

Implementing Software Engineering In a Small Workgroup¹

Karl E. Wieggers

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

The conference is over. You're back at your desk, battling the same alligators that filled the swamp before you left. You heard many interesting talks, with industry gurus preaching the latest software development wisdom and telling you how you should build systems now and in the future. What they didn't tell you was how to reduce to practice the highfalutin' ideas they endorse while still trying to get yesterday's project done by tomorrow. Besides, your projects aren't large enough to benefit from formal software engineering, and you don't have the resources to do it, anyway. The people in your group don't want to hop from one bandwagon to another, chasing the Holy Software Grail while your customers breathe down their necks. The default decision is business as usual, with a vow to do it better sometime next year.

In this paper, I'll describe how selected software engineering practices have been applied in and benefited several small software development groups in a large company. Our emphasis is on using structured development methods and a practical quality assurance process, not pursuing state-of-the-art techniques for all applications. We found that a systematic approach to continual software process improvement, combined with selective application of the many software engineering methods and tools now available, was not only feasible but rewarding. A commitment to continual improvement in our software development process has worked for us, and it can work for you. These, and related, experiences have been reported in more detail elsewhere [Wieggers, 1996].

Who We Are

The software team whose experiences are described here is part of the Research Laboratories at Eastman Kodak Company. We develop three principal kinds of software applications for our customers, who are Kodak photographic scientists and chemists:

- user interface-intensive, computation, data analysis, modeling, and graphics programs
- real-time lab automation systems
- information systems

Over time, this group developed a set of principles that helped guide our technical practices, management priorities, and team interactions. Such principles, often unspoken, help define the culture of team of people working together. By identifying and applying a small number of shared beliefs, a team can grow toward having a healthier software engineering culture [Wieggers, 1996].

We focus on delivering high-quality software. We believe productivity increases will materialize as a natural side benefit of quality, thanks to reduced maintenance over the product's lifetime. We further believe the greatest determinant in software quality is the degree and quality of customer involvement during development, because our biggest challenge is sharing a common vision of the final product with the customer. Without such a shared vision, some-

¹ This article was originally published in *Computer Language*, June 1993. It is reprinted here (with modifications) with permission from Miller Freeman, Inc.

one is bound to be surprised when the product is delivered. Software surprises are almost never good news.

We have defined key elements of our software process through a short set of written guidelines that help improve the quality and repeatability of what we do. Our philosophy of software development is to quickly iterate many times on requirements and design, but not on implementation. Ideally, we would like to write the code once, correctly.

To date, we have concentrated our software process improvement energy toward better requirements engineering, use of computer-aided software engineering (CASE) tools, software quality assurance, and software metrics. All these facets of software engineering require time, training, and persistence to yield results. I'll describe how we have improved our software development process by introducing these methods and tools into our daily work.

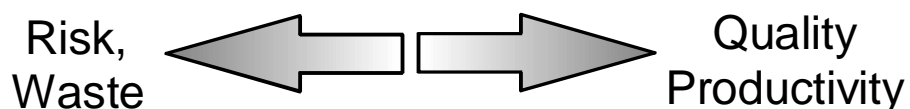
Process Improvement

Explicitly or implicitly, we all follow some process for developing software, traveling from vague user requirements to delivered code and documentation. The challenge is to improve the processes we use, with the goals of improving the quality of delivered software, increasing software developers' productivity, and optimizing job satisfaction in addition to quality and productivity. Even if your team doesn't define a "standard" software process, each individual applies a personal software process, which incorporates all they have learned over the years about the best ways for them to build software. Each individual can (and should) strive to continually improve his or her personal software process.

Several scales exist for assessing the sophistication of the software process used in an organization (Figure 1). The Software Engineering Institute's Software Capability Maturity Model defines five levels of process maturity: Initial, Repeatable, Defined, Managed, and Optimizing [Humphrey, 1989; CMU/SEI, 1995]. Meilir Page-Jones speaks of the ages through which a software organization evolves: anarchy, folklore, methodology, metrics, and engineering [Page-Jones, 1991]. Gerald Weinberg thinks in terms of the cultural patterns observed for software organizations: oblivious, variable, routine, steering, anticipating, and congruent [Weinberg, 1992].

