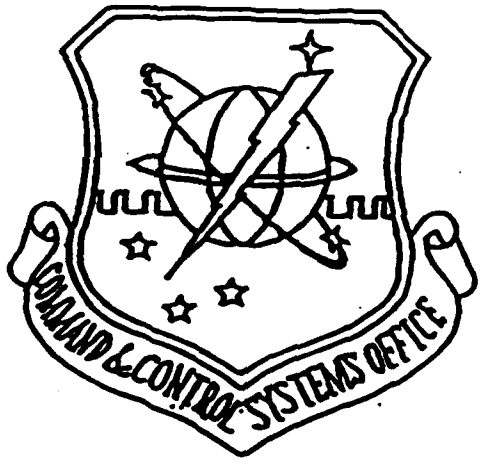
ADA EVALUATION PROJECT

REUSE OF ADA SOFTWARE MODULES

Prepared for

HEADQUARTERS UNITED STATES AIR FORCE
Assistant Chief of Staff of Systems for Command, Control,
Communications, and Computers
Technology & Security Division



Prepared by
Standard Automated Remote to AUTODIN Host (SARAH) Branch
COMMAND AND CONTROL SYSTEMS OFFICE (CCSO)
Tinker Air Force Base
Oklahoma City, OK 73145-6340
Commercial (405) 734-2457
AUTOVON 884-2457/5152

20 JUNE 1988

THIS REPORT IS THE TENTH OF A SERIES WHICH DOCUMENT THE LESSONS LEARNED IN THE USE OF ADA IN A COMMUNICATIONS ENVIRONMENT.

ABSTRACT

Software module reuse using Ada is discussed in the context of the Standard Automated Remote to AUTODIN Host (SARAH) Software Development Project. Reuse concepts are presented including designing, finding, and implementing reusable Ada components. Reuse in the SARAH project is discussed in terms of reuse goals, accomplishments, and problems encountered achieving reuse goals (including design comprises that had to be made). Current impediments to software reuse are presented along with suggestions about how they might be improved. Recommendations are given with reference to the current scope of potential reuse benefits.

Ada Evaluation Report Series by CCSO

Ada Training	March 13, 1986
Design Issues	May 21, 1986
Security	May 23, 1986
Micro Compilers	December 9, 1986
Ada Environments	December 9, 1986
Transportability	March 19, 1987
Runtime Execution	March 27, 1987
Modifiability	April 20, 1987
Testing	June 10, 1988
Module Reuse	July 5, 1988
Project Management	Summer 1988
Summary	Fall 1988

TABLE OF CONTENTS

1. INTRODUCTION

1.1. THE ADA EVALUATION TASK

This paper is one in a series which seeks to help potential Ada developers gain practical insight into what is required to successfully develop Ada software. With this goal in mind, Air Staff tasked the Command and Control Systems Office (CCSO) to evaluate the Ada language while developing real-time communications software. The task involves writing papers on various aspects of Ada development such as training, Ada design, environments, and security issues. This paper discusses Ada module reuse issues.

CCSO chose the Standard Automated Remote to AUTODIN (Automatic Digital Network) Host (SARAH) project as the vehicle basis for the Ada evaluation. SARAH is a small to medium size project (approx. 50,000 lines of executable source code) which functions as a standard intelligent terminal for AUTODIN users and is used to help eliminate punched cards and paper tape as a transmit/receive medium. The development environment for SARAH consists of a number of IBM PC ATs and Zenith Z-248 microcomputers. Source code is developed, compiled, and integrated on these machines. The compiler and symbolic debugger used are from Alsys, Inc. The SARAH software runs on IBM PC ATs, Zenith Z-248s, and Zenith Z-200s (TEMPEST microcomputers which are somewhat compatible with the PC ATs and Z-248s).

1.2. PURPOSE

The purpose of this paper is to provide a forum for CCSO to share knowledge about Ada module reuse gained from experience and exposure to the topic.

1.3. BACKGROUND -- WHY REUSE

Many subfunctions in new software systems are similar, if not identical, to those in previously developed systems. If software were properly designed, those subfunctions could be reused to produce new systems faster, more reliably, and at lower cost[1].

