Answer := Left(I) * Right(I) + Answer;
    end loop Add_Up_Square_of_Elements;
  end Dot_Product;
end Vector_Operations;

package Basic_Data_Types is
  type Meters_per_Second is digits 6;
  type Meters is digits 6;
  type Seconds is digits 6;
  type Meters_per_Second2 is digits 9;
  type Navigation_Indices is (East, North, Up);
  type Quaternion_Indices is (Q0, Q1, Q2, Q3);

  package Distance_Vector_Opns is new Vector_Operations
    (Elements => Meters,
     Indices => Navigation_Indices);

  subtype Distance_Vectors is Distance_Vector_Opns.Vectors;

  package Velocity_Vector_Opns is new Vector_Operations
    (Elements => Meters_per_Second,
     Indices => Navigation_Indices);

  subtype Velocity_Vectors is Velocity_Vector_Opns.Vectors;

  function "*" (Left : Meters_per_Second;
               Right : Seconds) return Meters;

  function "*" (Left : Meters_per_Second2;
               Right : Seconds) return Meters_per_Second2;

  function "/" (Left : Meters;
               Right : Seconds) return Meters_per_Second;

  function "/" (Left : Meters_per_Second;
               Right : Seconds) return Meters_per_Second2;
end Basic_Data_Types;

