Separate Money Tubs Hurt Software Productivity

Dr. Ronald J. Leach
Howard University

Most software development organizations operate under four goals: more, better, cheaper, and faster. Reuse of existing software is often considered as a way to achieve these goals. Unfortunately, the project accounting practices of many organizations unwittingly discourage software project managers from improving costs and quality simultaneously. Here we show how a simple change in management practice and project accounting can encourage software development that meets these four goals. The simple change is consistent with the work of industry leaders such as Barry W. Boehm, David Weiss, James Coplien, Chi Tau Lai, and others on product line architectures.

In the November/December 1995 issue of the *Journal of Systems Management*, Paul Newcum [1] listed 13 problems that pervade the software industry:

1. Complexity.
2. Jargon.
3. Imprecise and inconsistent specifications.
4. Lack of up-front prototypes.
5. Lack of reusable software components.
6. Lack of realistic costs and schedules.
7. Difficulties using new paradigms.
8. Unrealistic deadlines.
9. Not removing defects and errors.
10. Quality not pursued.
11. Defects and errors regularly placed in software.
12. Poor business functions delivered initially.
13. Poor measurements of design and programming.

Any practicing software engineer, software project manager, or chief information officer can attest to the accuracy of Newcum's assessment. Unfortunately, the interaction between several of these problems is not as well understood as it should be. This article considers two of these – reuse and quality – and illustrates how some current management practices discourage possible quality improvement and cost savings. It then suggests simple changes that can help achieve software that meets all four objectives most organizations operate under: developing *more* software *better*, *cheaper*, and *faster*.

An Extremely Rosy Scenario

Suppose that an organization has two software projects, A and B. Suppose that half of the source code that is developed in project A also can be reused in project B. Suppose also, for the sake of simplicity, that the two projects are the same size and that the two projects interface smoothly, with no additional costs due to lack of imprecise or inconsistent specification standards (another on Newcum's list of pervasive software problems). Finally, assume that the portion of project A that is reused in project B does not require any changes.

If the manager of project A produces the software on time and within budget, and the software meets the predetermined quality standards (usually measured in the number of software defects per thousand lines of source code, or number of failures per thousand hours of operation of the software), then everyone is happy and he or she is likely to be rewarded.

"Most development organizations that have successful reuse programs recognize that there is overhead associated with reuse; this overhead is simply part of the developing organization's cost."

What does this mean for the manager of project B? Suppose that he or she needs to reuse half of the source code from project A. Since only half of project B consists of new code, it is logical to assume that this project should have a smaller budget than project A. In most organizations, the determination of just how much smaller the budget should be depends upon the organization's experience with proper cost estimation for software projects with some amount of software reuse.

In this best of all possible worlds, the organization's cost estimation takes into account the amount of reuse and whether the requirements, design, or source code from project A are being reused in project B. (Earlier reuse is better than later, since the costs of all the remaining activities in the software life cycle of project B can be avoided from the point that the software is reused. There is no need to budget for requirements engineering, software design, coding, testing, or integration for any software that already exists.)

In this extremely rosy scenario, project A is produced on time, within budget, and with the expected level of quality; project B will also be produced on time, within budget, and with the level of quality expected by the organization. With perfect software reuse and extremely accurate cost estimation, project B has been created by a software development process that is both *cheaper* and *faster*. The high level of reuse has made the organization more efficient, giving us *more* software per month. It may even give us a *better* quality product for project B, because most of the software errors that normally occur during normal software development have already been removed (we hope) in project A.

Everything is wonderful. Or is it?

A Slightly Less Rosy Scenario

One problem that can occur even in this extremely rosy scenario is that project B may have a more stringent requirement for quality than did project A. For example, small errors that occur with improper capitalization of messages in a help system may not be worth fixing in a text editor. The quality of this system is probably sufficient, even with the error.

However, in a safety-critical application such as a user interface for a heart monitor, a confusing message can be the difference between life and death. An error in a seldom-used statistical routine can be ignored if it occurs in an inexpensive spreadsheet. The same error in software that controls the placement of coolant in a nuclear power plant can be disastrous. The problem in both cases is that software that is perfectly adequate for one application becomes dangerous when used in another. You can hear the legal team shuddering.

