U.S. Department of Commerce

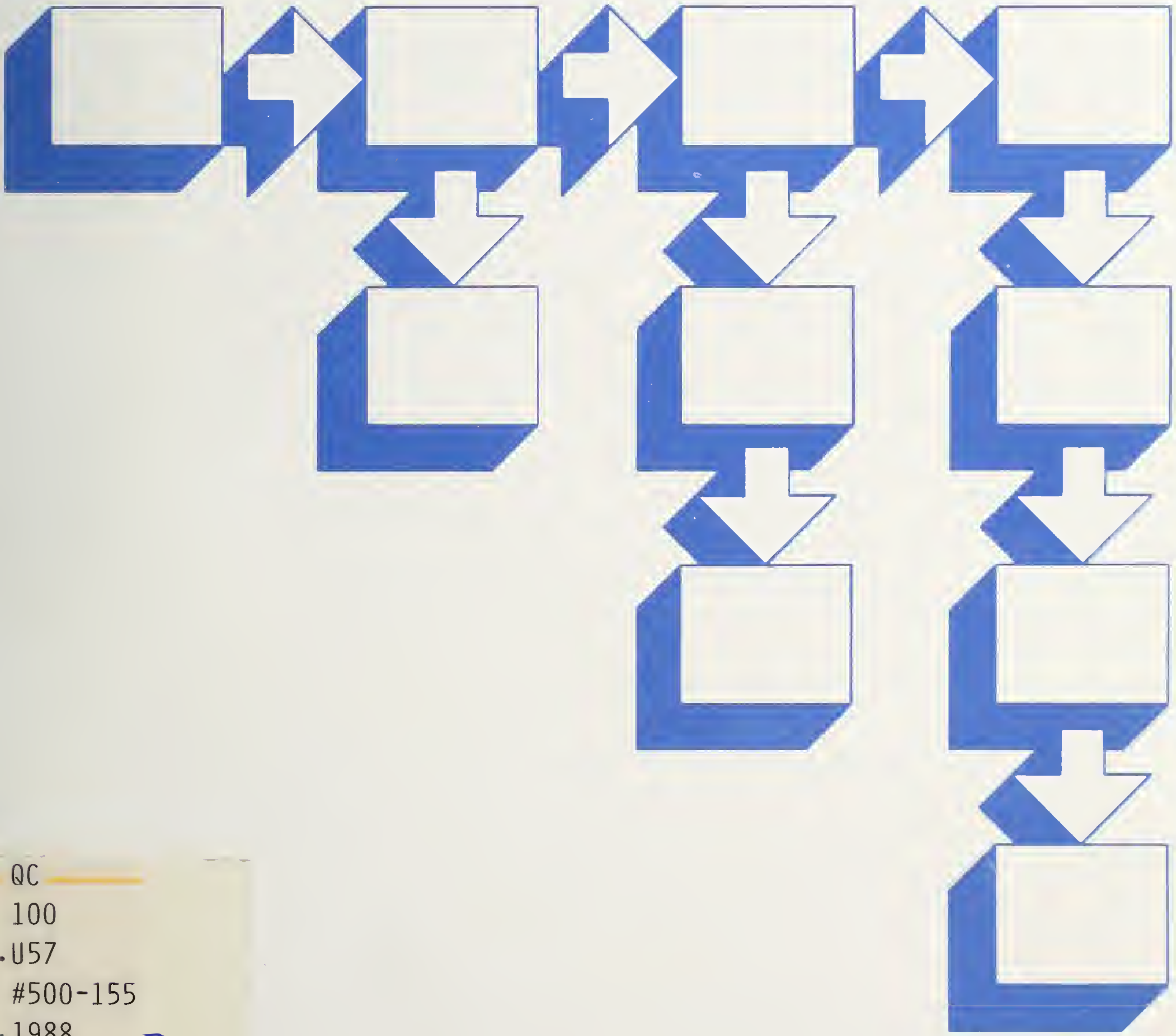
National Bureau of Standards

Computer Science and Technology

NBS Special Publication 500-155

Management Guide to Software Reuse

William Wong



The National Bureau of Standards[1] was established by an act of Congress on March 3, 1901. The Bureau's overall goal is to strengthen and advance the nation's science and technology and facilitate their effective application for public benefit. To this end, the Bureau conducts research to assure international competitiveness and leadership of U.S. industry, science and technology. NBS work involves development and transfer of measurements, standards and related science and technology, in support of continually improving U.S. productivity, product quality and reliability, innovation and underlying science and engineering. The Bureau's technical work is performed by the National Measurement Laboratory, the National Engineering Laboratory, the Institute for Computer Sciences and Technology, and the Institute for Materials Science and Engineering.

The National Measurement Laboratory

Provides the national system of physical and chemical measurement; coordinates the system with measurement systems of other nations and furnishes essential services leading to accurate and uniform physical and chemical measurement throughout the Nation's scientific community, industry, and commerce; provides advisory and research services to other Government agencies; conducts physical and chemical research; develops, produces, and distributes Standard Reference Materials; provides calibration services; and manages the National Standard Reference Data System. The Laboratory consists of the following centers:

- Basic Standards[2]
- Radiation Research
- Chemical Physics
- Analytical Chemistry

The National Engineering Laboratory

Provides technology and technical services to the public and private sectors to address national needs and to solve national problems; conducts research in engineering and applied science in support of these efforts; builds and maintains competence in the necessary disciplines required to carry out this research and technical service; develops engineering data and measurement capabilities; provides engineering measurement traceability services; develops test methods and proposes engineering standards and code changes; develops and proposes new engineering practices; and develops and improves mechanisms to transfer results of its research to the ultimate user. The Laboratory consists of the following centers:

- Applied Mathematics
- Electronics and Electrical Engineering[2]
- Manufacturing Engineering
- Building Technology
- Fire Research
- Chemical Engineering[3]

The Institute for Computer Sciences and Technology

Conducts research and provides scientific and technical services to aid Federal agencies in the selection, acquisition, application, and use of computer technology to improve effectiveness and economy in Government operations in accordance with Public Law 89-306 (40 U.S.C. 759), relevant Executive Orders, and other directives; carries out this mission by managing the Federal Information Processing Standards Program, developing Federal ADP standards guidelines, and managing Federal participation in ADP voluntary standardization activities; provides scientific and technological advisory services and assistance to Federal agencies; and provides the technical foundation for computer-related policies of the Federal Government. The Institute consists of the following divisions:

- Information Systems Engineering
- Systems and Software Technology
- Computer Security
- System and Network Architecture
- Advanced Systems

The Institute for Materials Science and Engineering

Conducts research and provides measurements, data, standards, reference materials, quantitative understanding and other technical information fundamental to the processing, structure, properties and performance of materials; addresses the scientific basis for new advanced materials technologies; plans research around cross-cutting scientific themes such as nondestructive evaluation and phase diagram development; oversees Bureau-wide technical programs in nuclear reactor radiation research and nondestructive evaluation; and broadly disseminates generic technical information resulting from its programs. The Institute consists of the following Divisions:

- Ceramics
- Fracture and Deformation[3]
- Polymers
- Metallurgy
- Reactor Radiation