with Basic_Data_Types;
package body Navigation_Operations is
  package BDT renames Basic_Data_Types;
  Accel : BDT.Acceleration_Vectors;
  Vel : BDT.Velocity_Vectors;
end Navigation_Operations;

```

Figure 16-2: Utilization of Ada Language Features

Many of the features that make Ada good for development of reusable software are also the most difficult for compilers to handle correctly and efficiently (e.g., generics, tasking, separate compilation, memory utilization). Thus, although it appears that Ada is effective for RTE applications, compilers must be evaluated to determine if they can effectively and efficiently handle the constructs needed for reusable software. Compiler benchmarking prior to software development can identify problem areas or inefficiencies, and allow the designers to work around them. Developers must know their compiler!

More specific examples of compiler problems have been documented in Reference [31], the CAMP-2 Final Technical Report. As Ada compilers mature, many of these problems should be resolved. If these inadequacies are not rectified, they could have a significant negative impact on the future development and use of reusable Ada software.

The CAMP 11th Missile work highlighted the need for mature compilers and highly optimized object code for RTE applications. The 11th Missile development team, which targeted a 1750A microprocessor, spent a significant amount of time debugging their validated compiler. The team uncovered 153 errors during testing; 96 of these were compiler errors. The major problems were with the handling of Ada generics and were related to the compiler's inability to handle complicated generics and the inefficiency of the generic instantiations (single body implementation). Many of the generic parts that were being incorporated into the application had to be *manually* instantiated, i.e., the generics had to be converted to non-generic code (see Reference [31]). Other problems encountered on the 11th Missile effort were with tasking (the second most prevalent cause of problems); visibility rule problems were a close third. Problems with separate compilation and storage utilization were also encountered.

In fact, because of these problems, the raw effort data without adjustment for compiler immaturity (i.e., for extra time required for testing and code modifications, such as manual instantiation), showed an 18% *decrease* in productivity, although after adjusting for the extra testing effort due to compiler problems, the project showed a 15% *increase* in productivity. These numbers give an indication of the impact of compiler immaturity.

Versions of the CAMP parts were submitted to five validated compilers and one pre-validated Ada compiler. Of these six Ada compilers, only one validated compiler was able to handle the parts submitted to it, and even that one was able to do so only after a year and a half of the CAMP team working with the vendor. Other software developers have encountered similar problems with validated compilers.

It is clear that compiler problems can negate the benefits of software reuse. Full productivity gains of parts-based software development will not be realized until compilers reach a level of maturity such that they do not cause development problems.

16.3 Chapter 13 Features

Chapter 13 of the *Ada Language Reference Manual* (see Reference [15]) describes language features that are largely system/implementation dependent, and many have considered them optional. The implementation-dependent nature of these features has made cost-effective testing by the ACVC difficult. Some features are now being tested in the validation suite.

The CAMP 11th Missile Application team found that these "optional" Ada features are not really optional in reuse-based RTE software systems. Table 16-1 lists the Chapter 13 features and indicates the ones which were

used on the CAMP 11th Missile development effort. Although many of the features could have been used, there were work-arounds for most of them. This is discussed in more detail in Reference [31], in the chapter entitled *Evaluation of Ada and Its Use in the 11th Missile Application*.

Table 16-1: Use of Chapter 13 Ada Features by the CAMP 11th Missile

Option	LRM	Used	Not Used
Length Clauses	13.2		
Size			
Integer		X	
Enumeration		X	
Fixed Point		X	
Floating Point			X
Record		X	
Array			X
Access			X
Storage_Size		X	
Small			X
Enumeration Representation	13.3	X	
Record Representation	13.4		
Alignment			X
Component		X	
Address	13.5		
Objects			X
Subprogram			X
Package			X
Task			X
Entry		X	
Change of Representation	13.6	X	
Package System	13.7		
Named Numbers	13.7.1	X	
Representation Attributes	13.7.2		X
Representation Attributes of Real Types	13.7.3		X
Machine Code Insertion	13.8	X	
Interface to Other Languages	13.9		X
Unchecked Programming	13.10		
Unchecked_Deallocation	13.10.1	X	
Unchecked_Conversion	13.10.2	X	

16.4 Ada 9X

The Ada revision process is currently underway and is scheduled for completion in 1993. This process includes revision of the language standard, standardization (ANSI and ISO approval, DoD adoption), and transition (update ACVC, training, etc.). Many factors must be considered during the updating process (Reference [1]):

- Language coherence as new features are integrated into the existing language
- Upward compatibility between the language standards so that existing applications are not made obsolete
- Impact of major changes to the standard, as well as to application development
- Impact of changes on compilers and object code
- Minimization of changes to the language definition
- The needs of various types of development efforts (e.g., RTE, reuse-based, etc.)
- Training needs

CHAPTER 17

COMPONENT LIBRARIES

A library is an essential element of a viable reuse-based software development methodology. It is a central facility for storage of information about reusable components and serves to overcome one of the primary barriers to software reuse: lack of adequate information about available components at a reasonable cost. A component catalog provides both a means to organize reusable resources, and mechanisms for search and retrieval of cataloged components. This type of facility has been referred to as a *catalog*, a *library*, a *repository*, and a *retrieval system*. Various authors have tried to distinguish among the terms, but here they are used interchangeably.

One reason for the failure of past software reuse efforts was that potential reusers were not aware of available parts and/or could not obtain sufficient information in a cost-effective way to conclude that the parts should be reused in their application. The key point here is *cost-effective*. If a potential reuser estimates that the cost of reuse is a significant percentage of the cost to develop from scratch, they will (barring other factors, such as customer or management mandate) develop from scratch. Reuse cost metrics are being developed, but most potential reusers still develop *guesstimates*. Software developers tend to over-estimate the cost to reuse and under-estimate the cost of custom development. The user must be provided with sufficient information about software parts in order to determine their applicability and evaluate the cost to use them. Some library systems have tried to develop a means to assist the user in evaluating components for a given application over and above simply providing descriptive information about the component (References [35], [10]).

A significant amount of work has been done in cataloging, and a wide range of alternatives has been both proposed and developed. A catalog can range anywhere from a hard-copy listing of available components to a sophisticated, on-line, distributed system that is integrated with other software development tools (Reference [33]). There is a broad range of features and facilities that can be incorporated; some of these are discussed in more detail in subsequent paragraphs. Because reuse is and will be practiced at many different organizational levels (e.g., project, company, corporate, industry), there may be variations in requirements for libraries that support these different levels. Despite this range of variability that is possible and will be needed, user feedback is critical in the process of defining and refining the features and facilities that are actually needed to support widespread software reuse.

A catalog system can boost productivity by providing the infrastructure needed to support reuse, making parts easy to locate, and facilitating the incorporation of those parts into an application. In other words, a catalog system can reduce the cost of reuse. Locating parts can range from simply telling the user where the source code for part XYZ can be found, to actually helping the him identify that he wants to use part XYZ and compiling that part for him.

A catalog system can provide —

- Means to identify potentially applicable parts
- Means to locate reusable components
- Means to evaluate reusable components. This can be as simple as providing descriptive information about the parts, such as functionality, space and timing requirements, restrictions, dependencies, etc., or as complex as trying to determine the differences between component requirements and user requirements.

- Means to retrieve reusable components
- Test information, including test procedure, test code, test results
- Comments of other users
- Use and cost metrics
- Automated data entry
- Support for configuration control
- Integrated set of functions
- Flexible classification scheme

17.1 Scope

Because software reuse can be practiced at many organizational levels, catalogs must also exist at these different levels. Catalogs can be developed for a particular project, company, corporation, or community (e.g., the DoD mission critical software development community). The scope of the catalog will impact features that are affected by user diversity which, of course, increases as the scope is broadened. For example, the type of user interface that will be acceptable or necessary will change as the scope changes. User interfaces are an important consideration in library development, particularly as software engineers become increasingly accustomed to multi-window graphical interfaces in their other software engineering tools. Other factors include security requirements and access control, accessibility, robustness requirements, and covered domain.

Scope affects support needs as well. For example, a catalog that is intended for use by a project or small company may need just a librarian to verify that coding and documentation standards have been met before a part is added to the catalog. Other cursory checks could include checking for test documentation and results. A large-scale community-wide or corporate-wide library might require a significant staff to not only check code and documentation for compliance with standards, but also perform independent testing, assess the value of parts, assist users in the use of parts throughout the lifecycle, develop new parts, and provide training. Such a staff could serve as the focal point for technology and cultural transition from custom code development to widespread software reuse. They could provide the training and support needed to alleviate risk to projects that are considering a parts-based approach to software development.

Scope impacts libraries in another way, as well. As the scope broadens, the issue of who bears the cost for development of new parts, maintenance and enhancement of existing parts, and user support may become significant. If the library is for DoD mission critical application developers, should the government fund it or should the contractors contribute to its support? Similarly, if the repository is corporate-wide, should the corporation pay for it or should the constituent companies fund it as long as they benefit from it?

If catalogs or libraries exist at a number of different levels (e.g., at the project, program, company, and sponsoring agency), a feedback mechanism is needed to keep the higher-level libraries populated with relevant components. Projects need to analyze their applications for areas of commonality, and once those areas are identified, they need to develop reusable components and make those components available to other developers. Projects that are using a reuse-based development approach should also revisit the domain analysis and model (if they exist) to determine if modifications or additions are needed (Reference [22]). Project inputs are an excellent way to enrich an existing parts set and keep libraries populated with relevant parts.

17.2 Library Users

A library serves both the parts developers and the parts users. The users may have diverse backgrounds and levels of expertise — some may be domain experts, while others may be software engineers. A library for reusable software components is much like an ordinary library. It provides an orderly repository and a means of managing and accessing available resources which, in the case of software reuse, may include deliverable code, requirements, design, test support, and documentation.

17.3 Library Entities

There are many trade-offs to consider when establishing a catalog and when evaluating one for use. It is easy to fall into the trap of wanting or trying to provide for all possible types of software entities and their representations. Reuse today is practiced primarily at the code level, although supporting lifecycle products should also be available in order to aid understanding and evaluation of the component, and to eliminate the need to reproduce them. As reuse expands to other lifecycle objects, the usefulness of cataloging these other entities (such as requirements specifications, designs, test procedures, etc.) and their various representations will increase.

Even when cataloging code-level entities, a decision must be made as to whether logical or physical entities will be cataloged. For example, during the prototype CAMP catalog development effort, Ada package specifications and bodies were cataloged separately. This proved to be a burden for the user who received too much data when performing queries (i.e., a query would return both the specifications and bodies that matched the query, resulting in twice as many "matches" being returned). During the re-engineering effort, a different approach was taken so that logical entities could be cataloged rather than physical entities.

The scope of the target library must be considered when identifying candidate components for inclusion in the library, i.e., the types of parts that will actually be reused will vary with the scope of the catalog; scope can affect both the types of entities that may be reused and the amount of reuse that an entity will see (Reference [33]). For example, if a requirements document were produced on a project and then the project were cancelled, that document could be reused (with adaptation) for a similar project if there was knowledge that it existed. It is most likely that the document would see reuse only within the same project group. Unless there was significant tool support, it would probably not be cost effective for other organizational groups to reuse that software entity.

17.4 Catalog Technologies

Because catalogs can range from simple to complex, the technologies required to support their implementation and use are also wide-ranging. A catalog that consists of a hard-copy listing of available components and their attributes can be produced with a simple editor and printer. At the other end of the spectrum, are knowledge-based and hypertext-based systems that allow complex queries about cataloged components and their interrelationships with other cataloged components. Thus, the enabling technologies include editors, database systems, information retrieval systems, user interfaces, knowledge-based system development, hypertext, etc.

Rudimentary catalog facilities can be implemented via a simple database capability, but there is basically no upper bound on the complexity of such a storage and retrieval system. Obviously, the more features incorporated into a catalog, the greater the cost. There are effective catalogs available that can be easily obtained and used by an organization or project. For example, the CAMP project has developed and distributed a catalog scaled for project or company use. It is implemented in Ada and incorporates no commercial third-

party software products; system dependencies have been minimized and isolated. These factors broaden the base of potential users. It is important to remember that, although there may be a number of groups developing reusable software, not every group has to develop their own catalog system; they can acquire existing catalog frameworks in which to store their components. Note the similarity between catalog development and reusable software:

- The fewer external dependencies, the more reusable the software
- A catalog can be used across domains

17.5 Screening of Parts

Prior to entry into the catalog, parts should be evaluated to verify that they meet some minimum level of acceptability. This is the function of a "parts librarian." Whether this function is actually embodied in a distinct individual is dependent upon the size and scope of the library and the use it sees. Components should not be added randomly to a library — it is critical to the success of any reuse effort that available components meet certain quality standards or, at the very least, that the quality of the parts be captured as an attribute value for each component entry.

Screening should include —

- Verification of correctness of part
- Standardization of descriptive information (both type and quality)
- Classification of the component by category, identification of keywords
- Distribution of update notices (this could be done with the assistance of the catalog system which could keep track of all users of parts and automatically notify them via a network of any relevant changes)
- Configuration management (e.g., limitation on proliferation of component variants)

Screening will increase both the start-up and sustaining cost of software reuse. The absence of any screening will most likely result in a library whose contents are largely unused — the risk of using them may well be cost-prohibitive to most projects. The amount of screening performed must be weighed against the cost of doing it and the benefits received. Some organizations will not be able to validate all of the components submitted for entry into their library, and perhaps because of the sophistication of their users, their reuse efforts will not suffer.

17.6 Entry of Parts