Writing the same and similar code over and over is time consuming and costly. For example, military command and control systems could potentially save time and money by reusing the graphics and display software that has been rewritten for many of these systems[2]. In addition to improved productivity and reduced cost, software reuse has potential to:

a. Increase reliability. Reused code and its accompanying design will have been extensively exercised and "tested under fire."

b. Facilitate rapid prototyping.[3]

Before the advent of Ada, reuse was generally not formally practiced. Some of the inhibiting factors were:

a. Language standardization was inadequate.

b. Languages were technically limited.

c. Reuse incentives were not present.

d. Indexing and retrieval of existing modules was difficult.[1]

Ada was designed with reuse in mind and addresses the first two factors. The language has features intended to facilitate reuse. Language standardization is very important. Subsets and supersets, characteristic of other languages, are not allowed (enforced by compiler validation). Other important features are packages and generics. Packages provide a specification portion that allows interfaces to be precisely defined. Generics provide a tool for customizing existing code to more easily fit into new applications, without changes to the generic module itself.

Software modules are infiltrated with application-specific characteristics. Two different systems may both use a binary table search algorithm, but inevitably something will be different about them (the type of records stored, the size and type of the table, dependence on hardware, compiler, or data representation, etc). This is a major part of the problem that we are trying to solve. By nature software deals with specific problems. The objective is to find ways to use common building blocks (reusable software modules) to build solutions to these specific problems.

1.4. SCOPE AND CONSTRAINTS

The primary mission of CCSO personnel who contributed to this paper is to develop software, not to conduct the research necessary to give a complete treatment to this topic. Treatment is limited to areas of concern to CCSO personnel. This excludes most treatment of legal issues such as copyright and liability concerns associated with software module reuse.

This paper addresses the reuse topic from two standpoints:

a. Reuse of existing software modules.

b. Development of reusable modules.

Code generation and 4th generation languages are considered by some to be types of reuse but are not addressed in this paper.

1.5. FORMAT

The first major section of the paper after this introduction, the CONCEPTS section, overviews some terms and concepts important to reuse. Reading this section may make the rest of the paper more meaningful. It may be skipped, however, without missing any data or descriptions from the SARAH project. Sections 3 and 4 talk explicitly about SARAHs reuse experiences. Section 5 also includes some discussions and examples taken from SARAH experiences.

2. CONCEPTS

2.1. FUNDAMENTALS

2.1.1. WHAT TYPES OF MODULES CAN BE REUSED

Most modules that have been properly designed are reusable to some extent. Modules with hardware or operating system dependences are more difficult to reuse unless on the same hardware or operating system.

In the interest of developing common terminology, we coin the following metric:

$$\text{Reusability} = \frac{\text{cost to create a new module}}{\text{cost to reuse an existing module}}$$

The cost to create a new module is the traditional expenditure to create a software module. The cost to reuse an existing module includes obtaining the module, making changes to the module, and additions or adjustments to the system to make it work. The best case would be when cost to reuse approaches zero, Reusability would then be very high. As cost to reuse approaches the cost to create, reusability approaches 1. When this occurs there is no apparent economic advantage to reuse. Other benefits that have not been factored into the equation may also need to be considered (i.e. reliability, training for reuse, etc.).

2.1.2. COUPLING AND COHESION

Coupling is a measure of the degree of interconnections between modules[3]. This is a very important characteristic in determining how reusable a module will be. When considering a module for reuse, we are concerned with the dependencies of the module (upward coupling). In Ada, this can be ascertained by looking at a module's "with" statements. These tell the analyst what other

modules must be available for the module of interest to be compiled.

Intuitively, few dependencies (low coupling) increases the potential for reuse. The lowest degree of coupling possible is none; the only dependencies would be the Ada predefined language environment itself. In this case, reuse of the module would require nothing more than compiling and using it.

Cohesion is a corollary to coupling. High cohesion indicates that a module's internal elements are tightly bound and related to one another; they are functionally and logically dependent[3]. This is an important characteristic of object oriented systems. Components designed in this way are likely to exhibit low-coupling and be more reusable.

2.1.3. MODULE SIZE vs REUSABILITY

The size of reusable software modules can range from code segments to major subsystems. A hierarchy of module sizes could be defined as follows :

a. Subsystem
b. Component
c. Module
d. Tool
e. Code segment

Tool packages are a group (usually grouped in an Ada package) of mostly small routines that perform a certain class of functions. A good example of a tool package is a string manipulation package. Tool packages are excellent candidates for reuse since the routines are usually small, general, and not dependent on the context or the Ada runtime system (important attributes for reusability).

There are a number of benefits to using tool packages. They save programmer time and reduce memory requirements. The goal is to use the same tool many times; therefore the function it performs does not have to be duplicated. Tool usage also adds consistency and reliability to the system.

