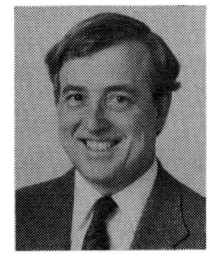
ware Engineering Enviromnents, W. Münke, Ed. Amsterdam, The Netherlands: North-Holland, 1981.
[22] W. B. Rauch-Hinden, "Reusable software," *J. Syst. Software*, vol. 2, pp. 78–92, Feb. 1983.
[23] W. Barden, *Business Programming Applications*, Tandy Radio Shack, Ft. Worth, TX, 1982.
[24] X. T. Bui, *Executive Planning with BASIC*, Sybex Inc., Berkeley, CA, 1982.
[25] R. L. Nolan, "Managing the crises in data processing," *Harvard Business Rev.*, pp. 115–126, Mar.–Apr. 1979.



T. Capers Jones received the B.A. degree in English from the University of Florida.

He is Manager of the Application Management Practice at the consulting firm Nolan, Norton, and Company, Lexington, MA. He was previously Assistant Director of Programming Technology at the ITT Programming Technology Center, Stratford, CT, and his background includes 12 years of research and managerial assignments with IBM.

An Essay on Software Reuse

THOMAS A. STANDISH

Abstract—**This paper explores software reuse. It discusses briefly some economic incentives for developing effective software reuse technology and notes that different kinds of software reuse, such as *direct use without modification* and *reuse of abstract software modules after refinement*, have different technological implications.**

It sketches some problem areas to be addressed if we are to achieve the goal of devising practical software reuse systems. These include information retrieval problems and finding effective methods to aid us in understanding how programs work.

There is a philosophical epilogue which stresses the importance of having realistic expectations about the benefits of software reuse.

Index Terms—**Software factories, software productivity, software reuse.**

I. Economic Propellants

THE quest for viable methods of software reuse has some strong economic propellants. Software demand is rising sharply on what appears to be an exponential growth curve, and the shortfall between supply and demand, currently measured in terms of 50 000–100 000 programmers, may rise to 1.2 million by 1990 if remedial measures are not taken [2], [12]. Since the supply of capable programmers will not rise nearly fast enough to close this demand-supply gap, an energetic search has begun to identify ways to improve programmer productivity. Improving productivity is a key focus of the new Software Initiative [4], [12], as well as being one of the four key areas of priority work for the new Micro Electronics and Computer Technology Corporation, a joint venture of 15 U.S. mainframe and semiconductor manufacturers.

Since economic analysis [1] indicates that the cost of software is an exponential function of software size, halving the size of the software which must be built much more than halves the cost of building it. Software reuse has thus become a keystone in many current efforts to improve productivity. One impressive way to lower the cost of building software is to pay either nothing at all or a small cost for retrieval (and perhaps instantiation) by reusing what has been previously built. Applying to software reuse what Bertrand Russell once said about the axiomatic method, we might say, "*Software reuse has the same advantage as theft over honest toil.*"

II. Actually, Software Reuse is Alive and Healthy

Recently, there has been much fresh curiosity about software reuse. An often-heard observation is that software reuse is an idea that has been around for a long time but which has never flourished. For example, there is puzzlement that McIlroy's notion [8] of a mass-produced software components

Manuscript received August 1, 1983; revised May 14, 1984. The Irvine Programming Environment Project is supported by DARPA under Contract N00039-83-C-0567.

The author is with the Programming Environment Project, Department of Computer Science, University of California, Irvine, CA 92717.

0098-5589/84/0900-0494$01.00

industry, presented at the NATO Software Engineering meeting in Garmish in 1968, has not come into existence as McIlroy envisaged. In searching for understanding as to why this idea (or any similar idea involving software publication industries) has not flourished, one often hears the speculation that individual concrete programs are too specialized to be reused in most cases—they contain too many detailed representational choices to be adaptable to new circumstances of potential use. This leads to statements such as the one R. M. Balzer made at the ITT Workshop on Reusability in Programming, that "code is not reusable," and to the speculation that what is needed is an effective method for expressing software abstractions and for generating particular instances by transformation or refinement, suited to many varying concrete settings of use.

Is it true that "code is not reusable?" Looking about, we can observe limited cases where direct reuse of software occurs without modification: libraries of mathematical subroutines (IMSL), operating system service calls (e.g., for opening files, terminal independent I/O), small granule capabilities as in Unix[1] and suites of large granule tools as in Toolpack [10].