It appears that software reuse cannot always provide improvements in software quality, and in fact may degrade performance if integrated with higher quality components.

Clearly software reuse is dangerous, and can be expensive. Or is it?

A Solution

The difficulty here is that there is no incentive for project A to produce any higher quality of software than is needed for its requirements. The manager of project A views the budget as a tub of money, which can be dipped into to get project resources. The manager of project B has a similar view, with perhaps a different sized tub.

Many organizations use what some have called the *every-tub-on-its-bottom* approach to funding software projects. In this funding approach, the manager of a project is given a budget for completion of his or her project. The manager is rewarded for completion of the project under budget and within schedule, and held responsible to some extent if the project is either over budget or late (or both). The *tubs* of money are considered by project managers as resources to be used solely for their own projects.

If upper-level management follows the every-tub-on-its-bottom approach, then there is no incentive for the manager of project A to improve project quality to improve costs for another project. Even if the manager of project A decided to do so, there are no additional resources available to increase the quality of the product. In this case, the goals of more and better are directly in opposition to the goals of *cheaper* and *faster*.

The solution is for the developing organization to apply a small *reverse tax* to every software project that is likely to produce software that will be reused in another project. It is called a reverse tax because it is added to the budget of the project teams to allow them to provide the extra quality for contribution to a pool of reusable code. The *increased* funds provided in this reverse tax allow potentially reusable software to be given a quality check for its number of known errors, adherence to standards, documentation, and so on. The activities in this quality check are often referred to as certification in the software reuse literature. Certification of potentially reusable software can be paid by the reverse tax.

There are several questions that arise when considering the application of this reverse tax:

- **Who pays for this tax – the customer or the developer?** The development organization is responsible. Most development organizations that have successful reuse programs recognize that there is overhead associated with reuse; this overhead is simply part of the developing organization's cost. (Of course, to some degree, the customer always pays for the cost of development as part of the total software life-cycle cost, operational cost, and the true cost of having the software that is really needed by the customer's organization.)

"Particular projects that will create reusable components have some form of reverse tax added to their budgets for incorporating additional quality into these reusable components."

- **How is it arranged?** Generally speaking, the development organization is responsible, although the customer may participate actively. Any organization considering a systematic approach to software reuse must do some *domain analysis* – the term used to describe, for example, determining how many additional projects are likely to need some portion of the current software [2, 3, 4]. Domain analysis is a generalization of systems analysis, in which the primary objective is to identify the operations and objects needed to specify information processing in a particular application domain. Domain analysis will precisely identify domains and software artifacts within these domains that are good candidates for reuse, and will estimate the economic benefits of reusing these software artifacts.
- **Who is doing this domain analysis?** The domain analysis is done by the development organization, in consultation with domain experts that may be outside consultants, members of the developer's internal staff, or even customer representatives.
- **Is there an overhead for systematic software reuse?** Of course there is overhead. Nothing is really free in the software industry. The overhead of systematic software reuse has been estimated at about 5 percent of overall cost, assuming that there is a metrics program, such as Capability Maturity Model® Level 2 or higher, in place [2].
- **Within the development organization, who pays for this overhead?** Other projects that are either concurrent with the selected projects, as well as future projects that will use the code pay for this tax. Particular projects that will create reusable components have some form of reverse tax added to their budgets for incorporating additional quality into these reusable components. The increased budget is intended to improve quality, not develop software from scratch.
- **Is there a net cost to the development organization?** There should be no net cost, provided that a reusable component that benefits from the *reverse tax* is actually reused.
- **Is there a net benefit to the development organization?** Yes, the net benefit is the difference between the cost of new development with a reused component versus the cost of new development. The cost savings increases greatly if the component is reused more than once.
- **How does the organization determine the appropriate amount of the reverse tax?** Additional testing and quality control measures (called certification) must be employed for each software artifact to be reused. The cost for this certification is gen-

® The Capability Maturity Model is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

erally under 5 percent per reused artifact.
- **What is the potential effect on future projects?** They may be cheaper to develop, since not all code needs to be developed from scratch, and any reused code is certified as being of very high quality.
- **Are there any other potential problems with this accounting approach?** In some cases, there might be legal roadblocks. These roadblocks are unlikely, however, since many projects reusing software artifacts are written for the same customer.

