Personal Process Improvement[1]

Karl E. Wiegers
Process Impact
www.processimpact.com

Process improvement is all the rage in software circles these days. However, many developers do not work in an organization with a management-driven process improvement program. Perhaps you're surrounded by process skeptics who succumb to the relentless daily project pressures, never investing energy in making tomorrow's project go better than today's. What's a lonely process enthusiast to do?

You can take several actions to improve your personal software engineering approach and perhaps your workgroup's processes, even without official management sanction or leadership. It's true that you will eventually hit a glass ceiling, at which point you'll be unable to influence people outside the scope of your own team. Nonetheless, anything you do to upgrade your own capabilities will save time, reduce rework and improve your productivity. Watts Humphrey's Personal Software Process (PSP) provides an aggressive approach to enhancing your development capability (see *A Discipline for Software Engineering*, Addison-Wesley, 1995), but not everyone is ready for the time commitment and rigor the PSP demands.

The suggestions here will take some time to learn and apply, but you don't need a lot of money for tools or training. Selectively adopt some of these practices as personal development habits and set an example for your team members to follow. You don't need anyone's permission to pump up your own software development knowledge, skills and practices. And when you achieve better results than before, others will take notice.

Project Planning and Tracking

Most developers are asked to prepare estimates for their work, but few are skilled estimators. Begin by recording both your schedule and effort estimates for individual tasks and your actual results. Comparing the two sets of numbers and understanding the differences will help you improve your estimating ability. You can also develop personal estimating heuristics, based on counts of program elements such as classes, methods, or GUI screens and widgets. Only data and experience can help you produce estimates better than those you will get from sticking a wet finger in the wind. Using data also helps you justify a project timeline to management. They may still not like the schedule, but estimates based on empirical evidence are harder to argue with.

To further enhance your personal estimation skills, create planning checklists for common project activities such as implementing a class, executing a system test cycle, or building a product from the components stored in your configuration management system. The checklists will help you avoid overlooking necessary tasks, a common cause of underestimation. Update the checklists as you gain experience.

---

[1] This paper was originally published in *Software Development*, May 2000. It is reprinted (with modifications) with permission from *Software Development* magazine.

Because no one really spends 40 hours per week working on an assigned project, keep track of how you actually use your time. Record the hours you spend every day on each development or maintenance activity phase. Don't try to account for every minute , but discover how many effective project hours you actually have available in an average week. Use this information to help you translate your task effort estimates (in labor-hours) into calendar time estimates. When planning, remember to leave time for quality activities, meetings, your other responsibilities and the inevitable rework.

Even if a detailed project plan isn't available, you can write short task descriptions to make sure you won't overlook steps you need to perform. Maintain an issues list to make sure that you track each one to closure, rather than leaving loose ends when you think you've completed an activity. Perform a retrospective review with your team members when you complete a project or phase to understand what went well and what could be improved. That is the essence of process improvement. Additionally, identify risk factors you can anticipate and control on future projects.

Requirements

Individual developers generally have little control over the software requirements presented to them. Therefore, you should strive to form collaborative relationships with those who supply your requirements, whether they are marketing staff, business analysts, managers or actual users. Ask to review a set of requirements before committing to implement them, and carefully examine each functional requirement to determine whether it contains enough information for you to design and implement it. Look for ambiguities and vague language that would lead to subjective interpretations. These weaknesses greatly increase the chance you will fall short of customer expectations. Watch out for these ambiguous terms: support, such as, and/or, etc., including, several, many, user-friendly, simple, typical, standard, usual, fast, robust, flexible, efficient and improved.

If you aren't sure of the intent behind a requirement, ask for clarification. Point out that you aren't sure what the customer needs, and you don't want to keep questioning people or rebuild the system if your first guess turns out to be incorrect. Look for possible exception conditions that were not addressed, and consider whether you could verify that each requirement was correctly implemented through testing or some other approach. Your critical eye will spot problems with requirements that an analyst or customer likely will not see.

