Familiar Metric Management - Software Reuse

Lawrence H. Putnam
Ware Myers

"While reuse has long been more of a hope than a promise, the greatest opportunity for dramatic productivity and quality improvement will be through improved ways to build new software on the progressively richer foundation of previously produced products." Watts S. Humphrey [1]

In the late 1980s when Humphrey was writing, reuse was still largely a hope, though even then some companies were making progress. It was still a hope for most companies in 1996 when Ed Yourdon estimated that the "passive" approach "typically hovers at the 15 to 20 percent level." [2] By "passive" he meant a developer reusing some of his or her own code or the code of nearby coworkers. There was no "active" reuse program. It was still a hope in 1996 when Paul Bassett estimated that five sixths of companies fell on what he called Organizational Reuse Maturity Level 1, Ad Hoc. [3] That level means that typical reuse is in the 0 to 40 percent range.

The promise is beginning to appear here and there. For instance, in 1992 Malcolm Rix reported to the Workshop on Software Reuse that Hewlett Packard had achieved 60 percent reuse on one instrument line. In 1996 Bassett wrote that the most recent QSM study of companies employing his frame technology methods found an average productivity index of 26.9, up from the 26.2 reported by the 1994 study. (Average of all business systems in the QSM database is 16.9.) Included in this study was a project at Automated Financial Systems Inc. with a productivity index of 33.1. That is one of the two or three highest ever recorded by QSM. The company made this record on a 635,363 SLOC check-clearing system which achieved 91 percent reuse. We could cite more examples. There are enough companies achieving reuse at these levels to establish that it can be done.

Peering into the future

It is becoming more essential that reuse be done, too. Many companies are reengineering their business processes. Reengineering a single process is ordinarily a task of some months -- or it would be if the supporting software could be developed on that time scale. There is little advantage to be gained from reengineering a process in months, if the software to implement it takes years. Automated Financial Systems completed its project in five months. Done in the usual fashion, with industry-average productivity, it would have taken 51 months.

On both measures, percent-of-reuse and time-to-completion, this company achieved a factor-of-10 gain, or so it seems. Wait a minute! Where did that reusable software come from? How much did it cost to build it? How long did it take? When we take questions like these into consideration, we see that reuse is not free. Of course, this being a metrics newsletter, you will not be surprised to find that it takes metrics to answer these questions. First, we have to inquire a little bit into what comes before the actual act of reuse.

A company has to stake out a domain-a potential set of application systems-for which reuse looks promising. It has to lay out the beginning of the architecture that will characterize these systems, as well as systems that are yet to come. It has to sort out of this architecture subsystems that will be common, or nearly so, to most of these application systems. The term, component systems, seems to be coming into use to describe these reusable subsystems.

Then it has to develop these component systems. One aspect of this development is to design in points of variation, so that one component system fits into many slightly different applications. It has to warehouse these component systems in a repository from which they can be obtained by the developers of each application system.

When all of this is accomplished skillfully, the application developers can indeed achieve 91 percent reuse in five months. It may not have escaped your notice, however, that domain analysis, architecture, component development, and repository support cost money and take time. Indeed, experience suggests that it may take two or three years of investment in these processes before the gains from reuse begin to pay off, as the figure illustrates. It is possible to shorten this investment period if you can buy suitable component systems on the market, instead of developing them in house. In that case, they still cost money. Moreover, they take time and money to learn to use skillfully.