One tedious task associated with maintenance of a parts catalog is the entry of component information into the catalog. Automation of this data entry task can facilitate library maintenance and is feasible if all of the components are in a standard format so that information could be automatically extracted from code and associated lifecycle products. Conflicts in documentation and coding style arise when parts sets are merged, thus full automation of catalog entries does not appear immediately feasible. One possible solution to this is that no parts are allowed into the catalog until they contain a standard header. This significantly increases the effort required to maintain a catalog, although there may be benefits to the end-user. Manual entry of part data is time-consuming and error-prone.

17.7 User Support

Several researchers have identified the need to provide the parts user with assistance in evaluating available parts. This should consist of more than providing keyword and other attribute searches, and is particularly critical as libraries expand and more parts, at least superficially, appear to provide similar or identical capabilities. As the number of available components increases, it will be increasingly important to provide a means to distinguish between similar components. One approach is to only catalog "generic" components, i.e., components that provide a solution to a class of problems, and then provide the user with a means to generate a specific instance. This approach is an extension, of sorts, to the Ada generic facility.

Users will need training in reuse-based application development. They will need to learn how to most effectively identify and incorporate reusable components into their applications, and how to estimate the cost of reusing a particular component.

17.8 Part Attributes

There has been some debate about the types of information that should be kept about a software part. It is of concern both to library developers and users. Two approaches are summarized here.

- Don't repeat in the catalog what the user can get from the code headers and the source code itself. The justification for this approach is that the user will look at the code anyway, so why repeat the information. Just the essentials should be kept in the catalog entry, and the user should look at associated lifecycle objects (LCOs) for any remaining information that he needs.
- Provide a complete picture of the part in the catalog so that user does not have to go to other LCOs unless he wants significantly more detail

Regardless of the approach taken, catalog attributes must capture both what the resource is and what it does (i.e., both semantic/functional information). The catalog should provide information on related lifecycle entities, such as requirements, design, source code, test code and test data, etc., as well as on the component itself.

Feedback from users of the parts should also be maintained in a parts catalog. This can serve two purposes: (1) it provides future users with additional information on how a part worked out in an actual application, (2) it provides feedback to parts developers on areas for improvement or areas where they excelled.

There are several sources for attribute information:

- Component Developer: He may enter or provide information gathered as the part was designed, implemented, and tested.
- Reusable Component: The component itself contains basic information (e.g., in a code header) that may be entered automatically or by a data entry person.
- Catalog/Library System: Certain attributes can be deduced by the library system. For example, part number and revision number can be generated, as can the last change date of the entry.

A catalog should provide information on efficiency of the components, environmental requirements, usage restrictions, usage instructions, location of documentation and source code, as well as test code and data, and other user's experiences with the parts. It should also contain the parts themselves or pointers to them. It should capture information about inter-relationships between cataloged entities, as well as information about attributes of a particular entity. Entities can be related in any number of ways. For example, a high-level reusable component may be a composite of lower level components. In this case, the user may want to know the components that comprise this one. Additionally, there are software lifecycle objects associated with each

lifecycle activity (e.g., requirements specification, design, etc.) and each of these may have one or more representations. A catalog should allow the user to navigate through the related lifecycle objects.

When entering component data, care should be taken to avoid proliferation of enumerated attribute values such as keywords. This will facilitate more accurate search and retrieval of components. For example, if keywords could be randomly selected whenever a component was added to the library, the situation could arise where some components were described by the keyword *Math* and others were described by the keyword *Mathematical*, and there would be no overlap in searching by specified keyword. There are several ways to overcome this problem:

- Have a controlled list of keywords that must be used when describing a component.
- Provide for synonyms and/or a thesaurus.
- Provide for partial searches/matches.

17.8.1 Classifying Parts

There is no consensus on how parts should be classified. Approaches decompose into two broad categories: one a fixed, enumerated classification scheme, and the other a flexible, faceted scheme. In the first approach, all of the categories for a particular domain are pre-determined and applied as new parts are added. This approach has the advantage that it is straightforward and relatively easy to implement. Prieto-Diaz and Freeman (see Reference [35]) took a different approach and proposed the use of a faceted classification scheme; this approach was first used in library science. Construction of classification categories is based on a synthesis process rather than having the categories enumerated in advance. Faceted classification schemes tend to be more flexible, but less straightforward to implement than enumerated schemes. Types of classification were discussed in Chapter 3.

17.9 Populating Libraries

Alternative library population techniques have been proposed:

- Systematic population that begins with a thorough domain analysis and proceeds through component development and cataloging
- Ad hoc or opportunistic population
- Systematic scavenging

These approaches can be used regardless of library scope, although there are pros and cons to each of them.

Systematic population has the potential for the greatest gain, but it also has the potential for the greatest required investment. Because it involves a systematic examination of domain of interest, there is a greater possibility of developing high-level reusable components that will have significant payback. The cost of this type of effort is generally relatively high. It requires the services of both software engineers and domain experts, and is time-consuming process with no well-established methodology. See Chapter 3 for more details.

The cost of ad hoc or opportunistic population techniques is relatively low, but the potential payback is also relatively low. With an ad hoc approach, components are added to a library as they are "discovered." One of the drawbacks of this approach is that there will be no uniform standards (e.g., coding or design standards) under which all of the components of a library will have been developed. This may increase the cost of reusing the components as they will have a less uniform presentation. If components are "brought up to standard," the cost of populating the library increases. Another drawback to this approach is that there may be gaps in the domain coverage provided by the reusable components, i.e., there may be some areas of the domain that are

relatively well-covered, and others that have no coverage at all. Additionally, because the components were not systematically developed to work together, there may be integration problems.

Systematic scavenging falls somewhere between systematic and ad hoc population. Systematic scavenging implies that there is actually some organization or individual that performs the scavenging and converts (if necessary) these scavenged objects into reusable entities. Scavenging can take the form of reviewing existing systems for candidate parts, attending walkthroughs, etc. Generally, components gathered in this way will require additional effort to make them reusable. For instance, they may require generalization and/or additional documentation. The cost of populating a library via this mechanism will generally be lower than if it were done via a domain analysis, although again, the coverage of the domain may not be as complete and there is generally a relatively hefty cost involved in making components reusable.

Consideration must be given not only to initially populating a library, but also to sustaining the library with reusable components. To this end, components need to be added in order to maintain domain coverage and to expand into additional domains. Projects are often the best source of inputs for library additions. They are close to the problem domain and, with appropriate reviews, can identify where additional reusable components may be needed. These additional components can be developed by project personnel if they have sufficient resources plus specialized parts development expertise, or by a project-directed component development group. Component development organizations are discussed in greater detail in Chapter 8.

17.10 Library Evaluation Criteria

Some factors to consider when evaluating a catalog or library are enumerated below.

- Growth potential

- Can the library expand to accommodate a growing collection of reusable software components?
- Can the library accommodate multiple types of software components (e.g., code, design, requirements)?
- If the library is for code parts, can it accommodate multiple languages
- Can the library be easily extended to accommodate types of reusable components that are not yet planned for (e.g., state charts, test procedures, ERA diagrams)?
- Can the library be easily integrated with other software development aids?
- Does the library support partitioning into multiple levels (e.g., project library, program library, company library)?
- Does the library support partitioning by domains, by projects, or by some other criteria?

- Usability

- What are the hardware and software requirements? Is third-party software required to support library operations? Is it tied to a particular hardware platform? Or is it (relatively) portable?
- What are the storage and memory requirements?
- What is the response time? How will increases in the size of the collection affect performance?
- Can relationships be specified between cataloged components and between representations of a given component (this can facilitate locating the desired representation of a component)?

- Can alternative representations of a given component be generated (e.g., given a code component, can the design be generated, and vice versa)?
- Human Factors Considerations
 - Is the system easy to use? Does the interface facilitate rather than hinder system use? Is there adequate documentation? Is there context-sensitive on-line help?
 - Is the system usable by personnel with diverse levels of expertise?
 - Is the interface consistent?
 - Are alternative user interface paradigms supported (e.g., natural language, structured queries, graphical queries)?
 - Is background processing of lengthy queries supported?
 - Is there "carry-over" between the functions in the library, i.e., if one function returns a list of parts, can that list be used in any other function that operates off of a list?
 - Are there browse facilities? I.e., can the user start at part X and browse through the catalog? Are there alternative ways to browse? E.g., can he start at Part Number X? Can he browse through all items in a list?
 - Is the catalog tied to the total software environment? Does it include configuration management functions? Is there a mechanism for generating/sending update notices to users? Is there a mechanism for obtaining user feedback?
 - Is hard-copy output supported in addition to interactive output?
 - Is there automated tracking of catalog utilization? I.e., number of uses, number of hits, problem reports?
 - Is automated documentation production supported?
 - Does the system support compilation, execution, editing of components of just retrieval?
 - Is there a mechanism for standards enforcement?
- Maintenance
 - Is the entry of component attribute information manual or (semi-)automated?
 - Are there consistency and correctness checks on the data entered into the catalog?
 - Is there access control to the cataloged data and to the catalog functions?
 - How are the maintenance and end-user functions partitioned?
 - What type of support is needed to sustain the library?
- Component Evaluation Support
 - Is sufficient information provided about each cataloged component to allow the user to make an informed decision to use the component? E.g., are benchmarks provided? Is there data about efficiency? Reliability?
 - Can the attribute values be easily examined?
 - Can the reusable components themselves be examined and retrieved?
 - Can multiple representations (or views) of a cataloged entity be examined or retrieved?
 - Is the classification structure sufficiently rich to handle all domains that may be needed at sufficient granularity to make searching productive?
 - Are size and complexity metrics maintained/available?

- Search

- Are there multiple ways to identify candidate components, e.g., attribute values, facets, classification scheme, domain-based search, application-based search?
- Does the library support multiple selection criteria and logical operators (i.e., ANDing, ORing, NOT, FOR_ALL, FOR EVERY)?
- Are textual fields searched? E.g., would the functional abstract field be searched for a particular keyword?
- Are the following types of searches supported: search by enumerated item, search by string, search by substring

17.11 Case Study: The CAMP Catalog

The CAMP program has been concerned primarily with reuse of code parts and their associated documentation and test objects. To this end, the CAMP catalog was designed to facilitate obtaining information about code parts, including functional, performance, and usage data, as well as information about the location of source code, their requirements, design, and testing. Each catalog entry is described by a number of attributes (see Figure 17-1). Attributes were classified as either required or recommended, thus, values may not be found for all of the attributes for all of the parts. Some of attributes are rather costly to obtain. For instance, performance data (e.g., source size/complexity characteristics, timing, fixed code object size, and accuracy information) requires that benchmarks be run to obtain this data. The *type* attribute describes the type of entity that is cataloged, for example, schematic parts, operational parts, etc. This attribute is site-tailorable, so that, although the remainder of the attributes are intended to support code components, the catalog could be used for other types of entities.

During the CAMP program, two versions of a component catalog were produced. The first was a prototype that was geared specifically toward the CAMP components. The second, re-engineered version is suitable for a broad range of domains. Both versions were scoped primarily for project or company use, and included functions to add, modify, delete, and print catalog part entries; search for parts; examine part entries and part code; and retrieve parts. The goal of the re-engineering effort was to produce a more robust, all-Ada version of the prototype CAMP catalog, and to provide enhanced functionality based on lessons-learned with the prototype. For example, site-tailorability of some catalog attributes was added during the re-engineering effort.

The prototype CAMP catalog contained separate catalog entries for Ada specifications and bodies, as well as for structuring packages, resulting in over 1100 catalog entries, although there were only 454 CAMP Ada parts. This organization was problematic for the potential reuser — there were too many similar entries, making it difficult for the user to find potentially applicable parts. This directly contradicts one purpose of a catalog, which is to lower the cost of reuse by facilitating the acquisition of information about available parts. The redefinition of the *type* attribute during the catalog re-engineering effort corrected this problem.

During the CAMP program, a specific domain — missile operational flight software — was examined, leading to the development of a hierarchical, domain-specific classification scheme for the parts (see Figure 17-2). The prototype catalog incorporated this classification scheme, and required code modifications in order to change it in any way. In the re-engineered catalog, the taxonomy was made both site-tailorable and modifiable. For example, if a site starts out with only the CAMP parts, the CAMP parts taxonomy may suffice, but if another set of parts for a different domain is acquired, the taxonomy will have to accommodate those parts as well. The re-engineered CAMP catalog can handle this situation and accommodate both classification schemes. The keyword field can be used to handle other classification data.

REQUIRED ATTRIBUTES	RECOMMENDED ATTRIBUTES
• Part Number • Revision Number • Part Name • Taxonomic Category • Functional Abstract • Type • Last Change Date of Entry • Development Date • Developer • Govt. Sensitivity Level of Entry/Part • Organization Sensitivity Level of Entry/Part • Withs • Withed By	• Dependencies • Design Documentation • Design Issues • Developed For • Keywords • Lines of Code • Location of Source Code • Location of Test Code • Performance Notes • Requirements Documentation • Remarks • Restrictions • Revision History • Sample Usage • Statement Count • Used By • User Comments

Figure 17-1: CAMP Catalog Parts Attributes

Another feature of the re-engineered CAMP catalog is the use of *reference lists* for attributes such as keywords, developers, and users. When parts are entered into the catalog, only values already in the reference lists are considered valid for those attributes. This promotes uniformity in the catalog entries, and facilitates search and retrieval of parts by limiting proliferation of attribute values. Initial reference lists are established at catalog installation time, and can be modified once the catalog is in place.