Figure 1. Several evolutionary scales for software development.

Proponent	Term	Levels					
Page-Jones	Ages	Anarchy	Folklore	Methodology	Metrics	Engineering	
Weinberg	Pat-terns	Oblivious	Variable	Routine	Steering	Anticipating	Congruent
Humphrey	Levels	Initial	Repeatable	Defined	Managed	Optimizing	



Each of these scales calibrates the continuum of ways a software group can function. The implication is that organizations operating toward the high end of any scale are more likely to produce high quality products with high productivity. Organizations at the low end of the scale have a greater risk of project failure and wasted effort.

Any process improvement effort should begin with some kind of assessment, to examine and understand your current software development process and identify any deficiencies. Such an assessment can be a big, formal, and expensive event based on an established process improvement framework such as the CMM. You may prefer to start with a simple survey of current practices, and their strengths and shortcomings. Our group chose to begin by holding an introspective brainstorming session in which all team members participated. Group participation can lead to consensus and a stronger sense of group ownership of the issues raised. We used a method called brainwriting. Everyone wrote problems and concerns on index cards, and everyone had a chance to annotate all the cards. In a second pass through the cards, we all proposed solutions to the problems on each card. We came up with 33 separate issues.

I summarized the brainstorming issues in a public file, and we agreed upon action items allocated to different group members. As the action items were addressed over time, we updated this file to reflect progress made and deliverables produced. We focused on items that promised to yield the greatest leverage in improving the products we create. Some of the items included:

- Writing group development guidelines to define clearly our preferred process and the deliverables expected at various stages during development
- Increasing customer involvement in our projects
- Measuring what we do by collecting certain software metrics
- Improving our knowledge of software engineering through training, conferences, books, and periodicals
- Improved selection, standardization, and application of software tools
- Defining an appropriate software quality assurance process to apply to our projects.

Recognize that you can't simultaneously incorporate every software engineering technique and tool into any organization, even if they were all valuable, which they aren't. Try to identify a small number of high-leverage changes that can be applied effectively to your group with fairly low pain. We select just one or two key areas (such as prototyping, design, testing, project management) for each new project in which we wish to concentrate on improvement, based on individual evaluations of our weaker skills.

We try to learn new skills, such as project management and planning, on projects for which that skill is not absolutely critical to success. That way, when we encounter a project in which the skill *is* critical, we will have at least a little experience. As we master new abilities and tools, they become a regular part of our activities on subsequent projects so we can concentrate on yet another area of improvement.

Don't try to do everything at once. Most teams cannot effectively work on more than two or three improvement activities at once. I once encountered a team of 20 developers that had no less than seven improvement initiatives underway. The predictable results are failure and frustration, with a return to business as usual. Of the many methods touted by the gurus, select only those portions that look promising for your particular environment and do not hesitate to discard methods that simply do not work for you. Avoid the temptation to swallow any dogma whole, but don't use this restraint as an excuse to not try anything new.

Once launched, you'll need to sustain the process improvement energy in the group. We devoted at least one weekly team meeting per month to a discussion of a particular process issue. Every discussion was followed by a written summary and action items to address the deficiencies we identified. Some of the topics we addressed included:

- Who are our suppliers? What problems do we have with them? What can we measure to indicate trends of quality in the products or services they provide us?

- How do our customers define software quality? (This was an open forum to which our customers and their managers were invited.)
- How can we improve the testing procedures we use? What can we measure to help us know when a beta-testing period can be completed and the product released?
- How can we use our metrics to help us prepare better software project estimates?
- What are obstacles to improving our software productivity and quality? What are some possible solutions? (This led to another brainstorming activity.)

Development Guidelines

Our first brainstorming session showed that the lack of defined expectations for how a new application should be built was a handicap to reaching peak performance. Consequently, we wrote some guidelines to define the group's major software development and project management activities. The drafting of each guideline was assigned to a different team member, and all team members approved every guideline. We defined nine initial guideline categories:

1. A model of our software development life cycle
2. Defining expectations for the key customer representatives on our projects ("project champions")
3. Documentation guidelines for source modules and external system documentation
4. Testing plans and procedures
5. Software quality assurance activities expected for each project and metrics to collect
6. Change control process for existing applications
7. Configuration control process
8. User interface development standards
9. User manual style guidelines.

