

ISSUES ON SOFTWARE REUSE

Capt James E. Cardow
Headquarters Air Force Logistics Command
(HQ AFLC/MMTEC)
Wright-Patterson Air Force Base, Ohio

Abstract

The reuse of software components and documentation in weapon systems offers a significant opportunity for the Air Force to achieve not only savings in resources, but also to improve the reliability of systems that are software intensive. The development of Ada and improvements in software engineering practices have reduced the technology barriers that prevented reuse from becoming standard procedure. Technology barriers such as the maturity of Ada compilers still exist, but they are being addressed. There are other major barriers that also need to be considered. These barriers fall into the categories of acquisition, technical, and managerial issues. Within each of the issue areas are a number of specific topics that need to be researched and resolved.

This paper reviews these issue areas, identifies the specific topics and provides recommendations on the work that needs to be accomplished to resolve the issues.

1 Introduction

Reusability of software is not a new concept. Programmers often refer back to software from previous developments and reuse existing code in new projects. Reuse of this nature, while beneficial, is at too low a level in the overall system development process and is very location and application specific. More importantly, it lacks the degree of formality and control to make a major impact on the total project. With the ever increasing demand for new software and the improvements in software technology, it is becoming increasingly apparent that widespread reuse of well designed, well developed, and well supported software will offer a tremendous opportunity for savings.

2 Background on Software Reuse

Before the issues in reusability can be effectively discussed, an agreement is needed on a definition of reusability and an agreement is needed on which elements to consider for reuse. Prieto-Diaz [5] defines the concept of reusability as "the use of previously acquired concepts and objects in a new situation." In the software development and support environment, this will include the use of any object or concept from a previous development for the creation or expansion of a another system. A more formal explanation of the elements

to be reused also comes from Prieto-Diaz in terms of two levels of reuse:

1. ideas and knowledge
2. particular artifacts and components

In the reuse of ideas and knowledge, software engineering principles and methods are currently transitioned, with developers, from one project to the next. Current efforts, however, lack the formality to capture and transfer specific ideas in terms of design trade-offs and decisions. Reuse of this information could save reevaluation of the same decision path numerous times, as similar components in a common domain are developed. In the reuse of particular artifacts and components, the key issue is the transition of the products from one development to the next. But, the final answer on what can be reused will come only from an examination of the products of a specific domain (missiles, guidance, simulators) and the products of the software development/support process. The list must be more than just software (source code). It must include the documentation, design, and even requirements if the benefits of the ideas and knowledge are to be transferred. In reporting on the DRACO project, Neighbors [4] highlights the fact that "reuse of analysis and design is much more powerful than the reuse of code." Unfortunately, it is also much more difficult.

3 Benefits and Costs of Reuse

Once agreement is reached on the definitions, there needs to be an understanding of both the potential benefits and costs of reusing software. The issues presented later in this paper should quickly confirm that extensive effort is required for successful reuse. Beyond the proof of concept and laboratory demonstration phases, managers will need to clearly see the potential benefits of pursuing reuse. At the same time any competent manager will want to know the costs.

Little hard evidence is available to support the benefits of reusability. There is a very natural, intuitive sense that given the high cost and extreme difficulty of software development, building on previous work can offer a savings. But, can we quantify what seems intuitive? A statistical benefits analysis on savings from reuse does not exist; reuse is not sufficiently widespread to support the necessary data base. However, the Common Ada Missile Parts (CAMP) program

[3] was able to project anticipated savings in productivity. These projections are based on the use of existing modules to develop new systems within a well defined domain, in this case, missiles. Their conclusions should be reviewed when considering reuse as part of a new development effort.

Beyond this intuitive savings in productivity should come the realization of a more reliable system. The reliability will increase if the parts are tested and improved by use in earlier systems. Software, unlike hardware, does not wear out; so if properly controlled, the reliability of components will increase with each reuse. Some experts recommend reusing a component three times before it is considered reliable.

Finally, the project should experience a savings in time (length of phases) due to the use of tested, reliable components. This savings should appear primarily in the development and integration of the system.

