Software Engineering Institute

The *Watts New?* Collection: Columns by the SEI's Watts Humphrey

Watts S. Humphrey

November 2009

SPECIAL REPORT
CMU/SEI-2009-SR-024

Software Engineering Process Management (SEPM) Program
Unlimited distribution subject to the copyright.

http://www.sei.cmu.edu

Carnegie Mellon

Table of Contents

List of Figures

List of Tables

Abstract

Since June 1998, Watts Humphrey has taken readers of *news@sei* and its predecessor *SEI Interactive* on a process-improvement journey, step by step, in his column Watts New. The column has explored the problem of setting impossible dates for project completion, planning as a team using TSP, the importance of removing software defects, applying discipline to software development, approaching managers about a process improvement effort, and making a persuasive case for implementing it. After 11 years, Watts is taking a well-deserved retirement from writing the quarterly column. But you can still enjoy vintage Watts New columns, including all of the above topics, in the *news@sei* archives (http://www.sei.cmu.edu/library/abstracts/news-at-sei/) or in the Watts New Collection.

Introduction

Since June 1998, Watts Humphrey, perhaps the best-known member of the SEI’s technical staff, has taken readers on a process-improvement journey, step by step, in his column *Watts New?*

These columns for *news@sei* have explored the problem of setting impossible dates for project completion (“Your Date or Mine?” on page 11), planning as a team, using TSP (“Making Team Plans” on page 17), the importance of removing software defects (“Bugs or Defects?” on page 23), applying discipline to software development (“Doing Disciplined Work” on page 29), approaching managers about a process improvement effort (“Getting Management Support for Process Improvement” on page 33) and making a persuasive case for implementing it (“Making the Strategic Case for Process Improvement” on page 37).

In a column in this volume, Watts presents an example of a process improvement proposal—complete with the numbers to back it up (“Justifying a Process Improvement Proposal” on page 43). We don't want to give away the ending but the five-year savings are about $10 million and the five-year return on investment is 683%!

We think these columns are an important contribution to the software engineering literature. As such, we have collected them into this complete set, which you can download as a PDF file. We hope that having The Watts New? Collection in one volume will make it easier to implement software process improvement in your organization.

1 Why Does Software Work Take So Long?

June 1998

In writing this column, I plan to discuss various topics of interest to software professionals and managers. In general, I will write about issues related to engineering productivity, quality, and overall effectiveness. Occasionally, I will digress to write about a current hot item, but generally I will be pushing the process improvement agenda. Because my principal interest these days is getting organizations started using the Personal Software Process (PSP) and Team Software Process (TSP), readers should know that a not-so-hidden agenda will be to convince them to explore and ultimately adopt these technologies.

Have you ever started what you thought was a two- or three-day job and have it stretch into a week or two? Before deciding you are just bad at estimating, look at how you spent your time. You will find you spend much less time on projects than you imagine. For example, on one project, several engineers used time logs to track their time in minutes. They averaged only 16 to 18 hours a week on project tasks. They were surprised because they all worked a standard 40-hour week.

This information soon turned out to be helpful. They were on a critical project and were falling behind. When they looked at the data, they found the design work took 50% longer than estimated. They knew they had a choice: either do the tasks faster, or put in more time. While there was pressure to race through the design, skip inspections, and rush into coding, the team resisted. They knew this would probably result in many errors and a lot of test time.

To meet their schedule, they needed to average 30 task hours a week. They all tried valiantly to do this, but after Christmas, they realized that just trying harder would not work. They went on overtime and are now starting early in the morning, working late some evenings, or coming in on weekends. While they now average 30 task hours a week, they have to work over 50 hours a week to do it. They are also back on schedule.

Because this team had detailed time information, they could recognize and address their problem in time to save the project. The data identified the problem and pointed them toward the solution. Without good data on where your time goes, your estimates will always be inaccurate and you won't know how to improve.

Working Harder

When people say they are working harder, they actually mean they are working longer hours. Barring major technology changes, the time it takes to do most tasks is relatively stable. The real variable is the time you spend on the tasks. But to manage the time you spend, you have to track it, and practically nobody does. Consider the following:

1. Our lives are filled with interruptions.
2. Software people do many kinds of tasks, and only some contribute directly to our projects.

3. Our processes are often informal and our working practices ad hoc.
4. Even if we wanted to, it is hard to do demanding intellectual work for long uninterrupted periods.

Interruptions

One engineer told me she had recently started to track her time and found she was spending much more time on interruptions than on her real work. For example, on one task of 108 minutes, her interruption time was over 300 minutes. This lost time, however, was not in big hour-long blocks but from an incessant stream of little 5- and 10-minute interruptions.

Interruptions come from many sources:

- telephone calls
- other engineers asking for help
- a coffee or rest break
- supply problems (i.e., printer or copier out of paper)
- equipment problems (the network dies)
- a power failure or a snow storm (everybody leaves)

Every interruption breaks your train of thought and takes time away from your work. With unplanned interruptions, you lose your place in the work and, when the interruption is over, you have to reconstruct where you were. This also causes errors.

For example, when I am in the middle of a design, I am often working on several things at the same time. While thinking through some logical structure, I realize that a name is misleading, a type must be changed, or an interface is incomplete. If I am interrupted in the middle of this, I often have trouble remembering all these details. While I have been unable to control the interruptions, I have found that maintaining an issue log helps me remember what I was working on when interrupted.

Non-Project Work

Most engineers also spend a lot of time on non-engineering tasks. Examples are

- handling mail
- setting up or updating their computing systems
- going to meetings
- looking for some specification, process, standard, or manual
- assisting other engineers
- attending classes

Few software development groups have adequate support. No one sets up or maintains his or her development system, few have groups to handle product packaging and release, and there is no clerical or secretarial support for mail, phone, or expense accounts. What is more, even when they have such support, many engineers don't know how to use it. This means that most of us spend more time on clerical and support work than on our principal development tasks. And every hour spent on these tasks is an hour we can't spend on development.

Lean And Mean Organizations

Often our organizations pride themselves on having very small support staffs. An almost universally accepted management axiom is that overhead is bad and should be eliminated. In the resulting lean and mean organizations, the engineers do their own clerical work. This is not an effective way to use scarce and expensive software talent.

By cutting overhead, management also eliminates the support staffs that funds in the overhead budget support. While some of these groups are not the least bit interested in supporting the engineers, many are. Eliminating them can have enormous costs. Among these costs is the time every engineer must spend sorting through email, answering the phone, getting supplies, doing expense accounts, and filing mail and documents. In addition to the lost engineering time, this also means that most mail is not answered promptly if at all, phones go unanswered, supplies are wasted or overstocked, and little if anything is properly filed or can be quickly found when needed.

Perhaps most expensive and annoying, every software engineer in such “lean and mean organizations” must set up and maintain his or her personal computing environment. Just because we have the skills to do this doesn't mean we should. Most of us could repair our cars or paint our houses if we chose to, but it would take us longer than using someone who does this for a living. And we have other things to do. Why should we have to handle our own computing support? The principal reasons that engineers spend less than half their time doing the tasks they were trained and hired to do is that, without adequate support, they have to support themselves. What is more, few engineers enjoy or are very good at being part-time clerks and technicians.

Ad-Hoc Working And Planning

When no one has taken the time to define and document the organization's practices and methods, they must be maintained informally. When you come to a task that you haven't done before or at least not recently, you look around to see how it should be done. It takes time and a lot of interruptions to find someone with the right experience and get their help. While this is vastly preferable to bulling ahead without exploring prior experience, it does cut into the working week.

A related but slightly different problem concerns planning. When projects don't make detailed plans, and when engineers don't know precisely where they fit into these plans, they must do what I call continuous planning. In continuous planning, the key tool is not the PERT chart or Microsoft Project, it is the coffee machine. When you finish a task, you go to your continuous planning tool to have a cup of coffee. While there you decide what to do next. In the process, you talk to any other engineers who are also doing continuous planning and see what they think. You will usually get some good ideas, but if you don't, you can always interrupt someone.

The common view is that planning takes too much time. By not planning, engineers can immediately start on their programming work. While this immediate progress will feel good, you won't know where you are. Like driving in a strange country without a map, you have to stop at every turn and decide where to go next. All these short stops will take far more total time than a properly thought-out plan would have taken in the first place.

You Also Need An Occasional Break

Finally, creative development is hard work. When designing a product or a system, we need uninterrupted time. But we cannot design complex products for more than a few hours at a time. The same is true of testing, reviewing, inspecting, coding, compiling, and many other tasks.

One laboratory decided to set up a dedicated group of experts to inspect a large and important product in final test. Every module that had test problems was sent to this group. For a while, they cleaned up a lot of defect-prone modules. Then, one of them later told me, they could no longer see the code they were inspecting. Everything began to blur. They even saw source code in their sleep.

Designing, coding, reviewing, inspecting, and testing are intensely difficult tasks. To have any hope of producing quality products, we must occasionally take breaks. But, to be reasonably efficient, and to do high-quality work, we need to control our own breaks, not take them when the phone rings or when somebody barges into our office or cubicle. Studies show that when engineers spend all their time on their principal job, their performance deteriorates. Some reasonable percentage of time on other tasks such as planning, process improvement, quality analysis, or writing papers can improve engineering performance. You will get more and better work done in the remaining 75% of your time than you would have accomplished in 100% of dedicated time.[1]

So, Keep Track of Your Time

To manage your personal work, you need to know where your time goes. This means you need to track your time. This is not hard, but it does require discipline. I suggest you get in the habit of using the time recording log, shown in Table 1-1 and Table 1-2. When doing so, enter the tasks and the times when you start and stop these tasks, and also keep track of interruption times. If you do this, you will soon see where your time goes. Then you can figure out what to do about it.

Manage Interruptions

Next, interruptions are a fact of life, but there are many ways to deal with them. Use “DO NOT DISTURB” signs and establish an ethic where everybody (even the managers) respects them. Forward phone calls or even unplug or turn off the phone. Also consider getting permission to work at home for a day or two a week.

[1] For a brief discussion of this issue, see *Managing Technical People, Innovation, Teamwork*, and *The Software Process*, Addison Wesley, 1997. A more complete discussion is in Donald C. Pelz and Frank M. Andrews, *Scientists in Organizations: Productive Climates for Research and Development*, Wiley, 1966, pp. 56, 65.

Another way to manage interruptions is to get in the habit of using an issue-tracking log. Then, when you think of something you need to do, make a note of it in the log so you will remember to do it later and you won't forget it when the phone rings. While you will still have to handle these issues, you are less likely to forget them and you can do them at a planned time.

Also, use this same principle with interruptions. When someone calls in the middle of a design problem, tell them you'll get back and then make a note on a sticky so you don't forget.

Table 1-1: Time Recording Log[2]

Engineer		Date	
Program		Module	

Date	Start	Stop	Interruption Time	Delta Time	Phase	Comments

2 Concerning the Recording Log, see my book *A Discipline for Software Engineering*, Addison Wesley, 1995. This log is also discussed in *Introduction to the Personal Software Process*, also by me and published by Addison Wesley in 1997.

Table 1-2: Time Recording Log Instructions

Purpose	• Use this form to record the time spent on each project task. • Either keep one log and note the task and product element for each entry or keep separate logs for each major task.
General	• Record all the time you spend on the project. • Record the time in minutes. • Be as accurate as possible. • If you need additional space, use another copy of the form.
Header	• Enter the following • your name • today's date • the project name • the name of the program or other • product element • If you are working on a non-programming task, enter the task description in the comments field.
Date	• Enter the date when you made the entry.
Example	• 4/13/98
Start	• Enter the time when you start working on a task.
Example	• 8:20
Stop	• Enter the time when you stop working on that task.
Example	• 10:56
Interruption Time	• Record any interruption time that was not spent on the task and the reason for the interruption. • If you have several interruptions, enter their total time.
Example	• 37—took a break
Delta Time	• Enter the clock time you actually spent working on the task, less the interruption time.
Example	• From 8:20 to 10:56, less 37 minut4es or 199 minutes
Phase	• Enter the name or other designation of the phase or step you worked on.
Example	• planning, code, test, etc
Comments	• Enter any other pertinent comments that might later remind you of any unusual circumstances regarding this activity.
Example	• Had a requirements question and had to get help.
Important	• Record all worked time. • If you forget to record the starting, stopping, or interruption time for a task, promptly enter your best estimate.

Learn to Use Administrative Support

Learn how to use support. While few engineers have a support staff to help them, many who do don't know how to use them. If you have a support person, think about every clerical-type task before you do it. Can this person do it for you? Even though it may take longer at first, use them whenever you can. At first the result may need to be redone. But be patient and help the support people understand your problems with their work. It will pay off in the long run.

Plan Every Job

Perhaps most important, learn to plan. Plan your own work and urge your teammates and the project leader to start planning. Proper planning takes time, but it can save much more time than it costs. You will end up planning anyway, but it is much better to do it in an orderly way, and not at the coffee machine.

Vary Your Work

You can do demanding work only for so long. I lose my ability to do intense creative work after an hour and a half to two hours. I need to stop for a break or even to switch to some other kind of work. Further, during these intense sessions, frequent short interruptions offer no relief. It then takes an extra effort to reconstruct my thought process.

What I suggest is to intersperse various kinds of work throughout your day. Do creative work when you are most fresh and productive and then switch to your email or an administrative task. Then perhaps do a design or code review possibly followed by a process-improvement task or data analysis. By varying the task types, your creative work will be of higher quality and you will actually get more done.

Define and Use a Personal Process

When you regularly make plans, a defined process will save a lot of time. The process provides a framework for gathering historical data and a template for making plans. And, by using historical data, your estimates will be more accurate.

Get and Use Historical Data

Finally, if you don't have administrative or technical support, use your time log to see what this lack costs you. Then tell your managers and show them your data. It might help them see the cost advantages of adequately supporting their engineers. Remember that the amount of work you produce is principally determined by two things:

1. the time the tasks will take
2. how much time you have available for these tasks

To manage your work, you must know where your time goes. Only then can you judge how much work you can do and when you will finish it.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful suggestions I received from Dan Burton, Alan Koch, and Bill Peterson.

2 Your Date or Mine?

September 1998

Congratulations, you've just been promoted. You get to lead the new project we just won. You will have six engineers, but two of them are half time for a month or two. You can hire four more. The delivery date is nine months.

What do you say?

Most engineers would say, “Gee thanks boss, I've always wanted to run a project and this sounds like a great opportunity. I'll give it my best shot, but the date looks awfully tight.” If that's your answer, you lose!

Who owns the nine-month date? When your boss offered you the promotion, whose date was the nine months? It was the boss's date. But, when you said, “Boy that's a tough date, I'll do my best to meet it,” whose date was it then?

Yours!

And don't ever forget it! You have just bought the ranch. Even though you had no intention of doing so, and you didn't even have time to think about it, bang, it hit you out of the blue. And there you are, the proud owner of a budding disaster.

So what else could you do? It turns out there is plenty you could do.

Getting Into Trouble

Projects usually get in trouble at the very beginning. They start with impossible dates, and nobody has time to think, let alone do creative or quality work. All that counts is getting into test, and the rush to test invariably produces a hoked-up product, a poor quality program, a late delivery, an unhappy customer, and disgruntled management.

While the promotion looks good at first glance, it is only good news if you handle it right. To have any chance of a successful project, there are things you must do right now. Before we talk about what to do, however, let's discuss the causes and consequences of this all-too-common situation.

Pressure

The only way to manage development work is with pressure. Managers know that relaxed projects rarely succeed. Projects can get into trouble by endlessly studying alternatives, not making decisions, or loading up a product with nice features. These all seem like good ideas at the time, but without pressure months can go by and nobody notices.

So expect pressure. Managers know they must push, and you can expect them to keep pushing, at least if they are awake and doing their jobs. Their only questions are

What dates will they push for?

How hard should they push?

The Problems With Pressure

Schedule pressure causes all kinds of counterproductive behavior. People don't plan their work, they rush through their designs, and they don't review their products. The big push is to demonstrate progress to management, and the only thing management recognizes as progress is getting into test.

Unfortunately, few managers really understand that this is the worst possible thing they could push for. Most software engineers know that racing to throw a product into test is a mistake but they don't know how to fight the pressure. Then they feel compelled to rush through requirements and design and to skip everything else but code and test. When they do, they know what will happen. And of course it does.

While you can always say management was unreasonable, you will be responsible for being late and producing a poor-quality product. Everyone can easily blame somebody else, but do you really want to spend your life this way?

I Told You So

You know intuitively when a schedule is too aggressive. You have a queasy feeling in the pit of your stomach. You can sense the dates are wildly optimistic, and you know this is a disaster just waiting to happen. So you tell the manager, “That schedule looks awfully tight to me,” and the manager responds, “But that is the date in the contract, or that's the date the customer demands, or that's the date the boss committed.” So you say, “Ok boss, if you say so, I'll try but don't be surprised if it takes longer.”

Now that you told the boss, if there are problems, you can always say I told you so. Unfortunately, that won't help. The minute you walk out of the room, it is your date. You may have been worried, but you took the job didn't you? Why did you take the job if you didn't think you could do it? If you don't settle the issue right now, you will be the goat, no matter what you say about the date.

Negotiating Schedules

So you must take a stand. Most engineers are so focused on the job that they don't think about what the manager is saying. When managers say the delivery date is nine months, they are making a bid. And you bought it without a counter offer. You'd never buy a house or a car or a boat this way. You'd debate the number.

Think about it this way. Management has just said, “We have this key project, and the best date we think you can make is nine months.” If you don't counter with another date, they will hold you to nine months. Unfortunately, if you just guess a later date, they will ask you why. And if you don’t have a good answer, they will either ignore your date or get somebody who won't argue.

Management doesn't know how long the job will take, and neither do you. If you knew, and if you could convince them you knew, they would accept your date. If the project really will take 12 months, the last thing most managers want is a commitment to deliver in 9 months. You work for them, and they will also be held accountable for your schedule. They could easily lose their jobs if your project fails, and all you would lose is a chance to have another disaster. That would probably be a relief, at least after this project.

Handling Pressure

You must start by convincing management that you know how long the project will take. To do this, however, you must know yourself. This, it turns out, is not very difficult. It takes time and some hard work, but when engineers make careful estimates, and when they use historical data to make these estimates, they are generally pretty accurate.

The way you determine the date is to make a plan, and to make a very detailed plan. Since this can take a lot of work, and since you want your team to be part of this planning process, you need to get your new team to help you. This actually is the best possible approach. It will not only produce the best plan, but the team will then be in a far better position to do the work. They will also be committed to the schedule.

Then, when you have the plan, go back to management and tell them what the date really is. When they argue with you, as they will, take them through your plan. Show them as much detail as they will sit still for. Walk them through the *numbers*, and the *task lists*, and the *historical data* you used for comparison. Talk about *product sizes* and *productivity rates*. Show them enough to really convince them that you know what you are talking about.

Be Flexible But Firm

Once you have made the sale, and management accepts your plan, stop selling and move on to the next subject. They will probably want to talk about alternatives. What would the date be with more staff, or reduced requirements, or a phased-version delivery plan? They may want you to present the plan to higher management, or to the customer. In fact, you will probably have to walk more people through the plan, so keep it dusted off, and make sure your backup is solid. Expect people to find any chinks or inconsistencies. Remember though, these are estimates. So tell them what you think and why, but remember that no one knows as well as you do how long the job will take.

If anyone can convince you that your estimates are off, be willing to make adjustments. As long as they have actual historical data to back up their opinions, and as long as these data are relevant to your project, consider the new facts. Under no conditions, however, make any such decisions on the spot. Any schedule change requires careful study and team agreement. So don't change your estimates without data and the time to review them with your team. Remember, almost all initial plans are tight, so don't cut your schedule without a very good story that the team agrees with.

Answering Management

So, when management says the date is nine months, the way to answer is to say, “That looks like a great opportunity boss. Let me get the team together and we'll take a look. We'll make the best plan we can, and be back to you in a couple of days.”

If you think the schedule will be longer than management wants, don't argue about it right now. You don't have the ammunition to win that argument, and all that would do is convince management that you have a bad attitude. If they think you are out to prove their date is wrong, they won't let you make the plan. Remember, they don't believe you can give them a good date. After all, nobody has made good schedules before, so why should you be first?

So start with a positive attitude, and really drive for a better date. In fact I once had a team come back with a five-week better schedule than management had asked for, and they're still holding to their plan. So give it an honest try, but then get the data to defend it.

How Does This Work

I have coached many development teams on how to do this, and it always works. Of course, we have started by softening up management, and we have been there to help at the beginning. But teams are surprised at how well this approach works and, after a good start, they can usually continue working this way. Management also quickly discovers that informed and committed teams do vastly better work. While most plans come in with longer dates than management wanted and management always asks lots of tough questions, when teams have good plans they can defend them. And when they defend their plans, they always convince management. Best of all, they end up working to their schedule not management's.

So addressing the schedule problems up front really pays off, both for the engineers and for management. While it takes management a few days to get over the shock, they will end up with a software team that knows what they are doing. These teams can report their progress against realistic plans, deliver on the agreed schedules, and produce fine products.

The Next Steps

So you need to know how to make a plan, how to present this plan to management, and then how to defend the plan when it is attacked. Then, once you have done all this, all you have to do is develop the product. But at least you will have a date that you and your team agree you can meet. And, most important, you will have a well-thought-out plan to guide the work.

While these methods are not difficult, they are not obvious. If you need help in how to do this, one place to look is at the various Personal Software Process (PSP) references that explain how to make individual plans.

The SEI teaches PSP courses (see http://www.sei.cmu.edu), and there are a growing number of university and commercial courses available. We are also introducing the Team Software Process (TSP), which shows teams of PSP-trained engineers how to handle this planning and commitment process. TSP introduction also walks teams through a launch process that produces their detailed plans and negotiates their schedules with management.

Final Comments

While this all sounds logical and simple, everybody's situation is different. If you remember two basic principles, however, you should be able to handle almost any case. First, you are paid to do what management wants you to do, so don't refuse a direct order. Second, always be willing to make a "best effort" but don't commit to a date without a plan. If management appears totally unreasonable, however, read up on negotiating strategies before you get in over your head.[3]

Most managers will be reasonable and respect your desire to make a plan, but occasionally one won't listen. While he or she could be totally unreasonable, it is more likely that higher level managers are applying heavy pressure that your manager is unwilling to buck. You should certainly try to turn such situations around, but that is normally very difficult. If you can't make headway pretty soon, get out from under your gutless manager as quickly as you can.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful suggestions from Linda Gates, Alan Koch, Warren Morrison, Mark Paulk, and Bill Peterson.

[3] Probably the best reference on this subject is *Getting to Yes*, by Roger Fisher and William Ury, Houghton Mifflin, 1981. I also summarize some key points about negotiating programming issues in Chapter 12 (Power and Politics) of my book, *Managing Technical People, Innovation, Teamwork, and the Software Process*, Addison Wesley http://www.sei.cmu.edu/library/abstracts/books/201545977.cfm.

3 Making Team Plans

December 1998

At the team kick-off meeting, management told the engineers that the company critically needed their new product in nine months. This group was introducing the Team Software Process (TSP), and I had convinced management that the team should make a development plan before they decided on the schedule. Management had agreed, and we scheduled a meeting for two days later to review the engineers' plan. Now, the 12-engineer team was assembled and ready to make their plan. They had a lot of questions.

1. How do they make a plan when they don't know the requirements?
2. How detailed should they make the plan, and how much of the project should they cover?
3. Suppose the plan doesn't finish in nine months; what do they do then?

And finally,

4. Since they knew so little about the product, could any plan they made now be useful?

These questions are the subject of this column.

How Do They Make A Plan When They Don't Know the Requirements?

The team was very concerned about the vague state of the requirements. While a couple of the engineers had a general idea of what the product was to do, they did not see how they could make a realistic plan without much more detail. I was coaching this team and pointed out that they could make an accurate plan right after final product delivery. Their plan would be most accurate then, but it would be least useful. On the other hand, they most needed a plan at the beginning of the project, but it would necessarily be least accurate. So, while they did not yet know very much about the product, they agreed to make the best plan they could produce.

Plan Accuracy

Clearly, the more you know about a product's requirements and design, the more likely you are to make a good plan. Thus, plans will always be least accurate at the beginning of a project, and they should get progressively more accurate as the work progresses. This suggests three things. First, you must plan, even if you don't know very much about the job. Second, you should recognize that the initial plans will be the least accurate. And third, you need to continually remake the plans as you learn more about the work.

How Detailed Should They Make the Plan, And How Much of the Project Should They Cover?

Planning is much like designing a product. It is a good idea to start with an overall architecture or process and then to lay out the entire structure. What development tasks are required, in what order, and how long will they take? Until you have an overall framework, a detailed plan could address the wrong tasks or focus too much effort in the wrong places.

For example, I was assigned to a project some years ago. There had been many small schedule changes, but everybody thought the project was on schedule. Nobody had ever produced an overall plan. However, when we did, we found that testing time had been dangerously reduced. The project was in serious trouble.

Without an overall plan, it is hard to see the cumulative impact of many small schedule slips. They all add up, however, and without an overall perspective, the latter phases will invariably be squeezed. So, while detailed plans are essential, they must be made in the context of an overall plan that runs from the start date all the way to the final product delivery. Therefore, the first step must be to make an overall plan.

Start With the Process, Then List the Products And Make An Estimate

Once the engineers agreed to make an overall plan, they had to decide on what development process to use. By starting with the organization's overall process framework, they defined their specific project process in less than an hour.

Next, they defined the products to be produced by each process phase. They estimated the sizes of the requirements and design documents and postulated an overall product structure. They judged what components would be required and how big each component was likely to be. Each engineer contributed to these discussions, and they compared this job with others they had worked on. It was surprising how much relevant information the 12 of them had.

Next, the team had to figure out the effort required to develop each of these products. Again, every engineer contributed his or her views. In some cases, they had real data for similar jobs. In other cases, they made overall judgments based on general recollections. In the end, they came up with estimates for every product. While some of these estimates were guesses, they were informed guesses made by experienced engineers who had previously done similar work.

Make the Schedule

The last overall planning step was to produce the schedule. Here, the engineers estimated how many hours they each had available for the project each week. Since many had prior obligations that would continue, they allowed time for this other work as well. When they were done, they had an estimate of the total hours the entire team would have available for each project week. Then they spread the work over these hours to produce the schedule.

By this time, the engineers had a pretty good idea of how big the job was. Thus, they were not surprised that the project would take much longer than the 9 months that management wanted. The full schedule actually turned out to be 18 months. At this point, the team had defined the complete process that they would use, produced a product list, made product-size estimates, and generated an overall plan—all in one afternoon. While they were still concerned about the plan's accuracy, they knew this was a big job, and there was no chance they could do the work in 9 months. They also had a lot of data to back up their 18-month schedule.

Next Came the Detailed Plan

The next step was to look at the work that lay immediately ahead. On the morning of the second day, the team made a detailed plan for the requirements phase. First, they examined the requirements process and broke it into the steps needed to produce, review, and correct each requirements product. To make sure their detailed plans fit into the overall plan, they started with the overall estimates and then estimated the engineering hours for each step. They then named an engineer for each task, and each engineer then used the same overall planning data as the starting point for a personal plan for the immediate next phase.

When the team put these plans together, the result was a shock. The combined detailed plans took much longer than the top-down plan for the same work. How could this be? The same engineers had made the plan and they had used the same product list, size estimates, and development rates.

The problem was *unbalanced workload*. The lead engineers were involved in every step of the work, and the less experienced engineers often had little to do. While the lead engineers could likely produce the best products and everyone felt that they should participate in every product review, this made them a serious bottleneck.

After some discussion, the team agreed to unload much of the lead engineers' work. By balancing the workload, the less experienced engineers got much more to do and the lead engineers concentrated on the most critical parts of the job. The final balanced plan produced the same schedule as the overall plan, and the team now felt they had a sound basis for doing the work. At this point, it was noon of the second day, and the team had all afternoon to assess project risks and to prepare a presentation for the management meeting.

What Happened

When the team presented their plan the next morning, management was impressed with the plan, but unhappy with the schedule. They really did need the product in 9 months, but, after considerable discussion, they were convinced that the 18-month schedule was the best that the team could do.

The team followed this plan in doing the job. The requirements phase took several weeks longer than planned and the design phase also took a little longer. But, the team stuck to their guns and did a quality job. As a consequence, there were fewer late requirements and design changes, implementation took less time than planned, and testing took practically no time at all.

In the end, the team finished the job 6 weeks ahead of the original 18-month schedule. Because of the well-thought-out design and the high product quality, marketing was able to contain the customer problem, and the product was a success.

Teamwork

Teams have a great deal of knowledge and experience, and when they are all involved in producing their own plans, they will invariably do a first-class job. After all, they will do the work, they have the most at stake, and they will derive the most benefit from having a realistic plan. With a detailed plan, teams know precisely how to do the work, and they feel obligated to finish on the dates to which they committed.

Closing Comments

First, early plans are invariably less accurate than those made later. The reason is that engineers often overlook tasks, they don't allow enough time to clear up requirements problems, and they assume that they will work full time on the job. Also, in many organizations, management fails to protect their teams from the normal turmoil and disruption of a running business. Thus, when you consider all the pressures in working software organizations, the early team plans are almost always aggressive. Thus, even if an earlier date is critically important, it is invariably a mistake to cut these initial plans. If the problem is severe, the team should make a new plan with different resource assumptions or work content. The best approach, however, is to wait until the end of the requirements phase to replan. Then everyone will better understand the work, and they can make a more accurate plan.

Second, there are lots of estimating tools and methods. While I am partial to the PROBE method described in one of my books,[4] estimating is a largely intuitive process. So, use whatever methods help your intuition. However, do not rely on some magic tool to produce the plan. While the detailed printout may look impressive, plans are only as good as the thought that went into them. Remember that the principal benefits of planning are the engineers' shared knowledge of how to do the work and the team's commitment to the plan. Use whatever tools and methods you have available to help make the plan and to check your results, but use these tools only to support your planning, not to replace it.

Third, to help you work efficiently and to coordinate your work with your teammates, you need a detailed plan for the work immediately ahead. While you can rarely produce a detailed plan for an entire development job, you should start with an overall plan and then produce a detailed plan for the phase you are about to start.

Fourth, when management is unhappy with your team's plan, don't change it without making a new plan. When you do, however, make sure you get different resource and work-content assumptions. Without changes in their planning assumptions, teams invariably think of previously overlooked tasks and end up with a longer schedule.

Finally, remember: if you cannot plan accurately, plan often. Plans are only as good as the knowledge on which they are based. As you gain new knowledge, produce new plans. As long as the previous plan is useful, however, don't bother making a new plan. But, the moment the plan ceases to provide helpful guidance, make a new plan.

[4] *A Discipline for Software Engineering*, Addison Wesley, 1995. http://www.sei.cmu.edu/library/abstracts/books/0201546108.cfm

The Commercial

While the methods I have described are not complex, they are not obvious. That is the purpose of the Team Software Process that we have developed at the SEI. It provides the guidance that teams need to follow these methods on the job. The catch, however, is that to use the TSP, engineers need to be trained in the Personal Software Process (PSP), and their management needs overall training and guidance on how to lead and guide TSP teams.

Acknowledgements

First, I would like to thank Walden Mathews for asking some perceptive questions about planning. The answers to his questions provided the basis for this column. Also, in writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful suggestions of John Goodenough, Mark Paulk, Bill Peterson, Marsha Pomeroy-Huff, and Dave Zubrow.

4 Bugs or Defects?

March 1999

One of the things that really bothers me is the common software practice of referring to software defects by the term “bugs.” In my view, they should be called “defects.” Any program with one or more defects should be recognized as defective. When I say this, most software engineers roll their eyes and feel I am out of my mind. But stick with me and you will see that I am not being unrealistic.

To explain why the term “bug” bothers me, I need to pose three questions. First, do defects really matter? Second, why not just worry about the important defects? And third, even if we have to worry about all the defects, why worry about what we call them?

Do Defects Really Matter?

To answer this question, we first need to find out if defects are or can be serious. To do that, we must define what we mean by “serious.” Here, while there will be wide differences of opinion, there is almost certainly a basis for general agreement. First, if a defect kills or seriously injures someone, we would all likely agree that it was a serious defect. Further, if a software defect caused severe financial disruption, we would all probably agree that it too was a serious defect.

Lastly, there are lots of less significant problems, such as inconvenience, inefficiency, and just plain annoyance. Here, the question of seriousness obviously depends on your point of view. If you are the one being inconvenienced, and if it happens often, you would likely consider this serious. On the other hand, if you are the supplier of such software, and the inconveniences do not cause people to sue you or go to your competitors, you would probably not judge these to be serious defects. However, if these defects significantly affected the bottom line of your business, you would again agree that these too were serious defects.

So we are left with the conclusion that if defects cause loss of life or limb, result in major economic disruption to our customers or users, or affect the profitability of our businesses, these are serious defects.

Do They Matter To You?

The real question, however, is not whether defects matter in a theoretical sense but whether they matter to you. One way to think about this would be in terms of paying the costs of dealing with defects. Suppose you had to pay the discovery, recovery, reporting, repairing, redistribution, and reinstallation costs for every defect a customer reported for your programs. At IBM in my day, these costs averaged around $20,000 per valid unique defect. And there were thousands of these defects every year.

Of course, the problem is not who made the mistake but why it wasn’t caught by the process. Thus, instead of tracing customer-discovered defects back to the engineers who injected them, we should concentrate on fixing the process. Regardless of their causes, however, defect costs can be substantial, and if you had to personally pay these costs, you would almost certainly take defects pretty seriously.

Why Not Just Worry About The Serious Defects?

At this point, I think most people would agree that there are serious defects, and in fact that the reports of serious defects have been increasing. Now that we agree that some defects are serious, why not just worry about the few that are really serious? This question leads to a further question: Is there any way to differentiate between the serious defects and all the others? If we could do this, of course, we could just concentrate on the serious problems and continue to handle all the others as we do at present.

To identify the serious defects, however, we must differentiate them from all the others. Unfortunately, there is no evidence that this is possible. In my experience, some of the most trivial-seeming defects can have the most dramatic consequences. In one example, an executive of a major manufacturer told me that the three most expensive defects his organization had encountered were an omitted line of code, two characters interchanged in a name, and an incorrect initialization. These each caused literally millions of dollars of damage. In my work, some of my most troublesome defects were caused by trivial typing mistakes.

How Many Defects Are Serious?

Surprisingly, the seriousness of a defect does not relate to the seriousness of the mistake that caused it. While major design mistakes can have serious consequences, so can minor coding errors. So it appears that some percentage of all the defects we inject will likely have serious consequences. But just how many defects is this? And do we really need to worry that much about this presumably small number?

Based on my work with the Personal Software Process (PSP), experienced programmers inject one defect in about every 10 lines of code (LOC) [Humphrey 1995]. While these numbers vary widely from engineer to engineer, and they include all the defects, even those found in desk checking or by the compiler, there are still lots of defects. And even for a given engineer, the numbers of defects will vary substantially from one program to the next. For example, when I have not written any programs for just a few weeks, I find that my error rate is substantially higher than it was when I was writing pretty much full time. I suspect this is true of other programmers as well. So engineers inject a large number of defects in their programs, even when they are very experienced.

Won’t the Compiler Find Them?

Now, even though there are lots of defects, engineers generally feel that the compiler will find all the trivial ones and that they just need to worry about the design mistakes. This, unfortunately, is not the case. Many of my programming mistakes were actually typing errors. Unfortunately, some of these mistakes produced syntax-like defects that were not flagged by the compiler. This was because some of my mistakes resulted in syntactically correct programs. For example, typing a “)” instead of a “}” in Pascal could extend a comment over a code fragment. Similarly, in C, typing “=“ instead of “= =“ can cause an assignment instead of a comparison. From my personal data, I found that in Pascal, 8.6% of my syntax defects were really syntax like, and in C++, this percentage was 9.4%.

Some syntax-like defects can have serious consequences and be hard to find. If, as in my case, there are about 50 syntax defects per 1000 lines of code (KLOC), and if about 10% of them are actually syntax like, then about 5 or so defects per KLOC of this type will not be detected during compilation. In addition there are 30 to 50 or so design-type defects that will not be detected during compilation. So we are left with a large number of defects, some of which are logical and some of which are entirely random. And these defects are sprinkled throughout our programs.

How About Exhaustive Testing?

Next, most engineers seem to think that testing will find their defects. First, it is an unfortunate fact that programs will run even when they have defects. In fact, they can have a lot of defects and still pass a lot of tests. To find even a large percentage of the defects in a program, we would have to test almost all the logical paths and conditions. And to find all of the defects in even small programs, we would have to run an exhaustive test.

To judge the practicality of doing this, I examined a small program of 59 LOC. I found that an exhaustive test would involve a total of 67 test cases, 368 path tests, and a total of 65,536 data values. And this was just for a 59 LOC program. This is obviously impractical. While this was just one small program, if you think that exhaustive testing is generally possible, I suggest you examine a few of your own small programs to see what an exhaustive test would involve. You will likely be surprised at what you find.

Just How Many Defects Are There?