These successful examples span a range of granularities from subcomponents (as in Unix) to components (as in mathematical libraries) to self-contained tools (such as EMACS). Sometimes large granule tools combine effectively with each other after a modest amount of customization or extension by nonspecialists, as in calling a complete text editor (for example, EMACS) as part of an interactive language interpretation system, and making EMACS knowledgeable about special templates in the language at hand [13].

Despite these limited successes, it is certainly the case that some software components are too specialized and concrete to be reusable. A common example of this is illustrated in the way we choose to reuse abstractions from data structures books. For example, suppose we are writing a compiler and we decide to use a *hash table* to represent a symbol table. We might select a book, such as Knuth, and we might choose to use, say *double hashing with pass bits* as a representation technique.

Data structures books present such algorithms abstractly and concisely, but to make use of them, we must supply special choices for things such as the table size, the space of keys, the particular hashing functions, and the format of the table entries. The abstract algorithm is thus refined into a concrete specialized instance, and is usually expressed in a convenient implementation language. However, there is a scant probability that we could ever reuse the concrete instance directly, because the new circumstances of use would probably require a completely new choice of concrete details (table size, hashing function, table entries, keys, etc.). Thus, it is the *abstraction* and not the concrete instance that gets reutilized. One view of the teaching of computer science is that it involves identifying and presenting useful abstractions—those which, at best, will be intellectual tools serviceable for a lifetime.

Software reuse is alive and well in the form of teaching and application of reusable software abstractions—the data structures and algorithms books are full of them, and the commerce in books and courses is flourishing.

[1] Unix is a trademark of Bell Labs.

Is it only the case, then, that we have failed to master the *mechanization and practical application* of deriving concrete instances from representations of abstactions? Well, here, too, there is limited success in the form of software application generators (e.g., for business forms management and limited database systems) and macro-generators. In some advanced research laboratories, such as Harvard's, extensive program transformation systems have been developed for capturing concise expressions of software abstractions and transforming them into concrete instances [3], [5].

The successful practice of software reuse appears to help considerably in the teaching of certain kinds of computer science courses. For example, in a compiler construction lab course at Irvine, we introduce a compiler, called *SmallGol*, study it for three weeks intensively to understand in detail how it works, modify it to handle extended classes of statements and expressions, and then start from scratch to write a completely new compiler for a small portion of Ada[2] emphasizing reuse of the parts of the *SmallGol* compiler. Reuse percentages, such as 68 percent,[3] are not uncommon and bear out the hypothesis that successful reuse of software components enables students to construct systems at much higher levels of system organization than would be possible if they had to write everything from scratch. Prof. Ken Bowles reports a similar success at UCSD using similar techniques which he has called, "the case method."

There appear to be fairly strong reasons for these successes. If students waste time in a compiler construction class rewriting low-level routines to search symbol tables, and translate sequences of digit characters into integers, experience indicates that, short of heroic measures, they do not have time to finish building even an unambitious compiler. In fact, we can make a stronger observation—unless we take advantage of software reuse methods, it is *nearly impossible*[4] to teach a meaningful compiler construction class in one ten-week quarter.

At normal rates of production, professional compiler writers write 1500 lines of documented debugged code per year. Thus, in one-fifth of a year on one-quarter time, students could be expected to produce 75 lines at the same rate. Unless strong use is made of software generators (e.g., parser generators, lexical analyzer generators) or unless software components (table look-up routines, data conversion routines) are reused, it is unlikely that students will be able to produce complete working (small) compilers in one quarter. By taking advantage of the leverage afforded by software reuse, experience indicates we can succeed dramatically in such software construction classes. (In the class cited, the measured rates of production averaged 2175 lines per person-month for well-documented debugged

[2] Ada is a registered trademark of the U.S. DoD (AJPO).

[3] The *metric* (applied to this and other examples in this essay) for assessing the percentage of reuse is the same as that used in one of the Japanese software factories—if a line of the source program is reused without change, it counts as a *reused line*; if it is modified in any way, even slightly, it counts as $\frac{1}{2}$ *reused* and $\frac{1}{2}$ *new*; and if it is freshly composed, it counts as a *new line*.

[4] On one memorable occasion, not only were students unable to produce working compilers starting from scratch, but they vented their frustration by smashing in the windshield of the instructor X's car and proudly wearing "I survived X" buttons at graduation.

code that was tested in live demonstrations for the final examination—and this rate of production matches that of the Japanese Software Factories [7], [9].)

