Software Reuse Myths

Will Tracz

Program Analysis and Verification Group[1]

Computer Systems Laboratory - ERL 402

Stanford, California 94305

TRACZ@SIERRA.STANFORD.EDU

Abstract

Reusing software is a simple, straightforward concept that has appealed to programmers since the first stored-program computer was created. Unfortunately, software reuse has not evolved beyond its most primitive forms of subroutine libraries and brute force program modification. This paper analyzes nine commonly believed software reuse myths. These myths reveal certain technical, organizational, and psychological software engineering research issues and trends.

1. Introduction

The concept of software reuse has been part of the programming heritage since the origins of the stored-program computer EDSAC at the University of Cambridge in 1949 where the first subroutine library was proposed. Until recently, little has been done to extend program reusability beyond this rather simple level. McIlroy [14] in 1968 envisioned software component factories, but apparently the only the Japanese listened. As this decade draws to a close, interest in applying this simple, but effective concept of not reinventing the wheel has been reborn. Again and again the role of reusable software has been recognized and discussed [9], [11], [2].

This paper identifies and examines nine myths that surround reusable software. These myths partially explain why software reuse has not had, to date, the broad sweeping effects envisioned by the programming prophets. The myths are as follows:

1. *Software reuse is a technical problem.*
2. *Special tools are needed for software reuse.*
3. *Reusing code results in huge increases in productivity.*
4. *Artificial intelligence will solve the reuse problem.*
5. *The Japanese have solved the reuse problem.*
6. *Ada[2] has solved the reuse problem.*
7. *Designing software from reusable parts is like designing hardware using integrated circuits.*
8. *Reused software is the same as Reusable software.*
9. *Software reuse will just happen.*

Each of these myths will be examined individually in the sections that follow.

---

[1]The author is an employee of the IBM Federal Systems Division, Owego, NY, participating in the IBM Resident Study Program at Stanford University.

[2]Ada is a registered trademark of the U.S. Government-Ada Joint Program Office.

2. Myths Revealed

2.1. Myth #1: Software Reuse Is a Technical Problem

Many good people have been lead astray by assuming that the software reuse problem needs a technical solution. While there are both technical and non-technical barriers inhibiting software reuse, if one looks at the most-often-stated reasons why software is not reused [20], the overwhelming majority of them may be classified as psychological, sociological, or economic. The only technical reasons cited are the lack of search methods to find the right pieces or the lack of quality components (to put in the library in the first place) to reuse. In the latter case, management plays a critical role in setting and enforcing standards, as well as motivating programmers to develop components, and to design software based upon them. The library issue will be addressed in the following myth. Finally, as stated by Ratcliffe [17], "... the whole western economic system may be against reuse." The development of software reuse has been stunted by intra-company and inter-company legal, contractual as well as political conflicts.

2.2. Myth#2: Special Tools Are Needed for Software Reuse

The term *special* is meant to imply tools tailored specifically to facilitate the reusable software engineering process. When reviewing the tools proposed to support software reuse, researchers generally include a library facility and perhaps a standards-checking program or syntax-directed editor. If one examines the most successful applications [12], [13] of software reuse to date, one finds that few, if any, tools at all are used. The Japanese software factories use a simple Key Word In Context (KWIC) index to locate the desired function. Furthermore, production software libraries seldom exceed 100-200 components in size, a number that is very easily managed manually by most programmers. (There are instances of very large repositories of subroutines as with the National Bureau of Standards Library, which contains over 2800 entries. In this case, a library system is a necessity.) In the instances where prototype reusable software libraries have been developed, they have either been created using a relational data base system [5], [15], or an information retrieval system [8], [1]. Finally, in most instances of software reuse, programmers modify existing programs, written by themselves or by a programmer on the same project [10]. In this case, because of the proximity and accessibility to the resource, the programmer does not need a sophisticated tool to locate the software to reuse.

2.3. Myth #3: Reusing Code Results in Huge Increases in Productivity

What is *huge*? Studies have shown [11] that even if 40% of a design and 75% of the code on a given project is reused, the resulting 50% reduction in testing, and comparable reductions in integration test, documentation, and system test, result in a net productivity gain of only 40%. In order to achieve an order-of-magnitude improvement in software productivity, one must resort to application generators, or highly parameterized components. (Still, cutting software development time roughly in half is not bad.) The real payoff is realized by the decreased maintenance costs! Maintenance cost reductions of up to 90% have been reported when reusable code, code templates, and application generators have been used to develop new systems.

On the other hand, one cannot ignore the initial start-up costs. Software designed for reuse costs between 20% and 25% more to develop and learn to use. The break-even point is not reached until after the second or third use.