So now, assuming you agree that exhaustive testing is impossible, and that some of these defects are likely to be serious, what next? First, we find that the programs that engineers produce have a lot of defects, at least before compilation and unit test. Also, we find that compilation and unit testing cannot find all of the defects. So a modest percentage of the defects are left after unit testing. But how many are likely left, and do these few defects really matter?

Here again, the data are compelling. From PSP data, we find that engineers typically find between 20 to 40 defects per KLOC when they first test their programs. From my personal experience, I also find that unit testing typically finds only about 45% to 50% of the defects in a program. This means that after unit test, programs will typically still have around 20 or more defects per KLOC remaining. This is an extraordinary number. This means that after the first unit testing, programs will typically have a defect every 50 or so LOC! And after integration and product test, there would still be 5 to 10 or more defects left per KLOC. Remember, all of these defects must be found in system testing, or they will be left for the customer.

Consider Some Data

Just so you know that this is not an exaggeration, consider some data from the Jet Propulsion Laboratory (JPL). This organization develops the spacecraft that NASA sends to explore the solar system. One of their internal reports listed all the defects found in the system testing of three spacecraft [Nikora 1991]. These software systems all had about 20 KLOC, and they were each tested for two or more years. The cumulative defects per KLOC found during this testing by week are shown in the figure. As you can see, they all had from 6.5 to nearly 9 defects per KLOC found in system test, and that does not guarantee that all the defects were found. In fact, from the figure, it seems pretty obvious that some defects remained. Note that these data were taken after the programs were developed, compiled, unit tested, and integration tested.

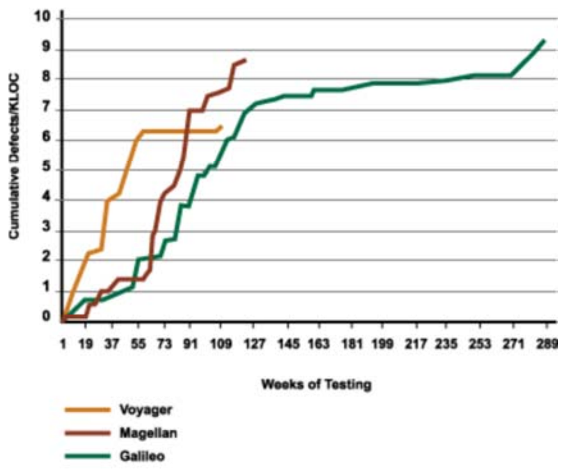

Figure 4-1: Spacecraft System Test Defects/KLOC

This, by the way, should not be taken as critical of JPL. Their programmers are highly skilled, but they followed traditional software practices. Normal practice is for programmers to first produce the code and then find and fix the defects while compiling and testing.

But Why Not Call Them "Bugs"?

So, by now, you presumably agree that some defects are serious, and that there is no way to tell in advance which ones will be serious. Also, you will probably agree that there are quite a few of these defects, at least in programs that have been produced by traditional means. So now, does it matter what we call these defects? My contention is that we should stop using the term "bug." This has an unfortunate connotation of merely being an annoyance; something you could swat away or ignore with minor discomfort.

By calling them "bugs," you tend to minimize a potentially serious problem. When you have finished testing a program, for example, and say that it only has a few bugs left, people tend to think that is probably okay. Suppose, however, you used the term "land mines" to refer to these latent defects. Then, when you have finished testing the program, and say it only has a few land mines left, people might not be so relaxed. The point is that program defects are more like land mines than insects. While they may not all explode, and some of them might be duds, when you come across a few of them, they could be fatal, or at least very damaging.

We Think In Language

When the words we use imply that something is insignificant, we tend to treat it that way, at least if we don't know any better. For example, programmers view the mistakes they make as minor details that will be quickly found by the compiler or a few tests. If all software defects were minor, this would not be a problem. Then we could happily fix the ones we stumble across and leave the rest to be fixed when the users find them. But by now you presumably agree that some software defects are serious, that there is no way to know in advance which defects will be serious, and that there are growing numbers of these serious defects. I hope you also agree that we should stop referring to software defects as "bugs."

Finally, now that we agree, you might ask if there is anything we can do about all these defects? While that is a good question, I will address it in later columns. It should be no surprise, however, that the answer will involve the PSP and TSP (Team Software Process) [Webb 1999].

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful suggestions of Dan Burton, Bob Cannon, Sholom Cohen, Frank Gmeindl, and Bill Peterson.

References

[Humphrey 1995]

Humphrey, Watts S. *A Discipline for Software Engineering*. Reading, Ma.: Addison Wesley, 1995.

[Nikora 1991]

Nikora, Allen P. *Error Discovery Rate by Severity Category and Time to Repair Software Failures for Three JPL Flight Projects.* Software Product Assurance Section, Jet Propulsion Laboratory, 4800 Oak Grove Drive, Pasadena, CA 91109-8099, November 5, 1991.

[Webb 1999]

Webb, Dave & Humphrey, Watts S. "Using the TSP on the TaskView Project," *CROSSTALK*, February, 1999.

5 Doing Disciplined Work

June 1999

Over the years, I have often been asked about how to get management support for process improvement. Typically, engineers want to use better software methods; but they have found that either their management doesn't care about the methods they use or, worse yet, some managers even discourage them from trying to improve the way they work. In addressing this subject, I have decided to break it into two parts. The first part concerns disciplined work: what it is, and what it takes to do it. Then, in later columns, I will address the problems of getting management support for process improvement.

In discussing disciplined work, I answer five questions. First, what do we mean by disciplined work? Second, why is disciplined work important? Third, what are the elements of disciplined engineering work? Next, if you have the basic training and motivation to do disciplined work, why can't you just do it? And finally, what kind of support and assistance do you need to consistently do disciplined work?

What Is Disciplined Work?

Discipline is an aspect of behavior. It involves consistently using sound methods. Discipline is defined as an activity or regimen that develops or improves skill; acting in accordance with known rules and proven guidelines. Disciplined behavior is generally needed whenever human error can cause harm, substantial inconvenience, or expense. The disciplined behaviors are then designed to reduce human errors, prevent common mistakes, and improve the consistency of the work. Also, many people are surprised to find that disciplined behavior generally improves efficiency, saves time, and even facilitates creativity.

Why Is Disciplined Work Important?

In every advanced field, you get better and more consistent results by using proper methods and applying known and proven techniques. This is true in factories, development organizations, and even research laboratories. No one would agree to an operation by a doctor who had not finished medical school, spent years as a resident, and been board certified. Similarly, when hiring an accountant, you would not consider someone who did not have a CPA certificate and was properly licensed. If you had to go to court, you could not use a lawyer who had not passed the bar and been qualified to practice in the state. Anyone doing biomedical or nuclear research knows that disciplined behavior can be a matter of life and death.

While the software field is too new to have qualification mechanisms like those in many other fields, we now know the practices required for good software engineering. In addition, we also know that when software engineers use these methods, they consistently produce quality products on their committed schedules and for the planned costs. Unfortunately, however, these methods are not yet generally taught in university curricula. Thus, to advance in this field, engineers need to get their own training and to develop their personal skills.

What Are the Elements of Disciplined Engineering Work?

Disciplined software engineering involves more than just producing good technical results. As is true in other advanced fields, you need to address every aspect of the job. While technical competence is essential, you also need to consider the customer's needs, handle business concerns, and coordinate with your teammates. If, for example, you handle the technical concerns but ignore those related to the customer, you will likely produce a product that solves the wrong problem. While you might not lose your job, it is never good for an engineer's career to be associated with a failed project.

Proper attention to customer-related issues requires understanding the requirements before starting the design, maintaining close customer contact throughout the work, and planning and negotiating every change with the customer. Important business issues involve planning, tracking, and reporting on the work; focusing on quality from the beginning; and identifying and managing risks. Key teamwork issues relate to agreeing on goals, making and meeting commitments, and reviewing and supporting teammates' work. Finally, your team needs a logical development strategy, a sound architecture, a comprehensive design, and a set of rigorously followed standards and methods.

Getting the Needed Skills

There are various ways to obtain the skills needed for disciplined work. While some of these skills can be learned in university programs[5], many must be learned through on-the-job experience.[6] Instruction in disciplined personal and team software methods can also be obtained from the SEI and its transition partners, but that requires your management's support. While qualified training is the route that I would suggest for anyone who can follow it, a principal concern of many engineers is that they can't get management support.

Why Not Just Do It?

While getting training is important, consistently using the methods that you know is even more important. Unfortunately, it is also much more difficult. A growing number of engineers are being trained in disciplined engineering methods, but many of them find that even though they know how to do good software work, they are unable to practice what they know. The reason is that disciplined work is very hard to do, particularly when you try to do it by yourself. Here, there are two contributing elements: lack of personal discipline and inadequate coaching and support.

[5] Available PSP courses teach planning, process, measurement, and quality methods, using my text: A Discipline for Software Engineering. If you can't get into such a course, you could learn the basics from the PSP introductory text, *Introduction to the Personal Software Process*.

[6] I have written a textbook for a new teamworking course. It is called *Introduction to the Team Software Process*, and it teaches the basics of the TSP.

Personal Discipline

How many times have you decided to do something but never really did it? Like New Year's resolutions, there are lots of things we know we should do like quitting smoking, not eating between meals, exercising every day, and many others. We kid ourselves that we could do these things if we really had to, but somehow we never do.

The problem here is that most of us try to maintain strict personal disciplines all by ourselves. For example, can you visualize working for years to become a concert-quality pianist in a deaf world? When the quality of your work is invisible and nobody knows or cares how you perform, it is almost impossible to follow rigorous personal disciplines. This is why professional athletes and performing artists have coaches, trainers, conductors, or directors.

Even at the pinnacle of their fields, professional performers need the help and support of a cheering section, the constant push and motivation of a coach, and the demanding guidance of an informed and caring trainer. This is not just a nicety; it is absolutely essential. We humans are a group species. We work best in groups, and we have great difficulty performing alone. We need somebody who knows and cares.

Coaching and Support

I was fortunate to be on a marvelous team when I was in college. Our coach had been on the U.S. Olympic wrestling team; and he was an energetic, enthusiastic, and terribly demanding coach. Nobody wanted to disappoint him. We all worked harder than any of us had ever worked before. In our first year, he took a team of rookies to the AAU championship of 13 states. What was most interesting to me was that the next year I transferred to a different school. The wrestling coach was a nice guy, but he was not demanding or enthusiastic. Not only didn't the team do well, I didn't either. Superior coaching makes an extraordinary difference, and it is necessary for any kind of disciplined personal work.

What Kind of Support Do We Need?

The issues that we face in software engineering are severalfold. First, our field has not yet developed a tradition of disciplined work. Thus we must change an industry-wide culture. Second, coaching is not a common management style. Managers in software, as in other fields, feel more natural acting like straw bosses. Few know how to use, build, and develop the skills of their people. But this is the essence of management: helping and guiding people to do the best work that they are capable of producing. When people don't perform as well as they should, managers should help them develop their skills and motivate them to rigorously use the methods they know.

In software engineering, good work requires engineering discipline, and disciplined work requires coaching. In a subsequent column, I will discuss getting managers to act like coaches.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful suggestions of Bob Cannon, Bill Peterson, and Mark Paulk. I also thank Jean-Marc Heneman for his questions and comments on this topic.

6 Getting Management Support for Process Improvement

September 1999

Over the years, I have often been asked about how to get management support for process improvement. Typically, engineers want to use better software methods but they have found that their management either doesn't care about the methods they use or, worse yet, even discourages them from trying to improve the way they work. In addressing this subject, I have decided to break it into two parts. The first part ("Doing Disciplined Work" on page 29) concerns disciplined work: what it is, and what it takes to do it. In this column I address the problem of getting management support for process improvement.

Obtaining Broad Management Support

Perhaps the biggest problem in starting an improvement effort is getting management support. The first and most important step is to get senior management backing. Without support from the very top, it is generally impossible to make significant changes. Next, however, you will need active involvement from all the appropriate managers, particularly those managers who directly supervise the work to be impacted by the change.

The reason for broad management support is that significant improvement programs generally involve substantial changes in the way people work. If you don't change the engineers' working practices, you can change the organizational structure and all its procedures, but nothing much will really change. Thus, to have a substantial impact on an organization's performance, you must change the way the engineers actually work. While this is possible, it is very difficult, and it requires the support of all levels of management. Senior managers must establish goals and adjust reward systems. Intermediate managers need to provide funding and change priorities. And most important, the working-level managers must make the engineers available for training, support process development, and monitor the engineers' work to make sure they follow the improved practices. So, how do you get this kind of support? To address this question, we discuss three issues:

1. Why do you want to make changes?
2. Which managers do you need support from?
3. Why should those managers support you?

Why Do You Want To Make Changes?

Since you are reading this column, you are probably interested in making process changes, and these changes are undoubtedly in the way your organization develops or maintains software. This means you are probably talking about some kind of process improvement, like getting a Capability Maturity Model (CMM) program underway or introducing the Personal Software Process (PSP) and Team Software Process (TSP). Whatever the approach, you will be changing the way software work is done.

The first question to address is: why? That is, why do you want to improve the software process, why should management support you in improving the software process, and why should the organization care about how software is developed? These are tough questions, but they are the very first questions managers will ask. You need to be able to answer these questions, and depending on which managers you talk to, they will ask these questions differently. This leads us to the next question.

Which Managers Do You Need Support From?

Depending on the size of your organization, there could be many management levels. Typically, the manager from whom most of us need support is the manager immediately above us. While there are lots of levels to discuss, let me assume that this immediate manager runs a project or a department. Unless you are in a very small organization, this manager probably works for some higher level manager, and this higher level manager probably works for some manager at an even higher level. Up there somewhere there should be a senior-level manager or executive who is concerned with the overall business, how it performs now, and how it will perform in the future. This senior manager is concerned with where the business stands competitively, how new technology will impact its products and services, and the changing needs of its customers.

The reason the manager's level is important to you is that improvement programs focus on long-term issues that are the principal concern of senior-level executives. Unless the managers below the executive level are specifically charged with working on process improvement, most of them will view improvement efforts as a distraction at best or, at worst, as a drain on critical resources.

The reason for this negative view is that process improvement deals with the overall performance of an organization. It concerns competitive capabilities, long-term cost effectiveness, development cycle-time improvement, and customer satisfaction. These are strategic issues that generally only concern the most senior executives. Even in the departments, laboratories, or divisions of large corporations, the performance measures for division general managers, laboratory directors, and department managers are invariably concerned with immediate short-term results: delivering products on time, managing tight budgets, or responding to customer-related crises.

While these issues are critically important, and they often spell the difference between organizational failure and success, a total concentration on these topics will not change the way organizations perform. If the organization is not cost competitive, or if it produces lower quality or less attractive products, a focus on current performance will not improve the situation. The immediate problems may be fixed and the burning issues resolved, but the organization will continue working pretty much as it always has. It will thus continue producing essentially the same results and generating essentially the same problems and issues. This brings us to the definition of insanity: doing the same thing over and over and expecting a different result.

Generally, only the managers who think strategically will support a process-improvement program. These are usually managers who have broad business responsibilities and are measured by total organizational performance. They probably have multiple functions reporting to them, like product development, marketing, manufacturing, and service.

Even senior managers, however, do not always think strategically. Most organizations, after all, are owned by stockholders who are interested in the stock price. And since the stock price is heavily influenced by quarterly financial results, even the most senior managers cannot afford to ignore short-term financial performance. Unfortunately, many of these managers don't worry about much else.

Why Should This Manager Support You?

Now we get to the critical question: Why should any manager support you? In general terms, there are three reasons why managers might be willing to support you:

1. What you want to do supports their current job objectives.
2. What you want to do will make them look good to their immediate and higher level managers.
3. What you want to do is so clearly right that they are willing to support you in spite of its impact on their immediate performance measures.

Getting Help From a Senior Manager

The relative importance of these reasons changes, depending on where the manager resides in the management chain. At the very top are the managers who are most likely to focus on long-term performance. This means that they will often support process improvement for all three reasons. Thus, if you can show that process improvement will have a significant long-term benefit, you will likely get support. You can generally accomplish this by showing how similar improvements have benefited other organizations or, better yet, how they have benefited other parts of your own organization.

For the CMM, for example, show how improvements in CMM level have improved the performance of other software organizations. Also, show where your organization stands compared with other organizations in your industry. For the PSP and TSP, you could show data on quality, productivity, or employee turnover and how such changes could impact your organization.

If you can get the attention of a senior manager, and if you have your facts straight, the odds are you can get this manager to seriously consider the subject of process improvement. Frequently this is when you might get an outside expert to give a talk or to do an assessment. While you may have to settle for a small initial step, the key is to get some action taken. Once you can get the ball rolling, it is usually easier to keep it in motion.

If the manager you are dealing with is not at the senior executive level but one level lower, this manager is probably not measured on strategic issues. Such managers would know, however, that their immediate manager had such a measure. Thus, your manager is not likely to be motivated by reason 1 but might be persuaded to support you for reason 2. Thus, by proposing something that will make him or her look good to higher level managers, this manager will personally benefit while also helping you to get the improvement ball rolling. What you want to ask for from this manager is help in taking the improvement story upstairs.

Getting Help At the First Management Level

Finally, the most common problem is dealing with a manager who is fairly far down in the organization. This manager not only is not measured on strategic issues, but his or her immediate manager is not either. This means that strategic objectives are not likely to be very compelling. At this point, you only have two choices:

- Convince this manager that the improvement is a strategic necessity for the organization.
- Show how the improvement effort can help to address immediate short-term concerns.

While the latter is often the approach you must take, it has a built-in trap. The reason is that if improvement is aimed at solving a short-term problem, as soon as the short-term pain is relieved, the need for improvement is gone. This is like taking aspirin for a splitting headache. If the headache is indeed a transient problem, that would be appropriate. If the pain is the first symptom of a stroke or a brain tumor, however, the delay could be fatal. While promptly taking an aspirin may usually be helpful for a stroke, you had better also see a doctor right away.

In the software process, the problems in most organizations are more like strokes and brain tumors than they are like headaches. While you may have no choice but to sell the improvement effort as a short-term solution, try to move to strategic issues as soon as you can get the attention of someone upstairs.

The next questions concern making the strategic case for improvements, making the tactical case, and moving from a tactically based to a strategically based improvement program. These will be topics of future columns.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Dan Burton, Jean-Marc Heneman, Julia Mullaney, and Bill Peterson.

7 Making the Strategic Case for Process Improvement

December 1999

In my previous column ("Getting Management Support for Process Improvement" on page 33), I wrote about management support for process improvement and how your approach should change depending on the manager you are dealing with. The questions left open were how to make the strategic case for improvement, how to make the tactical case for improvement, and how to move from a tactically based to a strategically based improvement program.

In this issue, I describe how to make the strategic case for process improvement. I start on the assumption that you can get the ear of a senior manager. You may work directly for this manager, or you may have obtained an appointment to make a presentation on the subject. In any event, you now have an appointment to see a senior manager. How do you prepare and what do you say?

The General Improvement Case

The approach to follow for almost any type of improvement effort would be much the same:

- Clearly define what you propose.
- Understand today's business environment.
- Identify the executive's current hot buttons.
- Make an initial sanity check.
- Start the plan with two or three prototypes.
- Estimate the one-time introduction costs.
- Determine the likely continuing costs.
- Document the available experience data.
- Estimate the expected savings.
- Decide how to measure the actual benefits.
- Determine the improvement's likely impact on the executive's current key concerns.
- Identify any other ways that the proposed improvement could benefit the business.
- Produce a presentation to give this story clearly and concisely.

Defining the Proposal

Before you do anything, define exactly what you want the executive to do. The best guide that I have found is to actually prepare an implementation letter for the executive's signature. Then in the meeting, if he or she says, "Great, let's do it," pull out the letter and hand it over as a proposed implementation instruction. While this reaction is not likely, the exercise will help you to produce a clear statement of what you intend to propose. Also, if you are several management levels removed from this executive, you should describe the letter as a proposed draft instruction that you have not yet reviewed with your immediate management. Better yet, show the draft letter to your manager first and get his or her suggestions on improving it.

Understand Today’s Business Environment

In preparing for the presentation, remember that there is no magic formula for convincing senior managers. Every case is different. The approach must vary depending on the situation and the executive’s current priorities. If, for example, this executive has just cut resources to meet a profit goal or the organization has just lost a major contract, this might not be a good time to propose an additional expense. So, plan your improvement strategy with a clear appreciation of what is happening right now in the business.

Identify the Hot Buttons

Next, find out what this executive is most concerned about. Since most executives give lots of talks and issue many statements and announcements, this is generally fairly easy to do. With few exceptions, executives use every available occasion to plug the topics they feel are highest priority. So get copies of some of this executive’s recent announcements and presentations, and look at the common themes. You will usually see a fairly consistent message. The manager may frequently mention profitability, or market growth, or development cycle time. Because executives are concerned with many things, he or she will almost certainly make many points. But if there is an overriding concern, much like a television commercial, this topic will pop up every time there is an opportunity. Once you know the executive’s current hot button, figure out how the process improvement you propose would address that concern, then make sure the improvement justification addresses this topic.

Make an Improvement Sanity Check

In preparing an executive proposal, the first step is to gather the known facts about the costs and benefits of the proposed improvement program. As soon as you have the data, make an initial sanity check: Does the proposed process improvement directly address the executive’s key concerns? If not, are the cost savings significant enough to justify the executive’s listening to the proposal? If the improvement directly addresses something the executive has been pushing for, then cost will not be a key concern. If cost savings are important, however, are the proposed savings large enough to be convincing?

Most executives know that improvements are rarely as effective as first proposed and that there are always hidden costs. A good rule of thumb is that improvements with savings of 2 or more times are usually impressive while numbers below 25% are likely to be ignored or subjected to very close scrutiny. If cost is important and you are not proposing a significant cost saving, consider putting off the presentation until you can make a stronger case.

Prototype Introduction

If the proposed improvement passes this sanity check, the next step is to analyze the costs of introduction. It is almost always a good idea to start an improvement program with one or more prototype tests. This not only reduces the initial introduction costs, it also maximizes your chances of success. Just about any change will affect both engineer and management behavior, and these changes are rarely natural or easy. Thus, many people will likely have initial problems following the new methods. To be successful, you must identify and resolve these problems at the very beginning. The longer it takes people to properly use the new methods, the more the introduction will cost and the longer it will take to show benefits. The principal advantages of starting with a prototype program are that the initial costs are lower and it is easier to watch a few limited pilot programs to make sure they are getting the needed support and assistance.

One major risk in any improvement program is that the prototype project could be cancelled or redirected. To protect your project from this risk, try to get two or three trial projects underway. That way, if one is killed or redirected, you will still have the others to fall back on.

Introduction Costs

While you will almost certainly follow a gradual introduction strategy, it is a good idea to show both the prototype and the total introduction costs. The reason is that the introduction strategy will probably change several times before you are done and you don't want to keep changing the cost-benefit story. Emphasize that you are presenting the total introduction costs for the entire organization, but that the initial costs for the prototype program will be much lower.

In any significant improvement, there will be initial introduction costs as well as continuing costs for sustaining the improvement. Since any process-improvement introduction will require some executive and management time, you need to make an appropriate allowance. Generally, however, the major costs will be the time to train and support the engineers. Even the introduction of a new tool takes training and support, so don't gloss over the introduction costs; they can amount to very big bucks.

For example, with a new programming language, a minimum of two weeks of intensive training is usually required, often followed by a period of close consultation during initial use. Similarly, a new tool will require an initial training session of several days plus guided practice sessions and continuing professional support for at least a few weeks.

In estimating these costs, remember one key guideline: Your story will be judged by its weakest point. If someone finds an error or a serious underestimate anywhere in the story, the assumption will be that similar errors infect the entire story. So be careful about making low estimates or assuming that some costs are insignificant. If you don't know the facts, find someone who does. Above all, don't make unsupported assumptions; your entire presentation could be discredited.

In addition to executive, manager, and engineering time, trainers and expert assistance will almost invariably be needed. This can add a significant cost, particularly if you plan to use outside assistance. On the other hand, the costs of building internal experts and trainers can be very large, and few executives will want to make such a significant commitment, at least until the proposed improvement has been proven with early tests.

The Continuing Costs

After the improvement has been introduced, there will be ongoing support costs. You may need continuing training to cover engineering turnover or staff growth. Expert assistance and support may also be needed. These costs can be substantial, so it is important to identify them. Describe them clearly up front and then justify them. If you don't give a complete cost story, management will sense that there are hidden costs and likely assume that these costs are much greater than they actually are.

The Process-Improvement Benefits

Next, we turn to the benefits. Here, you must address two points: first, how long will it take for the improvement program to recover the introduction costs, and second, how will the improvement address the executive's principal concerns? If you can show that the improvement will pay for itself, then the other benefits would be pure gravy. So start by making the cost case.

The way to make the cost case is to first gather the available facts on improvement benefits. Here, you are at a disadvantage. Costs are always easier to prove than savings. Executives know this, however, probably better than you do. After all, they spend much of their time justifying changes. So don't worry about proving an ironclad case; executives will rarely demand it. But they will want a logical story that hangs together and looks complete and realistic.

Improvement Experience

So, first, what are the available facts? Unfortunately, there are few statistically sound improvement studies, even for accepted process-improvement methods. While there may be some available analyses, you will probably have to rely on anecdotal evidence. This may not be as precise as a comprehensive statistical study, but such evidence can be even more convincing. The best case would be one in which someone in your industry has implemented the same improvement and described its benefits in a talk or a paper. If you can find a suitable example, summarize the general findings in the executive presentation, but then emphasize the results reported by your competition.

Calculating the Savings

There are many ways to save money. In the final analysis, however, most software cost savings result from personnel reductions. For example

- By introducing a design inspection program, you can eliminate defects early in the process and save considerable rework.
- A measured quality program can reduce the numbers of defects found in test and shorten testing time.
- A configuration-management system can save development time by ensuring that correct program versions are always available.
- A change-control system can reduce the number of uncontrolled changes and save development time.

While these savings are all real, they all have the disadvantage of being very hard to prove, either in advance or after the fact. As a result, the most convincing argument is generally that the XYZ Corporation cut their test time by x%, or that the ABC Company reduced customer-reported defects by y%. Starting from these numbers, you can then generally show the amount of money you would save if your organization had similar results.

Measuring the Benefits

In concluding the presentation, discuss how the prototypes will be designed to measure the improvement benefits. For example, if the proposed improvement is intended to reduce development cycle time, discuss how to demonstrate that it does. A common problem, however, is that few organizations have data on their current operations. Thus, even if you conduct a highly successful prototype experiment, you may have no way to show that it was successful. That is, you will have lots of "after" data but no "before" data with which to compare it. As part of the proposal, raise this issue and suggest ways to handle it.

Even when organizations have little or no data on their current operations, there are usually a few things that you can measure. For example, data are often available on the length of time by which projects have missed their planned delivery dates. There are also often records of the numbers of defects found in system test or reported by customers. Similarly, data can generally be found to calculate the percentage of the development schedule that is spent in integration and system test. Another good measure is the total development hours divided by the total lines of delivered source code. While no single measure can characterize the quality of an organization's processes, there are many possible measures that can be obtained from most accounting and project-reporting systems.

Because you need to apply these measures to the existing projects, it is important to start looking around for available data even before you make the proposal. Then you can use these data in justifying the proposed improvement. Also, you can be reasonably sure that there will be a way to measure the benefits when you are done.

Other Benefits

While cost savings are important, not all improvements can or should be cost justified. For example, if you can show that the change will improve schedule accuracy and predictability, reduce cycle time, or make your organization more competitive, management will often approve the proposal, even if it does not clearly save money. The key is to convince management that the improvement is good for the business and then, if possible, show that it will also pay for itself. If you cannot prove the savings story, however, don't give up. If the other benefits are compelling, management may be willing to proceed anyway.

Stay Tuned

In the next issue, I will use an example to show how to structure and give an executive presentation on process improvement. Following that, subsequent columns will deal with how to make the tactical case for improvements and then how to move from a tactically to a strategically based improvement program.

Letter to Business Week

Finally, I want to take the opportunity in this column to share with readers a letter I wrote to Business Week, in which I was quoted in the Dec. 6, 1999, issue:

To the Editor of *Business Week*

November 29, 1999

Dear Sir,

The article “Will Bugs Eat Up the U.S. Lead in Software?” nicely characterized the software quality problem and the fact that U.S. industry is slow to recognize and address it. Unfortunately, the article also implied that I was single-handedly responsible for the current work to address this problem.

While I did initiate and lead the work to produce the first version of the Capability Maturity Model (CMM), it was fully developed by a joint effort of the SEI, U.S. industry, and the Department of Defense. My more recent work on personal and team software process improvement also involves a team of SEI professionals and a growing number of industry and academic participants.

The U.S. needs more people who are concerned about this problem and willing to devote their lives to addressing it. It is thus important to recognize those who are already participating in this work and to encourage more to join us.

Watts S. Humphrey
SEI Fellow, Software Engineering Institute
Carnegie Mellon University

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Dan Burton, Marsha Pomeroy-Huff, Jim McHale, Mark Paulk, and Bill Peterson.

8 Justifying a Process Improvement Proposal

March 2000

My December 1999 column described how to make the strategic case for process improvement. In this column I provide an example of how to do this. This column thus assumes that you have the ear of a manager or executive who thinks strategically and will consider investments that will likely take a few years to pay off. In the next column, I will talk about dealing with tactically focused managers.

The Financial Justification Process

The financial justification process has five phases:

Phase 1: Decide what to do.

Phase 2: Estimate the likely costs.

Phase 3: Estimate the likely improvement benefits.

Phase 4: Produce the improvement proposal.

Phase 5: Close the deal.

The December 1999 column generally discussed these steps. This column walks you through a hypothetical case study in which Tom Jones develops a proposal to introduce the Team Software Process (TSP).

Phase 1: Decide What to Do

Tom reviewed the situation in his organization and found that management's top priority was to reduce development cycle time. He decided to do this by introducing the TSP. He also talked to experts about the TSP introduction strategy and found that it had the following seven steps:

Step 1	Hold an executive seminar for selected top managers and executives from the division or laboratory. Tom estimated that there would be 20 attendees.
Step 2	Give a half-day planning session to determine the improvement plan. Tom assumed that 10 of the first-day attendees would participate.
Decision 1	Tom assumed that these first two phases would be successful but decided to include a decision step to reduce the required initial commitment.
Step 3	Train the involved managers in the Personal Software Process (PSP) and TSP management methods. Tom planned to include the team leaders, the managers of the team leaders, and several other managers below the executive level. He assumed there would be 10 managers in this course.

Step 4	Provide PSP training to all the engineers who will be on the teams. Tom assumed there would be 2 teams with 8 engineers per team, or 16 engineers.
Step 5	Provide general PSP training to any team members who are not programmers. This would be systems, hardware, or test personnel, for example. He assumed that the first two TSP teams would be software only and that there would be four system-requirements team members in this category.
Step 6	Tom planned to train two engineers to be PSP instructors so they could support the two TSP teams, handle training for any team turnover replacements, and support further TSP introduction. These two instructor candidates would also attend PSP training in step 4, bringing that total up to 18.
Step 7	The final introduction step Tom planned was to launch the two TSP teams.
Decision 2	Assuming that these initial team launches were successful, Tom planned to ask management to proceed with broader TSP introduction.

Phase 2: Estimate the Likely Costs

After Tom defined the proposed introduction program, he estimated its costs in four parts:

- Labor costs
- Internal support costs
- Consulting, training, and external support costs
- Lost opportunity costs

Estimating the Labor Costs

For the labor costs, Tom estimated the number of people to be involved in each of the introduction steps as shown in Table 8-1 and Table 8-2. Because the TSP launches in step 7 would be part of the project, however, he did not count them as training time.

Next, Tom checked with the financial people and found that the cost for a day of engineering time was about $1,000 and that a manager or executive day cost about $2,000. While these rates seemed high to him, finance explained that they included all the costs for overhead, support, vacation, medical benefits, sick time, workers' compensation, insurance, retirement, Social Security, and taxes. Using these numbers, Tom calculated that the labor costs to train the two TSP teams and their managers would be $331,000 + $136,000 = $467,000.

Support Costs

Tom found that each TSP team would need coaching support of about 20% of the time of a TSP instructor/coach. These costs would add about 80 engineer days per year or $80,000 a year.

Finally, any new or replacement engineers on the teams would have to be trained. Assuming an annual turnover of 20%, that would be about three engineers to train per year at 14 days of training each, or another 42 days of training and $42,000 per year.

Table 8-1: PSP and TSP Introduction Program Training

Step	Item	People	Prep. Time	Class Days	Engineer Days	Manager Days
1	Executive Seminar	20		1.0		20
2	Planning Session	10		0.5		5
3	Manager PSP Training	10	0.3	4.0		43
4	PSP Course I & II	18	2.5	7.0+7.0	297	
5	General PSP Course	4	0.5	3.0	14	
6	Instructor/Coach Courses	2		10.0	20	
	Totals				**331**	**68**

Table 8-2: Total TSP Five-Year Costs

Cost Item	Cost Calculation	Total Cost
Internal introduction costs	$1,000*331 engineering days $2,000*68 manager days	$467,000
TSP coaching costs	2 engineers*40 days*1 year	$80,000
External introduction costs	25% to 75% of internal costs	$136,750 to $410,250
Total one-year costs		$683,750 to $957,250
TSP coaching costs	2 engineers*40 days*4 years	$320,000
Turnover training	3 engineers*14 days*4 years	$168,000
Total five-year costs		$1,171,750 to $1,445,250

External Costs

The costs for the external instructors and consultants for any improvement program are typically fairly large, but they are generally much smaller than the labor costs. These external costs would include delivering the courses listed in Table 8-1, launching and supporting the initial TSP teams, several team relaunches, and occasional consultation and assistance.

Without going out for external quotes, Tom assumed that the external support costs for this initial effort would add between 25% and 75% to the first-year labor costs, with the level of introduction cost determined by how rapidly management wanted to introduce the TSP.

Lost Opportunity Costs

Tom realized that while the engineers and managers were being trained they would not be developing or supporting products. To account for these costs, he would reduce the anticipated first-year cycle-time improvement by the three-week engineer training time.

Phase 3: Estimate the Likely Improvement Benefits

In estimating the improvement benefits, Tom found data on the benefits that other organizations had enjoyed with the TSP, and he also obtained data on the current performance of his organization. Finally, he used these data to estimate the likely improvement benefits for his organization.

Identify Available Improvement Data

Tom learned that Hill Air Force Base, on its first TSP project, increased productivity by 123% over the same team's prior project [Webb 1999]. Hill AFB also cut test time from 22% to 2.7% of development time, or a reduction of 88%.

He also found a presentation by Teradyne on its TSP results [Musson 1999]. Through the use of the TSP, Teradyne cut final test and field defects from a rate of 20 defects per thousand lines of code (KLOC) to 1 defect/KLOC. Historically, final test and field defects had cost Teradyne an average of 12 engineering hours per defect to find and fix.

By using the TSP, Tom also learned that Teradyne had reduced engineering and customer-acceptance test time from nine months for an earlier project to five weeks for the TSP. Engineering and acceptance test defects were cut from 5 defects/KLOC to 0.02 defects/KLOC, with no further defects found by the customers.

In addition, he found that a Boeing TSP project had cut test defects by 75% from the prior project average and reduced system test time by 96% [Vu].

Organizational Performance Data

To calculate the TSP benefits for his organization, Tom next needed data on how much time these development groups spent in test, the level of defects in the various test phases, and the cost of diagnosing and fixing each defect. With these data, he could then estimate the likely savings from introducing the TSP.

Tom assumed that the 16 engineers on the 2 trial projects would develop a total of about 80,000 lines of code in 12 months. He also found that the time typically spent in integration and system test was currently about 40% of the development schedule and that the defect levels in test and field use were much like those at Teradyne, or 20 defects/KLOC. He assumed that these 20 defects were split with 10 in integration test, 5 in system test, and 5 after product shipment. He also assumed that the engineering cost to diagnose and fix these defects was about 1.5 days per defect.

Estimate the Likely Cost Savings

As shown in Table 8-3, Tom now estimated the likely savings. First, for the reduction in test defects, he projected that the integration and system test defect/KLOC would be reduced from 15 to 1, for a savings of 14 defects/KLOC. For the total 80,000 lines of code planned for these two projects, he calculated that this would save 14*80 = 1,120 test defects. At a cost of 1.5 engineering days for each defect, this would be a reduction of 1,680 engineering days or $1,680,000.

Table 8-3: Estimated Savings

#	Item	Change	Days	Savings
#1	Integration/system test defects	- 1,120 defects	1,120*1.5=1,680	$1,680,000
#2	Test time	- 4.1 months	4.1*21*16=1,378	$1,378,000
#3	Maintenance costs	-398 defects	597	$597,000
#4	First-year savings (#2+#3)			$1,975,000
#5	First-year costs (Table 8-2)			$957,250
#6	First-year ROI (100*#4/#5)			206%
#7	Five-year savings (#4*5 years)			$9,875,000
#8	Five-year costs (Table 8-2)			$1,445,250
#9	Five-year ROI (100*#7/#8)			683%

To check these savings, Tom also looked at test-time reductions. The Boeing results showed a test-time reduction of 96% and Hill AFB reported an 88% reduction. Tom assumed that the TSP would reduce his organization's integration and system test time by 85%, or from 4.8 months to 0.7 months for a 12-month project. This 4.1-month savings, at 21 working days per month and 16 engineers, came to a savings of 1,378 engineering days. At the typical engineering day rate of $1,000, the test time savings were then $1,378,000. Since this estimate was a little lower than the $1,681,000 savings based on defect reduction, he decided to use the lower number in the justification calculations.

