Steve McConnell

Books
Articles
Interviews
Presentations
About Me
Contact Me

Software Development, **August 1996**

Software Quality at Top Speed

Software products exhibit two general kinds of quality, which affect software schedules in different ways. The first kind of quality that people usually think of when they refer to "software quality" is low defect rate.

Some project managers try to shorten their schedules by reducing the time spent on quality-assurance practices such as design and code reviews. Some shortchange the upstream activities of requirements analysis and design. Others--running late--try to make up time by compressing the testing schedule, which is vulnerable to reduction since it's the critical-path item at the end of the schedule.

These are some of the worst decisions a person who wants to maximize development speed can make. In software, higher quality (in the form of lower defect rates) and reduced development time go hand in hand. Figure 1 illustrates the relationship between defect rate and development time.

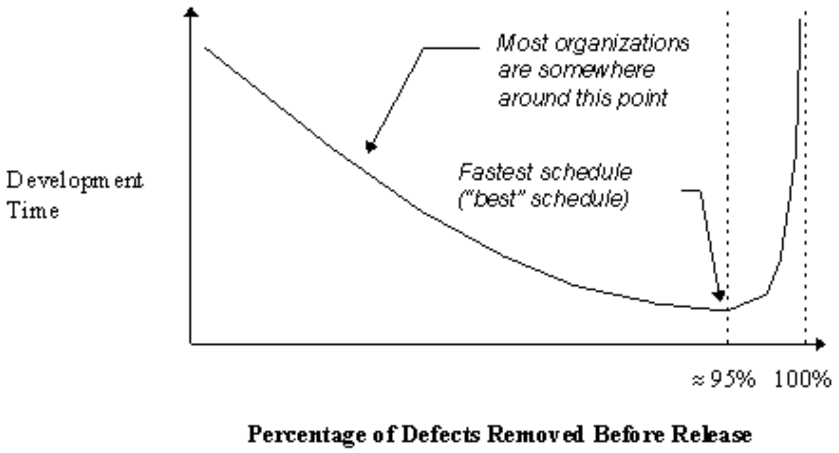

Figure 1. Relationship between defect rate and development time. As a rule, the projects that achieve the lowest defect rates also achieve the shortest schedules.

A few organizations have achieved extremely low defect rates (shown on the far right of the curve), and when you reach that point, further reducing the number of defects will tend to increase the amount of development time. This applies to life-critical systems such as the life-support systems on the space shuttle. It doesn't apply to the rest of us.

The rest of us would do well to learn from a discovery made by IBM in the 1970s: Products with the lowest defect counts also have the shortest schedules (Jones 1991). Many organizations currently develop software with defect levels that give them longer schedules than necessary. After surveying about 4000 software projects, Capers Jones reported that poor quality was one of the most common reasons for schedule overruns (1994). He also reported that poor quality is implicated in close to half of all canceled projects. A Software Engineering Institute survey found that more than 60 percent of organizations assessed suffered from inadequate quality assurance (Kitson and Masters 1993). On the curve in Figure 1, those organizations are to the left of the 95-percent-removal line.

That 95-percent-removal line--or some point in its neighborhood--is significant because that level of pre-release defect removal appears to be the point at which projects achieve the shortest schedules, least effort, and highest levels of user satisfaction (Jones 1991). If you're finding more than 5 percent of your defects after your product has been released, you're vulnerable to the problems associated with low quality, and you're probably taking longer to develop your software than you need to.

Design Shortcuts

Projects that are in a hurry are particularly vulnerable to shortchanging quality-assurance at the individual-developer level. Any developer who has been pushed to ship a product quickly knows how much pressure there can be to cut corners because "we're only three weeks from shipping." For example, rather than writing a separate, completely clean printing module, you might piggyback printing onto the screen-display module. You know that's a bad design, that it isn't extendible or maintainable, but you don't have time to do it right. You're being pressured to get the product done, so you feel compelled to take the shortcut.