2.4. Myth #4: Artificial Intelligence Will Solve the Reuse Problem

How can we automate something we have no expertise in? Actually such a statement is not entirely fair, because AI does have something to offer. There are strong similarities between the problem domain analysis performed by systems analysts trying to extract common components for reuse in similar applications and the domain analysis performed by heuristic search algorithms trying to match requirement specifications to program frames or schemas. However, automatic generation of code from requirements is still a research area. Expert systems have been designed to assist programmers in locating components [4] that match desired functions, and instantiate them [6], but the lack of a good notation to represent the semantics of software is still the major roadblock to unleashing the power of AI approaches.

2.5. Myth #5: The Japanese Have Solved the Reuse Problem

Many hold the somewhat mystical belief that the Japanese have solved the problem (fill in the blank). The success of the Japanese software factories is not based on any technological breakthroughs, but on the formalization of the process and the product. A question that gets answered "no" only once in a Japanese software factory is "Does a part exist that performs this function?" Japanese programmer training and sense of commitment to standards also strongly facilitate reusable software engineering.

Another reason for the success of the Japanese software factory may be summarized in the following paraphrased motto: "Ask not what you can do for your software, but what your software can do for you." By making a business decision to address a particular problem domain and recognizing the leverage software reuse plays, the Japanese have justified amortizing the cost of developing the **critical mass** of reusable software and the associated software engineering environment ultimately necessary to succeed.

2.6. Myth #6: Ada Has Solved the Reuse Problem

Writing a generic package in Ada does not necessarily make it reusable any more than writing a Fortran subroutine or assembly language macro. The adaptability (and reuse potential) of a software component depends on the amount of domain analysis performed and the degree a module is parameterized to reflect this. Furthermore, the type of parameterization facilities provided by the programming language may not always support the degree or form of adaptability desired, as is the case with Ada generics. The same holds for a class in Smalltalk or other object-oriented languages. While certain language features do facilitate the development of reusable software, the language, in itself, is not enough to solve the problem.

2.7. Myth #7: Designing Software From Reusable Parts is Like Designing Hardware Using Integrated Circuits

Why don't there exist software building blocks which programmers can wire together to build systems similar to integrated circuits If they did exist, what type of CASE (Computer Aided Software Engineering) environment would be necessary to support them? Are electrical engineers that much smarter than software engineers? Superficially, comparing software design and hardware design [7] is a very appealing analogy. At one level of complexity, the analogy holds. Structured programming relies on a select handful of basic structures (e.g., An "if-statement" is similar to a 2-way multiplexor. A "for-loop" is like a counter). Unfortunately, the analogy breaks down [16] when one realizes that both the number and the complexity of software components far exceeds those currently used by logic designers. Because of the variety of applications, the wide spectrum of problem-domain-specific components, and, most of all, the amount of "glue" necessary to connect software components together, the similarities between software and hardware design have yet to be fully exploited. Other hard problems include identifying the building blocks and defining (and documenting) the interfaces and parameterization. Finally, economic factors that differentiate hardware design from software design. For practical reasons hardware designers must constrain their design to be based on available components, whereas software designers can create designs based on custom components. Furthermore, unused functionality in a hardware chip doesn't effect the chip's performance, whereas excess code can effect program size as well as performance.

2.8. Myth #8: Reused Software is the Same as Reusable Software.

A corollary to this myth is that "A good way to develop reusable software is to take an existing program and add parameters." Both these myths fail to emphasize the need to design for and document for reuse. Unplanned reuse of software (also called software salvaging) occurs frequently in the software community. Programmers often extract modules or code segments, and then modify them to meet their needs. This is an error-prone and time-consuming process, which could be avoided if the software were designed initially with reuse in mind. As the corollary implies, reuse should be considered at design time, not after the implementation has been completed. The emphasis should be on *planned* reuse. Special attention needs to be placed on interface design and modularization (e.g., low coupling and high cohesion).

2.9. Myth #9:Software Reuse Will Just Happen

Judging from the limited success software reuse has enjoyed to date, most software reuse is not planned; therefore, the full potential has not been realized. Yet, times are changing. As hardware costs decrease and performance increases, customers are becoming less willing to buy a costly customized piece of software when a slightly more inefficient (due to some overhead in parameterization) but less costly software may do. In order to reach this goal, components need to be designed, documented and implemented for reuse [3] according to some guidelines [18], [3], [19]. Finally, management needs to provide the incentives to motivate and reward the application of this technology [21].