In addition, Tom also felt that there would be a maintenance cost reduction. For field and customer defects, the Teradyne data showed a reduction from 5 defects/KLOC to 0.02 defects/KLOC. For the 80,000 lines of code planned in his organization, he estimated a reduction of 398 customer-reported defects. At 1.5 engineering days each, this would be a maintenance-cost savings of $597,000, for a total savings of $1,975,000 in one year. He then compared this to the maximum one-time improvement cost of $957,250 to give a one-year return on investment of 206%.

Finally, Tom assumed that the initial two TSP teams would continue to use the TSP on subsequent projects. For the next four years, additional costs would be needed to cover training for engineering turnover ($168,000) and 20% of the time for the two PSP instructor/coaches ($320,000). The five-year savings would then be $9,876,000 and the five-year improvement cost would be $1,445,250, for a return on investment of 683%.

Cycle-Time Benefits

Next, Tom used the reduction in test time to estimate cycle-time improvement. With the assumed 12-month project schedule and 40% of the schedule spent in test, normal testing time would be 4.8 months. By assuming an 85% reduction in test time, he estimated that the 4.8 months would be reduced to 0.7 months. Thus, the typical 12-month schedule would be cut to eight months, or about a 32% cycle-time reduction. For the first year, he also reduced this 4.1-month cycle time improvement by the three-week initial team training time, for a net of 3.35 months schedule savings.

Phase 4: Produce the Improvement Proposal

At this point, Tom had completed the estimating work and needed to produce a brief management presentation. He decided on the following seven-part outline:

Presentation *Part 1*	Opening summary. On one chart, he briefly summarized the problem, how he proposed to address it, and the likely benefits.	
	The problem	To improve cycle time and not increase costs
	The solution	Introduce the Team Software Process (TSP)
	Likely benefits	A 32% reduction in cycle time A $2 million one-year savings, $10,000,000 in five years A $950,000 one-time introduction cost About $500,000 in subsequent four-year support costs A one-year return on investment (ROI) of 206% A five-year ROI of 683%
Presentation *Part 2*	The proposal. He briefly described the proposal. He first asked for a minimum commitment to steps 1 and 2 of the introduction program. Second, assuming that the first two steps were successful, he proposed to proceed with the two-team TSP pilot program. Third, after the two teams were underway and the early experience was satisfactory, he planned to proceed with broader TSP introduction.	
Presentation *Part 3*	Description of the TSP. As backup, he prepared a brief description of the TSP and its objectives.	
Presentation *Part 4*	Summary of the TSP benefits obtained by other organizations. Also as backup, he prepared a summary of the available data.	
Presentation *Part 5*	Summary of the introduction plan. As backup, he made a list of the principal phases and decision points in the introduction plan.	
Presentation *Part 6*	Summary of the estimated cost savings. As backup, he made a summary of the cost savings calculations with the key assumptions and supporting data. He also reviewed these figures with finance before the presentation.	
Presentation *Part 7*	Summary of the estimated cycle-time reduction. As backup, he included a summary of the cycle-time improvement calculations with the key assumptions and supporting data.	

Phase 5: Close the Deal

Tom’s final step was to make the presentation and get the order.

Closing Comments

While this example is for a specific process-improvement method, the principles are quite general. To relate this example to the discussion in the December column, the five phases in this example relate to the topics in the December column as follows:

Phase 1	Decide what to do:
	Clearly define what you propose. Understand today’s business environment. Identify the executive’s current hot buttons. Make an initial sanity check.
Phase 2	Estimate the likely costs:
	Start the plan with two or three prototypes. Estimate the one-time introduction costs. Determine the likely continuing costs.
Phase 3	Estimate the likely improvement benefits:
	Document the available experience data. Estimate the expected savings. Decide how to measure the actual benefits. Determine the improvement’s likely impact on the executive’s current key concerns. Identify any other ways that the proposed improvement could benefit the business.
Phase 4	Produce the improvement proposal:
	Produce a presentation to clearly and concisely give this story.
Phase 5	Close the deal.

I hope this example will help you make your own business case. If you have questions, I suggest you look at the December 1999 column, which contains a more generic discussion of these same topics.

Stay Tuned In

In the next issue, we will discuss the issues of convincing tactically focused managers and executives to start a process-improvement program. Following that, a subsequent column will deal with how to move from a tactically based to a strategically based improvement program.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Sholom Cohen, Frank Gmeindl, Julia Mullaney, Jim Over, Mark Paulk, and Bill Peterson.

References

[Musson 1999]

Robert Musson, presentation at the 1999 Software Engineering Institute Symposium, Pittsburgh, PA, August 30 to September 2, 1999.

[Vu]

John Vu, presentation to the U.S. Department of Defense on the Boeing process-improvement program.

[Webb 1999]

Dave Webb & Watts S. Humphrey. "Using the TSP on the TaskView Project," *Crosstalk*. *12*, 2 (February 1999): 3 - 10.

9 Making the Tactical Case for Process Improvement

March 2000

In the December and March columns, I addressed how to make the strategic case for process improvement. It is easier to justify a process improvement program when you deal at a strategic level. Process improvement is a long-term strategic investment, and it is often hard to justify to managers whose principal focus is short term. This column discusses how to handle the shorter term tactical improvement case. We start on the assumption that you are in an organization where the management is focused on tactical issues and where nobody, at least nobody at a senior level, is willing to look beyond the current month or quarter. These managers are generally so consumed by current problems that they cannot consider anything that will pay off in a year or more. While the likelihood of success for a short-term improvement program is low, the situation is not entirely hopeless.

In this column, I talk about tactically based improvement programs and some of the strategies that you can use to bring them off. While there is no guarantee that your efforts will work, there generally are ways to achieve useful results in a relatively brief period. Although you may not succeed, it is better to try something than do nothing. So give it a shot. You might make some useful improvements, and, even better, your success might convince some managers to think and act strategically. This column recommends a series of short-term tactical steps that can lead to a long-term strategic improvement program.

The Tactical Situation

In the typical organization, management claims to be thinking strategically. However, imagine that you work for a company in a highly competitive industry that is struggling to improve its quarterly earnings. While process improvement is accepted as a great idea, nobody will invest any significant money in it. What is more, the costs in your improvement proposal must be justified, and you must show that the costs can be recovered in less than a year without seriously disrupting any project.

Management claims that its goal is to be the industry leader, but the first question managers are likely to ask is, "Who else is doing this and what good has it done them?" While you might be tempted to suggest that they sound like followers instead of leaders, restrain yourself and try to think about how to justify this proposal in a way that management will buy. This is the situation. What can you do?

Possible Approaches

While management knows all of the buzzwords, would like to act strategically, and talks about being the industry leader, managers are hypnotized by their short-term problems. Either business realities will not permit a long-term view, or some higher level manager is under severe pressure to show improved quarterly financial results. Under these conditions, you must ignore all the high-sounding phrases about leadership, quality, and improvement, and focus instead on a few pragmatic steps that will fit the current realities.

The only way to break through management's resistance is to somehow demonstrate that the organization's current short-term problems cannot be fixed with short-term Band-Aids. You must take a strategic view. The two principal approaches you can take are to (1) make the strategic improvement activities tactically attractive, or (2) start a small tactical effort and gradually build it into a strategic improvement program.

Making a Strategic Improvement Program Tactically Attractive

The U.S. Air Force's decision to evaluate bidders based on their process maturity made the Capability Maturity Model (CMM) process-improvement program attractive to many managers. The Air Force evaluations forced many tactically focused managers to invest in process improvement to avoid losing business. This illustrates the advantage of having a customer demand quality improvement: it makes strategic improvement programs tactically attractive.

This strategy will generally work when you have a customer that is interested enough in quality to require that its suppliers commit to improvement programs. If you have such a customer, you can often connect your process-improvement proposal to that customer's demands. If you can get the support of your marketing department, your chances of success are pretty good. While you must keep the scale of your improvement program realistically related to the size of the likely new business, if an important customer insists on a quality program, you probably have a sound basis for a process improvement proposal.

Another approach that is almost as effective is to connect your improvement efforts to an already approved corporate improvement effort. Examples would be obtaining ISO 9000 certification or initiating a 6-sigma software quality improvement effort. Again, if you can show that the improvements you espouse will assist the organization in meeting already established goals, you have a reasonable chance of getting the improvement effort approved.

Build a Small Effort into a Strategic Program

If neither of these strategies work, you probably will not be able to make a strategic program tactically attractive, at least not in the short term. Under these conditions, you must focus on justifying a series of small improvement steps. You could identify one or two narrowly focused efforts that can be completed rather quickly and that don't cost a great deal of money. Then put them into place and use the resulting benefits to help justify taking the next step.

If you are careful about what improvements you pick, and if you build support for each step as you take it, over time, you can probably keep the improvement program moving forward. Then you can gradually convert your short-term tactical activities into a longer term strategic effort.

The critical issue in this situation is getting approval for the initial effort, and then getting the engineers and project managers to support each step as you take it. As long as you can demonstrate that the program is not too expensive and is producing results, and as long as you have the support of the project managers and working engineers, you can probably keep the program going. Then, given a little time, you should be able to show that you are saving the company money and improving project performance. This should allow you to gradually increase the size of the improvement program.

Suggested Tactical Improvement Priorities

In picking improvement efforts, concentrate on activities that are low cost, can be implemented by one or two projects, will produce immediate measurable results, and will attract strong project support. While there are several candidate improvement activities that meet these criteria, the ones that are probably the best bets for organizations at CMM levels 1 or 2 are code inspections, design courses, and the Personal Software Process/Team Software Process (PSP/TSP). These can all be focused on individual projects, and they all support various aspects of a CMM-based process improvement strategy.

Pick efforts that can be implemented without a broad organization-wide effort that requires senior management approval. Then, if the involved project leaders agree to support the proposal, management will generally go along. Because these improvement efforts can be implemented without requiring changes in the entire organization, they are good candidates for quickly demonstrating the benefits of process improvement programs.

Code Inspections

Code inspections can be put into place quickly, and they pay enormous dividends [Fagan 1976, Gilb, Humphrey 1989]. While design inspections could also be helpful, they take more time and money and don't have nearly as high a payoff-unless you have an effective design process in place. Therefore, it is usually best to defer initiating design inspections.

To start a code inspection program, first find a project leader who agrees to implement the initial trial program, and then train all of the engineers who will do the inspections. In addition, get some member of your staff qualified to moderate the inspections and to help the engineers to do the inspections properly. It would also be advisable to hire an expert to teach the initial courses. In starting an inspection program, a number of available references can be helpful [Fagan 1976, Fagan 1986, Gilb, Humphrey 1989], including Appendix C of my book *Introduction to the Team Software Process*, which describes the inspection process [Humphrey 2000a].

After you complete the first code inspections, you can usually use the engineers who did them as references to help convince the leaders of the other projects. While it will take time to put a complete code inspection program in place, it will provide substantial benefits, and it should give you a firm foundation for further improvements.

Design Courses

Until code quality is reasonably good, design improvements generally will not improve test time or product quality substantially. The reason is that poor quality code will result in so many test defects that the design problems will be lost in the noise. However, once you have code inspections in place, you get substantially improved code quality and reduced test time. That is when design courses would make sense.

While it is almost impossible to justify the costs of a design course, you probably will not need to do so. Most people intuitively understand that design is important, and engineers generally will be interested in taking a design course. Start by identifying a project leader who is willing to sponsor the initial test, then find a qualified instructor and get some design courses taught. Assuming that the course is of proper quality, other projects will want to join in, and demand for the course will grow quite quickly.

The only additional requirement is that you have one or two qualified people available to consult with the engineers and to advise them on how to use the design methods when they start applying them on the job. Then, after the design methods are in place, the engineers will have the criteria to judge the quality of the designs that they inspect. That is the time to introduce design inspections.

The PSP and TSP

After successfully introducing the inspection program and the design courses, you will have a substantial level of credibility and a modest staff. Then, you can think about tackling a more challenging improvement effort. This would be a good time to get yourself or one of your people trained as a PSP instructor and TSP launch coach [Humphrey 1995]. While this will take a few months and cost a little money, the training is readily available and will enable you to introduce the PSP and TSP into one or two projects in your organization [SEI 2000].

After getting one or more staff members qualified as PSP instructors, look for a trial project. Training volunteers is the easy-sounding approach, but to successfully introduce the PSP and TSP, you must focus on entire teams, including the managers. Once you identify a team and have a qualified PSP instructor available, you can introduce the PSP and TSP in only three or four months. Even though it will generally be several months before you have data on completed projects, TSP's planning and tracking benefits will be apparent very quickly.

To spread the PSP and TSP to other projects, first convince the managers. The material in my March column should be helpful [Humphrey 2000b]. To convince the engineers to participate, get the first TSP team to meet with the potential new team, and have all the managers leave the room. The engineers from the first project will be most effective at convincing the second team to use the PSP and TSP. Once you have a few TSP teams in place, you will have the data, experience, and support to launch a broader based improvement program.

The Commitment System

Assuming that your tactically focused improvement efforts have so far been successful, you can start to move toward a broader based CMM improvement effort. Be cautious about moving too fast and keep your proposals modest until you are reasonably certain that they will be accepted. The next step is to fix the organization's commitment system.

The commitment process defines the way organizations commit project costs and schedules to customers. Improvements in the commitment system generally produce significant benefits very quickly. The basic principles of the software commitment system are well known [Humphrey 1989, Humphrey 2000a, Paulk]. Improving the commitment system is an important early step in a CMM-based improvement program.

Changing the commitment system is much more difficult than anything you have attempted so far. For some reason, managers who make plans before they commit to constructing a building, starting a manufacturing program, or even taking a trip, do not appreciate the need for planning software projects. Changing the commitment process requires that the managers change their behavior. If you thought changing engineering behavior was difficult, wait until you try to change management behavior. This is why a full-scale CMM-based process improvement program can be difficult to launch.

While the commitment system is probably the most important area to improve, unlike inspections or the PSP/TSP, it cannot be done for only one or two projects. If you can change the commitment system, however, you should be able to launch a full-scale, strategic-based process improvement program. This should include a full CMM-based effort, as well as an expansion of the PSP and TSP to cover all of the development and maintenance projects in the organization.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Anita Carleton, Sholom Cohen, Jim McHale, Julia Mullaney, Mark Paulk, Bill Peterson, and Marsha Pomeroy-Huff.

A Note to My Readers

After publishing the March column, I found that I had made a mistake in calculating the maintenance savings in the example. The maintenance numbers were about twice what they should have been. The March column has now been corrected. I hope this mistake has not caused you any inconvenience or embarrassment.

References

[Fagan 1976]

Fagan, Michael. "Design and Code Inspections to Reduce Errors in Program Development." *IBM Systems Journal, 15*, 3 (1976).

[Fagan 1986]

Fagan, Michael. “Advances in Software Inspections." *IEEE Transactions on Software Engineering, SE-12*, 7 (July 1986).

[Gilb 1993]

Tom Gilb, Dorothy Graham, *Software Inspection*. Edited by Susannah Finzi. Reading, MA: Addison-Wesley, 1993.

[Humphrey 1989]

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

[Humphrey 1995]

W. S. Humphrey, *A Discipline for Software Engineering*. Reading, MA: Addison-Wesley, 1995. http://www.sei.cmu.edu/library/abstracts/books/0201546108.cfm

[Humphrey 2000a]

Watts S. Humphrey, *Introduction to the Team Software Process*, Addison-Wesley, Reading, MA 2000. http://www.sei.cmu.edu/library/abstracts/books/020147719X.cfm

[Humphrey 2000b]

Watts S. Humphrey, “Justifying a Process Improvement Proposal,” *news@sei*, March, 2000. http://www.sei.cmu.edu/library/abstracts/news-at-sei/wattsmar00.cfm

[Paulk 1995]

Mark C. Paulk, Charles V. Weber, Bill Curtis, & Mary Beth Chrissis. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Reading, MA: Addison Wesley, 1995.

[SEI 2000]

Information on PSP and TSP courses is available on the SEI website. http://www.sei.cmu.edu/training/

10 Moving the Goal Posts

July 2000

In this column, I talk about the nature of process improvement and why it is such a dynamic and challenging field. The future will be much like the past in many respects, but it will also be very different. However, as we look ahead, there are some reasonably reliable guides that can help us to address the problems we will face.

Brownian Motion

Just about every time I visit an engineering organization, the people tell me, “We’re different.” Of course they are. We are all different, but what is surprising is how often truly different organizations behave in the same way. However, it is also surprising how often seemingly similar organizations, when faced with nearly identical conditions, behave quite differently. People are both predictable and unpredictable. Much like Brownian motion in physics, there is no way to precisely predict individual behavior. However, on average, overall behavior is highly predictable. So, what does this mean for process improvement? Essentially, the following:

- First, there is no one best way.
- Second, every situation is different. Each solution must consider the people and their backgrounds, beliefs, and circumstances.
- Third, the lessons of the past are the only practical guides we have for the future. While we cannot predict precisely what will work in any specific case, we can establish highly reliable general guidelines.
- Fourth, the principles behind the quality movement are just as sound today as they were in the past. Those who argue that the new Internet age changes all the old truths will continue reliving the same history that many of us have painfully survived.

What these lessons tell us is that a single-minded approach to solving any human problem will almost certainly be wrong, if not for everybody, at least in many cases. There is no single best answer. People are extraordinarily creative, both in the ways that they solve problems and in how they create problems. Therefore, we must recognize that problems will change, and we must continually seek newer and better ways to address the problems that we face at each point in time.

When the Problems Change, the Solutions Must Also Change

The other day, I read the following newspaper headline: “The quality of U.S. automobiles lags behind Japan and Europe.” As Yogi Berra once said, this is “déjà vu all over again.” After 20-plus years, can quality still be a problem for Detroit? It almost certainly is, and the best way to tell is that the General Motors board of directors cut executive bonuses. That is a guaranteed way to get management’s attention.

GM, Ford, and Chrysler have been working on quality improvement for more than 20 years, but they still have about 150 defects per 100 new cars. However, unlike 20 years ago, these are not primarily manufacturing defects. Most are design problems. Detroit solved the quality problems of 20 years ago, and if the Japanese had not kept moving the goal posts, Detroit would be in fat city. But the world did change, and Detroit is still dead last in the quality sweepstakes.

The world changes, and it does not change all by itself. Everything we do changes it. In another lesson from physics, Heisenberg showed that you can know a particle's location or its velocity but not both. When you measure one, you change the other. People are just like that. As soon as you fix the process, the problem changes. Does that mean that we should give up? Not at all; it just means that we cannot relax. We must keep thinking, and resist the temptation to blindly rely on the solutions and formulas of the past. Continue to follow the same principles, certainly, but don't blindly follow the same path. Sooner or later it will lead to a dead end.

Finding the Goal Posts

While process problems are often unique, they all stem from human failings, and these are common to all of us. Because the same human failings have persisted through the ages, we cannot expect to eliminate them. The process improvement challenge is to devise ways to live with and compensate for normal human behavior. We must recognize, however, that soon after we compensate for a given set of failings, human nature will find creative countermeasures. So, in spite of all our efforts, the battle for improvement will continue indefinitely. Hopefully, however, technology will keep improving and each step will move the goal posts a little further down the field.

Human Failings

While software professionals are marvelously creative and highly energetic, we sometimes feel lazy or want to take a break. We are also a race of procrastinators, and when we can't avoid or put off some difficult or unpleasant task, we try to replace it with an easier task or get someone else to do it. If we find that we still must do the job, we tend to do it as quickly and superficially as we can get away with. This means that for every complex and difficult task, the process improvement challenge is to devise ways to get people to consistently do their work in a highly professional way.

What makes this so challenging is that once we figure out some way to do this, it is only a matter of time before people devise a clever way around our fancy new process. Take estimating, for example. The Capability Maturity Model (CMM) calls for engineers to be involved in and agree with the project estimate. However, soon after an organization puts a new planning procedure in place, some group will almost certainly find an estimating method that uses expert estimators or complex and arcane tools. Experts will then make the estimates and the engineers won't be involved. Even though this destroys the intent of the planning process, unless processes are defined very carefully, people will adopt new practices that conform to the letter of the defined process but not to its intent.

What this Means for Process Improvement

What this means for you and me is that process improvement must not be directed at only the process. The principal objective must be to change human behavior. However, to change human behavior, we must consider and compensate for normal human failings.

For example, we now find that even in CMM Level 5 organizations, people have learned to compensate for their new processes. In some of the Level 5 organizations I have visited, the measurement and process analysis work is handled by the process and quality groups, and the engineers continue to work essentially as they did at Level 1. This totally misses the point of CMM Levels 4 and 5, which is to have engineers *use* data, not just gather and report it. This implies that even the goal posts defined by the CMM levels must be moved to keep pace with our rapidly changing technology.

In the last analysis, to improve engineering performance, organizations must change the behavior of the engineers and their managers. If you find that some change has stopped producing the desired results, find out why and then devise another improvement to solve the new problems.

The Implications for the Future

We have made great strides in the last 10 or more years, and we must continue to build on our successes. However, the goal posts are moving, and the problems we will face in the future will almost certainly be different from those of the past. Think of it this way: You could build a 10-foot boat in your garage, but a 1,000-foot ship would require entirely different tools, technologies, and processes. Similarly, in transportation, going from 3 to 300 miles per hour requires several changes in technology. In the software business, we think nothing of factors of 100. We use the same tools, methods, and processes for a program with 10,000 lines of code (LOC) as we do for a 1,000,000-LOC programming system.

Our ability to master the software-intensive technologies of the future will be largely guided by the ability of engineering teams to match their behavior to the more demanding tasks they will face. We cannot expect that our current tools, technologies, and processes will be adequate in the future. The challenges will keep increasing, and we must continually evolve our methods to keep pace. We must think of process improvement in multi-dimensional terms and include the educational system, as well as industry. An informed customer community will also be important, and we must consider all levels of the engineering organization: executives, managers, teams, and engineers. Much as in the automobile industry, we must retain the solutions of the past, but we must broaden our perspective to consider all the relevant aspects of the problem.

While it is always risky to predict the future, some trends are now pretty obvious:

- Systems will get larger, more complex, and more integrated.
- Engineering teams must also become more highly integrated.
- Compatibility, reliability, usability, privacy, and security will be increasingly important.
- While schedules must be as short as possible, they must be absolutely reliable.
- The quality of every engineer’s personal work will be even more important than it has been in the past.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Noopur Davis, Jim McHale, Don McAndrews, Julia Mullaney, and Marsha Pomeroy-Huff.

11 The Future of Software Engineering: Part I

First Quarter, 2001

In this and the next few columns, I discuss the future of software engineering. This column focuses on trends in application programming, particularly as they concern quality. In subsequent columns, I address programming skills, trends in systems programming, and the implications of these trends for software engineering in general. While the positions I take and the opinions I express are likely to be controversial, my intent is to stir up some debate and hopefully to shed light on what I believe are important issues. Also, as is true in all of these columns, the opinions I express are entirely my own.

Current Trends

Some 50 years ago when I finished graduate school, I started to work with computers. For almost all of the intervening time, people have been dreaming up new and better ways to use computing devices and systems. While the volume of application development has grown steadily, so has the list of applications that people want to develop. It seems that the more programs we write, the more we understand what we need, and the more programs we add to the application-development backlog. So far, the rate of growth in application demand has continued to accelerate.

As economists say, unsustainable trends are unsustainable. However, to date, there is no sign that this growth in demand is slowing. I suspect that it will continue for the foreseeable future, though we will certainly see changes in the types of application programs. The reason for this growth is that application programming is a very effective way to meet many kinds of human needs. As long as people continue devising new and cleverer ways to work and to play, we can expect the growth in computer applications to continue. The more we learn about computing systems and the more problems we solve, the better we understand how to address more complex and interesting problems. Therefore, because human ingenuity appears to be unlimited, the number of programs needed in the world is also essentially unlimited. This means that the principal limitation on the expanding use of computing systems is our ability to find enough skilled people to meet the demands.

In discussing these topics, I break the software world into two broad categories: applications and systems. I won’t try to clearly define the dividing line between these categories because that line is both indistinct and changing. By “application programming,” I refer to solving human problems with computing systems. Here the focus is on defining the problem and figuring out how to solve it with an automated system. Systems programming focuses on how to provide automated systems, aids, tools, and facilities to help people produce and run application programs. In looking toward the future in these areas, I believe that the two most significant issues concern software quality and the demand for skilled people. I address application software quality first.

Application Program Quality

The quality of application software today is spotty at best. A few programs are of high quality and many are downright bad. The discussion of application quality has two principal parts. First, computers are being used for progressively more critical business applications. This means that, even at current quality levels, the impact of software defects will grow. This implies that the demand for high-quality application software will also grow.

To date, people buy and use software without apparent concern for their quality. While they complain about quality problems, software quality has not yet become a significant acquisition consideration. Until it is, we cannot expect suppliers to measure and manage the quality of their products. However, as has been amply demonstrated in other fields, when quality is not measured and managed, it is generally poor.

When the public gets concerned about quality, these attitudes will quickly change. It will not take many high-profile disasters to cause executives to worry. Then, they are likely to demand quality guarantees and warranties from their suppliers before they entrust their businesses to new computing systems. When customers cannot be assured of the quality of a software package, they will either go to a competitor or forgo the application entirely and continue using prior methods. This would not be because automated methods would not have been helpful, but simply because the risk of failure would be too high.

Growing Program Complexity

The second part of the program quality discussion concerns size and complexity. The size and complexity of application programs is increasing. The size data in Figure 1 show just how fast this growth has been. The IBM size measures are from my personal recollection, the Microsoft NT size data are from published reports, and the spacecraft size data are from Barry Boehm [Boehm 1981, Zachary 1994]. The TV data are for the embedded code in television sets and are from a talk by Hans Aerts and others at the European SEPG conference in June 2000. According to my prior definitions, the IBM and Microsoft products are system software while the spacecraft and TV programs are application software.

These data show that the size of the software required for various kinds of systems and applications has been growing exponentially for the past 40 years. The trend line in the center of the chart shows a compound growth rate of ten times every five years. This is the same as Moore’s law for the growth in the number of semiconductors on a chip, or a doubling every 18 months.

While the size growth of system software has been phenomenal, it appears to be slowing, at least from the appearance of the IBM and NT data in Figure 1. I discuss these systems software trends in a later column. The critical point from an application quality point of view is that the growth trend for applications software appears to be continuing.

The Defect Content of Programs

Assuming that the same methods are used, the number of defects in a program increases linearly with its size. However, the rate at which application program users experience problems is largely determined by the number of defects in a program rather than their density. Therefore, even though the defect density may stay about the same or even improve, merely producing larger programs with the same methods will produce progressively less reliable systems. So, either the quality of future application software must improve—at least in step with the increasing sensitivity of the applications—or businesses must limit their use of computing systems to less critical applications.

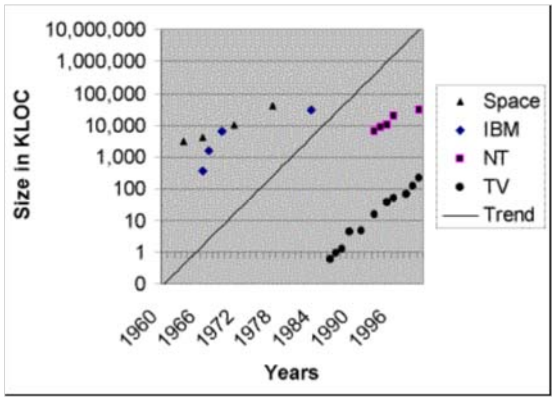

Figure 11-1: Program Size Growth

To see why program reliability depends on defect numbers instead of defect density, consider an example. A 200 KLOC (thousand lines of code) program with 5 undetected defects per KLOC would have 1,000 defects. If you replaced this program with an enhanced program with 2,000 KLOC and the same defect density, it would have 10,000 defects. Assuming that the users followed a similar usage cycle with the new application, they would exercise the larger program at about the same rate as the previous one. While this would presumably require a faster computer, the users would be exposed to ten times as many defects in the same amount of time. This, of course, assumes that the users actually used many of the new program's enhanced functions. If they did not, they presumably would not have needed the new program.

An Application Quality Example

Oil exploration companies use highly sophisticated programs to analyze seismic data. These programs all use the same mathematical methods and should give identical results when run with identical data. While the programs are proprietary to each exploration company, the proprietary parts of these programs concern how they process enormous volumes of data. This is important because the volume of seismic data to be analyzed is often in the terabyte range.

A few years ago, Les Hatton persuaded several oil-exploration companies to give him copies of nine such programs [Hatton 1994]. He also obtained a seismic exploration dataset and ran each of these nine programs with the identical data. The results are shown in Figure 2. Here, the range of calculated values is shown for several consecutive program iterations. As you can see, this range generally increased with the number of cycles, and after a few runs, it reached 100%. When one of these companies was told about some of the conditions under which its program gave unusual results, the programmers found and corrected the mistakes, and the program's next results agreed with the other programs.

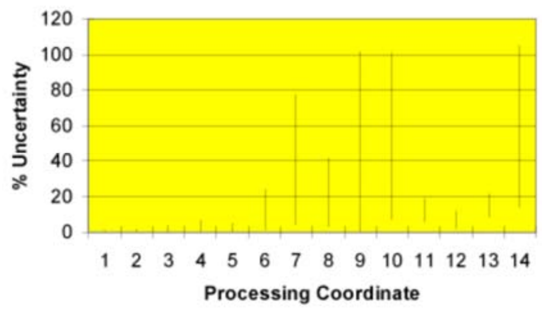

Figure 11-2: Growth in Seismic Program Uncertainty

The results produced by these oil-exploration programs contained errors of up to 100%, and these results were used to make multi-million-dollar decisions on where to drill oil wells. Based on this study, it appears that these programs provided little better guidance than throwing dice. I am not picking on these programs as particularly poor examples. Their quality appears to be typical of many application programs.

The Quality Problem

In discussing the quality of application programs, we need to consider the fact that defective programs run. That is, when engineers produce a program and then run extensive tests, they generally can get it to work. Unfortunately, unless the tests were comprehensive, the tested program will likely contain a great many defects.

Any testing process can only identify and fix the defects encountered in running those specific tests. This is because many program defects are sensitive to the program's state, the data values used, the system configuration, and the operating conditions. Because the number of possible combinations of these conditions is very large, even for relatively simple programs, extensive testing cannot find all of the defects.

Since the size of application programs will continue to increase, we need to consider another question: will we be able to test these programs? That is, how well does the testing process scale up with program size? Unfortunately, the answer is not encouraging. As programs get larger, the number of possible program conditions increases exponentially. This has two related consequences.

1. The number of tests required to achieve any given level of test coverage increases exponentially with program size.
2. The time it takes to find and fix each program defect increases somewhere between linearly and exponentially with program size.

The inescapable conclusion is that the testing process will not scale up with program size. Since the quality of today's programs is marginal and the demand for quality is increasing, current software quality practices will not be adequate in the future.

The Impact of Poor Quality

As the cost of application mistakes grows and as these mistakes increasingly impact business performance, application program quality will become progressively more important. While this will come as a shock to many in the software community, it will be a positive development. The reason is that suppliers will not generally pay attention to quality until their customers start to demand it. When software quality becomes an important economic consideration for businesses, we can expect software organizations to give it much higher priority.

While one could hope that the software industry would recognize the benefits of quality work before they are forced to, the signs are not encouraging. However, improved product quality would be in the industry's best interests. It would mean increased opportunities for computing systems and increased demand for the suppliers' products. This would also mean continued growth in the demand for skilled software professionals.

In the next column, I discuss the need for application programming skills and how this need is directly related to the quality problem. Following that, I discuss related trends in systems programming and the implications of these trends for software engineering.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Sholom Cohen, Noopur Davis, Alan Koch, Don McAndrews, Julia Mullaney, Bill Peterson, and Marsha Pomeroy-Huff.

References

[Boehm 1981]

Boehm, Barry. *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[Hatton 1994]

Hatton, Les. "How Accurate is Scientific Software?" *IEEE Transactions on Software Engineering*, *20*, 10 (October 1994): 785-797.

[Zachary 1994]

Zachary, G. Pascal. *Showstopper!* New York: The Free Press, 1994.

12 The Future of Software Engineering: Part II

Second Quarter, 2001

This is the second of several columns on the future of software engineering. The first column focused on trends in application programming, particularly related to quality. This column reviews data on programmer staffing and then covers application-programming skills. Future columns deal with trends in systems programming and the implications of these trends for software engineering and software engineers.

In my previous column ("The Future of Software Engineering: Part I" on page 61), I started a discussion of the future of software engineering and reviewed the trends in application programming. In this column, I consider the growing demand for people to write application programs. I also explore the implications of the current trends in application programming. In the next few columns, I will examine the trends in systems programming and comment on the implications of these trends for software engineering and software engineers. While the positions I take and the opinions I express are likely to be controversial, my intent is to stir up debate and hopefully to shed some light on what I believe are important issues. Also, as is true in all of these columns, the opinions are entirely my own.

Some Facts

The demand for software engineers is at an all-time high, and it continues to increase. Based on recent census data, there were 568,000 software professionals in the U.S. in 1996. In 2007, there are projected to be 697,000 [Clark 2000]. Since 177,000 are also projected to leave the field during this time, this implies a ten-year need for more than 300,000 new programmers. That is a 50% gross addition to the current programming population.

The Census Bureau estimate of 568,000 programmers seems low to me, and I suspect this is because of the criteria used to determine who was counted as a programmer. Howard Rubin quotes a number of 1.9 million programmers as the current U.S. programming population [Rubin 1999]. I have also seen data showing that the number of programmers in the U.S. doubled from 1986 to 1996. While good data are sparse for such an important field, the demand for programmers has clearly increased in the past ten years, and it is likely to continue increasing in the future.

If you consider that most professionals in most fields of engineering and science must now write at least some software to do their jobs, the number of people who write, modify, fix, and support software must be very large. If the growth trends implied by the census data apply to the entire population of casual and full-time programmers, the demand for new programmers in the next ten years is likely to run into the millions.

Future Needs

Judging by past trends, it is clear that just about every industrial organization will need more people with application programming skills and that most programming groups will be seriously understaffed. Since many software groups are already understaffed, and the current university graduation rate of software professionals is only about 35,000 a year, we have a problem [U.S. 2000]. In general terms, there are only two ways to address the application-programming problem.

1. Somehow increase the supply of new programmers.
2. Figure out how to write more programs without using more programmers.

Since it takes a long time to increase the graduation rate of software professionals, the principal approach to the first alternative must be to do more of what we are doing today—that is, to move more software offshore and to bring more software-skilled immigrants into the U.S.

While many organizations are establishing software laboratories in other countries, particularly India, this is a limited solution. The principal need is for skilled software professionals who understand the needs of businesses and can translate these needs into working applications. There is no question that the coding and testing work could be sent offshore, but that would require good designs or, at least, clear and precise requirements. Since producing the requirements and design is the bulk of the software job, going offshore can only be a small part of the solution.

Obtaining more software-skilled immigrants is an attractive alternative, particularly because India alone graduates about 100,000 English-speaking software professionals a year. However, the U.S. has tight visa restrictions, and many other groups also have claims on the available slots. Also, since the demand for software skills is increasing rapidly in India, and since many Indian professionals can now find attractive opportunities at home, the available numbers of Indian immigrants will likely be limited in the future.

The Automobile Industry Analogy

To examine the alternative of writing more programs without adding more programmers, consider the automobile industry. Back before Henry Ford, only the wealthy could afford cars. Then Henry Ford made the automobile affordable for ordinary folks. Once the manufacturers started catering to the needs of the masses, the automobile industry changed rapidly.

Many innovations were required before people could feel comfortable driving without a chauffeur. They needed the closed automobile body, automatic starters, heaters, clutches, transmissions, and a host of other progressively more automatic and convenient features. This combination of innovations made operating an automobile simple and easy for almost anyone.

With the aid of these innovations, people could learn to drive without chauffeurs. When all this happened, the chauffeur business went into a tailspin. Soon, as the comfort, convenience, and reliability of cars increased, driving an automobile was no longer a specialty; it became a general skill required of just about everyone. Today, most people learn to drive an automobile before they get out of high school. While there are still professional drivers, the vast majority of driving is now done by the general public.

The Computer Field Today

Today, the computer field is much like the early days of the automobile industry. Many professionals have learned to use computing systems, but few are willing to rely on them for critical work, at least not without expert help and support. In the computer field, the chauffeur equivalents are with us in the guise of the experts who develop applications, install and tailor operating systems, and help us recover from frequent system crashes and failures. Even on the Internet, our systems today often exhibit strange behavior and present us with cryptic messages. While these systems are far easier to use than before, they are not yet usable by the general public.

