Editor in Chief: Steve McConnell ■ Construx Software ■ software@construx.com

An Ounce of Prevention

Steve McConnell

"A stitch in time saves nine," the old saying goes. "An ounce of prevention is worth a pound of cure." In software, these expressions translate into the common observation that the longer a defect stays in process, the more expensive it is to fix.[1] Industry reports about the magnitude of the cost increase have varied over the years. The highest ratio I've seen published came from Barry Boehm and Philip Papaccio in 1988.[2] They reported that requirements defects that made their way into the field could cost 50 to 200 times as much to correct as defects that were corrected close to the point of creation. Of course, "50 to 200 times" is a rough average, and in the worst cases, the sky is the limit for defect costs—literally. The US space program had two high-profile failures in 1999: in both, correcting a defect "in the field" was not possible, and the software errors that went undetected until the software was in the field ended up costing hundreds of millions of dollars.

I've previously presented a rough rule of thumb that early, upstream defects generally cost 10 to 100 times as much to remove late downstream as they do to remove close to the point where they are created.[1] These observations have been used to justify a focus on upstream quality assurance activities such as extensive requirements work, design work, and technical reviews.

These old sayings and rules of thumb have come under attack in recent years. Some people claim that software defects aren't as expensive to correct as they used to be; costs don't increase as quickly as they used to. In other words, an ounce of prevention is not worth a pound of cure, but perhaps only an ounce of cure.[3] Some claim that we are expending more effort on prevention than we would by fixing the defects later—that we're spending a pound of prevention to avoid an ounce of cure.

Old support for an old saying

One common project dynamic is to cut corners because "we're only 30 days from shipping." If you're in a hurry, for example, you might decide that you don't have time to design and code a separate, completely clean printing module. So you piggyback printing onto the screen display module. You know that's a bad design that won't be extensible or maintainable, but you don't have time to do the right design.

Three months later, when the product still hasn't shipped, those cut corners come back to haunt you. You find that the people using the prerelease software are unhappy with printing, and the only way to satisfy their requests is to significantly extend the printing functionality, which can't be done with the piggybacked version. Unfortunately, in the three months since you took the shortcut, 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.

We have understood the dynamic in play in this example at least since the 1970s when IBM observed that software quality and software schedules were related. It found that the products with the lowest defect counts were also the products with the shortest schedules.[4]

Work on a software project generally follows a pattern of a small number of high-leverage upstream decisions providing the basis for a much larger number of lower-leverage downstream decisions. Thus we make high-leverage requirements decisions that provide the basis for medium-leverage design decisions, which in turn provide the basis for low-leverage code, test-case, and end-user-documentation decisions.

A small mistake in upstream work can affect large amounts of downstream work. A change to a single sentence in a requirements specification can imply changes in hundreds of lines of code spread across numerous classes or modules, dozens of test cases, and numerous pages of end-user documentation.

Capers Jones reports that reworking defective requirements, design, and code typically consumes 40 to 50 percent or more of the total cost of most software projects and is the single largest cost driver.[5] Tom Gilb reports that about half of all defects usually exist at design time,[6] which is confirmed by Jones's data. If half of all defects are upstream defects, you should be able to save effort by detecting defects earlier than system testing. Jones reports that, as a rule of thumb, every hour you spend on technical reviews upstream will reduce your total defect repair time from three to ten hours; that is, one ounce of prevention is worth three to ten ounces of cure.[7]

Has this dynamic changed in recent years? Recent data from Hughes Aircraft shows that the average requirements defect still takes 10 times as much effort to correct during system testing as it does during requirements analysis.[8]

The dynamics of defect-cost increase are inherent in the nature of software engineering work. It doesn't matter whether the project follows an old-fashioned waterfall life-cycle model or uses a cutting-edge iterative approach—design, code, test cases, and documentation will have dependencies upon requirements regardless of whether the project is done all at once or divided into numerous incremental releases.

Overall, I see no indication either from industry data or analysis that the dynamics of defect-cost increase have changed in recent years.

What does that ounce of prevention look like?

While the underlying dynamic of defect-cost increase has not changed, our understanding of how to detect upstream defects has improved considerably. Not too many years ago, we thought that the best way to detect requirements defects was to capture an exhaustive set of requirements in a monolithic requirements specification and then to subject that specification to intensive reviews. Although industry data suggests that this approach is cost-effective compared to the alternative of jumping straight into coding and then fixing requirements defects at construction time, we now know of numerous alternatives that are often preferable to the monolithic-requirements-specification approach:

- *Involve end users as early as possible.* Several studies have found that end-user involvement is key to stable requirements and software project success.[9]
- *Create a throwaway prototype.* Create a throwaway UI and put the prototype in front of real end users. Get feedback. Revise the prototype until the user is excited about the system. Then build the system. This approach correlates with good requirements stability and low system cost.[5]
- *Deliver the software incrementally.* Write production code for a small amount of the system. Put that functionality in front of the user. Revise the requirements, design, and code until the user is excited about the system. This approach does not entirely eliminate the defect-cost increase dynamic, but it shortens the feedback loop from requirements to user feedback in a way that reduces the number of downstream dependences that will be based on erroneous upstream work. This sort of incremental delivery approach correlates with high user satisfaction and lower total development costs.[1,6]
- *Conduct a requirements workshop.* Fast requirements elicitation techniques such as joint application development sessions are an effective way to shorten the time required to collect accurate requirements while simultaneously reducing requirements volatility downstream.[5]

IEEE Software

DEPARTMENT EDITORS

Bookshelf: Warren Keuffel, wkeuffel@computer.org

Country Report: Deependra Moitra, d.moitra@computer.org

Design: Martin Fowler, ThoughtWorks, fowler@acm.org

Loyal Opposition: Robert Glass, Computing Trends, rglass@indiana.edu

Manager: Don Reifer, Reifer Consultants, dreifer@sprintmail.com

Quality Time: Jeffrey Voas, Cigital, voas@cigital.com

STAFF

Group Managing Editor **Crystal Chweh**

Senior Lead Editor **Dale C. Strok** dstrok@computer.org

Associate Lead Editors **Jenny Ferrero and Dennis Taylor**

Staff Lead Editor **Shani Murray**

Staff Editor **Scott Lorenz Andresen**

Magazine Assistant **Dawn Craig** software@computer.org

Art Director **Toni Van Buskirk**

Cover Illustration **Dirk Hagner**

Technical Illustrator **Alex Torres**

Production Artists **Carmen Flores-Garvey and Larry Bauer**

Acting Executive Director **Anne Marie Kelly**

Publisher **Angela Burgess**

Membership/Circulation Marketing Manager **Georgann Carter**

Advertising Assistant **Debbie Sims**

CONTRIBUTING EDITORS

Greg Goth, Kirk Kroeker, Nancy Mead, Ware Myers, Judy Shane, Margaret Weatherford

Editorial: All submissions are subject to editing for clarity, style, and space. Unless otherwise stated, bylined articles and departments, as well as product and service descriptions, reflect the author's or firm's opinion. Inclusion in *IEEE Software* does not necessarily constitute endorsement by the IEEE or the IEEE Computer Society.

To Submit: Send 2 electronic versions (1 word-processed and 1 postscript or PDF) of articles to Magazine Assistant, *IEEE Software*, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1314; software@computer.org. Articles must be original and not exceed 5,400 words including figures and tables, which count for 200 words each.

- *Perform use case analysis.* Rather than being satisfied with the users' first explanation of what they want a system to do, examine the system's expected usage patterns to better understand users' real needs.
- *Create the user manual first.* Some organizations have had good success creating their user manuals as a substitute for or supplement to a traditional requirements specification. End users seem to be better able to understand the contents of a user manual than a traditional requirements specification, and requirements elicitation goes more smoothly.

New twists on old sayings

Software engineering advances by periodically reexamining questions that we think we've already answered. An ounce of prevention is still generally worth a pound of cure, but some recent developments have improved the "ounces of prevention" at our disposal. I find it encouraging that so many good techniques have emerged in the past few years.

References

1. S. McConnell, *Rapid Development*, Microsoft Press, Redmond, Wash., 1996.
2. B.W. Boehm and P.N. Papaccio, "Understanding and Controlling Software Costs," *IEEE Trans. Software Eng.*, vol. 14, no. 10, Oct. 1988, pp. 1462–1477.
3. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, Reading, Mass., 2000.
4. C. Jones, *Applied Software Measurement: Assuring Productivity and Quality*, 2nd ed., McGraw-Hill, New York, 1997.
5. C. Jones, *Estimating Software Costs*, McGraw-Hill, New York, 1998.
6. T. Gilb, *Principles of Software Engineering Management*, Addison-Wesley, Wokingham, U.K., 1988.
7. C. Jones, *Assessment and Control of Software Risks*, Yourdon Press, Englewood Cliffs, N.J., 1994.
8. R.R. Willis et al., *Hughes Aircraft's Widespread Deployment of a Continuously Improving Software Process*, tech. report CMU/SEI-98-TR-006, Software Eng. Inst., Carnegie Mellon Univ., Pittsburgh, 1998.
9. *Charting the Seas of Information Technology*, tech. report, The Standish Group, Dennis, Mass., 1994.

EDITOR IN CHIEF:
Steve McConnell
10662 Los Vaqueros Circle
Los Alamitos, CA 90720-1314
software@construx.com

EDITOR IN CHIEF EMERITUS:
Alan M. Davis, Omni-Vista