

Revisiting a Software Engineering Culture



THE MORE
THINGS
CHANGE..

BY KARL WIEGERS

In July 1994, I published an article in *Software Development* magazine titled “Creating a Software Engineering Culture.” I received more emails in response to that article than for the seventy articles I’ve written since then—combined. This response motivated me to write my first book, also titled *Creating a Software Engineering Culture* (Dorset House, 1996), in which I described fourteen principles that can help any software team build high-quality products through the sensible application of appropriate processes—and maybe even have a good time along the way.

Much has happened in the software industry since the book was published. The agile movement arrived, many companies are outsourcing portions of their IT work, and multi-site development is commonplace. I’ve heard that the more things change, the more they stay the same. In this article, I reflect on those fourteen principles to see whether they apply to twenty-first-century software development.

1. Never let your boss or your customer talk you into doing a bad job.



Software professionals sometimes are pressured into doing what they consider to be a “bad job.” Uninformed managers or customers might challenge a technical person’s intent to perform a specific activity (such as a peer review) or to create a particular deliverable (such as a design model). I knew a manager who informed his developers that, to save time, he did not want anyone doing any unit testing. System testing on that project took twice as long as planned. Are you surprised?

As with all philosophical principles, this one should be tempered by reality. This is not an open invitation for developers to debate endlessly the technical decisions of the architect. Nor is it intended to encourage analysts and developers to build a Cadillac when a Chevy will suffice. Someone needs to make key project decisions, and the team must respect those decisions. By “bad job,” I mean an action that is unprofessional or clearly will not lead to the desired outcomes.

Resisting the pressure to do a poor job is a matter of professional ethics and

personal integrity. In a time when jobs are tight, an individual put in this situation must balance his professional notion of appropriate behavior against the possible threat of losing his job. The ACM and the IEEE Computer Society have jointly adopted a software engineering code of ethics and professional practice (see the StickyNotes for a link). It behooves all software professionals to familiarize themselves with this code and remember it when being pressured to do the wrong thing.

2. People need to feel the work they do is appreciated.

Software development today can be a high-stress job with long hours, rapid change, staff turnover, and the constant threat of outsourcing. Most software professionals are nicely compensated for their efforts. Nonetheless, most people appreciate receiving rewards and recognition for their contributions.

At a seminar, I once asked seventy employees of a company if they had a recognition program. Only a few raised their hands. At a break, the human resources manager whispered to me, “They all have a recognition program.” I whispered back, “It’s not working.” An invisible recognition program won’t be much of a motivator. An effective program helps with staff retention, morale, and quality of work life.

Recognition can be a powerful tool for the software manager. Suppose you’re attempting to implement peer reviews. Publicly recognizing the efforts of team members who make a good-faith

effort to conduct reviews signals that peer reviews are a desired behavior.

3. Ongoing education is every team member’s responsibility.

This principle applies now more than ever. All software practitioners and managers should identify their personal areas for skill growth and continually enhance their capabilities. The skills mix that contemporary practitioners need is changing as more development is outsourced. Some foresee reduced demand for American programmers and a greater need for business analysts in the future.

Conference and training seminar attendees obviously are trying to better their software skills. However, I think we can do a better job as an industry to foster professionalism and continuous learning. I’d like to see more practitioners become active members of professional software organizations, such as the Association for Computing Machinery, the IEEE Computer Society, and the American Society for Quality. Books and magazines are good learning mechanisms, but I’d like to see higher circulations for software periodicals. I’d also like to see more people pursue professional certifications and advanced degrees, and I have great respect for those who do.

4. Customer involvement is the most critical factor in software quality.

Shortly after I wrote my initial article, The Standish Group released its oft-cited CHAOS report on the sad state of soft-



ware project success. The significance of the CHAOS data has been debated lately, but we still should respect the finding that lack of user input is a major contributing factor to struggling or failed projects. Customer involvement is particularly difficult on outsourced projects, where there can be thousands of miles and a dozen time zones between developers with questions and customers with answers.

I would broaden this principle to say “stakeholder involvement” rather than “customer involvement.” Customers are a subset of stakeholders, and users are a subset of customers. Key stakeholders need to be engaged throughout the project.

The past decade has seen good progress aligned with this principle. Usage-centered design, use cases, and the agile notion of having on-site customers have all contributed to this advancement. My teams at Kodak devised ways to work closely with our customers back in 1985. We developed the idea of product champions, key representatives who presented and reconciled the needs of specific user communities. Every project should look for champions to serve as the literal voice of the customer for the product’s significant user classes.

5. Your greatest challenge is sharing the vision of the final product with the customer.

I still think this is one of the most important concepts in software development. By “sharing the vision,” I mean analysts must represent and communicate requirements information among all project stakeholders (not just their customers) to align them toward a common objective. I read a lot of requirement specifications, and almost all contain exclusively natural language text. This is not the only way to share the requirements vision.