For computing systems to be widely used, we need systems that work consistently and are problem free. We also need support systems that serve the same functions as automobile starters and automatic transmissions. Then professionals in most fields will be able to automate their own applications without needing skilled programmers to handle the arcane system details.

Another prerequisite to the widespread use of computing systems is that the professionals in most fields be able to produce high-quality application programs with little or no professional help. The real breakthrough will come when it is easier to learn to write good software than it is to learn about most business or scientific applications. Then, instead of requiring that skilled software people learn about each application area, it will be more economical and efficient to have the application experts learn to develop their own software. At that point, software engineering will become a general skill much like driving, mathematics, or writing, and every professional will be able to use computing systems to meet the vast bulk of his or her application needs.

Growing System Size and Complexity

While such a change will be an enormous help, it will not address all aspects of application programming. To see why, consider the trends in the size and complexity of application programs. If history is any guide, future application programs will be vastly larger and more complex than they are today. This means that the development of such systems will change in a number of important ways.

As I wrote in the prior column, the first and possibly most important change is in quality. Those who need software simply will be unable to use programs to conduct their businesses unless they are of substantially higher quality than they are today. The second trend is equally significant: the current cottage-industry approach to developing application programs must give way to a more professional and well-managed discipline. This is not just because of the increasing size of the programs and their more demanding quality specifications, but also because the business of producing such programs will grow beyond the capability of most people to master quickly.

In other words, the day has largely passed when we could hire somebody who was reasonably familiar with the programming language of choice and expect him or her to rapidly become productive at developing application programs. As application programs become larger and more sophisticated, the required application knowledge and experience will increase as well. Soon, the cost and time required to build this application knowledge will be prohibitive. Therefore, a host of new methods must be developed to make application programming more economical and far less time consuming than it is today.

Reuse

My argument to this point has concerned getting more people to write programs. However, there is another alternative: finding ways to produce more applications with fewer people. One proposed solution to this challenge is through reuse. While this seems like an attractive possibility, recent history has not been encouraging. In fact, history indicates that reuse technology will be largely confined to building progressively larger libraries of language and system functions. Unfortunately, this added language complexity will cause other problems. This is not because reuse is unattractive; it is just at too low a level to address the application needs of most users.

The software community has been adding functional capability to programming languages for the 47 years since I wrote my first program. This approach has not solved the programming problems of the past, nor is it likely to solve those of the future. The principal reason is that by adding more microscopic functions to our languages, we merely restate the application development problem in slightly richer terms.

For example, when I wrote my first program we had to control the starting and stopping of the I/O devices and the transfer of each character. Now such functions are handled automatically for us, but we are faced instead with much more sophisticated languages. Instead of a simple language you could summarize on a single sheet of paper, we now need entire textbooks.

Granted, increased language richness reduces the detailed system knowledge required to manage the computer's functions, but it still leaves us with the overall design problem, as well as the problem of determining what the design is supposed to do for the user. Then, the application programmer has the final challenge of translating the design into a functioning and reliable program.

This leads to the problem that will force us out of the cottage-industry approach to programming. That is the simple impossibility of quickly becoming fluent in all the languages and functions needed to produce the complex application systems of the future. While reuse in traditional terms may be helpful for the professional programming population, it is directly counter to the need to make our technology more accessible to people who are not full-time programming professionals.

Packaged Applications

To handle the volume needs of many users, companies are starting to market packaged applications much like those offered by SAP and Oracle. That is, they produce essentially prepackaged application systems that can be configured in prescribed ways. Rather than custom-designing each application, this industry will increasingly develop families of tailorable application systems. The users will then find the available system that comes closest to meeting their requirements and use its customization capabilities to tailor the system to their business needs.

To make these systems easily tailorable by their customers, companies will design their systems with limited, but generic, capabilities. Then, in addition to tailoring the system, the users must also adjust their business procedures to fit the available facilities of the system. As the experiences of SAP and others have demonstrated, this approach is not trouble free, but it can provide users with large and sophisticated application systems at much lower cost than a full custom-application development.

Judging by the growth of SAP, Oracle, and others, this has been an attractive strategy. Rather than developing applications to meet an unlimited range of possible user needs, users will increasingly adapt their business operations to fit the functions of the available application systems. While this represents a form of reuse, it is at a much higher level than the approaches generally proposed, and it generally requires a thoughtfully architected family of application products or product lines. Just as with the transportation, housing, and clothing industries, for example, once people see the enormous cost of customized products, they usually settle for what they can find on the rack.

Application Categories

Application development work in the future will likely involve three categories of work:

1. developing prepackaged applications that users can tailor to their needs
2. tailoring business systems to use prepackaged application systems
3. developing and supporting unique applications that cannot be created with prepackaged software

The programmers needed for the first category will be professionals much like those needed for developing systems programs, but they will generally have considerable application knowledge. I will write more about this category in later columns.

For category two, we will probably see a substantial growth in the volume of application customization. The people doing this work will be more like business consultants than programmers, and many will not even know how to design and develop programs. These people will be thoroughly trained in the packages that they are customizing and helping to install.

The reason for the third category is that, even though the prepackaged application strategy will likely handle most bread-and-butter applications, it will not handle those applications needed to support new and innovative business activities. Since these applications will not have been used before, nobody will know how to produce prepackaged solutions. As a result, there will be a volume of applications that cannot be solved by prepackaged solutions. Therefore, even with a wide variety of available prepackaged applications, the need for customized application development will not disappear.

Custom Application Programming

Custom application work must be handled by people who know how to write programs and who also understand the application specialty. These people must be experts on a wide variety of specialties, and must be able to write high-quality programs. For these people, we must develop suitable methods and training—to help them develop quality programs on their own. Even though they will not work full time as professional programmers, I believe that this category of programmer will ultimately comprise the vast majority of the people writing programs. Since they will not spend all—or even most—of their time writing programs, we must simplify our languages and develop new languages that are designed for casual use. We must develop tools and support systems that will help these people to produce high-quality programs at reasonable rates and costs. We must also tailor support systems so that writing applications to run on top of a well-designed systems program will not require extensive technical support and hotline consultation.

In sum, what I am proposing is that, instead of having more and more trained programming professionals, we will solve our programming needs by teaching everybody to program. Although these people will not be professional programmers, they will be even more important to the software community because they will be our most demanding customers. They will be operating at the limits of the systems we software professionals provide, and they will be the first to identify important new opportunities. Therefore, they will probably be the source of much of the future innovation in our field.

In the next few columns, I will write about the trends in system programs, what they mean for the programming community, and the implications of these trends for software engineering.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of David Carrington, Sholom Cohen, Don McAndrews, Julia Mullaney, Bill Peterson, and Marsha Pomeroy-Huff.

References

[Clark 2000]
David Clark. “Are Too Many Programmers Too Narrowly Trained?” *IEEE Computer* (March 2000): 12-15.

[Rubin 1999]
Howard Rubin. “Global Software Economics.” *Cutter IT Journal* (March 1999): 6-21.

[U.S. 1999]
The U.S. Department of Education, National Center for Education Statistics. National Education Survey (HEGIS), “Degrees and Formal Awards Conferred.” Completions survey and Integrated Postsecondary Education Data System (IPEDS), June 1999.

13 The Future of Software Engineering: Part III

Third Quarter, 2001

In the previous two columns, I began a series of observations on the future of software engineering. The first two columns covered trends in application programming and the implications of these trends. The principal focus was on quality and staff availability. In this column, I explore trends in systems programming, including the nature of the systems programming business. By necessity, this must also cover trends in computing systems.

The Objectives of Systems Programs

The reason we need systems programs (or operating systems) is to provide users with virtual computing environments that are private, capable, high performance, reliable, usable, stable, and secure. The systems programming job has grown progressively more complex over the years. These programs must now provide capabilities for multi-processing, multi-programming, distributed processing, interactive computing, continuous operation, dynamic recovery, security, usability, shared data, cooperative computing, and much more.

Because of the expense of developing and supporting these systems, it has been necessary for each systems program to support many different customers, a range of system configurations, and often even several system types. In addition, for systems programs to be widely useful, they must provide all these services for every application program to be run on the computing system, and they must continue to support these applications even as the systems programs are enhanced and extended. Ideally, users should be able to install a new version of the systems program and have all of their existing applications continue to function without change.

Early Trends in Systems Programs

At Massachusetts Institute of Technology (MIT), where I wrote my first program for the Whirlwind Computer in 1953, we had only rudimentary programming support [Humphrey 1988].[7] The staff at the MIT computing center had just installed a symbolic assembler that provided relative addressing, so we did not have to write for absolute memory locations. However, we did have to program the I/O and CRT display one character at a time. Whirlwind would run only one program at a time, and it didn't even have a job queue, so everything stopped between jobs.

Over the next 10 years, the design of both computing machines and operating systems evolved together. There were frequent tradeoffs between machine capabilities and software functions. By the time the IBM 360 system architecture was established in 1963, many functions that had been provided by software were incorporated into the hardware. These included memory, job, data, and device management, as well as I/O channels, device controllers, and hardware interrupt systems.

[7] I was a computer systems architect at Sylvania Electric Products in Boston at the time.

Computer designers even used micro-programmed machine instructions to emulate other computer types.

Microprogramming was considered hardware because it was inside the computer's instruction set, while software was outside because it used the instruction set. While software generally had no visibility inside the machine, there were exceptions. For example, systems programs used privileged memory locations for startup, machine diagnostics, recovery, and interrupt handling. These capabilities were not available to applications programs.

While the 360 architecture essentially froze the border between the hardware and the software, it was a temporary freeze and, over the next few years, system designers moved many software functions into the hardware. Up to this point, the systems programs and the computer equipment had been developed within the same company. Therefore, as the technology evolved, it was possible to make functional tradeoffs between the hardware and the software to re-optimize system cost and performance.

One example was the insertion of virtual memory into the 360 architecture, which resulted in the 370 systems [Denning 1970].[8] Another example was the reduced instruction set computer (RISC) architecture devised by John Cocke, George Radin, and others at IBM research [Colwell 1985]. Both of these advances involved major realignments of function between the hardware and the software, and they both resulted in substantial system improvements.

With the advent of IBM's personal computer (PC) in 1981, the operating system and computer were separated, with different organizations handling the design and development of hardware and software. This froze the tradeoff between the two, and there has since been little or no movement. Think of it! In spite of the unbelievable advances in hardware technology, the architecture of PC systems has been frozen for 20 years. Moore's law says that the number of semiconductors on a chip doubles every 18 months, or 10 times in five years. Thus, we can now have 10,000 times more semiconductors on a single chip than we could when the PC architecture was originally defined.

Unfortunately this architectural freeze means that software continues to provide many functions that hardware could handle more rapidly and economically. The best example I can think of is the simple task of turning systems on and off. Technologically speaking, the standalone operating system business is an anachronism. However, because of the enormous investments in the current business structure, change will be slow, as well as contentious and painful.

The Operating Systems Business

Another interesting aspect of the operating systems business is that the suppliers' objectives are directly counter to their user's interests. The users need a stable, reliable, fast, and efficient operating system. Above all, the system must have a fixed and well-known application programming interface (API) so that many people can write applications to run on the system. Each new appli-

[8] At this time I was managing the systems software and computer architecture groups at IBM.

cation will then enhance the system's capabilities and progressively add user value without changing the operating system or generating any operating system revenue. Obviously, to reach a broad range of initial users, the operating systems suppliers must support this objective, or at least appear to support it.

The suppliers' principal objective is to make money. However, the problem is that programs do not wear out, rot, or otherwise deteriorate. Once you have a working operating system, you have no reason to get another one as long as the one you have is stable, reliable, fast, and efficient and provides the functions you need. While users generally resist changing operating systems, they might decide to buy a new one for any of four reasons.

1. They are new computer users.
2. They need to replace their current computers and either the operating system they have will not run on the new computer or they can't buy a new computer without getting a new operating system.
3. They need a new version that fixes the defects in the old one.
4. They need functions that the new operating system provides and that they cannot get with the old system.

To make money, operating systems suppliers must regularly sell new copies of their system. So, once they have run out of new users, their only avenue for growth is to make the existing system obsolete. There are three ways to do this.

1. Somehow tie the operating system to the specific computer on which it is initially installed. This will prevent users from moving their existing operating systems to new computers. Once the suppliers have done this, every new machine must come with a new copy of the operating system. While this is a valid tactic, it is tantamount to declaring that the operating systems business is part of the hardware business.
2. Find defects or problems in the old version and fix them only in the new version. This is a self-limiting strategy, but its usefulness can be prolonged by having new versions introduce as many or more defects as it fixes, thus creating a continuing need for replacements. The recent Microsoft ad claiming that "Windows 2000 Professional is up to 30% faster and 13 times more reliable than Windows 98," looks like such a strategy, but I suspect it is just misguided advertising [WSJ 2001]. The advertising community hasn't yet learned what the automotive industry learned long ago: never say anything negative about last year's model.
3. Offer desirable new functions with the new version and ensure that these functions cannot be obtained by enhancing the old version. This is an attractive but self-limiting strategy. As each new function is added, the most important user needs are satisfied first so each new function is less and less important. Therefore, the potential market for new functions gradually declines.

This obsolescence problem suggests a basic business strategy: gradually expand the scope of the operating system to encompass new system-related functions. Examples would be incorporating security protection, file-compression utilities, Web browsers, and other similar functions directly into the operating system. I cover this topic further in the next column.

The obvious conclusion is that, unless the operating systems people can continue finding revolutionary new ways to use computers, and unless each new way appeals to a large population of users, the operating system business cannot survive as an independent business. While its demise is not imminent, it is inevitable.

In the next column, I will continue this examination of the operating systems business. Then, in succeeding columns, I will cover what these trends in applications and systems programming mean to software engineering, and what they mean to each of us. While the positions I take and the opinions I express are likely to be controversial, my intent is to stir up debate and hopefully to shed some light on what I believe are important issues. Also, as is true in all of these columns, the opinions are entirely my own.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Marsha Pomeroy-Huff, Julia Mullaney, Bill Peterson, and Mark Paulk.

References

[Colwell 1985]
R.P. Colwell, et al. "Instruction Sets and Beyond: Computer, Complexity, and Controversy," *IEEE Computer, 1819* (September 1985).

[Denning 1970]
P.J. Denning, "Virtual Memory," *Computing Surveys, 2,* 3 (September 1970): 153-189.

[Humphrey 1998]
W.S. Humphrey, "Reflections on a Software Life," *In the Beginning, Recollections of Software Pioneers*, Robert L. Glass, ed. Los Alamitos, CA: The IEEE Computer Society Press, 1998, 29-53.

[WSJ 2001]
The Wall Street Journal (February 2001): A18.

14 The Future of Software Engineering: Part IV

Fourth Quarter, 2001

This is the fourth of five columns on the future of software engineering. The first two columns focused on trends in application programming, particularly related to quality and staffing. The previous column ("The Future of Software Engineering: Part III" on page 73) covered systems programming and the systems-programming business. In this column, I explain the three kinds of operating-systems (OS) businesses and predict where these businesses are likely to go in the future.

Systems Programming

To refresh your memory, the principal points made in the previous column ("The Future of Software Engineering: Part III" on page 73) were as follows:

- The objective of systems programs is to provide users with a virtual computing environment that is private, secure, and reliable.
- Over time, systems-control functions have gradually migrated from software to hardware. For example, when I wrote my first program, we had to read and handle each character. Hardware now does that. Similarly, many functions that were previously handled by software, such as memory management, interruption handling, and protection, are now handled by hardware. While the advent of personal computers temporarily halted this migration, it is driven by technology and will almost certainly resume in the future.
- The objectives of the organizations that make and market operating systems and the objectives of their users are naturally opposed. To maintain and grow their businesses, operating-system suppliers must continually enhance their systems or otherwise entice their users to upgrade. Conversely, computer users seek stability, reliability, and compatibility, and generally want to continue using their current versions. Since operating systems do not wear out, rot, or otherwise deteriorate, the installed life of old versions of operating systems could become very long indeed.

The principal conclusion of the previous column ("The Future of Software Engineering: Part III" on page 73) was that a standalone business for operating systems is not viable over the long term. Ultimately, the supply of attractive new functions will be depleted. Then, while people will need occasional enhancements and new operating-system versions for new hardware, they will prefer to stay with their existing operating-system version, rather than buy a new one. This rather stable operating-system business will likely be viable, but it will be very different from what we know today.

Even though the operating-system business will probably not survive by itself, it would be a natural companion to a hardware business. In that case, you might expect the hardware companies to absorb the operating-systems businesses. However, in today's world, it seems more likely that the operating-systems businesses will absorb the hardware companies.

The Internet

One might argue that the Internet changes everything. It is true that the Internet is a radical change and that it presents enormous opportunities for innovation and creativity. However, since the Internet revolution is still in its infancy, there are likely to be many surprises. But, since I am being controversial in this column, I might as well stick my neck out and hazard some predictions.

One likely way to couple computers and the Internet would be essentially to move the Internet inside the application programming interface (API). This would use the Internet to reference data and programs, regardless of their physical location. It would also presumably permit multiple remote systems to cooperate much as they could if they were at the same location. Producing these capabilities would be a substantial challenge and, when accomplished, would provide a glorified data, file management, and distributed computing capability. While such offerings will almost certainly be started by the software businesses, the Internet can be viewed as just another device. As advantageous as this Internet capability would be, its support would best be handled by hardware. Ultimately, the most efficient and cost-effective way to handle device support is as a hardware facility and not as a new operating-system function.

Second, the idea that people will use the Internet as a pervasive computing resource is not realistic. People will certainly use the Internet for communications, for retrieving programs and data, and for incidental and cooperative processing. However, most will not use it as some kind of computing utility, and anyone who believes they will does not understand history. The problem is not communication speed or computing capacity. We had computing centers decades ago with fast access and private terminals. Even when computing power greatly exceeded people’s needs, users were not satisfied with remote support. The problem was not technical; it was both personal and political. People simply wanted to control the resources they needed, and no amount of remote capacity could satisfy them. This was true then and, as computing capability becomes less and less expensive, it is inconceivable to me that people will want to use remote computers for their bread-and-butter needs. This will be particularly true when they can get supercomputer power of their own for less than it costs to buy the desk on which they will put it.

Application Service Providers

This implies that the advent of application service providers (ASPs) is an anomaly. ASPs provide computing capability, essentially as a utility. Many view ASPs as the wave of the future and have invested a great deal of money in them. While it is possible that ASPs will become big business, the prime reason that organizations subscribe to ASPs is to avoid the cost and expense of operating their own computing systems [Kerstetter 2001].

In essence, the reason that the ASP business is attractive is not because it is a fundamentally new way of doing business. It is attractive as a way to avoid the costs and headaches of current computing systems. This implies that the entire ASP industry depends on our inability to make computing systems that are easy for the public to install and use. However, depending on the continued unresponsive performance of an entire industry is highly risky. That may be a viable strategy for a niche offering, but now that the ASP and software service businesses are growing faster and generating more profit than all other parts of the computer industry combined, we can expect things to change.

There is no question that many organizations can make a great deal of money operating computing facilities for their clients. However, this business presents a tempting target for someone to produce a computing system that is so simple to install and operate that most people could do it with little or no training. Then, much of this service industry would be replaced by a new class of highly usable systems. Of course, there would still be three important parts of the software service business that would not go away:

1. the custom business of adapting computing systems to the unique needs of businesses
2. adapting the business practices of some organizations to the features and capabilities of available systems
3. incidental use of the large volumes of programs or data resources that are likely to be available in online libraries

While these three categories are all likely to be important businesses, they are not the principal reason that most organizations currently use ASPs. Most do so to avoid the cost and aggravation of installing, maintaining, and using their computer systems.

The Time Scale

Before concluding my argument for why a standalone operating-system business is not viable, I must comment on timing. I started preaching about the importance of usability many years ago. At the time, I was IBM's director of programming and the company's 4,000 systems programmers all worked for me. While I should have been able to affect what they did, and while nobody disagreed with me, I was unable to get much done. Since many others have long preached the same usability story, you might wonder why so little has been accomplished. I have concluded that there are four reasons:

1. A great deal has already been done, but the steps to date have been only a small fraction of what is needed.
2. Since true usability will require an enormous computing capability, we are just now beginning to get the technology we need.
3. The software community has had many other, more pressing problems.
4. Even when usability is a top priority, there is so much to do that it will take a long time to build the kinds of systems that are needed.

This suggests that the current ASP and software service businesses are likely to be viable for many years, but not forever.

Kinds of OS Businesses

In exploring what is likely to happen in the future, we must first consider the three main kinds of operating-systems businesses and their characteristics. These three business types are as follows:

1. First are hardware manufacturers that offer operating systems as product support. Examples of this are IBM, Apple, Sun, and others.
2. Second are standalone operating-systems businesses like Microsoft with its Windows offerings.

3. The third case is the “open-source” operating-system movement. Here, the prime examples are Linux and Unix.

The Hardware-Coupled OS Business

In the hardware-coupled operating-systems business, the objective has been to sell hardware. In projecting what will happen in the future, the automobile industry provides a useful analogy. For the first 50 years or so, automobile technology was engine-centric. That is, the design and marketing of automobiles featured the engine’s power and reliability. Leading up to and following World War II, however, this changed. While engines continued to be important, they were no longer a principal discriminator in the buying decision. In fact, today, few people could tell you the horsepower or displacement of their car’s engine. The last 50 years or so of the automobile industry have been largely dominated by comfort, style, service, and economy. We are also beginning to see safety, quality, and environmental concerns emerge as important buying discriminators.

This suggests that the hardware-coupled OS business will evolve from selling power, cost, and function to featuring usability, installability, reliability, and security. Since the hardware and software are likely to be marketed together, there will be little motivation to add capabilities that do not sell new systems. The profit motive would also limit new functions and features to those that could be financially justified. Since this is precisely the kind of business IBM had before we unbundled software, experience shows that operating-systems development will be tightly constrained and that there will be little motivation to add features purely to improve the capabilities of the existing systems. The key is what sells new products.

The Standalone OS Business

Not surprisingly, the objective of the standalone operating-systems business is to sell operating systems. Since one of the principal ways to sell them is with new hardware, we can expect the OS vendors to strive to increase the market for the hardware that uses their systems. While selling new operating systems with new hardware is an attractive business, it is largely captive to the ups and downs of the hardware business. This suggests that operating-systems vendors will add functions and features to make their systems attractive to installed hardware users. These OS vendors will then urge the customers to upgrade to the new operating systems without necessarily buying new hardware.

Because of the growing volume of application programs and because of the necessity of continuing to support an increasing number of old applications with every new OS version, the API must become progressively more stable. It might even become public. Then, possibly many years down the road, some clever and well-financed entrepreneur will produce a new OS that emulates the API of one or more of the dominant operating systems. This new operating system would presumably integrate the latest hardware and software technology to offer dramatically improved performance, usability, reliability, and security. Since the stand-alone OS suppliers would have trouble competing with software alone, they would either have to team up with hardware suppliers or lose much of their business.

The Open-Source OS Business

The third case is the open-source OS business. Here, the motives are entirely different. There is no desire to sell new hardware or software, only to provide a more usable, installable, reliable, and secure system. The great attractiveness of the open-source OS business is that it caters to the desires of a steadily growing body of installed users. These people feel that their current systems are marginally adequate and do not want to change or evolve their OS versions. They are not even terribly interested in the latest "gee-whiz" chip. They would just like installable, usable, reliable, maintainable, and secure systems that do precisely what their current systems do. While the operating-systems suppliers could largely eliminate the attractiveness of the open-system offerings by dramatically improving the user characteristics of their systems, that is not likely to happen very quickly. As a result, the open-source business will likely continue to grow. This also suggests that the open-source movement is, at least to some degree, competing with the software service and ASP suppliers.

Middleware

All of the preceding argument has ignored an important segment of the software industry: middleware. By middleware, I mean that growing family of programs that are used to administer, support, and use computing systems. This kind of software includes programs to handle administrative, operational, and support activities; provide support for application development; and furnish generic application support for system developers and users. Since, as I noted in the first column of this series, the volume of application programs will continue to grow for the foreseeable future, all of these middleware categories are also likely to continue to grow.

The challenges in the middleware business include all of the challenges of starting and running a new business in a competitive industry. They also include the challenge of resisting the threats and blandishments of the OS suppliers. Middleware businesses really are in the middle. While they must have creative and marketable ideas and the funds and know-how to start and run a business, once their ideas are financially successful, they become attractive targets. Since all three types of operating-systems businesses must continually add features to their systems to survive, the natural trend will be for the OS suppliers to incorporate the most attractive middleware features into their systems. They might either acquire the middleware companies or simply appropriate their ideas. This suggests that most middleware businesses will be transient. While they are likely to continue to be valuable sources of innovation, they will have to do four things to survive:

1. continue to have good ideas
2. be very effective marketers
3. have substantial financial support
4. protect their intellectual property

Summary

While the current situation is likely to continue essentially as it is today, at least for many years, technology will ultimately win, and we will see the standalone operating-system business merge into the larger computing-systems business. This, I am convinced, is the long-term answer. However, since the nature of the first two types of operating-systems businesses has been essentially static for more than 20 years, the long term could be very long indeed.

In my next column, I will explain what these trends in applications and systems programming mean for software engineering and what they mean for each of us.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Marsha Pomeroy-Huff, Jim McHale, Julia Mullaney, and Bill Peterson.

References

[Kerstetter 2001]
Kerstetter, Jim, “Software Shakeout,” *Business Week* (March 2001): 72-80.

15 The Future of Software Engineering: Part V

First Quarter, 2002

This is the fifth in a series of columns on the future of software engineering. The previous four columns addressed some of the likely trends in application programming and systems programming. This column covers overall trends in the industry and probable scenarios of the future, focusing on the forces at work on software-intensive businesses and how businesses are likely to change in response to those forces.

In the previous four columns, I covered application programming, systems programming, and some of the likely future trends in these areas. In this column, I focus more broadly about the overall trends in our industry and what we will likely see in the future. In particular, I address the forces at work on software-intensive businesses and how businesses are likely to change in response to these forces. I then close with some comments on how software professionals can better prepare themselves for the challenges of the future.

As I asserted in the previous columns, there are some differences in the forces on the systems, applications, and middleware businesses, but there are also some commonalities. These common forces will affect all of us, regardless of what we do. First, to segment the discussion into manageable chunks, I start by reviewing the principal kinds of software businesses. Then, I identify the common forces on our industry. Next, I explore the implications of these forces on a software-intensive business. Finally, I explore how these forces will likely change the kind of work that software people do and what this is likely to mean for each of us.

While the positions that I take and the opinions I express are likely to be controversial, my intent is to stir up debate and, I hope, to shed some light on what I believe are important issues. Also, as is true in all of these columns, the opinions that I express are entirely my own.

The Kinds of Software Businesses

As I noted in the earlier columns in this series, we can expect the volume of application-development work to continue growing. This work will require people who know and understand various application domains and are also competent programmers. For such work, skill needs will span the full gamut from traditional business and accounting applications to embedded controllers that manage complex devices and processes. On the other hand, systems developers will principally be concerned with developing, maintaining, and enhancing the operating and support systems needed by the application-development community.

The operating-systems community will be split into three rather loosely defined groups: the software houses like Microsoft and Oracle; the systems businesses like Apple, Sun, and IBM; and the growing body of open-source programmers supporting systems like Unix and Linux. While it is too early to tell exactly how this business mix will evolve, the fuzzy middleware boundary between the operating-systems and application worlds is where a large body of people are now developing and marketing software. Their objective is to fill the cracks between the operating systems and application domains. As pointed out in Part IV of this series of columns, this middleware business faces unique challenges, and it is likely to be the principal competitive battleground for the next several years. This intersection was precisely the focus of Microsoft's antitrust lawsuit, and it will continue to be both the legal and competitive focus for some time to come.

We can also expect the interface between the systems and application worlds to fluctuate in response to evolving user needs. The development community that most quickly and effectively satisfies users' needs is likely to earn a larger share of the competitive pie. While various suppliers may use contractual or other means to force users to use their products exclusively, such strategies have only been temporarily effective in the past, so they are not likely to work over the long term. Of course, the long term could be very long indeed. However, if an offering is not truly in the users' best interests, sooner or later it will lose out to the better competitor. While the fight may take a long time and it may be won either in the marketplace or in the courts, the final result is inevitable.

The Forces on Software Businesses

There are many forces on software businesses, but the most significant ones I see today are the following:

- The functional content and complexity of systems is increasing rapidly, as is the size of the software parts of these systems. As noted in the first column of this series, the size of the software used for any given function has been growing by roughly 10 times every 5 years. If software growth continues at this historical rate, this will mean an increase of 10,000 times in the next 20 years.
- Increasingly, software plays a central role in controlling and managing systems. Software is not just getting bigger, it is a crucial part of the products and services in almost all industries.
- Most computing systems will be interconnected. The Internet is merely the latest step in the long progression from stand-alone computing to pervasive computing networks.
- We will see more internal and external threats to our systems. In the past, when our principal preoccupation was getting systems to work, we assumed a friendly and law-abiding environment. Now, in the interconnected world, these systems must work, be safe, and stay secure, even when exposed to attack by criminals and terrorists.

One could write pages on each of these topics but I will just state these forces as facts and deal with their implications. I next explore each of these four forces and what businesses should do about them.

Software Size and Complexity

The principal concern with size and complexity is the scalability of the development process. To handle the massive increases in system scale, organizations must employ processes that scale up. To appreciate the scalability problem, consider transportation. Vastly different technologies are involved in traveling at 3 miles an hour, 30 miles an hour, 300 miles an hour, or possibly even 3,000 miles per hour. You can't transition from one speed range to the next by merely trying harder. You need progressively more sophisticated vehicles that use progressively more advanced technologies.

Unfortunately, we have yet to learn this lesson in software. We attempt to use the same methods and practices for 1,000 lines of code (LOC) programs as for 1,000,000 LOC systems. This is a size increase of 1,000 times, and the commonly used test-based software development strategies simply do not scale up. Organizations that don't anticipate and prepare for scalability problems will someday find that they simply cannot get systems to work safely, reliably, and securely, no matter how much their people test and fix them. When systems hit this wall, you can either test until the available time or money runs out, or you can scrap the system and do it over again correctly. Unless you are a Microsoft or an IBM, however, you probably can't afford to start over. As systems get larger, we can expect most organizations that keep following their current test-based processes to face this problem. It is only a question of time until they do.

The Central Role of Software

The second force is software's now-central role in controlling and managing business-critical systems. This is because much or even all of the functions that customers find attractive about modern products and services is embodied in their software. And it is just these attractive functions that make products unique in the competitive marketplace.

Many executives view software as a problem that they don't understand and have no idea how to manage. They try to subcontract their software work or to find some other magic solution that will relieve them of the problems of managing software. This is almost always a mistake. When management subcontracts the technologies that make their products unique, they lose the ability to manage their future. I have just published a book that explains this problem and some of my experiences with it. It might give you some ideas on what to say to management about the importance of software in your organization [Humphrey 2002].

Interconnectedness

The third major force on the software industry concerns the growing interconnectedness of systems. Much like telephone systems, the value of a computing system is increasingly determined by the number of other systems to which it can connect. In the past, companies could focus on something IBM used to call “exclusivity.” Now, however, systems that work only with hardware and software from one vendor are less and less attractive. In the old “exclusivity” days, IBM would sell, install, service, or support systems only if they were composed entirely of IBM products. As long as IBM was the dominant supplier of all important offering elements, this strategy worked quite well. But with the advent of the PC and the rapid introduction of many PC clones, IBM could no longer force its customers to use its products exclusively.

So, in the interconnected world, the keys to broad market acceptance are compatibility, interoperability, and interchangeability. Each of us is working on a small part of one enormous, worldwide, borderless computer-plex, and we are just now glimpsing its implications. While it is hard to predict what this trend will mean, it is clear that this new environment will force us out of the comfort and security of single-system thinking. We will need to think in interconnected ways and to remove any limitations that make it hard to interconnect and to interoperate our systems. We must recognize that the interconnected systems of the future will be used in ways that their developers could not imagine. It is precisely this ability of our systems to be used in new and innovative ways that will make them attractive to the users of the future.

Real-World Threats

The fourth force on the software industry is one we are just now facing. This new world is vastly different from the closed and comfortable one of the past. It is populated with many wonderful people but also with a few unpleasant, inconsiderate, and even threatening characters. With stand-alone systems, our exposure to the realities of a dangerous and unpleasant world were limited. But with the Internet, and with the growing interconnectedness of our systems, this is no longer the case.

This new environment will affect businesses in many ways. In particular, as we increasingly invoke the law to punish miscreants, those who have been damaged will also seek to recover damages from the organizations that built unsafe or insecure systems. Soon, secure and safe systems will be an economic necessity, and users will band together to seek damages from suppliers who don't provide such systems. There is currently a movement to modify contract law to protect software vendors from these problems. It is called UCITA, or the Uniform Computer Information Transactions Act. While UCITA has been enacted into law in two states (Maryland and Virginia), there is growing opposition to it and further expansion is unlikely unless it is substantially changed.

What These Trends Will Mean to All of Us

Regardless of your place in this future, there are some strategies that you should consider, both to make your organization more competitive and to make your personal employment more secure and rewarding. The first and broadest consequence of the increasingly central role of software is that most professional workers will be involved in developing, supporting, marketing, or using software. As a consequence, the trends that affect the software world will also affect most of us. The principal challenge is to have the vision and imagination to capitalize on this future world and to help make it happen in an orderly and useful way.

The most obvious force on our industry is security. We will probably always have criminals and terrorists, so we must write our programs to operate in a threatening and unfriendly world. To appreciate what this means, consider that over 90% of the Internet's software security vulnerabilities result from common types of software defects. That means that the software security problem is, at least for now, a quality problem. If quality was not important before, it soon will be. This suggests that you should examine your personal quality practices and look for and adopt a set that are demonstrably effective. Then, follow these quality practices religiously. While you should consider all of the available candidates, my personal recommendation is the Personal Software Process (PSP) [Humphrey 1995].

From a project and organizational perspective, you should also look for processes that are demonstrably scaleable. When you find a scaleable process that fits your organization's needs, start a movement to adopt that process. While you might argue that one working-level developer could not possibly get a business to make such a change, every important change is started by one person, and that person is rarely a manager or an executive. Usually, it is someone like you who is close enough to the problem to appreciate its implications. Talk to the people around you, build a support network, and then start talking to the managers. You will be surprised at what you can accomplish.

Regarding a scaleable process, my favorite is the Team Software Process (TSP) [Humphrey 2002]. However, before you pick your candidate process, look around and see what other methods are available. Also look for documented evidence of the effectiveness of these processes. Then, pick up the spear and get this method adopted by your organization. After all, in the last analysis, it is your job you are fighting for.

The Accelerating Pace of Change

To appreciate what these trends mean for each of us, remember that the world is now changing faster than it ever has before. The accelerating pace of change has been with us for so long that it seems almost trite to discuss it, but it does mean that the tools and methods we will use in the future will be vastly different from those that we use today.

In describing what this means to you and me, the best example I can think of is my personal experience. When I graduated from college in 1949, ENIAC, the first digital computer, had just recently been demonstrated. After a few years of graduate school and a brief university job, my first industrial position was designing a digital cryptographic system. Within two years, I was designing computers, and I have been working with computers ever since. Except for a class that I took in cost accounting, not one of my other college courses has been directly applicable to my subsequent work. This does not mean that my education was wasted but just that it was not enough. In this rapidly changing world, if you do not keep learning and remain open to new ideas and challenges, you will not play an important or even a very useful role in this challenging and exciting future.

Some Final Comments

The future of software engineering is quite unpredictable, but we can perceive some trends, particularly by considering the forces at work on our industry. In the last analysis, it is up to each of us to continue learning and to continue preparing ourselves for the challenges ahead. Then we will be prepared to take advantage of whatever opportunities present themselves.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Don McAndrews, Julia Mullaney, Mark Paulk, and Marsha Pomeroy-Huff.

References

[Humphrey 1995]

Watts S. Humphrey. *A Discipline for Software Engineering*, Reading, MA.: Addison Wesley Publishing, 1995. http://www.sei.cmu.edu/library/abstracts/books/0201546108.cfm

[Humphrey 2002]

Watts S. Humphrey. *Winning with Software: an Executive Strategy*, Reading, MA.: Addison Wesley Publishing, 2002. http://www.sei.cmu.edu/library/abstracts/books/0201776391.cfm

16 Surviving Failure

Second Quarter, 2002

You’re on a project and it's headed south. While everybody is trying their hardest, and you are doing your level best to help, you can feel it in your bones: the project is doomed to fail. What can you do? You have three choices.

1. Keep plugging away and hope things will improve.
2. Look for another job.
3. Try to fix the problems.

Keep Plugging Away

While continuing to plug away is essential, it will not actually improve things, and it is not very professional. Often the best way to guarantee project failure is to keep working in the same way. Inertia is a form of surrender. You are acting helpless and hoping somebody will save the day, or at least hoping that the crash will not be fatal. So, plug away by all means, but do something else as well.