To answer the questions on the cost of reuse rather bluntly, the cost will not be trivial. Costs will include developing software to higher standards and will include making trade-offs during the development process. The trade-offs must weigh short-term savings on current developments against potential savings on future efforts. Performance trade-offs may also play a role, and also need to be considered. Much like the hardware counterpart, software components that nearly match requirements might be available for use, but their use implies accepting some performance less than that achievable through optimally designed components. Managers will need to decide if this lesser performance is within an acceptable limit for their system and if the savings from reuse will benefit the overall program. The key point is that reuse is not free. Reuse implies not just the act of copying existing pieces, but using high quality components to meet a specific need. The decision to pursue a reuse plan on a project is a commitment for long-term results, at some higher short-term cost.

4 Technology Barrier

Historically, reuse has not been a widespread practice. This is due to the development of computer technology and the demand for software. As computer technology passed through each successive generation, two things occurred: compatibility improved and demand for software increased. This was best summed up by Edgar Dijkstra's acceptance speech for the 1972 Allan Turing Award. In that speech he said, "As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now that we have gigantic computers, programming has become an equally gigantic problem." During the evolution up to this point, unique solutions were given preference, and the demand for software, while growing, did not exceed the ability to create it. Thus, unique computers and programming languages made reuse more difficult than beneficial. Now, 17 years after Dijkstra's speech, the demand for software far exceeds our capacity to

create it. Technology has, however, evolved to a point where computers are far more compatible, and more importantly, computer programming languages have moved to a higher level of standardization and portability.

5 Ada in Software Reuse

The development of the DOD programming language Ada is one step that will significantly aid the reuse effort. The benefits of Ada for creation of reusable components according to Wegner [7] include:

- a rich variety of program units
- systematic separation between interface specifications and implementation details
- strong data typing
- generic program units
- program libraries

Certainly all of the barriers to full scale reuse of Ada components have not been removed, but they are being addressed. Anderson [1] points to language refinement, compiler maturity, and compiler global optimization as some of the issues requiring additional work. But, the significant technology barriers to reuse have been reduced.

6 Issues Inhibiting Reuse

William Tracz [6], in the keynote speech of the Spring 1988 National Symposium and Workshop on Software Reusability quoted Mary Shaw of the Software Engineering Institute as saying "software reuse will become the expert systems of the 1990's." This refers to the hype and overselling of the potential and benefits of reuse in the same way that expert systems were hyped in the 1980's. This, no doubt, will become a true prediction, if reuse is not *adopted* by the two communities most in need of software reuse: the software acquisition community and the software support community. Both are burdened with the cost and complexity of software. (Note, it is the software acquisition community, not the software development community, that needs to take the lead, because it is the acquisition community that bears the burden of rising costs and increasing delays.) It is for the acquisition and support communities that this paper is intended. These are the communities that must understand the issues and costs of software reuse and the communities that must develop the strategy for benefiting from software reuse. Benefits that will come, not from hype, but from well planned and implemented solutions to resolving the issues. The issues come in three major categories: acquisition, technical, and managerial issues. Acquisition issues deal with both the legal issues surrounding the reuse of software and the DOD standards that apply to software acquisition. The technical

issues deal with the mechanics of having reusable software: the documentation standards, the domain assessments, and the testing of reusable components. The managerial issues deal with the software libraries and the unique aspects of cataloging software for reuse. Within each of the issue areas are a number of specific topics that need to be researched and resolved. The remainder of this paper will address each issue area, first by explaining the issue, and then by providing a recommendation on the necessary actions to address the issue.

7 Acquisition Issues

There are issues that need to be overcome in the reuse of software components that have nothing to do with technology. These issues relate to the acquisition policies, practices, and procedures governing the procurement of software systems. Software is currently covered under copyright laws, and considerable attention is applied to protection of the software developers. Software is also currently considered a high risk area for new developments, an area where quality measures are difficult to apply and assess. For this reason, contractors and government contract managers are reluctant to trust existing software in their projects without knowing the responsibilities and guarantees on the parts of the developers, purchasers, and users. The managers fear that using software developed outside of their control may impact their program. Finally, the acquisition policies and procedures in place today were created with unique development in mind, and do not address the impacts of reuse or provide guidance for including reuse in contracts.

The purpose of this section then is to provide recommendations that will have an impact primarily on contracting for software. The recommendations are intended to address the confusion over the copyright laws, to identify the responsibilities, to examine the effect of warranties, and to review the government directives for impacts on software reuse. The goal is to provide developers with incentives to develop *reusable* components, to provide contractors with incentives to use *reusable* components, and to provide contract managers with incentives to require the use of *reusable* components. Along with these incentives there must be a clear understanding of the protection provided to all parties, as well as an understanding of the responsibilities of each party.

