Managing Software Productivity and Reuse

Barry Boehm, University of Southern California

Your organization can choose from three main strategies for improving its software productivity. You can work faster, using tools that automate or speed up previously labor-intensive tasks. You can work smarter, primarily through process improvements that avoid or reduce non-value-adding tasks. Or you can avoid unnecessary work by reusing software artifacts instead of custom developing each project. Which strategy will produce the highest payoff?

LEADING INCREASED PRODUCTIVITY STRATEGY

I performed an extensive analysis (B.W. Boehm, "Economic Analysis of Software Technology Investments," in Analytical Methods in Software Engineering Economics, Thomas Gulledge and William Hutzler, eds., Springer-Verlag, 1993) that addressed this question for the US Department of Defense. This analysis factored in existing labor distributions by phase and activity, normal commercial trends, technology capability trends, and technology transition delays. The results showed that a proactive DoD strategy could achieve the following percentage savings over and above the normal improvements accrued via a business-as-usual approach:

- Working faster: 8 percent
- Working smarter: 17 percent
- Work avoidance: 47 percent

The analysis also concluded that all three strategies were worth pursuing in concert, as their benefits were largely complementary. For this column, however, I'll focus on reuse because it offers the biggest potential payoffs. Before launching into a major reuse program, however, you should know that several potential pitfalls await the unwary.

PITFALLS

Some of the most frequent pitfalls people encounter when trying to achieve reuse savings include the following:

Field of Dreams. This pitfall derives from the mistaken belief that if you "build a repository of components, reusers will come." An early DoD Reuse 2000 initiative, whose main goal was simply to accumulate 2,000 reuse candidates as quickly as possible, found that the program's components languished virtually unused thanks to nontechnical factors such as risk aversion and not-invented-here (NIH) reactions

Component versus Interface Focus. The Common Ada Missile Packages program addressed risk and NIH effects via a community effort, but the components developed for it proved insufficiently reusable because the program lacked a domain architecture with appropriate interface specifications for reusable components.

Overgeneralization. The National Library of Medicine MEDLARS II publication system was built with many layers of abstraction to support a wide range of future publication systems. The system was scrapped when, after two expensive hardware upgrades, it still proved incapable of processing the MEDLARS II workload.

Scalability. Reuse via very high level languages (VHLLs), while effective for many small systems, does not scale up well. The New Jersey Department of Motor Vehicles auto registration system was developed using the IDEAL VHLL; the system performed so poorly that eventually more than a million New Jersey cars roamed the streets without license renewals.

Technical Obsolescence. In the 1970s and early 1980s, TRW won many digital processing competitions with a formidable domain architecture and set of reusable components based on Digital's VAX processors with attached vector-processor boxes. By the mid-1980s, though, this architecture lost out to more powerful approaches that involved distributed processing and custom ASICs.

The "Critical Reuse Success Factors and Resources" sidebar provides tips and pointers to sources that can help you avoid these pitfalls and make reuse successful in your organization.

STATISTICS SHOW REUSE WORKS

When I advocated investments in software technology at DARPA, I felt continually frustrated by the hardware guys' ability to show curves indicating exponential growth in the DoD's number of transistors owned or number of Internet packets handled, while the counterpart software curves continued to show a relatively flat 8-10 delivered source instructions per person-day.

In self-defense, and with the help of Tom Frazier's cost analysis group at IDA, we came up with a set of curves that counted executable machine instructions of DoD software as lines of code in service. LOCS let us count software much the same way the hardware guys counted DoD transistors: by adding up the average number on each ship, airplane, workstation, and so on used by the DoD, then multiplying by the number of ships, airplanes, workstations, and so on owned by the DoD.

Figure 1 shows the resulting trends in LOCS of DoD software and the DoD cost in dollars per LOCS between 1950 and 2000. I've conservatively estimated the figures for 2000 by multiplying 2 million DoD computers by 100 million executable machine instructions per computer, which gives 200 trillion LOCS. Based on a conservative $40

billion-per-year DoD software cost, the cost per LOCS is $0.0002. These cost improvements come largely from software reuse.