Look for Another Job

Choice two is to look for another job, either in your current organization or elsewhere. This is always an option, and you should consider it if things get bad enough, but job-hopping has serious drawbacks. First, the situation in the new organization may not be much better—and it could be worse. Second, since projects often fail, you cannot continually run from failure or your resume will look like an employment catalogue. While this is not as serious a concern as it once was, it costs money to hire, orient, and train people. Unless management believes you will stay long enough to recoup their investment, they will not hire you. Third, changing jobs is disruptive and could involve a move and a new home. Once you have done that a few times, it loses its charm. Finally, in any organization, it takes time to become established and accepted. Until you are known and respected by management, you will not be considered for the best jobs. By moving, you start all over again at the bottom of the seniority list.

Fix the Problems

Assuming that you don’t want to give up, disrupt your life, or become unemployable, your best choice is to fix the problems before it is too late. Doing this, however, is tricky and it could actually damage your career if not done properly. Remember, the bearer of bad news often gets the blame. So if you are outspoken about the project’s problems, expect to be made the scapegoat. This does not mean that you shouldn’t act like a professional and try to fix the problems, just that you must do it very carefully.

Think Like a Manager

Since you must deal with management to solve most project problems, try to put yourself in their shoes. Consider the problems they face and decide what you could do that would help. In doing this, you can safely make three assumptions.

1. Management already suspects that the project is in trouble.
2. They want solutions, not problems.
3. Managers do not want competition.

Management Already Senses the Problem

Managers have lots of ways to get information, and the higher they are in the organization, the more sources they have. Managers also often develop a good intuitive sense and they can smell trouble even before anyone tells them. Once managers have worked with a few projects, the troubled ones take on a distinct character. The people begin to look worried and uneasy, the laughter and fun disappear, and status reports get vague and imprecise.

There are also various test, support, financial, and administrative groups that deal with most projects, and their people will hear of, or at least sense, the first signs of trouble. These people will almost certainly have passed on what they have learned to management and, if it is bad news, you can be sure that it will travel fast. So, management either knows about the problems already or has a strong suspicion.

Management Wants Solutions, Not Problems

Busy managers have lots of problems. In fact, a manager’s time is largely devoted to solving problems, whether generated by the projects, passed down from higher management, or imposed by the customer. If you go to your manager with another problem, expect to be greeted like the plague. However, if you show up with an offer of help instead, you will likely be received with open arms.

Managers Do Not Want Competition

You have a manager who is responsible for your assignments, evaluations, pay, and promotion. If your manager sees you as supportive, you can likely get help in fixing the project. However, if your manager suspects you of competing for his or her job or thinks that you are out to get exposure to senior management, expect to get cut off at the knees. If your manager is experienced, you will not even know that you have been skewered until much later, if ever.

So, watch the chain of command and start with your immediate manager. Don’t do anything your manager doesn’t know about and agree with. While that doesn’t mean your manager must know every step before you take it, be completely open and honest. Explain your approach, make sure you both understand the plan, and that you both agree on what you can do without prior approval. However, if the manager does not agree and you go over his or her head to a more senior manager, expect you or your manager to ultimately be fired.

If your manager agrees, he or she may let you carry the story upstairs, but most will do it themselves and you will not be involved or even get any credit. The key is to not worry about credit and visibility, but to concentrate on solving the problems. If you do that, sooner or later you will get plenty of visibility. There is a wonderful line by Dick Garwin, the designer of the first hydrogen bomb: “You can get credit for something or get it done, but not both” [Broad 2001].

A Strategy for Survival

When you are on a troubled project, the basic survival strategy is to act professionally. It has six steps.

1. Understand the source of the problem.
2. Decide how to fix it.
3. Fix what you can fix by yourself.
4. Review what you have done with your manager.
5. Decide on a strategy for the next steps.
6. Agree on what you can do to help.

Understanding the Problem

While problems come in many flavors, the most common involve unreachable goals. So stop and really think about the current situation and how it happened. Until you understand the problem, it will be very hard to fix.

Generally, the software problems I have seen are caused by either unrealistic schedules or inadequate resources. In either case, management has imposed, demanded, or agreed to a schedule that the current team is unable to meet. Under these conditions, just continuing to plug away and hoping for some kind of miracle merely postpones the day of reckoning and makes it harder to address the problems.

Schedule problems are particularly troublesome because they invariably lead to a host of other problems. When engineers strive to meet an impossible schedule, they are invariably working without a plan, or they have a plan that is unrealistic and useless in guiding the work. When you’re in trouble, panic is your worst enemy and that is exactly how teams behave without plans. Everybody rushes the requirements and design work so they can start coding and testing. Pretty soon, nobody knows where the project stands, modules are overlooked, fixes get lost, work is duplicated, and records are misplaced. The project is out of control.

Deciding How to Fix the Problem

Tiny projects can occasionally survive panics, but the larger they are, the harder they crash. The best rule to remember is: when you are in a hole, stop digging. Put down the shovel and make a plan. With few exceptions, when projects are in trouble, the most critical need is to make a plan. Involve the whole team and make as detailed and realistic a plan as you can.

While making a plan will sound plausible to anyone who believes in planning, most programmers can't see how this could help them to get all their code written and tested. Unfortunately, there is no simple answer that will satisfy everybody. However, there are several reasons that, when taken together, should convince even a skeptical software professional.

1. Every troubled software-intensive project I have seen that was in serious trouble did not have a realistic plan.
2. Every large software project that I have worked with that did not have a plan was in trouble.
3. When a team makes a plan at the beginning of the job, it can generally negotiate a realistic schedule with management and the customer.
4. Even in the middle of a crisis project, stopping to develop a plan will calm the panic, produce a clear understanding of the situation, and provide guidance on getting out of the hole.
5. With the plan in hand, the team can negotiate a recovery plan with management and the customer.
6. When the team follows its plan in doing the work, the team members will know what to do, the team will be able to track and report status, and management will know what to expect and when.

So planning is not magic, but it sure helps.

Fix What You Can Yourself

Before running to your manager with a recommendation to make a plan, look at your own work. Make a list of what you must do and then make a plan for doing it. If this plan shows that you can finish on the desired schedule, maybe you overreacted and the team can finish on time. But if not, you will have a convincing story to show your manager about your problems and how you plan to address them.

In making this plan, observe a few cautions. To the extent that you can, base the plan on historical data [Humphrey 1995a, Humphrey 1995b, Humphrey 2002]. Also, be discreet in making the plan. After all, if your manager thought planning was a good idea, the team would probably have a plan and you would not be in this mess. So get your story straight before talking about it. Next, if your friends or teammates have some planning experience, get them to review your plan and identify any holes or errors. Once you have a plan you believe in, talk with your manager.

Talking With Your Manager

In talking with your manager, concentrate on your own work and the plan you have made for doing it. If your plan shows that you can't meet the committed dates, ask for guidance. Maybe your plan includes too many tasks or the manager might see some way to simplify the work. If this leads the manager to asking the other team members to make similar plans, you are on the road to success. But if not, and if the plan helped you to have a realistic discussion with your manager, discuss what you have done with your teammates and suggest that they do the same thing. Then, when more team members have plans, you can all go to the manager and show that the problem is bigger than he or she thought.

Agree on the Next Steps

If, as is likely, the composite of all the team members' plans shows that the project is in serious trouble, suggest that the team make a complete plan. If the manager agrees, he or she will then likely go to higher management to review the project and get agreement to a replanning effort. Then, with that plan, you, your teammates, and your manager will have a sound basis for renegotiating the team's schedule. If you get this far, you will be able to turn the failed project into at least a partial success. If not, consider your other alternatives: either keep your head down and continue plugging away, or find another job.

Conclusions

As long as you continue to work quietly on a failed project, you are part of the problem. By taking a more active role and addressing your part of the job first, you can become part of the solution. This will help the project and your career, and it will help you to behave like a true professional. It will also lead to a satisfying and rewarding way to work.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Jim McHale, Julia Mullaney, Marsha Pomeroy-Huff, and Bill Peterson.

References

[Broad 2001]

Broad, William J. "Who Built the H-Bomb? Debate Revives." *The New York Times* (April 24, 2001): D1.

[Humphrey 1995a]

Humphrey, Watts S. *A Discipline for Software Engineering*. Reading, MA: Addison Wesley, 1995. http://www.sei.cmu.edu/library/abstracts/books/0201546108.cfm

[Humphrey 1995b]

Humphrey, Watts S. *Introduction to the Team Software Process*. Reading, MA: Addison Wesley, 1999. http://www.sei.cmu.edu/library/abstracts/books/020147719X.cfm

[Humphrey 2002]

Humphrey, Watts S. *Winning with Software: An Executive Strategy*. MA: Addison Wesley, 2002. http://www.sei.cmu.edu/library/abstracts/books/0201776391.cfm

17 Learning from Hardware: Planning

Third Quarter, 2002

There has been a long-term effort to apply traditional engineering methods to software. While some portray these methods as the answer to software's many problems, others argue that they are rigid, constraining, and dehumanizing. Who is right? The answer, of course, is that the appropriateness of any method depends on the problems you are addressing. While highly disciplined methods can be bureaucratic and reduce creativity, a complete lack of structure and method is equally if not more damaging.

Engineering and Craftsmanship

One way to look at this issue is as a continuation of the craft versus engineering debate that has raged for over a century. In his recent paper, Kyle Eischen describes the long-running argument about individual craftsmanship versus structured, managed, and controlled engineering methods [Eischen 2002]. Before the advent of software, craft-like methods had always had a serious productivity and volume disadvantage. There simply were not enough skilled craftsmen to meet society's demands for quality goods and services.

Factories were developed as a way to meet the need for volumes of quality goods, and manufacturing plants continue to serve these same needs today. The objective, of course, was not to destroy the crafts but to devise a means for using large numbers of less-expensive workers to produce quality products in a volume and for a cost that would satisfy society's needs. However, the widespread use of factories has required orderly processes and products that were designed to be economically manufactured in volume. This has led to many of today's engineering practices.

While these volume-oriented engineering methods have generally been effective, they have also often been implemented improperly. This has caused resentment and has reduced engineering efficiency and produced poorer quality products. However, as Deming, Juran, and many others have pointed out, this is not because of any inherent problem with engineering methods but rather with how these methods have been applied [Crosby 1979, Deming 1982, Humphrey 1989, Humphrey 2002, Juran 1988].

The Software Problem

As far as software is concerned, our current situation is both similar and different. The need again is to economically produce quality products in volume. Further, since the volume of software work is increasing rapidly, there is a need to use larger numbers of workers. While this might imply the need for factory-like methods that use lower skilled people, this cannot be the case with software. This is because software is highly creative intellectual work.

The push toward more of an engineering approach for software is not caused by a shortage of skilled people. Even though we have periodically had programmer shortages, these shortages have not been caused by a lack of potential talent. There appears to be an almost unlimited supply of talented people who could be trained for software jobs, if given suitable incentives. The push for engineering methods comes from a different source, and it is instructive to examine that source to see why we face this craft-engineering debate in the first place.

The Pressure for Improvement

The source of pressure for software process improvement is the generally poor performance of most software groups. Products have typically been late, budgets have rarely been met, and quality has been troublesome at best. From a business perspective, software appears to be unmanageable. Since software is increasingly important to most businesses, thoughtful managers know that they must do something to improve the situation.

From a management perspective, software problems are both confusing and frustrating. Businesses require predictable work. While cost and schedule problems are common with technical work, most engineering groups are much better than we are at meeting their commitments. Senior managers can’t understand why software people don’t also produce quality products on predictable schedules and with steadily declining costs. They need these things to run their businesses and they expect their software groups to be as effectively managed as their other engineering activities. This management unhappiness spans the entire spectrum from small one- or two-person software projects to large programs with dozens to hundreds of professionals.

Software Background

While the performance of software projects has been a problem for decades, software people have not historically applied traditional engineering methods. They have not typically planned and tracked their work, and their managers often either didn’t believe the software problems were critically important or they didn’t know enough about software to provide useful guidance.

This situation is now changing, and the pressure for better business results is causing the software community to apply the principles and practices that have worked so effectively for other engineering groups. Among the most important of these practices is project planning and tracking. Therefore, the pertinent questions are

Do engineering planning and tracking methods apply to software?

If they do, need they be rigid and constraining?

To answer these questions, we need to look at why planning and tracking were adopted by other engineering fields and to consider how they might be used with software.

Plan and Track the Work

For any but the simplest projects, hardware engineers quickly learn that they must have plans. The projects that don’t have plans rarely meet their schedules and, during the job, nobody can tell where the project stands or when it will finish. On their very first projects, most hardware engineers learn to make a plan before they commit to a schedule or a cost. They also learn to revise their plans every week if needed and to keep these plans in step with their current working situation. When engineering groups do this, they usually meet their commitments.

Some years ago, I was put in charge of a large software group that was in serious trouble. Their current projects had all been announced over a year earlier, and the initial delivery dates had already been missed. Nobody in the company believed any of the dates, and our customers were irate. The pressure to deliver was intense.

When I first reviewed the projects, I was appalled to find that no one had any plans or schedules. All they knew was the dates that had been committed to customers, and nobody believed them. While everyone agreed that the right way to do the job would be to follow detailed plans, they didn’t have time to make plans. They were too busy coding and testing.

I disagreed. After getting agreement from senior management, I cancelled all the committed schedules and told the software groups to make plans. I further said that I would not agree to announce or ship any product that did not have a plan. While it took several weeks to get good plans that everyone agreed with, they didn’t then miss a single date. And this from a group that had never met a schedule before.

The Key Questions

If planning is so effective for everybody else, why don’t software people plan? First, software people have never learned how to make precise plans or to work to these plans. They don’t learn planning in school, and the projects they work on have not generally been planned. They therefore don’t know how to plan and couldn’t make a sound plan if they tried. Second, nobody has ever asked them to make plans. When plans are made in the software business, the managers have typically made them and the engineers have had little or nothing to do with the planning process. The third reason that software people don’t plan is that, without any planning experience, few software people realize that planning is the best way to protect themselves from unrealistic schedules. The fourth reason is that management has been willing to accept software schedule commitments without detailed plans. When management realizes the benefits of software plans, they will start demanding plans and then, whether we like it or not, software people will have to plan their work.

The Answers

So the answer to the first question, “Do these engineering methods apply to software?” is a clear and resounding yes. The answer to the second question, “Are these engineering methods really rigid and constraining?” depends on how the methods are introduced and used. Any powerful tool or method can be misused. The guideline here is this: Does the method’s implementation assume that some higher authority knows best, or is the method implemented in a way that requires the agreement and support of those who will use it?

Any method that requires unthinking obedience will be threatening and dehumanizing to some, if not to all, of the people who use it. This is true whether the method requires you always to plan, refactor, or document, just as much as if the method requires that you never plan, refactor, or document. All methods have costs and advantages, and any approach that dictates how always to do something is rigid and constraining. The key is to learn the applicable methods for your chosen field, to understand how and when to use these methods, and then to consistently use those methods that best fit your current situation.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Noopur Davis, Julia Mullaney, Bob Musson, and Marsha Pomeroy-Huff.

References

[Crosby 1979]

Crosby, Philip B. *Quality is Free, The Art of Making Quality Certain.* New York: Mentor, New American Library, 1979.

[Deming 1982]

Deming, W. Edwards. *Out of the Crisis.* Cambridge, MA: MIT Center for Advanced Engineering Study, 1982.

[Eischen 2002]

Eischen, Kyle. "Software Development: An Outsider's View." *IEEE Computer 35*, 5 (May 2002): 36-44.

[Humphrey 1989]

Humphrey, Watts S. *Managing the Software Process.* Reading, MA: Addison-Wesley, 1989. http://www.sei.cmu.edu/library/abstracts/books/0201180952.cfm

[Humphrey 2002]

Humphrey, Watts S. *Winning with Software: An Executive Strategy*. Reading, MA: Addison-Wesley, 2002. http://www.sei.cmu.edu/library/abstracts/books/0201776391.cfm

[Juran 1988]

Juran, J. M. & Gryna, Frank M. *Juran's Quality Control Handbook, Fourth Edition.* New York: McGraw-Hill Book Company, 1988.

18 Learning from Hardware: Design and Quality

Fourth Quarter 2002

This column continues the discussion started in the third quarter column about how software people can learn from hardware engineering methods. The prior column reviewed the pros and cons of using hardware planning methods for software work and discussed how using these methods could help us meet our businesses' needs. In this column, I continue this same discussion with a focus on how engineering quality practices and design principles could be adapted to software and what we might gain from doing so. Hardware development strategies are dominated by manufacturing and service cost considerations. Even though we don't need a factory to produce volumes of software products, we do have large and growing testing and service costs and we can learn a great deal from the ways in which hardware engineers have addressed quality and design problems. This column discusses some of these hardware engineering practices and suggests ways in which their use would improve the performance of software groups.

In the September column, I discussed what we could learn from hardware engineering, particularly about planning. Hardware engineers have developed a family of planning practices that they have used with great success. The prior column reviewed the pros and cons of using these planning methods for software work and discussed how these methods could help us meet our businesses' needs.

It is no fun to be late, to have unhappy customers, and to be unable to predict when you will finish a job. By following sound engineering planning methods, software work can be more productive, more predictable, and more enjoyable for the engineers themselves. In this column, I continue this same discussion with a focus on how engineering quality practices and design principles could be adapted to software and what we might gain from doing so.

Hardware Costs

Hardware development strategies are dominated by cost considerations. The principal costs are those the factory incurs in producing the products, as well as those the service organization expends in handling product warranty and repair work. Even though we don't need a factory to produce volumes of software products, we do have large and growing service costs and we can learn a great deal from the ways in which hardware engineers have addressed quality and design problems.

The Design Release

One of my first jobs when I joined IBM some years ago was to manage the development and release-to-manufacturing of a hardware product. We had built a working model and had complete parts lists, assembly drawings, and component specifications. While I thought we had a complete story, the manufacturing and service groups put us through the ringer for two exhausting days.

It took me a while to realize why they were being so difficult. They would not accept the release until we convinced them that our design provided the information they needed to meet cost, schedule, quality, and production volume commitments. Once they accepted the design release, these manufacturing and service groups would be committed to producing, warranting, and repairing these products on a defined schedule, with specified product volumes, and for the estimated costs. Their ability to do this would determine whether or not IBM made money on the product.

Since manufacturing and service were the two largest direct cost items for IBM's hardware products, these groups had learned how to manage costs and they were not about to accept a release that had potential cost problems. While the manufacturing and service people were hard to convince, they imposed a valuable discipline on the development engineers. By making us produce complete, precise, and clear designs, they motivated us to think about manufacturing and service quality during design. This release discipline provided a solid foundation for all of the subsequent hardware quality improvement programs.

The Need for Precise and Detailed Designs

Most hardware engineers quickly learn the importance of a precise and detailed design. On their very first jobs, they learn the difference between designing a laboratory prototype and releasing a design to the factory. Manufacturing groups will not accept a design release unless it provides the information they need to define the manufacturing processes, estimate the costs of production units, predict cost as a function of production volume, calculate warranty and service costs, order and fabricate all of the parts, and assemble and test the system. Many people need the design information and it is essential that they all get precisely the same story. Design documentation is also essential to enable the inevitable design changes and to track and control these changes.

The Need for Documented Software Designs

The need for precise and documented designs in software is both similar to and different from hardware. There are five principal reasons to document a software design:

- to discipline the design work
- to facilitate design reviews
- to manage change
- to preserve and communicate the design to others
- to enable a quality and cost-effective implementation

Some people can hold very complex designs in their heads. However, regardless of how gifted you are, there is some upper limit beyond which you will no longer be able to do this. When you hit this limit, your design process will fail and the failure will not be graceful. Even very complex designs are not beyond our mental capacities when we follow sound and thoroughly documented design practices. Then we will not face the design crises that often result in complete project failures.

By documenting your designs, you also facilitate design reviews. This will both improve the quality of your designs and improve your personal productivity. The connection between quality and productivity is easy to see in hardware because a major redesign generally results in a lot of scrapped hardware. For software, however, these scrap and rework costs are equally significant, although not as visible. In the simplest terms, it is always more productive to do a design job correctly the first time than it is to do and redo the design several times.

Furthermore, a precise and documented design facilitates design changes. Anyone who has worked on even moderate-sized systems knows that, to control the inevitable changes, they must have precise records of the design before and after the change. Also, many programs will still be used long after their designers are no longer available, and many of these programs must be modified and enhanced. Without reasonably clear and complete design documentation, it is expensive to maintain or enhance almost any product. A well-documented design will add significantly to the economic life and value of the programs you produce. What is even more important, by increasing the economic value of your products, you also increase your personal value.

Separate Implementation

Another reason to thoroughly document your designs is to facilitate the growing practice of subcontracting software implementation and test. You cannot efficiently use people in lower cost countries to do this work unless you have a thorough and well-documented design. Otherwise, these off-shore groups would have to complete the designs themselves. This would waste much of the time and money the subcontract was supposed to save.

With a complete and well-documented design, the designers can move on to newer jobs while the implementing groups build and test the products. Without a complete and well-documented design, the designers will be needed throughout the implementation and test work. One way to ensure that the software designs are complete and implementable would be to require that the implementing groups review and sign off on the design before they accept a design release. While the software designers might initially object to this practice, it would impose the discipline needed to truly capitalize on the implementation and test talent potentially available in developing countries.

New and Innovative Products

One argument against producing complete and well-documented designs is that software requirements are often imprecise and rapidly changing. When this is the case, the requirements, design, and implementation work must all be evolved together. This permits the users to test early product versions and to provide feedback on their improving understanding of the requirements. If these development increments are small enough and are done quickly enough, there will be fewer requirements changes to address and development can proceed rapidly and efficiently.

These requirements problems are most severe with new product development. However, in most established software groups, new product development is the exception. Only a small percentage of development time is generally spent on building new products. Most of the development work in most software organizations is devoted to repairing and enhancing existing products. Since the original designers are rarely available for this work, a documented design is needed to allow other groups to modify and enhance the original designs.

Software Service Costs

Software service costs are largely a function of product quality. While the largest proportion of user service calls are usually for what are called no-trouble-founds (NTF), we once did a study and found that over 75% of these unreproducible NTF calls were attributable to latent product defects. NTF problems are enormously expensive. They waste the users' time and they require multiple service actions before they can be found and fixed. To minimize service costs and to reduce testing time and cost, early attention to quality is essential.

Measure and Manage Quality

Quality products are not produced by accident. While most software professionals claim to value quality, they take no specific steps to manage it. In every other technical field, professionals have learned that quality management is essential to get consistently high quality products on competitive schedules. They have also learned that quality management is impossible without quality measures and quality data. As long as software people try to improve quality without measuring and managing quality, they will make little or no progress.

The lessons from hardware quality practices are instructive. Hardware quality problems have the same categories as software. They include requirements mistakes and oversights, design problems, and implementation defects. In addition, hardware groups must also worry about raw materials defects. Because of their rigorous design and design release procedures, most hardware manufacturing organizations find that their quality problems are generally due to manufacturing problems and not to design or requirements issues. This is why manufacturing quality programs concentrate almost exclusively on raw materials quality and on the quality of the manufacturing processes. The quality of the design is managed by the product developers and verified during the design release to manufacturing.

These manufacturing quality control practices are based on two principles. First, that the quality of the product is determined by the quality of the process that produced it. Second, that the quality of the process can be managed by measuring the work. The manufacturing engineers then use these measures to control and improve the manufacturing processes.

Quality and Fix Time

One way to think about quality is to consider how the process would change as a function of defect fix times. For example, programmers generally think that it takes only a few minutes to fix defects in test. They base this on their experience with most of the defects they find in unit testing. In system test, however, the time to find and fix defects typically extends to many hours or even days. While most of these defects are fixed rather quickly, some take much longer. The average time to find and fix each defect is generally 10 to 20 or more hours.

Suppose, however, that the fix times in test were much longer, how would that affect the software process? The lessons from the hardware community are instructive. Some years ago, my laboratory had a small semiconductor facility for making special-purpose chips. The turn-around time for producing a custom chip from a completed design was six months. As a result, correcting any design or fabrication errors required at least six months. Since our products were for a highly competitive marketplace, the number of chip-fabrication turnarounds was critical. The engineering objective was to release products with only one turn-around. With a little practice, their designs were of such high quality that they were generally able to meet that goal.

In the software business, the time to fix defects in final test is increasing and in some cases it can run into months. For example, for imbedded products like television sets and appliances, the general practice is to use more expensive technologies for the initial models so that they can quickly make any software corrections. Then, when the change rate drops sufficiently, they switch to a cheaper technology. As technology continues to get more complex and as competitive forces continue to increase, we will soon have to produce defect-free software before system test. At least we will for high-volume imbedded products. While testing will always be required, the software quality objective should be a one-cycle system test.

Engineered Software

While the quality management methods for the software process are necessarily very different from those used in hardware manufacture, the same principles apply. In summary, these principles are: product quality is determined by process quality; produce and document clear, complete, and precise designs; and measure and manage quality from the beginning of the job. By following these principles, many software groups are now delivering defect-free products more predictably and faster than they ever delivered products before [Humphrey 2002].

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Dan Burton, Julia Mullaney, and Bill Peterson.

Reference

[Humphrey 2002]

Humphrey, Watts S. *Winning with Software: An Executive Strategy*. Reading, MA: Addison-Wesley, 2002. http://www.sei.cmu.edu/library/abstracts/books/0201776391.cfm

19 Some Programming Principles: Requirements

First Quarter 2003

In this and the next several columns, I discuss some principles for programming work. These are principles that, when followed, will consistently produce superior software that meets the needs of customers and businesses. This column concentrates on the principles that are inherent in software work because of the nature of software products and their requirements. These principles also concern the characteristics of the people who use these products. These programming principles are as follows:

- When we program, we transform a poorly understood problem into a precise set of instructions that can be executed by a computer.
- When we think we understand a program's requirements, we are invariably wrong.
- When we do not completely understand a problem, we must research it until we know that we understand it.
- Only when we truly understand a problem can we develop a superior product to address that problem.
- What the users think they want will change as soon as they see what we develop.

The Programming Job

As any experienced programmer knows, it is hard for people to be absolutely and completely precise about anything. However, to produce a usable program, we must specify exactly what the program is to do under every possible circumstance. The difficulty of being precise is brought home to me whenever someone gives me directions.

"Go three blocks, turn left at the gas station, then take the third street on the right." "But," I interrupt, "precisely where do I start and in what direction should I face?"

For some reason, the questions that I must ask to get precise directions always seem to annoy people. Why should this annoy them? Don't they know that, without precise and complete information, I might get lost and waste a great deal of time?

The answer is, no they do not. In fact, even when they know exactly what they want done, most people are unable to tell you precisely what to do. What is worse, they will even get annoyed when you press them for the required details. This attitude complicates the programmer's job. To work properly, computing systems must be given absolutely precise instructions. Therefore, to write any program, the programmer must reduce that problem to a precise form before it can be executed by a computer. This means that we must somehow persuade one or more knowledgeable users to explain all of the problem's details. In summary, there are four reasons why this is hard.

- First, the users will not know how to be precise about their needs.
- Second, they will get annoyed when we press them to be precise.
- Third, we will often think that we understand the problem before we really do.
- Fourth, the users will often misunderstand or not even know what they need.

Programming is and always will be a precise intellectual activity that must be performed by people. While machines can help us in this work, they can never figure out what we need, so they can never replace us. The problem is that people are error prone and programs are extraordinarily sensitive to errors. The programming challenge we face is to devise processes and methods that will help us to produce precise intellectual products that are essentially defect-free.

Understanding the Problem

The difference between thinking that you understand a problem and truly understanding it is like night and day. It is amazing how often I think that I know something but then find that I really do not. My mother used to explain that being positive was “being wrong at the top of your voice.” When it matters, like when writing a program, preparing a talk, or drafting a paper for publication, I try to prove that what I think is true really is true. When I look at data, consult a reference, or talk to an expert, I often find that my initial opinion was either wrong or too simplistic. The real story is invariably much more interesting and often more complex than I had initially realized.

It is easy to settle for half-baked answers. Some years ago, a programming manager told me about a conversation that she overheard between two of her programmers. They were developing a new version of IBM’s COBOL compiler and were debating a particular feature. One of them felt that the users would prefer one approach and the other felt that a different format would be more convenient. This manager told me that, even though neither of them really knew which approach would be best, they ended up agreeing on some middle ground that they both thought would be OK.

On any large product, there are hundreds to thousands of these minor decisions and, when programmers don’t really know the proper or most convenient or most efficient answer, they usually guess. Even if the odds of being right are 50-50, they will make hundreds to thousands of incorrect decisions. Then, the odds of their producing a convenient and highly-usable product will be essentially zero.

Researching Problems

There are lots of ways to research problems. For example, I often write about how people develop programs and the costs and benefits of various programming practices. Ever since I started my research work on personal and team programming methods over 13 years ago, I have gathered data about my own and other people’s work. I now have a database of over 11,000 programs that I regularly use to verify some opinion or to answer some question. Depending on the question or problem, data can be very helpful.

The second approach is to ask someone who knows the answer. The problem here is that, like the COBOL programmers, you may not have an expert on hand and will often settle for just getting a second opinion. While it might make you feel better to have someone agree with your guess, it is still a guess. You can be pretty sure that it will either be wrong or a simplistic approximation of the true situation.

Often, we cannot find a convenient expert and the time required to get expert help seems likely to delay the project. However, this is only an excuse. Once you build all of your guesses into a program's design, the costs of fixing it later will be vastly greater than any conceivable delay. So bite the bullet—take the time to get the facts needed to do a truly superior job. It will invariably pay off.

Getting Informed Input

Some years ago, one of the projects in my organization was developing a very large programming product. I was convinced that usability was a key need and kept pushing the project managers to have a panel of experts review their designs. The managers kept telling me that it was too early to hold such a review. They argued that they had not yet completed enough of the design work. Then, one day, they told me that it was too late. It would now be too expensive to make any changes. I insisted, however, and we held a two-day review.

I sat in on the meetings with a panel of a dozen user experts. The programming managers could not answer several of the more detailed operational questions and had to call their programmers for help. At the end of two days, these managers had agreed to make half a dozen changes before they shipped the first release. They now agreed that, without these changes, the product would have been unusable. After I moved on to a different position, this management team never held another expert review. Since these were very smart people, I concluded that they must not have appreciated the enormous importance of getting informed user input.

There are many ways to research requirements questions. Sometimes you will need experts but often a simple prototype and one trial use will tell you the answer. You can also get associates or even non-programmer support people to try to use a prototype. By observing their reactions, you can resolve many usability issues. In other cases, you can accumulate several questions and build a prototype to try out the most promising alternatives. Then you can often get some experts to test the prototypes and to give you their views. In any event, keep track of your assumptions and guesses, and verify them as soon as you can.

Building Superior Products

During my time as IBM's Director of Programming, I received a constant stream of requests to incorporate one or another field-developed program into the IBM product line. These programs were developed by IBM's customer support personnel to help their accounts solve critical problems. We did accept a few of these programs and, in a few years, these few field-developed programs were consistently at the top of our customer satisfaction ratings. Their ratings were well above those for the standard products that had been developed in our laboratories.

In contrast to the company's "standard" products, these field-developed programs had been designed by people who were intimately familiar with the users' environment. These developers intuitively understood precisely how the product should work and didn't have to guess what would be more convenient. They knew. If you want to develop a superior product, either become intimately familiar with the user's environment or have someone with that familiarity readily available to answer your questions. You will need to consult them almost every day.

Changing Problems and Products

If expert advice were all that was needed, requirements would not be such a serious problem. However, there are three issues that complicate the story. First, the users will not be able to tell you what they want. Generally, these are not development people and they cannot visualize solutions to their problems. They could tell you how they work right now but they will have no idea how they would work with a new product, even if its designs were based on exactly how they work today. The second problem is that few users understand the essence of the jobs they are doing. By that, I mean that they do not understand the fundamentals of their jobs, so they cannot help you to devise a product solution that addresses the real operational needs. All they can usually do is help you define how to mechanize the often manual system that they are currently using. The third problem is a kind of uncertainty principle. By introducing a new product, you actually change that user's environment. Often, this will completely change the way the job should be done and it could completely obsolete the original requirements.

What these three problems tell us is that your initial definition of the requirements will almost certainly be wrong. While you must get as close as possible to a usable solution, you must recognize that the requirements are a moving target that you must approach incrementally. To produce a truly superior and highly-usable product for a realistic cost and on a reasonable schedule, you must follow a development strategy and process that assumes that the requirements will change. This requires that you encourage early changes and that you develop and test the program in the smallest practical increments. Also, if possible, get the users to participate in the early testing. Then, when you find what you did wrong or what your customers could not explain or what the users did not know, the amount of rework will be small enough to do in a reasonable time. Then you can quickly take another user checkpoint to make sure you are still on the right track.

These few principles are fundamental to just about every programming job. What is most interesting is that experienced programmers will generally recognize these principles as correct, even while they work in an environment that does not apply them. The challenge that we programmers face is to convince ourselves, our managers, our executives, and even our customers to devise processes, to plan projects, and to manage our work in a way that is consistent with these principles. While it might seem simple and easy to convince people to do something so obvious, it is not. There are many reasons for this, but the most basic reason is that welcoming early requirements changes exposes our projects to unlimited job growth, which has historically led to serious schedule overruns. These are problems that I will address in the next few columns where I discuss the principles for designing products, guiding projects, leading teams, and training and coaching the people who do programming work.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Dan Burton, Sholom Cohen, Luke Dzmura, Jim Over, Bill Peterson, and Marsha Pomeroy-Huff.

20 Some Programming Principles: Products

Second Quarter 2003

This is the second in a series of columns on programming principles. The first column discussed those principles that relate to program requirements and why requirements must change throughout the development process. The conclusion was that we must use development processes that recognize the inevitability of requirements changes and that will work effectively even in the face of frequent and often substantial changes. In this column, I review those product-related principles that govern our work in developing software-intensive systems and the unique challenges we face in doing software work.

The Nature of Computer Programs

Computer programs are fundamentally different from other types of products. Software doesn't wear out, rot, or deteriorate. Once a program works and as long as its operating conditions remain the same, it will continue to work indefinitely. Software products can be reproduced for essentially nothing and distributed worldwide in seconds. From a business perspective, software is almost an ideal product. It is the most economical way to implement most logic and it is the only way to implement complex logic. As an intellectual product, software can be legally protected almost indefinitely and this makes software products potentially very profitable. These basic software characteristics provide great opportunities. However, they also present us with six major challenges. To properly address these challenges, we must substantially change the way we do our work and the methods we use to run our projects.

The First Challenge: Software Work Is Intellectual

Because software work is intellectual and our products are intangible, we cannot easily describe what we do or demonstrate our results. This makes it extremely difficult for non-software people to manage software groups or even to understand what we do. As more people get involved with and become knowledgeable about software, this will be less of a problem. However, today, few customers, managers, or executives have an intuitive feeling for the software business. The problem is that you can't touch, see, or feel software. When we tell them that the product is nearly ready to ship, they have no real appreciation for what we are saying. We can't take them out into the laboratory and show them the first software model being assembled, describe which parts must be added before final test, or show them parts being welded, machined, or painted.

The problem here is trust. Businesses run on trust, but many managers follow the maxim: "Trust but verify." Unfortunately, with software, management can only verify what we say by asking another software person who they hope is more trustworthy. Since we typically don't have any way to prove that what we say is true, management can only tell if we are telling an accurate story by relating what we currently tell them with what we have said before and how that turned out. Unfortunately, the abysmal history of most software operations is such that very few if any software developers or managers have any credibility with their more senior management.

The result is a critical challenge for the software community: we work in an environment where senior management doesn't really believe what we tell them. This means that, almost invariably, management will push for very aggressive development schedules in the hopes that we will deliver sooner than we otherwise would. To maintain this schedule pressure, management is not very receptive to pleas for more time. Furthermore, even when we break our humps and actually deliver a product on the requested schedule, management isn't very impressed. After all, that is only what we said we would do. Other groups do that all the time.

The Second Challenge: Software Is Not Manufactured

Because software can be reproduced automatically, no manufacturing process is required. This means that there is no need for a manufacturing release process and therefore that there is no external process to discipline our design work. When I used to run systems development projects, the hardware had to be manufactured in IBM's plants. Before I could get a cost estimate signed off or get a price and product forecast, I needed agreement from the manufacturing and service groups. This required that the product's design be released to manufacturing. In this release process, the manufacturing engineers reviewed the design in great detail and decided whether or not they could manufacture the product for the planned costs, in the volumes required, and on the agreed schedules. We also had to get service to agree that the spare parts plan was adequate, that the replacement rate was realistic, and that the service labor costs were appropriate.

As you might imagine, release-to-manufacturing meetings were extremely detailed and often took several days. The development engineers had to prove that their designs were complete enough to be manufactured and that the cost estimates were realistic and accurate. While these release meetings were grueling and not something that the development engineers enjoyed, once you passed one, you had a design that the manufacturing people could build and everybody knew precisely where your program stood.

Unfortunately, software development does not require such a release process. The consequence is that design completion is essentially arbitrary and we have no consistent or generally-accepted criteria that defines what a complete design must contain. The general result is poor software-design practices, incomplete designs, and poor-quality products. The challenge here is that, without any external forces that require us to use disciplined design practices, we must discipline ourselves. This is difficult in *any* field, but especially for software, since we have not yet learned how to reliably and consistently do disciplined software work.