[1]Headquarters and Laboratories at Gaithersburg, MD, unless otherwise noted; mailing address Gaithersburg, MD 20899.
[2]Some divisions within the center are located at Boulder, CO 80303.
[3]Located at Boulder, CO, with some elements at Gaithersburg, MD

NBSC
QC100
.U57

Computer Science
and Technology

no. 500-155
1988
c.2

NBS Special Publication 500-155

Management Guide to Software Reuse

William Wong

Systems and Software Technology Division
Institute for Computer Sciences and Technology
National Bureau of Standards
Gaithersburg, MD 20899

April 1988

U.S. DEPARTMENT OF COMMERCE
C. William Verity, Secretary

National Bureau of Standards
Ernest Ambler, Director

Reports on Computer Science and Technology

Library of Congress Catalog Card Number: 88-600528
National Bureau of Standards Special Publication 500-155
Natl. Bur. Stand. (U.S.), Spec. Publ. 500-155, 38 pages (Apr. 1988)
CODEN: XNBSAV

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1988

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington DC 20402

CONTENTS

FIGURES

ABSTRACT

This document, the second in a series on software reuse, focuses on the improvement of productivity and quality of software as well as the reduction of software risks. Software reusability can provide substantial economic benefits. Initial reusability efforts should emphasize an understanding of the concept of software reuse, and encourage the use of existing well-developed software specifications, designs, methods, techniques, tools, and other reusable information. This report presents general management guidance in software reuse. While there is no magic solution to the problem of achieving the goals of software reuse, the report discusses various aspects, problems, issues, and economic reasons of software reuse, and identifies those techniques and characteristics which will assist management in improving software reuse.

KEYWORDS

Domain knowledge; reusable software library; software acquisition; software classification; software commonality; software component; software factory; software management; software portability; software reusability; software reuse; software risks.

PREFACE

The Institute for Computer Sciences and Technology (ICST) of the National Bureau of Standards (NBS), has a responsibility under Public Law 89-306 (Brooks Act) and Public Law 100-235 (Computer Security Act) to promote cost effective selection, acquisition, and utilization of automatic data processing resources within the Federal Government. ICST efforts include research in computer science and technology, technical assistance, and the development of standards and guidelines for computer and related telecommunication systems. ICST is developing a series of software reuse reports designed to assist Federal agencies in improving software productivity and quality as well as controlling software development and maintenance costs.

This report is the second publication issued in this area. The first report, "A Management Overview Of Software Reuse", NBS Special Publication 500-142 [WONG86], was issued September 1986. NBS-SP 500-142 was designed to be used as a reference document. This report discusses various aspects, problems, issues, and economic reasons of software reuse, and identifies those techniques and characteristics which will assist management in improving software reuse.

1. INTRODUCTION

One of the most effective means of improving the productivity of software development would be to increase the proportion of software which is reused. Reusable software would not only increase productivity, but would also improve the reliability of software and reduce development time and cost. However, there are many technical, organizational, economic, cultural, and legal issues to be resolved before widespread reuse of software becomes a reality.

The basic causes of increased software costs include the explosive growth in size, complexity and increasing criticality of modern software systems, and rising personnel costs. Software costs for both development and maintenance are largely related to the labor-intensiveness of the process and the inadequate use of available technology.

Before addressing the technical and economic reasons why software should be reused, it is important to gain a perspective on recent advances in the state-of-the-practice. It is often argued that software reuse is feasible, since hardware reuse has been successful. However, this analogy is not as straight forward as it may appear. Hardware tends to be relatively simple, or consists of replicated logic elements, while software must deal with substantially more complex application issues. In addition, hardware primitive components have had a much longer time to stabilize. It takes years between the initial engineering release of a hardware component and its subsequent widespread commercial use. Software has yet to gain the benefit of this maturing process, and thus still displays a higher degree of variability [GRAB84].

Current software management concerns have focused on how to reduce software development and maintenance costs, the need for a standard operating system interface, an automated programmers' support environment, automatic programming by a computerized software factory, reusable software libraries, and organization-wide software development and configuration management guidelines and standards. Each of these represents promising innovations with major payoff potential over the next five to ten years [SILV85]. However, immediate software cost savings and software programmer productivity improvements could be gained simply through small changes in the way software projects and knowledge are managed.

A common misconception of software reuse is that it is limited to the use of existing source code. Software reuse should be broadly defined as the reuse of any information that may be collected and later used to develop other software. This definition includes reuse of available software requirements, specifications, system design, source code, modules, operating systems, documentation, analysis data, test information, maintenance information data bases, and software development plans and methodologies. The reuse of automated tools for generating software, a well-designed reusable software library for classifying and retrieving various error-free software components, and an integrated software support environment to improve software lifecycle processes are also part of the scope of software reuse efforts.

This report is organized into eleven sections. Section 2 describes the software crisis. Section 3 discusses the nature of software reusability. Section 4 presents the software reuse process. Section 5 addresses advantages of software reuse. Section 6 discusses difficulties in software reuse. Section 7 presents feasibility of software reuse. Section 8 describes the role of prototyping in software reuse. Section 9 addresses reusability and software acquisition. Section 10 presents software portability. Finally, Section 11 summarizes the importance of software reusability.

2. THE SOFTWARE CRISIS

The problems in the development and maintenance of software have increased rapidly over the past decade. There has been an explosive growth in size, complexity, and critical nature of modern software applications. There is also a lack of integrated software development environment for supporting the software lifecycle process. The inability to manage the complexity of software often results in insufficient definition of requirements and specifications, extended development time, and software cost overruns.

The software problem is not only the high software development cost, but also the poor quality of software. In fact, many organizations spend 60%-70% of their resources in maintaining old software, which includes eliminating bugs and incorporating the changes of requirements [FIPS106].

"A Management Overview of Software Reuse" [WONG86] published by the National Bureau of Standards, discussed the following reasons that software is costly:

1) Requirements of new software systems are more complex than before. Almost every national defense system contains embedded computer software which performs mission-critical functions. These software systems have high performance expectations which require the software to be highly flexible and reliable.

2) There is a lack of professional training. The need to train software professionals and end-users in new technology is often overlooked. Training programs can serve as a feedback mechanism for collecting information from users about their experiences in adapting and using these rapidly changing modern programming techniques and practices.

3) The demand for qualified software professionals exceeds the number available. There is a growing shortage of software professionals. The United States Air Force (USAF) Scientific Advisory Board has estimated that the demand for software professionals will continue to exceed available resources. There is and will be a substantial shortfall of qualified software professionals if remedial measures are not taken [USAF83, BOEH82]. As a result, the difficulty of developing quality software will continue to rise.

4) There is only limited use of software development tools and methodologies. Existing software development tools and methodologies have not been widely adopted and used to develop and maintain software. Many software managers do not know what kind of information is currently available for improving the traditional software lifecycle processes. It is difficult for them to identify the information needed for selecting the right tools and methods without the appropriate information management techniques. As a result, software productivity has only increased an estimated 3%-8% per year [HORO84].