7.1 Data Rights

The concept of reusable software is, for the most part, at odds with the data rights of the developer of software. Data rights protect the developer from the unauthorized reuse of the developer's work; reuse attempts to exploit it. Gabig [2] presents an explanation of the existing categories of data rights in use by the government and attempts to reduce the confusion over ownership, or rights, to the products. The rights are based on the current laws, with the interpretation

determined by how the development was funded. Most of this confusion is managed in government software purchases by the different categories of data rights, when the categories are appropriately applied. The existence of the confusion will only be increased by including the use of existing software in new systems or requiring that new software be developed for reuse in future systems.

To resolve this issue, a set of guidelines is needed that explain the rights of the software developer, the software purchaser, and the software supporter in terms of reusable software. The guidelines need to include all aspects of software development including documentation and other non-code objects. The guidelines need to establish a better understanding of data rights as applied to software and documentation within a potential library system. The guidelines also need to provide incentives for development of reusable components that will stimulate interest in reuse. Currently software is purchased by the government under different categories (limited, unlimited, government purpose, and restricted rights). Reuse must take each of these into account.

7.2 Responsibility Policy

Software reuse creates a real dilemma in terms of responsibility for all parties involved with the software. The issue here is defining the responsibilities surrounding the use of software as Government Furnished Software (GFS). The responsibilities include those on the part of the software developer as a result of use of the components and those of the government when software is purchased that has the potential for reuse. Responsibilities should include all aspects of financial, civil, and criminal actions. Questions must be answered such as whether the component developer can be held responsible for delays caused by use of the component or for damage caused by use of the component. The simple answer is no, but then what program manager will be willing to accept use of the components.

This issue calls for the development of a new policy to cover responsibility for software and documentation within the library system. Specifically, it must address the issues of GFS as it applies to use in development contracts.

7.3 Warranty Policy

Software and software components are generally not considered a warranted item. Unfortunately, the acceptance of unreliable software is so prevalent that disclaimers are part of almost every commercial software package, and the government fully anticipates *latent defects* as part of the delivery of software. This same degree of unreliability would be totally unacceptable in hardware. This fact will create the greatest resistance to reuse from program managers, because it puts the quality of the components outside of their control. Aspects of this issue include examining the mechanism to implement a software warranty, proposing deficiency identification

and certification procedures, determining methods for compensation when failures do occur, and exploring the potential of implementing system warranties rather than software or hardware warranties.

To complete this effort it may be important to first establish a task team of legal, technical, and acquisition experts to consolidate the issues and then to define a new policy on software warranties for reusable components. Also applicable are some of the issues explored later in terms of quality.

7.4 Acquisition Standards

As stated earlier, software reuse has not been a part of the traditional software development and support process. Therefore, it is no surprise that reuse is not a prominent part of the standards that govern software acquisition. While the standards in their current form may not be true barriers to reuse, they do not contain sufficient support for reuse. This barrier could be as simple as not considering reuse when the contracts for software are developed.

Resolving this issue involves examining the existing acquisition standards (DOD-STDs 2167A and 2168) to identify conflicts with reuse. Some recommended improvements could be:

- inclusion of new Data Item Descriptions for analysis of available components before development begins
- identification of potential for future reuse
- inclusion of test results and procedures as deliverables for components that have a high potential for reuse

Another solution could be the development of tailoring guidelines to improve potential for reuse.

8 Technical Issues

Current and past software development techniques have centered on development of software components to solve a specific problem, within the constraints of that project. The result has been a continuous *reinvention of the wheel* with all of the expected duplication of expense and effort. The opinion has been that software reuse was difficult and that the effort to reuse software would be greater than any potential savings. As stated earlier, the technology issues in reusing software are being addressed, especially in terms of Ada. The recent selection of Christine Anderson, the catalyst behind CAMP, as the head of the committee reviewing changes to the Ada standard should ensure those issues are not forgotten. But there are technical issues that are not technology barriers that need to be agreed on before reuse progresses. (Note technology barriers are obstacles in the technology itself, for example, compiler maturation; while technical issues are areas that are solvable but require technical agreement, for example, testing standards.)

The primary purpose of this section is to identify the technical issues that inhibit software reuse and propose solutions that involve the development of the standards and procedures to support reusability. Many of the issues are raised, correctly so, by managers with a reluctance to accept components of uncertain quality for use in their systems. Therefore, many of the solutions are aimed at quality assurance of the components.