The Third Challenge: The Major Software Activities Are Creative

Because software can be reproduced automatically, a traditional manufacturing process is not required and no manufacturing resources are needed. This is a major change from more traditional products where the manufacturing costs are often more than ten times the development expenses. In software, the principal resources are development and test. This mix imposes a new set of demands on management: they must now learn to manage large-scale intellectual work.

In the past, large-scale activities have generally concerned military operations or manufacturing processes. Typically, large numbers of people have been needed only for repetitive or routine activities like reproducing already-designed products. In software, large-scale efforts are often required to develop many of the products. In directing large numbers of people, management has typically resorted to autocratic methods like unilaterally establishing goals, setting and controlling the work processes, and managing with simplistic measures. The problem here is that large-scale intellectual work is quite different from any other kind of large-scale activity. Autocratic practices do not produce quality intellectual work and they are counterproductive for software. In fact, such practices often antagonize the very people whose creative energies are most needed.

To address this challenge, management must understand the problem and they must also get guidance on what to do and how to do it. While the proper management techniques are not obvious, they are not very complex or difficult. And once they are mastered, these management techniques can be enormously effective.[9]

The Fourth Challenge: Software Lives Forever

Because software essentially lives forever, product managers face an entirely new and unique set of strategic issues. For example, how can they continue to make money from essentially stable products, and what can they do to sustain a business in the face of rampant piracy? The problem is that immortal products that can be reproduced for essentially nothing will soon lose their unique nature and cease to be protectable assets. While this is not a severe problem when the software is frequently enhanced, once products stabilize, they will be exceedingly hard and often impossible to protect, at least for anything but very short periods.

This may not seem like a serious problem today, but it soon will be. A large but ultimately limited number of basic functions will be required to provide future users with a stable, convenient, accessible, reliable, and secure computing capability. While such functional needs have evolved rapidly over the last 50 years, the rate of change will inevitably slow. This fact, coupled with the users' growing needs for safety, security, reliability, and stability, will require that the rate of change for many of our products be sharply reduced. This in turn will make it vastly more difficult to protect these products.

The reason that this is important to programmers is that if the uniqueness of our products cannot be protected, our organizations will be unable to make money from the products we produce. They will then no longer be able to pay us to develop these products.

Lest this prediction sound too dire, the programming business will not wither away. I am only suggesting that the part of the programming business that provides basic system facilities will almost certainly have to change. On the other hand, I cannot visualize a time when application programming will not be a critical and valuable part of the world economy. In fact, I believe that skilled application programming will become so important that the programming profession as we now know it will no longer exist: every professional will have to be a skilled application programmer.

[9] See my book, *Winning with Software: an Executive Strategy*. Reading, Mass., Addison Wesley, 2002.

The Fifth Challenge: Software Provides Product Uniqueness

Because software contains the principal logic for most products, it provides the essential uniqueness for those products. This means that the software's design is an essential product asset and that the key to maintaining a competitive product line is maintaining a skilled and capable software staff. This is an entirely new consideration for a management group that has viewed software as an expense to be limited, controlled, and even outsourced, rather than as an asset to be nurtured, protected, and grown. The pressure to limit software expenses is what caused IBM to lose control of the PC business. Management was unwilling to devote the modest resources needed to develop the initial PC software systems. This gave Bill Gates and Microsoft the opportunity to replace IBM as the leader of the software industry.

As software becomes a more important part of many products, whole industries are likely to lose control of their products' uniqueness. This control will be in the hands of the programmers in India or China or whoever else offered the lowest-cost bids for outsourcing the needed software work. In effect, these industries are paying their contractors to become experts on their products' most unique features. Ultimately, these contractors will very likely become their most dangerous competitors. Over time, these industries may well find themselves in a position much like IBM's in the PC business: manufacturing low-profit commodity-like hardware to run somebody else's high-margin software.

The Sixth Challenge: Software Quality is Critical

Because software controls an increasing number of the products we use in our daily lives, quality, safety, security, and privacy are becoming largely software issues. These increasing quality needs will put enormous pressure on software businesses and on software professionals. The reason is that software safety, security, and privacy are principally software design issues. Without a complete, fully-documented, and competently reviewed design, there is practically no way to ensure that software is safe, secure, or private. This is a problem of development discipline: since we don't have to release our products to a manufacturing or implementation group, there is no objective way to tell whether or not we have produced a complete and high-quality design.

This development discipline problem has several severe consequences. First, it has never been necessary for software people to define what a complete software design must contain. This means that most software engineers stop doing design work when they believe that they know enough to start writing code. However, unless they have learned how to produce complete and precise designs, most software engineers have only a vague idea of what a design should contain.

With the poor state of software design and the growing likelihood of serious incidents that are caused by poor-quality, insecure, or unsafe software, we can expect increased numbers of life-critical or business-critical catastrophes. It won't take many of these catastrophes to cause a public outcry and a political demand for professional software engineering standards. This will almost certainly lead to the mandatory certification of qualified software engineers. Then, the challenge for us will be to determine what is required to be a qualified software engineer and how such qualification can be measured.

Conclusions

Since we have been living with all of these problems for many years, you might ask why we should worry about them now. The reason is that the size and scope of the software business has been growing while software engineering practices have not kept pace. As the scale and criticality of software work expands, the pressures on all of us will increase. Until we learn to consistently produce safe, secure, and high-quality software on predictable schedules, we will not be viewed as responsible professionals. As the world increasingly depends on our work, we must either learn how to discipline our own practices or expect others to impose that discipline on us. Unfortunately, in the absence of agreed and demonstrably effective standards for sound software engineering practices, government-imposed disciplines will not likely be very helpful and they could even make our job much more difficult.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Dan Burton, Don Firesmith, Marsha Pomeroy-Huff, Julia Mullaney, Bill Peterson, and Alan Willett.

21 Some Programming Principles: Projects

Third Quarter 2003

This is the third in a series of columns on programming principles. The first column in March discussed some general principles of programming, with particular emphasis on the changing and ill-defined nature of software requirements. The second column in June addressed those software principles that relate to the nature of our products. These principles concern the fact that our products are intangible, can last essentially forever, and are increasingly important to our businesses and to society in general.

In this column, I discuss the principles that relate to software engineering projects. Many of these principles are common to engineering projects of all kinds, but software projects present some issues that make our work uniquely challenging and rewarding. In discussing project principles, it is important to start with the fundamental purpose or objective of most software projects. This is to develop or enhance a product to meet a business need. In fact, this defines the following important principle about almost any software project.

The principal objective of almost all software projects is to meet a business need.

This means that the schedule, cost, and quality of the work is of paramount importance. Therefore, in addressing the principles that govern software projects, I will talk about schedule, cost, and quality. Of course, by quality, I refer to the ability of the product to reliably produce the user's desired results. While there are many other important project considerations, they all relate directly or indirectly to schedule, cost, and quality performance. In closing, I will comment on the benefits of successful projects and the characteristics of project success from both a business and an engineering perspective.

Project Schedule Performance

Project schedule performance has three interesting characteristics. First, with few exceptions, if you don't meet the committed schedule or a revised schedule that everyone knows about and has previously agreed to accept, your project will not be judged fully successful. In other words, schedule performance comes first. For example, in assessing the best projects, any that are late, even by only a few days or weeks, never make it to the top of the list.

Second, and particularly for projects that last for more than a few weeks, the important stakeholders need to know where you stand and if you are likely to finish on time. The end users need to make installation and conversion plans, the testers need to schedule their resources and facilities, and management needs to know if there will be any business problems or if they will have to step in to accelerate or redirect the work.

Keeping managers and customers properly informed requires accurate and timely status reports. This is the most common area where software projects run into difficulty. Since software engineers rarely know precisely where they stand against their schedules, they cannot make a convincing report on their status or accurately forecast when they will finish. From a management perspective, this proves that they do not know how to manage their work. This leads to distrust, management meddling, and often even to project redirection or cancellation. In fact, I have seen management interference destroy projects that otherwise would have been reasonably successful, all because of poor status-tracking and reporting.

Third, schedule performance by itself is not personally rewarding. While it is essential to meet other people's criteria for project success, the satisfaction that comes from meeting a schedule is ephemeral. After a rather brief period, management's reaction will become "So what was the big deal?" You just did what you said you would do. Consider schedule performance like a down payment: it is essential to get into the game and it will improve your personal reputation with management, but, by itself, it will not produce lasting rewards or personal satisfaction.

Project Cost Performance

Most engineers feel that if they meet the schedule, any cost overruns will be small and not worth worrying about. But that is becoming less and less true. As many software projects last longer and become more expensive, both cost and schedule management are increasingly important. To see why, suppose that you were building a new house and that the builder assured you that he would finish on schedule. Then, just before the final closing, he told you that your changes had cost more than expected and that he had to pay some overtime to finish on the promised date. The bill is therefore $50,000 more than you had previously agreed. This would likely cause a serious problem. The mortgage commitment probably would not cover the added costs and you probably don't have that kind of money lying around. While meeting the schedule was nice, the cost overrun could easily be a deal breaker.

Cost is equally important for software work. However, the time to address cost problems is when you first detect them, not at the end when no one has any flexibility. If your customer wants a change, if you have technical problems, or if your original estimates were way off, you should figure out what the job is now likely to cost and get agreement before you proceed with the work. While this will involve lots of nitty debates during the project, it will avoid the big cost surprises at the end. When you are at it, also make sure that you put all of these cost negotiations and agreements in writing.

Of course, the problem that cost management presents for most software projects is that we rarely know enough about the costs of our work to estimate the impact of small changes. This again is a fairly unique problem for software, but it is a problem we must learn to address if we are ever going to effectively manage the costs of our work.

Project Quality Performance

While cost and schedule performance are important, our product must work. Also, cost and schedule performance are not, in the long run, satisfying from an engineering point of view. We like to build great products. If you finished development on schedule and within planned costs but the product was a disaster, the brief glory you got from delivering on time would quickly fade. While being known for meeting schedules is important, being known as the developer of a poor-quality product is an engineering kiss of death.

The world is changing and the importance of delivering quality products will only increase. After they have lived through a few software disasters, many software developers properly conclude that it is better to deliver good products late than to produce poor products on time. This strategy, however, will continue to have business problems. The engineers who get ahead in the future are almost certain to be those who deliver quality products on time and for their committed costs. If no one in your organization can consistently do this, there are lots of organizations all over the world that would love to take your place. Many of these organizations are already demonstrating the ability to do superior work on schedule and they are growing very quickly. So, if you want to keep your job, it would be a good idea to learn how to meet both the business *and* technical needs for your products.

Project Success Criteria

The real satisfaction we get from our work is the thrill of working on a great team, creatively using the latest technology, and producing superior products that truly meet our users' needs. As engineers, we are creators and we want our creations to be accepted, used, and appreciated. This requires quality products. In short, the truly successful and rewarding projects of the future will be those that produce quality products on schedule and for their committed costs. These are the only projects that will consistently satisfy the engineers, their managers, and the users.

The criteria that engineers and managers use for judging project success historically have been very different. As I will discuss in the next column, it is important that we learn to consistently meet both management's success criteria as well as our own. The key point to remember, however, is that management decides who to hire and how much to pay for our work. This means that we must meet management's criteria if we want to get ahead. In the next column, I will conclude this series on programming principles with a discussion of the perceptions that engineers and managers have about project success and what this means for each of us.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Dan Burton, Don Firesmith, Marsha Pomeroy-Huff, Julia Mullaney, Bill Peterson, and Alan Willett.

22 Some Programming Principles: People

Fourth Quarter 2003

This is the fourth and last in a series of columns about programming principles. The prior columns discussed the principles that relate to programming requirements, software products, and the projects for developing these products. This column deals with people and the human aspects of the software process. While this is an enormous subject and no brief column could possibly do justice to the vast body of relevant knowledge, a few key principles are particularly important in determining the performance of software people and the teams on which they work.

While most people behave in reasonably predictable ways, software people are unique, both in their creative abilities and in the nature of the work they do. Software professionals are among the brightest people on earth. Most of us got into this field because we were excited by the thrill of working with a challenging and very special technology. However, the problem many of us face is that the environment in which we work rarely supports and motivates consistently high-quality creative work.

In addressing this subject, I discuss the factors that govern the performance of software professionals, the most effective ways to obtain superior performance, and the key issues to consider in motivating and guiding teams of creative professionals.

The Performance of Software Professionals

Much as in other professions, the performance of software people is governed by four things.

- their understanding of the job they have to do
- their knowledge of and skill at using the best known methods for that job
- their discipline to properly and consistently use their knowledge and skill
- the quality of the support system that motivates and controls their activities

These four governing factors form the basis of the four people principles discussed in this column.

People Principle Number 1

If the programmers do not understand the job they are to do, they will not do it very well.

This seems to be such an obvious point that it is hardly worth discussing. However, it is critically important and often overlooked. If the members of a development team are not intimately familiar with the job their product is to perform and the way the users of that product will use it, the project will almost certainly be troubled and the product will likely be a failure. If you can’t put users or people with user experience on your development team, at least ensure that the team has ready access to people with such knowledge. To produce quality products, a close and cooperative relationship with such people is absolutely essential.

People Principle Number 2

The people who know and use the best methods will do to best work.

While software developers usually get extensive training on tools and methods, they generally get little or no guidance on their personal practices. This is not true of any other sophisticated technical field. Chemical engineers are not born knowing how to conduct experiments, analyze the composition of materials, or follow sound laboratory practices. Doctors learn their profession through extensive training, by completing several years of internships and specialty studies, and by mastering the methods that their predecessors have found most effective.

In software, we have yet to learn the truisms that some methods and practices are more efficient and cost-effective than others and that the cost and quality of the products we produce is governed by the practices we use in developing them. Today, on many projects, the developers do very similar work but their personal practices are very different. I have studied such teams and found that even developers who do similar work use different methods and, what is worse, they are generally unaware of the methods their peers are using. Except for occasional help with problems or complex tools, most software people work largely alone and are unaware of how others work or the best ways to do each of their tasks.

If every scientist had to personally discover Bernoulli's principle, develop Maxwell's equations, or invent calculus, we would still be in the dark ages. The explosive growth of science and engineering didn't start until people defined their practices, measured their work, and communicated precisely. This allowed others to repeatably produce the same results. The essence of science and engineering is learning from the experiences of others. Until we build a body of professional software practices and teach new professionals to consistently and properly use these practices, programming will remain an unsatisfactory craft that produces defective, insecure, and even unsafe products.

People Principle Number 3

When programmers know how to select and consistently use the best methods, they can do extraordinary work.

We now have data on several thousand programmers who have taken the Personal Software Process (PSP) course as well as data on over 100 Team Software Process (TSP) teams that have been launched[10]. It is now clear that developers can learn and use highly-effective personal practices and that, when they use these practices, they produce much better work than they ever did before. In a recent study of 20 TSP teams that provided data on delivered product defect levels, these products had 100 times fewer defects than average industrial products and 16 times fewer defects than typical products produced by CMM[11] level 5 organizations [Davis 2003]. In most cases, the-

10 TSP team projects start with a launch where the members learn the project requirements, define their own processes, and develop and negotiate their plans with management.

11 CMM is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

se were first-time TSP teams. Since personal and team performance typically improves with experience, we can expect even better performance in the future.

The obvious problem with requiring that developers use the best practices is in determining what the best practices and methods are. However, the reason that the PSP and TSP are so effective is that they provide developers the tools and methods they need to make this decision for themselves. With the PSP, professionals learn how to follow a defined process, how to modify that process when they need to, and how to measure and plan their personal work. By using their own data, developers can see what methods work best for them and they can make informed decisions about how to do their work.

Similarly, when TSP teams are launched, they examine the job that they have to do and consciously decide on the best strategy, process, and plan for the work. While this may not seem to be the best possible way to do the job, it is the one that the team members think would be best, and these are the only people who know their personal skills, abilities, and interests; the work that must be done; and how they can best work together as a team. So, while there may be theoretically better ways to do the work, the team's informed decision on its own strategy, process, and plan will actually produce the best way for this team to do this job.

People Principle Number 4

Superior software work is done by highly motivated developers.

When people are discouraged, antagonized, or even just unhappy, they cannot do their best work. The key to getting superior work from creative people is to energize the entire team and to motivate all of the members to do their very best. But what motivates software people and how can one build and sustain this motivation? In an interesting study of software projects, Linberg compared management's typical views of project success with those of the team members [Linberg 1999]. While managers typically think in terms of cost, schedule, and product success, the developers viewed their projects quite differently.

For example, Linberg asked one group of experienced developers what project they viewed as the most successful one on which they had worked. They had just completed what he referred to as project A and over half of them cited this as the most successful of their careers. This was in spite of the fact that they had all worked on 8 or more projects and that this job was delivered in twice the desired time and for three times the planned cost. The four factors that the team members listed as making this project successful were as follows.

- a personal sense of being involved and making a contribution
- frequent celebrations where the team and management complemented them on their achievements and milestones
- positive feedback from marketing and senior management
- the autonomy to do the job the way that they thought was best

These are the things that motivate software developers. While these same factors motivate people in almost all walks of life, they are particularly important for getting superior software work. In many fields, people's personal practices are visible and relatively easy to measure and monitor. In software, much of the work is intellectual and not measurable or manageable without the developer's cooperation. This is why motivation, personal discipline, and sound professional behavior is critically important for software development. If software people do not want to work in a particular prescribed way, they won't and, unless the software people themselves tell them, it is unlikely that anyone will know. This is why contributing, being involved, being rewarded, getting positive feedback, and having autonomy are particularly important for software developers.

Whether you manage software development or do development work yourself, remember and follow these four principles. To produce superior products, the developers must

- understand the job the product is to do
- know the best methods for doing the work
- consistently select and use these best methods
- be highly motivated to work on this team doing this job

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Dan Burton, Anita Carleton, Julia Mullaney, Bill Peterson, and Marsha Pomeroy-Huff.

Thanks for your attention and please stay tuned in.

Watts S. Humphrey
watts@sei.cmu.edu

References

[Davis 2003]

Davis, N. & Mullaney, J. *The Team Software Process (TSP) in Practice: A Summary of Recent Results* (CMU/SEI-2003-TR-014). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/library/abstracts/reports/03tr014.cfm

[Linberg 1999]

Linberg, K. "Software Developer Perceptions about Software Project Failure: A Case Study." *Journal of Systems and Software 49,* 2 (December 1999): 177-192.

23 Defective Software Works

2004 | Number 1

Over the years, many people have written to me with questions about software quality, testing, and process improvement. Jon Hirota asked how to get organizations to invest in software quality; John Fox asked if I see a movement away from system test and toward quality processes; Bob Schaefer wondered what I thought would happen in the area of software integration and testing; Dan St. Andre asked what software development managers can do to encourage executive management to meaningfully address software quality; and Pete Lakey wondered if and how the software community should use statistical process control techniques. I won't directly answer all of these questions but I will discuss these quality and testing issues here and in my next column. While it has taken me an embarrassingly long time to respond to these letters, they still raise a critical question: "How important is software quality and how should quality practices change in the future?"

First, What Do We Mean by Quality?

While the classical definition of product quality must focus on the customer's needs, in this and the next column, I will concentrate on only the defect aspect of quality. This is because the cost and time spent in removing software defects currently consumes such a large proportion of our efforts that it overwhelms everything else, often even reducing our ability to meet functional needs. To make meaningful improvements in security, usability, maintainability, productivity, predictability, quality, and almost any other "ility," we must reduce the defect problem to manageable proportions. Only then can we devote sufficient resources to other aspects of quality.

The functional and operational characteristics of a software product should and will continue to be important, but that is where most people now focus, and there is little risk that they won't continue to do so in the future. If a product doesn't have attractive functional content, it won't sell, regardless of how few or how many defects it contains. Unfortunately, many software groups treat the importance of functional quality as an excuse to concentrate almost exclusively on product function and devote little attention to better managing the defect problem.

While there is irrefutable evidence that the current "fix-it-later" approach to defect management is costly, time consuming, and ineffective, don't expect this to change soon. It is too deeply ingrained in our culture to be rooted out easily. However, since I'm an optimist, I'll keep trying to change the way the world views software quality. And by that, I mean the way we manage defects.

Second, How Important is Software Quality?

The key question is: "Important to whom?" Developers are necessarily preoccupied with defects. They spend the bulk of their time trying to get their products to work. And then, even when the products do work, the developers spend even more time fixing test and user-reported problems.

While few developers recognize that their schedule and cost problems are caused by poor quality practices, an increasing number do. This is a hopeful sign for the future.

Not surprisingly, development management tends to view quality as important only when executives do. And the executives view quality as important only when their customers do. When customers demand quality, executives almost always pay attention. While their actions may not always be effective from a quality-management perspective, they will almost always respond to customer pressure. And even if their customers do not demand improved quality, government regulation can cause both executives and development managers to pay more attention to quality. This is true for commercial aircraft, nuclear power plants, and medical devices. There is little question that, with commercial aircraft for example, the close and continuous regulatory scrutiny coupled with painstaking reviews of every safety incident hold this industry to high quality standards.

Third, Why Don’t Customers Care About Quality?

The simple answer is: “Because defective software works.” The reason it works, however, is because software doesn’t wear out, rot, or otherwise deteriorate. Once it is fixed, it will continue to work as long as it is used in precisely the same way. But, as software systems support an increasing percentage of the national infrastructure, they will be used in progressively less predictable ways. When coupled with the explosive growth of the Internet and the resulting exposure to hackers, criminals, and terrorists, the need for reliable, dependable, and secure software systems will steadily increase. If experience is any guide, as these systems are used to perform more critical functions, they will get more complex and less reliable. Unfortunately, this probably means that it will take a severe, disruptive, and highly public software failure to get people concerned about software quality.

Two forces could change this complacent attitude. One is the Sarbanes-Oxley Act, which makes chief executives and chief financial officers personally responsible for the quality of their organizations’ financial reports. This has caused executives to inquire into the accuracy of their financial reporting systems. What they find is often disquieting. The general accuracy of such systems is usually reasonably good, but there are many sources of error. While these errors have not been a serious concern in the past, they will become much more important when the senior executives are personally liable.

The second issue is closely linked to the first. That concerns software security. Although this is not yet well recognized, when software systems are defective, they cannot be secure. Executives are also just beginning to realize that software security is important to them because, if their systems are not secure, they almost certainly cannot be accurate or reliable. Since they are now personally liable for the accuracy of their financial systems, they now are beginning to appreciate the need for secure financial systems.

How Defective is Software?

So the basic question concerns defects. First, some facts. The number of defects in delivered software products is typically measured in defects per thousand lines of code (KLOC). Figure 1 shows some data on recent history. Here, Capers Jones has substantial data on delivered product defect levels, and he has compared these with the CMM maturity of the organizations that developed the software [Jones 2000]. Noopur Davis has converted the Jones data to defects per million lines of code (MLOC), as shown in Figure 23-1 [Davis 2003]. For example, organizations at CMM level 1 delivered systems with an average of 7,500 defects per MLOC while those at level 5 averaged 1,050 defects per MLOC. What is disquieting about these numbers is that today most products of any sophistication have many thousands and often millions of lines of code. So, while one defect per KLOC may seem like a pretty good quality level, a one million LOC system of this quality would have 1,000 defects. And these are just the functional defects.

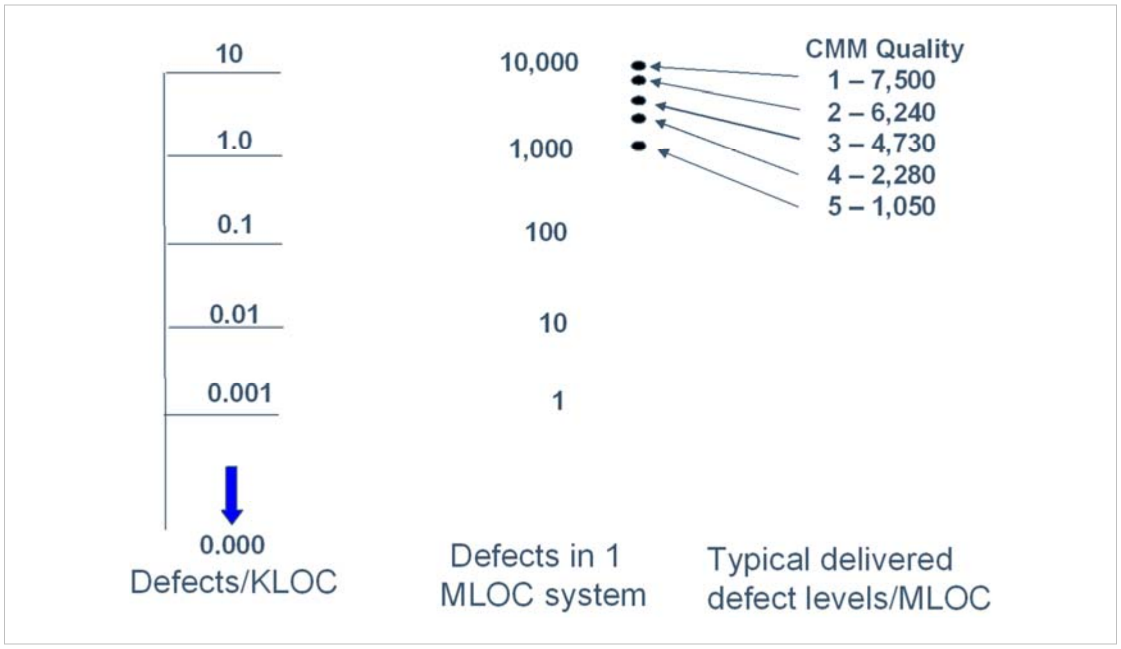

Figure 23-1: Typical Software Quality Levels—in Delivered Defects

Today, most programmers are unaware of many types of security defects. While some security defects are generally recognized as functional defects, others are more subtle. A security defect is any design error that permits hackers, criminals, or terrorists to obtain unauthorized access or use of a software system. Since many of these defects do not cause functional problems, systems that are riddled with security flaws may work just fine and even pass all of their functional tests. And, since many programmers are worried only about getting their programs to function, they are almost totally unaware of their products' potential security vulnerabilities.

In one program that had been supporting a Web site for over two years, a security audit found one functional defect and 16 security defects. So, today's one MLOC systems with 1,000 functional defects could easily have many thousands of security defects, and almost any of these could provide a portal for a hacker, thief, or terrorist to steal money, disrupt records, or to otherwise compromise the accuracy of the organization's financial system. No wonder executives are worried about the Sarbanes-Oxley Act.

Putting Software Quality into Perspective

At present, over 90% of all security vulnerabilities are garden-variety functional defects. This means that our initial goal must be to reduce functional defect levels. From a security perspective, one defect per KLOC is totally inadequate for large systems. But what does one defect per KLOC mean in human terms? One thousand lines of source code, when printed in a listing, typically take about 30 to 40 printed page s. This means that one defect in 30 to 40 printed pages is poor quality. No other human-produced product comes close to this quality level.

But even though one defect per KLOC is far beyond the levels that humans ordinarily achieve, this quality level is totally inadequate. At this quality level, most large systems will contain many thousands of defects, and any one of them could open the door to security problems. So, if 1,000 defects in a one MLOC (million line-of-code) system is inadequate, what would be adequate? That, of course, is impossible to say, but 10 defects per MLOC would be an advance from where we are now. However, I am afraid that even 10 defects per MLOC will not be adequate forever.

Reaching a level of 10 defects per MLOC would mean only one defect in 3,000 to 4,000 pages of listings. That goal seems preposterous. Are such quality levels needed, and is there any hope that we could achieve them? Unfortunately, we almost certainly need such levels today. Criminals, hackers, and terrorists now have and will continue to devise more sophisticated and automated ways to probe our systems for security vulnerabilities. And any single vulnerability could open the door to serious problems.

The problem is not that 10 defects per MLOC is difficult to achieve. The real problem is that, even if it is an adequate security level today, it will almost certainly not be adequate for very long. In the next column I will discuss this problem and explore some of the issues we face in producing systems that have such extraordinary quality levels.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Julia Mullaney, Bill Peterson, Marsha Pomeroy-Huff, and Carol Woody. The letters from John Fox, John Hirota, Pete Lakey, Bob Schaefer, and Dan St. Andre were also very helpful and I thank them as well.

References

[Davis 2003]

Davis, N. & Mullaney, J. *The Team Software Process (TSP) in Practice: A Summary of Recent Results* (CMU/SEI-2003-TR-014). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/library/abstracts/reports/03tr014.cfm

[Jones 2000]

Jones, C. Software Assessments, Benchmarks, and Best Practices. Reading, MA: Addison Wesley, 2000.

24 Security Changes Everything

2004 | Number

The last column began a discussion of software quality. It stressed the growing importance of security as a motivating force for software quality improvement and noted that the recent Sarbanes-Oxley Act is changing management's attitudes about software security and quality. The quality of today's software products falls far short of what is needed for safe and secure systems. We can no long afford to power our critical corporate and national infrastructure with defective software. Since security is such an important concern for many businesses, the pressures for improvement will almost certainly increase.

In this column, I discuss why software quality has not been a customer priority in the past and how that attitude has influenced the quality practices of our industry. I then discuss the changes we will likely see in these quality perceptions and what our customers will soon demand. Finally, I describe the inherent problems with the test-based software quality strategy of today and why it cannot meet our future needs. In the next column, I will outline the required new quality attitude and the prospects for success in our quest for secure and high-quality software.

The Quality Attitude

Even though current industrial-quality software has many defects, quality has not been an important customer concern. This is because even software with many defects works. This situation can best be understood by examining the "Testing Footprint" in Figure 1. The circle represents all of the possible ways to stress test a software system. At the top, for example, is transaction rate and at the bottom is data error. The top left quadrant is for configuration variations and the bottom left is for the resource-contention problems caused by the many possible job-thread variations in multiprogramming and multiprocessing systems. At the right are cases for operator error and hardware failure. The shaded area in the center represents the conditions under which the system is tested.

Since software systems developed with current industrial quality practices enter test with 20 or more defects per thousand lines of code, there are many defects to find in test. Any reasonably well-run testing process will find and fix essentially all of the defects within the testing footprint. If this process is well designed, if it covers the ways most customers will use the system, and if it actually fixes all the defects found, then, as far as these users are concerned, the system would be defect free.

Trying Harder

While current quality practices have been effective in the past, this test-based quality strategy cannot meet our future needs for three reasons.

First, testing is a very expensive way to remove defects. On average, it takes several hours to find and fix each defect in test and, because of the large numbers of defects, software organizations typically spend about half of their time and money on testing.

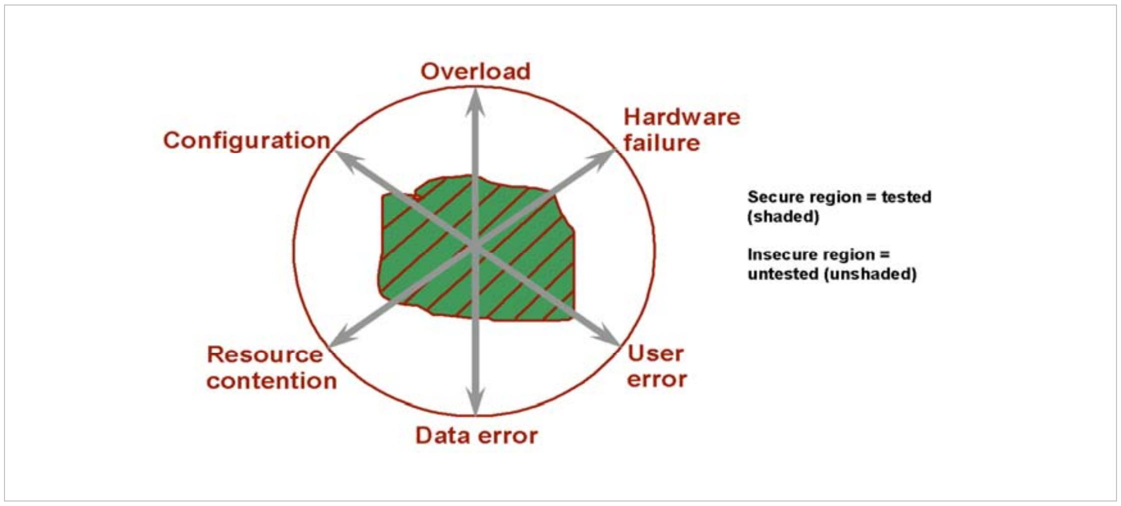

Figure 24-1: The Testing Footprint

The problem is that testing is being misused. Even with all the time and money software organizations spend on testing, they still do not produce quality products. Furthermore, the evidence is that even more testing would not produce significantly better results. The problem is not that testing is a mistake. Testing is essential, and we must run thorough tests if we are to produce quality products. In fact, testing is the best way to gather the quality data we need on our software products and processes.

The second reason that testing cannot meet our quality and security needs stems from the nature of general-purpose computers. When these systems were first commercially introduced, the common view was that only a few would be needed to handle high-volume calculations. But the market surprised the experts. Users found many innovative applications that the experts had not imagined. This user-driven innovation continues even today. So, in principle, any quality strategy that, like testing, requires that the developers anticipate and test all the ways the product will be used simply cannot be effective for the computer industry.

The third reason to modernize our quality practices is that we now face a new category of user. In the past, we developed systems for benign and friendly environments. But the Internet is a different world. Our systems are now exposed to hackers, criminals, and terrorists. These people want to break into our systems and, if they succeed, they can do us great harm.

The reason this will force us to modernize our quality practices is best seen by looking again at the figure. Since the test-based quality strategy can find and fix defects only in the testing footprint, any uses outside of that footprint will likely be running defective code. Since over 90% of all security vulnerabilities are due to common defects, this means that the hackers, criminals, and terrorists have a simple job. All they have to do is drive the system’s operation out of the tested footprint. Then they can likely crash or otherwise disrupt the system.

One answer to this would to be to enlarge the tested footprint. This raises the question of just how big this footprint can get. A quick estimate of the possible combinations of system configuration, interacting job threads, data rates, data errors, user errors, and hardware malfunctions will likely convince you that the number of combinations is astronomical. While there is no way to calculate it precisely, the large-system testing footprint cannot cover even 1% of all the possibilities. This is why modern multi-million line-of-code systems seem to have a limitless supply of defects. Even after years of patching and fixing, defect-discovery rates do not decline. The reason is that these systems enter test with many thousands of defects, and testing typically finds less than half of them. The rest are found by users whenever they use the system in ways that were not anticipated and tested.

The Quality Requirement

Indeed, "security changes everything." Software pervades our lives. It is now common for software defects to disrupt transportation, cause utility failures, enable identity theft, and even result in physical injury or death. The ways that hackers, criminals, and terrorists can exploit the defects in our software are growing faster than the current patch-and-fix strategy can handle. With the testing strategy of the past, the bad guys will always win.

It should be clear from this and the last column that we must strive to deliver defect levels that are at least 100 times better than typically seen today. Furthermore, we need this for new products, and we must similarly improve at least part of the large inventory of existing software. While this is an enormous challenge and it cannot be accomplished overnight, there are signs that it can be done. However, it will require a three-pronged effort.

First, we must change the way we think about quality. Every developer must view quality as a personal responsibility and strive to make every product element defect free. As in other industries, testing can no longer be treated as a separate quality-assurance or testing responsibility. Unless everyone feels personally responsible for the quality of everything he or she produces, we can never achieve the requisite quality levels.

Next, we must gather and use quality data. While trying harder is a natural reaction, people are already trying hard today. No one intentionally leaves defects in a system, and we all strive to produce quality results. It is just that intuition and good intentions cannot possibly improve quality levels by two orders of magnitude. Only a data-driven quality-management strategy can succeed.

Finally, we must change the quality attitude of development management. While cost, schedule, and function will continue to be important, senior management must make quality ***THE*** top priority. Without a single-minded drive for superior quality, the required defect levels are simply not achievable.

Stay tuned in, and in the next column I will discuss the required quality attitude.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Marsha Pomeroy-Huff, Julia Mullaney, Dan Wall, and Alan Willett.

25 The Quality Attitude

2004 | Number 3

This is the third in a series of columns on software quality and security. The first column, "Defective Software Works," discussed the software quality problem, why customers don't care about quality, and how bad software quality really is. The conclusion was that even defective software is of higher quality than just about any other humanly produced product. The second column, entitled "Security Changes Everything," described why testing alone cannot produce quality software. It explained that defective software cannot be secure, and it described the need to improve quality by at least 100 times over where we are today.

This column discusses the quality attitude and how software professionals and their managers must change their view of quality if we are to make much headway with software quality and software security. Quality is key to software performance, whether the concern is with function, usability, reliability, security, or anything else. In the next column, I will outline a strategy for solving these problems and describe steps that software professionals and their management can take.

The Testing Attitude

For as long as I have been writing programs, programmers have believed that when they clean up their programs in test, the programs will work. While this is often true, it isn't always true. This difference is the source of many of our software quality and security problems.

