Next-Generation Software Economics

by **Walker Royce**
Vice President and General Manager
Strategic Services
Rational Software

Software engineering is dominated by intellectual activities focused on solving problems with immense complexity and numerous unknowns in competing perspectives. The early software approaches of the 1960s and 1970s can best be described as craftsmanship, with each project using a custom process and custom tools. In the 1980s and 1990s, the software industry matured and transitioned to more of an engineering discipline. However, most software projects in this era were still primarily research-intensive, dominated by human creativity and diseconomies of scale. The next generation of software processes is driving toward a more production-intensive approach dominated by automation and economies of scale. Next-generation software economics are already being achieved by some advanced software organizations. Many of the techniques, processes, and methods described in a modern process framework have been practiced for several years. However, a mature, modern process is nowhere near the state of the practice for the average software organization.

This article introduces several provocative hypotheses about the future of software economics. In 1987 Barry Boehm published a one-page description of the "Industrial Software Metrics Top 10 List." This assessment remains a good objective framework for discussing the predominant economic trends of software development. Although many of the metrics are gross generalizations, they accurately describe some of the fundamental economic relationships that resulted from the conventional software process practiced over the past 30 years. Quotations from Boehm's list are presented in italics in the paragraphs below. After each quotation I summarize some of the important economic results of the conventional process and speculate on how a modern

software management framework, or iterative process like the Rational Unified Process (RUP), can improve these performance benchmarks. Although there are not enough project data to validate my assertions, I believe that these expected changes provide a good description of trends a software organization should look for in making the transition to a modern process.

1. *Finding and fixing a software problem after delivery costs 100 times more than finding and fixing the problem in early design phases.*

   This metric dominates the rationale for most software process improvement. Modern processes, component-based development technologies, and architecture frameworks are explicitly targeted at improving this relationship. An architecture-first approach will likely yield ten-fold to hundred-fold improvements in the resolution of architectural errors. Consequently, a modern process places a huge premium on early architecture insight and risk-confronting activities.

2. *You can compress software development schedules 25% of nominal, but no more.*

   An N% reduction in schedule would require an M% increase in personnel resources (assuming that other parameters remain fixed). Any increase in people requires more management overhead. In general, the limit of flexibility in this overhead, along with scheduling concurrent activities, conserving sequential activities, and other resource constraints, is about 25%. This metric should remain valid for the engineering stage of the life cycle, where the intellectual content of the system is evolved. However, if the engineering stage is successful at achieving a consistent baseline, including architecture, construction plans, and feature scope, schedule compression in the production stage should be more flexible. Whether a line-of-business organization is amortizing the engineering stage across multiple projects, or a project organization is amortizing the engineering stage across multiple increments, there should be much more opportunity for concurrent development.

3. *For every $1 you spend on development, you will spend $2 on maintenance.*

   Anyone working in the software industry over the past 10 to 20 years knows that most of the software in operation is considered to be difficult to maintain. A better way to measure this ratio would be the productivity rates between development and maintenance. One interesting aspect of iterative development is that the line between development and maintenance has become much fuzzier. There is no doubt that a mature iterative process and a good architecture can reduce scrap and rework levels considerably. Given the overall homogenization of development and maintenance activities, my guess is that this metric should change to a one-for-one

relationship, where development productivity will be similar to maintenance productivity.

4. *Software development and maintenance costs are primarily a function of the number of source lines of code.*

   This metric is primarily due to the predominance of custom software development, lack of commercial components, and lack of reuse inherent in the era of the conventional process, in which the size of the product was the primary cost driver and the fundamental unit of size was a line of code. Commercial components, reuse, and automatic code generators can seriously pollute the meaning of a source line of code. Construction costs will still be driven by the complexity of the bill of materials. More components, more types of components, more sources of components, and more custom components will result in more integration labor and drive up costs. Fewer components, fewer types, fewer sources, and more industrial-strength tooling will drive down costs. Next-generation cost models should become less sensitive to the number of source lines and more sensitive to the discrete numbers of components and their ease of integration.

5. *Variations among people account for the biggest differences in software productivity.*

   

   This is certainly a key piece of conventional wisdom: Hire good people. This metric is also a subject of overhype and underhype. When you don't know objectively why you succeeded or failed, the obvious scapegoat is the quality of the people. It is subjective and difficult to challenge. In any engineering venture where intellectual property is the real product, the dominant productivity factors will be personnel skills, teamwork, and motivations. To the extent possible, a modern process emphasizes the need for high-leverage people in the engineering stage, when the team is relatively small.

6. *The overall ratio of software to hardware costs is still growing. In 1955 it was 15:85; in 1985, 85:15.*

   The need for software, its breadth of applications, and its complexity continue to grow almost without limits. The main impact of this metric on software economics is that hardware continues to get cheaper. Processing cycles, storage, and network bandwidth continue to offer new opportunities for automation. Consequently, software environments are playing a much more important role. From a modern process perspective, I can see the environment doing much more of the bookkeeping and analysis activities that were previously done by humans. Configuration control and quality

assurance analyses are already largely automated, and the next frontier is probably significant improvements in automated production and automated testing.

7. *Only about 15% of software development effort is devoted to programming.*

   The amount of programming that goes on in a software development project is probably still roughly 15%. The difference is that modern projects are programming at a much higher level of abstraction. An average staff-month of programming produced maybe 200 machine instructions in the 1960s and 1000 machine instructions in the 1970s and 1980s. Programmer productivity in the 1990s can produce tens of thousands of machine instructions in a single month, even though only a few hundred human-generated source lines may be produced.

8. *Software systems and products typically cost three times as much per instruction as individual software programs. Software-system products (i.e., system of systems) cost nine times as much.*

   This exponential relationship is the essence of what is called "diseconomy of scale." Unlike other commodities, the more software you build, the more expensive per unit item it is. This diseconomy of scale should be greatly relieved with a modern process and modern technologies. Under certain circumstances, such as a software line of business producing discrete, customer-specific software systems with a common architecture, common environment, and common process, an economy of scale should be achievable.

9. *Walkthroughs catch 60% of the errors.*

   This may be true, but walkthroughs are not catching the errors that matter. All defects are not created equal. In general, walkthroughs and other forms of human inspection are good at catching surface issues and style issues; few humans are good at reviewing even first-order semantic issues in a code or model segment. How many programmers get their code to compile the first time? Human inspections and walkthroughs will not expose the significant issues (resource contention, performance bottlenecks, control conflicts, and the like); they will only help resolve them. While the environment catches most of the first-level inconsistencies and errors, the really important architectural issues can only be exposed through demonstration and early testing and resolved through human scrutiny.

10. *80% of the contribution comes from 20% of the contributors.*

This is true across almost any engineering discipline. These are the fundamental postulates that underlie the rationale for a modern software management process framework. 80% of the engineering is consumed by 20% of the requirements. 80% of the software cost is consumed by 20% of the components. 80% of software scrap and rework is caused by 20% of the errors. 80% of the resources are consumed by 20% of the components. 80% of the progress is made by 20% of the people. These relationships are timeless and constitute the background philosophy to be applied throughout the planning and conduct of a modern software management process.

As this discussion illustrates, the next generation of software processes, and specifically the techniques presented in the Rational Unified Process, can help software organizations achieve unprecedented economic advantages. As we head into an era that leverages a more production-intensive approach for software development, dominated by commercial components and automation, we can start to achieve economies of scale.

References

Boehm, B.W. "Industrial Software Metrics Top 10 List," IEEE Software 4, 5 (September 1987), pp. 84-85.

Royce, W. E. *Software Project Management: A Unified Framework*. Addison-Wesley Longman, 1998.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!