System wide tool usage can have the disadvantage of increasing coupling of the subsystems. Any time a tool package is used by a subsystem, the tool package becomes a dependency of the subsystem.

Some have commented that module size (proportional to payoff) tends to be inversely proportional to reuse potential[4]. Among the reasons for this are that larger, subsystem size, modules tend to have more dependencies (coupling). These dependencies must be addressed if the component is to be reused. The component can be changed to remove the dependencies, the depended

upon modules can be reused along with the module of interest, or the module of interest can be rewritten.

Dependencies need not prevent the reuse of subsystems. Since the payoff to reusing these large modules can be high, the cost of fitting the module may be a small price to pay.

As before, the reusability equation can be consulted. If the cost of fitting the subsystem approaches the value of the benefit, then the reusability of the subsystem may be questionable.

2.2. DESIGNING REUSABLE MODULES

Some important characteristic of reusable modules follow:

a. Object oriented groupings,

b. Low-coupling with the other system modules, high cohesion.

c. Implementation dependent characteristics are minimized, identified, and isolated (this includes dependences or characteristics of the Ada runtime environment (data representation, execution speed, etc), and characteristics of operating systems and hardware).

d. Be general.

e. Be fully specified.

2.2.1. SYSTEM MODULARIZATION

The system design should be decomposed into subsystems that are grouped by the class of operations they perform and the type of data they manipulate. An example might be to group all the operations and types dealing with reading and writing to disks, or the grouping of all the operations on a certain type of message correspondence. This is the foundation of Object Oriented Design (OOD)[3, 5, 6]. This can be contrasted to the more traditional functional top-down design that bases decomposition on steps in a data flow.

Object oriented grouping results in modules which are less coupled, more cohesive, and thus easier to maintain and more practical to reuse. Modules will be more general (thus more reusable) since OOD tends to put off data flow details. (Many systems will require a disk management subsystem but few will have the same data flow requirements as the original system.) Older design methods tend to create more application-specific modules.

2.2.2. ISOLATION OF IMPLEMENTATION DEPENDENCIES

Since many subsystems will have implementation dependencies, to make these subsystems reusable, these implementation dependencies should be isolated and identified in a separate package. In this way if the subsystem is reused (or the entire system is ported) with a new compiler and/or operating system, rewriting the implementation-dependent package should render the subsystem reusable.

An example of this may be a disk management subsystem. Some parts of this subsystem will need to be familiar with filename details, do operating system calls to access the disks, etc. These parts of the subsystem will have to be rewritten if the entire disk subsystem is reused. They should be identified and isolated in a separate package to make the modifications easier to identify and accomplish.

2.2.3. SPECIFICATION OF MODULES

Reuse will never really catch on unless modules are **fully specified**, in terms of not only their functionality but also their performance.

Areas to be addressed in specifying a reusable module include:

a. All inputs.
   - all allowable ranges for inputs (i.e. constraints).
   - all input modes (IN, OUT, IN OUT).

b. All outputs.
   - all allowable ranges for outputs.
   - outputs in terms of inputs ('transfer function').
   - exception conditions, and when and why they are raised, and what, if any, internal action is taken.

c. Functionality.
   - interactions between inputs and outputs (whereby the internal action of the process is described in detail - in other fields, this is known as the transfer function).

d. Performance.
   - RAM executable code requirements (static or quiescent RAM requirements).
   - RAM heap requirements (dynamic RAM requirements).
   - Tasking.
     - the number of tasks running internally (minimum, typical andmaximum), and atwhat priority.
     - the minimum, typical, and maximum time units for the execution of each procedure (be it a sequential or tasking-related).

The above are obviously very dependent upon implementation (compiler, compiler options, and hardware target), and therefore

they should be detailed in terms of the implementation. For example "values were obtained using an ALSYS compiler, bound with UNCALLED => REMOVE, optimized for execution time; run-time performance was evaluated on an IBM PC AT running at 8MHz."

At some stage in the future (hopefully), some enterprising agency will publish normalization tables so that the performance on one machine/compiler can be translated into a corresponding performance on another machine/compiler, thus establishing predictable performance, an area that needs more attention in the field of software.

Variable or marginal requirements should be expressed in terms of unit entities. For example, if a reusable editor was being specified, it may have heap requirements stated as follows:

```
   86 bytes / line;
    5 bytes / page (plus 66 lines);
4000 bytes / open file (Text Window);
 512 bytes / open file (File buffer);
```

This would allow a potential user to calculate that for a worst case situation of five files open, each file having 10 pages (3,300 lines), he would require: 86*3,300 + 5(5*10 + 4000 + 512) = 307 Kbytes of heap. This would allow him to make a decision as to suitability of the module for his specific requirement.