These guidelines helped clarify the expectations for the development process activities and the deliverables produced. For guidelines to be useful, they must be short and realistically attainable on each project. Look for the value added to your process and products by each guideline you write, rather than simply decreeing a standard based on the dogma in the latest book or article you read. Become familiar with the IEEE software engineering standards [IEEE, 1997], and try to borrow as many ideas as you can from other organizations. My philosophy on standards, procedures, and guidelines is to adopt first, adapt second, and invent last.

Your software development guidelines must be kept current. Part of our ongoing process improvement efforts involved reviewing our guidelines about a year after writing them to discard useless components and add anything we think we need. We take these guidelines seriously, although they are not cast in stone and can be disregarded in specific instances if doing something differently clearly makes more sense.

CASE Results

After acquiring training in structured systems analysis and design and applying these modeling techniques by hand on one project, we began employing CASE tools on our projects. Our first acquisition was a fairly primitive product I'll refer to as PaleoTool. PaleoTool's user interface was frustratingly cumbersome to use. However, it did help us build a well-structured system for real-time control of a laboratory robot that met customer needs and required very little maintenance during a year of extensive use.

The value of CASE and modeling was reinforced when we had to do a major enhancement a year later. It proved easy to modify the existing CASE models to design in the additional functionality. The contract programmer hired to do the implementation from my new design found the existing models and documentation most helpful in getting up to speed. The result again was a robust program that performed well with virtually no maintenance.

We eventually migrated to a full *Cadre Teamwork* installation, a high-end CASE tool at the time. Using *Teamwork*, we successfully built several information systems and scientific applications. We extensively used *Teamwork's* data-flow diagram editor, data-dictionary facility, entity-relationship editor, and state-transition diagram editor.

Our extensive CASE experience convinced us of the value of these tools in the analysis, design, and documentation of high-quality software systems. Once you commit your software development process to the discipline of structured methods, CASE tools are the only practical way to go through the large number of iterations in a model that leads to the highest quality product. Data-flow diagrams and other models also help us communicate with our customers. A wide variety of object-oriented modeling tools are now commercially available, augmenting the traditional structured methods embedded in older tools. Our CASE experience also taught us much about how to incorporate new technologies and tools into the culture of our team [Wiegers, 1996].

We did not observe a significant effect on the time needed for initial delivery of a system built with CASE models versus an unstructured approach. However, we feel CASE has had a positive impact on our productivity by reducing the time spent on corrective maintenance (the system has fewer defects at delivery) and perfective maintenance (it is easier to add enhancements).

Focus on Requirements

Requirements are the foundation for all subsequent engineering work and project planning. However, many software groups still struggle to collect, document, and manage high-quality requirements. One of our initial areas for software engineering improvement was to improve our requirements engineering processes.

Project Champion

The concept of a project champion, or key customer representative, is a cornerstone of our group's software development philosophy. For large projects with diverse customer community, a team of two to four champions may be assembled to represent different user classes. For projects without a specific customer clamoring for the program, we invite a project champion from among those potential users who seem most enthusiastic about the project, or from frequent users of similar, existing applications.

The project champion's role is to interface with the customer community and work closely with the developers to ensure that the system requirements are properly captured. The champion must communicate with other members of the customer community, collect input on their needs for the new system, and reconcile incompatible requirements expressed by different customers. By working from a unified set of requirements supplied by the project champion, we have fewer false starts, less confusion, and less rework. The result is a higher quality system that better meets customer needs.

The project-champion approach has worked well for us for nearly a decade. A side benefit is that a few key customers learn more about our software engineering process. Some project champions have even served as powerful advocates on our behalf, intervening with impatient customer managers who don't understand our approach to software development. It is very helpful to have allies among customers who will assure their managers that our projects are making adequate progress, even though no code has been delivered yet.

The project champion typically maintains that role after the initial system is delivered. All requests from customers for changes in the delivered system must be approved by the project champion. This policy helped stabilize our change control process. We wrote a change management tool that stores customer change requests in a database and provides automated communication among the customer, developer, and project champion as a change request is submitted and acted upon. The use of a change control process and tool helps us manage our backlog of requested enhancements and reported bugs for the many applications we support.