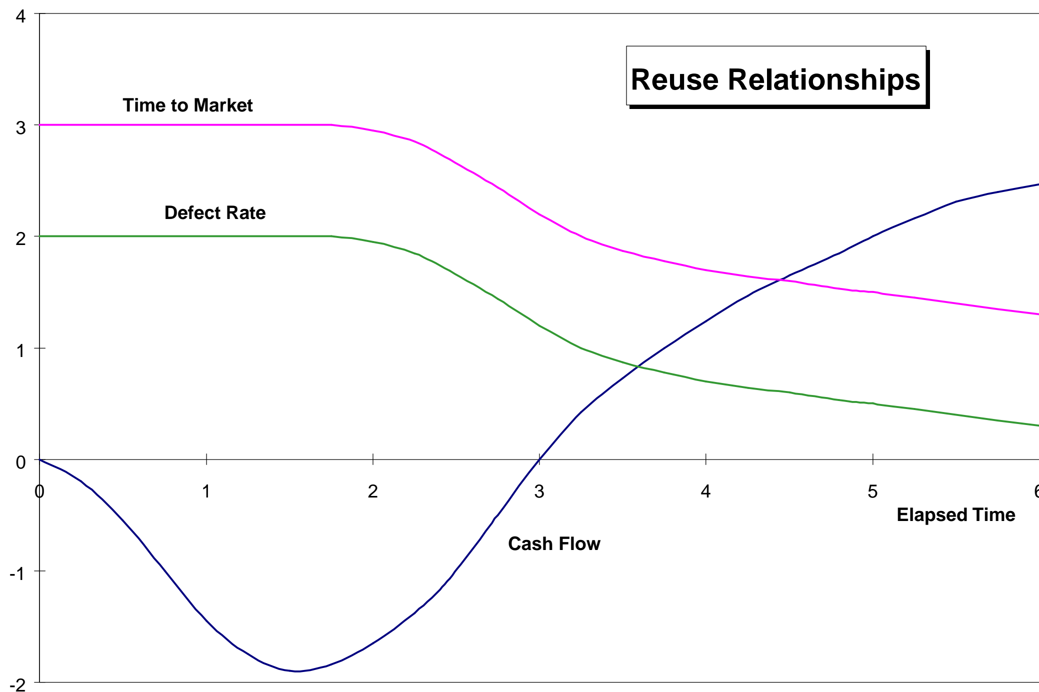

Figure1. Investment is small during an initial period while a planning team is figuring out the reuse program. Costs grow when domain analysis and architectural work get under way and increase still more when component groups begin to function. After several years the gains from reusing component systems begin to offset the startup costs. As reusable components come into use, time to reach market declines. The defect rate follows a similar curve.

Peering into the unfamiliar

Up to now the software industry has developed software for the most part in one-of-a-kind projects. The management numbers—size, effort, development time, defect rate, and process productivity—applied to a single project. Under conditions of reuse, software development will be spread over domain analysis, architecture planning, component systems development, and component support, before it gets to the actual project that develops an application system. Will this require a change in metrics?

Let's start at the back end. The development of application systems is comparable to present project-based development-with some differences. In fact, there are two main differences. The first is that part of the requirements analysis and architecture planning has already been taken care of at the domain level. The application developers still have to specialize the requirements and the architecture to their particular situation. The second difference is that the component system developers have provided a selection of reusable components. The application developers still have to perceive the need for a component system, find it, understand how it works, and in many cases specialize it (set the variation mechanisms) for their application.

Still, from a measurement perspective, the application development organization does turn out a system. The code or the function points can be counted. The time and effort can be measured. The process productivity index can be computed. When we do this for a project such as the check-clearing system, what we will call the "total" productivity index turns out to be very high, in this case, 33.1. We can regard it as a reflection of the value of reuse.

The developers on this project, however, actually produced only 56,592 SLOC. The rest of the 635,363 SLOC was reused. Giving them credit only for the code they actually produced reduces their productivity index to 22.9. That is still way above average. They were doing a good job, but it was not up in the stratosphere. Let's call it the "working" productivity index. We can use it to estimate time, effort, and defects for the new part of the project. We can use these estimates to control the progress of the application project.

Let's move upstream now to the development of a component system. The efforts of these developers show up after measurable time and effort in code that can be counted. Therefore, we can compute their process productivity. There is a Catch 22. It is harder to develop a component system than an application system. The component system has to be capable of variation (so it can fit in widely). It has to be of high reliability (so that application developers will trust it).

That leads to the current rule of thumb: it takes about three times as much time and effort to develop a reusable component system as a comparable non-reusable system at the application level. That implies that the productivity index of a component-system project is going to be considerably lower than that of a comparable application system. (It also implies that a component system has to be used three times before it has amortized its cost.) So, it won't be fair for management to hold component developers to the same productivity standard as application developers. This relatively low productivity index will, however, be an appropriate guide to the planning and control of component system development itself.

1. Watts S. Humphrey, *Managing the Software Process*, Addison-Wesley, Reading, MA, 1989.
2. Edward Yourdon, *Rise & Resurrection of the American Programmer*, Prentice Hall, Upper Saddle River, NJ, 1996
3. Paul G. Bassett, *Framing Software Reuse: Lessons from the Real World*, Prentice Hall, Upper Saddle River, NJ, 1996.