Similarly, all run-time requirements should be specified in terms of minimum, typical, and maximum values. The potential user could then perform calculations to assess the most suitable of a range of available modules. Consider a package designed for communications functions. Unless the user can calculate that it will execute (on his particular hardware) at the required rate, he will not be able to use it. It's no use using an available communications package if it can't keep up with the data transmission speed. And it's no use trying to use it unless you can have at least a calculated chance of it working. The essence is the ability to calculate, this is determined by the quality of the performance specifications of the module.

2.3. FINDING REUSABLE MODULES

2.3.1. MODULE LIBRARIES/REPOSITORIES

Module repositories are one source of Ada modules. The most well known repository among SARAH personnel is the Ada Software Repository (also know as SIMTEL20). As of September, 1987 this repository contained 1,515 files or about 805,000 lines of source code[3]. SIMTEL20 information is available on DDN. The DDN address for subscribing to Ada Software Repository is <ADA-SW-REQUEST@SIMTEL20.ARPA>.

2.3.2. COMMERCIAL PACKAGES

We know of two sources of commercially available Ada software modules. The GRACE package from EVB software Engineering[6], and the WIZARD package from Grady Booch[9]. Both are large boxes of tools and data structures. Each comes in variations to fit many different application environments.

2.3.3. OTHER DoD ORGANIZATIONS

Since Ada is being used across the DoD, it's possible that another DoD organization has already written some modules that you can use. Exchanging source code between DoD organizations should be encouraged. Caution should be exercised, however, because most existing code is too application specific to be reusable. In general, modules are reusable only if they were properly designed to be reusable.

2.4. IMPLEMENTING REUSABLE MODULES

2.4.1. IDENTIFY CANDIDATES FOR REUSE EARLY

It's important to identify the modules you plan to reuse as early in the project as possible. This is especially true for larger (subsystem size) modules. (See the discussion of the FLIPS project (paragraph 4.2) which utilized the SARAH VDT subsystem.)

2.4.2. MODIFY MODULES TO FIT (IF NECESSARY)

Often, modules to be reused must be modified to fit the application. Or, for larger modules, depended upon units may have to be reused also or rewritten.

The typical method of reusing a subsystem should probably be to make the Ada package specifications of the depended upon packages part of the subsystem. Commonly the depended upon packages provide low-level services such as input/output. If the subsystem is ported to a new operating system environment, the package bodies will need to be rewritten. If not, the depended upon packages may be reused (with changes if necessary).

Ideally, incorporation of an existing module can occur without being concerned about how a module works (what's in the package body). Interfacing to Ada modules occurs via the specification part of the module (the package specification). If the module is properly designed, reuse may be accomplished without changing or being concerned about the internals of the module.

2.5. REUSE OF CONCEPTS AND IDEAS

Reuse of ideas (coding and design ideas) can be just as important a reuse concept as reuse of code. In many cases ideas can be reused when the code itself is too application-specific to be reused. This is a fundamental concept to computer science. It's

the next best thing to actual reuse. Much time can be saved reusing an idea that worked, rather than develop a method from scratch.

3. REUSE GOALS OF THE SARAH PROJECT

3.1. USE OF EXISTING REUSABLE MODULES

This section discusses our goals for reusing existing software specifically for the SARAH project.

3.1.1. LARGE-SCALE REUSE GOALS (TOP-LEVEL)

SARAH's top-level design was developed using an object-oriented methodology. The exact approach used is described in an earlier paper[1].

The design consists of several major subsystems. These are Editor, Print, Disk, VDT (including a screen manager, a sound manager, and key manager), Message Masks, and the Communication Subsystems (transmit processing, receive processing, line protocol, etc). When these subsystems were defined, we hoped we would be able to find an existing subsystem to satisfy the requirement for one or more of them.

After some preliminary searching, we concluded that the only subsystem we had any reasonable chance of finding was an editor. So, our large-scale goal for use of existing software became to find and incorporate an editor into the system.

3.1.2. SMALL-SCALE REUSE GOALS (LOW-LEVEL)

For our low-level designs, we wanted to utilize existing tool packages where we could.

Our goal at this time became to limit our use of tool packages to those that would most exploit the benefits. The most important benefit to us was the potential for reducing the memory requirement. This was a significant constraint from the start. Our Personal Computer (PC) target has limited memory and Ada has a tendency to generate large executable files[11]. We were also interested in the time savings since our project was working under a short suspense.