A batch interface was added to the re-engineered catalog primarily to provide the user with an efficient means of entering large quantities of data. It can also facilitate the addition of automated data entry if a catalog site would like to add this feature at some time.

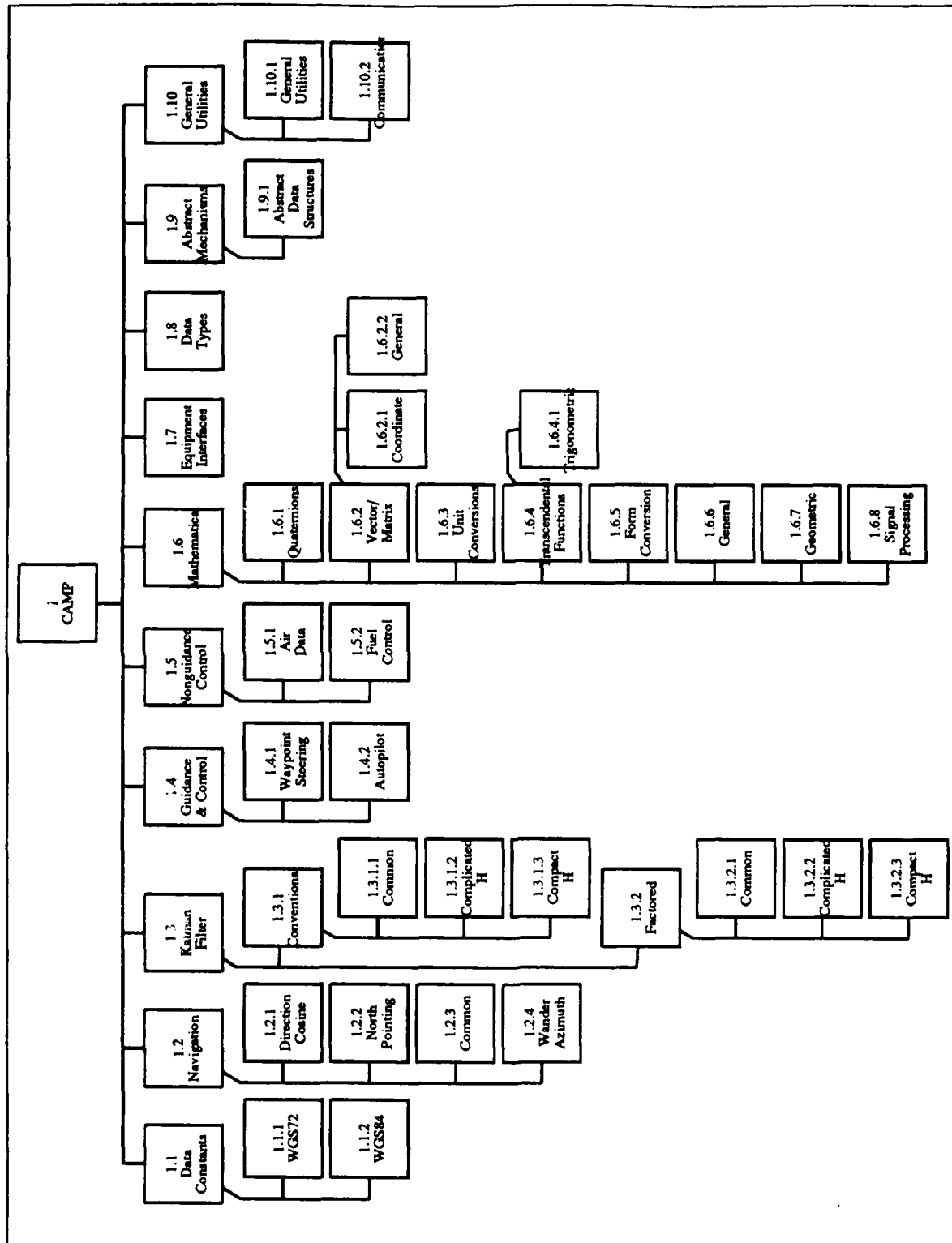


Figure 17-2: CAMP Parts Taxonomy

CHAPTER 18

APPLICATION-BASED SEARCH and RETRIEVAL

The potential reuser should be provided with a tool that allows multi-level access to available reusable components. For instance, a basic catalog with a relatively straightforward querying capability is sufficient for users who have a fairly clear idea of the types of parts they are looking for. In order to assist the system designer and requirements analyst in the identification of potentially applicable parts early in the design cycle, a different view into the catalog may be necessary. The user should be able to interact with the catalog at an application or domain level rather than at a component level. This type of functionality was demonstrated during the CAMP program in the AMPEE System *Parts Identification* functions.

Application-based search and retrieval facilities provide high-level access into a parts catalog. They are intended for use by a system or software engineer early in the system/software development cycle — as early as during proposal development or during system requirements and design. The goal is to allow the user to identify candidate software parts based on the project-specific requirements rather than on specific component attribute values. This requires some type of mapping between the user's application requirements and the reusable components' requirements. The use of formal specification languages for both parts and applications has been considered as a way to facilitate this mapping, but it is generally considered to be too great a burden on the user at this time — it forces the user to learn yet another language. Their use has been advocated primarily because it may be possible to more precisely map application requirements to parts if the requirements of both the parts and the new application are rigorously specified.

This type of tool is not essential to a reuse effort, but it can help maximize reuse. It can facilitate "make or buy" trade-offs by providing information on software that is already available (i.e., can we "buy" it) or do we have to make it (i.e., develop it from scratch)? Components can also be identified for rapid prototyping and benchmarking, thus facilitating sizing and timing studies. Availability of parts can impact cost, thus this type of tool can facilitate development of cost estimates. It can also be used as a training aid for new personnel.

There are two approaches to this type of parts exploration: an *architectural* approach and a *system*, or *application*, approach. The architectural approach provides the user with a model of the software for the domain of interest and allows him to traverse it, identifying potentially applicable parts at any node of the model. This approach looks at the structure of the software and the subsystems that comprise the application domain. The system, or application, approach requires input about the user's application; potentially applicable parts are then identified for him. It looks at the system requirements for the application. Both approaches require that domain-specific knowledge be captured within the function. One is based on the assumption that there is a common software architecture for a class of applications. The other is based on knowledge of the domain, the available parts, and their relationships.

The value of these functions over a catalog with a simple querying mechanism is that the user does not have to know specifically the parts, or types of parts, that he is looking for. He can interact with the functions at a system level, which is more appropriate early in a software development effort. Once parts are identified via these functions, more detailed descriptions can be obtained by querying the catalog directly.

Both approaches to parts exploration were prototyped during the CAMP program. They were targeted at the CAMP parts and the missile operational flight software domain, but the concepts are broadly applicable. Within the CAMP prototype, these approaches were referred to as *Application Exploration* and *Model*

Walkthrough. Figure 18-1 depicts these use of the Application Exploration function, and presents a conceptual overview of this the Missile Model Walkthrough function.

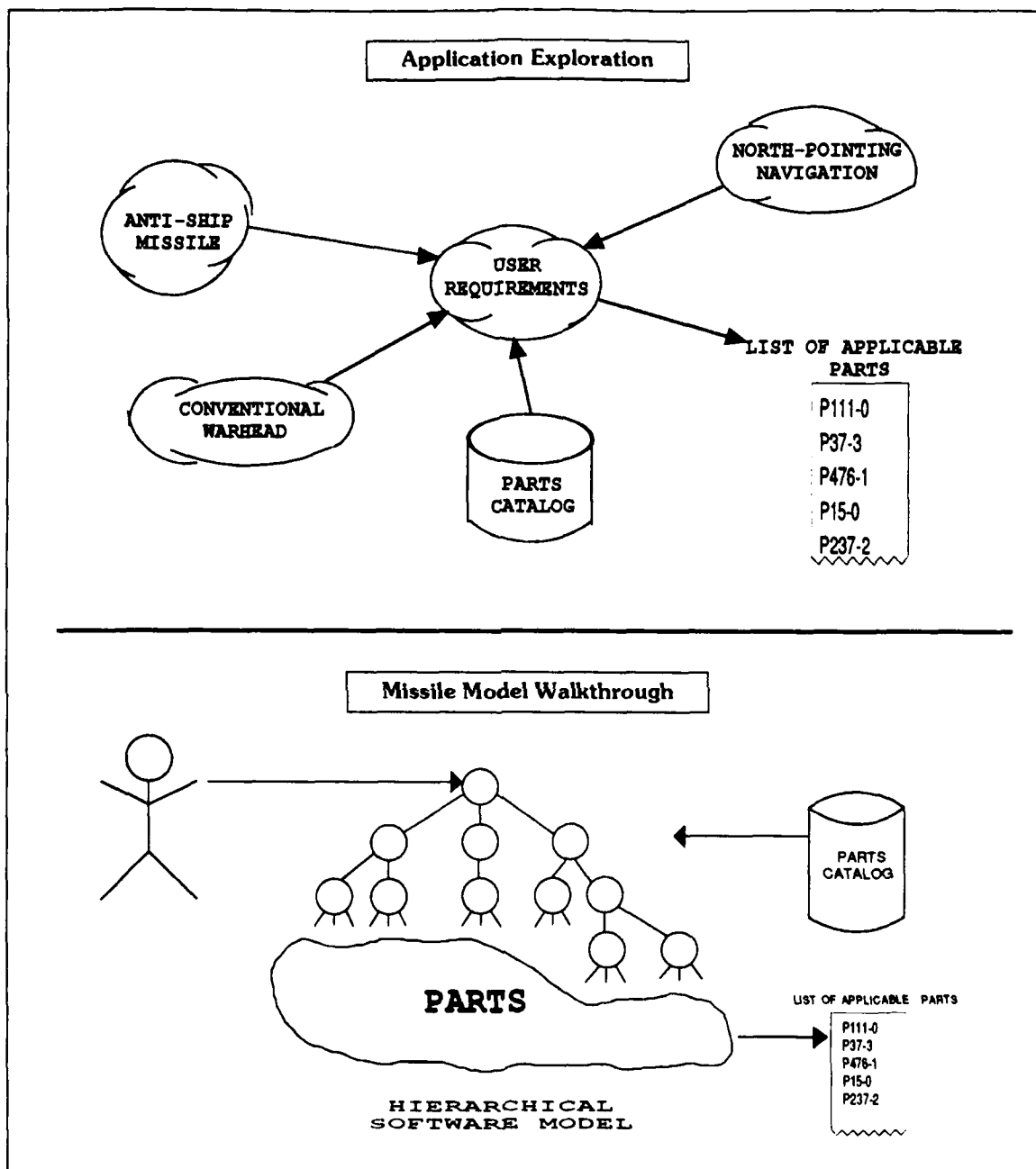


Figure 18-1: CAMP Parts Exploration Approaches

The application approach (which was captured in the Application Exploration function) contains knowledge of both the parts and the application. For example, in the CAMP prototype, if a user indicated that his target was a ship, then the system would know that he needed a seeker (application knowledge) and that if he needed a seeker, then he could use CAMP math parts, the Data Bus Interface Constructor, and the Finite State Machine Constructor in the implementation of the seeker (knowledge about the relationships between parts and the application). This illustrates the application of both rule-based and parts-based reasoning within this function.

The architectural approach was embodied in the Missile Model Walkthrough function. Figure 18-2 illustrates the CAMP missile software model. This model of missile flight software is based on knowledge of the software parts required for previously developed missiles, i.e., based on past experience, it has been determined that most, if not all, missiles of a given type, e.g., anti-ship missiles, require a particular set of software parts, and this set of parts form a hierarchical structured model of software parts required for a particular type of missile. For example, a user who knows he will be working on a navigation system, would begin exploring at the highest level of the model for navigation. The function would then guide him through the available parts, using information from the parts catalog. From navigation, he could move down to operational parts, then into either vector operations or trigonometric parts. If he chose trigonometric parts and knew he would need a sin function, the system would return a list of parts on that category for sin function. Figure 18-3 presents an example of how this function might be used; the Advanced Medium Range Air-to-Air Missile is used as an example. The user's inputs are shown in the box on the lefthand side, and the categories of parts that would be selected are shown in the box on the righthand side.

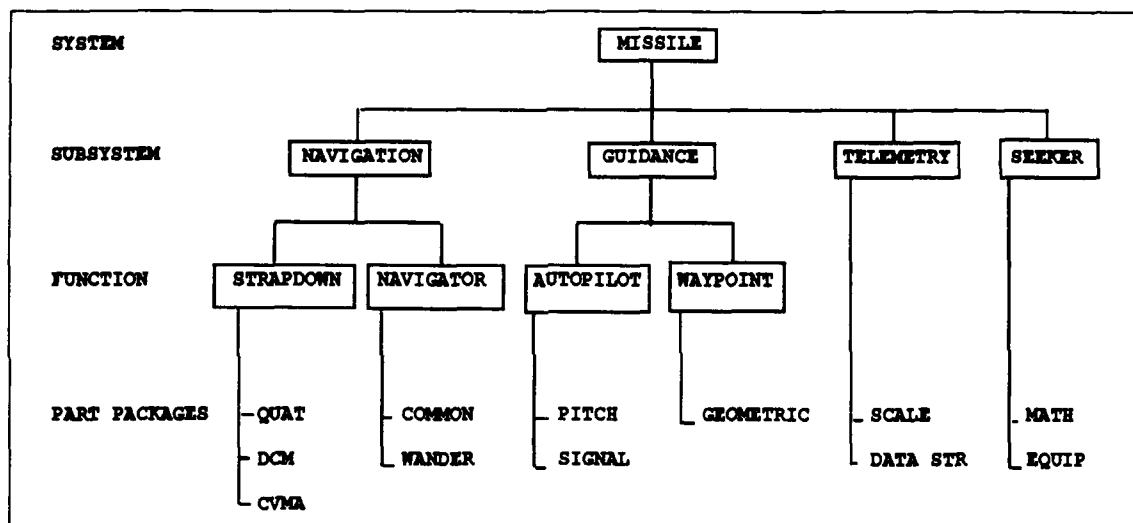


Figure 18-2: CAMP Missile Software Model

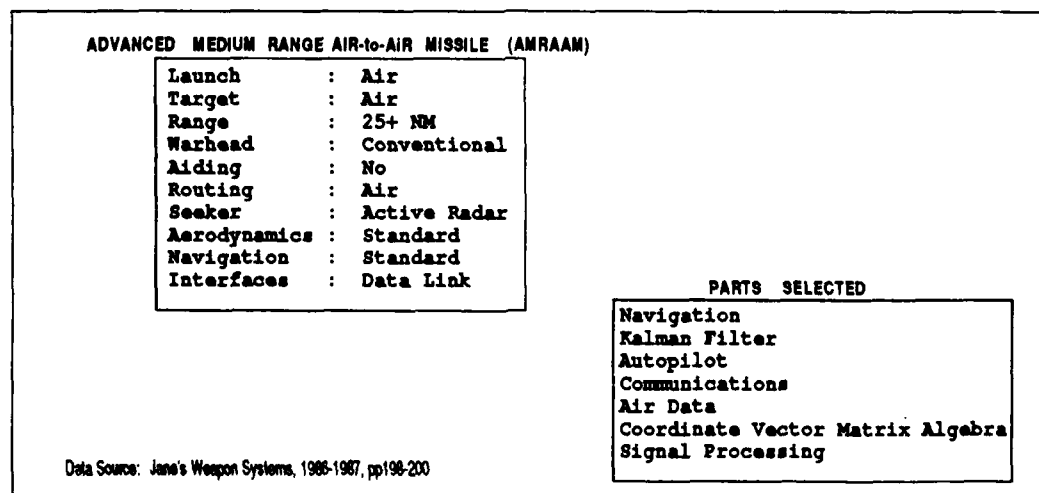


Figure 18-3: CAMP Missile Model Walkthrough Example

The types of search mechanisms discussed here have significant potential for maximizing reuse by facilitating early identification of candidate software components. They have been prototyped in the missile operational flight software domain, but they are equally applicable to other domains as well. In order to accommodate "real" projects, the functions will need to operate at a sufficient level of granularity to identify meaningful components; this becomes more complex as additional domains are incorporated into the functions.

CHAPTER 19

SOFTWARE CONSTRUCTION TOOLS

Component generation, tailoring, and composition tools can facilitate reuse-based software development. *Composition* tools assist the user in constructing new applications from existing components. This may require the generation of code to glue the parts together. *Generation* tools assist the user in the generation of new components. *Tailoring* tools assist the user in tailoring existing components to the user's application. They operate most effectively if the operational domain can be constrained, thus permitting more domain-specific information to be captured and used to drive the tailoring process. Although manual tailoring guidelines can be developed, automated tools can reduce the incidence of errors in the tailoring process and enhance productivity.

These types of tools reduce the level of detail that must be dealt with in, i.e., less information is required on the part of the user to effectively use reusable software components in a new application. The goal of most of these tools is for production of deliverable quality code, but they may prove particularly useful for developing prototypes where some of the constraints can be relaxed. A significant amount of research has been and is being performed into approaches to automating these functions. Much of this work is being conducted by universities and research laboratories, and may not provide significant payback for some time. Other approaches can have a more immediate impact (e.g., code generators within limited domains).

Despite the benefits of these tools, they remain in relatively short supply, particularly production-quality tools. Automated code generation has been a goal almost since the beginning of programming, but has yet to be fully realized. It is a difficult task at best, and is only made more difficult in the RTE arena by the imposition of additional constraints on the resulting code, e.g., space and time constraints. Code generation within limited domains is feasible, i.e., if the problem space can be sufficiently constrained, it is possible to generate high quality code. Limiting the problem domain reduces the number of available choices and allows more domain-specific knowledge to be captured and used in the code generation process.

With all of these benefits, the logical question is, "Why aren't these types of tools more commonly available?" Generally, they require a significant investment to develop, and there is, as yet, no guaranteed success. The code or types of code that a generation system would support must either be particularly critical or heavily used in order to make their development cost-effective. There have been some success with tools of this type in narrow domains. In fact, when FORTRAN was first introduced, some considered it to be an automated code generation system. Draco (Reference [32]) was an early research system that tied code generation and reuse together. Although the benefits of automated code generation are obvious, the realization of their promise has eluded researchers for many years. In the following case study, the CAMP prototype construction/generation approach is discussed.

19.1 Case Study: The CAMP Constructors

During the CAMP program, a prototype composition/tailoring/generation tool was developed; it was referred to as a *component constructor*. Twelve such constructors were developed. Each was a software system capable of producing project-tailored Ada software components given domain knowledge, programming knowledge, and knowledge of the application requirements. The goal of these constructors was to produce usable application code tailored to the user's requirements, and to facilitate software reuse by reducing the need for both detailed knowledge about specific parts and the level of Ada expertise that is required to use the parts.

The CAMP constructors provided the user with a means of generating customized software components from existing software parts and standard designs. Code generation was attainable in this case because of the narrow domain in which it was attempted. Code generation in the general sense is not yet feasible.

These constructors facilitate the use of both *complex generic* parts and *schematic* parts (these were discussed in Chapter 3). *Complex generic* parts are Ada parts that require both data types and operators, and may also have complex interactions with other reusable parts (e.g., the CAMP Kalman filter or autopilot parts). Information about selected generic parts is contained within the constructor. The constructor defines data types, generates the instantiation for the generic, and identifies other CAMP or user-supplied parts which may be needed by the complex generic part.

Schematic parts are parts whose commonality cannot be captured directly using Ada language features (e.g., a static sparse matrix). A schematic part consists of a "blueprint" or template of the part's structure. The constructors use the blueprint, together with a set of component construction rules and the user's specific requirements to create a new software component. This new component is generated following the basic outline of the schematic part; the user's requirements provide the tailoring information. Figure 19-1 depicts the CAMP approach to generating specific tailored components from schematic parts. Although in the CAMP implementation, Ada was the language under consideration, the concept of schematic parts is applicable to other languages as well. Schematic parts extend the range of reusable software components.

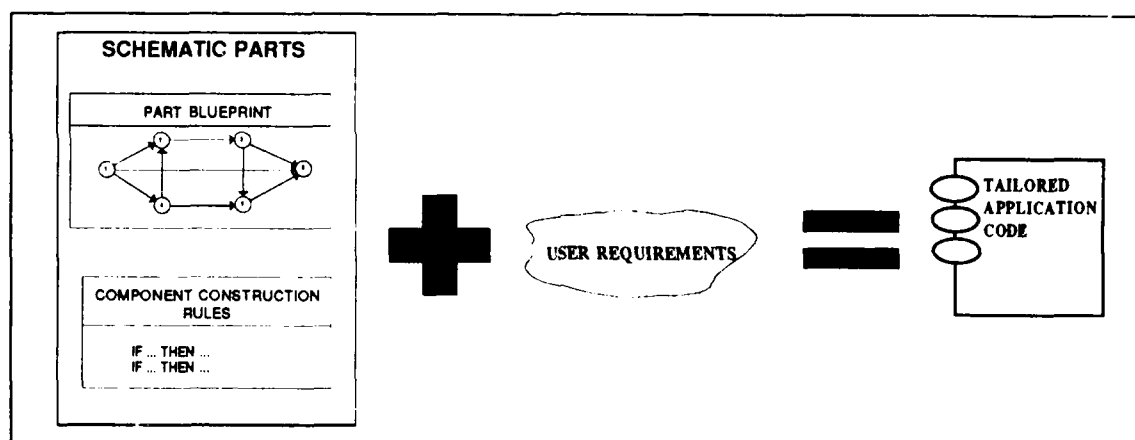


Figure 19-1: CAMP Schematic Reuse

The user enters requirements to the constructor by responding to prompts which are based on component knowledge stored in both the constructor and the requirements knowledge base. The knowledge consists of things such as data types and other applicable reusable parts. At each requirements definition session, the user creates a set of requirements which can be named, stored, and later retrieved and replayed. This requirements set is used by the constructor to generate the constructed Ada component; a "help" file is also constructed which contains useful information for the user of the component. Previously entered requirements sets can be retrieved and used either "as is" or with modification to regenerate the component.

The 12 constructors prototyped during the CAMP program covered Kalman filter operations, autopilot (2), data type definition, data bus interface, finite state machine, navigation operations (2), and abstract processes (time and event driven sequencers, a task shell constructor, and a process control constructor). In the following paragraphs the Kalman filter constructor is discussed in detail.

19.1.1 An Example: The CAMP Kalman Filter Constructor

Figure 19-2 presents a simplified view of a Kalman filter for missile operational flight software; Kalman filters are also used in other application areas. As previously discussed, the Kalman filter receives sensor and navigation position input, and calculates position and ISA corrections which feed back into the navigation and ISA systems.

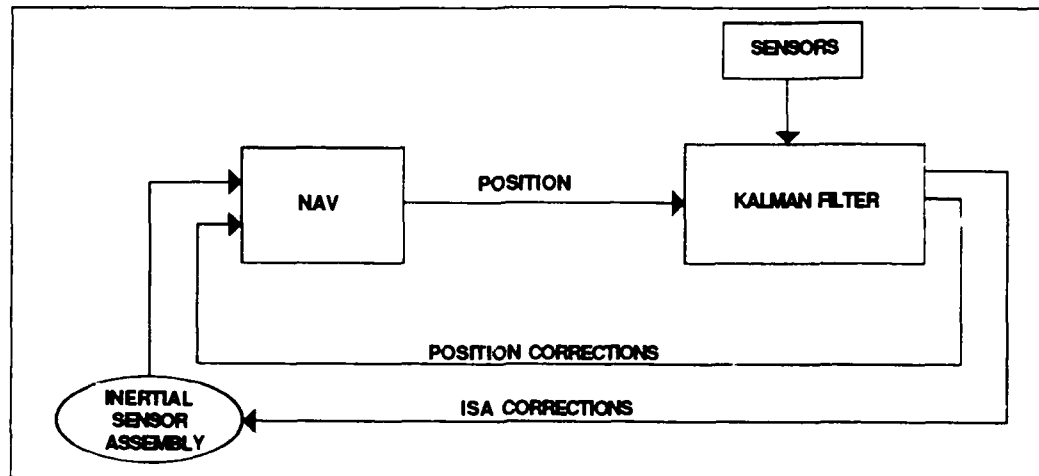


Figure 19-2: High-Level View of Missile Kalman Filter

The CAMP Kalman Filter part is made up of a number of Ada packages; Figure 19-3 depicts the package structure.

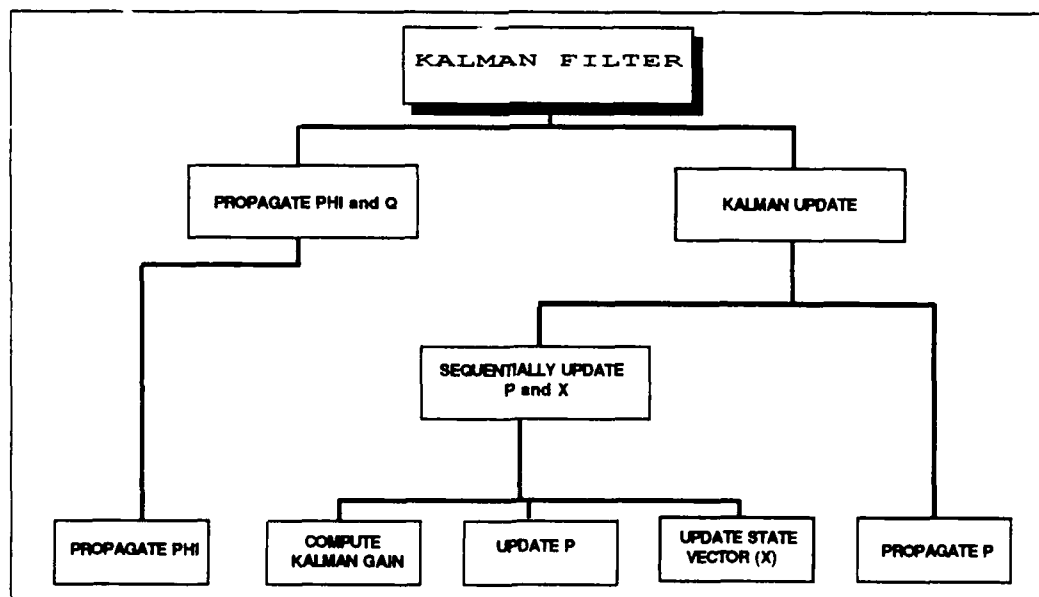


Figure 19-3: CAMP Kalman Filter Parts Hierarchy

19.1.1.1 Kalman Filter Matrices

In order to address the efficiency needs of Kalman filter operations, special matrix parts were created. In all, five different CAMP matrix parts were created to support specific Kalman filter operations. There is a generalized matrix package which supports all types of matrix multiplications, matrix-scalar operations, etc. on a general-purpose matrix, but due to efficiency constraints, the general-purpose matrix is suitable primarily for study and analysis, i.e., it is not efficient enough for most embedded applications. Only other special-purpose packages were developed for diagonal, symmetric, and sparse matrix types.

Two types of storage are supported for symmetric matrix types: full storage and half storage. The difference is in whether the entire matrix is stored (full storage), or if only half of the matrix is stored with the symmetric properties being used to maintain the other data points (half storage). The advantage of using full storage is that access to individual data values is faster; the advantage of the half storage matrix is that storage is saved.

Two types of sparse matrices are available — dynamically sparse and statically sparse. The dynamically sparse is slower in terms of access time because the part must verify that the value is non-zero before performing an operation. With static sparse matrices, the zero and non-zero elements are known in advance, and the non-zero elements can be accessed directly without performing the zero-test first.

Because of the variability in static sparse matrices, these data types cannot be captured as Ada generics — the generic CAMP parts can, at best, incorporate dynamically sparse matrix types. Incorporation of static sparse matrix types is limited to Kalman filter components generated either by hand or by the Kalman filter constructor. During the CAMP domain analysis, the Kalman filter experts indicated that statically sparse matrices were generally used, and that their development was both time-consuming and error-prone. It is possible to capture the algorithm for generating both the data type definition and the operators, thus a constructor was developed to facilitate the process.

19.1.1.2 Tailoring

The Kalman filter constructor allows the user to tailor the CAMP Kalman filter parts by supplying application-specific data type information. In the prototype implementation, the data type information can be supplied in one of two ways: (1) by the constructor with the user providing a small amount of application-specific tailoring information (referred to as the *default data type* approach), or (2) via interactive definition of the types with the user who is prompted for all of the information required to define all of the data types.

Using the *default data type* approach, the user must provide the following information:

- The precision of the floating point type that is used for the matrix elements
- The number of states, and if he wants to refer to them by number or name
- The names of the states (if he has indicated reference by name)
- The number of measurements, and if he wants to refer to them by number or name
- The names of the measurements (if he has indicated reference by name)

The user's requirements lead to the definition of data types for the matrix elements and the matrix indices (states and measurements); all of the matrix types are produced via instantiations of other CAMP generic components.

If the user wants to define the data types himself, type definitions are generated for any matrix that the user declared as statically sparse; all of the required operations for these types are also generated. These type definitions are based on input received from the user on which matrix elements are non-zero. The static sparse

matrix is declared as a record of non-zero elements. Operators are produced that allow multiplication between the static sparse matrix types and other matrix types. As in the first case, any non-static-sparse matrix types are still produced via instantiations of CAMP matrix parts.