3. THE NATURE OF SOFTWARE REUSABILITY

A common misconception of software reuse is that it is limited to the use of existing source code. There is, however, a wide variety of approaches that address software reusability. Reusable software includes any information that may be collected and later used to develop other software. Reusable software includes available software development methodologies, software requirements, specifications and designs, source code, modules, documentation, analysis data, test information, and maintenance information data bases. The reuse of automated tools for generating software and a software support environment to improve software lifecycle processes are also part of the effort in software reuse [WONG86].

The use of subroutine libraries and off-the-shelf software are the most common examples of reusing existing software. For many commercial applications, modestly priced packages are available which can be incorporated into a software system. Similarly, well-developed existing packages for scientific, government, aerospace, and mission-critical applications are available.

Some good examples of software reuse are:

1) "a study done by the Missile System Division of the Raytheon Company reported that 40%-60% of actual source code was reused in more than one software system [HORO84]."

2) "85% reuse factors have been reported in Japanese software factories [STAN83]."

3) "60% of the design and code on all business applications is reusable [LANE84]."

4) "one reason for the wide and rapidly growing popularity of the UNIX (a registered trademark of AT&T) system is that its design philosophy is based upon reusability. UNIX succeeds in the area of reusability because of the low perceived complexity of its interface (i.e., files and pipes). The actual interface is complicated because most of the details are hidden away in the environment [MUSA85]."

5) "75% of program functions are common to more than one application, and only 15% of the source code found in most programs is unique and novel to a specific application [JONE84]."

6) "reuse has a place in the creation of some very complex systems, it indicates that 12 of 16 software programs involving satellites were based on 68%-95% of the existing software which have the potential for reusability at NASA's Goddard Space Flight Center [NISE85]."

While these examples indicate the possibilities of successful reuse, the state-of-the-practice has not adequately taken advantage of these opportunities in order to make widespread software reuse a reality.

4. THE SOFTWARE REUSE PROCESS

Software reuse is the use of previously acquired concepts or objects in a new situation. Actually, reuse is a continuous matching process between new and old situations, and, when matching succeeds, duplication of the same actions. The iterative refinement process of a software development lifecycle can be viewed as another effective way of reusing existing software. In this notion, reusability evolves as an iterative process of refining requirements, specifications, design, programming, testing, and implementation throughout the software lifecycle to meet the users' needs. It is necessary that the software development lifecycle process become fully automated in the future, making the rapid prototyping approach to software development through reuse and maintenance truly feasible. This automation based software development environment approach will provide an integrated set of tools that directly supports software programmers with the "corporate memory" of knowledge as well as system requirements and specifications, design, testing, implementation, and maintenance processes [BALZ83].

4.1 Levels Of Reusable Software Information

It is important software reuse be viewed as the reuse of any information that may be collected and later used to develop other software. This definition includes reuse of available software requirements, specifications, design, source code, modules, operating systems, documentation, analysis data, test information, maintenance information data bases, software development plans and methodologies. The reuse of automated tools for creating software and an integrated software support environment to improve software lifecycle processes are also considered part of the reusable information. This software reuse information can be categorized into three different levels, some more amenable to reuse than others. Figure 1, summarizes these levels of reusable software information.

FIGURE 1 - THREE LEVELS OF REUSABLE SOFTWARE INFORMATION

1) reuse of ideas
   (e.g., specification, design, development methodologies and techniques).

2) reuse of domain knowledge
   (e.g., documentation, technical textbooks, plans, personnel, analysis data, test and maintenance information data base).

3) reuse of particular components
   (e.g., source code, subroutines, modules, operating system, packages, programming languages, tools).

1) **Reuse of Ideas** - In civil engineering, reuse of ideas consists of applying general engineering concepts such as standard design equations for determining the dimensions and materials of a beam. An example of how components are reused is selecting the beam that best meets design criteria from a set of standard beam shapes, cross sections, and materials. In software engineering, software requirements, specifications, design, development methodologies and techniques are software development ideas that can be reused to build a new system. Balzer and Neighbors present examples of how to capture and reuse an existing system to build a new application in the software development process. If the process of transforming a system specification to an executable implementation can be recorded and replayed, then when the requirements and specifications change, the implementation can be generated by reusing the previous development with slight changes. In order to effectively use this approach, the software development process must be systematically automated [BALZ83, NEIG83].

2) **Reuse of Domain Knowledge** - At present, the reuse of domain knowledge or domain information is not widely recognized. Some identify reuse of domain knowledge with Artificial Intelligence (AI) technologies (e.g., knowledge based systems). However, domain knowledge which is residing in a software programmer's head gets reused frequently in every software application that is developed or modified. Reuse of software personnel is a common way of reusing domain knowledge. As Coron et al described in their work, "domain knowledge can also be embedded in the architecture of functional collections of a reusable software library, an example is the set of libraries available on the X-Windows System. The subroutines in these libraries are arranged in a distinct hierarchy, covering four levels of programming functionality such as dialogs (the highest level), field editors, intrinsics, X library primitives (the lowest level). Developers of software applications using X-Windows reuse not only the subroutines in the X and X-Ray libraries, they also reuse a systematic technique for building windows which is enforced by the architecture of the libraries [CORO87]." The creators of the libraries have succeeded in taking the results from their domain analysis (i.e., the key concepts and methods for developing an effective windowing system) and embedding them in the structure of the libraries so that they encourage the reuse of the results of the analysis.

Application generators can also be viewed as examples of the use of domain knowledge. For well-established domains such as report generation and language parsers, the basics of generating applications in that domain are captured in a tool, (i.e., the applications generator), and only the application-specific details need to be supplied to use the tool to generate the software.

3) **Reuse of Particular Components** - The use of subroutine libraries and off-the-shelf software are the most common examples of reusing existing software components. For software component reuse to be attractive and successful, the overall effort to reuse existing software components must be less than the effort to create new software. Before a software component is reused, it must be:

   a) Identified, Located and Retrieved - Candidate software for reuse must be found among all the reusable components that are archived in the "software database management system (SDMS)." The SDMS must present users with a lucid classification scheme that appeals to their intuition. Each candidate software component must be specified in such a way that the software developer is likely to be able to find it. A complete match or a close match is made between component need and a software component available in SDMS. The major contributors are good specifications for identifying an existing software component and an SDMS with a good classification scheme.

   b) Understood - Understanding a software component means knowing what it does, how it does it, and how it can be reused. What is the software component's function? How reliable is its operational behavior and what are the performance characteristics? What are the environmental requirements and the interface through which it is modified and incorporated into the software under development?

   c) Adapted - When the software component is being reused, it must be able to be tailored or modified. Two typical kinds of modifications are:

      1- Making new entities (types) from old by modifying the entity. For example, making a binary sort routine from a binary search routine by adding functionality to the search;

      2- Making new instances of types. For example, instantiating or making specific Ada (a registered trademark of the U.S. Government, AJPO) generics with parameters that particularize it for the software in which it is included. Changing parameters is preferable to changing source code to make a new entity or new instance. Source code should be tailored from the outside using parameters [DIAZ86].

4.2 Reusable Software Classification

The effectiveness of software reuse depends upon the ability to locate and retrieve an appropriate software component from a large collection of components in a well-designed and well-documented reusable software library. A classification scheme is a domain knowledge structure that organizes collections of items to satisfy the needs of the software developers to be able to reuse an existing component for building a new system.