3. Conclusion

This paper has presented one perspective on why software reuse has not played a major role in improving programmer productivity. The realities of the myths discussed are as follows:

1. *Software reuse is a technical and non-technical problem.*
2. *No "special "tools are needed for software reuse. Available data base technology can be applied to help organize and retrieve software in large repositories.*
3. *Reusing code will not result in an order-of-magnitude increase in productivity and quality.*
4. *Artificial intelligence technology can play a role in solving the reuse problem.*
5. *The Japanese have taken the first steps toward solving the reuse problem.*
6. *No single language alone can solve the reuse problem.*
7. *Designing software from reusable parts is not like designing hardware using integrated circuits.*
8. *Reusing software that was not planned for reuse is harder than reusing software that was designed for reuse.*
9. *Software reuse will not just happen.*

While reusable software solve the software crisis by itself, it has the potential to make a significant impact. By exposing the preconceived myths about software reuse, this paper should help programmers and managers direct their efforts and resources more effectively and, thus, achieve more readily the goal of reusable software engineering.

References

1. Arnold, S.P., and Stepoway, S.L. The Reuse System: Cataloging and Retrieval of Reusable Software. Proceedings of COMPCON '87, February 23-27, 1987, pp. 376-379.

2. Biggerstaff, T. and Richter, C. Reusability Framework, Assessment and Directions. Proceedings of The Hawaii International Conference on System Sciences, January 7-10, 1987, pp. 502-512.

3. Braun, C.L., Goodenough, J.B., Eanes, R.S. Ada Reusability Guidelines. 3285-2-208/2, SofTech, Inc., April, 1985.

4. Braun, U. An Expert System for the Retrieval of Software Building Blocks. TR 05.373, IBM Laboratory Boeblingen, 1986. In German.

5. Burton, B.A., and Broido, M.D. A Phased Approach To Ada Package Reuse. Proceedings of Software Technology for Adaptable Reliable Systems (STARS) Workshop, April 9-12, 1985, pp. 83-98.

6. Defense Technical Information Center. CAMP: Common Ada Missile Packages. Pamphlet.

7. Cox, B.J. Object-oriented Programming, Software-ICs and System Building. Proceedings of National Conference on Software Reuseability and Maintainability, September 10-11, 1986.

8. Frakes, W.B., and Nejmeh, B.A. Software Reuse Through Information Retrieval. Proceedings of The Hawaii International Conference on System Sciences, January 7-10, 1987, pp. 530-535.

9. Freeman, P. Reusable Software Engineering: Concepts and Research Directions. Proceedings of ITT Workshop on Reusability in Programming, September 7-9, 1983.

10. Grabow, P.C., and Nobles, W,B. Reusable Software Concepts and Software Development Methodologies. Proceedings of National Conference on Software Reuseability and Maintainability, September 10-11, 1986.

11. Horowitz, E., and Munson, J.B. "An Expansive View of Reusable Software". *IEEE Transactions on Software Engineering SE-10*, 5 (September 1984), 477-487.

12. Lanergan,R.G. and Grasso, C.A. "Software Engineering with Reusable Design and Code". *IEEE Transactions on Software Engineering SE-10*, 5 (September 1984), 498-501.

13. Matsubara, T., Sasaki, O., Nakajim, K., Takezawa, K., Yamamoto, S. and Tanaka, T. SWB System: A Software Factory. In *Software Engineering Environments*,
North-Holland Publishing Company, 1981, pp. 305-318.

14. McIlroy, M. D. Mass Produced Software Components. Proceedings of 1969 NATO Conference on Software Engineering, 1969, pp. 88-98.

15. Onuegbe, E.O. Software Classification as an Aid to Reuse: Initial Use as Part of a Rapid Prototyping System. Proceedings of The Hawaii International Conference on System Sciences, January 7-10, 1987, pp. 521-529.

16. Polak, W. Maintainability and Reusable Program Designs. Proceedings of National Conference on Software Reuseability and Maintainability, September 10-11, 1986.

17. Ratcliffe, M. "Report on a Workshop on Software Reuse held at Hereford, UK on 1,2 May 1986". *SIGSOFT Software Engineering Notes 12*, 1 (January 1987), 42-47.

18. STARS. STARS Reusability Guideline V4.0.

19. St. Dennis, R. J., Stachour, P., Frankowski, E., Onuegbe, E. "Measurable Characteristics of Reusable Ada Software". *Ada Letters 5*, 2 (March-April 1986), 41-49.

20. Tracz, W.J. Why Reusable Software Isn't. Proceedings of Workshop on Future Directions in Computer Architecture and Software, May, 1986.

21. Tracz, W.J. Software Reuse: Motivators and Inhibitors. Proceedings of COMPCON87, February, 1987.