Interns often seem little more than itinerant laborers, but the author's internship at a telecommunications company provided opportunities to both learn and contribute. The lessons learned in just one year indicate how common certain problems are in software development and how important basic practices can be.

A Fresh Perspective on Old Problems

Ryan Fleming

FROM THE TRENCHES: Wolfgang B. Strigel, editor • wstrigel@spc.ca

In July 1997 I began a one-year internship as a programmer in the information services department of a small but rapidly growing telecommunications company. Initially I spent my time debugging, extending, and updating the company's in-house billing/provisioning system, and dealt primarily with code, code-level quality, and program documentation issues. Soon, however, opportunities arose for me to contribute ideas that ultimately affected the entire department. These opportunities in turn provided me with valuable lessons about essential yet often overlooked software development issues—from programming practices to team organization.

Although many of my tasks were solo missions, I soon began helping other developers debug their programs. During one debugging session, a developer said of the problem at hand, "I don't know why [the program] does that." He then shared the practice of "waving the dead chicken," making nearly random, pseudo-logical code changes to "solve" the problem. I suggested looking at the manual or asking another developer, but he continued making "guess and check" code changes. Eventually—and by sheer luck, I'm convinced—we got the code working, but neither of us understood what was wrong or how we fixed it. The developer went happily on his way; I went to the manuals for an explanation.

Do you ever feel trapped in tunnel vision? Do you ever want to hit your personal "Reset Button" to rid yourself of preconceived ideas? This is how I felt when I read Ryan Fleming's article. It felt like I was peeling away layers of perception that had accumulated over years of working and consulting in the software industry.

This article describes a student's observations during a one-year internship in an Information Services department. His comments address many of the key issues in our industry. Fleming questions some of the ingrained practices and attitudes that we probably should revisit periodically to determine their usefulness. The article also made me reflect on the state of our corporate practices. If a student (albeit a very perceptive one) can identify so many of our problems in a year, why do we still have these problems? Should you determine that many of his observations are symptomatic of your own organization, it may be time to hit another reset button and make some changes.

—Wolfgang Strigel, From the Trenches editor

From Alchemy to Engineering

With a different developer, I again encountered this "process." After a few iterations, we arrived at a working solution. I wanted to figure out why it worked; the developer wanted to move on. When I protested, the developer replied, "Hey, programming isn't science. It's more like magic or alchemy."

Software developers have debated for at least a decade whether programming is alchemy, craft, science, or engineering. Peter DeGrace and Leslie Stahl convincingly argued that programming is not a science in *Wicked Problems, Righteous Solutions*.[1] I believe programming is far too comprehensible and repeatable to be considered alchemy—most developers would agree that it is a craft, but fewer would go so far as to say it's engineering.

The programmer's image may not evolve from craftsman to engineer any time soon, but we take a giant step if we refrain from contributing to the perception of programming as alchemy. As programmers we sometimes run into the mysterious and seemingly unexplainable, but it's the failings of individuals that make programming seem mysterious. Every time we fail to understand a problem or its solution, we undermine programming as a respectable craft or engineering field.

Focus on Process, Not Tools and Tricks

During my internship I often observed and experienced the pull toward tools and tricks instead of process. This tug-of-war manifests in technical staff 's comments about how much better, easier, or faster they could do their job if they had a certain tool or trick. The implication is that the staff 's woes arise from low product quality, difficult development, or slow progress. Their claims may have some validity, but it's dangerous to view tools and tricks as panaceas. While they may streamline the development process, trying to produce a product faster by randomly applying tools and tricks without a guiding process can actually decrease productivity and lead to chaos.

Applying an effective process to technical projects seems difficult and elusive. Although thousands of books, papers, lectures, and consulting companies address the subject, some of what I've read indicates software developers are frequently involved in poorly run projects.[2] I believe process is hard not because it's technically demanding but because it calls for, among other things, a resolved and disciplined staff, high accountability in areas that seem to never be clearly defined, and effective communication. Process requires an understanding of abstract issues that change for each project undertaken and each staff member utilized. In short, any development process is primarily an interpersonal undertaking, not a technical one.

Management therefore plays a crucial role. Most facets of sound process are management's responsibility—in my experience, the preoccupation with tools and tricks seems to begin when management (or technical staff, or both) is unaware of, uninterested in, or opposed to development process due to the misguided belief that any process, particularly one involving documentation, will hinder productivity. The heavy focus on doing rather than planning makes new tools seem like "silver bullets"[3]—ideal solutions that will solve their most pressing woes.