In order to facilitate the use of the constructed code, a help or summary file is also produced. This file contains information on the names and locations of the generated code, and on what packages need to be *with'd* into the user's application in order to make use of this code. The Kalman filter operations come from the CAMP generic Kalman filter parts. The real variability of the Kalman filter is captured in the data types package.

19.2 Future Directions

For now, the future trends in software generation/composition point to more domain-specific tools rather than generalized code generation. By limiting the scope of the problem, much more can be accomplished, thus we will undoubtedly see more emphasis on software generation in narrow, high-potential domains. Generation will probably expand to include both code and documentation. There will also be greater emphasis on integrating this type of tool into a software development environment rather than providing it as a stand-alone tool.

APPENDICES

APPENDIX I

OVERVIEW of the CAMP PROGRAM

The Common Ada Missile Packages (CAMP) program is a contracted research and development effort that grew out of the so-called DoD *software crisis*. This crisis has been characterized by escalating software costs, delays in deliveries, products of questionable quality, and a shortage of adequately trained personnel. Two initiatives that have potential for partially ameliorating the crisis are programming language standardization and software reuse. The DoD has provided Ada as the standard programming language for mission-critical applications, and significant work is being done in software reuse.

The main goal of the CAMP program has been to establish the feasibility and value of reusable Ada software within mission-critical real-time domains. This has required a careful examination of a particular domain, the development of reusable Ada components, the development of automated support for software reuse in the software development lifecycle, and the application of both the reusable components and the automated tools to a realistic application. Although the CAMP project was specifically tasked with exploring the missile operational flight software domain, there is significant carry-over of lessons-learned to other RTE domains. The CAMP program has been sponsored by the U.S. Air Force Armament Laboratory at Eglin AFB.

I.1 Phase 1: Feasibility Study

The CAMP program began in 1984, with a 12-month feasibility study. There were two major objectives: (1) to determine if sufficient commonality existed within the missile operational flight software domain to warrant the development of reusable software parts, and if commonality was found, to identify and specify the parts; and (2) to determine the aspects of parts engineering that could be fully or partially automated, and to develop the requirements and top-level design for a parts composition system to support reuse.

The major tasks performed during CAMP-1 are described below.

- Domain Analysis: A domain analysis was conducted to determine if sufficient commonality existed to warrant the development of reusable software parts. This analysis involved the examination of documentation and source code for 10 existing missile software systems. The result of this analysis was the identification of over 200 common operations, objects, and structures that could be captured as reusable parts. A Software Requirements Specification (SRS) was prepared for these parts in accordance with DOD-STD-2167.
- Ada Parts Design: An architectural design was developed for each part that was identified and specified. The design was documented in a Software Top-Level Design Document (STLDD) in accordance with DOD-STD-2167.
- Part Composition System (PCS) Investigation: The purpose of this investigation was to determine the aspects of software engineering with parts that could be automated. The result of this investigation was the development of an SRS for a prototype tool called the Ada Missile Parts Engineering Expert (AMPEE) system. The goal of this tool was to assist a software engineer in identifying, locating, understanding, using, and managing the reusable Ada missile parts.
- AMPEE Design: After the requirements were specified, the architectural design of the AMPEE system was developed and documented in a STLDD.

I.2 Phase 2: Technology Demonstration

While CAMP-1 concentrated on feasibility analyses, Phase 2 of the CAMP program (CAMP-2) was a 32-month technology demonstration phase that began in September 1985. The goal of CAMP-2 was to demonstrate the technical feasibility and value of reusable Ada missile parts and a PCS by building and using them on a realistic

application. Both the parts and the PCS were coded, tested and then used in the "11th Missile" testbed program — a program aimed at validating software reuse in a realistic RTE application. Benchmarks were also developed for evaluating the CAMP parts and Ada compilers. The main CAMP-2 tasks are described below.

- **Parts Construction:** During this task, the detailed design of the parts that were identified during CAMP-1 was developed. The parts were then coded and tested. During this task, approximately 200 additional parts were identified and implemented, bringing the total number of parts developed to 454.
- **AMPEE System Construction:** During this task, the detailed design of the prototype parts composition system was developed, and the system was coded and tested. The system was prototyped in LISP and ARTTM on a SymbolicsTM workstation. The AMPEE system consisted of a parts catalog, a parts exploration facility, and a set of component constructors. The catalog provided a means to identify and locate parts. The parts exploration facility provided the user with high-level access into the parts catalog before he had detailed knowledge of the types of parts he needed. This facility allowed the user to base his queries on application or domain knowledge rather than specific parts knowledge. The component constructors were developed to assist the user in tailoring and composition of software parts for a particular application.
- **11th Missile Application Development:** The "11th Missile" application was so-called because it was not one of the original 10 missile systems that were examined during the CAMP-1 domain analysis. This task required the construction of the navigation and guidance systems of an actual missile application using the CAMP Ada parts and the AMPEE system, and testing of the developed system in a MIL-STD-1750A hardware-in-the-loop simulation. It served to validate the concept of a parts-based approach to software development. Although not a full-scale engineering development, the 11th Missile software was documented in accordance with DOD-STD-2167.
- **Armonics Benchmarks:** The purpose of this task was to use the CAMP parts to develop a suite of benchmarks that could be used to measure the effectiveness and efficiency of Ada compilers for armonics applications. The benchmarks provide information on compilation speed of selected parts and on the object code size of these parts after compilation. They also provide data on execution speed and accuracy of operations performed by the parts. There are three categories of benchmarks: Integrated Execution Benchmarks, Compilation Benchmarks, and Computation Benchmarks. The Integrated Execution Benchmarks are a set of 3 benchmarks designed to measure the effectiveness and efficiency of Ada compilers in complex real-time applications such as guidance and navigation operations. The Compilation Benchmarks are a set of 3 benchmarks designed to determine the effectiveness of Ada compilers in handling non-trivial use of Ada features, such as Ada generics. The Computation Benchmarks are a set of 11 benchmarks designed to measure the effectiveness and efficiency of Ada compilers in dealing with computationally intense functions.

I.3 Phase 3: Technology Transfer and Refinement

The primary goals of Phase 3 (CAMP-3) are technology transfer and technology refinement. The main tasks are described below.

- **CAMP Parts Set Maintenance and Enhancement:** This task called for the enhancement, expansion, and maintenance of the CAMP-2 Ada parts set. As a result of this effort, 32 existing parts were modified and 50 new parts were added, bringing the total CAMP parts count to 500+.
- **Re-engineering of the Parts Engineering System:**
 - The CAMP-2 prototype catalog was re-engineered in Ada to enhance broad usability. To this end, no commercial third-party software was incorporated. This effort required development of the code, associated documentation (SRS and SDD), and a user's manual. It also required a four-month maintenance and distribution task during which the parts catalog software and user's manual were distributed to 100 customer-approved government agencies and contractors.

- The parts exploration facility and constructors are also being re-engineered. The parts exploration facility will be more readily adaptable to domains other than the missile operational flight software domain that was examined during CAMP. A new approach to component constructors is also being explored. Although the CAMP-2 approach demonstrated the feasibility and value of constructors, it was costly to implement and maintain. During CAMP-3, a *meta-constructor* that can be used in the development of component constructors will be prototyped. Both the re-engineered parts exploration facility and the meta-constructor will be developed in Ada and will include a DOD-STD-2167A SRS and SDD and development of a user's manual.
- **Technology transfer:** This effort includes development and distribution of this manual, development and distribution of an executive overview videotape that covers the software crisis and the opportunity that this has created for Ada and software reuse, and a presentation/demonstration at SIGAda.

Figure I-1 identifies the CAMP products.

CAMP Ada Parts • Domain Analysis • Software Requirements Specification • Top-level and Detailed Design Documents • Test Plan • Test Procedure • Version Description Document • User's Guide • Parts Catalog • Ada parts (500+)	11th Missile • Software Requirements Specification • Top-level Design Document • Test Plan and Report	Parts Engineering System • Prototype tool set and documentation (SRS, TLDD, SDDD, Test Plan, User's Guide) • Re-engineered tool set and documentation (SRS, DDD, User's Guide)
	Armonics Benchmarks	
	Final Technical Report (CAMP-1 & CAMP-2)	
	Parts Development and Use Manual	
	Executive Overview Videotape	

Figure I-1: CAMP Products

APPENDIX II

SUMMARY of the CAMP 11TH MISSILE APPLICATION

The CAMP 11th Missile Application was a testbed effort undertaken during Phase 2 of the CAMP program. It involved the construction of an actual missile application using the CAMP Ada parts and the CAMP prototype Parts Composition System (PCS), and testing of the developed system in a MIL-STD-1750A hardware-in-the-loop simulation. The initial goals of the application were as follow:

- Construct a complete missile application using CAMP parts and the PCS, and test it in a MIL-STD-1750A hardware-in-the-loop simulation
- Test the CAMP parts and the PCS, and recommend corrections and improvements
- Evaluate the suitability of the CAMP parts and the PCS for real-time embedded missile applications
- Quantify the productivity improvement attributable to the use of CAMP parts and the PCS

Although not explicitly stated as goals, the 11th Missile Application also served as the basis for (1) an evaluation of the suitability of Ada for real-time embedded missile applications, and (2) an evaluation of the suitability of an Ada/1750A compiler for real-time embedded applications.

The 11th Missile task required development of both code and DOD-STD-2167 documentation, including a Software Requirements Specification, a Top-Level Design Document, a Software Test Plan, and a Software Test Report. At completion, the code consisted of approximately 21,000 lines of Ada, and 21 lines of assembly code.

II.1 What Is the 11th Missile?

During the CAMP domain analysis, the software from 10 missile systems was examined. In order to test the CAMP parts and prototype tool set in a realistic setting, an "11th Missile" was held in reserve during the CAMP domain analysis, and a portion of its software was implemented using them. The 11th Missile Application was based on a cruise missile application that was originally implemented in JOVIAL J73. The original application had five MIL-STD-1750A processors and an Inertial Sensor Assembly (ISA), which communicated by means of a MIL-STD-1553B data bus (see Figure II-1).

The shared memory contained terrain altitude data. The processors and their primary functions are described in Table II-1. In addition to those functions, all processors were programmed to (1) restart the application program, (2) return control to start-up ROM, (3) communicate via the 1553B bus, and (4) issue periodic status messages.

The 11th Missile Application was a re-implementation (starting with a new requirements specification) of the navigation, ground alignment, Kalman filtering, ISA interface, lateral guidance, lateral-directional autopilot, and support functions of the original software. The navigation, Kalman filtering, lateral guidance, and lateral-directional autopilot were chosen because there were CAMP parts to support those functions. The other functions were required to complete a functioning computer program.

Two versions of the 11th Missile Application were written. The first version ("Parts Method") was written using the CAMP parts without assistance from the CAMP PCS. The second version ("PCS Method") used the PCS to generate most of the Kalman filter code. The PCS Method implementation was not a complete rewrite of the code; the PCS-generated Kalman filter code was integrated with the rest of the Parts Method code and unit tested.

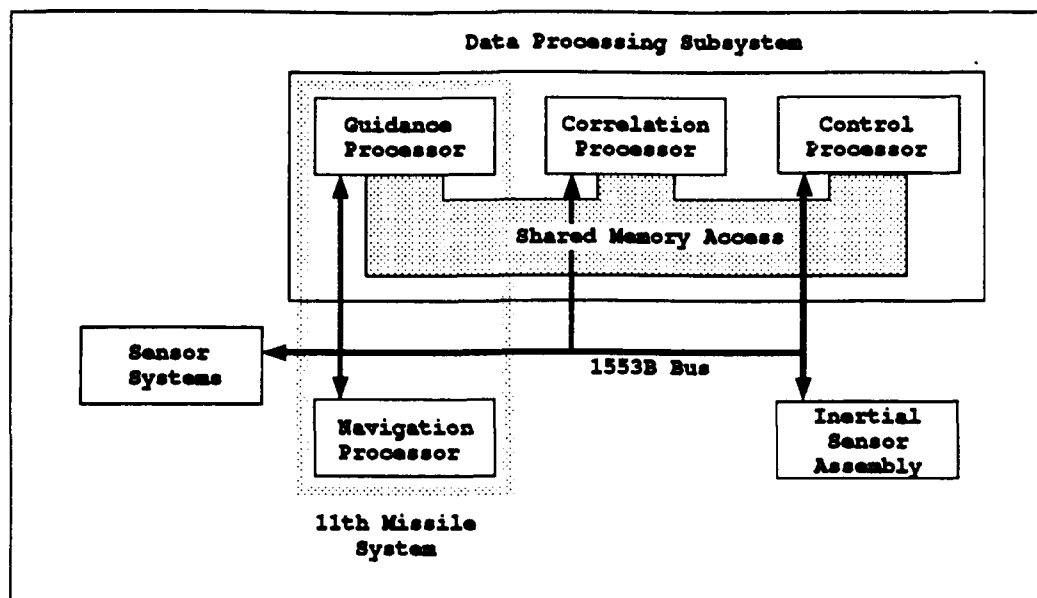


Figure II-1: 11th Missile Hardware Design

Table II-1: Processors and Their Functions

PROCESSOR	FUNCTIONS
Control	Communicates with operator console Downloads, starts, and stops software in all other machines Mode logic
Navigation	Wander-azimuth navigation Transfer alignment Ground alignment 21-state Kalman filter Start up, test, and communicate with ISA
Guidance	Waypoint-steering lateral guidance Vertical guidance Lateral-directional autopilot
Correlation	Dedicated to Terrain Profile Matching
Sensor	Controls sensor system hardware

II.2 Requirements

The 11th Missile navigation requirements came from several sources. The existing application's navigation requirements were documented in an out-of-date MIL-STD-1679 Computer Program Development Specification (CPDS); it served primarily as an outline of the high-level requirements. There was also a MIL-STD-1679 Computer Program Product Specification (CPPS), which included Ada Design Language (ADL). Interviews with the original software developers were held to determine which requirements had changed since the CPPS; in these cases, the requirements were abstracted from the updated ADL or, occasionally, from the JOVIAL code.

The guidance requirements for the existing application were specified in a MIL-STD-483 B-5 development specification. The lateral guidance and lateral-directional autopilot algorithms were reused from the Medium Range Air-to-Surface Missile (MRASM) program, therefore, these requirements were stable and the specification was up-to-date and complete.

The 1553B bus protocols and the formats of all bus messages were specified in a database maintained by the original application. That project used the database to automatically generate the JOVIAL code that specified the message formats for the application programs, the FORTRAN code for the real-time-simulation software, and the interface requirements specification. This database was used during the 11th Missile Application to get up-to-date message specifications and change notices.

II.3 Results

Various compiler problems were encountered during the 11th Missile development effort, but they all had one thing in common — they involved the use of generic units. Some of these problems are enumerated below.

- Difficulties in handling instantiations. One compiler was able to compile all of the CAMP parts required to develop the application, but when the entire application was compiled, the compiler crashed. The application did compile on another validated compiler.
- Inability to resolve overloading of operators when a generic formal subprogram ("+" in this case) matched an operator already defined by the language (see Figure II-2).
- Inability to identify generic actual subprograms to be used as defaults even though they were directly visible. Two variations of this problem occurred and are illustrated in Figure II-3. In the first, the correct subprogram was directly visible as the result of *with* and *use* clauses on the subprogram's package. In the second case, the correct subprogram was directly visible since it was a generic actual subprogram to the part where problems were encountered.
- An inability to handle separate compilation of generic units, even though compiler documentation indicated this optional feature was implemented.

Data collected during this effort indicated that with a mature compiler, use of the CAMP parts on this application could result in a productivity improvement of up to 15%. This figure is based on an adjusted testing hour total (4779 actual hours for testing vs. 2630 adjusted hours), where adjustments were made for the effort expended on compiler problems (see Reference [29] for more details). The adjusted test effort estimate of 2630 hours is 45% of the adjusted total effort, which is roughly in line with the 40%-design/20%-code/40%-test rule of thumb. Table II-2 shows effort data for the 11th Missile Application.

Table II-3 shows the size of the 11th Missile software in both lines-of-code (LOC) and Ada statements. The "generated" code consisted of two sparse matrix operators that were generated using portions of preliminary versions of a CAMP-developed software tailoring tool. Parts statement counts were estimated from the overall ratio of statements to LOC for the parts (0.634 statements/LOC). The "other reused" test software was FORTRAN and assembler code used for the hardware-in-the-loop tests. It was estimated that there were 0.9 statements/LOC for this software. Table II-4 shows the actual productivity.

CAMP parts constituted 18.1% of the Parts Method implementation of the 11th Missile code and 3.1% of the test code (Table II-3). Using the adjusted effort as a basis, the 11th Missile developers saved 896 work-hours — 15% of the development effort — by using the CAMP parts. Table II-5 compares the adjusted effort with the estimated effort required had the parts not been used.

Specification:

```

generic
  type M_Indices      is (<>);
  type N_Indices      is (<>);
  type P_Indices      is (<>);
  type Left_Elements  is digits <>;
  type Right_Elements is digits <>;
  type Result_Elements is digits <>;
  type Left_Matrices  is array (M_Indices, N_Indices) of Left_Elements;
  type Right_Matrices is array (N_Indices, P_Indices) of Right_Elements;
  type Result_Matrices is array (M_Indices, P_Indices) of Result_Elements;
  with function "*" (Left : Left_Elements;
                    Right : Right_Elements) return Result_Elements is <>;
  with function "+" (Left : Result_Elements;
                    Right : Result_Elements) return Result_Elements is <>;
function Matrix_Matrix_Multiply
  (Left : Left_Matrices;
   Right : Right_Matrices) return Result_Matrices;

```

Body:

```

function Matrix_Matrix_Multiply
  (Left : Left_Matrices;
   Right : Right_Matrices) return Result_Matrices is
  Answer : Result_Matrices;
begin
  for M in M_Indices loop
    for P in P_Indices loop
      Answer(M,P) := 0.0;
      for N in N_Indices loop
        Answer(M,P) := Answer(M,P) +
          Left(M,N) * Right(N,P);
      end loop;
    end loop;
  end loop;
  return Answer;
end Matrix_Matrix_Multiply;

```

- Compiler was unable to resolve this overloading

NOTE: Constrained arrays were used in the design of this part in order to improve the efficiency of the part. While it was recognized that unconstrained arrays would have made the part more flexible and hence more reusable, the need for efficiency for real-time embedded applications was considered of greater importance.

Figure II-2: Overloaded Operator Caused Problems for Compiler

When attempting to instantiate the following generic:

```
generic
  type Angles      is digits <>;
  type Inputs      is digits <>;
  type Outputs     is digits <>;
  type Sin_Cos_Ratio is digits <>;
  with function Sin (Input : Angles) return Sin_Cos_Ratio is <>;
function Example (Input : Inputs) return Outputs;
```

One compiler couldn't resolve the default even though the appropriate subprogram was directly visible through 'with' and 'use' clauses:

```
generic
  type Angles is digits <>;
  type Ratios is digits <>;
package StdTrig is
  type Radians      is new Angles;
  type Sin_Cos_Ratio is new Ratios range -1.0..1.0;
  function Sin (Input : Radians) return Sin_Cos_Ratio;
end StdTrig;

with StdTrig;
package BDT is
  type Real is digits 9;
  package Trig is new StdTrig (Angles => Real,
                               Ratios => Real);
end BDT;

with BDT; use BDT;
with Example;
procedure User_Application is
  use BDT.Trig;

  function Attempted_Instantiation is new Example
    (Angles      => BDT.Trig.Radians,
     Inputs      => BDT.Real,
     Outputs     => BDT.Real,
     Sin_Cos_Ratio => BDT.Trig.Sin_Cos_Ratio);

begin
  ...
end User_Application;
```

\ } problem encountered
 } with this
 / instantiation

Another compiler couldn't resolve the default even though it was visible as a generic formal subprogram:

```
generic
  type Angles      is digits <>;
  type Inputs      is digits <>;
  type Outputs     is digits <>;
  type Sin_Cos_Ratio is digits <>;
  with function Sin (Input : Angles) return Sin_Cos_Ratio is <>;
package Sample is
  ...
end Sample;

with Example;
package body Sample is
  function Attempted_Instantiation is new Example
    (Angles      => Angles,
     Inputs      => Inputs,
     Outputs     => Outputs,
     Sin_Cos_Ratio => Sin_Cos_Ratio);

end Sample;
```

\ } problem encountered
 } with this
 / instantiation

Figure II-3: Compilers Had Problems Finding Default Subprograms

Table II-2: 11th Missile Effort

Phase	Effort (hours)
Requirements	708
Architectural Design	883
Detailed Design & Code	1306
Testing	4779 (actual)/2630 (adjusted)
Other	371
Total	8047 (actual)/5898 (adjusted)

Table II-3: 11th Missile Size - Parts Method

	Lines of Code	State- ments
Operational Code		
New	15708	8697
Generated	1108	471
Mod. Parts	897	458
Parts	3911	2480*
Total	21624	12106
Test Software		
New	19752	12605
Parts	1114	706*
Other Reused	14544	13090*
Total	35410	26401
* Estimated		

Table II-4: 11th Missile Productivity - Parts Method

	New Operational	All Operational	New Developed	All Developed
LOC/Work-Month	304.5	419.2	687.4	1105.7
Stmt/Work-Month	168.5	234.7	413.0	746.5
Work-Hours/LOC	0.51	0.37	0.23	0.14
Work-Hours/Stmt	0.93	0.66	0.38	0.21

Table II-5: Effect of Parts on 11th Missile Effort

Phase	Effort (hours)	
	With Parts (Adjusted)	Without Parts (Estimated)
Requirements	708	708
Architectural Design	883	883
Detailed Design & Code	1306	1604
Testing	2630	3228
Other	371	371
Total	5898	6794
Effort Saved: 896 hours		
Productivity Improvement: 15%		

APPENDIX III PARTS DESIGN ALTERNATIVES

Six methods for the design of reusable Ada parts were investigated during the CAMP program. Figure III-1 illustrates these methods. As a result of this investigation, a hybrid approach known as the semi-abstract data type method was developed and used on the CAMP program. The semi-abstract data type method is based on the generic and overloaded methods. The six basic methods that were investigated will be described in this appendix; the semi-abstract method is described in Chapter 5.

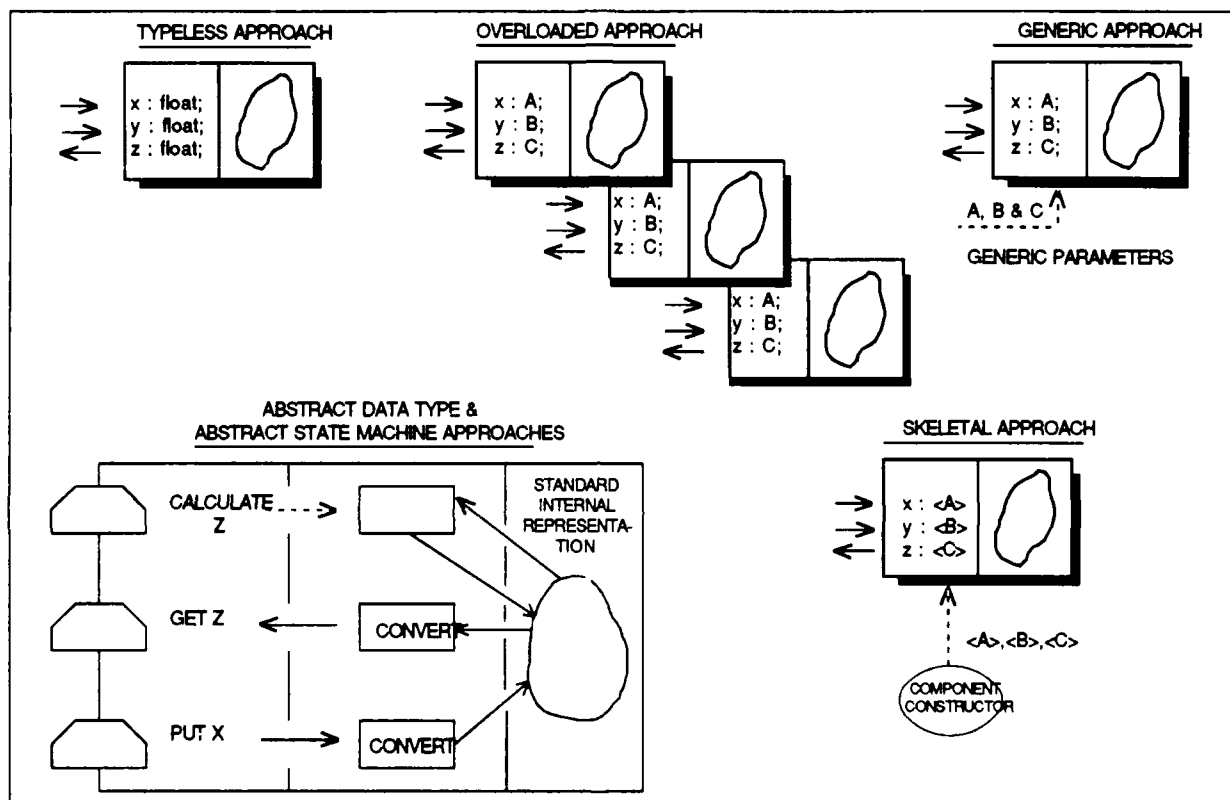


Figure III-1: Reusable Parts Design Methods

In discussing the alternative methods, a specific CAMP part will serve as an example. This part, `Compute_Earth_Relative_Horizontal_Velocities`, has three inputs:

- Nominal_East_Velocity (VEL_{NE})
- Nominal_North_Velocity (VEL_{NN})
- Wander_Angle (WA)

It processes these inputs through the following equations:

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

producing the following outputs:

- True East Velocity (VEL_E)
- True North Velocity (VEL_N)

The computations performed in this example require trigonometric functions on *Wander_Angle* plus multiplication and subtraction operators. In addition, the multiplication operator must perform its operation on data of different types, namely, a velocity type and a sine- or cosine-of-an-angle type. These mathematical functions must also be provided for all possible combinations of data types for velocities and angles. For example, if velocity is measured in feet-per-second and angle is in radians, the following mathematical operations are required:

- Sine and cosine operations on radians;
- Multiplier for feet-per-second by the result of the sine and cosine operations; and,
- Subtractor for the result type of the multiplier.

The discussion will explain the methods for parts design and evaluate their effectiveness with regard to the following four evaluation criteria:

1. Efficiency and appropriateness of the interface;
2. Control for preventing misuse;
3. Availability of required mathematical operators and functions; and,
4. Degree to which the user's job is simplified.

III.1 Typeless Method

The *typeless* method assumes that all data objects and actual parameters will be of some universal type. In the case of the CAMP program, which was dealing with the missile operational flight software domain, the assumption was that if the typeless method were used, the data objects and actual parameters would be of the universal float type. The benefit of this approach is that it alleviates the need for special mathematical operators and functions since they are already defined in standard packages. The disadvantage is that the compiler and run-time system cannot perform type checking to prevent misuse of the part.

For example, consider the case of an SDI-related experiment in 1985. The experiment required an orbiting receiver to track a ground-based laser. The transmitter was positioned at an elevation of approximately 10000 feet; this elevation was to be entered into the flight computer of the receiver's orbiting platform. The flight computer was programmed to accept ground elevation in *nautical miles* not *feet*, however, so when 10000 was entered, the platform oriented the receiver to point to a position 10000 miles above the surface of the earth, exactly 180 degrees from the correct location, 10000 *feet* above the surface. Strong typing of parameters could prevent this type of problem. Figure III-2 illustrates the data type and object declarations which could be used. This would restrict the input values of *Transmitter_Elevation* to a reasonable range for units of nautical miles.

