Creating a Software Engineering Culture[1]

Karl Wiegers

Process Impact
716-377-5110
www.processimpact.com

Rarely in history has a field of endeavor evolved as rapidly as software development is right now. The struggle to stay abreast of new technology, deal with accumulated development backlogs, and cope with people issues has become a treadmill race, as software groups work as hard as they can just to stay in place. The Software Engineering Institute (SEI) and hordes of gurus exhort us to improve our development process, but how can we afford the time? Not every member of an organization feels the need to change. It is too easy to dismiss process improvement efforts as just the latest management blathering. Therein lies the seeds of conflict, as some members of a team embrace new ways of working, while others mutter "over my dead body."

The culture of an organization is a critical success factor in its process improvement efforts. "Culture" includes a set of shared values and principles that guide the behaviors, activities, priorities, and decisions of a group of people working in the same area. When coworkers align along common beliefs, it is easier to induce changes that will increase the group's effectiveness and its probability of survival. A shared culture is one difference between a "team" and a "bunch of bozos on a bus" (to paraphrase Larry Constantine).

This article describes some of the shared philosophy – the culture – that evolved over several years in a small software group in a very large corporation. Our group develops custom applications to support photographic research at Eastman Kodak Company. I will describe how each belief or value influences us to act in certain ways as we continually try to improve the way we build software systems. We feel this culture has improved our effectiveness as software engineers, the relationship and reputation we have with our customers, our level of teamwork, and the enjoyment we obtain from coming to work every day. Our general approach to implementing software engineering in this small group was described previously in *Computer Language*.[1]

We are now in our fourth year of continual process improvement. During that time the group has more than doubled in size, some roles have changed, and those few people who did not endorse the common culture have moved on. Sharing our philosophy with candidates during interviews made it easier to select those who would easily assimilate into the team and support the values we've agreed upon. And it has to be easier to move into an organization that has thought about what it believes than to leap blindly into the job fire and see how it works out.

Customer involvement is the most critical factor in software quality.

Any successful enterprise requires that you both do the right thing, and do the thing right. We expect that professional software developers know how to do the "thing" right. Doing the right thing, though, requires an unambiguous understanding of what your customer expects. To this end, we strive to maximize participation of our customers in our development activities.

---

[1]This paper was originally published in *Software Development*, July 1994. It is reprinted (with modifications) with permission from *Software Development* magazine.

Involving customers in development is no doubt simpler in a situation like ours, where our "customers" are fellow employees, usually in the same building we inhabit. The problem of customer involvement is more challenging when you are writing for the commercial marketplace; the involvement of marketing types may further confuse the issue. But for in-house development or even contract work, there is no excuse for not having end users involved from day one.

We have formalized our expectations for customer involvement by requiring that each project have one or more "project champions" from the user community. The project champion acts as our primary interface to the customer groups who are prospective users of the system being built. We simply decline to work on a project if customer management is not willing to commit appropriate project champions to the development team. Our software development guidelines include written project champion expectations, but the details are negotiable.

Besides the obvious value of having the voice of the customer directly available to the project team, we have discovered a side benefit of the project champion model. The champions gain insight into our structured software development approach and usually become strong advocates of our process with their peers and managers. The champions understand that the process we follow is aimed at building the right product as efficiently as possible. These influential members of the user community help smooth over the sometimes adversarial nature of systems development.

Our hardest problem is sharing the vision of the final product with the customer.

The classic failure of software development is that the product delivered only vaguely matches the expectations of the customer (assuming something is indeed delivered). Our first process improvement activity focused on improving our requirements specification process, in an effort to reduce this expectation gap. Every project, even small ones developing internally reusable software components, goes through a formal, written requirements specification process. We also make extensive use of both vertical and horizontal user interface prototypes to help refine user requirements and explore design alternatives.

"Formal" need not mean "cumbersome, voluminous, unwieldy, incomprehensible, all-inclusive, unchangeable, and interminable." Our objective is to simply prepare an explicit, consistent, unambiguous, structured document that clearly identifies the features that must be contained in the delivered system to meet user needs. Of course, requirements frequently must be modified as development progresses. This is fine, as long as everyone involved understands the impact and agrees to the changes.