Make sure you understand each requirement's relative priority, so you can implement them in the most appropriate sequence. Priorities are best defined through collaboration between customer representatives and technical people (see "First Things First: Prioritizing Requirements," *Software Development*, Sept. 1999). The customers can evaluate the relative customer value each requirement, use case or product feature provides, while developers can judge the relative cost and technical risk of implementing each item.

You might find that the requirements documents you receive from marketing or the business analyst don't meet your needs, because they are poorly structured, lack essential information or include spurious material that distracts you from the key content. Too often, critical documents that are intended to bridge the interface from one community to another are written only from the document author's perspective. For example, marketing might produce a marketing requirements document that seems reasonable to them, but has too many shortcomings to be useful to the downstream consumers of the document. It might be missing a clear statement of the product's scope and limitations, lack information about the product's intended usage, or describe conflicting quality characteristics.

Try to work with the folks who supply you with such key documents to agree on a document template that meets development's needs. Explain why a different document structure or content will accelerate development by reducing rework. This is the first step toward a collaboration that will make those critical documents as concise and valuable as possible.

Avoid the temptation to toss in extra functionality that you're just sure the users are going to love. Such gold-plating increases a product's development cost, but often adds little value. Instead, present your ideas to the customer representatives, so they can be evaluated against the other contemplated requirements. Make sure your customers understand the cost and other implications of adding new functionality or changing the requirements after you've started implementation. Don't implement proposed requirements until they've been approved by those responsible and allocated to a specific product release or build number. If customers request some surreptitious changes, politely steer them to the project's defined change control process. This helps ensure that sound business decisions are made to incorporate the most appropriate changes, thereby improving your team's chance of meeting its project commitments.

Design and Coding

The hard part of programming is thinking, not typing. Take the time to design your software before you implement it, to avoid having to build it over and over again to meet functional and performance needs. I've never produced an optimum design on my first try, so I learned long ago the value of iterating my designs. Learn about, adopt and share with your colleagues some standard design notations. The Unified Modeling Language (UML) is the hot design language these days, but the classical structured modeling techniques still have their place.

Models are communication mechanisms, so avoid the temptation to invent your own design notations. Using established conventions facilitates effective communication; contributing to the software Tower of Babel does not. Lead your group to agree on the design notations you'll use and the tools that will help you iterate your way to robust designs. Apply the principles of sound software design, such as strong cohesion, loose coupling and information hiding.

Get to know your programming tools. Really understanding the capabilities and limitations of your language, editor, compiler and other tools can improve your efficiency and effectiveness, enhancing your ability to develop high quality, industrial-strength code.

All programmers think their coding style is the best; indeed, why would they continue to follow it if they knew of a better way? However, we can all learn from the masters and improve our own programming approaches. Read some of the classic programming books, such as Steve McConnell's *Code Complete* (Microsoft Press, 1993), Jon Bentley's *Programming Pearls, 2nd Edition* (Addison-Wesley, 1999) and Donald Knuth's three-volume *The Art of Computer Programming, 3rd Edition* (Addison-Wesley, 1998). Apply the guidance in these books to develop—and consistently follow—sensible coding standards and sound construction techniques. As a simple example, I modified my commenting style after I spotted a better approach in *Code Complete*. I knew my old style wasn't perfect, but I liked the way it looked. Finding a better way encouraged me to confront the shortcomings in my current technique and overcome my intrinsic resistance to change.

Adopt good check-in/check-out configuration management discipline, keeping source code as well as other key project documents under version control. When you check in a module, record the reason for the changes you made when the tool prompts you; don't just enter a period or a null character to save a few seconds of typing. Document and automate the build procedures for your system so that anyone on the project can correctly build the product. Get in the habit of

frequent, even daily, builds of the growing product, using automated “smoke” tests to reveal bugs shortly after they are introduced.