Prototyping and Dialog Mapping

Many of the applications we build are user interface intensive. We now routinely build models and prototypes of proposed user interfaces, which customers review for usability and correctness early in the design process. We use both paper prototypes of screens and windows, and executable prototype interfaces. The purpose of prototyping is to answer questions that you have early in development cycle, as inexpensively as possible. Effectively used, prototyping is a technique to reduce the risk of building the wrong product, or of building the right product badly.

My philosophy is to do the minimum amount of work on an executable prototype required to get the screens (or reasonable facsimiles) to display. With such a prototype, users can experiment with the navigation options and preview the functionality available at each screen to assess the look and feel. If we are experimenting with a specific interaction technique, we build enough working functionality into the prototype to let users make informed assessments of that technique.

Since we're trying to get answers to specific questions with the help of the prototype, we want to make sure useful responses are generated from user evaluations of the prototype. Rather than simply let users run rampant with the prototypes, we create a prototype evaluation script. This is a series of directed activities and feedback questions to help us obtain specific information about the best way to do things.

We also obtained good results from using CASE tools to build high-level models of a proposed user interface. These models show all the allowed navigation pathways among screens, but no details about screen design. Many user interfaces can be thought of as finite state machines, with only one display (screen or window) active at a time (a state), and defined conditions for moving from one state to another (a transition). Thus, the state-transition diagram is an appropriate interface modeling tool. Such diagrams are sometimes called dialog maps. We have constructed dialog maps of systems containing over 100 screens and windows using the state-transition diagram editor in *Teamwork*.

Customers have responded positively to the use of dialog maps as a high-level abstract view of an entire interactive software system. The dialog map helps us find missing or incorrect navigation pathways early, when they are cheap to fix. We can also spot opportunities for reuse and redundancies in the user interface, without worrying about the details of screen design prematurely.

Software Quality Assurance

Many large organizations have a dedicated software quality assurance (SQA) department of some kind. The software engineering literature stresses the importance of separating the quality assurance function from the development function, but a small group cannot afford to dedicate anyone to a purely software quality role. We found that a team size of about 10 was necessary before we could justify having one person specialize in quality practice across multiple projects. Before the group grew to that size, we tried an innovative approach that incorporated this philosophy of separating quality assurance from development and appeared to be effective for small organizations with limited resources.

The SQA Team Member

Each new project is assigned a development team consisting of:

- The primary developer and any other contributing team members
- Another group member who coordinates the software quality assurance function for that project
- One or more project champions (part-time role only)
- Another software person, from inside or outside our group, to serve as an additional participant in inspections (“extra pair of eyes”)

The SQA team member is responsible for writing the formal system test plan and for conducting reviews of deliverables produced during development (requirements specifications, designs, code, and documentation). Each group member is asked to play the SQA role on someone else’s project.

The software literature touts formal reviews inspections as a superior method for identifying defects. Our reviews involve at least the primary developer, the software quality assurance team member, and the extra pair of eyes, in addition to other possible participants. We also require the project champion to participate in reviews of the requirements specification. The purpose of the review is to identify (not correct) defects in the products being inspected. The earlier we find a defect, the cheaper it is to correct, because less rework has to be done to modify work products derived from the one that contains the original defect.

Our technical reviews have yielded significant benefits by revealing errors, omissions, inconsistencies, or misunderstandings in the documents being examined. The group members learn a lot from each other by participating in inspections and observing how others practice software engineering. The SQA team member also learns enough about someone else’s project to serve as backup maintainer if necessary in the future.

The SQA team member also is responsible for writing a system test plan, which in turn is reviewed by the primary developer. This process begins with the requirements specification and user interface prototype. It proceeds in parallel with system implementation. Unit testing is primarily the responsibility of individual developers.

The SQA team member can review the requirements specification for testability and write most of the tests early in the development process. Writing a test plan is an excellent way for someone besides the primary developer to become intimately familiar with a system. Writing and reviewing a test plan is almost as useful as executing it, in terms of identifying problem areas such as requirements that were not implemented. Each test case is traced back to an individual specification in the functional requirements document.