```
type Nautical_Miles is digits 6;  
  
subtype Ground_Elevation is Nautical_Miles range -1.0 .. 6.0;  
  
Transmitter_Elevation : Ground_Elevation;
```

Figure III-2: Strong Data Typing Example

The specification shown in Figure III-3 illustrates the application of the typeless method to the sample part. A user application accessing this procedure could pass any object of the type *FLOAT* as actual parameters. The compiler could not perform type checking to prevent possible type mismatching and there could be no run-time

checking to ensure correct ranges for the actual parameters. This method produces parts which are easy to use, but offers no protection against misuse.

```

procedure Compute_Earth_Relative_Horizontal_Velocities
(Nominal_East_Velocity : in    FLOAT;
 Nominal_North_Velocity : in    FLOAT;
 Wander_Angle          : in    FLOAT;
 East_Velocity         : out   FLOAT;
 North_Velocity        : out   FLOAT);

```

Figure III-3: Typeless Method Example

III.2 Overloaded Method

To allow a greater choice in data typing, 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. The code segment shown in Figure III-4 illustrates the overloaded method applied to the example when Wander_Angle is in Radians and the velocities are of type Feet_Per_Second and Meters_Per_Second.

```

package Overloaded_Method is

  procedure Compute_Earth_Relative_Horizontal_Velocities
    (Nominal_East_Velocity : in    Feet_Per_Second;
     Nominal_North_Velocity : in    Feet_Per_Second;
     Wander_Angle          : in    Radians;
     East_Velocity         : out   Feet_Per_Second;
     North_Velocity        : out   Feet_Per_Second);

  procedure Compute_Earth_Relative_Horizontal_Velocities
    (Nominal_East_Velocity : in    Meters_Per_Second;
     Nominal_North_Velocity : in    Meters_Per_Second;
     Wander_Angle          : in    Radians;
     East_Velocity         : out   Meters_Per_Second;
     North_Velocity        : out   Meters_Per_Second);

end Overloaded_Method;

```

Figure III-4: Overloaded Method Example

Other overloaded subprograms would allow Wander_Angle in degrees and semicircles. This is the method used by Ada packages such as STANDARD, CALENDAR, and TEXT_IO to provide identical operations on different data types.

The overloaded method offers the protection of strong data typing with simplicity of design and use of parts. The designer will decide which combinations of data types to allow for each part and will explicitly declare the parameter interfaces for each overloaded subprogram. He will also define all of the mathematical parts which the subprograms will use: sine and cosine for Wander_Angle, and the multiplication and subtraction operators. Ada type checking will ensure 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 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. This method provides simplicity of use with the protection associated with strong data typing.

The major disadvantage of this method is the large number of parts declared at the architectural level. For the data types stated above (Feet_Per_Second and Meters_Per_Second for velocities and Radians, Degrees, and Semicircles for angles), the `Compute_Earth_Relative_Horizontal_Velocities` procedure would require six specifications and bodies to accommodate the different combinations of data types. A navigation package encapsulating a complete set of reusable navigation parts could easily grow to over 100 subprograms. Thus, the overloaded method, while simple to use, would be almost impossible to develop and maintain.

III.3 Generic Method

The *generic* method uses Ada generic units to provide parts which are tailorable to user-defined data types. Figure III-5 shows the generic method applied to the `Compute_Earth_Relative_Horizontal_Velocities` procedure using *generic formal types* for Velocities, Angles, and Sin_Cos_Ratio (the type returned by a call to Sine or Cosine), and *generic formal subprograms* for the required trigonometric functions and the multiplication operator. The subtraction operator operates only on the generic velocity type and is implicit from the generic definition.

```
generic
  type Sin_Cos_Ratio is digits <>;
  type Velocities    is digits <>;
  type Angles        is digits <>;
  with function "*" (Left  : Velocities;
                    Right : Sin_Cos_Ratio) return Velocities is <>;
  with function Sin  (In_Angle : Angles)    return Sin_Cos_Ratio is <>;
  with function Cos  (In_Angle : Angles)    return Sin_Cos_Ratio is <>;
  procedure Compute_Earth_Relative_Horizontal_Velocities
    (Nominal_East_Velocity : in Velocities;
     Nominal_North_Velocity : in Velocities;
     Wander_Angle         : in Angles;
     East_Velocity        : out Velocities;
     North_Velocity       : out Velocities);
```

Figure III-5: Generic Method Example

The generic formal subprogram parameters are used within the body of the part to perform mathematical operations on objects of the generic data types. For example, the sine and cosine operations on `Wander_Angle` are performed by the functions supplied as actual parameters for the generic `Sin` and `Cos`. The user must define operators to perform these functions on objects of the actual type for Angles returning an object of the `Sin_Cos_Ratio` type. This large number of generic parameters could place an enormous burden on the part user, requiring him to create and supply all of the needed actual parameters, both types and subprograms. For the example, the user must supply three data types plus three subprograms as actual parameters.

A method which uses default parameters could alleviate some of this overhead from the part's user. If the total parts design includes typical data types and provides functions for typical combinations of these data types, then the user could provide predefined types as actual type parameters and the actual subprogram parameters will default to the predefined functions. Figure III-6 illustrates this method. Using the same example, the design could incorporate a separate data types part supplying a `Radians` type and trigonometric operations on `Radians`. The multiplication operator could be similarly predefined. This approach yields a *tunneling* of parameters, where explicit use of a type (e.g., `Radians`) allows tunneling of operators (e.g., sine, cosine, `"*"`) on

that type. The advantage of this method is clear: the user obtains the protection associated with strong data typing and the flexibility of using a choice of data types without the need to define his own types or operators.

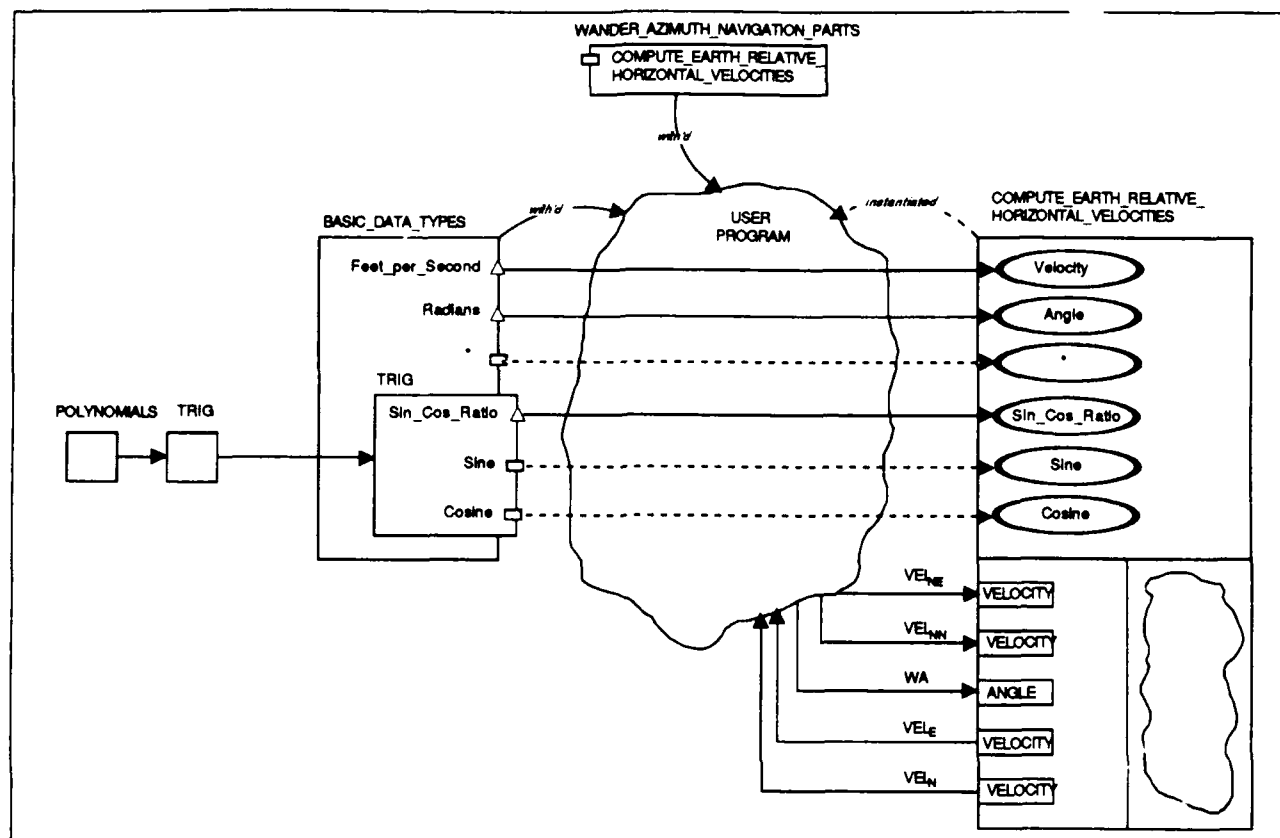


Figure III-6: Tunneling of Parameters

III.4 Abstract State Machine Method

The *abstract state machine* method affords the part user a very high-level interface to reusable parts. In this method, the interface is a package structure defining all of the characteristics of the missile state relevant to a group of navigation operations. The interface is strictly through the operations, as the user does not have direct access to or knowledge of the data structure on which the operations work (Reference [5], p. 202). The state machine allows the underlying structure to change without the users' knowledge.

The state machine implementation of a navigation system would provide all of the operations needed to perform the navigation function, both those changing the state and those reporting the state. One such function would be the *Compute_Earth_Relative_Horizontal_Velocities* operation which would both update and report the velocity. Figure III-7 contains a code segment to illustrate the abstract state machine method.

The abstract state machine approach utilizes generic units to tailor operations to the user's requirements. Like the generic method, this approach enforces strong data typing and provides protection against misuse. However, because all operations are encapsulated in a single package, the user is presented with an "all or nothing" solution: specify all of the generic parameters for all operations, whether needed or not, or don't use the package.

```
generic
  type Sin_Cos_Ratio is digits <>;
  type Velocities   is digits <>;
  type Angles       is digits <>;
  type Altitudes    is range <>;
  with function "*" (Left : Velocities;
                    Right : Sin_Cos_Ratio) return Velocities is <>;
  with function Sin (In_Angle : Angles) return Sin_Cos_Ratio is <>;
  with function Cos (In_Angle : Angles) return Sin_Cos_Ratio is <>;
  . . .
package Navigation_State_Machine is

  procedure Compute_Earth_Relative_Horizontal_Velocities
    (Nominal_East_Velocities : in Velocities;
     Nominal_North_Velocities : in Velocities;
     Wander_Angle : in Angles;
     East_Velocities : out Velocities;
     North_Velocities : out Velocities);

  procedure Update_Altitude
    (Vertical_Velocities : in Velocities;
     Current_Altitude : out Altitudes);

  -- --operations to provide state information

  function Current_East_Velocities return Velocities;

  function Current_North_Velocities return Velocities;

end Navigation_State_Machine;
```

Figure III-7: Abstract State Machine Method Example

An alternative approach is to encapsulate the data typing and structure within the package, forcing the part user to convert his types to conform to those provided by the abstract state machine. While defining all internal data types and operations makes the part easier to use, the overhead of conversion to the Internal data structure would be prohibitive. This conversion would entail not only data typing, but also unit conversion, from meters to feet, radians to degrees, etc.. The package could provide interfaces to simplify the unit conversion, but could do little to alleviate the overhead.

The state machine approach does offer an advantage of creating 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 needs. 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. Like the overloaded method, this increases the cost of creating parts, yet is an effective method when the choice of a data structure cannot, for reasons of efficiency or simplicity, be established in the package specification.

III.5 Abstract Data Type Method

Like the abstract state machine method, the *abstract data type* method offers the part user a high-level interface to reusable parts. This interface consists of a predefined set of operations on a data structure, and, unlike the abstract state machine, the interface includes the data structure itself. The user, therefore, knows the structure he is dealing with and, depending on the implementation, may even be able to access the structure directly. In the abstract data type method the user is aware of changes to the state of the structure which are affected by the exported operations.

In most implementations, access to objects of the abstract type are restricted to operations defined in the package specification. In contrast to the abstract state machine, this type definition is part of the specification, and the package body must operate on that unique structure. If a part user wants to use the operations of the abstract data type but use a different data structure, he must not only rewrite the body which will operate on the data structure, but also rewrite the specification that defines the structure. This method is used extensively in abstract data structures such as vectors and matrices, stacks and queues, but is less appropriate for more complex data structures such as those used by a navigation system or Kalman filter.

A package which implements the navigation system according to the abstract data type method looks quite similar to that of the abstract state machine (see Figure III-8). The major distinction is the **private** section of the specification which defines the abstract data structure.