Two months later, the product still hasn't shipped, and those cut corners come back to haunt you. You find that users are unhappy with printing, and the only way to satisfy their requests is to significantly extend the printing functionality. Unfortunately, in the two months since you piggybacked printing onto the screen-display module, the printing functionality and the screen-display functionality have become thoroughly intertwined. Redesigning printing and separating it from the screen display is now a tough, time-consuming, error-prone operation.

The upshot is that a shortcut that was supposed to save time actually wasted time in the following ways:

- The original time spent designing and implementing the printing hack was completely wasted because most of that code will be thrown away. The time spent unit-testing and debugging the printing-hack code was also wasted.
- Additional time must be spent to strip the printing-specific code out of the display module.
- Additional testing and debugging time must be spent to ensure that the modified display code still works after the printing code has been stripped out.
- The new printing module, which should have been designed as an integral part of the system, has to be designed onto and around the existing system, which was not designed with it in mind.

All this happens, when the only necessary cost--if the right decision had been made at the right time--was to design and implement one version of the printing module. And now you still have to do that anyway.

This example is not uncommon. Up to four times the normal number of defects are reported for released products that were developed under excessive schedule pressure.2 Projects that are in schedule trouble often become obsessed with working harder rather than working smarter. Attention to quality is seen as a luxury. The result is that projects often work dumber, which gets them into even deeper schedule trouble.

Error-Prone Modules

One aspect of quality assurance that's particularly important to rapid development is the existence of error-prone modules, which are modules that are responsible for a disproportionate number of defects. Barry Boehm reported that 20 percent of the modules in a program are typically responsible for 80 percent of the errors.5 On its IMS project, IBM found that 57 percent of the errors clumped into 7 percent of the modules.1

Modules with such high defect rates are more expensive and time-consuming to deliver than less error-prone modules. Normal modules cost about $500 to $1000 per function point to develop. Error-prone modules cost about $2000 to $4000 per function point to develop.2 Error-prone modules tend to be more complex than other modules in the system, less structured, and unusually large. They often were developed under excessive schedule pressure and were not fully tested.

If development speed is important, make identification and redesign of error-prone modules a priority. Once a module's error rate hits about 10 defects per thousand lines of code, review it to determine whether it should be redesigned or reimplemented. If it’s poorly structured, excessively complex, or excessively long, redesign the module and reimplement it from the ground up. You’ll shorten the schedule and improve the quality of your product at the same time.

Quality-Assurance and Development Speed

If you can prevent defects or detect and remove them early, you can realize a significant schedule benefit. Studies have found that reworking defective requirements, design, and code typically consumes 40 to 50 percent of the total cost of software development (Jones 1986). As a rule of thumb, every hour you spend on defect prevention will reduce your repair time from three to ten hours.2 In the worst case, reworking a software requirements problem once the software is in operation typically costs 50 to 200 times what it would take to rework the problem in the requirements stage (Boehm and Papaccio 1988). It’s easy to understand why. A 1-sentence requirement can expand into 5 pages of design diagrams, then into 500 lines of code, 15 pages of user documentation, and a few dozen test cases. It’s cheaper to correct an error in that 1-sentence requirement at requirements time than it is after design, code, user documentation, and test cases have been written to it.

Figure 2 illustrates the way that defects tend to become more expensive the longer they stay in a program.