We recently adopted the IEEE standard for Software Requirements Specifications (SRS),[2] which suits our needs nicely. An IEEE SRS contains sections to capture many of the pieces of information that are ultimately needed to build the system: constraints, external interfaces, and quality attributes, in addition to the actual functional requirements.

The project champions are key participants in creating the SRS. Typically, one of our team members does most of the actual writing, but it is truly a collaborative effort. The champions are responsible for resolving ambiguities and conflicts with the end users they represent, so that the SRS contains a single set of unified requirements. We refuse to get caught in the trap in which it is left to the programmer to resolve conflicting requirements and negotiate agreements with the different customers involved. We are rarely qualified to make those business-oriented decisions, so we leave them to the champions. The champions help determine which functionality adds the most value to the system while controlling the chrome. The project champion model is a cornerstone of our engineering culture.

We recently began including more specific quality attribute requirements in the SRS, such as maintainability, extensibility, portability, reliability, reusability, and performance. The project champions are playing an increasing role in agreeing on these less tangible requirements. Quantifying these attributes in the SRS forces the customer to think about them ahead of time, rather than waiting until the system is delivered and being disappointed because an important need was never explicitly discussed.

The quality of the requirements document must still be verified. Every SRS undergoes a thorough inspection by a group comprising the developers involved, the project champions, a software quality assurance representative from our team, and at least one outside person who has no specific involvement in the project. The outsider can be another of our team members, someone from a different software group, or anyone else who can critically evaluate the SRS from an unbiased perspective. Defects identified during the inspection are classified and recorded for appropriate action.

There are those who feel that this much structure around requirements specification is unnecessary or even counterproductive. On the contrary, we find it to be the foundation of quality in the successful systems we deliver. Our group has been measuring work effort distributions over different development and maintenance phases of all projects since 1990. Our data indicates that we spend an average of 21% of our total development effort on requirements specifications. I think this is a reasonable investment in "doing the right thing." To those customers, managers, or developers who claim we can't spend this much time on specifications, I quote the old sign from the chemistry lab : "If you don't have time to do it right, when will you have time to do it over?"

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

This principle seems self-evident, as quality is all the rage in every enterprise these days. But too many quality efforts are sacrificed on the altar of productivity, as managers and workers alike claim that they could produce more if only they didn't have to do all this quality-related stuff.

In software, the quality/productivity trade-off comes down to work versus rework, also known as "maintenance." Our group's philosophy is that the time we invest in building a quality product in the first place is amply repaid over the lifetime of the product through lower maintenance costs. The less tangible benefit is the goodwill of the customer. We believe that most customers are willing to wait a bit longer to get a better product that will help them do their work correctly without wasting their time through failures and poor usability. Our recent project experiences support this contention.

Our work effort metrics reinforce the concept of quality first. We classify maintenance into four categories: corrective (fixing bugs), perfective (adding enhancements), adaptive (modifications in response to a changing environment), and user support. Since we began our process improvement program, the fraction of our total work time that is devoted to corrective maintenance has declined to a steady state of about 2%, a source of considerable pride to us. Having this data available helps us pinpoint where our maintenance effort goes so that we can better focus improvement activities for the greatest leverage. Every hour we do not spend fixing an existing program is an hour we can spend writing something new to help our customers do their jobs better.

Of course, this isn't an infinitely deep well of opportunity. Once maintenance is reduced to an acceptably low level, we must seek additional productivity improvements elsewhere.

Another way to think about quality is to consider that each step in the software development process is the "customer" of the previous step. For example, the products created during requirements specification become the raw materials for design. This is true for any software development life cycle, as all (waterfall, spiral, evolutionary, object-oriented, etc.) involve the tasks of specification, design, implementation, and testing. Figure 1 shows one way to depict the customer/supplier relationships of the software development process. The product quality that can be attained at any step is limited by the quality of the raw materials supplied to it. This perspective encourages you to adopt practices that will assure the quality of the deliverables produced at each step in the cycle.

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

The Software Engineering Institute defines a scale of software process maturity, running from level 1 (initial) to level 5 (optimizing).[3] Virtually no organizations are performing at levels 4 or 5, although some individual projects have been measured at those levels. The premise behind SEI's scale is that a software organization that continually improves the way in which it does its work will get better results than an organization in which each member does things in his own random way.