Modern development tools don't help matters.

New tools offer all sorts of bells and whistles, and many practically write programs for you to propagate the illusion of higher productivity. Often, after throwing money into new tools, the staff realizes that they still suffer the same development woes. Unfortunately, the perceived solution—to buy more and better tools—instigates a costly cycle of capital investment that yields little change in productivity or quality.

Process failure is the best way of discovering which process paths need to be better defined.

Process will probably fail before it succeeds

The company's rapid growth had created a large, sudden influx of new employees. Since the company's IS department lacked organization and standards, new IS personnel were often used poorly, resulting in occasional chaos. Realizing that the old ways were no longer good ways, the IS director set out to reorganize the department. Because most senior staff seemed to have little time to contribute or interest in the reorganization efforts, I decided to feed the IS director any information I could find about department reorganization, software development process, and standard development practices. This information—mostly from books—was very well received, and the IS director soon sought my opinions on a wide variety of issues. As you may imagine, this was a rare opportunity for an intern.

The IS director's efforts to introduce a more formal development process faced a major obstacle: people are usually apprehensive of changes in the workplace, and may even be openly hostile. Staff might seek to undermine the new process by aggressively pointing out its flaws, suggesting that minor changes to the old way would be enough, and even covertly subvert the new process to try to illustrate its inadequacies.

Along with the challenges of gaining staff buy-in, the first attempt at a new process won't generally result in a *good* process. Process strives to free technical staff from having to think too much about the everyday activities involved in intra- and interdepartmental communication. Since communication involves more than one person, simple, effective solutions don't usually originate from one person (that is, management) on the first attempt. Good solutions evolve over time, and many people contribute to them.

Given these obstacles, a new development process seems more likely to fail than to succeed. I would go so far as to offer that a draft process *must* fail before it succeeds: process failure is the best way of discovering which process paths need to be better defined. Also, a failure in process can often be attributed to the company's changing needs, and most of us would agree that companies must evolve to remain successful.

All processes are not created equal

When a process fails, this may indicate that it addressed the wrong issues or the wrong environment, or was presented in the wrong format. The success of a process depends heavily on how well it fits the environment it is released into.

This became clear to me as I learned about the programming an IS department does versus that which software engineering strives for. Most IS programming tasks are short, with very restrictive deadlines. Applying a "real" software engineering process to an IS department would introduce so much documentation, design, and development overhead that it would cripple the technical staff's ability to respond adequately to information requests.

However, many "real" software engineering process concepts could prove useful, even desirable, to IS development. The trick is to mold concepts from software engineering, management, and/or technical communication into effective, efficient, and (hopefully) well-received solutions. Applying staff-generated ideas and solutions, along with a good dose of documented process analysis from books and articles on how others solved similar problems, can yield good process solutions for most companies.

Avoid "automation mania"

The company's IS department faced many problems that stemmed from never saying "no" to a request. Although this created the perception of a "can-do," miracle-working IS department, many requests did not appropriately use IS personnel. High-tech environments seem to spawn the temptation to automate everything. Too often technical staff run around determining whether they *can* do something without asking whether they *should* do it. Creating elaborate solutions can be fun but diverts focus from more essential questions such as, Is the task worth automating? Can the automation be cost justified? Who will be responsible for the task after automation? If the automation frees up company resources, where will they be redirected? By examining the effects of a suggested automation relative to use of company resources, we can prevent automation of activities that don't need it.

What does the customer really want?

One of the first things I was supposed to do when I got a new assignment was meet with the staff member who requested service. I often felt frustrated communicating with nontechnical staff members. It's a common misconception that customers have no idea about what they really want in a software product—in fact, users often know what they want a software product to do but have no idea how to communicate their desires to help developers deliver a solution. Developers must extract the technical content from nontechnical communication so that they can create some specification on which to build the product. This complex communication process is what makes the identification of product specifications so hard.

Look for simple solutions

One subsystem in the company's software system proved particularly complex, and none of the developers wanted anything to do with it. At one point, a senior developer mentioned that it was needlessly complex and should have been scrapped and rewritten a long time ago.

I've heard of the gift of genius being attributed to people who present the simplest solution to a complex problem. Genius or not, there is much to be said about simple solutions: they usually yield less code to maintain, are easier to understand, are simpler to document, are quicker to implement, and have low technical overhead (no special tools or languages, for instance).