As another successful application of software reuse, we built a program to generate color slides on a color CRT. This was an exercise in "rapid prototyping" which relied heavily on software reuse in order to make the prototyping truly *rapid*. We speculate that the achievement of a 62 percent software reuse level helped to give the measured rate of software production a 19.2 advantage over nominal rates of software productivity (using the software producitivity estimation techniques given in [1]). The slide maker was compiled in Ada, exported to TRW over a computer network, and used by TRW to make slides for a presentation at NCC. In Japanese software factories, reuse factors of 85 percent have been quoted [9], as have rates of software production more than eight times the comparable domestic rates. McNamara stresses that no technological "breakthroughs" have been necessary to achieve these improvements—only ordinary techniques such as key-word searching of software module abstracts and writing modules to be table-driven and reasonably parametrized.

III. Problems, Issues, and Approaches

As the reader may have gathered, we believe a case can be made (if somewhat anecdotally) that a number of valuable software reuse techniques are known today and that, in limited cases, they appear to work to advantage.

We believe that while breakthroughs and revolutionary new ways of doing business may be helpful, convincing existence proofs—such as the Japanese software factories [7], [9]—are at hand, which indicate that a great deal can be accomplished by integrating what we know how to do today and by accomplishing technology transfer of currently working, successful software reuse paradigms into production-engineered systems of practical use. "Organizational breakthroughs" may well be more imporant than "technological breakthroughs" if this is to be achieved.

It will probably not be music to the ears of an audience, devoted to advancing the frontiers of technology, to hear that having better technology might well be less critical to success than convincing management that organizational breakthroughs and focused investment are the key missing ingredients needed for success. Yet, when technology, through successful efforts of technologists, has advanced far enough to provide a successful basis for a solution and the missing necessary pieces for a solution are political and managerial, an intelligent technologist might decide to back political and organizational initiatives. Thomas Edison is an example of a technologist who was smart enough to understand this.

Having set this context, the following is a brief list of problem areas that we believe could lead to an improved technology of software reuse, if addressed successfully:

1) Information Retrieval: We must learn how to organize, index, describe, and reference software components effectively. We believe that a system of "software component folders" could be organized and indexed by conventional techniques for indexing papers in the computer science literature, and that by having each component in a software library in a form susceptible to parametric variation and refinement, an effective initial solution to this problem could be found.

2) Software Generators: We need a systematic way to generate concrete instances of abstract software modules by substitution of specific parameters. More generally, we need to master program transformations that change data and algorithmic representations as we move across layers of the refinement hierarchy. We believe three developments indicate promise: Ada generics, conventional macro-generation [11], and the ECL program transformation system [3].

3) Component Composition Paradigms: What are the methods by which we compose components when we build systems? Nested function calling, Unix pipes and redirected I/O, and shared interface designs (as in Ada package specs) coupled with information hiding (for firewalling private information against name clashing and unauthorized use) provide three examples that could be used as a basis of a took kit of general methods. Is more needed? For a start, we doubt it.

4) Program Understanding: Before we can reuse certain kinds of software (i.e., the kinds we have to modify or upgrade), we must understand how the software works. Already, if maintenance costs 70-90 percent of the life cycle, and understanding occupies 50-90 percent of maintenance cost [6], *program understanding time* may be the dominant time in the entire software life cycle and thus the dominant cost. So a big sleeping issue may be the problem of program understanding—how do we make it easy and cheap for people to understand how programs work? Will it be the case that the dominant cost in those kinds of software reuse which do not involve reusing modules as black boxes will be the cost of program understanding? If so, we are led, in turn, to ask—how do we create effective software explanations?

5) Benefits Analysis: We need to do our homework on what kinds of benefits we can expect to achieve from mastering various sorts of software reuse techniques, such as *direct reuse of concrete modules, reuse after refinement*, and *reuse after modification*. How much of a new system can we expect to synthesize by these different forms of reuse? And, for different families of systems, what percentages of them could be candidates for subsequent reuse? What shall the granularity of reuse be? Here, it seems a great deal of insight could be gained from collecting and analyzing data derived from examples where software reuse techniques have been successfully applied in practice [7], [9]. We should expect payoff in narrowly focused application areas where we recreate slightly customized versions of a single species of system over and over from large, well-understood parts libraries (as is the case in at least one Japanese software factory). We would predict low payoff from ill-understood areas that have not undergone much engineering evolution (see Part IV).

IV. Philosophical Epilogue

We ought not to have overinflated or underinflated expectations about software reuse. As an engineering discipline matures, its terminology, methods, composition paradigms, and basis of components undergo expansion and standardization.