```

generic
  type Sin_Cos_Ratio is digits <>;
  type Velocities   is digits <>;
  type Angles       is digits <>;
  type Altitudes    is range <>;
  with function "*" (Left : Velocities;
                    Right : Sin_Cos_Ratio) return Velocities is <>;
  with function Sin  (In_Angle : Angles)      return Sin_Cos_Ratio is <>;
  with function Cos  (In_Angle : Angles)      return Sin_Cos_Ratio is <>;
  . . .
package Navigation_State_Machine is

  type Navigation_Model is private;

  procedure Update_Earth_Relative_Horizontal_Velocities
    (Nominal_East_Velocity : in Velocities;
     Nominal_North_Velocity : in Velocities);

  procedure Compute_Earth_Relative_Horizontal_Velocities
    (Updating : in out Navigation_Model);

  procedure Update_Vertical_Velocity
    (Vertical_Velocity : in Velocities);

  procedure Compute_Altitude
    (Updating : in out Navigation_Model);

  -- --operations to information from data structure

  function Current_East_Velocity
    (Based_On : Navigation_Model) return Velocities;

  function Current_North_Velocity
    (Based_On : Navigation_Model) return Velocities;

  private -- Definition of Navigation Abstract Data Structure

  type Navigation_Model is
    record
      Missile_Velocity : Velocities;
      Missile_Altitude : Altitudes;
      . . .
    end record

end Navigation_State_Machine;

```

Figure III-8: Abstract Data Type Method Example

This method is similar to the abstract state machine approach in that it utilizes generic units to tailor operations to the user's requirements. It uses these generic units to enforce strong data typing and protect against misuse. The drawback is that the user is again presented with an "all or nothing" solution. The abstract data type method offers an alternative approach in which the data types are defined in the package specification, eliminating all of the generic units. While defining all internal data types and operations will ease the use of the part, the overhead of conversion to the internal data structure would be prohibitive.

III.6 Skeletal Code Method

The *skeletal code* method provides the part user with code templates, which may be manipulated in an editor or through some other tool. This approach gives the part user the flexibility of generic units, without the complexity of the generic instantiation. A sample template, as shown in Figure III-9 for the Compute_Earth_Relative_Horizontal_Velocities, would look similar to the code for the typeless method.

```
procedure Compute_Earth_Relative_Horizontal_Velocities
  (Nominal_East_Velocity : in ____;
   Nominal_North_Velocity : in ____;
   Wander_Angle         : in ____;
   East_Velocity         : out ____;
   North_Velocity        : out ____);
```

Figure III-9: Skeletal Code Template Method Example

This approach adds complexity by requiring the part user to complete much of the environment. Outside the part, he must edit the skeletal code into his existing design, inserting data types and overloaded operators as required. While the generic method provides a generic *specification*, and forces conformity through the Ada generic matching rules, the skeletal method can only provide user documentation to support creation of the environment. If two or more designers are using similar parts, they may choose different values for completing the templates, duplicating parts of the environment. There would also be a tendency to avoid strong data typing to alleviate the overhead attached to creation of overloaded operators and functions.

An expert system, interfacing to the code templates, could support use of the skeletal code method. The expert system could prompt the user for information it needs to fill in the blanks, but rules, stored in the expert system knowledge base, would allow the system to complete the environment, filling in additional types, operators, and any additional subprograms. The expert system approach appears to offer a long-term solution to the difficulties of the skeletal method by building the environment as a by-product of a user dialog.

III.7 Summary

The six approaches that were evaluated, as well as the approach that was derived from these, are summarized below.

- **Typeless:** Simple to design and use, but not robust. All objects are declared floating point types so there is no error checking on data types.
- **Overloaded:** There is a separate version of each part to allow for the different combinations of data types that the user might want; this is the method used in the Ada packages STANDARD and CALENDAR. Parts are simple to design and use — the designer decides the combinations of data types that will be allowed for each part and explicitly declares parameter interfaces for these overloaded subprograms; the Ada disambiguation facility resolves calls so the user does not have to specify which version of a part he wants. Strict type checking ensures that actual and formal parameters match and that the values of the actual parameters fall within the ranges allowed by

the type definitions. The major disadvantage is the large number of parts that have to be declared at the architectural level.

- **Generic:** This method uses Ada generic units to provide parts which are tailorable to user-defined types. The major disadvantage is that generally the user would need to supply a large number of generic parameters. This burden can be alleviated by the judicious use of default parameters (i.e., the designer provides for typical combinations of data types, and the instantiation will use these functions as actual subprogram parameters). This method provides flexibility while simplifying use. Parts can be set up for "tunneling" of operators (i.e., types and operators are predefined, and the types can then be used to instantiate the generics and the operators will get pulled along). The major advantage of this approach is that it incorporates strong typing and is flexible.
- **Abstract State Machine:** This is a "black box" approach. Problems with data typing and mathematical operators are alleviated because these are all fixed by the designer. The user is provided with a high-level interface to the parts. There is no direct access to the data structures themselves; all access is through the operators provided in the interface. Efficiency problems can arise because of the need to perform data type conversions so that data will fit the internal representation. This could be overcome by creating multiple bodies that are efficient for a given situation, but would increase the cost of parts development. It can be effective when the data structure cannot be established in the package specification (e.g., Kalman filter operations).
- **Abstract Data Type:** The data structure is declared in the private section in a package specification. The user is still "stuck" with the data structure that the designer provides. If he wants to change it, he must change both the specification where the data structure is defined and the body. This method differs from the Abstract State Machine approach in that the interface consists of both the predefined set of operators and the data structure itself.
- **Skeletal Code:** This approach provides the part user great flexibility — the user is provided with a template that he must fill in. Problems may arise when more than one person is working a project — the template may be filled in differently by different engineers resulting in much duplication of effort in producing an environment for such a part. There might also be a tendency to avoid strong data typing because of the overhead in creating functions and operators for strongly typed data. This approach could probably benefit the most from automated tools.
- **Semi-Abstract Data Type:** The semi-abstract data type method is based on the generic and overloaded methods. Efficiency is of the utmost importance. The method utilizes derived types and subprograms, generic instantiation, and subprogram overloading. Multiple layers of generics provide the user with a broad selection of parts for an application. Predefined types and operators can be used to instantiate generic parts. This method is also characterized by what is referred to as a *part bundle*. A *bundle* is a collection of parts dealing with a particular type or class of operation, together with their environment. A key feature of the semi-abstract data type approach is the open architecture — the user can substitute his own parts for predefined parts anywhere in the part hierarchy. The semi-abstract data type is under the user's control.

References

1. Ada Board. Ada Board's Recommended Ada 9X Strategy. Office of the Under Secretary of Defense for Acquisition, Washington, D.C., 20301, 1988.
2. "Murtha Fires Brickbats at Software". *Advanced Military Computing* (Oct 1989).
3. Biggerstaff, Ted. (1989). *Reuse in Practice* Workshop, Pittsburgh, PA.
4. Boeing Aerospace Co. Software Interoperability and Reusability. Tech. Rept. RADC-TR-83-174, Rome Air Development Center, July, 1983. Volume I, p. 105.
5. Booch, G. *Software Engineering with Ada*. Benjamin Cummings, Menlo Park, CA, 1983.
6. Booch, G. *Software Components with Ada: Structures, Tools, and Subsystems*. Benjamin Cummings, Menlo Park, CA, 1987.
7. Bott, M.F., and P.J.L. Wallis. Ada and Software Reuse. Proceedings of the Ada Europe Software Reuse Seminar, June, 1988.
8. Bott, M.F., R.J. Gautier, A. Elliott. Guidelines for the Use of Ada in Reusable Software Components. Making Reuse Happen: Component Engineering, Ada Europe Software Reuse Seminar, June, 1988.
9. Braun, Chris and Mamie Lui. Software Reuse in Networking Applications. Proceedings of the Sixth Washington Ada Symposium, ACM SIGAda, June, 1989, pp. 185-189.
10. Burton, Bruce A., Rhonda Wienk Aragon, Stephen A. Bailey, Kenneth D. Koehler, and Laren A. Mayes. "The Reusable Software Library". *IEEE Software* Volume 4, Number 4 (July 1987), 25-33.
11. Campbell, Jr., Grady H. Abstraction-Based Reuse Repositories. Proceedings of Aerospace Software Engineering Conference, AIAA, October, 1989, pp. 368-373.
12. Cohen, Sanford G., Timothy T. Taylor. Common Ada Missile Packages - Phase 2 (CAMP-2), Volume III: CAMP Armonics Benchmarks. Tech. Rept. AFATL-TR-88-62, Air Force Armament Laboratory, Air Force Systems Command, United States Air Force, Eglin, Air Force Base, Florida, 32542, November, 1988. (Distribution limited to DoD and DoD contractors only.).
13. Cohen, Sholom. Locating Resources for Reuse-Based Development. Reuse in Practice Workshop Position Papers, ACM SIGAda, July, 1989.
14. DeLauer, R.D. Interim DoD Policy on Computer Programming Languages. Letter issued by the Undersecretary of Defense, Research and Engineering.
15. DoD. *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD-1815A-1983 edition, United States Department of Defense, 1983.
16. DoD. Defense System Software Development. DoD Standard DOD-STD-2167A, Department of Defense, Feb, 1988.
17. Elliott, A., R.J. Gautier, P. H. Welch. Component Engineering in Ada (Some Problems and Some Advice). Making Reuse Happen: Component Engineering, Ada Europe Software Reuse Seminar, June, 1988.
18. Frakes, W.B. Representation Methods for Software Reuse. Tri-Ada '89 Proceedings, ACM SIGAda, Oct, 1989.
19. Freeman, Peter. "Conceptual Analysis of the Draco Approach to Constructing Software Systems". *IEEE Tutorial: Software Reusability* (1987).
20. Gaisney, John E., Jr. An Economics Foundation for Software Reuse. AIAA Computers in Aerospace VII, AIAA, Oct, 1989, pp. 351-360.

21. Gruman, Galen. "Early Reuse Practice Lives Up to Its Promise". *IEEE Software* Volume 5, Number 6 (November 1988).
22. Holibaugh, Robert, Sholom Cohen, Kyo Kang, Spencer Peterson. Reuse: Where to Begin and Why. Tri-Ada '89 Proceedings, ACM SIGAda, Oct, 1989.
23. Kitaoka, B. Establishing Ada Repositories for Reuse. Tri-Ada '89 Proceedings, ACM SIGAda, Oct, 1989, pp. 315-323.
24. Lanergan, R.G. and C.A. Grasso. "Software Engineering with Reusable Designs and Code". *IEEE Transactions on Software Engineering* Volume SE-10, Number 5 (November 1984).
25. Leavitt, R. Some Practical Experience in the Organization of a Library of Reusable Ada Units. Proceedings, Third Annual National Conference on Ada Technology, 1985, pp. 70.
26. Lubars, Mitchell F. "Code Reusability in the Large versus Code Reusability in the Small". *ACM SIGSoft Software Engineering Notes* Volume 11, Number 1 (Jan 1986).
27. McNicholl, D.G., C. Palmer, et al. Common Ada Missile Packages (CAMP), Volume I: Overview and Commonality Study Results. Tech. Rept. AFATL-TR-85-93, Air Force Armament Laboratory, Air Force Systems Command, United States Air Force, Eglin, Air Force Base, Florida, 32542, May, 1986. (Must be acquired from DTIC using access number B102654. Distribution limited to DoD and DoD contractors only.).
28. McNicholl, D.G., C. Palmer, et al. Common Ada Missile Packages (CAMP), Volume III: Part Rationales. Tech. Rept. AFATL-TR-85-93, Air Force Armament Laboratory, Air Force Systems Command, United States Air Force, Eglin, Air Force Base, Florida, 32542, May, 1986. (Must be acquired from DTIC using access number B102656. Distribution limited to DoD and DoD contractors only.).
29. McNicholl, D.G., C. Palmer, et al. Common Ada Missile Packages (CAMP), Volume II: Software Parts Composition Study Results. Tech. Rept. AFATL-TR-85-93, Air Force Armament Laboratory, Air Force Systems Command, United States Air Force, Eglin, Air Force Base, Florida, 32542, May, 1986. (Must be acquired from DTIC using access number B102655. Distribution limited to DoD and DoD contractors only.).
30. McNicholl, D.G., S. Cohen, C. Palmer, et al. Common Ada Missile Packages - Phase 2 (CAMP-2), Volume I: CAMP Parts and Parts Composition System. Tech. Rept. AFATL-TR-88-62, Air Force Armament Laboratory, Air Force Systems Command, United States Air Force, Eglin, Air Force Base, Florida, 32542, November, 1988. (Distribution limited to DoD and DoD contractors only.).
31. McNicholl, D.G., C. Palmer, J. Mason, et al. Common Ada Missile Packages - Phase 2 (CAMP-2), Volume II: 11th Missile Demonstration. Tech. Rept. AFATL-TR-88-62, Air Force Armament Laboratory, Air Force Systems Command, United States Air Force, Eglin, Air Force Base, Florida, 32542, November, 1988. (Distribution limited to DoD and DoD contractors only.).
32. Neighbors, J.M. Software Construction Using Components. Tech. Rept. 160, Department of Information and Computer Science, University of California, Irvine, 1980.
33. Palmer, Constance. A CAMP Update. AIAA Computers in Aerospace VII, AIAA, Oct, 1989.
34. Prieto-Diaz, Rubin. Domain Analysis for Reusability. Software Reuse: Emerging Technology, IEEE, Oct., 1987, pp. 347-353.
35. Prieto-Diaz, Rubin, and Peter Freeman. "Classifying Software for Reusability". *IEEE Software* (January 1987), 6-16.
36. Sammet, Jean E. "Why Ada is Not Just Another Programming Language". *Communications of the ACM* Volume 29, Number 8 (August 1986).
37. SofTech. Ada Reusability Guidelines. Tech. Rept. 3285-2-208/2, U.S. Air Force Systems Command, U.S. Air Force Systems Command, Electronic Systems Division, TCS, Hanscom AFB, MA, 01731, 1984.

38. St. Dennis, R. A Guidebook for Writing Reusable Source Code in Ada, Version 1.1. Tech. Rept. CSC-86-3:8213, Honeywell Computer Sciences Center, Golden Valley, MN, May, 1986.
39. Software Reuse with Ada, Stockholm, Sweden (Conference), October, 1988.
40. Private Conversation. (Oct 1988). Based on conversation with attendee at *Software Reuse with Ada* workshop in Stockholm.
41. Tracz, Will. "Software Reuse Myths". *ACM SIGSoft Software Engineering Notes* Volume 13, Number 1 (Jan 1988).
42. Vogelsong, T. Reusable Ada Packages for Information Systems Development (RAPID): An Operational Center of Excellence for Software Reuse. Tri-Ada '89 Proceedings, ACM SIGAda, Oct, 1989, pp. 324-330.
43. Woodfield, Scott, N., David Embley, Del T. Scott. "Can Programmers Reuse Software?". *IEEE Software* (1987), 52-59.