The Quest for Productivity

During my internship, IS staff felt they could dig out of our backlog if the team could simply produce more. Productivity is an elusive, misunderstood beast—both management and technical staff strive to produce more faster, and many people mistake high productivity for good productivity. Well, I don't believe it, and I'm not alone. Authors from Frederick P. Brooks to Steve McConnell have written about productivity,[2,3] and emphasize that *good* (quality) does not stem from *more* and *faster*. Further, they imply that *more* and *faster* actually oppose *good* when pursued individually.

In fact, many software professionals believe that the pursuit of quality is the key to higher productivity. Yet some still try to improve productivity by shortchanging quality assurance activities. They mistakenly believe that QA activities such as specification, architecture, design documents, and reviews of those documents unnecessarily waste time and don't contribute to producing the "real" software product. To the contrary, QA activities clarify and solidify the work to be performed by the technical staff. This minimizes the need for rework; less rework means less overall time spent on any single project, which means higher productivity.

Suggesting means volunteering

Once I started contributing to reorganization efforts, I ended up volunteering to implement my suggestions every time someone wanted to pursue them. This is reasonable since the person who made the suggestion best knows its intent. On the other hand, staff may not feel compelled to contribute good ideas because they don't want the responsibility of implementing their suggestions.

It's obviously unfortunate if technical staff feel penalized for contributing ideas. To remedy this organizational practice, perhaps the group leader might assign the person with the suggestion to a group of staff members with the skills necessary to effectively analyze and act on the idea. Perhaps the group could be composed of staff who have a direct interest in the suggested changes. In any case, the last thing a company should want to do is discourage the presentation of new ideas.

Quality assurance minimizes the need for rework, which means higher productivity.

Maintenance: tag, you're it

Very soon after I made my first correction to a piece of the company's software system, I found myself correcting more new errors in the same piece of software. The new corrections weren't needed to fix errors created by my previous alterations; they were necessary because of entirely different bugs. After several iterations of this cycle over a few months with the same piece of software, I wanted someone else to work on it—since when did I become the resident expert on this piece of code? The person who wrote it is still on staff—why doesn't he fix these errors?

Coworkers appreciate effective communication skills more than many other technical skills.

The answer to my queries was, "Hey, you touched it last." "Touching" software means altering it. Since all source code at the company was stored in a code management system, each revision is tagged with the name of the person who last checked it in, which makes it simple to determine who touched a program last. Before long I realized that this practice was more like a department policy: most of the senior development staff had large portions of the company's software system that they "touched" often, and as a result, they'd been responsible for making corrections to these systems for so long that the systems were like permanent chains of torment attached to them for eternity.

My experiences at the company provided little support for this method's effectiveness. Granted, the last person who touched a program probably has the freshest perspective on the code, which will allow the developer to make new changes more rapidly. But this depends heavily on how well the developer really tried to understand the program he corrected. Another lure of this software maintenance technique is that work assignments are easy to hand out since the same person usually gets all the problems for the same systems.

This method's shortcomings outweigh its benefits. First, there's no distribution of intelligence through the department concerning the software systems maintained—only one person, or a small group of people, really has a reasonable grasp of any single program. Further, because developers are stuck with maintaining the same pieces of software over time, and since they have such "specialized" and "valuable" knowledge, they have limited opportunity to be involved in new development due to the time needed to maintain "their" systems. The ultimate blow comes when a senior developer leaves the company, taking his "specialized" and "valuable" knowledge with him.

Finally, such a method may make developers unwilling to work on different software systems because each one they touch becomes a new link in their chain of torment. Obviously, senior developers catch the brunt of the torment. They are responsible for corrections to the largest portions of the software system, they have limited opportunity for involvement in new development projects, and they also lose their intra-department mobility. They cannot effectively change positions or responsibilities since the chains of previous software involvement hold them fast as resident "experts" with "their" software systems.

Any solution to such problems will be difficult to implement, but certain department practices could alleviate the burden of developers' "software system chains." First, a team approach to software system maintenance would help disseminate system knowledge. A maintenance team could consist of a pair of staff members who work together to resolve maintenance issues. Teams would not be fixed but would pair people based on availability or random rotation—pairs could be senior–junior, senior–senior, or junior–junior as the maintenance activities allow. The point is that the problem, analysis, and solution knowledge are shared. The maintenance task may be prolonged somewhat due to communication efforts between the staff members, but the result would be a higher level of cross-training between software systems, and increased development staff mobility.