The SQA Challenge

This approach to software quality assurance uses our limited resources effectively to improve the quality of delivered systems, as evidenced by a high level of customer satisfaction and low maintenance burden. One consequence is that each project requires the part-time commitment of several people besides the primary developer. Spending time on test plans or reviews for someone else’s project means your own work will be delayed.

I can accept this trade-off because I have seen that the quality benefits are large enough to justify that price. Adopting a software quality assurance approach such as ours requires everyone in the group to believe that the short-term productivity cost is more than repaid by reduced long-term maintenance over every system’s production lifetime. Such a holistic, cost-of-quality perspective on software development and maintenance helps justify quality practices that may seem to the uninformed to lengthen the development schedule for no good reason.

Metrics Results

Noted author and consultant Tom DeMarco has stated that “You can’t control what you can’t measure” [DeMarco, 1982]. Unfortunately, the obstacles to software metrics collection are sufficiently daunting that many organizations do not know how to start. While it’s not practical to try to quantify every aspect of your work, we felt it was important to begin measuring some key aspects of our group’s projects, products, and processes. Our objectives in software metrics included:

- Better understand how we spend our time in projects and where we might make changes
- Monitor our productivity over time and assess the impact of process changes on productivity
- Do a better job of estimating the size of new projects, cost to the customer, and expected calendar time to completion
- Track the defects identified during and after project development so we can concentrate our attention on process improvements most likely to improve quality

For over five years, we have collected precise metrics on the time spent by group members on different phases for each of our projects (work effort). Each team member spends an average of only 10 minutes per week collecting and reporting the work effort metrics. This activity has been very successful, and the data have been useful in ways we had not anticipated. We distinguish the phases of software development activity in Table 1. These accounting categories apply regardless of the software development life cycle being followed (water-fall, evolutionary development, and so on).

Table 1. Phases of Software Development Activity

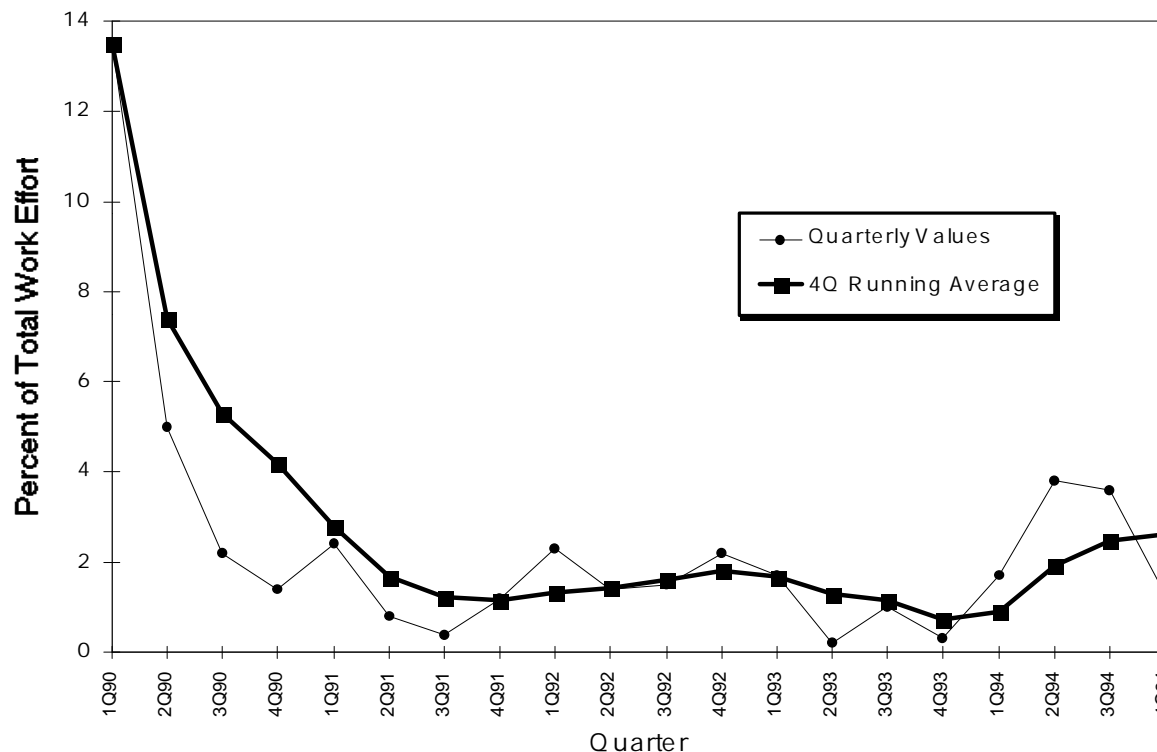
Phase	Description
Preliminaries and Project Planning	Work done prior to beginning the actual system definition and construction, such as specific training, feasibility discussion, project management, hardware and software evaluation and selection, and so on. Also project planning and tracking activities.
Requirements Specification	Gathering customer requirements, writing functional specifications, building and evaluating user-interface prototypes, building an essential data-flow model, and reviewing the specification documents.
Design	Data, process, and user-interface modeling of the proposed system design, program architecture, screen layout and navigation design, database design, module design, and reviewing design documents.
Implementation	Creating, documenting, and reviewing the executable components of the system (code, screens, databases, and help displays).
Testing	Writing, reviewing, and executing unit and system test plans.
Maintenance	All work done on the system after delivery, including corrective (fixing defects), perfective (adding enhancements), and adaptive (changing software to keep it functioning in a changed environment) maintenance, and user support.
Writing	Writing user manuals, on-line help, and system documentation.