Our programs are often used in unanticipated ways and it is impossible to test even fairly small programs in every way that they could possibly be used. To appreciate the testing problem, consider the simple maze in Figure 1. The highlighted corners are possible branches and an example path from corner A to corner B is indicated by the arrows. Considering only the forward-going paths and ignoring loops, there are 252 possible paths from corner A to corner B in this 5 by 5 matrix. This matrix has only 25 branches. For 100 branches, we would need a 10 by 10 matrix, which would have 184,756 possible forward-going paths.

Most useful programs have many more than 100 branch instructions. For example, one of my C++ programs has 996 lines of code (LOC) and 104 branches. This is about one branch for every ten instructions. Even if large programs only had ten branches per thousand lines of code (KLOC), a 1,000,000 LOC program would have 10,000 branches. For a 100 by 100 square matrix with 10,000 branches, there are an astronomical possible forward-going paths from corner A to corner B. Few developers are even willing to run 252 tests, let alone tests.

A brief analysis shows that testing this number of paths is impossible. For example, if you simultaneously ran a million tests a second for 24 hours a day on a million computers, it would take longer than the lifetime of the universe to run all of these tests. Even then, you would only have tested one set of data values.

Since it is impossible to exhaustively test large programs, we face the next question: “Is this much testing really necessary?” To answer this question, we must consider how many defects there are in typical large software systems. We now know how many defects experienced software developers inject. On average, they inject a defect about every ten lines of code. An analysis of data on more than 8,000 programs written by 810 industrial software developers is shown in Table 25-1.

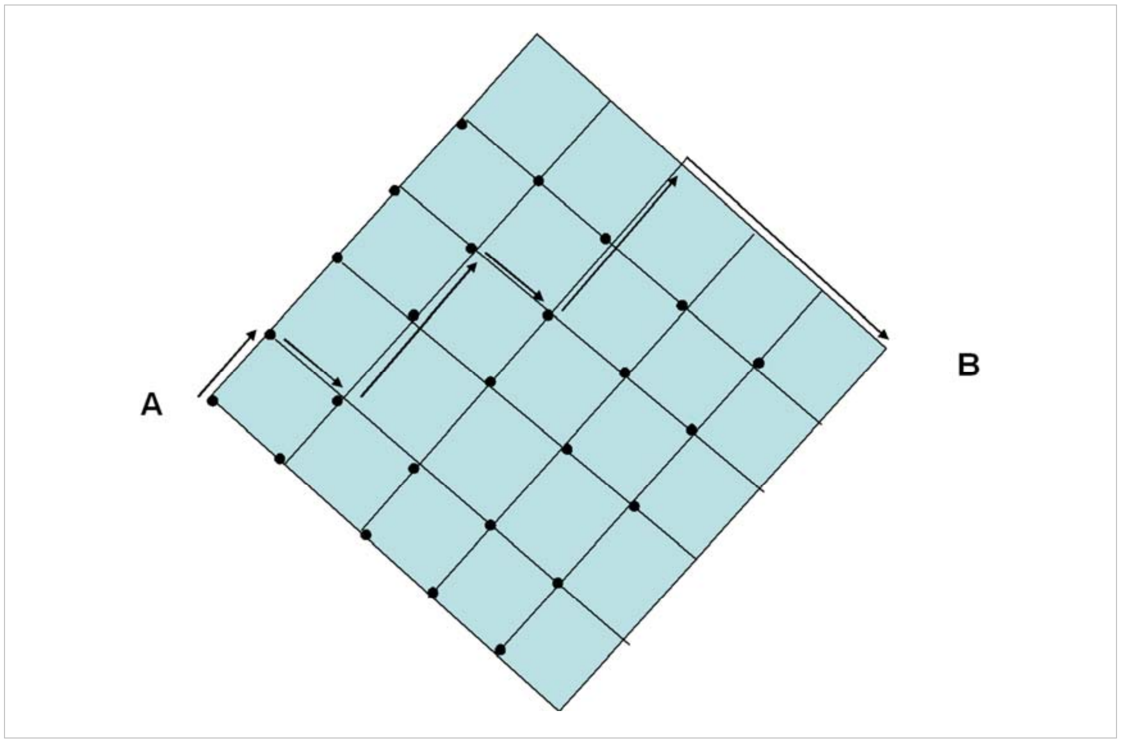

Figure 25-1: Possible Paths Through a Matrix

The average injection rate for these developers is 120 defects per KLOC, or one defect in every eight lines of code. Even the top 10% of the developers injected 29 defects/KLOC and the top 1% injected 11 defects/KLOC. Even at the injection rate for the top 1% of software developers, a 1,000,000 LOC system would enter compiling and testing with 11,000 defects. More typical developers would have 120,000 defects in their products.

Table 25-1: Defect Injection Rages for 810 Experienced Software Developers

Group	Average Defects per KLOC
All	120.8
Upper Quartile	61.9
Upper 10%	28.9
Upper 1%	11.2

That is why most large programs, even after compiling and testing, have from 10 to 20 defects per KLOC when they enter system testing. For a 1,000,000 LOC system, that would be between 10,000 and 20,000 defects. With current practices, large software systems are riddled with defects, and many of these defects cannot be found even by the most extensive testing. Clearly, while testing is essential, *it alone cannot provide the quality we need to have secure systems.*

So, the first required attitude change for software professionals and their managers is to accept the fact that testing alone will not produce quality software systems. Also, since defective software cannot be secure, testing alone won't produce secure systems either.

The Defeatist Attitude

The next attitude that we must change concerns the feasibility of producing secure and high-quality software. Many software experts will tell you that there is no such thing as defect free software. Their obvious conclusion is that we must learn to live with poor-quality and insecure software products. While the previous discussion of testing may have convinced you that this attitude is correct, it is not. What these professionals are really saying is that there is no way to prove that a software system is defect free.

Unfortunately, it is true that there is no way to prove that a software system is defect free. But that is also true for every other kind of product. There is no way to prove that an automobile or a jet airplane or any other product is defect free, but we don't accept that as an excuse for poor quality automobiles or jet aircraft. The question is not to prove that a software product is defect free, but rather to prove that we have taken all practical steps and used all available methods to make the product as close to defect free as possible. There are many ways to do this, and they have been proven highly effective in many other technical fields. While the methods are simple, they are not easy. They require measurement, analysis, a lot of personal attention, and, above all, a conviction that quality is absolutely essential.

The current common view that it is impossible to produce defect free software is an excuse for not making the effort to do quality work. A more subtle form of this attitude is relying exclusively on tools to identify errors. How many times has some programmer told you that a new environment, language, debugger, or analyzer is the answer to the quality or security problem? But then, even after using all of these fancy new gadgets, the resulting products still have defects and are still insecure. The problem is attitude, and no one who counts on some tool or gadget to handle quality will ever produce large, secure software products.

The Quality Attitude

Having the right attitude can make an enormous difference. There is growing evidence that defect free software is possible. Some development groups are now producing reasonably large-scale software products that have had no defects found by their users. While these products actually may have latent defects, for all practical purposes, they are defect free. Today, a few development teams can consistently produce such software.

The Challenge Ahead

We need to learn from and extend these proven quality methods and practices to our highest priority software needs. Today's critical need is to apply these proven quality practices to the really complex and interconnected systems that support the nation's critical infrastructure.

What should be clear from this discussion is that we have not tried hard enough to produce defect free software, so we really cannot know if such high-quality products are possible or not. The next column will discuss the quality methods that people are using today to produce exceptionally high-quality software and the steps required to extend these practices to the truly massive systems that will be common in the future. These systems must be secure and, to be secure, they must be essentially defect free. Learning how to consistently produce such systems is our challenge for the future.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Tim Morrow, Julia Mullaney, Bill Peterson, Marsha Pomeroy-Huff, Alan Willett, and Carol Woody.

In Closing, an Invitation to Readers

In these columns, I discuss software issues and the impact of quality and process on developers and their organizations. However, I am most interested in addressing the issues that you feel are important. So, please drop me a note with your comments, questions, or suggestions. I will read your notes and consider them when planning future columns.

Thanks for your attention and please stay tuned in.

Watts S. Humphrey
watts@sei.cmu.edu

26 A Personal Quality Strategy

2005 | Number 1

This is the fourth column in a four-column series on software quality and security. The first column discussed the software quality problem, why customers don’t care about quality, and how bad software quality really is. The second column described why testing alone cannot produce much higher quality software than we get today. In the third column, I described the quality attitude and how software professionals and their managers must change their view of quality if we are to make much headway improving software quality and security. This column discusses the stages of quality and strategies for addressing the quality problems at each stage. It also describes practices to consider as you improve your personal performance as a software developer. Finally, I comment on the challenges ahead and how the strategies described here can help you to address them.

Quality Stages

There are many ways to look at quality, but, from a software development perspective, we can think about quality as having five stages.

Stage 1—Basic code quality: syntax and coding constructs

Stage 2—Detailed design: the logical construction of programs and the actions required so that these programs perform their specified functions

Stage 3—High-level design: system issues such as interfaces, compatibility, performance, security, and safety

Stage 4—Requirements focused: determining the meaning of the requirements and particularly deducing what is written between the lines.

Stage 5—User driven: users and what we must do to provide them with truly great products. While last in this list, user concerns must have top priority.

Stage 1—Basic Code Quality

At the stage of basic code quality, quality is personal. Either you are fluent in the programming language or you are not. If you are fluent, your principal concerns are typos and occasional obvious mistakes. I once tracked my errors for a large data-entry job and found that I made 1.74 errors per thousand keystrokes, or 2.4 errors per hour of key entry. Since a line of my C++ code averages 17 keystrokes, my key-entry error rate alone injected 29.5 defects per 1,000 lines of code.

This was just typing. I did not consider program semantics, functions, or design. To correct this type of basic mistake, you need to track the defects and understand the errors that caused them. Until you do, you cannot prevent these types of defects or get better at finding and fixing them. Many developers object to tracking defects that the compiler could quickly find. The reason to record these defects, however, is to understand the mistakes you make no matter how they are found. Once you have defect data, analyze it. By recording your defects and analyzing the data, you are likely to cut your defect-injection rate by about 50%. Also, keep recording the defect data. If you don't, your injection rate will creep back up.

If you are not yet fluent with the programming language, take the same three steps: track your defects, analyze the data, and continue doing this as a regular part of your programming work. By tracking and analyzing my defects and using the defect data to guide my personal design and code reviews, I substantially cut the time it took me to learn a new programming language and also dramatically improved the quality of my products.

The obvious next question is, "Why bother with these Stage 1 defects, since the compiler will find most of them anyway?" The brief answer is that tools and testing will not find a significant number of your defects, and any that you miss will be very expensive for you, the testers, or the users to find and fix later. (For a more complete answer, consult my previous three columns mentioned above.) If you don't clean up the Stage 1 defects first, they will make it harder for you and your teammates to find and fix the more sophisticated defects at the higher quality stages. You will also waste a lot of test time fixing defects that you could have quickly found and fixed beforehand.

Stage 2—Detailed-Design Quality

At the detailed-design stage, the problems are more sophisticated. Most programmers make design mistakes not because they don't know how to design, but because they never actually produced a design. They may have drawn some bubble charts or sketched a few use cases, but they never reduced these designs to a specification for writing code.

The practice of designing while coding is error prone. From data on 3,240 programs written in Personal Software Process (PSP) courses, the SEI has found that experienced developers inject fewer defects when designing (2.0 defects per hour) than when they design while coding (4.6 defects per hour). If you want low-defect designs, you must produce those designs, instead of just creating them while coding.

There are two big advantages to producing designs. First, you will make less than half as many design mistakes, and second, you will have a design that you can review and correct. However, in doing the design review, use good review methods. This is too big a subject to cover here, but it is covered in my book *The Personal Software Process—A Discipline for Software Engineers*, published in March 2005.

To do design reviews properly, you must spend enough time doing them. From the data on the same 3,240 PSP programs, the average defect-removal rate during design reviews was 3.3 defects per hour. Therefore, if you inject defects at the rate of 2.0 per hour and remove them at 3.3 per hour, you must spend about 36 minutes reviewing the design for every hour you spent producing it.

The detailed-design stage is particularly important because this is where you design the logic paths that must be tested. If some of the paths through your module have subtle defects and if you don’t find and fix them, they will be left for the testers or users to find. Since you presumably will have tested your program before passing it on to test, the remaining defects will be the ones that did not show up in your testing. In other words, these defects do not prevent the program from working except under specialized conditions. To find them, the test must cover the right paths and it must have the proper parameter or variable values.

Assuming that your programs have about the same proportion of branch instructions as my C++ programs, your typical 500 line-of-code (LOC) module would have about 50 branch instructions. From the data in Table 26-1, your program would then have about 3,400 possible test paths, any one of which could contain a design defect that you missed. That is why the strategy of testing in quality takes so long and produces defective products.

Table 26-1: Testing Paths

Maze Size	Branches	Paths	Maze Size	Branches	Paths
1	1	2	11	121	705,432
2	4	6	12	144	2,704,156
3	9	20	13	169	10,400,600
4	16	70	14	196	40,116,600
5	25	252	15	225	1.55E+08
6	36	924	16	256	6.01E+08
7	49	3,432	17	289	2.33E+09
8	64	12,870	18	324	9.08E+09
9	81	48,620	19	361	3.53E+10
10	100	184,756	20	400	1.38E+11

The key practices at Stage 2, the detailed-design stage, are to produce complete designs, review the designs yourself, and have your teammates carefully inspect them. Then, of course, implement the program and thoroughly review, inspect, and test it to make sure that the code properly reflects the design. If you and your teammates do this for every module you and your team develop, you will improve the quality of your programs by at least 10 times and probably much more. You will also save a lot of test time.

Stage 3—High-Level Design Quality

Through Stage 2, the defects are yours. You inject them, and you are the best person to find and fix them. Above this stage, you must work with others. The defects at Stage 3 concern the interfaces, interdependencies, and interactions of your program with the other parts of the system. Defects in any module could affect system properties such as performance, security, and safety. These properties result from the correct operation of all or most of the parts of the system. This is another case where sound design practices are important. For security, for example, a properly-produced high-level design would specify the authentication practices and the data-security and protection conventions that the modules should follow.

At Stage 3, the key quality practices are first, to get and review the high-level design specifications for your module. If these specifications don't exist or are inadequate, work with your teammates and system designers to clarify the design specifications. Second, after you obtain these specifications, follow them in producing the module design. Then, third, review the design to ensure that it meets these specifications and that it will work with all of the other modules. If you don't do this, there is a good chance that the modules you develop will not work properly with the rest of the system. Often, such problems cannot be fixed without a major module redesign or a complete module replacement. Finally, get your team's help in thoroughly inspecting and correcting the design. This is particularly important because, at this stage, your products begin to address system-level issues that you may not even be aware of.

Stage 4—Requirements-Focused Quality

At Stages 1, 2, and 3, you work with familiar issues. At the requirements stage, however, you are no longer an expert. The two principal difficulties with requirements problems are, first, that requirements are not your field of specialty. Therefore, it is easy to think that you understand something when you do not. Second, misunderstandings and errors in requirements interpretation can easily cause you to build the wrong product.

If not caught early, requirements misunderstandings can be fatal. This is why you should start with a thorough examination of the requirements and resolve any confusion or uncertainty with requirements experts before starting to design the product. While many requirements details need not be settled at this point, it is hard to know what is a detail and what is fundamental. Once you are familiar with the requirements, settle the critical points before starting the design work and resolve the remaining issues in parallel with the design and implementation work. The two key practices at Stage 4 are to understand the requirements and to resolve any confusion or uncertainty before starting on the design. In implementing these practices, get your team's help as well as the help of the organization's systems designers or requirements experts. Some of these people have been thinking about this requirement for a long time and will likely understand the user's needs better than you possibly could with just a few brief weeks or months of exposure.

Stage 5—User-Driven Quality

The user stage is a totally different ballgame. Here, as shown in Figure 1, the problems are not with what you know or even what you don't know, they are with what you don't know that you don't know. This problem is particularly tricky because the users often don't understand the issues either. A truly great product must perform a desirable user function in a convenient and elegant way. While the users may think that they know what they need and have strong opinions on how the product should work, they will often be wrong. They won't completely understand or be able to specify the functions that they want, and their view of how to build a product to perform these functions will approximate what they do today, rather than some elegant new approach.

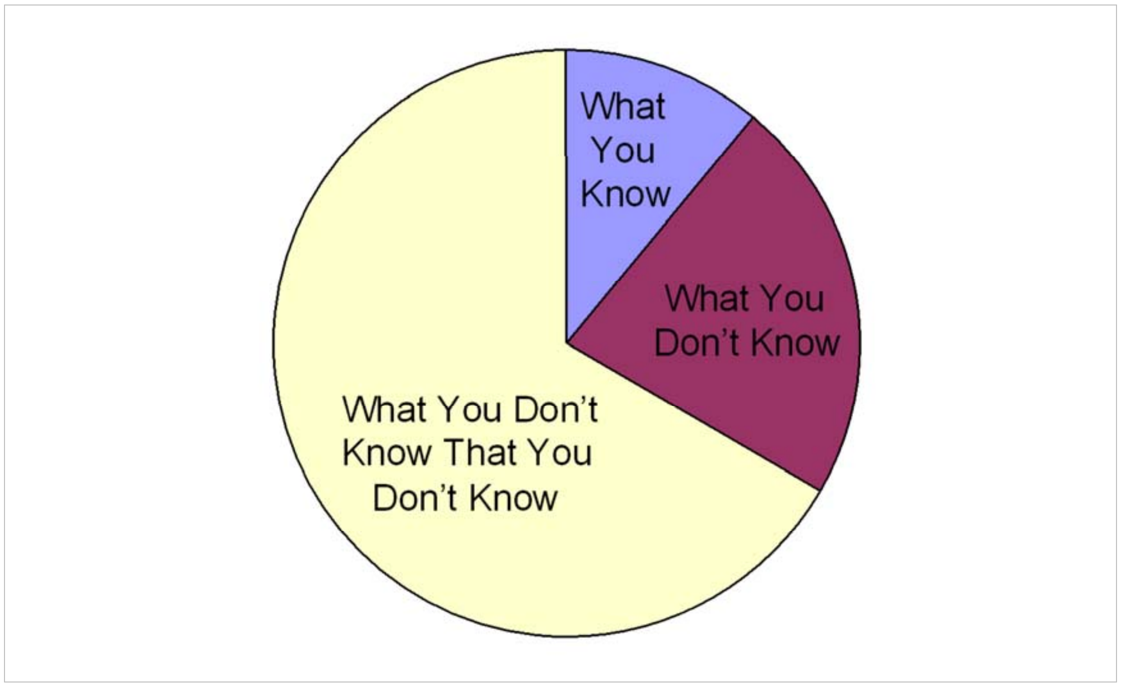

Figure 26-1: What We Know and Don't Know

The challenge at the user stage is to get a clear understanding of what the users think they want and then to develop an intuitive sense for what they really need. Once you do this, keep thinking about the problem and you could come up with a truly creative solution. This is how breakthroughs like the spreadsheet or mouse were conceived. A developer who understood the problem had a "crazy" idea that worked. While these creative leaps don't happen often, preparing for them will help you to build a fine product. You will have solved a problem in a better way than you otherwise would have, and you will have given yourself the chance to do something great.

What makes the user-driven stage so different is that elegant solutions often change the problem, and sometimes they change it in fundamental ways. Your objective at this stage should be to develop a deep enough intuitive understanding of the user's needs so you can see the opportunities for dramatic departures that will transform the problem while enabling a breakthrough solution.

Putting It All Together

Quality is an individual issue. If you really want to do great work, there are lots of ways to do it. You may have to settle for incomplete requirements and specifications, and you may not even be able to talk to any users, but you can always do a first-class job at Stages 1, 2, and 3. Keep thinking about the higher quality stages and what you can do to extend yourself and your team to do great work. You may not succeed often, but it is worth the try. Major advances take time, a lot of insight, and some perspiration. They also take an occasional inspiration. Even if it takes many years, a great product is something that you will always be proud of. So keep trying to do quality work. It will make your life much more rewarding.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Dan Burton, Jim McHale, Bill Peterson, Marsha Pomeroy-Huff, and Dan Wall.

In Closing, an Invitation to Readers

In these columns, I discuss software issues and the impact of quality and process on developers and their organizations. However, I am most interested in addressing the issues that you feel are important. So, please drop me a note with your comments, questions, or suggestions. I will read your notes and consider them when planning future columns.

Thanks for your attention and please stay tuned in.

Watts S. Humphrey
watts@sei.cmu.edu

27 Large-Scale Work—Part I: the Organization

2005 | Number 2

With this column, I start a new series of articles about the issues faced by large-scale development projects. Since large-scale development is an enormous subject, I plan to touch only on those topics that I think are particularly interesting or especially important. I currently plan to cover two problems. First, large software projects are almost universally troubled, and second, all large-scale systems-development projects of almost every kind now involve large amounts of software. Unfortunately, this implies that almost all kinds of large-scale development projects will be troubled unless we can devise a better way to develop the software parts of these large-scale systems. The increasing need for large-scale system-development projects raises many questions and presents a significant challenge to those of us in the development business.

Emergent Properties of Systems

Perhaps the greatest single problem with large-scale system development concerns what are called the emergent properties of these systems. These are those properties of the entire system that are not embodied in any of the system's parts. Examples are system security, safety, and performance. While individual components of a system can contribute to safety, security, and performance problems, no component by itself can generally be relied on to make a system safe, secure, or high performing.

The reason that emergent properties are a problem for large-scale systems is related to the way in which we develop these systems. As projects get larger, we structure the overall job into subsystems, then structure the subsystems into products, and refine the products even further into components and possibly even into modules or parts. The objective of this refinement process is to arrive at a series of "bite-sized projects" that development teams or individual developers can design and develop.

This refinement process can be effective as long as the interfaces among the system's parts are well defined and the parts are sufficiently independent that they can be independently developed. Unfortunately, the nature of emergent properties is that they depend on the cooperative behavior of many, if not all, of a system's parts. This would not be a problem if the system's overall design could completely and precisely specify the properties required of all of the components. For large-scale systems, however, this is rarely possible.

While people have always handled big jobs by breaking them into numerous smaller jobs, this can cause problems when the jobs' parts have interdependencies. System performance, for example, has always been a problem, but we have generally been able to overpower it. That is, the raw power of our technology has often provided the desired performance levels even when the system structure contains many inefficiencies and delays.

As the scale of our systems increases and the emergent properties become increasingly important, we now face two difficult problems. First, the structural complexity of our large organizations makes the development process less efficient. Since large-scale systems are generally developed by large and complex organizations, and since these large organizations generally distribute large projects across multiple organizational units and locations, these large projects tend also to have complex structures. This added complexity both complicates the work and takes added resources and time.

The second problem is that, as the new set of emergent properties becomes more important, we can no longer rely on technology to overpower the design problem. Security, for example, is not something we can solve with a brute-force design. Security problems often result from subtle combinations of conditions that interact to produce an insecure situation. What is worse, these problems are rarely detectable at the module or part levels.

A War Story

Some years ago, while I still worked at IBM, I was made director of programming. My organization had about 4,000 software professionals in 15 laboratories and 6 countries. While this group was responsible for developing all of the software to support IBM's products, about half of the group was working on one big system: OS/360. These people were highly capable and motivated, but the OS/360 schedule had slipped three times during the past year and a half, and no one believed any of our dates. Since IBM was starting to ship the 360 hardware without the software, the lack of a believable schedule was a major marketing problem.

Marketing argued that the customers would be willing to run their existing system software temporarily with the emulators provided with the 360 systems, but that they would not order many new systems until they had a believable plan for OS/360 software support. We needed a schedule right away, but it had to be one that we would meet without fail. If we had another schedule slip, no one would believe us again.

Before this job, I had run several software projects and also managed some large hardware projects. Therefore, I had learned that it is practically impossible to produce a reliable schedule without first producing a detailed plan for the work. Since the several thousand developers in my group had never before made detailed plans, they didn't know how to do it. What was worse, they didn't believe that planning was important. They knew how to code and test, so that is what they did. Without plans to guide them, the projects were pretty chaotic, and nobody had time to do anything but code and test.

I had to do something to make planning important. So, to get everybody's attention, I established a new operating procedure: Without a detailed plan, no one could announce or ship any software product. I even threatened to cut the budget for any project that did not have a plan. I gave the groups 60 days to produce plans and to review them with me personally. This made planning a crisis. When I got the plans, I reviewed them all and made the developers defend them. We lengthened many of the schedules but never cut a single one. In fact, I even added a 90-day cushion to every committed date.

This worked. We did not miss a single date for the next two and a half years. Soon, however, we started to eat up my 90-day cushion. We had not appreciated the consequences of a multi-release delivery plan. Every schedule slip for every release had a cumulative effect on every subsequent release. After about two years, these small schedule slips finally added up to my 90-day cushion. However, by then everybody believed our dates, so the subsequent minor schedule adjustment was not a problem. But after that, we no longer committed delivery dates to customers that were more than about a year out. By then, we were meeting all of our delivery dates, and even beating the hardware.

The project-planning procedure worked extremely well. While it imposed a demanding planning discipline on the development groups, they continued producing plans even after I moved on to another job. Unfortunately, with the mechanism we established for plan management, it was easy to add further procedures, so many managers did. Over time, this simple plan-management system became a bureaucratic jungle. The initial planning controls we established had been relatively simple and saved IBM billions of dollars. However, with the added controls, projects could no longer respond quickly to special customer needs, and bureaucracy became a big company problem.

The Tropical Rain Forest

The fundamental problem of scale is illustrated by analogy to the ecological energy balance in a tropical rain forest. In essence, as the forest grows, it develops an increasingly complex structure. As the root system, undergrowth, and canopy grow more complex, it takes an increasing percentage of the ecosystem's available energy just to sustain the jungle's complexity. Finally, when this complexity consumes all of the available energy, growth stops.

The implication for both projects and organizations is that, as they grow, their structure gets progressively more complex, and this increasingly complex structure makes it harder and harder for the developers to do productive work. Finally, at some point, the organization gets so big and so complex that the development groups can no longer get their work done in an orderly, timely, and productive way. Since this is a drastic condition, it is important to understand the mechanisms that cause it.

Organizational Growth

In principle, organizations grow because there is more work to do than the current staff can handle. However, this problem is usually more than just a question of volume. As the scale increases, responsibilities are subdivided and issues that could once be handled informally must be handled by specialized groups. So, in scaling up the organization, we subdivide responsibilities into progressively smaller and less meaningful business elements. Tasks that could once be handled informally by the projects themselves are addressed by specialized staffs. Now, each staff has the sole job of ensuring that each project does this one aspect of its job according to the rules. Furthermore, since each staff's responsibility is far removed from business concerns, normal business-based or marketing-based arguments are rarely effective. The staffs' seemingly arbitrary goals and procedures must either be obeyed or overruled.

This growth process generally happens almost accidentally. A problem comes up, such as a missed schedule, and management decides that future similar problems must be prevented. So they establish a special procedure and group to concentrate on that one problem. In my case, this was a cost-estimating and planning function that required a plan from every project. Each new special procedure and group is like scar tissue and each added bit of scar tissue contributes to the inflexibility of the organization and makes it harder for the developers to do their work. Example staffs are pricing, scheduling, configuration management, system testing, quality assurance, security, and many others.

The Enemy

Since these specialized staffs are all designed to monitor and control the projects, the projects view them as the enemy. So, if they can get away with it, the projects simply ignore the staffs. To guard against this, management establishes a review procedure where the staffs have the authority to sign off at key project milestones. If the project team does not do its job properly, the staff does not let the project proceed. What is most insidious about this situation is that the staffs are all enforcing rules that management and most objective observers would agree are needed. However, because of the tangle of approvals and sign-offs, the process consumes a great deal of the project's energy. Also, once the staffs have power, they often use that power to push pet objectives that are not strictly in their official charter.

While this review and approval strategy generally guarantees that the projects pay attention to the staffs, the projects must also surmount an increasingly complex bureaucratic tangle of approvals just to do their work. This problem is bad enough when all projects were similar but, as projects get larger and more complex, they, too, become increasingly specialized. They then must use custom-tailored processes and procedures that are almost impossible to establish in a large and complex organizational jungle. Because of the energy consumed in getting bureaucratic approvals, the projects generally have to follow a process that doesn't precisely fit their needs. Now, just like a tropical rain forest, an increasing proportion of the organization's resources are devoted to delaying or stopping projects. Because of the enormous energy required to fight the bureaucracy, the projects have progressively less energy left to design and build their products. That is when rapid growth stops.

While there is no magic solution to the problems of project scale, the situation is not hopeless, and there are useful steps we can take. Subsequent columns outline the most promising avenues for addressing these problems. I will also outline some principles to consider when working on large-scale development projects, some of the consequences of scale-related development problems, and some strategies for solving them.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Dan Burton, Julia Mullaney, Bill Peterson, Marsha Pomeroy-Huff, and Mark Sebern.

In Closing, an Invitation to Readers

In this particular series of columns, I discuss some of the development issues related to large-scale projects. Since this is an enormous subject, I cannot hope to be comprehensive but I do want to address the issues you feel strongly about. So, if there are aspects of this subject that you feel are particularly important and would like covered, please drop me a note with your comments, questions, or suggestions. Better yet, include a war story or brief anecdote that illustrates your ideas. I will read your notes and consider them when planning future columns.

Thanks for your attention and please stay tuned in.

Watts S. Humphrey
watts@sei.cmu.edu

28 Large-Scale Work—Part II: The Project

2005 | Number 3

This is the second column in a series on large-scale development work. The first column briefly discussed several important large-scale-project issues and then addressed the problems of organization size. It described how organizations get more complex as they grow and how this complexity saps the energy that the organization has available for productive work. I discussed the reasons for these problems as well as their consequences for development projects. In this column, I continue this discussion with primary focus on the project itself and the principal project-management issues. Large projects generally exist in large organizations, and they are also usually spread across multiple departments and locations. This affects their ability to operate and to produce high-quality products for predictable costs and schedules.

Large-Systems Problems

While large-scale projects face a host of problems, the principal topics I will discuss are project management and people management. Technical issues are critically important, and no amount of process, project, and people management can compensate for poor technical work, but this subject is too vast to usefully discuss in a brief column. Therefore, this column focuses on project management, and subsequent columns will deal with the people-management topics in managing large, distributed, multi-organizational projects. The major problems I have seen concern delegation and coordination.

A War Story

One of the best examples I know of the typical problems of managing large projects is a situation that developed shortly after I took over as IBM's director of programming. Our two largest locations were in Poughkeepsie, N.Y., and San Jose, California. Because the Poughkeepsie group had grown to nearly 1,000 developers and because it still needed many more people to handle its committed work, we had decided to move the data-management mission from Poughkeepsie to San Jose. The problem was that the interface between the Poughkeepsie operating-system work and the San Jose data-management products had not been defined.

We had told the two groups to agree on the interface definition and to come to me if they could not. Unfortunately, the interface definition had become a highly contentious issue with no clearly best answer, and the two groups were unable to agree. One morning, the two laboratory managers with about 30 of their best technical people showed up in my office with two competing proposals. Although I knew less about this subject than anyone else in the room, I ended up designing the interface. I told the groups to either make my definition work or to agree on a better solution and to let me know what they agreed on. They never came back.

This was a management failure. The problem had started many months before when what should have been a technical issue had become highly political. Then the technical experts were essentially forced to defend their managements' positions rather than carefully considering the pros and cons of the various alternate ways to design the interface. We therefore ended up making a political decision on a technical issue.

The Management Error

While the way I handled this interface problem was clearly a management failure, what was the specific failing, and what should I have done differently? In retrospect, I believe that the problem was caused by differing goals and objectives. The Poughkeepsie manager was the OS/360 project manager while the San Jose manager was a laboratory manager with multiple projects, one of which was the OS/360 data-management work. Therefore, they had no common basis for agreeing on what was the best solution. Rather than making an instant decision, I should have made these two groups resolve the problem properly.

I now believe that the proper approach would have been to start with the goals to be met by the decision. That was a topic where I, as the senior manager, could have made a real contribution. Then, after reaching agreement on the goals to be met by the interface decision, we should have defined the criteria for making the decision. After settling the goal and criteria questions, I should then have sent the groups off to make the decision themselves and asked them to either report their conclusion and rationale to me or to come back if they needed further help.

The Generic Problem

The basic problem that this story illustrates is that technically trained managers love to make technical decisions, particularly after they have become executives. We all have this image of decisive executives who can be relied on to make almost instant decisions. This gives them a gratifying sense of power and fosters an image of infallibility. There is this myth that indecision is bad and that projects must have clear and precise direction at all times. This is not only wrong, but dangerous, particularly for very large and highly complex systems.

The design of large systems involves many decisions, most of which could be made in several ways with little, if any, effect on overall system performance. However, a few of these decisions are critical. If these critical decisions are not made properly, the consequences could be severe. Unfortunately, these decisions don't come with a sign that says they are critical. The two examples that come immediately to mind both concern NASA's handling of two space shuttle decisions: the technical question of O-ring safety for a low-temperature launch and the question of potential damage caused when falling hunks of insulation hit the shuttle wing.

The Critical Point

To me, one of the most critical aspects of managing large-scale projects is making sure that decisions are properly made. The executive's responsibility must be to identify the right people to make the decision, insist that the goals used for making the decision be defined and documented, and require that the criteria for the decision be established. While there are far too many technical decisions in large-scale projects for management to require that they all be made in this way, there are a relatively few times when technical decisions are escalated to senior management. However, whenever they are, these decisions are almost certainly technical issues that have become political.

If the executive does not insist that each of these politically tinged technical decisions is properly made, he or she is likely making a very big and possibly fatal mistake. If ever, this is the one time when the executive should insist that the decision be made in the right way. While these decision situations always come up when there is no time and when everybody, including the executive, is in a rush to get on with the job, this is precisely the time when proper decision making is most important. When executives insist that rush decisions be made in the proper way, they are demonstrating their ability to be technical executives. Therefore, the first two ground rules for the proper management of large-scale projects are the following:

1. Insist that all technical decisions be made by the proper technical people.
2. Make sure that, in making these decisions, the technical decision makers thoughtfully evaluate the available alternatives against clearly defined criteria.

The Broader Lesson

While this might appear to be the end of the story, there is a broader lesson that is even more important. This concerns management development. There are problems at every management level in large organizations. At every level, managers or executives should be most concerned with the way their subordinates operate. The challenge at every level is to delegate and to get those at lower levels to take responsibility for making the decisions that they should make. The managers should insist that their subordinates make all of these decisions and that they make them in the right way.

With few exceptions, the depth of technical knowledge in organizations increases at lower levels of the management hierarchy. This means that when decisions are made with proper goals and sound criteria, they are invariably better decisions when they are made at the lowest possible levels in the organization. So delegation is critical, but with delegation must also come the responsibility to monitor behavior. So, while looking at cost, schedule, and quality performance, executives and managers should also look at the key decisions and how they were made.

It might seem that only senior managers and executives need be concerned with these issues. However, a principal problem for executives in large organizations is the difficulty of getting their technical people to take responsibility and to make sound and timely decisions on their own. The problem, of course, is not just the ability of technical people to take responsibility but also their ability to handle this responsibility responsibly.

In the next column, I discuss how developers and team leaders should act when making technical decisions. I will also outline some principles for team leaders and team members to consider when working on large-scale development projects and some strategies that can help in following these principles.

Acknowledgments

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Bob Cannon, Julia Mullaney, Bill Peterson, and Marsha Pomeroy-Huff.

In Closing, an Invitation to Readers

In this particular series of columns, I discuss some of the development issues related to large-scale projects. Since this is an enormous subject, I cannot hope to be comprehensive, but I do want to address the issues that you feel strongly about. So, if there are aspects of this subject that you feel are particularly important and would like covered, please drop me a note with your comments, questions, or suggestions. Better yet, include a war story or brief anecdote that illustrates your ideas. I will read your notes and consider them when planning future columns.

Thanks for your attention and please stay tuned in.

Watts S. Humphrey
watts@sei.cmu.edu

29 Large-Scale Work—Part III: The People

2005 | Number 4

This is the third column in the series on large-scale development work. The first column briefly summarized the issues of large-scale development work, addressed the problems of organizational size, and described how organizations necessarily become less efficient as they grow. The second column dealt with project issues, particularly with the management-decision process. Often, the most serious problems in large projects concern the way decisions are made. In this column, I continue this discussion with a principal focus on the people and teams that make up the project staff and the issues involved in maximizing their performance.

Yet Another War Story

While I was still at IBM, Peggy, a project manager, came to me for advice on how to handle a disagreement with marketing over her planned product. She had worked in my group a few years before and knew that I had dealt with such problems many times. She wanted my advice on how to proceed. Marketing staff insisted she add some functions that she did not believe were worthwhile. She debated this issue with the marketing representatives, and they were adamant. However, she had also talked to IBM's GUIDE user group about the issue, as well as to several customers who were members of her user advisory board, and none of them felt that the functions were important enough to warrant delaying the product's availability. They wanted the product as soon as they could get it, and these added functions would cause a delay of several months.