Classification is the act of grouping like things together. All members of a group or class produced by classification share at least one characteristic which members of other classes do not possess. Classifications display the relationships between things, and between classes of things and the result is a network or structure of relationships which may be used for many purposes. Classification is a fundamental tool for the organization of knowledge. A library is an example of classification where a collection of reusable information has been organized for easy access and retrieval.

Reusable software can be classified in terms of: size, life cycle phase product, the domain of applications in which the software will be reused, or the originating organization [GRAB84]. The amount of successful reuse is dependent upon the users' awareness of its existence and the domain of its applicability. Thus, it will be important for the reusable software library developer to provide the capability of retrieving various software components in different applications. The developer must not only build the attribute of wide applicability into the library, but also must communicate this attribute to the users of the library system.

Some of the obvious pitfalls that can diminish the economic benefit of software reuse include:

1) Wide applicability is built into the library, but that attribute is not communicated to the user through the library classification system.

2) A library is designed that has wide applicability over a narrow domain of applications, but could have been designed to cover other application areas.

3) A library is designed that has narrow applicability, but could have been designed to have wider applicability either over a single application area or over many application areas.

Proper design and classification is imperative. Narrowing of the domain of applicability will lead to the proliferation of software modules with the resulting increase in cost along with the unnecessary complication of the reusable software retrieval system.

If a software package is classified as application-specific, the likelihood of the package being applied outside of that domain will be small. For example, software classified in the domain of accounting will likely be used only for accounting. As reusable software libraries are established, it is important that software placed into these libraries be designed with as large a domain of applicability as possible.

4.3 Software Commonality

The more times a software component is used, the more economic benefit can be gained. The degree of reuse depends upon the domain of applicability of that software. The wider the applicability either across many different applications or within a single application, the greater the possibilities for reuse. Classification systems for application software reuse can be applied across two domains:

1) Degree of commonality within an application area, and

2) Degree of commonality across application areas.

An application area is a distinct business or industrial grouping. For example: missiles, aircraft, spacecraft, weapons, ships, lasers, command/control, radar, business accounting, finance, education, payroll, medicine, etc. Software in any of the above domains is a good candidate for software reuse. The objective is to design software packages that will increase the amount of software reuse. The design process should explore the possibility of expanding the capability from a specific application area to a broader domain so that software can be reused across many application areas. Figure 2, presents an example of a software package commonality index for a spacecraft system [NISE86]. The higher the index, the greater the domain of applicability that is predicted. The scale goes from 0 to 6 with 0 yielding the least commonality (i.e., degree of reuse is low) and 6 yielding the most (i.e., degree of reuse is high).

FIGURE 2 - LIST OF EACH PACKAGE AND ITS COMMONALITY INDEX

Function	Degree of Commonality	Function	Degree of Commonality
sort	6	software design	4
data structure	6	software development	4
abstract processes	6	software verification	4
computer system	6	mission function	2
software maintenance	6	input routines	2
math functions	5	output routines	2
geometric functions	5	system functions	0
matrix functions	5	warhead control	0
vector functions	5	system inputs	0
process functions	5	system outputs	0
communications	5		
guidance functions	4		
navigation functions	4		
telemetric functions	4		
computer languages	4		

Reprinted from "The Design For Reusable Software Commonality" [NISE86].

5. ADVANTAGES OF SOFTWARE REUSE

The degree of benefits from software reuse depend on the complexity and size of the software product and the differences between the old and new applications. The more complex a software system, the higher the anticipated cost to reuse it. Because a significant effort will be required to understand the structure and function of the system, modifications required to reuse a complex system will be more difficult. Debugging the modifications will be costly. Systems based on well-designed, well-tested, and well-documented reusable software, in principle, should cost less and contain fewer defects because the software has successfully been tested and used. If a software component is reused several times, the incremental cost of creating and cataloging it can be amortized over the number of times it is used. Similarly, there is benefit in reusing well-developed specifications, designs, tools, analysis data, and support environments. Effort to achieve software reusability can be seen as a capital investment. Improving productivity and quality are two main advantages in reusing existing software.

5.1 Productivity

Reusing well-designed, well-developed, and well-documented software improves productivity and reduces software development time, costs, and risks. Examples of productivity improvement include:

1) Software reuse "amplifies" programming capabilities [BIGG84]. Reusing available software allows concentration of resources on improving the software product. The programmer has less work to do in developing a piece of software when large portions of the software or design are reused.

2) Software reuse reduces the amount of new documentation and testing required, because the software component which is known to be reliable decreases the potential of unforeseen errors.

3) When the system is developed based on reusable components, it becomes easier to maintain and modify because the software developers are more familiar with the reusable components from which it is constructed, and they can more rapidly understand the complete system design.

4) It often takes less time and effort to use an existing well-designed, well-tested, and well-documented software component than to attempt to rewrite it.

5.2 Quality

Improvements in the quality of software developed from well-designed, well-tested, and well-documented reusable software components can be attributed to:

1) Software components that are designed to be reused.

2) Documentation which is developed according to established organization-wide software standards. This results in software that is well understood and likely to be used appropriately.

3) Software components that are well tested and certified for reuse. The more software is reused, the greater the probability that errors will already be found and corrected. It can also reduce future maintenance efforts.

4) Software development based on well-designed, well-tested, and well-documented reusable software offers opportunities for increased system performance when frequently used software components are transported into new systems.

6. DIFFICULTIES IN SOFTWARE REUSE

Although the concept of reusable software appears attractive from both economic and technical view points, it represents a major deviation from the traditional approach to software development. Effective software reuse may involve substantial up-front investment in order to lay the basis for future gains. It may be initially difficult to implement in an organization. As previously indicated, many technical, organizational, cultural, and legal issues make reusing software difficult. These issues include:

1) The specifications of the software are either non-existent or sufficiently ambiguous so that it is not possible to determine exactly what the software does without understanding all of the source code.

2) The cost of changing the software to perform the specific function is greater than the cost of writing new software.

3) Although the software to perform the specific function may exist, nobody on the project knows about it or those who know of its existence don't know how to find it.

4) Software that can perform the required task is available, but it is so general that it is too inefficient for the task.

5) Lack of organization-wide standardization makes it extremely difficult to share software with confidence.

6) Lack of a standard data interchange format limits both sharing data among applications and systems reusability.

7) Organizational liabilities and data rights are significant issues which impact the concept of software reusability. If not classified and managed properly, legal concerns regarding data rights and liabilities may heavily affect software lifecycle management and development.

6.1 A Software Professional Viewpoint

This section addresses both technical and cultural biases of why software programmers resist using someone else's code or design. The technical issues focus on the lack of well-designed, well-developed, well-documented, reliable reusable software component libraries and the lack of an integrated software engineering environment to support software development efforts throughout the entire software lifecycle. On the cultural side, the issues focus on the lack of confidence in reusing someone else's code and the doubt that software which is developed by another person or organization, for another system can be reused in a new system without any modifications. These issues include:

1) It is easier to write it oneself, than to try to locate it, figure out what it does, and find out if it works. If it has to be modified, then it also might be faster to write it from scratch.