Using the resources provided by this reverse tax allows a project to produce a higher quality system than it might do otherwise. If another project can reuse the higher quality source code that was produced by project A, then the initial extra cost due to the higher quality is recovered for the organization. Therefore, there is no additional cost from an organizational view.

As stated before, the situation changes for the better if there are several software development projects that can use the reusable code produced by project A. Improving project A's quality by reducing errors, improving documentation, and standardizing all software interfaces can simultaneously improve the quality and reduce the cost of all systems that reuse the source code from project A. This is clearly the way to get software projects that simultaneously achieve all four goals. Software development can be more, better, cheaper, and faster.

It is clear that the relatively simple institutional changes in accounting practice described in this article can make it possible for projects to improve both productivity and quality with a decrease in overall cost. The approach is essentially risk-free, because the reverse tax on any project is small, reducing any need for a major change in institutional practice and the inherent cultural risks associated with major institutional change. At the same time, this approach can help create a culture in which software reuse is enthusiastically adapted by all level of software engineers.

More Extensive Approaches

These ideas are, by no means, new. Barry Boehm [5] introduced the Win-Win approach, also known as *Theory W*, to software cost modeling and commercial off-the-shelf integration. His work subsumes the points made in this article. A more detailed discussion of incentives and disincentives to reuse can be found in [2].

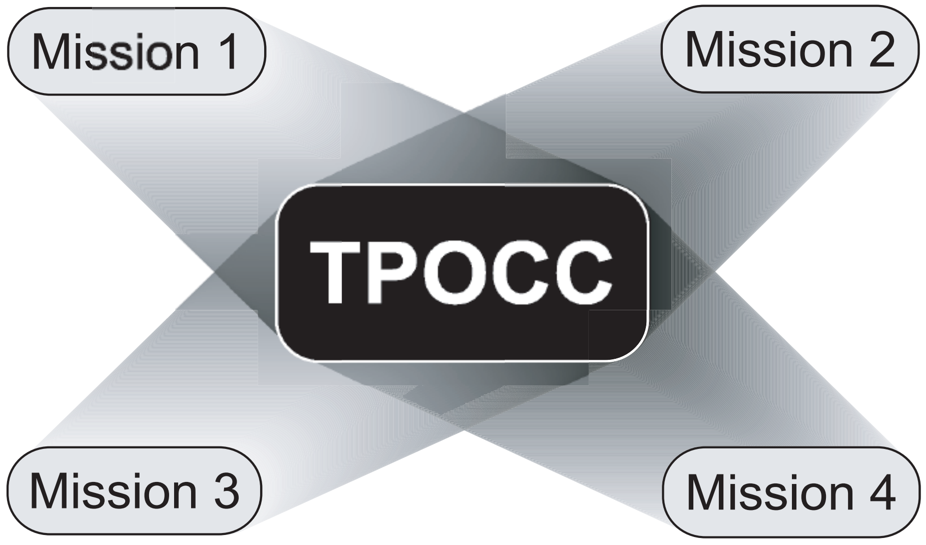

Figure 1: *Example of a Reusable Core of Spacecraft Control Software*

David Weiss and C. Lai [6] have written an important book on software product-line architectures. Much of their work is based on their experiences at Lucent Technologies and the resulting cost savings and quality improvement. An important follow-up paper [7] appeared in 1999. Note that the approach suggested here is much less formal than the complexity needed for the product-line architecture approach suggested by Weiss, Lai, and others.

Their work has been followed up by a series of publications on product-line architectures, including ones readily available from the Software Engineering Institute. Withey's report [8] is typical.

However, note that the ideas presented in this article have been used several places without the institutional reorganization needed to support a complete transition to a product-line architecture approach. For example, the author worked extensively on software for ground control of spacecraft at NASA Goddard Space Flight Center in Greenbelt, M.D., during a time of transition to a more reuse-based software development. The software team in what was then called the Control Center Systems Branch won a center-wide award for cost savings.

That branch was responsible for the ground systems that control the initial interface between a spacecraft and ground-based computer control centers. The control system software consists of large amounts of code organized into several subsystems to perform the following operations, among others:

- Determine the current position of the spacecraft.
- Control the operation of the spacecraft.
- Receive and relay telemetry information from the spacecraft.
- Detect significant events in spacecraft operation.
- Display the status of the system.