We also planned to write some of our own tool packages, mainly for the purpose of enforcing certain system data standards and for consistent operations on these data objects. We expected to reap savings in storage and development time from these also.

We started searching for a string manipulation package and began designing our own tool packages.

3.2. DESIGNING SARAH MODULES TO BE REUSABLE

This section addresses our goals for making SARAH software reusable. Among these design goals were that the system be portable, and that as many modules as possible be reusable. These goals are different, yet they have similarities, especially as pertains to reusing larger (subsystem size) modules. Of course these design goals were secondary to the primary design goal of satisfying the system requirements with an understandable and maintainable system.

Using an object oriented method[10], we modularized the system into independent subsystems, each one with a different class of responsibility. These areas were Disk Manager, Print Manager, VDT Manager (Visual Display Terminal) Editor, Validation, and Communications. A Mask subsystem evolved and became separate from the Editor (Mask is used to create messages in valid Defense Communications Agency formats). Each subsystem was designed to be solely and completely responsible for its area of responsibility. No overlap was allowed.

Following OOD principles, each subsystem was designed to be as cohesive as possible. For example, VDT Manager was designed to have all the types and functions needed to do any type of screen, sound, or keyboard operation, within SARAH. Also, no other subsystem is allowed to access these resources for which VDT Manager is responsible. In this way, we hoped, VDT Manager would be reusable as a complete cohesive entity; available for a new system to easily develop its own user-friendly interface.

Each subsystem was designed to isolate and identify implementation dependent areas. An example of this is the Disk Manager subsystem. SARAH runs on PCs with the MS-DOS (or PC DOS) operating system. This means that Disk Manager must be aware of some of the characteristics of this operating system (filename formats, directory formats, legal drive characters, etc.) During the design of Disk Manager an attempt was made to isolate all of these implementation-dependencies in a package called Disk Definitions. The reason was that if SARAH was ported to a new operating system, say UNIX (or the Disk Manager subsystem was reused in a UNIX environment), the only major change needed to Disk Manager would be a rewrite of Disk Definitions.

4. REUSE ACCOMPLISHMENTS OF THE SARAH PROJECT

4.1. USE OF PREVIOUSLY EXISTING MODULES

The SARAH project used few existing reusable modules. The string-manipulation package obtained from the Ada Software Repository was the only existing module (along with a few code

segments) that we used. This package was about 250 executable source lines. We had to make a minor modification, but it proved to be reliable.

We searched the Ada Software Repository and called around looking for an editor but couldn't find one. We eventually quit looking and began writing our own.

We did not use any commercially available modules.

The Mobile Information Management System (MIMS) project at Offutt Air Force Base sent us a copy of their source code. We did not, however, use any of this code in SARAH.

4.2. REUSE OF SARAH DEVELOPED MODULES

Shortly before the completion of the first release of SARAH, a new Ada project was started elsewhere in CCSO. This project is called the Flight Information Processing System (FLIPS). FLIPS is being developed with the same compiler (Alsys Ada for PCs) and for the same target (Zenith Z-248). This project was planned from the start to reuse as many of the SARAH modules as possible to speed development. In fact, one of the FLIPS analysts was sent over to work with the SARAH project for four months to gain experience with the SARAH subsystems.

So far, the FLIPS project has incorporated nine SARAH packages into their system. This is roughly 2,500 lines of code making up about 60% of their current prototype version. These packages consist of the SARAH VDT Manager subsystem and the Buffer Manager package (SARAHs central manager for memory allocation and deallocation).

About three weeks were spent incorporating these modules into the FLIPS design (using two people). FLIPS personnel estimate that it would have taken at least six to eight months to create this code from scratch. Although none of these modules were adapted without change, FLIPS developers were generally happy with the reusability of these subsystems.

They also commented that reuse of these modules was possible primarily through the personal experience of their analyst who worked with SARAH. "Without this familiarity," they commented, "much more time probably would have been spent identifying and modifying these modules. A cataloging system and a generally standard method of documenting modules would be a great help to future systems development."

One catch that the FLIPs personnel had to accept when using VDT Manager was using Buffer Manager as well. This is because, like most of the other SARAH subsystem, VDT Manager is dependent

upon Buffer Manager. This problem is more fully discussed in paragraph 5.2.1.

A number of other interested DoD agencies have received copies of the SARAH source code. The subsystem generating the most interest is the VDT Manager. So far, we haven't received any feedback on the reuse of these modules.