8.1 Coding Standards

Before managers are willing to risk project success by using existing components, there will need to be minimal requirements for acceptability of a component. In terms of the coding of a component, the standard needs to include all the requirements expected in a coding standard, plus those that uniquely effect software intended for reuse. At the minimum, this should include items such as the size of modules, the level of nesting, and naming conventions. Plus, the standard needs to define methods of documenting other considerations, such as the internal and external interfaces and the precision of accuracy requirements for the component.

The recommended solution to this issue is development of a software coding standard for components in the library. The purpose being to ensure only well structured code is placed in the library. This could be an addendum to Ada coding standards already completed or in development such as the Ada Coding Standard under development at Gunter AFB, AL.

8.2 Reusability Metrics

Beyond the coding standards placed on a component, managers should be concerned with the reusability of a candidate component. This should include factors that effect the ability of the component to be integrated into a system, factors such as interface complexity, efficiency of execution, use of global variables, implementation dependencies (restrictions on processors, etc.), and the coupling and cohesion of the component's parts.

Resolving this issue requires development of metrics for measuring *design for reuse*. This metric should enable a potential user to gage the portability of the components and provide some quality metrics for comparing similar components. This could potentially be an automated software tool, but minimally should consist of a detailed checklist, formula, and scale.

8.3 Documentation Standard

The third issue facing a manager should be the quality of the documentation for the component. Well documented components are essential for reuse since it can be assumed that few components will be used *as is* from the library. The people making changes will need to clearly understand the components they are using. If the effort to reuse software is

greater than the effort to create it from scratch, reuse will, and should, fail.

Documentation standards are, therefore, needed to resolve this issue. Standards that provide consistency within the library, that address documenting the requirements levied on the component, that address the design constraints imposed during development of the components, and that address other issues such as the use of generic components.

8.4 Testing Standard

A manager who decides to use components from a library should be especially concerned with the testing of these components. Reusable components should aid the program effort by providing high quality building blocks, not introduce new problems into the system.

One way to ensure the quality is to establish minimal levels of testing to be applied to any component before acceptance in the library. Different levels of testing may be required for different levels of components, but the manager should know the difference. For instance, not all components may require the level of testing needed for systems that effect safety of flight, but the manager should not have to guess on the quality.

8.5 Test Documentation Standard

We have already assumed that most components will be changed after selection and that the components must be properly integrated into the larger systems. With these assumptions, we need to ensure the integrity of the overall new system. To ensure integrity, we can assume that extensive regression testing will be required. The overall savings can then be increased if the effort put into the original testing is also reused. The original procedures can be reused or modified as appropriate to meet the new needs of the user. Information to be captured should include: test procedures, test data, and test drivers.

To accomplish this, standards are needed for documenting the testing (procedures, data, and drivers) of components in the library.

8.6 Domain Specific Analysis

The final issue in this section is more concerned with when to apply reuse, rather than how to apply it. All current indications support the concept that software reuse is most successful in limited domains, especially those that have been carefully analyzed to determine areas having the highest probability of being reused. (This is a fundamental premise in CAMP.)

For reuse to spread, it is important to provide direction for making this analysis in new domains, that is, to gain from the experiences of domains that have performed reuse efforts. To fill this void, procedures (methods) are needed for making an analysis of a specific domain. The procedures should assess the domain and highlight areas of high payoffs for reuse. (The CAMP results should provide a sound starting point for this effort.)

9 Management Issues

This paper has reviewed the issues in the acquisition of reusable components, the use of reusable components in new acquisitions, and the concerns of managers when faced with reuse. Now the issues surrounding the library itself need to be considered. These issues need to be resolved before a reusable software library can be put into place. If multiple libraries are to be established, it is essential to define the policies and practices that will govern the library so that commonality across libraries is maintained. Currently no sufficient criteria have been established to define what components belong in the library, what management policies should guide the selection of those components, what characteristics the components should have, or when the components should be used. Once these policies are defined, components meeting the requirements can be captured and the library can be formed.

The objective of this area, then, is to establish the framework from which an Air Force software library system can evolve. To do this it is important to provide the guidelines, up front, for aspects of managing a library, such as cataloging components, identifying components, and determining access rights to the library. These are all decisions that are essential in moving components and the people managing programs that will create the components toward the goal of reusability.