Quality Practices

Each developer is responsible for the quality of the work she performs. Unfortunately, few developers are adequately trained in the arts of quality analysis and measurement, testing and technical review. Read a book on software testing, such as Brian Marick's *The Craft of Software Testing* (Prentice Hall, 1997), and actively apply the information to your own work. Decide what you mean when you say you are finished testing a program. Are you simply out of time? Did the program survive whatever tests you tossed at it? Or did you consciously select the quality filters and testing coverage that will let you reach an acceptable quality level?

Effective unit testing is another crucial skill. I once assisted a programmer who had spent three weeks testing a program but hadn’t recorded a single test. We had to start the testing all over because we didn’t really know what he had done.

Simply thinking of test cases often reveals errors in the code. You can document your tests in the form of code comments to make them repeatable. Develop automated test harnesses to accelerate regression testing, such as test functions or methods that are invoked through conditional compilation. When you modify a program, execute these regression tests to ensure that nothing else failed. Document your integration and system test cases and procedures to avoid having to redevelop them in the future and to enable someone else to properly test your programs if necessary. As defects are discovered, add new test cases to help you confirm each fix and grow a robust regression test suite.

Use quality tools to scour your code for as many potential problems as you can. At the least, use static code analyzers such as Lint (public domain or from Gimpel Software), Compuware NuMega’s CodeReview for Visual Basic and Parasoft’s CodeWizard to find subtle defects the compiler and a casual visual scan might miss. A developer once told me that Lint reported 10,000 errors and warnings when his team ran their program through it, so they never used Lint again! Finding 10,000 errors and warnings suggests to me that the program had some quality problems. The ostrich approach doesn't make the program more reliable.

Even if Lint isn't available, respect and correct compiler errors and warnings. The best programmers I’ve ever known taught me to make the compiler as fussy as possible and listen to everything it tells you. Your development team members should agree to use the same stringent compiler settings and to correct errors and warnings as soon as they are detected. If something is wrong with your code, fix it now when it’s cheap, rather than later when it’s painfully expensive.

Another class of quality tools is the dynamic code analyzer, exemplified by Rational’s Purify, Compuware NuMega’s BoundsChecker and Parasoft’s Insure++. These tools can reveal run-time errors such as memory leaks, writing past the end of arrays and assorted pointer problems. Tools won’t detect missing requirements or logic errors, but they can help you avoid a lot of debugging time and user grief.

Peer reviews are a valuable technique that can improve the quality of any work product. Find some colleagues you respect professionally and trust personally, and begin looking over each other’s requirements, designs, code and tests. Take a class on software inspection and introduce your team to these formal reviews. Many people complain that reviews consume too much time. While they do take time, anyone who has experienced the review benefits of finding defects efficiently will never want to return to programming in isolation. Reviews don’t waste your time—bugs do.

Maintenance

Maintaining ill-structured, undocumented and fragile legacy systems is no one's idea of fun. The mystery is why so many developers continue to create low-quality software for their colleagues or themselves to maintain in the coming years. Developers, managers and customers often balk at taking the time to document software, yet they complain bitterly about how hard it is to modify existing applications and the distraction that maintenance poses to new development work.

Take a stand against future maintenance nightmares. The time you spend documenting your designs and commenting your code will be amply repaid when you or your successors perform system surgery in the future. Clear documentation that matches the code enhances your ability to modify the program, while documentation that conflicts with the code is useless or even misleading. An old army maxim says that when the map and the terrain disagree, you should always believe the terrain. If the code and its comments do not agree, the maintainer is forced to believe the code, even if it is incorrect.

You might find yourself in a maintenance black hole, devoid of essential system information. Without reliable documentation, the maintainer must reverse engineer the code every time he has to make a change. While it is rarely cost-effective to completely reconstruct the documentation for a completed system, avoid digging a deeper hole. Document what you learn through reverse engineering, so you or a coworker need not rediscover that knowledge in the future. Record what you discover about a legacy system's interfaces to the rest of the world and share that knowledge with your team.