Space system software is extremely complex because it has severe requirements for fault tolerance, must interface with many other systems, and has some real-time requirements as well. An additional complexity is that the software must begin development far in advance of a projected launch of a spacecraft and therefore the level of technology of both hardware and support software (operating system, compilers, tools, commercial software, etc.) is not easy to determine during the beginning of development.

Reuse has been a concern for many years. However, the changing demands of spacecraft, the fluidity of graphics standards, the need for isolation from networks such as the Internet for security purposes, the long lead time for projects, and the need for severe restrictions on the weight of onboard computers all have made the development of a reuse program more difficult.

The initial step in any program of software reuse – domain analysis – was facilitated by a core group of talented domain experts, including both NASA employees and contractor personnel. The domain experts were already motivated by financial pressures and their desire to produce software in an efficient manner. They identified a reusable core of spacecraft control software (TPOCC in Figure 1) and mission-specific software.

Accounting practices were modified on individual projects to incorporate the reverse tax to ensure that the TPOCC reusable core was of exceptionally high quality. There was little resistance, because the amount of work was overwhelming and any method to improve product quality and efficiency was accepted readily.

Examination of internal software discrepancy reports (*bug reports*) showed that the TPOCC reusable software core had several orders of magnitude errors fewer than other systems and subsystems, suggesting that the reverse tax funds saved by not duplicating software development had been properly allocated to improve quality of the most heavily reused components.

Other Benefits

Most of the problems that Newcum listed can be addressed by the simple change in project accounting proposed here. The apparent complexity of software projects is reduced by standard interfaces between component software *parts*. Client/server designs are consistent with high quality software with well-defined interfaces. Schedules can be made more realistic for projects that reuse high quality code, since there will be fewer problems integrating error-prone software with poorly specified interfaces. It is easier to support good business functions by reusing software components that are known to work.

It is less obvious, but equally true, that encouraging reuse by providing incentives to improve quality can improve design and encourage the use of up-front prototypes. Having a list of proven software components with standard interfaces can make the development of prototypes much faster.◆

Acknowledgement

This research was partially supported by the National Science Foundation under grant number 0324818.

References

1. Newcum, Paul. “13 Pains in My Software! With Healthy Medications for Each.” Journal of Systems Management Nov./Dec. 1995: 28-31.
2. Leach, R.J. Software Reuse: Methods, Models, Costs. New York: McGraw-Hill, 1997.
3. Cohen, Sholom G., Jay L. Stanley Jr., A. Spencer Peterson, and Robert W. Krut Jr. “Application of Feature-Oriented Domain Analysis to the Army Movement Control Domain.” Pittsburgh, PA: Software Engineering Institute, June 1992.
4. Prieto-Diaz, R. “Domain Analysis: An Introduction.” Software Engineering Notes 15.2 (Apr. 1990): 47-54.
5. Boehm, B., P. Bose, E. Horowitz, and M.J. Lee. “Software Requirements Negotiation and Renegotiation Aids: A Theory-W Based Spiral Approach.” International Conference on Software Engineering, Seattle, WA, Apr. 23-30, 1995.
6. Weiss, D.M., and C. Lai. Software Product Line Engineering. Addison Wesley Longman, New York, 1998.
7. Coplien, J., D. Hoffman, and D. Weiss. “Commonality and Variability in Software Engineering.” IEEE Software Nov./Dec. 1999: 37-45.
8. Withey, J. “Investment Analysis of Software Assets for Product Lines.” Pittsburgh, PA: Software Engineering Institute, Nov. 1996.

About the Author

Ronald J. Leach, Ph.D., is professor and chair of the Department of Systems and Computer Science at Howard University. Leach has had grants and contracts from many government agencies and companies and has given lectures on three continents. He does research in software engineering, with special interest in reuse, metrics, fault tolerance, performance modeling, process improvement, and the efficient development of complex software systems. He is the author of five books and more than 65 published technical articles. Leach has a Bachelor of Science, Master of Science, and doctorate degree in mathematics from Maryland University, and a Master of Science in computer science from Johns Hopkins University.

Department of Systems and Computer Science
School of Engineering
Howard University
Washington, D.C. 20059
Phone: (202) 806-6650
Fax: (202) 806-4531
E-mail: rjl@scs.howard.edu