2) There are few tools to help find components or compose a system from the reusable parts.

3) There are few software development methodologies that stress reusing code, let alone reusing a design or a specification.

4) It is more fun to write it oneself.

5) It would imply a sign of weakness not to be able to do it oneself.

6) "It is not my code". This is part of the "Not Invented Here" (NIH) syndrome.

7) There was no consideration by the system analyst, who specified the system, that portions of an existing system could be salvaged and reused.

8) There is little emphasis and little taught in academia on reusing software.

9) The source code or tool in question is not supported. If a bug is found, no one will take the responsibility to fix it.

6.2 A Software Managerial Viewpoint

Managers often make decisions based on more than just technical issues. Organizational and cultural issues are part of the policy making process. Many managers have little incentive to reuse existing software because they feel threatened with potential cuts in budgets and resources due to the payoffs of software reusability. Reasons for not adopting a reusable software approach include:

1) If no tools or components exist, then it will take time and manpower to create the tools and components, and to gain the expertise in their use. Such costs are generally not within the budget of a single project.

2) If special tools (e.g. application generators, or preprocessors) are used to create a program, then a customer might expect these tools to be delivered along with the product for maintenance purposes.

3) If the tools do exist for making programmers more productive, then this will make the project dependent on fewer personnel. Any reduction in staff might be perceived as reducing the manager's "empire".

4) If a defect appears in a program developed using reused components, who is legally responsible for damages?

5) If there are no standards to control what is entered into the reusable components library, then time and money must be spent setting and maintaining the standards for the library.

6.3 A Cognition/Cultural Viewpoint

Computer programming is simply one form of problem solving. Understanding the merits of existing programming paradigms from the perspective of cognitive psychology has provided valuable insight in dealing with complexity [TRAC79]. There is a strong need for a proper software development environment to facilitate the reusable software engineering paradigm. Tools and training must be available to deal with system complexity and assist the software developer in finding and understanding what reusable software components exist. A summary of the evidence gathered as it applies to reusable software follows:

1) The data a person can manipulate consciously at one moment in time is limited to 5 to 9 pieces of information [MILL56]. This limit on complexity can be overcome by proper integration or modularization of components (i.e., by collecting units of information into semantically meaningful pieces or packages). This argument also supports information hiding and object-oriented design [PARN83].

2) Experienced programmers develop applications through recursive mental process of matching pieces of the problem with solution segments with which they are familiar. Therefore, portions of designs are reused each time a piece of software is written.

3) Internal conceptualization of the knowledge base in which program/design segments reside tends to evolve, with experience, toward a uniform content for all programmers. In other words, experienced programmers tend to think alike and express their solutions in similar forms [TRAC87].

4) Programmers cannot reuse something they don't understand. Furthermore, expert programmers follow certain explicit rules of discourse regarding naming conventions and programming style which enhance program read ability and comprehension. This implies that for something to be reused, it has to be designed, developed, and documented according to an accepted set of software development standards [TRAC87].

7. FEASIBILITY OF SOFTWARE REUSE

During the last several years, there has been increasing interest in making software reusable. However, due to a number of factors, only limited success has been achieved. Managerial and cultural problems are the major stumbling blocks. On the technical side, part of the issue centers around the large differences between "code reusability in the small" and "code reusability in the large" issues including programming practices, location of reusable components, standard design, and the strong protection of proprietary software.

Even if the reuse of a software component is valuable, its reuse may not be feasible. In considering the feasibility of software reuse, both technical and organizational aspects should be examined; management incentives must be provided, problem areas defined, sufficient personnel supported, and the viability of reuse among differing versions of the same system considered.

As Wegner states: "Reuse of a component in successive versions of an evolving program appears to be a more important source of increased productivity than reuse of code in different applications. Components are rarely portable between applications and even if they are, the incremental benefit of using a component in two applications is only a factor of two. But the number of versions of a system over its lifetime can number in the hundreds or thousands [WEGN83]."

It is important to recognize the basis for the concern about the viability of component (source code) reuse across applications. Across applications in this context means applications which are at least somewhat dissimilar, as opposed to derivative versions of an application such as a specific version of an accounting system tailored to a particular customer.

It is generally agreed that application generators are capable of developing custom-tailored programs within a well-defined application area such as business accounting systems. The problem with component reuse arises as the application areas become complex and ill-defined, and as organizational boundaries are crossed. The former makes it difficult to specify exactly what is needed from each component with sufficient precision to ensure that a reused component is, in fact, what was needed. In terms of crossing organizational boundaries, the "reuse" of project personnel who are familiar with the software components and their roles and limitations, is important.

Source code reuse is perhaps the most difficult form of reuse. The reuse of high-level software (i.e., requirements, specifications and high-level design) should be easier, since the high-level software is unlike source code that is usually closely related to hardware and operating system characteristics. Source code reuse consequently is much more difficult to describe and reuse. As Balzar pointed out, "Previous implementations will not be the basis for reusability. They will not be gathered into libraries to be reused. All such attempts (with the exception of mathematical subroutine libraries) have failed and continue to fail because even with our most sophisticated forms of parameterization, the implementations are far too instantiated to mesh with the potential uses. We have no technology to characterize the set of "small" changes to functionality and/or environment that arise in these potential uses [BALZ83]." A survey of reusability performed by [CHAN83] identifies instances of feasible software reuse:

1) "Most of the successful instances of software reuse in industry involved similar, well-defined application problems. They used the same operating system and hardware. Some identified projects were able to reuse entities even though the applications were substantially different. In most situations, the functions or program units tended to be too closely linked to the specific application to be reusable across different applications."

2) "Most instances of reusability were achieved on the specification level (by matching of interfaces) and on design level (by expressing the design in a sufficiently abstract design language)."

3) "It was easier to achieve reuse at the application level than within the operating system. Application reuse was amplified considerably by common operating system and common hardware".

4) "In many cases reusability across projects was achieved because all or some of the key people worked on both projects".

The observations made above are summarized in Figure 3.

FIGURE 3 - FEASIBILITY OF SOFTWARE REUSE

---

1) software reuse across organizational boundaries should be confined to a well-defined problem area;

2) software reuse among different versions of the same system is more viable than reuse among different application areas;

3) the wider the application area, the higher level of reuse (e.g., reuse across organizational boundaries should probably be higher than the code level);

4) an integrated development environment must be common among those who reuse the same components;

5) reusability across project boundaries is often only possible when some or all of the key people have worked on both projects;

Factors that affect software reusability include:

1) **Size and complexity of the software component** - As the size and complexity of the software component increases, the feasibility of reusability decreases. Small, simple software components are usually easier to design, test, and maintain than large, complex components.

2) **The lifecycle phase which the component represents** - As the lifecycle phase of the component approaches implementation, the feasibility of reuse usually decreases. It is likely easier to reuse a requirements document than to reuse source code. Source code has more elements associated with it (e.g., operating system, utilities, parameter, interfaces, standards routines) than the high-level abstractions used in a requirements document.