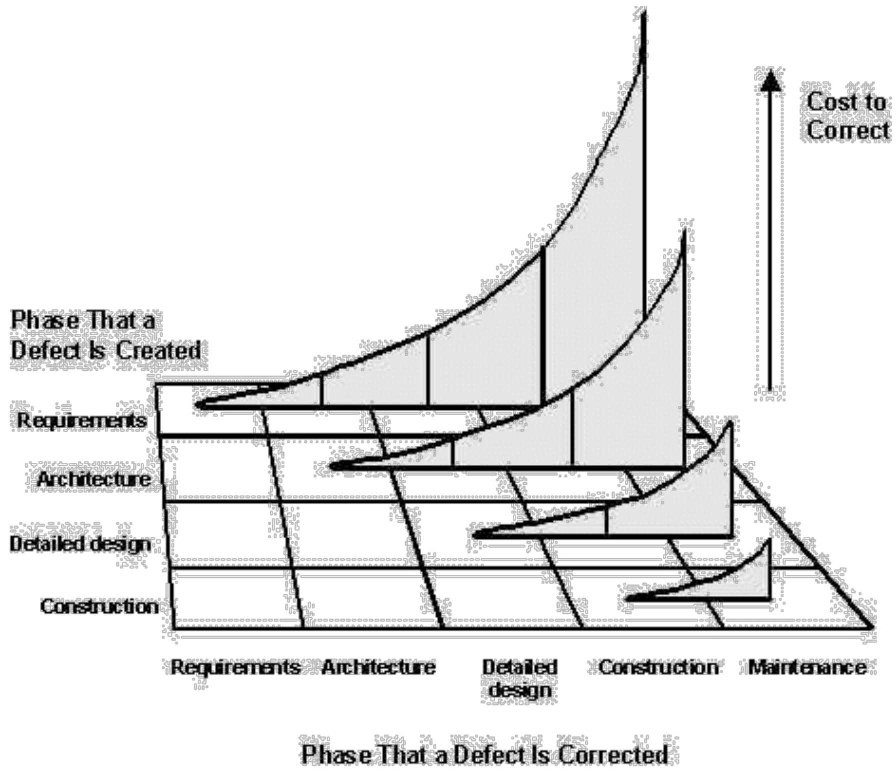

Figure 2. The longer a defect remains undetected, the more expensive it becomes to correct.

The savings potential from early defect detection is huge--about 60 percent of all defects usually exist by design time (Gilb 1988), and you should try to eliminate them by design time. A decision early in a project not to focus on defect detection amounts to a decision to postpone defect detection and correction until later in the project when they will be much more expensive and time-consuming. That’s not a rational decision when time is at a premium.

Quality-Assurance Practices

The various quality-assurance measures have different effects on development speed. Here is a summary.

Presentations

The most common quality-assurance practice is undoubtedly execution testing, finding errors by executing a program and seeing what happens. The two basic kinds of execution testing are unit tests, in which the developer checks his or her own code to verify that it works correctly, and system tests, in which an independent tester checks to see whether the system operates as expected.

Testing is the black sheep of QA practices as far as development speed is concerned. It can certainly be done so clumsily that it slows down the development schedule, but most often its effect on the schedule is only indirect. Testing discovers that the product's quality is too low for it to be released, and the product has to be delayed until it can be improved. Testing thus becomes the messenger that delivers bad news.

The best way to leverage testing from a rapid-development viewpoint is to plan ahead for bad news--set up testing so that if there's bad news to deliver, testing will deliver it as early as possible.

Technical Reviews

Technical reviews include all the kinds of reviews that are used to detect defects in requirements, design, code, test cases, and other work products. Reviews vary in level of formality and effectiveness, and they play a more critical role in maximizing development speed than testing does.

The least formal and most common kind of review is the walkthrough, which is any meeting at which two or more developers review technical work with the purpose of improving its quality. Walkthroughs are useful to rapid development because you can use them to detect defects earlier than you can with testing.

Code reading is a somewhat more formal review process than a walkthrough but nominally applies only to code. In code reading, the author of the code hands out source listings to two or more reviewers. The reviewers read the code and report any errors to the code's author. A study at NASA's Software Engineering Laboratory found that code reading detected about twice as many defects per hour of effort as testing (Card 1987). That suggests that, on a rapid-development project, some combination of code reading and testing would be more schedule-effective than testing alone.

Inspections are the most formal kind of technical review, and they have been found to be extremely effective in detecting defects throughout a project. Developers are trained in the use of inspection techniques and play specific roles during the inspection process. The "moderator" hands out the material to be inspected before the inspection meeting. The "reviewers" examine the material before the meeting and use checklists to stimulate their reviews. During the inspection meeting, the "author" paraphrases the material, the reviewers identify errors, and the "scribe" records the errors. After the meeting, the moderator produces an inspection report that describes each defect and indicates what will be done about it. Throughout the inspection process you gather data about defects, hours spent correcting defects, and hours spent on inspections so that you can analyze the effectiveness of your software-development process and improve it.