While few of us are likely to reach the stratospheric level 5 any time soon, every software group has opportunities for improvement. We should be looking constantly to acquire and share best practices in the many subdisciplines that constitute the complete software development cycle. A culture in which sharing of ideas and practices is encouraged, in which ongoing education is supported and rewarded, will climb the process maturity ladder faster.

We believe that process improvement must be evolutionary, not revolutionary, but your culture may be bolder. You might begin with a group brainstorming session to identify the most pressing improvement opportunities. Then devise a simple plan for implementing these improvements, with explicit target dates, clear deliverables, and specific responsibilities accepted by different team members. Participation by all team members builds a stronger sense of ownership of the process and the results. Imposing changes by management fiat should be the last resort. Your process should be reexamined periodically to judge its effectiveness (perhaps with the help of metrics data) and uncover new improvement opportunities or critical needs.

We have also found it valuable to set annual team quality improvement goals. Each year the team collectively identifies six to ten specific goals in four or five key result areas. Quantitative objectives are agreed upon, and we track progress throughout the year. We have succeeded in making these goals be drivers for changing the way we do our work, rather than just being targets we hope to magically hit at the end of the year. The selection and achievement of measurable and meaningful improvement goals also sends a message to our management that we are serious about improving how we do our work.

Culture is a vital success factor in introducing software metrics programs, a characteristic of more mature organizations. Team members must feel confident that the data will not be used against them and that it will be used for some tangible benefit. Our successful collection and use of work effort metrics is predicated on the notion that the data belongs to the team members, not the team leader. The leader shares the data summaries so that the entire team can reach conclusions and agree on process changes based on the data, as well as share in successes indicated by the metrics collected. I fear that software metrics programs are likely to fail in organizations that lack this mutual trust and commitment to quality improvements.

Ongoing education is every team member's responsibility.

While filling some new positions in our team recently, I asked each candidate how he or she kept up with the software literature. The answers were discouraging. Some didn't understand the question. Few had read any software books lately. Very few were in advanced degree programs. Most of the magazines read were the free trade publications, not technical periodicals. Almost none belonged to professional computing organizations such as the IEEE Computer Society or the Association for Computing Machinery.

Our group subscribes to over 15 technical computing publications, covering a wide range of topics: databases, C programming, PC technology, UNIX, mainframe, software engineering, software quality, and so on. We purchase (and read) many books. We attend (and present) at conferences. Several team members have acquired undergraduate or advanced degrees in computer science or software engineering through night studies. Each team member can obtain formal training in skills directly relevant to work activities at company expense.

Some of these improvement activities are expensive, but many are not. In a rapidly changing technical area like software development, every professional should spend time studying the published literature and looking for ways to improve what he does. Our culture also encourages sharing of ideas, brainstorming, and consulting with peers for advice. Everyone feels comfortable approaching others for technical input and informal critiquing of work products. A competitive environment in which people are reluctant to share their knowledge is not conducive to improving the team's performance.

The software manager is not exempt from the continual learning challenge. Much has been published about software project management, but writings on software *people* management are harder to find. An excellent place to start is with DeMarco and Lister's *Peopleware*.[4] This small book is an easy read with a wealth of information about the importance of human issues in software quality, productivity, and job satisfaction. I also highly recommend *The Decline and Fall of the American Programmer*, by Ed Yourdon.[5] Don't lose interest or panic after reading the highly hyped first chapter on the title topic. The rest of the book is an excellent treatise on contemporary high-quality software development practices we should all implement.

We prefer to have a peer, rather than a customer, find a defect.

We've all received calls from irate customers, complaining that bugs in our programs wasted their time, gave the wrong results, or otherwise aggravated them. Our group's philosophy is that we'd rather suffer the slight embarrassment of having another team member find a flaw in a design or program than to have to face a user with homicide on his mind.

This value translates into a willingness to have our development products formally inspected by peers in an effort to identify faults (defects) before they cause failures. All types of deliverables undergo inspections: requirements specifications, design models, code, test plans, system documentation, user manuals. Although not every deliverable produced is inspected, most of us now feel a little nervous if we haven't had someone else look over our work before we continue. Specification inspections have the greatest leverage, since an error corrected at that stage need not be corrected, at much higher cost, when it is eventually found farther down the road.