3) **Domain of application** - The domain of applications in which the software is to be reused determines how well-defined and how flexible the software component must be. If the software component is used within a narrow domain of applications where terminology and assumptions are well understood, the definition of the software component need not be exceptionally rigorous. However, when the domain of applications is broad, the software component must be very rigorously defined, since the terminology and assumptions will be more varied and less well-known.

4) **Organizational boundaries** - The number of "organizational layers" that separates the person who reuses a software component from the person who initially developed the component can affect software reuse. Examples of this boundary include "within the same division in an organization", "between sections of the same organization", or "between different organizations". In general, the more organizational layers between the software component creator and the reuser, the greater the difficulty in reusing that software.

Circumstances under which software has been successfully reused include:

1) **Small Software Project** - A majority of the software developed in an organization is written by individuals or small teams of software programmers associated with a single project. Software is reused for the following reasons:

   a) It was written by a person who is reusing it.

   b) It was written by another person in the project.

   c) An application is being developed where a previous version or a similar program is available.

d) The software is for a function that:

   - is well understood;
   - has only a few data types;
   - relies on a stable underlying technology; and
   - has standards within the problem domain (scientific subroutines are examples of this type of software)

2) **Software Factory Approach** - The Japanese have taken a different approach to programming. Instead of software development, they view it as software production. They cite programming productivity increases because:

   a) They have established a critical mass in the number of reusable components and programmers available to use and develop them [JONE84].

   b) They have taken the separate phases of the software development process and assigned them to different organizations within the software factory.

   c) They have developed an integrated set of tools and standards to support reuse in the software development lifecycle. Because of the large number of software programmers using the tools, their initial development cost can be economically justified [JONE84].

   d) Software reuse is part of their training process. One software factory gives programming exercises each month to all its software programmers. These exercises require referencing the reusable software components library in order to be completed with the minimum of effort [TRAC87].

3) **Reusable software components library** - A well-designed reusable software components library can substantially improve software productivity and quality by increasing the efficient reuse of error-free code for both new and modified software systems. However, there are difficulties inherent in selecting and effectively integrating reusable software into new or existing software systems. There are a number of critical issues to be addressed in developing large libraries of reusable software components, such as configuration and change control, quality assurance, cataloging, documentation, data rights and liability.

8. THE ROLE OF PROTOTYPING IN SOFTWARE REUSE

Prototyping is an iterative process for developing and refining software requirements and specifications. It is "the building of trial versions of software systems that emphasize the preparation of immature versions that can be used as the basis for assessment of ideas and decisions in preparation of a version that is complete and deliverable [FISH87]." Prototyping offers a number of attractive advantages, such as the early resolution of high-risk issues, and the flexibility to adapt to changing environmental characteristics or perceptions of users' needs. Two major types of approaches to prototyping are referred to as specification-driven and components-driven.

In specification-driven approaches, prototypes serve to make requirements and design notions visible to system users and software developers. The major objective is to improve and refine the users' requirements and specifications. Evolving a prototype to an operational system is a secondary objective, but must not be mandated unless mature software component repositories exist. The prototype often is thrown away after the feasibility assessment is complete.

At the other end of the spectrum, the components-driven prototype is expected to be an experimental model of the full scale development system. The prototype is assembled with as many existing components from a library and off-the-shelf packages as possible. The objective is to determine what modifications to the collection of components are needed in order to make the production system acceptable to the users.

In practice, any software prototype development effort is between the two extremes. In terms of reusability, prototypes have multiple objectives. Rapid prototyping should be used if a significant amount of the new software can be derived from the reuse of existing software components. These prototypes can be used to validate the current functional requirements and to examine the system to see where payoffs can be realized through optimization of high-frequency paths. Furthermore, there may be several iterations of prototypes to address feasibility of the sought-after system, to assess the attributes of components from libraries, to analyze the interaction of reusable components, to specify off-the-shelf software packages, and to determine the shortfall of requirements to meet the users' needs.

An example of prototyping given by Guimaraes [GUIM87] describing the prototyping practice of one Chicago bank provides a good illustration of a throw away prototype:

"End users spent an average of 250 hours developing each throw away prototype (primarily requirements definition) for a group of six large applications. They then invested an average of 45 hours more per prototype on working with system developers to add other procedures. The system development group itself expended between 75 and 225 hours reusing what had already been done in the prototype. Thus, the bank supported efforts that were between 30% and 90% redundant".

The economic gains of employing evolutionary prototyping for software development coupled with proper planning, tools, Very High Level Languages (VHLLs), discipline, methodology, and user interfaces can substantially improve software productivity and quality.

9. REUSABILITY AND SOFTWARE ACQUISITION

The primary objective of software acquisition is to identify commonality of available software functionality, and to be able to use these software components to develop timely, cost-effective, reliable new software systems. The introduction of well-designed, well-developed, and well-documented reusable software into the software acquisition and development lifecycle will change the current views and practices of the software acquisition and procurement processes. Software management and software developers can take advantage of existing software components and then concentrate on software acquisition of the unique requirements for developing new software systems. This should result in better systems and shorten the software development cycle.

In order to successfully acquire and adapt existing software, incentives are required to foster the changes. If the benefits of software reuse are to be realized, a "sustaining" software development environment must be created that promotes the concept of software reuse and encourages the use of available software components. Explicit policy support and references to software reuse in a Request For Proposal (RFP) and a Software Contract are necessary to address the potential long term benefits available from software reuse. The RFP should require software developers to identify the extent of reusable software that is appropriate for the target system, within acceptable development cost, schedule, and risk level.

Major issues which impact the current software acquisition and procurement practices include:

1) RFP and Software Contract must be structured so that software reusability is explicitly addressed and encouraged.

2) Software developers must address and show how development costs and risks will be minimized by the selection of reusable software in developing new systems in an RFP.

3) Contract review and monitoring of reusability activities must be provided. Status review of software reuse must be addressed throughout the entire software development lifecycle.

4) Data rights and liability issues must be addressed and managed properly between users and software developers.

5) Software developers should be encouraged and rewarded when they meet software reuse goals and objectives by using well-developed existing software when possible.

10. SOFTWARE PORTABILITY

Software portability refers to the ease with which a piece of software can be transported to and reused in a different environment without any modification. The adaptability of a software component is defined by the ease with which its properties can be modified.

A software component is portable if the effort required to transport it is much less than the effort required for its initial implementation and if it retains its initial qualities after the transport. Software users are increasingly demanding that specific software be portable because software has become more and more expensive in comparison with hardware costs. It should also be noted that the increasing complexity of software systems has more often required that they be written in Very High Level Languages (VHLL). This factor has contributed greatly to software portability.

Obviously, software portability increases flexibility and reusability. Portability involves the consideration of environmental factors such as hardware, operating system, programming languages, interface with other software components, etc. It is very expensive, in time and in other resources, to recreate software for every new machine. It is important to know how to design and build a piece of software, so that it can be transported to other environments in order to facilitate reuse.

The advantages of designing and building portable software include:

1) As software developers build and sell portable software, they can appeal to a much wider market. Instead of building new software for each possible computer environment, it is in the developer's own interest to reduce the effort needed to change from one environment to another. The overall cost of developing portable software, and then transporting and reusing it, is preferable to rewriting the same software several times.