Software teams must expect to develop requirements iteratively and grow and record a shared understanding of the product over time. Effective analysts know how to use multiple techniques to represent product requirements. Possible “views” of the requirements include usage scenarios or use cases, lists of functional and quality-of-service requirements, graphical analysis models, prototypes, test cases, and screen designs. My observations,

though, suggest that today’s requirements analysts do little modeling. This is a shame, because visual models provide a powerful complement to textual requirements descriptions. Requirements development and management are all about communication, and requirements analysts need a full bag of tricks to help them communicate effectively.

6. Continual improvement of your software development process is both possible and essential.

Unless you can honestly proclaim, “I am building software today as well as software can ever be built,” you should always be improving some aspect of your personal capabilities and your team’s performance. Process improvement is simply a matter of doing something better than you did it the previous time. Even adopting agile techniques is process improvement by this definition, although the words “agile” and “process” rarely appear together.

The Capability Maturity Model for Software (SW-CMM) drove many organizations’ improvement efforts during the 1990s. The SW-CMM has largely been supplanted by the Capability Maturity Model Integration (CMMI). I suspect, though, that large-scale, formal, process improvement strategies are falling out of favor. The agile movement is to some extent a backlash against the “high-ceremony,” rigorously defined and enforced processes that sometimes imposed excessive overhead. As with any bipolar situation, either extreme is always inappropriate. Every software organization can benefit by adopting better ways of working that are flexible and adaptive.

Before you adopt any specific process improvement strategy, assess why you’re not already achieving your desired performance. As you pursue process improvement, watch out for these common traps:

- Lack of management commitment
- Unrealistic management expectations
- Stalling on action plan implementation
- Making achieving a maturity level the primary goal
- Failing to scale processes to project size

7. Written software development procedures can help build a shared culture of best practices.



Radical divergence in the way team members work can lead to inefficiencies. It’s harder for developers to work from design specifications written in myriad fashions using multiple modeling conventions. Therefore, many organizations find it valuable to develop a body of process knowledge and assets that enhance the capabilities of all team members.

People sometimes balk at the notion of using standard templates and processes. They fear that standards will force them to produce documents that consume effort without adding commensurate value. Certainly, this can happen if people aren’t being sensible. Some fear that processes are an attempt to remove individuality and stifle creativity. I’ve never found this to be the case. I follow a process for writing books, but it doesn’t restrict what words I put on the page. Processes and templates must be flexible in order to work for diverse circumstances. I use the phrase “shrink to fit”—begin with a broadly applicable process or template, then mold it to suit the nature and size of each project.

The term “best practices” has taken some heat in recent years. Who decides what practices are best, and on what basis do they make that evaluation? Most software engineering practices are context dependent; a practice that works well in one project environment might be inappropriate in another. I often talk about “good practices” rather than best practices because a new practice only has to be better than your prior approach to be valuable.

8. Quality is the top priority; long-term productivity is a natural consequence of high quality.



All software practitioners want to improve their productivity. What’s inhibiting your productivity today? In many cases, it’s rework—redoing something you thought was already finished and then retesting or re-reviewing it. Rework includes correcting defects, re-implementing changed or misunderstood requirements, and adding

overlooked functionality. Those few organizations that measure their rework effort find scary numbers, typically 30 to 50 percent of total development effort.

Various studies have shown that it can cost more than one hundred times more to correct a requirement-related defect found by the customer than if it had been discovered during requirements development. This implies that taking actions to improve quality—in its many dimensions—can increase productivity by reducing late-stage rework. Some claim that this accelerating cost-of-change curve flattens with agile development techniques, but I haven't seen data to support this.

Before you go looking for the newest tool or methodology that claims fabulous productivity benefits, analyze the root causes that reduce your team's productivity. Poor quality likely will be one of them.

9. Strive to have a peer, rather than a customer, find a defect.

This principle summarizes my personal experiences since 1988 with the benefits of peer reviews and inspections, although reviews do not replace various types of testing. Inspection repeatedly has been demonstrated to yield up to a 10-to-1 return on investment. A seminar attendee recently told me that her organization adopted inspections after a major project failed with more than 600 known defects. The second attempt was a success, with only about sixty defects. She attributed much of the improvement to their use of inspections.

Nonetheless, my surveys of seminar and conference audiences suggest that depressingly few practitioners know about the thirty-year-old technique of software inspection. Even fewer routinely perform effective inspections or other types of peer reviews. Pair programming is a way to have a colleague find errors and make improvements during construction. However, pair programming is no substitute for having someone who isn't intimately familiar with the work look at it with a fresh perspective and different assumptions.

I would never consider working in an organization in which peer review wasn't a standard part of the culture. If you believe

in this principle, you won't feel that a body of work is complete until other pairs of eyes have looked at it.

10. A key to software quality is to iterate many times on all development steps except coding: Do this once.

This is one of those philosophies that isn't fully achievable in practice, but it's still worth keeping in mind as you define your working methods. It is faster and cheaper to iterate on concepts, requirements, and designs than on code. Even if the requirements could be pinned down precisely, developers might need to iterate on designs. Refactoring to simplify and streamline existing designs can add value, but refactoring is not a substitute for iterating on design before diving into the source code editor.

Iteration demands tools and a culture that facilitate taking multiple stabs at how best to understand a problem and solve it. Design modeling, prototypes, and evolutionary delivery of executable code are all mechanisms for iteration. But the earlier in the lifecycle that iteration is performed, the cheaper it is.