Because they can be used early in the development cycle, inspections have been found to produce net schedule savings of 10 to 30 percent (Gilb and Graham 1993). One study of large programs even found that each hour spent on inspections avoided an average of 33 hours of maintenance, and inspections were up to 20 times more efficient than testing (Russell 1991).

Comment on Technical Reviews

Technical reviews are a useful and important supplement to testing. Reviews find defects earlier, which saves time and is good for the schedule. They are more cost effective on a per-defect-found basis because they detect both the symptom of the defect and the underlying cause of the defect at the same time. Testing detects only the symptom of the defect; the developer still has to isolate the cause by debugging. Reviews tend to find a higher percentage of defects (Jones 1986). And reviews serve as a time when developers share their knowledge of best practices with each other, which increases their rapid-development capability over time. Technical reviews are thus a critical component of any development effort that aims to achieve the shortest possible schedule.

The Other Kind of Quality

I mentioned at the beginning of the article that there were two kinds of quality. The other kind of quality includes all of the other characteristics that you think of when you think of a high-quality software product--usability, efficiency, robustness, maintainability, portability, and so on. Unlike the low-defect kind of quality, attention to this kind of quality tends to lengthen the development schedule.

Summary

When a software product has too many defects, developers spend more time fixing the software than they spend writing it in the first place. Most organizations have found that an important key to achieving shortest possible schedules is focusing their development processes so that they do their work right the first time. "If you don’t have time to do the job right," the old chestnut goes, "where will you find the time to do it over?"

Additional Reading

The publications included in this list are also referenced throughout this article. Use them to your advantage when seeking furthers details about this subject.

Boehm, Barry W. "Improving Software Productivity." *IEEE Computer,* September 1987, pp. 43-57.

Boehm, Barry W. and Philip N. Papaccio. "Understanding and Controlling Software Costs," *IEEE Transactions on Software Engineering,* v. 14, no. 10, October 1988, pp. 1462-1477.

Card, David N. "A Software Technology Evaluation Program," *Information And Software Technology*, v. 29, no. 6, July/August 1987, pp. 291-300.

Gilb, Tom and Dorothy Graham. *Software Inspection* (Wokingham, England: Addison-Wesley), 1993.

Gilb, Tom. *Principles of Software Engineering Management* (Wokingham, England: Addison-Wesley), 1988.

Jones, Capers, ed. *Tutorial: Programming Productivity: Issues for the Eighties, 2nd Ed.* (Los Angeles: IEEE Computer Society Press), 1986.

Jones, Capers. *Applied Software Measurement: Assuring Productivity and Quality*. New York: McGraw-Hill, 1991.

Jones, Capers. *Assessment and Control of Software Risks*. Englewood Cliffs, N.J.: Yourdon Press, 1994.

Jones, Capers. *Programming Productivity* (New York: McGraw-Hill), 1986.

Kitson, David H. and Stephen Masters. "An Analysis of SEI Software Process Assessment Results, 1987-1991." In *Proceedings of the Fifteenth International Conference on Software Engineering* (Washington, DC: IEEE Computer Society Press), 1993, pp. 68-77..

Russell, Glen W. "Experience with Inspection in Ultralarge-Scale Developments," *IEEE Software*, vol. 8, no. 1 (January 1991), pp. 25-31.

About the Author

Steve McConnell is chief software engineer at Construx Software, a Seattle-area software-construction firm. He is the author of *Code Complete*, editor of *IEEE Software*’s "Best Practices" column, and an active developer. The material in this column was adapted from his

new book, *Rapid Development: Taming Wild Software Schedules*. You can reach him via email at stevemcc@construx.com or on the web at http://www.construx.com/stevemcc/.

Email me at stevemcc@construx.com.

Construx
Software Development Best Practices