There is considerable evidence in the software literature that inspections are the single most effective quality activity you can perform. If you don't know how to get started, try the buddy system. Pair up with one of your associates and offer to review his products if he'll look over yours. Several books that describe techniques for reviews, inspections, and walkthroughs have been published.

An organization's culture is very important in determining whether inspections will be successful. Again, you need an atmosphere of trust and mutual respect. Participants must be careful to critique the product and not the producer. Conversely, the producer should not view improvement suggestions as a criticism of his abilities or skill. There should be a willingness to learn from each other. I've learned something from every inspection in which I've participated, as either a reviewer or a developer.

It's not easy to put your work on the table for your peers to see and chew up. But if you can learn to check your egos at the door, every inspection you conduct will help you deliver a better product to your customers and help the participants improve their software skills.

My philosophy is to "inspect early and often." Programmers often are reluctant to let a peer see an incomplete product, preferring to finish it before exposing it to constructive criticism. I think this is a mistake. Remember, the leverage comes from finding problems as early as possible.

There are two reasons to encourage developers to submit materials for peer review early in the game. First, if an inspection of a preliminary or partial version of a document or program finds some systematic improvement opportunity (as with programming or writing style), the necessary changes can be put in place early. However, if the same problems are found only after 5,000 lines of code are written, the amount of rework may be so dauntingly large that the changes never get made.

Second, a lot more of our ego is tied up with a finished product than with a preliminary version. Hence, we are more resistant to making changes if the item being inspected is supposed to be the final deliverable. So try to force yourself to let others examine work products that are still at a stage where suggestions are welcome. Just let the reviewers know that the product being inspected is not in final form, so they know how to evaluate it.

Written software development procedures can help improve quality.

A strong indication of the process maturity of a software organization is whether it uses a reproducible, documented process for each new project. One of our earliest process improvement activities was to write a set of concise software development procedures. These are guidelines, not laws: if there is a VERY GOOD reason for doing something other than what the guidelines indicate, fine. But it should be a VERY GOOD reason, not a personal whim. Our procedures total less than 50 pages, but they address most of the important aspects of our work. Most importantly, people actually use them. You can encourage this in subtle ways: when someone asks how he should do something, ask him what your procedures say about it.

Don't fall into the "not invented here" trap when it comes to writing procedures (or anything else, for that matter). If you are in a good-sized company, there are probably already many sets of departmental procedures you can borrow from or simply adopt. The IEEE publishes many comprehensive software development standards.[2] We have recently begun to follow the IEEE standards for several aspects of development, including software requirements specifications, test documentation, quality assurance plans, and anomaly (defect, enhancement, and incident report) tracking.

The existence of software development guidelines helps build the shared culture of practices and expectations that improves a team's performance. This is particularly true if the team members are involved in writing, critiquing, and selecting the guidelines. Plan to review and update your guidelines periodically as part of your continual improvement activities.

The key to software success is to iterate as many times as possible on all development steps except coding: do this once.

There is ample evidence in the software literature that it is vastly more expensive to correct a defect in a delivered product than if the defect is found at an early development stage. Based on that data and my own experience, I have become convinced that we should try to write the code only once, doing whatever we need to before implementation to make sure we are writing the correct code, correctly.

To this end, we put a lot of work into perfecting our specifications and designs, without falling into the "analysis paralysis" trap. I figure that an extra 10 hours put into improving the specs now can easily save 100 hours later if it keeps me from delivering the wrong code. This is also why I believe in the value of building models of our systems before we write the code. We make extensive use of CASE tools for front-end modeling, particularly data flow diagrams, entity-relationship diagrams, and state-transition diagrams for user interface models (dialog maps).

The real power of CASE tools is the ease with which you can modify a diagram when you spot an opportunity for improving it. We can't possibly get it right the first time. Instead, we try to get it close the first time, and iterate, with input from others, until the model is as good as we can make it. How do you know when to stop? When you don't keep finding another change to make under every rock, and when the partitioning and the process or object interfaces "feel right" (this takes some experience).

I think of the use of structured software development methods as a way to avoid surprises. No one wants the kind of surprise that begins when a user says, "But I thought it was supposed to ...." Iterating on the requirements (statement of the problem) and the design (proposed solution to the problem) helps mightily to control the surprise factor.