The other practice is, of course, documentation. Senior staff or maintenance teams should take time to compose system documentation for the benefit of other developers and themselves—this could remove a link from their software maintenance chain.

Communication: keep it clear

Good communication skills are essential in a technical environment, and most of my successes stemmed from good communication. Certain practices seemed most useful:

♦ Mentally plan what you're going to say. Though impromptu speech often prevails in casual conversations and even meetings, it can really hurt communication when complex ideas come out confused.

♦ Avoid technical jargon where possible, but if you must use technical terms, be sure everyone involved shares similar interpretations.

♦ The phrase "a picture is worth a thousand words" particularly applies in a technical environment—one good diagram can save much verbal and documentation energy.

♦ Actively listen. Work to grasp the speaker's understanding of the topic. Ask questions to clarify what you don't understand. I've often left a conversation believing I understood a problem only to find it was still unclear when the time came to implement a solution.

♦ Finally, don't interrupt—it confuses the issue, locks people out of the conversation, and is simply rude.

Coworkers appreciate effective communication more than many other technical skills. Improving your communication skills will greatly benefit you and those you work with.

Fight for what you believe in

As an active contributor to department reorganization efforts, I constantly had to defend my ideas. I quickly learned that fighting for something in a corporate environment can be exhausting, so pick your battles carefully—the only ones you stand a chance of winning are those sparked by a violation of your personal, professional, or technical beliefs. Leave fighting for what's "right" to philosophers, lobbyists, and upper management.

A corollary to this lesson: continually examine what you believe. Be ready to compromise, and know ahead of time what compromise means to you. When necessary, admit when your perspective is wrong, graciously accept defeat, and agree to disagree with your opponents. Give your opponents every chance to present their point of view, and strive to examine beliefs that differ from yours. Professional and technical issues are complex; what's "right" is a matter of perspective. Your reputation will be based on your demonstrated beliefs.

A Healthier Small IS Department

One of the IS director's primary concerns was establishing a foundation for continued department health. During my year at the company, I formulated some ideas about the characteristics of healthy small departments.

Bolster weakness with process

Human beings make mistakes, procrastinate, avoid responsibility, engage in petty bickering, gossip, and participate in many other nonconstructive activities. We're not entirely corrupt; we just need direction, constraints, and focus. Enter process.

A well-defined process for completing work assignments provides direction, constraints, and focus for a group of professionals working toward common goals. The primary goal of process is to free individuals from organization-related decision making so they can direct energy to solving the problems at hand.

Once a work process is established, it must remain flexible to remain effective. As situations arise that require changes, a procedure should be available to submit, evaluate, and implement alterations to the work process. When a work process change is submitted, the group and management should evaluate the suggestion and, if they accept it, make staff assignments to begin implementing the change. Once the new process item is in place, allow some time to go by and then evaluate its effectiveness.

A common mistake I witnessed was that the group and management failed to actively develop, specify, and evaluate a work process' standards, conventions, procedures, policies, and implementation on an ongoing basis. This stems from a mistaken belief that a work process can remain unaltered over time and still be effective. Teams should periodically review all aspects of the work process, even if the meeting's content is limited to "Everything is going well; let's go back to work." The meeting should stress the importance of following the stated work process and address suggested changes, recent work process problems, project problems, and other group concerns. Group members should be encouraged to contribute their input, and rewarded for finding solutions to process-related problems. In any event, the primary accomplishment of a defined work process is minimizing the chances that individuals will become victims of their innate human weaknesses.

Once a work process is established, it must remain flexible to remain effective.

Strengthen the team by sharing knowledge

A group guided by common principles under a defined work process benefits even more by limiting specialization of its members' knowledge. Topic familiarity, job title, and work responsibilities require some specialization, but generally, no group member should have significantly more knowledge about any item within the working environment than another member with an equivalent background, position, and responsibility. Group members need not have equivalent knowledge bases, simply comparable ones.

Several ways exist to distribute knowledge, such as pairing (assigning two or more people to a task), presentations, educational documents, and, of course, system documentation. Group members and management should actively seek opportunities to share knowledge with others—from simple word processor tricks to complex system architectures.

Without knowledge of current methods and tools, computer professionals quickly drift toward obsolescence, limiting their options.