As you add new functionality, model it using your design conventions and show where it connects to the existing application, even if you don't have complete models of the entire system. Create and record test cases to verify the new code, and modify existing test cases as necessary. Introduce changes at the highest level of abstraction that each change affects. For example, document a functionality change in the requirements specification, then cascade the change into affected design elements, code, test cases and other artifacts. Create a portion of a requirements traceability matrix to capture these logical links. Simply diving into the code to implement a new feature creates disconnects between the code, the requirements and the system test cases, and results in a brittle system that is less amenable to future change.

Try to leave maintained code in better shape than you found it, but avoid the temptation to rewrite a module just because you don't like its style or structure. I fell into this trap once and wasted several hours before I caught myself. However, if you find yourself making many corrections in a particularly error-prone module, consider rebuilding it to reduce future maintenance effort.

A valuable process improvement is to keep records of bug reports and enhancement requests through a standard change control process. You can start such a process on your own, then share it with your teammates to establish a consistent way to manage change on your project. A simple spreadsheet may be enough to start recording bugs, but for the maximum benefit, try out a dedicated issue-tracking tool to support the process. GNATS is the public domain GNU bug tracking system for Unix; see http://www.alumni.caltech.edu/~dank/gnats.html. Commercial configuration management tools typically include a problem tracking component (visit http://www.methods-tools.com for examples). Remember, though: a tool is not a substitute for a process.

Track the time you spend dealing with each change or fix to understand the true cost of quality shortfalls or overlooked requirements on your project. While some change is legitimate and unavoidable, excessive change requests or problem reports can indicate deficient requirements

development, poor programming or inadequate testing. Ferreting out the causes of wasteful project rework gives you clear improvement opportunities on which to focus in the future.

At times, I've suffered a spate of "bad fixes" when performing maintenance hastily. Either I failed to fix the defect properly or I introduced new defects in the process. Track the percentage of your maintenance fixes that aren't done correctly and understand why. My bad fix causes included failing to think enough about the problem, addressing a symptom rather than the underlying disease, inadequate regression testing and just plain sloppy programming when rushing to get something done. Use those insights to improve your checklists that identify the steps to perform when implementing a change. Set a goal to reduce your bad fix rate by, say, 50 percent in the next three months and track your progress toward that goal.

The Process Improvement Mindset

I've never known anyone who could honestly claim, "I am building software today as well as software can ever be built!" If you can't say this, you should always be looking for a better way to do your work. From my perspective, process improvement is simple: consistently apply the practices you know give you good results, and continually work on a few selected areas that you know would give you better results. Sustained improvement demands a personal commitment to never stop learning.

It takes time to internalize new ways of working, to effectively apply unfamiliar techniques and accept them as better than your traditional approaches. Once you've adopted improved practices, never let your boss, your customers or your colleagues talk you into doing an inferior job.

Take Your Personal Process Pulse

	Rarely or Never	Sometimes	Usually	Always
1. Do you carefully review the requirements specification before implementing it?				
2. Do you create explicit design models or documents?				
3. Do you hold design reviews and code reviews?				
4. Do you track how you spend your project time?				
5. Do you record your task estimates?				
6. Do you compare actual task outcomes to the estimates and learn from the difference?				
7. Do you avoid adding cool new functionality no one asked for?				
8. Do you follow a consistent coding style standard?				
9. Do you use Lint or other code analysis tools?				
10. Do you use run-time code analyzers to find memory leaks and the like?				
11. Do you plan your unit tests systematically?				
12. Do you write down your unit tests?				
13. Do you improve your process based on the defects you find?				
14. Do you track bug reports systematically?				
15. Do you have documented and automated build procedures?				
TOTAL POINTS:				

Scoring: Give yourself 0 points for each Rarely or Never answer, 1 point for each Sometimes, 2 points for each Usually, and 3 for each Always.

35-45 points: Serve as a software engineering mentor for others in your team.
25-35 points: Adopt the discipline to consistently apply the practices you now do part of the time.
15-25 points: Pick out two practices you don't currently perform, learn about them, and try them – next week.
0-15 points: There are probably many points of pain in your software development daily life. Start addressing the items on which you scored the lowest, beginning next Monday.