By monitoring the trends we saw in our work effort metrics data over time, we could identify process improvement opportunities, and see the impact of the software engineering changes we made. As an example, Figure 2 shows how the fraction of our total software work effort devoted to post-release defect correction changed over time, since we began our process improvement activities. From an initial level of more than 13% of our total work, it dropped to a sustained average of about 2% of total work. We're proud of this, because it demonstrates that our combined quality practices have helped free up time from maintenance for new development work.

The work-effort tracking helped us set numerical targets for process improvement goals, such as increasing the fraction of total work devoted to design and testing to levels we feel are more appropriate. Without knowing where our time was going in the first place, we could not have analyzed our process this precisely and focused our improvement efforts. Having the measurement infrastructure in place then permitted us to track progress toward such goals.

Our work-effort metrics data also improved our ability to estimate the time required to complete new projects. We demonstrated a correlation between the number of specifications in a functional requirements document and the post-requirements phase work effort hours needed to deliver the product. While this simple correlation probably won't work for every future project, we are confident that collecting data on our project performance will help us build a better crystal ball for predicting the effort and schedule for new projects.

Figure 2. Trends in defect-correction work effort over five years.



Translating This to Your Organization

This paper has described a number of improved software engineering approaches that several real software groups have been able to implement. Since all software development groups are different, the techniques that worked for us might not all work for you. However, it is both feasible and necessary for *all* software development teams to identify the shortcomings in their current practices and begin to do something about them - starting now!

None of these methods will work without an explicit commitment on the part of the manager and team members to the continual improvement of your software development process. With realistic expectations, the discipline to persevere with your methods in the face of management or customer pressure, and the willing participation of all group members, even a small software organization can significantly improve the systems it builds and develop a healthier software engineering culture.

References

Carnegie Mellon University/Software Engineering Institute. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Reading, Mass.: Addison-Wesley, 1995.

DeMarco, Tom. *Controlling Software Projects*. New York: Yourdon Press, 1982.

Grady, Robert B., and Deborah L. Caswell. *Software Metrics: Establishing a Company-Wide Program*. Englewood Cliffs, N.J.: Prentice Hall, 1987.

Humphrey, Watts S. *Managing the Software Process*. Reading, Mass.: Addison-Wesley, 1989.

IEEE Software Engineering Standards Collection, 1997 Edition. Los Alamitos, Calif.: IEEE Computer Society Press, 1997. (for details, see the IEEE web site at www.computer.org)

Jones, Caper. *Applied Software Measurement*. New York: McGraw-Hill, 1991.

Page-Jones, Meilir. "Managing in the Object-Oriented Environment," *Proceedings of the Fourth International Teamworkers Conference*, Cadre Technologies, Inc., 1991.

Weinberg, Gerald M. *Quality Software Management: Systems Thinking*. New York: Dorset House Publishing, 1992.

Wiegiers, Karl E. *Creating a Software Engineering Culture*. New York: Dorset House Publishing, 1996.