Increase your value by learning and improving

Most computer professionals are aware of the intense need to refine their practices and remain abreast of new methods and technologies, but many do not act on it. It's very easy to fall off the technology wagon, especially as technical professionals advance toward and into management positions—unfortunately, many never recover after they fall off, in part due to the rapid pace of technology evolution. Without knowledge of current methods and tools, computer professionals quickly drift toward obsolescence, thereby limiting their options for advancement and employment. I've quickly realized that there are few if any free rides in computer-related professions.

Not only does the failure to stay current affect professional and personal growth, it also affects a company's growth. Companies that lack current professionals cannot take advantage of new technologies to increase productivity, reduce operating costs, and move into new markets. Although this may or may not threaten a company's market position, it can certainly affect profit margins since older technologies frequently incur larger costs due to the lack of available knowledgeable staff and supporting tools.

To solve this problem, management must increase efforts to expand staff 's training and education opportunities. It can do this by openly offering opportunities to learn new methods and technologies from seminars, educational institutions, and literature. The company should also aggressively pursue inhouse education and training, perhaps asking more knowledgeable staff members to conduct presentations or training seminars in their areas of expertise.

Education really shouldn't be optional—employees should be required to demonstrate some new technically relevant (though not necessarily job-related) knowledge they've acquired through seminars, classes, or literature at least once a year. This "demonstration" could be accomplished by the completion of a document, presentation, or project that applies the new method or technology. Since group members ideally share knowledge, the entire staff stands to benefit when one person acquires new knowledge. Any staff of reasonable size has immense learning opportunities—just imagine how much information is floating around among your technical staff right now, knowledge currently unshared.

The two sides of job-related experience

The single greatest lesson I learned concerns a trait that management seems most to desire: experience. Although perhaps not the industry norm, some of my dealings with "experienced" professionals have been disappointing—sometimes experience can be a dangerous thing. The commonly held belief that "experience is the best teacher" takes on new meaning when you consider that the lessons being taught might be bad ones.

Additionally, experience-based lessons tend to stick better than those from readings, examples, instructions, and seminars. A highly experienced professional could easily impart many "sticky" bad habits, erroneous information, and warped perspectives to novices. I've encountered experienced professionals who can't (or won't) see past their experience to notice that the world around them has changed, or that an idea that runs against their experience might actually be a good one.

Of course, experience has much potential value. Experienced developers seem sensitive to what might violate basic good architecture, design, and programming practices. I've seen experienced developers raise warnings over issues I saw as mundane only to find that the direction I was headed would have led me into some serious problems. Experience also imparts strong problem-solving skills. Exposure to many different problems over time gives developers a large toolbox of techniques for decomposing and solving all sorts of problems.

Although I occasionally tried to “empty” some experienced developers’ toolboxes, I never did hit anything that remotely resembled a “bottom.” Experienced developers usually have better communication and organization skills, too, delivering their points efficiently and clearly. This makes them better able to obtain agreement and rally peers around an idea.

Finally, experienced developers seem to suffer less from novices from functional fixedness, that is, the inability to recognize a use for an item or concept other than that originally presented. This characteristic seems to afford them much creativity and the ability to connect seemingly disjoint concepts.

Overall, though, I’m still suspicious of experience—it could be restrictive if stringently adhered to. If seasoned developers commonly use their “experience” to reject new, different methods, perhaps we should reconsider the importance we place on this concept, sometimes to the detriment of innovation and fresh perspectives.

I learned a lot during my year in the software industry—some good, some perhaps not so good. As I gain more experience, I’ll be interested to see how my conclusions hold up. At the very least, I hope this article reminds some professionals of a few things they may have been overlooking since their rise in the industry’s ranks. ❖

References

1. P. DeGrace and L. Stahl, *Wicked Problems, Righteous Solutions: A Catalogue of Modern Software Engineering Paradigms*, Yourdon Press, Englewood Cliffs, N.J., 1990.
2. S. McConnell, *Rapid Development*, Microsoft Press, Redmond, Wash., 1997.
3. F.P. Brooks, *The Mythical Man-Month*, Addison Wesley Longman, Reading, Mass., 1995.

About the Author

Ryan Fleming is a senior in the Software Engineering Technology program at the Oregon Institute of Technology, and expects to earn a BSc in June 1999. In 1997, after completing his junior year, Fleming took a one-year internship as a programmer with a telecommunications company. His interests span a wide range of software engineering techniques and technologies, particularly software project management, object-oriented analysis and design, and programming languages.

Readers may contact Fleming at 1331 Avalon St., Apt 17, Klamath Falls, Oregon 97603; e-mail flemingr@internetcds.com or flemingr@oit.edu.