5. WHY REUSE GOALS CAN BE DIFFICULT TO ACCOMPLISH

5.1. DESIGNING AND WRITING REUSABLE CODE TAKES LONGER

Reusable code does not happen by accident. It takes more time to write reusable code than it takes to write code just to satisfy the requirements of the given application. Some think it takes many times longer.

This is one of the major impediments to the development of reusable code. Projects are usually run on tight schedules. Satisfying project requirements and meeting project milestones are usually given more importance than designing the system modules to be reusable for the next project.

Making design of reusable code a high project priority is the only way around this problem. Take the time to train personnel in how to design and write reusable code; then give them the time to do so. Recoup of this investment will occur two or three projects down the road when the organization has established a useful pool of good quality, reusable modules. As this pool grows, and reuse becomes a way of doing business, benefits can multiply.

5.2. REUSE GOALS CAN CONFLICT WITH OTHER DESIGN GOALS

5.2.1. CONFLICT WITH SIZING GOALS

Reusable modules are often larger than their applicationspecific counter parts. Combine this with the fact that Ada tends to generate larger executable files than most languages[11], and you run into potential conflicts with sizing goals.

Some projects may not have this problem. In this day of cheap memory and expensive labor, the development and maintenance time saved using reusable modules can easily outweigh the cost of memory.

But a project with strict sizing requirements, like many embedded systems, will have to look hard at this potential conflict. The savings available from using reusable modules may have to be compromised to satisfy sizing requirements.

One type of reuse that has the potential to save memory is the use of tool boxes (tool packages). An efficient, reliable tool box can prevent the need for redundant routines in the system.

5.2.2. CONFLICT WITH EXECUTION SPEED GOALS

Modules designed for reuse may run slower than modules designed for speed. However, this is not always the case; a well designed, well thought out reusable module may run very efficiently. A rule of thumb, however, is that generality is inversely proportional to power[4] (efficiency). This needs to be considered both when using an existing reusable module or designing a reusable module for time critical code segments.

The average system executes a small portion of its code much more frequently than the rest of the code. This very frequently executed code may be the only part of the system in which the potential loss of speed is more significant than the benefits of reuse.

The rigid performance requirements of many real-time embedded mission-critical applications (interrupt control, cyclic execution, cyclic execution, predictable timing, storage control, and flexible scheduling) conflict with the goal of developing reusable software components[12].

5.2.3. CONFLICT WITH STANDARDIZATION GOALS

System designers may want to introduce standard types and standard tools for use across the system. Reasons for these standards can be to reduce code size, reduce labor costs, or to introduce an element of consistency to important system parameters.

Use of standard types and tools increases the coupling of the system and thus reduces the reusability of the individual modules of the system. This was a major design conflict with SARAH and is described in the next section.

5.2.4. SARAH's REUSE DESIGN CONFLICTS

The major design factor which reduces the reusability of SARAH subsystems is a high degree of coupling between them and system tool packages. The reason for this coupling is to satisfy both sizing and standardization goals.

The package to which most SARAH subsystems is coupled is the Buffer Manager. The major reason for the existence of this package is to prevent progressive memory fragmentation from multiplying until the system fails. This danger exists because neither the MS DOS operating system or the Alsys Ada run time environment provides any type of garbage collection function. Since SARAH is designed to handle message traffic 24 hours a day for indefinite periods of time, this problem had to be addressed.

Buffer Manager is the central point for allocation and deallocation of memory. By ensuring that all memory buffers

allocated are of constant size, Buffer Manager effectively removes the fragmentation problem.

Buffer Manager also acts as a tool package for standardizing file formats. All linked-list work required for creating and modifying files is accomplished using its tools.

Elimination of the fragmentation potential and standardizing file formats and file manipulation tools were important design goals. Achievement of these goals have contributed greatly to the success of the system. However, their achievement was accomplished at the cost of reducing the reusability of most of the SARAH subsystems.

5.2.5. BALANCE REUSE GOALS WITH OTHER DESIGN GOALS

System designers are nearly always faced with decisions on how best to resolve conflicting design goals. Reuse is certainly a design goal with potential to conflict with other design goals.

SARAH designers were faced with just these types of decisions. Some reusability was sacrificed to achieve system standardization goals as described above. However, some efficiency goals were sacrificed in a effort to achieve some degree of reusability and transportability.