Compare computer graphics as it was 30 years ago to the way it is today. Today we have an impressive vocabulary of con-

cepts which are useful for synthesizing artifacts: windowing, clipping, inking, rubber-banding, latching, menuing, perspective transformations, shading, hidden-line elimination, and so on. There are large numbers of algorithms, representations, techniques, hardware devices, concepts, and abstractions to support engineering activities in computer graphics.

Only after a field of engineering has undergone considerable evolution and considerable traffic in applications can we expect the basis for a useful practice of software componentry to emerge, because only after the reusable abstractions have been discovered, standardized, and taught widely is there a basis for mental training required for effective naming, classification, and reuse. For effective reuse, it seems axiomatic that trained people will have to know about the existence of reusable components, about what they are good for, and about how to use them in applications. If, for example, this is so for electronics hardware components, should it be any less so for software components?

The mental investment represented by this sort of vast engineering edifice is staggering and the costs of exploration needed to produce it are even more staggering. We should not expect useful arts of software reuse to emerge in arbitrary subfields, and least of all in uncultivated ones (e.g., databases supporting knowledge-based systems). Success, if it is attainable at all, might well be attained first in subfields where we have a highly developed art of system building complete with a mature engineering substrate, and in areas where we will have a continuing demand for nearly identical special systems that will have to be created anew for each new computer system we build (e.g., assemblers, business data processing interfaces, limited kinds of database systems, etc.).

References

[1] B. W. Boehm, *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[2] B. W. Boehm and T. A. Standish, "Software technology in the 1990s: Using an evolutionary paradigm," *Computer*, vol. 16, Nov. 1983.

[3] T. E. Cheatham, J. A. Townley, and G. H. Holloway, "A system for program refinement," Center Res. Comput. Technol., Harvard Univ., Cambridge, MA, TR 5-79, Aug. 1979.

[4] L. E. Druffel, S. T. Redwine, Jr., and W. E. Riddle, "The STARS program: Overview and rationale," *Computer*, vol. 16, Nov. 1983.

[5] *The ECL Programmer's Manual*, Center Res. Comput. Technol., Harvard Univ., Cambridge, MA, TR 32-74, Dec. 1974.

[6] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," *Commun. Ass. Comput. Mach.*, vol. 21, June 1978.

[7] Y. Matsumoto *et al.*, "SWB sytem: A software factory," in *Software Engineering Environments*, H. Hünke, Ed. Amsterdam, The Netherlands: North-Holland, 1981, pp. 305–318.

[8] M. D. McIlroy, "Mass produced software components," in *Software Engineering*, P. Naur and B. Randell, Eds. Garmisch, Germany: NATO Sci. Committee, Jan. 1969, pp. 138–155.

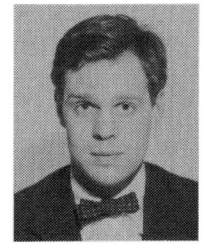
[9] D. McNamara, "Japanese software factories," presented at Comput. Sci. Colloq., Univ. California, Irvine, May 1983.

[10] L. J. Osterweil, "Toolpack–An experimental software development environment research project," *IEEE Trans. Software Eng.*, vol. SE-9, Nov. 1983.

[11] D. A. Smith, "Rapid software prototyping," Ph.D dissertation, Univ. California, Irvine, May 1982.

[12] "Software technology for adaptable, reliable systems (STARS) program strategy," Dep. Defense, Nat. Tech. Inform. Service, no. AD A128981, Mar. 1983.

[13] S. H. Willson, "Experimental template-driven Ada editor based on EMACS," Program. Environ. Project, Dep. Comput. Sci., Univ. California, Irvine, Jan. 1983.



Thomas A. Standish (M'83) received the B.S. in mathematics (magna cum laude) from Yale University, New Haven, CT, in 1962, and the Ph.D. in computer science from Carnegie Institute of Technology, Pittsburgh, PA, in 1967.

He is a Professor of Computer Science at the University of California, Irvine. He is also Chairman of the Board of the Irvine Computer Science Corporation. Previously, he was Chairman of the Department of Computer Science, University of California, Irvine. Before that, he taught at Harvard and Carnegie-Mellon, and was a Senior Scientist at Bolt Beranek and Newman, Inc. He also served as Editor-in-Chief of the *ACM Monograph Series* and as Programming Languages Editor for *Communications of the ACM*. In 1980, he published the book *Data Structure Techniques*.

Dr. Standish is on the IEEE Technical Committee on Software Engineering.