11. Managing bug reports and change requests is essential to controlling quality and maintenance.

Change happens! It can be disruptive when it isn't anticipated. Experienced project managers realize that they need to structure their project lifecycles to anticipate and accommodate change. Because requirements will evolve during a project, it's essential to include contingency buffers in estimates to accommodate some growth. Incremental development strategies help teams cope with change and growth, providing multiple opportunities for reprioritization and project redirection.

The phrase "change control" has a bad reputation in some circles, but change control is not about inhibiting change. It's about establishing structures to ensure that suggested changes are evaluated properly and their impact assessed, to help the right people make good business decisions about which changes to adopt. A change-control system was the most effective process we

introduced into the Internet development group at Kodak, helping this fast-moving organization cope with an endless stream of work requests.

12. If you measure what you do, you can learn to do it better.

I've been involved with software metrics activities at the individual, team, and organizational levels. I find that collecting and analyzing data on project effort, time, and defects give me a richer understanding of the work I'm doing and the way I'm performing that work. Data also allows me to estimate future projects better than memories alone permit.

If you try to establish a metrics program, you'd better have a good answer if participants ask, "Why are we measuring this particular data item?" Start small, with a focused list of metrics that you believe will answer specific questions about the way your projects and products perform. Stay alert for the following common measurement traps:

- Measuring the wrong things
- Imprecise metrics definitions
- Collecting data that isn't used
- Using metrics to motivate, rather than understand
- Evaluating individuals using metrics data

People sometimes resist metrics because they fear that measurement will become an end in itself or that the measurement activities will consume an inordinate amount of time. Although certainly possible, I haven't encountered these problems. In my experience, collecting software metrics data is just a habit that you need to develop, not a time-sapping burden.

13. Do what makes sense; don't resort to dogma.

The only thing I'm completely inflexible about is not being dogmatic. Formal process improvement got something of a bad rap in the 1990s because some well-meaning people made it too rigid. It's easy to get caught up in the rigor and detail of a thoroughly defined process or a comprehensive modeling methodology. If you become a slave to



the process or the methodology, then you're working for it; it's not working for you.

A sensible process is not a straitjacket. Processes that are appropriate to your projects and culture provide a structured guide that can help nearly every team do a consistently better job. The desired project outcome is not to follow the process, fill out the forms, and execute the plan, but rather to build software that satisfies the stakeholders' business objectives. Process improvement is a means to an end, not the end in itself.

14. You can't change everything at once. Identify those changes that will yield the greatest benefits, and begin to implement them next Monday.

No matter how many opportunities you identify for personal, team, and organizational improvement, people can absorb only so much change at a time. I once met a team of twenty people who

were working on seven different process improvement activities concurrently. Twenty people simply can't make seven significant changes while still getting their jobs done. Rather than overloading on change, focus your improvement efforts on three, two, or perhaps only one initiative at a time—but never zero.

Retrospective Redux

It seems that these fourteen principles that made sense to me more than a decade ago still apply to contemporary software development. Perhaps these aren't universal truths, but they represent values that I think apply in many situations. Of course, identifying the principles that are meaningful in your situation is entirely up to you. If this set isn't an exact fit, have your team members think about what aspects of software engineering and management are important to them. Then take actions that are consistent with these values, principles, and priorities to steer your team to a healthier software engineering


culture—beginning next Monday. {end}

Karl Wieggers, PhD, is principal consultant with Process Impact in Portland, Oregon, where he specializes in requirements engineering, peer reviews, process improvement, and project management. He is the author of the books Software Requirements, More About Software Requirements, Peer Reviews in Software, and Creating a Software Engineering Culture and can be reached at www.processimpact.com. Karl would like to thank Wayne Allen, Kathy Getz, Kevin Jameson, and Kathy Rhode for their valuable review comments.

Sticky Notes

For more on the following topics, go to www.StickyMinds.com/bettersoftware

- ACM/IEEE Computer Society software engineering code of ethics
- Further reading




ASQ Certification
Make the Connection

ASQ's Software Quality Engineer Certification (CSQE) demonstrates your knowledge and proficiency in the software field. It helps you address the challenges of your current job, while providing employment mobility in a rapidly changing economy.

Define your career.
CSQE EXAM - December 2, 2006
Application Deadline - October 6, 2006

CALL 800-248-1946 OR VISIT
www.asq.org/certification/software-quality-engineer

Priority Code: SVAAA46



ASQ
MAKE GOOD GREAT™



STOP THE FREAK, KILL THE CREEP, BRING ORDER TO YOUR SDLC TECHNIQUE ...

THE COMPLETE SDLC SOLUTION ON TIME, ON BUDGET, ON THE MARK

With Pragmatic's "Software Planner" a 100% start-to-finish web-based solution that provides comprehensive project tracking, defects, test cases, robust team collaborative tools and much more, you'll stop creep without the employee "freak". Visit www.SoftwarePlanner.com to receive a free two-week trial ... Enter the promotional code "BSA" to receive a \$299 discount.



Software Planner
www.softwareplanner.com