One example of this is the communications subsystem. The communications protocol could have been written in assembler language to increase speed and reduce memory requirements. However, our goal of achieving reusability for this subsystem prevailed and the protocol was written in Ada using the task model. (It should be noted that this loss of speed and space will probably become less significant when more efficient compilers become available.)

5.3. PROBLEMS OBTAINING REUSABLE MODULES

5.3.1. NO STANDARD TAXONOMY

A method of classifying and categorizing modules is essential if reuse is to be successful on any significant scale (particularly on an inter-organizational basis). Both the Grace and Wizard commercial products have implemented their own taxonomy methods[3, 9]. But no standard taxonomy has been recognized by the software community.

A taxonomy method must allow selection of modules based not only on function (e.g. queue, stack, binary search, etc.) but also on precision, robustness, generality, and/or timespace performance[8]. An example of one taxonomy method now in use is the one used for EVB's Grace package: the attributes used in classifying their modules are bounded/unbounded/limited iterator/ non-iterator, managed/unmanaged, protected/sequential/guarded/ controlled/multiple/multi-guarded, operation concurrent/object concurrent, priority/non-priority, and balking/non-balking. As

you can see, there is a lot more to classifying a module than by its basic function. A standard needs to be developed, accepted, and taught. Such an accomplishment would carry us great strides towards not only more successful reuse programs but also to help us be more precise during the design process.

5.3.2. NO STANDARD DESIGN METHODOLOGY FOR REUSE

The design methodology that best supports reuse is a much debated topic. Most believe that Object Oriented Design (OOD) best supports reuse since it bases the modular decomposition of a software system on the classes of objects the system manipulates rather than on the functions the system performs[3, 5, 6]. Although OOD is widely discussed, it's not widely used (yet). Most still use the de facto standard Structured Analysis/Structured Design[6]. There is much division among software engineers as to the implementation of OOD; some feel that Ada is perfectly suited for use with OOD[3, 12, 13, 14, 15], others with a narrower definition of OOD feel that it's not and other languages must be used to achieve successful reuse[5, 16]. Much work is being done in this area and for good reason; we need a standard design methodology that fosters reuse. The High Order Language Working Group provided a standard language but not a standard design methodology.

In the absence of a standard design methodology for support of reuse and other modern design goals, we formulated our own methodology. For a description see the Ada evaluation paper titled "An Architectural Approach to Developing Ada Software Systems[10]."

5.3.3. NO STANDARD FOR RATING QUALITY/RELIABILITY

The lack of a standard for rating the quality and reliability of reusable modules is a deterrent to reuse. One of the major inhibitors to reuse is fear of the unknown. Few developers will risk their reputation by including in their system modules they suspect of being of dubious quality or reliability. Consumers Union does not test software modules. None, except a handful of commercial vendors, provide any sort of guarantee with a software module. A rating method must be established and used by software libraries to reduce the risk of using existing software modules.

5.3.4. NEED BETTER REUSE LIBRARIES/REPOSITORIES

Part of the Software Technology for Adaptable Reliable Systems (STARS) program is to foster the creation of reuse libraries[17]. The SIMTEL20 library at the White Sands Missile Range is one result of this effort. Other libraries are also being developed at universities and DoD organizations.

The problem with these libraries stems partly from two of the previously identified problems: lack of anccepted or adequate taxonomy method and lack of a way of rating modules on the basis of quality or reliability. We can state from experience that

these two deficiencies alone render the SIMTEL20 library of far less value than its potential.

An ideal library would contain many modules; utilize a standard taxonomy method; utilize a method for rating the quality and reliability of the modules; and would have a browse feature to allow users to examine abstracts, OOD documentation, and source code for any modules of interest[14].

6. FINAL WORDS WITH RECOMMENDATIONS

6.1. INTER-ORGANIZATIONAL REUSE MOSTLY FOR SMALL MODULES

At the current state of the discipline, inter-organizational reuse is a viable option for tool boxes containing common algorithm-size modules. String manipulation packages, searches, sorts, stacks, queues, lists, filters, pipes, pattern matching routines, etc. would all fall into this category.

We recommend that these be purchased rather than searched for in a reuse library/repository (although we have no experience with commercial modules, others do[14]). Users that purchase modules will more easily find the exact module they need and will have more confidence in the quality of the module. The Grace and Wizard products[8, 9] are both comprehensive packages of common algorithms. They contain not one but many versions of each algorithm; so you can, utilizing their taxonomy system, select the one that best fits your application.