2) Since software is becoming more expensive than hardware, its lifecycle must be made longer, that is, it must be designed and developed to survive hardware changes.

3) Anticipation that a piece of software will be portable is likely to have beneficial effects on its programming. The coding will have to be cleaner, more systematic, more disciplined, and more readable, Hence the reliability, reusability, maintainability, and overall quality of the software will be improved.

11. SUMMARY

Software reusability can provide substantial economic benefits. Initial reusability efforts should emphasize understanding the concept of software reuse, and encouraging the use of existing well-developed software specifications, designs, methods, techniques, tools, and other reusable information. Reusing well-designed, well-developed, well-documented software can significantly enhance the ability to develop timely, cost-effective, reliable software systems.

The concept of reusable software is attractive from both technical and economical points of view. However, it may be initially difficult to implement in an organization. Many technical, organizational, legal, and cultural issues must be resolved. Software management must recognize the increasingly critical and pervasive role of software, its characteristics, and the software information management problems which must be addressed. These major issues include:

1) Current views and practices of the software acquisition and procurement processes must be changed so that software reuse is explicitly addressed and encouraged.

2) Incentives must be provided to encourage reuse of existing software.

3) Organizational liabilities and data rights issues significantly impact the concept of software reuse. If not classified and managed properly, legal concerns regarding data rights and liabilities may heavily affect software lifecycle development and management.

4) Most software programmers and managers tend to view software reusability from the perspective of simply reusing source code, whereas reusing other programming artifacts (e.g., requirements, specifications, designs, plans, tests, and methodologies) lead to more productivity. Other reusability paradigms (e.g., application generators, translation systems, Very High Level Languages (VHLLs), automated tools, and automation based software support environment) have proven successful [TRAC87].

5) Meaningful, properly designed, tested, verified, standard guidelines, and classified reusable software components need to be developed before they can be reused.

6) Tools and training must be available to deal with system complexity and assist software programmers in finding and understanding what software components are available from reusable software components library.

7) The feasibility of software reuse has been demonstrated by the Japanese Software Factories partly because of the concentration of software programmers (critical mass) that maximizes their return on tool investment [JONE84].

8) Many software applications are common and generic. Such source code is a logical target for standard functions, and reusable modules [BIGG84].

9) Software can be transportable only if standardization and reusability are goals and objectives in the original design.

10) Criteria for accepting and retaining a software entity for a reusable software components library (e.g., frequency of reuse and degree of reusability) must be established.

11) Characteristics that promote reuse of software (e,g., generality, portability, modularity, independence, maintainability, self-descriptiveness, and verifiability) should be the basis for developing standards, techniques, and measurements.

Effective software reuse requires a substantial investment up-front in order to establish the basis for future gains. While there is no magic solution to the problem of achieving the goals of software reuse, this report provides general management guidance in software reuse. Managerial problems should be considered with technical concerns when an organization attempts to reuse existing software. No matter how good the software development methodology, managerial issues will often determine the effectiveness and viability of a particular approach. Therefore, software management and software professionals must recognize the substantial economic benefits of software reuse, and properly address and resolve the issues in order to make widespread reuse of software a reality.

REFERENCES

[BALA83] Balzer, R., "Evolution as a New Basis for Reusability" Proceedings Workshop on Reusability in Programming, ITT Programming, September, 1983.

[BIGG84] Biggerstaff, T. J., Perlis, A. T., "Foreword: Special Issue on Software Reusability", IEEE Transactions on Software Engineering, IEEE Computer Society, September, 1984.

[BOEH82] Boehm, B. W., Standish, T. A., "Software Technology in The 1990's", Appendix to Software Initiative Plan, 1982.

[BOEH84] Boehm, B. W., Gary, T. E., Seewaldt, T., "Prototyping Versus Specifying: A Multiproject", IEEE Transactions on Software Engineering, IEEE Computer Society, May, 1984.

[CHAN83] Chandersekaran, C. S., Perriens, M. P., "Towards an Assessment of System Reusability", Proceedings Workshop on Reusability in Programming, ITT Programming, September, 1983.

[CORO87] Coron, H. M., Marden, J., Wong, E., "FULCRUM: A Reusable Code Library Toolset", Proceedings of Fifth Annual Pacific Northwest Software Quality Conference, Portland, Oregon, October, 1987.

[CURT84] Curtis, B., "Cognitive Issues in Reusability", Proceedings of ITT Workshop on Reusability, ITT Programming, September, 1983.

[DENN81] Denning, P. J., "Throwaway Programs", Communications of the ACM, February, 1981.

[DIAZ86] Prieto-Diaz, R., "Classification of Reusable Modules", DoD STARS Workshop, March, 1986.

[FIPS106] "Guideline On Software Maintenance", Federal Information Processing Standards Publication 106, National Bureau of Standards, June, 1984.

[FISH87] Fisher, G. E., "Application Software Prototyping and Fourth Generation Languages", NBS Special Publication 500-148, National Bureau of Standards, May, 1987.

[GRAY86] Gray, M. M., "Guide to the Selection and Use of Fourth Generation Languages", NBS Special Publication 500-143, National Bureau of Standards, September, 1986.

[GRAB84] Grabow, P. C., "Reusable Software Implementation Technology Review", Hughes Aircraft Company, 1984.

[GOUG84] Goguen, J. A., "Parameterized Programming", IEEE Transactions on Software Engineering, IEEE Computer Society, September, 1984.

[GUIM87] Guimaraes, T., "Prototyping: Orchestrating for Success", Datamation, December, 1987.

[HORO84] Horowitz, E. W., "An Expansive View of Reusable Software", IEEE Transactions on Software Engineering, IEEE Computer Society, September, 1984.

[JONE84] Jones, T. C., "Reusability In Programming: A Survey Of The State Of The Art", IEEE Transactions on Software Engineering, IEEE Computer Society, September, 1984.

[JONE86] Jones, T. C., Programming Productivity, McGraw-Hill Book Company, 1986.

[LANE84] Lanergan, R. G., "Software Engineering with Reusable Design and Code", IEEE Transactions on Software Engineering, IEEE Computer Society, September, 1984.

[LECA86] Lecarme, O., Gart, M.P., Software Portability, McGraw-Hill Book Company, 1986.

[MATS84] Matsumoto, Y., "Some Experience in Promoting Reusable Software: Presentation in Higher Abstract Levels", IEEE Transactions on Software Engineering, IEEE Computer Society, September, 1984.

[MILL56] Miller, G. A., "The Magical Number Seven Plus or Minus Two", Psychological Review, 1956.

[MUSA85] Musa, J., "The Expert Outlook", IEEE Spectrum, January, 1985.

[NEIG83] Neighbors, J. M., "The Draco Approach to Constructing Software from Reusable Components", Proceedings of ITT Workshop on Reusability in Programming, September, 1983.

[NISE85] Nise, N., Mckay, C., Dillehunt, D., Kim, N., and Giffin, C., "A Reusable Software System", Proceedings of the AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference, Long Beach, CA, October, 1985.

[NISE86] Nise, N., Giffin, C., "The Design For Reusable Software Commonality", DoD STARS Workshop, March, 1986.