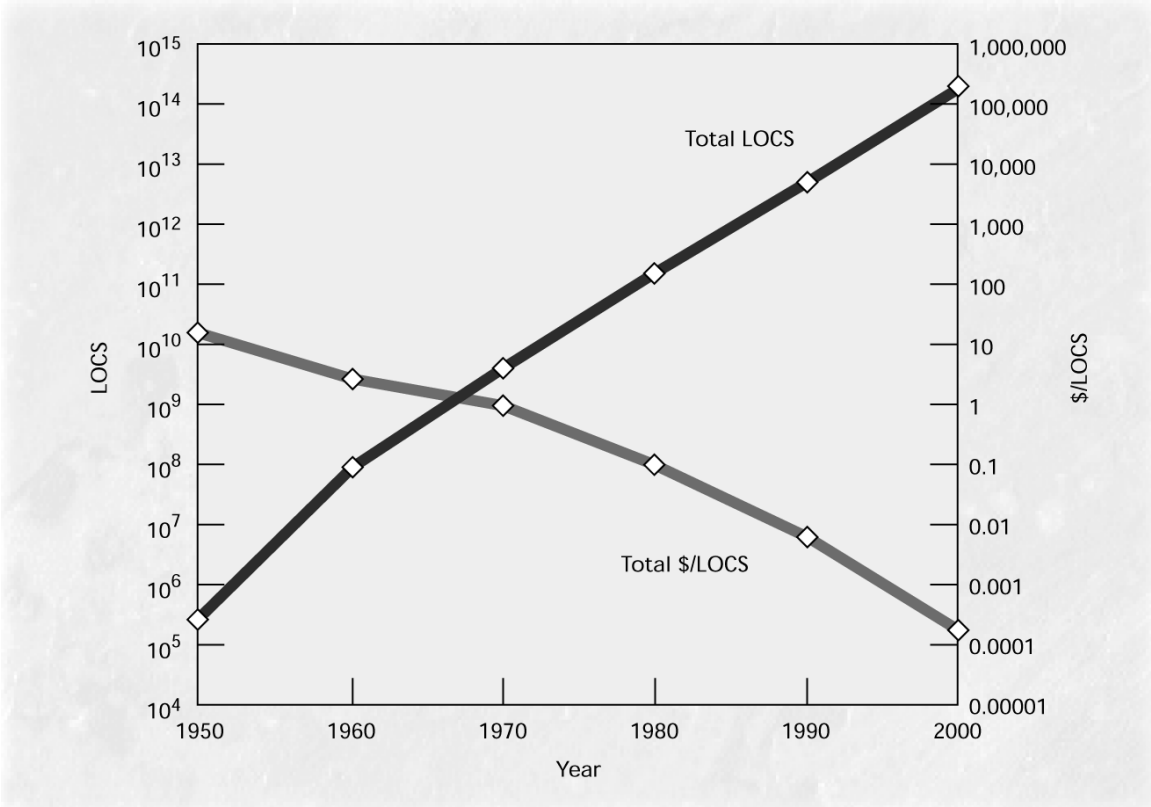

Figure 1. Thanks to reuse, while the DoD's lines of code in service have skyrocketed over the past 50 years, the cost per LOCS has plummeted.

You might object that not all these LOCS add value for their customers. But you could raise the same objection for all the transistors being added to chips each year and all the data packets transmitted across the Internet. All three commodities pass similar market tests.

Figure 2, from a chart by Lawrence Bernstein ("Software Investment Strategy," Bell Labs Technical Journal, Summer 1997, pp. 233-242), shows some corroborative trends of the software expansion factor: the ratio of machine lines of code to a source line of code. Bernstein's historical data shows an order-of-magnitude increase every 20 years, with the most significant recent gains coming from software reuse. The LOCS curve in Figure 1 reflects a combination of productivity trends and trends in the numbers of copies of software products being used.