6.2. INTRA-ORGANIZATIONAL REUSE POSSIBLE ON LARGER SCALE

"The major source of reusable modules is within companies and from software module subindustries[18]." This quote from Grady Booch is a reasonable statement for subsystem size modules; especially when you consider the obstacles to reusing modules of this size.

Legal problems are simplified when modules are reused within organizations (both liability and copyright considerations). Confidence in the quality of a module is increased when you have lunch with the people who designed and wrote it. Problems in understanding and implementing the module can be more easily worked out when you can walk over and talk to the authors (or use other organizational lines of communication). Company coding and design standards are more easily maintained, the target/host hardware is more likely to be the same, maintenance of the module is easier to coordinate, some personnel may even work on both the project that created the subsystem and the project wanting to reuse it (as was the case with the SARAH and FLIPS projects), and the price is right.

To exploit the advantages of intra-organizational reuse, many organizations are creating their own libraries of reusable Ada modules[14]. CCSO is on the road to just such an accomplishment. An organizational reuse library can contain modules created within the organization, purchased modules, and modules obtained from repositories (once they have been used and tested).

6.3. NEW STANDARDS REQUIRED TO MAKE REUSE COMMON

The software development discipline must mature to the point of being able to effectively reuse software modules; few would argue with this. What's needed for reuse are some of the same things needed to push our discipline towards maturity: Standards.

The DoD, with its STARS program, is leading the way towards establishing needed standards. A standard programming language was a significant step. Fostering the development of code libraries is also important though currently a much less mature effort. Support of education and research through the Software Engineering Institute is another important STARS effort [17].

But we have not gone far enough. We need a standard taxonomy for categorizing modules, standards for specifying modules, and standards for rating the quality and reliability of modules. Some semblance of a standard design methodology which supports both designing reusable modules and reusing existing modules would be a welcomed contribution.

The mechanism for establishing these standards is not clear. It took an act of government to establish Ada as a standard programming language. But MS DOS became a de facto standard micro computer operating system without government action.

Proper standards can help propel the status of software reuse from an art to a science, an important step for Software Engineering.

A. REFERENCES

1. Software Engineering Institute, "Ada Adoption Handbook, version 1," May 1987, pp 53-55.

2. Harold C. Brooks. Personal Communication. January 1987.

3. Grady Booch, "Software Engineering with Ada," Textbook; Benjamin/Cummings Publishing Company, Inc., 1983.

4. Ted Biggerstaff and Charles Richter; "Reusability Framework, Assessment, and Directions," IEEE Software, March 1987, pp 41-49.

5. Bertrand Meyer, "Reusability: The Case for Object-Oriented Design," IEEE Software, March 1987, pp 50-64.

6. I. Sommerville, "Software Engineering, Second Edition;" Textbook; Addison-Wesley Publishing Company, 1985, Chapter 4.

7. C^2MUG Bulletin, Sep/Oct 1987, pp 5-6.

8. Grace Software, EVB Software Engineering, Inc., 5303 Spectrum Drive, Frederick, MD, 21701.

9. Wizard Software, 835 S. Moore St., Lakewood, CO, 80226

10. Command & Control Systems Office Ada Evaluation Paper #2 "An Architectural Approach to Developing Ada Software Systems," May 21, 1986.

11. Command & Control Systems Office, Ada Evaluation Paper #7, "Runtime Execution Considerations for Ada Software Development," March 27, 1987

12. Anthony Gargaro, "Reusability Issues and Ada," IEEE Software, July 1987, pp 43-51.

13. Ed Berard, "Creating Reusable Ada Software," Presentation given at the Sunbelt SigAda meeting, August 1987, Central State University, Oklahoma.

14. Harold B. Carstensen, Jr., "A Real Example of Reusing Ada Software," Magnavox Electronic Systems Company, 1313 Production Road, Ft. Wayne, Indiana, 46808.

15. Jean E. Sammet, "Why Ada is Not Just Another Programming Language," Communications of the ACM, August, 1986, pp 722-731.

16. Gail E. Kaiser and David Garlan; "Melding Software Systems from Reusable Building Blocks," IEEE Software, July 1987, pp 17-42.

17. Col Joseph Green, Jr., USAF Director of the Software Technology for Adaptable Reliable Systems (STARS) program, "Rational for the New STARTS Program", Presentation given at Computer Resources and Data Configuration Management Workshop, Bellevue, Washington, September 15-19, 1986.

18. Grady Booch, "On the Concepts of Object Oriented Design," presentation given at the Sunbelt SigAda meeting, March 3, 1988, Phillips Research Center, Bathesville, Oklahoma.