9.1 Library Description

Software libraries can effectively be constructed on a number of levels. Trade-offs on a number of topics including cost, benefit, acceptability, and workability should be evaluated for each potential approach. Large government-owned libraries (warehouses), smaller government-owned libraries for unique domains, individual contractor-owned libraries, or independent commercial libraries are all viable options.

To address this, the best approach to creating a library system for Air Force weapon system software needs to be determined. This includes determining the effectiveness of large central libraries versus smaller distributed libraries.

9.2 Catalog Methods

Once an overall decision is made on the description of the library, the issue becomes how to catalog the items in the library. Currently, with many independent efforts examining reuse, each is defining a unique scheme. This is equivalent to having each book library define a catalog scheme and then hoping to share books. The effort would soon overcome the benefits.

The solution is the development of a common cataloging methodology to allow numbering of the software and documentation within the system. Part of this solution requires defining the level of components in the library. This solution can be thought of as the equivalent of establishing a new Library of Congress Numbering System.

9.3 Retrieval Methods

One thing that must be realized is that adding components is much simpler than finding and extracting a needed component. The retrieval process must provide:

- fast, efficient on-line search
- timely, interactive retrieval
- recommendations on components that partially fill requirements
- recommendations on multiple components that, when consolidated, fill or partially fill the requirements
- the capability to browse through extended definitions of the components

The solution to this issue is easier to state than it is to accomplish. The solution is to develop a retrieval system for the library that allows fast, accurate definition of the required component in a user friendly mode. (The retrieval system may be best approached as an Artificial Intelligence Expert System as was proposed by CAMP and other efforts.)

9.4 Certification Policy

Once the decision has been made to establish a software component library, and the issues of cataloging and retrieving have been addressed, there must be a discussion on evaluating potential components to include in the system. This will include defining a level of both acceptability and trust for a candidate.

To accomplish this, policy is needed to set certification levels for software and documentation components. The policy must provide consistent guidelines for determining the acceptability and trust levels of the components. For instance, software with a trust level of one might be considered to have the highest reliability and be usable in all systems while components at level two could be used in all but life critical applications.

9.5 Access Policy

With the establishment of a library comes the need to define access policy for the library. The issues are clear, but they still need to be discussed. The issues are: who can enter components into the library, who can retrieve components and use them, and who controls the access.

A policy recommendation on access is the first step in this area.

9.6 Component Change Policy

Next the configuration management practices of the library should be addressed. While configuration management is already a nightmare on normal systems, there are issues in reuse that compound this significantly. Issues such as how to store multiple versions of the same component need to be resolved. Typically, improvements involve defect removal or enhancement, but with reuse they could involve the slight changes in requirements from one user to the next. Another issue is when and if to notify present users of changes to current versions. Consider the scope of notifying all users of a reference book when the book is improved so that they can change their derived reports.

Once again the solution involves determining policy. This time the policy should cover how to handle changes to the components in the library.

10 Conclusions

This paper reviews a number of the issues that will hamper the widespread introduction of software reuse for the Air Force. The paper attempts to provide recommendations on the steps to be taken now, to lay the ground work, for reuse to become an accepted reality. Software reuse with cataloging is a partial solution to the *software crisis*, but like most of the other parts of the solution, it requires a management appreciation for the software problem and a commitment to investing in *capital* for software support. Resources are required to implement these recommendations, for without resources very little will happen. The results of applying resources, however, will be well worth the effort and expense.

Acknowledgement

This paper is based on a proposed Air Force action plan for constructing and cataloging software for reuse in mission critical computer resources. The plan was developed by the author with a great deal of help and guidance from many reviewers throughout the community.

References

- [1] C. Anderson. The CAMP project: A pragmatic approach to software reuse. In *Military Computing Conference*, May 1988.
- [2] J. Gabig and R. McVoy. The DOD's rights in technical data and computer software clause - part ii. *NCMA Journal*, Winter 1988.
- [3] D. McNicholl, C. Palmer, and et. al. Common Ada missile packages (CAMP). Technical Report AFATL-TR-85-17, Air Force Armament Laboratory, June 1985. Volumes I, II, III.
- [4] J. Neighbors. The DRACO approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, pages 564-574, September 1984.
- [5] R. Prieto-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, pages 6-16, January 1987.
- [6] W. Tracz. Software reuse myths. In *National Conference on Software Reusability*, pages A1-A12, April 1988.
- [7] P. Wegner. Varieties of reusability. In *Workshop on Reusability in Programming*, pages 30-44, 1983.