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

Once in awhile you may be asked to cut corners on the quality of the work you do. Sometimes the requester is a supervisor who doesn't appreciate the value of the quality activities you practice or who feels budget or market pressures you may not. Sometimes it is a customer who wants you to concentrate on his specific needs when you've identified an opportunity to generalize beyond his requirements to provide enhanced value to a broader user community. I'd like to think we are beyond the stage of "don't bother with those specs, start writing code," but I fear this call of the software dinosaur still echoes throughout the land.

It is not easy to resist these pressures from the people who pay the bill or your salary. Sometimes you have no choice. But in a quality software culture, standard practices will be followed in times of crisis as well as in normal times. We have tried to adopt personal standards that motivate us to stick to our software process guns in the heat of the battle. Consider educating your boss so he better appreciates the value of your disciplined approach. Of course, you might need to temper this idealism with the reality of keeping your job, but we prefer to start from the position of quality *uber alles*.

People need to feel that the work they do is noticed.

When I was the supervisor of our software team, I initiated a simple recognition program. When someone reached a minor milestone on his project or made a contribution such as helping out a fellow team member with a problem, I gave him a small package of M&Ms, with a tag attached expressing congratulations or thanks as appropriate. Bigger achievements generated bigger bags of M&Ms or more substantial recognition awards.

As I expected, the candy disappeared immediately, but I was pleasantly surprised to see that people kept the tags visible around their desks. The important part was not the 50-cent bag of candy, but the words indicating that I noticed and valued the progress my team members were making. I also gave this sort of microrecognition award to people outside the team who helped us in some way. It brought smiles to their faces and goodwill to our relationships.

M&Ms already were something of an in-joke in our group; some other social recognition technique might work better for you. Interestingly, the group members themselves indicated that they preferred to have the M&M presentations made publicly at our weekly staff meetings, indicating the desire for peer recognition of even small achievements. We also spend a few moments at weekly staff meetings to give all team members a chance to pass along some "positive reinforcement" to others.

However you choose to do it, some kind of public praising and commendation seems to help build the spirit of striving for excellence that we all want in our teams.

Do what makes sense: no dogma.

Books and articles on software development methodologies abound. A common mistake made by organizations attempting process improvements is to adopt some published methodology lock, stock, and barrel. This is a recipe for wasted time, wasted money, resentment, and a feeling that process change is not possible in your team.

Every methodology has good ideas and silly ideas, in varying proportions. My advice is to select what seem to be the best ideas you can find and try to implement them in a nonthreatening but meaningful way. As new techniques and tools become adopted as part of your culture, move on to the next set of good ideas. If you find that a method you tried (*really* tried!) doesn't add value, don't do it again.

Process improvement efforts will more readily succeed when the culture permits experimentation and backing out of failed experiments. People who perceive that a risk of failure is acceptable are more willing to explore new ways to do things. If, however, the methods are imposed from on high, without regard to how well they work on *your* problems in *your* environment, you can expect rough sailing ahead. Our group has agreed to not adopt any one methodological dogma, but to routinely carry out those analysis and design activities we have learned by experience help us produce better systems for our customers.

Summary

This article describes how one small software group has gradually adopted a culture that provides a framework of values in which to make decisions, set priorities, and choose a path to higher performance. You can't buy a culture in shrink-wrap; you must roll your own. Every software team works in a different context of expectations, pressures, application domains, and technologies. The process of agreeing upon principles and values provides many opportunities for improving both the work environment and the work results. A shared culture is essential to progress through the software process maturity sequence to the discipline of repeatable and measurable software development processes.

Acknowledgement

The greatest determinant of software development effectiveness is the team members themselves. I was fortunate to start with a team of people who worked together well, cared about quality, and

were willing to try some new things. As we added new members, the team kept getting better. My thanks to all of those who cooperated with me in trying to improve the work we do.

References

1. Wiegers, Karl E. *Computer Language*, June 1993, p. 55.
2. *IEEE Software Engineering Standards Collection, 1993 edition*, IEEE, Inc., 1993.
3. Humphrey, Watts S. *Managing the Software Process*, Reading, MA: Addison-Wesley, 1989.
4. DeMarco, Tom and Lister, Timothy. *Peopleware*, New York, NY: Dorset House, 1987.
5. Yourdon, Edward. *The Decline and Fall of the American Programmer*, Englewood Cliffs, NJ: Yourdon Press, 1992.