[PARN83] Parnas, Clements, and Weiss, "Enhancing Reusability With Information Hiding", Proceedings of ITT Workshop on Reusability in Programming, Newport, RI, September, 1983.

[RAUC83] Rauch-Hindin, W. B., "Reusable Software", Electronic Design, February, 1983.

[SOLO84] Soloway, E., "Empirical Studies of Programming", IEEE Transactions on Software Engineering, September, 1984.

[SIMO87] Simos, M., "Alternative Technologies for Software Reusability", DoD STARS Workshop, March, 1986.

[SILV85] Silverman, B. G., "Software Cost and Productivity Improvements: An Analogical View", IEEE Computer, May, 1985.

[STAN83] Standish, T., "Software Reuse", Proceeding of ITT Workshop on Reusability in Programming, Newport, RI, September, 1983.

[TRAC79] Tracz, W., "Computer Programming and the Human Thought Process", Software Practice and Experience, Vol. 9, 1979.

[TRAC87] Tracz, W., "Software Reuse: Motivators and Inhibitors", Paper Number CH2409-1/87/0000/0358501.00 IEEE 1987.

[USAF83] USAF Scientific Advisory Board, "Report on the High Cost and Risk of Mission-Critical Software", December, 1983.

[WEGN83] Wegner, P., "Varieties of Reusability", Proceedings of ITT Workshop on Reusability in Programming, September, 1983.

[WONG86] Wong, W., "A Management Overview of Software Reuse", NBS Special Publication 500-142, National Bureau of Standards, Gaithersburg, MD, September, 1986.

GLOSSARY

algorithm - a finite set of well-defined rules that gives a sequence of operations for performing a given task.

applications software - software which performs a specific task such as word processing, spread sheet analysis, etc. (compare with system software).

compiler - a computer program which translates a high order language program into machine language which can be executed by the central processing unit.

component - a basic part of a system or computer program [*].

custom software - software specially developed for an individual application.

design methodology - a systematic approach to creating a design, consisting of the ordered application of a specific collection of tools, techniques, and guidelines [*].

design specification - a specification that documents how a system is to be built. It typically includes system or component structure, algorithms, control logic, data structures, data set use information, input/output formats, and interface descriptions [*].

development environment - a systematic approach to the creation of software with a set of integrated tools to support the software development lifecycle. The environment includes support tools for requirements and specifications, designing, editing, compiling, testing, configuration management, documentation, and project management.

development methodology - a systematic approach to the creation of software that defines development phases and specifies the activities, products, verification procedures, and completion criteria for each phase [*].

domain analysis - a generalization of system analysis in which the objective is to identify the operations and objects (e.g., design, component, specification, requirement, development method, etc.) needed to specify information processing in a particular application domain.

[*] - Adapted from IEEE Standards Glossary of Software Engineering Terminology (IEEE Std. 729) for consistency of definition.

documentation - technical data, including computer listings and printouts in human-readable form which 1) document the design or details of the software, 2) explain the capabilities of the software, or 3) provide operating instructions for using the software to obtain the desired results from computer equipment. It also includes program listings or technical manuals describing the operation and use of programs.

evolutionary prototyping - a software lifecycle based on the development of successive prototype systems to validate requirements and to expressly evolve into a delivery system [NBS148].

integration - the process of combining software components, hardware components, or both into an overall system [*].

interface - 1) a shared boundary between software modules and/or systems; 2) a hardware component which links two or more devices; 3) that function of a computer program which presents information to an operator and accepts user responses.

methodology - a well-defined development process that provides for controlled and orderly progress toward completion of a software system that meets all specified requirements within specified budget and schedule contraints.

module - a well defined section of a computer program with a specific function.

requirements specification - a specification that documents the requirements of a system or system component. It includes functional requirements, performance requirements, interface requirements, design requirements, and development standards [*].

simulation - the representation of selected characteristics of the behavior of one physical or abstract system by another system. In a digital computer system, simulation is done by software [*].

software engineering - the systematic approach to the development, operation, maintenance, and retirement of software [*].

[*] - Adapted from IEEE Standards Glossary of Software Engineering Terminology (IEEE Std. 729) for consistency of definition.

software lifecycle - the period of time that starts when a software product is initiated and ends when a product is no longer available for use. A software lifecycle typically includes phases denoting activities such as initiation, requirements analysis, design, implementation, test, installation, operation, and maintenance.

software product - software that has been developed, tested, and documented to a level suitable for delivery to a customer.

software tools - packages, computer programs, and computer systems used to help design, develop, test, analyze, or maintain computer programs, data, and information systems. Examples included high order languages, data base management systems, requirement analyzers, statistical analysis packages, and application generators.

validation - determination of the correctness of the final program or software produced from a development project with respect to the user needs and requirements [NBS75]. Validation is usually accomplished by verifying each stage of the software development lifecycle.

verification - in general, the demonstration of consistency, completeness, and correctness of the software at each stage, and between each stage, of the development lifecycle [NBS75].

X-Windows - The X-Windows Systems was developed at the Massachusetts Institute of Technology to provide high-performance/high-level device-independent graphics capability supporting a windowing interface on UNIX systems.

NBS-114A (REV. 2-8C)

U.S. DEPT. OF COMM. **BIBLIOGRAPHIC DATA SHEET** *(See instructions)*	1. PUBLICATION OR REPORT NO. NBS/SP-500/155	2. Performing Organ. Report No.	3. Publication Date April 1988

4. TITLE AND SUBTITLE

Computer Science and Technology: Management Guide to Software Reuse

5. AUTHOR(S)

William Wong

6. PERFORMING ORGANIZATION *(If joint or other than NBS, see instructions)*

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

7. Contract/Grant No.

8. Type of Report & Period Covered

Final

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)*

Same as item 6.

10. SUPPLEMENTARY NOTES

Library of Congress Catalog Card Number 88-600528

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)*

This document, the second in a series on software reuse, focuses on the improvement of productivity and quality of software as well as the reduction of software risks. Software reusability can provide substantial economic benefits. Initial reusability efforts should emphasize an understanding of the concept of software reuse, and encourage the use of existing well-developed software specifications, designs, methods, techniques, tools, and other reusable information. This report presents general management guidance in software reuse. While there is no magic solution to the problem of achieving the goals of software reuse, the report discusses various aspects, problems, issues, and economic reasons of software reuse, and identifies those techniques and characteristics which will assist management in improving software reuse.

12. KEY WORDS *(Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)*

Domain knowledge; reusable software library; software acquisition; software classification; software commonality; software component; software factory; software management; software portability; software reusability; software reuse; software risks.

13. AVAILABILITY

[✓] Unlimited

☐ For Official Distribution. Do Not Release to NTIS

[✓] Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

☐ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

38

15. Price

ANNOUNCEMENT OF NEW PUBLICATIONS ON COMPUTER SCIENCE & TECHNOLOGY

Superintendent of Documents,
Government Printing Office,
Washington, DC 20402

Dear Sir:

Please add my name to the announcement list of new publications to be issued in the series: National Bureau of Standards Special Publication 500-.

Name ____

Company ____

Address ____

City ________________ State ______________ Zip Code ____________

(Notification key N-503)

☆U.S. Government Printing Office : 1988 - 201-597/82558