

To Improve Your Process: Keep It Simple

Arguments about whether software development is exactly like manufacturing miss the point. The concepts of quality used in manufacturing, like quality Six Sigma, can also be used effectively in software development, as this case study illustrates. The author, well-known in quality-improvement circles, has written several best-selling books on Six Sigma and other quality related subjects. A self-professed quality fanatic, he is a Fellow of the American Society for Quality and founder of Quality America, a Tucson, Arizona, company that develops quality-improvement software.

-Dave Card

The question that haunts all developers and their manager is "How can we assure the quality of our code?" Unfortunately, there is no simple way to do it. But neither is it an impossible task. In fact, it can be done with reasonable effort.

That is the good news. The bad news is I can't tell you exactly how to do it. You see, there is no "right" way to improve quality. Every organization must come up with an approach that works for them. In fact, the process of reaching a consensus on how to improve quality is part of the solution. A "solution" imposed from the outside is by definition *not* the answer.

So I can't hand you the answer on a silver platter. But I can give you some pointers, based on how we improved code quality at Quality America. As our name implies, quality is our business – one of our major products is quality-improvement software. But we still had to grow our own approach to quality improvement, an approach we are still refining.

Quality Investment. With annual sales-per-employee of over \$150,000 and rising, my employees are very productive. Is this because I have been unusually fortunate in hiring the world's best employees? Perhaps. I am certainly proud of them. But it is more likely that I have discovered something other successful companies around the world already know: Investing in people pays off. That's why I devote more than 12 percent of my payroll budget to training.

New employees are taught that Quality America is a collection of processes and that they will be responsible for some of

these processes. They learn about measuring and monitoring and how to use flowcharts and cause-and-effect diagrams, among other tools. Teamwork and group dynamics are also part of the training.

Programmers are given the same training. They learn about existing development procedures – and they learn that any process can be improved. Programmers are responsible for their own quality. In fact, we stress *only* quality. Productivity and quality are related, but sending a mixed message like "productivity with quality" is confusing.

QUALITY HOUR. In the drive to improve quality, the first thing I did was to mandate that all employees devote the last hour of each work day to quality improvement (I partially exempted those employees with direct customer contact). This mandate had a dramatic and positive effect on morale, convincing everyone that I was serious. During the first six to eight weeks, "quality hour," as it came to be called, was pretty much entirely devoted to training in the tools and philosophy of quality.

Then we got down to specifics. For the software division, I assembled a team of programmers and technical support staff to lead the effort. It was interesting to observe that the other divisions had a much easier time defining goals and making plans than did the software group, which had no widely accepted model to build on.

However, they did realize that the traditional approach to software development – code and test – simply cannot guarantee quality. Even when (costly) testing procedures catch errors before delivery, errors are expensive to fix. The least expensive problem is the problem that never existed in the first place! The goal is to predict the quality of the code *before* it is written. This was truly a challenge, but the team had nothing better to do during the quality hour, so they persisted.

PROCESS FOCUS. The team realized that quality improvement must focus on process, not product. It begins with a definition of just what constitutes the development process. To be sure, there are creative elements that can never be definitely identified. But a great deal can be.

They began by completely analyzing the development process and streamlining procedures. In doing so, they followed the KISS principle – keep it simple, stupid!

They determined that two factors were overwhelmingly important in predicting code quality: size and complexity. They studied sample modules until they were able to identify complexity by reviewing actual code, then they wrote pseudocode for each module and learned how to relate the pseudocode to the actual code.

They revisited the way design specifications were derived from customer wants and how design specifications were converted into product specifications. Finally, they set up a system of team reviews that required programmers to submit program plans before they began coding.

All these procedures were developed by teams of people who actually do the work. Every procedure was agreed to by every member, after sometimes intense discussion. As a veteran writer of “traditional” quality procedures, I was struck by how differently programmers develop procedures. Instead of paperwork and testing, the focus was on the development process, including documenting the process, programming style, and team reviews.

BLOBS. An example of how the team kept things simple is their innovative approach to investigating code reliability. The team wanted to be able to identify bad code before it was written. So they set about the complicated task of defining “good” code and “bad” code and the key

factors in the development process that lead to good and bad code.

First, they analyzed about 800 discrete modules from a product, written in C, that had been on the market for several years. They used an analysis program called PC Metrics to compute 14 metrics (like complexity, size, effort, and expected error rates) for each module. Then they correlated the metrics to the time it took to write the module, the number of maintenance actions required, and the number of errors.

Analyzing these metrics was difficult. They were often highly correlated with each other, and it was hard to find consistency in metrics that measured related aspects of quality. They couldn’t answer the question “is the quality of this code good?” – the patterns weren’t evident.

Next they applied a sophisticated technique called principal components analysis. This narrowed their focus to five control metrics, which they tracked on a multivariate control chart using SPC-PC II software. This helped them identify key development factors, but they felt they could learn even more if metrics analysis was simpler.

The team decided to convert the metrics into meaningful pictures that anyone could easily understand. Using the analysis program Systat, they converted each module into a single visual icon, which they called a blob. Figure 1 shows a set of blobs from an analyzed program. The more uniform the circle, the better the code. In this case, it’s immediately apparent that the Opensws and Textsize functions need improvement.

The patterns revealed by the blobs quickly identify potential problem areas long before the software reaches testing. Now developers could tell if the

module was likely to be troublesome simply by looking at it.

The next step was to determine what made some modules worse than others. One way we did this was to play a pattern-matching game. We took printouts of the blobs, spread them out on a huge conference table, then grouped them according to shape. The resulting affinity diagram showed that most troublesome functions produced distinct icons, which accurately predicted problems. This exercise also revealed that some kinds of statements routinely create bad code, even when written by expert programmers.

RESULTS. Did our quality-improvement program work? If the measure of success is the number of errors, the answer is yes. Within nine months, we started to see improved code. The error count per 100,000 lines of code dropped by a factor of 10. If the measure of success is the time required to deliver a new product, yes again. The number of man-months per 100,000 lines of new code dropped by a factor of nearly three. In fact, by nearly every measure the answer is yes.

In the end, the procedures we developed make testing, which we still perform, more a validation tool than a quality tool. Still, it is never enough. Quality involves continuous improvement. We’ve made our customers happy. Great. Now let’s make them ecstatic!

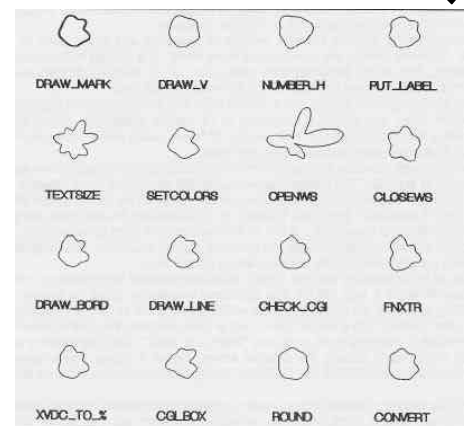


Figure 1–Blobs