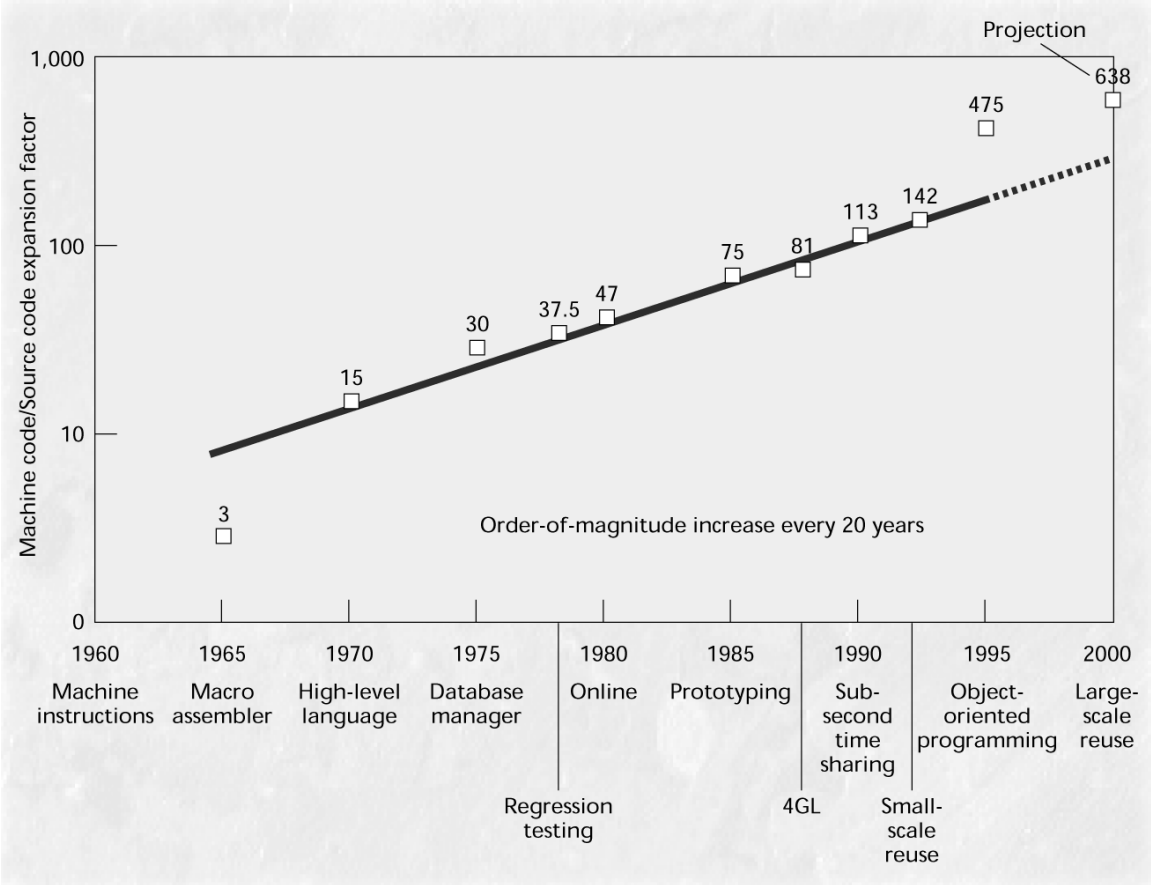

Figure 2. Although several trends have increased the ratio of machine lines of code to source lines of code, reuse caused the most significant increase.

Even with the significant productivity gains shown in Figures 1 and 2, we need to do much more in both the technical and management domains to fully capitalize on software reuse. These efforts should focus on stronger technical foundations for software architectures and component composition; more change-adaptive components, connectors, and architectures; creating more effective reuse incentive structures; domain-engineering and business-case analysis techniques; better techniques for dealing with commercial off-the-shelf software integration; and creating appropriate mechanisms for dealing with liability issues, intellectual property rights, and software artifacts as capital assets. A 47 percent productivity boost is just the beginning; further progress will bring even greater gains.

Barry Boehm is director of the USC Center for Software Engineering. He developed the Constructive Cost Model (COCOMO), the software process Spiral Model, and the Theory W (win-win) approach to software management and requirements determination. Contact him at boehm@sunset.usc.edu.

Critical Reuse Success Factors and Resources

The eight critical success factors that follow include references to several valuable sources for both the management and technical aspects of reuse.

- *Adopt a product line approach.* This approach involves determining the right product lines for your organization, developing domain-specific software architectures for your product line, and developing product-line solutions. The CMU Software Engineering Institute's Product Line Practices Web site (http://www.sei.cmu.edu/activities/plp/plp_init.html) contains a wealth of useful guidelines for doing this.
- *Perform a business case analysis to determine the right scope and level of expectation for your product line.* Donald J. Reifer's Practical Software Reuse (John Wiley & Sons, 1997) and Wayne C. Lim's Managing Software Reuse (Prentice Hall, 1998) have excellent treatments of product-line business-case analysis.
- *Focus on achieving black-box reuse.* Once you open up and modify reusable component, you incur number of added costs, which can compromise your business case. Jeffrey S. Poulin's Measuring Software Reuse (Addison Wesley Longman, 1997) and my own forthcoming multiple-author book, Estimating Software Costs with COCOMO II (Prentice Hall, 2000), include helpful data and models for reasoning about black-box versus white-box reuse.
- *Establish an empowered product line manager and stakeholder buy-in.* This is the most critical success factor of all. Without a manager empowered and accountable for making product line investments and ensuring that the reusable artifacts get used, and without buy-in from the various asset producers, purveyors, and users, no amount of technology will make much difference. *Software Reuse*, by Ivar Jacobson, Martin L. Griss, and Patrik Jonsson (Addison Wesley Longman, 1997), offers excellent case studies of Ericsson's and Hewlett-Packard's experiences in this regard.
- *Establish reuse-oriented processes and organizations.* For example, serious "model clashes" occur when trying to reuse assets within a requirements-first waterfall process model. If you lock in on a one-second response time requirement, and none of your reusable components—such as commercial off-the-shelf database management systems—can process your workload in less than two seconds, you have a very expensive custom component to develop. You are also likely to need new organizational entities for such functions as reusable asset certification, version and configuration control, repository management, and adaptation to change. *See Practical Software Reuse* and "Process Support of Software Product Lines," (*Proceedings ISPW-10*, Barry Boehm, Marc Kellner, and Dewayne Perry, eds., IEEE Computer Soc. Press, 1998) for further information on these issues.
- *Adopt an incremental approach, employing carefully chosen pilot projects and real-world feedback.* The Hewlett Packard incremental approach in Software Reuse offers a particularly good example of this approach.

- *Use metrics-based reuse operations management.* Your reuse business case and incremental plan provide a good framework for tracking progress with respect to expectations and making appropriate adjustments where necessary. Lim's and Poulin's books are particularly strong on reuse metrics and their use in management.
- *Establish a proactive product-line evolution strategy.* This is your guard against the technical obsolescence pitfall. Your product line will be affected by rapidly moving technologies such as CORBA, DCOM, the Web, and Java infrastructures. Unless you invest in monitoring, experimenting with, and adapting to such trends, you risk obsolescence.