Marketing still held out. I suggested that Peggy inform marketing that she was proceeding without their approval and that she write a note to her management and copy marketing management explaining her position. I told her that the marketing people would be surprised and that they wouldn't know what to do. This was not a sufficiently important issue to involve senior management, so if she were firm and had a sound business and technical position, marketing would cave in. That is what she did, and marketing did cave in.

Bureaucratic Blackmail

As organizations grow, they become increasingly bureaucratic: staffs and review departments are established to monitor the operating groups. The larger organization also needs larger bureaucratic functions to handle the organization's routine activities like delivering the mail, handling the payroll, running the facilities, and obtaining and distributing supplies. As the size of these staffs increases, their efficiency and effectiveness become a major concern, and management is likely to establish even more bureaucracy to monitor, measure, and track the growing bureaucracy.

So, bureaucracies grow seemingly without bound. Every new problem causes a reaction that often adds to the bureaucracy and makes organizations less responsive, more rigid, and less efficient than before. Of course, some bureaucracy is essential, and bureaucratic procedures can often be helpful. There are lots of routine things that must be done in any large organization, and neither management nor development groups want to worry about them. With an administrative function to handle them, these routine things become automatic.

Once these functions become automatic, however, they also become rigid and hard to change. Therefore, since development's job is to develop new things, which invariably requires some degree of change, and since the bureaucracy's job is to resist change, conflict is inevitable. This means that in large organizations, to get anything done, development groups must know how to overcome bureaucratic resistance. This task is not trivial because, once these bureaucratic groups get power, they tend to use that power in ways that management had not intended.

IBM management did not want the marketing group dictating to the project manager which functions should be included in her product. While marketing should have a voice, and development groups should take advantage of their guidance, product managers must be responsible for the performance of their products. This means that they must have the final say on what goes into their products, how the company's money is spent, and how their products are introduced and offered.

While Peggy's experience would seem to put the marketing people in a bad light, this is just an example of a staff group that overreached. There are many other cases in which such groups properly use their sign-off authority and help product managers produce truly superior products. There are also other cases in which staff groups identify real product problems and help to avert real mistakes.

Perhaps the worst case I know of in which a group used bureaucratic procedures to force a technical position happened some years ago when IBM was considering developing a new low-cost digital fax machine. The initial product concept was for a simple machine that would produce high-quality fax output. At the time, there was no comparable machine available. The marketing people kept pushing for additional functions, and the product manager kept caving in. Finally, after many months, the planned product's costs had grown so much that the product could no longer meet the target price for a low-end machine. By the time the product was replanned for the proper market niche, a competitive machine had been introduced, and the market opportunity was lost.

If this product manager had fought for the original low-end product concept, IBM would have had an attractive new product, well ahead of the competition. So the key question is, "How can a developer operate in a large and increasingly rigid bureaucratic morass?" As the size and power of these bureaucratic groups increase, it seems that everybody is trying to block the developer's work. Everyone can say "no," and, seemingly, no one can say "yes." It's like "death by a thousand cuts." If you spend your life fighting the bureaucracy, you won't have time to do anything else. How can developers get anything done? There are a few guidelines that can help.

First, don't sweat the small stuff. While none of us likes inefficiency and seemingly-stupid bureaucratic rules and regulations, assume that this stuff is there for a reason, and as long as it doesn't block your ability to do your job, or if you can get around it without too much pain, learn to ignore it. Somebody else is worrying about organizational efficiency, so concentrate on your job of developing products.

In one example of this problem, Bret had just been made manager of a project to build a new communication device. As he talked to the developers, one of the new engineers complained about the stock room. It never had the right parts in stock, and the engineers often had to wait weeks for the parts they needed. Bret thought that this should be an easy problem to solve, so he talked to the stock-room manager. He found that there were all kinds of outdated rules governing stocking levels, reorder procedures, and purchase authorizations. After several days of digging, he found that, to change the system, he would need the lab director's approval. He wisely decided not to pursue the matter.

Second, if the bureaucracy gets in the way, try to think of how you could do your job in a way that would not cause those bureaucratic problems. We often have choices in doing our work, and if some simple adjustments will suffice, make the changes and don't fight over them. Before you go into battle, make sure there is no simple procedural fix that will do the job.

Instead of going to the lab director, Bret discussed the situation with Pete, a seasoned technician. When he told Pete about the problem, Pete asked: "What do you need?" Bret told him, and the next morning, Pete walked into his office with the parts. When Bret asked where he got them, Pete described the "midnight requisition" system that the technicians had developed to get scarce parts. Because the stock room was so antiquated, the technicians on all the projects had gradually accumulated enough private stock of frequently needed parts to meet their projects' emergency needs. By checking with his buddies, Pete quickly got the parts he needed.

Third, when the bureaucracy really does get in your way, you will have to do battle. When you do, focus on getting your job done. In these fights, remember that the bureaucracy has one strong point—procedures. You are doing battle because what you want to do doesn't fit their established procedures, and you will always lose battles over procedures. So don't propose changing procedures or even new ways of interpreting procedures. Your strong points are business oriented: customer responsiveness, product enhancements, or other business needs that you are charged with addressing. As you escalate issues, your position should be, "I don't understand your procedural problem; all I know is that we have to address this business problem, and I need your help in figuring out how to do that."

Some weeks later, while testing the first system prototype model, Bret's team had a disaster and burned out several critical parts. A customer demo was scheduled in a couple of weeks, and the stock room said it would take a month to get the needed parts. Pete's midnight requisition system couldn't help but he did locate a supply of the parts on eBay. Since the stock room had no way to buy parts on eBay, however, Bret was stymied. This time, he went to the lab director and got the authorization to buy the parts on eBay.

You may have to keep escalating a bureaucratic problem until you reach an executive who is responsible for both bureaucratic and business issues. But once you do, if you have a clear and convincing business story, that executive will always support you. Then you will get the help you need, and the bureaucrats will be directed to help you.

In the next column, I continue this discussion of large-scale work with a special focus on the politics of organizations and why democratic principles are particularly hard to apply in large development groups.

Acknowledgments

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Marsha Pomeroy-Huff, Susan Kushner, and Julia Mullaney.

In Closing, an Invitation to Readers

In this series of columns, I discuss some development issues related to large-scale projects. Since this is an enormous subject, I cannot hope to be comprehensive, but I do want to address the issues that you feel strongly about. So, if there are aspects of this subject that you feel are particularly important and would like covered, please drop me a note with your comments, questions, or suggestions. Better yet, include a war story or brief anecdote that illustrates your ideas. I will read your notes and consider them when planning future columns.

Thanks for your attention, and please stay tuned in.

Watts S. Humphrey
watts@sei.cmu.edu

30 Large-Scale Work—Part IV: The Management System

2006 | Number 3

This is the fourth column in a series on large-scale development work. The first column ("Large-Scale Work—Part I: the Organization" on page 141) briefly summarized the issues of large-scale system development and discussed the problems of organization size, particularly how, as organizations grow, they become less efficient. The second column ("Large-Scale Work—Part II: The Project" on page 147) dealt with project problems and particularly with the management decision process. Often, the most serious problems in large projects concern the way decisions are made. In the third column ("Large-Scale Work—Part III: The People" on page 151), I focused on the people and teams that make up the project staff and the issues involved in maximizing their performance.

In this column, I address the problems of applying democratic management principles to large projects. As has been seen in politics, autocratic management is not effective for running large and complex modern countries. Autocratic management is also ineffective for running large and complex industrial organizations and is even less effective for managing large-scale development work.

Moving from an autocratic to a democratic management system is not a trivial process, however. Not only are management actions required, but both the developers and their management must be actively involved in establishing and retaining democratic management principles. This column explores democratic principles and the need for problem ownership, particularly in large systems-development programs.

Why Democratic Practices are Needed

The power of the democratic form of management derives from a few basic principles. The two most significant for this discussion concern the wisdom of groups and ownership. First, the combined wisdom of groups is vastly better than that of any individual for making complex judgments and producing accurate estimates. While basic design concepts should be produced by one or at most a very few individuals, the major concerns for large development programs are with the myriad smaller-scale but complex decisions that must be made throughout the development process [Brooks 2000]. Examples of some system-wide technical decisions are the following:

- power levels and standards
- weight distribution among components
- memory-size allowances
- message standards and protocols
- error and recovery standards and controls
- sub-system functional allocation
- system and component performance targets

In large organizations, these and many other technical questions often become management concerns, but they are rarely best settled by the managers. The challenge is to somehow ensure that technical decisions are made by the most knowledgeable technical people and not by the managers. This, however, raises the problem I discussed in “Large-Scale Work—Part II: The Project” (page 147) where a design problem had become so politically charged that I, as a senior executive, was forced to make an important technical decision.

To make sure that technical decisions are made before some director or vice president has to produce an arbitrary solution, the developers should somehow be convinced to work out these decisions at a technical level. However, the individuals involved may be in different groups, they may not even know each other, and they might not have the close working relationships needed to informally resolve complex systemwide technical problems. Even worse, many of the more senior managers are often technical people who are dying for the chance to make a technical decision.

So the first reason for a democratic, or at least a local, decision-making process is so that the technical decisions are made by the technical people who are best informed about the subject. When properly led, a small group of the most knowledgeable people working together will invariably arrive at a better conclusion than any senior manager could possibly produce. Of course, with larger projects, these democratic principles must be coupled with some communications and coordination process to get the right people to work together to make the decisions.

Importance of Ownership and Shared Vision

The second reason that democracies are so powerful concerns ownership and shared vision. To appreciate why ownership is so important, ask yourself when was the last time you washed a rental car. It is not that you value a clean car any more when you own it, but that it should not be your responsibility to wash a rental car. It belongs to someone else, and someone else should wash it.

When people think that they own even a small part of their country, or business, or project, they behave differently. One of the great tragedies of modern democracies is that many citizens do not have a sense of ownership. This problem was made evident to me some years ago when I got home from work late on a cold, rainy election night and didn’t want to go back out. Finally, I decided that I had to vote. I arrived at the polling place just as officials were locking the doors. They opened them just for me. I cast the last ballot in town.

At the time, a small group was trying to defeat the local school budget. The group believed that the town was spending too much money on education. I had seven children in the schools and knew that the budget was already very tight. The next day, everyone learned that the vote on the school budget was an exact tie. Had I not voted, it would have been defeated, and the entire school system would have had to go on an austerity budget. When the election was rerun, almost everybody voted, and the budget passed by a wide margin.

A shared vision is needed to ensure that the combined energy and motivation of the development group is coherently focused. When a group has a common vision, the members’ energies produce a powerful combined impact.

In a big country, business, or project, it is hard to believe that we can each make a difference, but we can. The key is not to wait for some special authority, but to act like an owner. If something is wrong, take action, and do what it takes to get it fixed.

Problem Ownership on Large Projects

On large projects, problems come up all the time—some function doesn't do what was expected, a test facility isn't available in time, an approval is delayed, or a requirement is not clear—and most of them are readily solved with little effect. However, almost every one of these problems requires changes to plans, and a few of these changes can significantly add to the project's workload, particularly if they are not handled effectively.

For any project to be successful, every issue must be handled promptly and properly. Tasks must be redefined, resources redirected, and plans updated and adjusted. Sometimes management must be informed, and occasionally, commitments must be renegotiated. On a large project, who does this? Since every one of these problems and changes is unexpected, they weren't assigned to anyone, and they aren't anyone's job. That is, nobody but the project manager owns them. And what is worse, the project manager usually doesn't know anything about most of these problems and won't even find out about them until they become serious. At some point, each problem will have to be solved for a developer to do his or her assigned job. Then management must get involved, and only then will the problem be solved.

On most projects, things aren't really quite this bad. Developers are responsible people, and when they see problems, they don't just ignore them; they tell somebody. Unfortunately, however, that somebody may not recognize the importance of the problem, may think somebody else is handling it, or may be too busy to do anything immediately and then forget about it. While the developer may think that everything required has been done, a true owner would pursue the matter until the problem was resolved.

Unless every single member of the project team feels responsible for the success of the project and acts like a true owner, some problems will be ignored or forgotten until it is too late to fix them without major project disruption. Successful projects require an environment in which somebody can be counted on to pick up every single problem and insist that it be addressed. This may require the developer to go personally to several levels of management before being comfortable that the problem is being properly addressed or can be safely deferred. That is ownership.

Without a broad and general feeling of ownership, every big project is a likely disaster. With ownership, most problems can be recognized and addressed early enough to be fixed without major program impact.

Why Lack of Ownership Is a Problem

Since the ownership problem seems so obvious and the solution so simple, you might wonder why there is a problem at all. Why don't people just act like owners? Consider the case of litter. When you see litter in public, do you pick it up? Few people do, but in a democracy, we are all part owners of every public space.

There are two reasons that development people do not act like owners. First, the management system looks autocratic. It is not that managers are tyrants, but that they sit at the apex of large hierarchical structures and issue orders and directions. Since this is exactly what tyrants do, the troops consider their more senior managers to be tyrants. The one thing that most people suspect about tyrants is that they shoot the bearers of bad news. Since none of us wants to get shot, and since we regularly read about the problems of whistle-blowers and outspoken malcontents, most of us are reluctant to speak out.

The second ownership problem concerns the natural reluctance of developers to overstep their assigned responsibilities. They have a lot of work to do, and they are uneasy that, if they spend time on anything that is not assigned, they may be criticized for doing something that is "not their job." So, while ownership is a great concept, it must be developed and nurtured. It will not happen by itself.

In the next column, I discuss the various ways to develop a feeling of project ownership for both teams and the team members. Future columns will then discuss self-directed teams, managing self-directed teams, and scaling up.

Acknowledgments

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of David Carrington, Julia Mullaney, Harry Levinson, and Jim Smith.

In Closing, an Invitation to Readers

In this particular series of columns, I discuss some of the development issues related to large-scale projects. Since this is an enormous subject, I cannot hope to be comprehensive, but I do want to address the issues that you feel most strongly about. So, if there are aspects of this subject that you feel are particularly important and would like covered, please drop me a note with your comments, questions, or suggestions. Better yet, include a war story or brief anecdote that illustrates your ideas. I will read your notes and consider them when planning future columns.

Thanks for your attention and please stay tuned in.

Watts S. Humphrey
watts@sei.cmu.edu

Reference

[Brooks 2000]
Frederick P. Brooks, Touring Award Lecture, SIGGRAPH, 2000.

31 Large-Scale Work—Part V: Building Team Ownership

2006 | 5

This is the fifth column in a series on large-scale development work. The first column briefly summarized the issues of large-scale development work. It then addressed the problems of organization size and how, as organizations grow and mature, they necessarily become less efficient. The second column dealt with project issues and particularly with the management decision process. Often, the most serious problems in large projects concern the way decisions are made. In the third column, I focused on the people and teams that make up the project staff and the issues involved in maximizing their performance. The fourth column addressed the problems of applying democratic management principles in large organizations. As demonstrated in politics, autocratic management is not an effective way to run a large and complex modern country. Autocratic management has also proved troublesome for large and complex organizations and projects. In this column, I discuss why so many organizations appear to be autocratic and why project ownership is important.

Why Are Organizations Autocratic?

Most organizations aren't really autocratic, they just look that way. The problem is not that the managers are truly autocrats, but that their people act as if they were. For example, during my years at IBM, I ran a development laboratory of several thousand people. I made a practice of walking through the lab several times a week and chatting with folks. I soon got to know the crew in the model shop pretty well. Two of the development engineers had starting coming in late and hanging their coats on the model shop coat rack so nobody would know they were late. At the end of the day, they left right on time. The entire shop was in an uproar.

One day, the shop elected a spokesman, and he came to see me. He was obviously very upset, and he told me the entire story. In my staff meeting later that day, I told this story to the senior managers. I concluded by saying that this was unacceptable behavior, and it had to stop. Thereafter, I was known as an autocrat who insisted on punctuality. That, of course, was not the case, but there was nothing I could do about it.

While a great deal could be said about the subject of image and how to develop or improve one, only three points are pertinent to this discussion. First, managers are people, and they occasionally get annoyed and take action without thinking about how it might affect their image. Second, technical managers have so much to do that they simply can't take to time to explain everything that must be done. They just have to do what they think is right and assume that their people will trust them and follow their guidance. The third part of the autocratic image problem concerns positive actions, and this is where many managers fail. Managers must provide the guidance and support their teams need to develop a feeling of ownership among all of their members.

How Do You Develop Owners?

In my lab, the machinists clearly felt and acted like owners. They had all been with the company for many years and had large parts of their savings invested in IBM stock. They had an ethic of getting to work early and working until they finished that day's jobs, even if it meant staying late. But at least two of the development engineers did not have this feeling. So how do you overcome this kind of problem?

I recently heard a story that described what a now-retired vice president did to handle this problem. In this large military contractor's development laboratory, dates were often set by the customer, by some other team, or by executive fiat. The development teams couldn't do much but struggle with impossibly tight schedules. This VP understood that such schedules would often slip, but he expected his teams to do their best to meet these dates anyway.

When team leaders came to him with proposals to slip their dates, however, he would consider the proposals, but only with two caveats. First, these team leaders had to provide an understandable rationale for the slip. Second, they had to have a recovery plan that they were certain to meet. In effect, he said: "The first slip's on me, but the second one's on you." This made the team leaders owners of their recovery plans and schedules. As far as these team leaders were concerned, this executive was no autocrat.

So making team leaders into owners would seem to be relatively straightforward: management just has to let them set their own dates. While this may help, it doesn't really address the problem of making the developers into owners. To do that, management must have the entire team, not just the team leader, set the dates.

Negotiating with Management

To make developers feel like owners, management must really make them owners. This can be done only by having all the members of each development team participate in making its own plan. Then they all must participate in negotiating that plan with management. To do this, however, the teams must know how to make reasonably accurate plans, and they must be guided through this planning process by someone who can coach them on how to make accurate plans and on how to negotiate plans with management.

The obvious next question is: "How much evidence does it take to convince management to agree to the team's plan instead of the one that management originally wanted?" Managers have commitments too, and sometimes these commitments are very hard to change. So teams can expect their managers to be hard to convince. On the other hand, managers are frequently seasoned professionals, and many of them have also been developers. They have generally been around long enough to understand the common problems that development teams have with schedules.

So the key is for teams to do more than just complain that this is a tough schedule and that they don't see how to meet it. They have to make very detailed and thorough plans. And, in doing so, they must work with the entire team and do their very best to produce a plan that meets management's date. Then, if they can't, they will know why and be able to explain the reasons to management.

While this may sound too easy, it turns out to be surprisingly effective. I have worked with hundreds of teams while at IBM and the SEI, and I know of no case where a team has followed this advice and not succeeded in convincing management that it had a better and more realistic plan for doing the job. Even when the team's plan took twice as long as management wanted or took twice as many people as originally planned, management has still bought the team's recommendation or a negotiated variant. The results of this strategy have also turned out to be good business: when teams produce their own plans, their typical schedule errors drop from 50% or more to less than 10% [Davis 2003].

Self-Directed Teams

While it can take a lot of work to make a really detailed and accurate plan, it is not really that hard to do. In fact, the Team Software Process (TSP) provides an entire process framework and a set of management and team-member courses that have proved successful at doing just this [Humphrey 02, 05, 06a, 06b]. TSP also addresses more than just the schedule issues that most often lead to autocratic behavior. It shows teams how to own and manage all aspects of their work. With TSP, teams select their own strategies, define their own processes, and make their own plans. They then manage their work to these plans and keep management posted on their progress.

For large-scale work, however, the problem is not just getting a team or two to make plans and to believe that they own these plans, it is building a feeling of ownership across an entire large-scale program. While the first part of doing this must be making plans and establishing ownership at the team level, that is only one step. How do you then extend this individual team-based ownership strategy to cover an entire large-scale program? This is the scale-up problem that I will address in the next column.

Acknowledgments

First, I want to thank Phil Gould for his e-mail describing the story of the VP of a large defense contractor. It is a marvelous example of how executives, by being sensitive to the ownership issue, can make an enormous difference in how their teams feel about their work and their executives. Also, thanks to Bob Schaefer for his e-mail about convincing management to agree to new plans.

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of David Carrington, Harry Levinson, and Bob Schaefer.

In Closing, an Invitation to Readers

In this particular series of columns, I discuss some of the development issues related to large-scale projects. Since this is an enormous subject, I cannot hope to be comprehensive, but I do want to address the issues that you feel most strongly about. So, if there are aspects of this subject that you feel are particularly important and would like covered, please drop me a note with your comments, questions, or suggestions. Better yet, include a war story or brief anecdote that illustrates your ideas. I will read your notes and consider them when planning future columns.

Thanks for your attention and please stay tuned in.

Watts S. Humphrey
watts@sei.cmu.edu

References

[Davis 2003]

Davis, Noopur, & Mullaney, Julia. *Team Software Process (TSP) in Practice* (CMU/SEI-2003-TR-014). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/library/abstracts/reports/03tr014.cfm

[Humphrey 2002]

Humphrey, Watts S. *Winning With Software: An Executive Strategy*. Boston, MA: Addison-Wesley, 2002. http://www.sei.cmu.edu/library/abstracts/books/0201776391.cfm

[Humphrey 2005]

Humphrey, Watts S. *PSP: A Self-Improvement Process for Software Engineers*. Boston, MA: Addison-Wesley, 2005. http://www.sei.cmu.edu/library/abstracts/books/0321305493.cfm

[Humphrey 2006a]

Humphrey, Watts S. *TSP: Leading a Development Team*. Boston, MA: Addison-Wesley, 2006. http://www.sei.cmu.edu/library/abstracts/books/0321349628.cfm

[Humphrey 2006b]

Humphrey, Watts S. *TSP: Coaching Development Teams*. Boston, MA: Addison-Wesley, 2006. http://www.sei.cmu.edu/library/abstracts/books/201731134.cfm

32 Large-Scale Work—Part VI: The Process

2007 | 1

This is the sixth column in a series on large-scale development work. The prior columns covered several important preparatory topics, but with this column the focus switches to process issues and the methods for actually doing the work. Part I introduced the structural problems of the massive organizations that typically do large-scale work. Part II addressed the management decision process, why this process is critical for large-scale work, and how to fix it if it is broken. Part III discussed how to get things done in a bureaucratic organization, and Part IV addressed the need for teams to take charge of their own work. Part V then described self-directed teams and how their planning and self-management skills help them to consistently produce superior project results.

The final columns in this series deal with actually doing the work. Once you have a suitable organization structure, management style, and work environment, two challenges remain: how to select the right process and how to get people to follow that process in doing the job. This column addresses the process question.

The Process Problem

The current situation with large-scale system-development processes can be summarized as follows:

- Large and complex computer-based systems are now critical to the economic and military welfare of the United States and to much of the developed world.
- With current methods, development of these systems is typically late and over cost and results in poor-quality products.
- Since future development programs will be far more challenging, the methods of the past will produce even worse results and often will not deliver any products at all.
- To address this problem properly, organizations must adopt sound processes that can be scaled up to the larger development challenges of the future.

Selecting a Process

Large and complex computer-based systems are now becoming widely used in the United States and in much of the developed world. If history is any guide, developing these systems with the methods of the past will almost certainly yield unsatisfactory results. These projects have almost always failed because of project-management problems. While the solutions to these problems are known and have proven to be highly effective, they are not yet widely practiced. Therefore, to successfully develop the even larger and more challenging systems of the future, we must start to use processes that address these historical problems. To do this, the processes must meet five requirements.

1. control development cost and schedule predictably
2. handle changing needs responsively
3. minimize the development schedule

4. be scalable
5. produce quality products predictably

Cost and Schedule

To control cost and schedule predictably, organizations must use processes that do five things:

1. Have the people who will do the work estimate and plan that work.
2. Track the progress of the work precisely and regularly.
3. When progress falls behind plan, promptly identify and resolve the causes.
4. When the requirements change, re-estimate and revise the entire plan promptly.
5. Anticipate and manage risks.

Changing Needs

To handle changing needs responsively, it is essential to do the following:

1. Examine every proposed change to understand its implications for the development plan.
2. Pay particular attention to each change's impact on completed work, including requirements, design, implementation, verification, and test.
3. Estimate all of the cost and schedule consequences of making the needed changes.
4. If the cost and schedule implications are significant or if they exceed the currently approved plan, get management approval before proceeding.

Minimize the Development Schedule

The proven methods to minimize the development schedule are straightforward but not always easy: increase the project staff, minimize the amount of rework, and reduce the amount of work. Rarely are there questions about how to increase project staff or reduce the amount of work; the problem is in actually implementing the obvious solutions. Minimizing rework, however, is a quality problem that is more challenging to solve. Here again, there are known and proven methods, and they rest on five principles:

1. It costs more to build and fix a defective product than it would have cost to build it properly the first time.
2. It costs more to fix a defective product after it has been delivered to users than it would have cost to fix it before delivery.
3. It costs more to fix a product in the later testing stages than in the earlier design and development stages.
4. It costs less to fix product requirements and specification errors in the earlier requirements and specification stages than in the later design and implementation stages.
5. It is least expensive to prevent defects entirely.

A Scalable Process

The fourth requirement for a process to be suitable for large-scale system-development work is that it be scalable. To be scalable, a process must meet the following three criteria:

1. It must use robust and precise methods at all levels, especially at the working systems-engineer and software-development level.
2. Technical and project decisions must be based on and give greatest weight to the knowledge and judgment of the development-level professionals.
3. The process must consistently use data that are derived from accurate, precise, and auditable process and product measurements.

These three points are not generally as widely understood as the cost and schedule problems and call for further elaboration.

Method Scalability—A truly fundamental problem in developing the large-scale systems of the future is that commonly used development methods are nearing their scalability limits and will likely be incapable of handling the much larger challenges of the future. As system scale increases, new methods are generally needed, but the smaller scale methods used in building the supporting system foundations must also be suitable. For software, the principal concern is with the design and quality-management methods commonly used. When developers design and implement the modules and components of massive systems, they typically use the same methods they used with small stand-alone programs. These methods are often little different from the methods they used to write the practice programs they developed when first learning to program. That would be like using the same practices to build the foundation for a 100-story building that you used in building a one-story structure. However, in software development, we do not just scale up by 100 times, we often scale up by 1,000 or 10,000 times, and we typically continue to use the identical development methods, regardless of the job’s size.

Management Scalability—Management scalability concerns the estimating, planning, job assignment, tracking, reporting, and control of development work. The current common practice is for managers to make the plans, allocate the work, report progress, and provide project control. However, the suitability of project planning, work allocation, tracking, and control decisions depends on the level of available project information. Furthermore, the level of available project information is inversely proportional to management level. It is therefore clear that this common management-centered planning, allocation, tracking, and control process will not likely work very well when scaled up to support massive systems-development programs.

Measurement Scalability—Why is it that financial accounting methods work for very small organizations and are equally effective for very large corporations? The fundamental reason is that the financial community uses precisely defined methods and auditable data. These methods also assume that the data can be in error, and they include consistent auditing and checking practices to ensure that the data at every organizational level are both accurate and precise. Few of the methods commonly practiced in the development of software-intensive systems use data of any kind, and those that do typically rely on data gathered by other groups, such as accounting, testing, and field support. To have a scalable process, all of the management, technical, and quality practices used in that process must use data that are derived from complete, accurate, precise, and auditable process and product measurements.

Predictable Quality

The fifth and final requirement for a process to be suitable for large-scale work is that it be able to produce quality products predictably. This is one of the greatest challenges faced by the software community today. The problem is that current practices do not acknowledge that no testing program can identify more than a small percentage of the defects in a large and complex product, and that the larger and more complex the product, the smaller this percentage will be. Further, this means that to get a high-quality product out of testing, one must put a high-quality product into testing. The requirements for a process that will do this are as follows:

- The process must include a family of early defect-prevention and defect-detection activities.
- The process must define and use measures that verify *process* quality during process enactment, and these measures must be applied at every management and development level. This will ensure that all or almost all product defects are found and fixed before testing begins.
- The process must also include measures that verify *product* quality, both before and after testing.
- The process must ensure that quality plans are produced and reviewed regularly, and that deviations from these plans are detected and corrected promptly during process enactment.

Today's commonly used software-development processes do not incorporate quality measurement and analysis. This is a crucial failing since, with even rudimentary measures, the solution to the software-quality problem would have been obvious, and sounder and more efficient software-quality practices would have long since been adopted. The simple fact is that without measurements, no serious quality program can be effective. While developers can make rudimentary quality improvements without measures, to achieve the defect levels required in modern complex systems, we must strive for defect levels of a few parts per million. Such levels are not achievable without complete, consistent, precise, and statistically based quality measurement and analysis.

Conclusions

Development processes must meet the following requirements to successfully produce the large, complex, and critical systems of the future:

- control development cost and schedule predictably
- handle changing needs responsively
- minimize the development schedule
- be scalable
- produce quality products predictably

Finally, the most critical point of all is that a process is of no value if people do not use it. This is the topic for the remainder of this series on large-scale work. For further information about how to improve large-scale systems-development processes, see my SEI technical report on the subject, *Systems of Systems: Scaling Up the Development Process* [Humphrey 2006].

Acknowledgments

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Dan Burton, Bob Cannon, David Carrington, Marsha Pomeroy Huff, and Bill Nichols.

In Closing, an Invitation to Readers

In this series of columns, I discuss some of the development issues related to large-scale projects. Since this is a complex subject, I cannot hope to be comprehensive, but I do want to address the issues that interest you most. So, if there are aspects of this subject that you think are particularly important and would like to see covered, please drop me a note with your comments, questions, or suggestions. Better yet, include a brief anecdote that illustrates your ideas. I will read your notes and consider them when planning future columns.

Thanks for your attention and please stay tuned in.

Watts S. Humphrey
watts@sei.cmu.edu

Reference

[Humphrey 2006]
Humphrey, Watts S. *Systems of Systems: Scaling Up the Development Process* (CMU/SEI-2006-TR-017). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2006.

33 Large-Scale Work—Part VII: Process Discipline

2007 | 2

This is the seventh and final column in a series on large-scale development work. Part I introduced the structural problems of the massive organizations that typically do large-scale work. Part II addressed the management decision process, why this process is critical for large-scale work, and how to fix it if it is broken. Part III discussed how to get things done in a bureaucratic organization, and Part IV addressed the need for teams to take charge of their own work. Part V then described self-directed teams and how their planning and self-management skills help them to consistently produce superior results. Part VI discussed the requirements a process must satisfy to consistently and predictably produce high-quality large-scale systems. This column discusses how to get the developers to follow their selected processes in doing their development work. As we will see, this challenge is not trivial, but it is also not hopeless.

Process Requirements

In Part VI, we discussed the requirements for a process that will predictably produce high-quality large-scale systems. In summary, the five basic requirements are to

1. control development costs and schedules predictably
2. handle changing project needs responsively
3. minimize the development schedule
4. be scalable
5. produce quality products consistently

While the actions required to accomplish all of this are not particularly complex, they do require that all of the systems, software, and hardware developers and their teams consistently follow certain essential practices. The five required practices are to

1. estimate, plan, and track personal and team work
2. estimate the impact of every change and negotiate needed schedule or resource adjustments before implementing the change
3. consistently use the team's or project's selected methods for every step of the work
4. plan, measure, and manage the quality of every part of the process
5. take immediate corrective action whenever the quality of any product or product element fails to meet team or project standards

For people to work in this way, they must have the proper skills, be highly motivated, and have the leadership and support to consistently do superior work.

Skill Requirements

The skills required fall into three categories: technical, project, and quality.

Technical Skills—Although the required technical skills are by far the most complex, difficult, and time-consuming to learn, they are not usually the source of most project problems. Technical topics are the principal focus of current academic curricula, and these are the topics the developers find most interesting. Yet even though insufficient technical skills are not usually the cause of project problems, when such skills are insufficient, projects invariably have problems. In short, suitable technical skills are essential, and without them, even the most motivated and best-led and supported teams will not likely succeed, at least not on predictable schedules.

Project Skills—Project-related skills principally concern estimating, planning, tracking, and configuration management. While these skills are relatively easy to learn and can be taught in only a few days, without them, developers and their teams cannot do predictable work. The main reason that such skills are not required on most projects is lack of understanding by the developers and their management. Technical professionals are not taught self-management skills during their educations. While this self-management problem has been particularly serious for software, it is also a problem for every development discipline.

This is an unfortunate failure because truly superior work is invariably done by people who manage themselves. Unfortunately, if technical professionals do not know how to manage themselves, their managers have to manage them. Then the managers make the plans, define the commitments, allocate the work assignments, and monitor work status and performance. If this does not sound like a fun way to work, that is because it isn't. The only way out of this trap is for developers to start planning and managing their own work.

Quality Skills—While quality problems are not new to the systems-development business, the character of these problems has changed. Early systems had thousands of largely identical parts, and all of these parts were interconnected by wires. The quality challenge at the time was twofold: to find parts of high enough quality and with sufficiently long expected lifetimes to get the systems to work. Then, the priority was to ensure the quality of all the connections among these parts. A useful early test was to literally shake these systems and, if there were intermittent errors, look for and fix the bad connections.

With integrated circuits, these early quality problems were largely solved. Now, the problem is with the multiplicity of part types. We used to have hundreds to thousands of copies of a relatively few part types, but now we also have thousands of different designs. Of course, with thousands of anything, you potentially have quality problems. These problems were first apparent with software, where we never had standardized parts and every module was unique. Large software-intensive systems have always had thousands of parts, and with really large systems, we now even have millions of different designs that all must work correctly.

With software, quality problems have been serious, but they have always been simpler and easier to fix than hardware quality problems. The reason is that, because the costs and time delays associated with fixing a defective software component were much less severe than with fixing defective hardware, it has been practical to fix software components during testing. With hardware, it is impossible to fix defective chips during testing. As a result, hardware quality has traditionally been viewed as a manufacturing problem and not something that the developers needed to worry about.

With embedded software and with software logic now widely used in hardware design, software quality problems are becoming pervasive in the hardware world. They are also becoming as time-consuming and expensive to fix as traditional hardware problems. In fact, nearly a decade ago, an auto company executive told me that the company had started to make hardware changes to avoid changing the software. So we must all learn to adopt sound quality-management skills. The hardware designers must learn from the manufacturing community, and the software engineers must participate in this learning process.

The skills required for quality management are relatively simple, but they are not easy. All that is required is to use a precisely defined personal process, to track and record every defect, and to use these defect data to modify the process both to prevent defects and to find and fix almost all of the defects before the start of testing. The manufacturing community has defined and refined the required practices. Now the development community must begin to adopt these practices. While this is partly a training problem, the training is relatively simple and can be completed in a few weeks. The most difficult issues are motivation, leadership, and support.

Motivational Requirements

There is only one motivational requirement: motivating the systems, hardware, and software developers to adopt and consistently use sound quality methods. To do this, these developers must be trained in quality-management skills, and they must work on teams in which all the members follow these same practices. However, because gathering and using data are important parts of quality management, and since it is essential that these data be accurate and complete, developers must be interested in their data and motivated to gather and use these data to manage their personal work. If they are not, the data will almost certainly be incomplete and imprecise. Finally, and most important, the entire management team must both motivate the developers to follow their team-defined quality practices and refrain from using any of the resulting data in any way that threatens the team members. The need is for the kind of trusting environment that makes creative work possible. While a trusting environment is important for all engineering programs, it is essential for large-scale work. Without trust, there is only limited communication, and without communication, large programs cannot be managed.

Leadership Requirements

Leadership is the key, but leadership is different from managing. Napoleon, when asked how he made his army cross the Alps into Italy, said: “One does not *make* a French army cross the Alps; one *leads* it across” [Humphrey 1997]. Leadership requires vision, an ability to define compelling goals, and the foresight to adopt improved practices and methods before they are widely adopted and before the need to use them has become obvious. Leadership involves setting goals, defining directions, and exciting the troops.

In leading large-scale operations, all of the traditional leadership practices are essential, but they are not sufficient. With large engineering programs, the leadership challenge is to build a cooperative management culture that enables rational fact-based decision-making and minimizes political infighting. In one example, the manager of one part of a large program encountered an unanticipated problem and needed additional resources. When he explained this to the leadership team, we knew that we could not expect help from anyone else; we would have to fix the problem ourselves. The managers then asked me to leave so they could try to work out a solution among themselves. When I returned, they had solved the problem. Each manager had looked in his own group to identify any lower priority work and had either cut or delayed it. With the right kind of culture, managers know that they will get help when they need it and are generally willing to help others when they can.

Leadership is about more than just setting direction, taking risks, and motivating the troops. It involves building a trusting and cooperative culture. It must also recognize and reward superior work. This requires that the developers be properly trained, encouraged to manage their own personal work, have the right kind of support, and be properly recognized and rewarded. This, of course, means that the leaders must know what superior work looks like, look for and identify such work, and ensure that it is properly rewarded.

Support Requirements

The final need is for support. In addition to the support provided by proper training and effective leadership, there is the need for coaching support. In sports and the performing arts, we all recognize the need for coaches. In fact, when a team is doing badly, it is always the coach who gets blamed. The coach, conductor, or director is the one who guides, motivates, and challenges the team members. However, we in the development community have been slow to recognize the need for coaching support.

Regardless of the field, whenever it is important for people to do consistently high-quality work, teams need coaching support. It is hard to do complex and creative work, and it is doubly hard to do it with extreme care and precision. If no one notices or cares about what we do or how we do it, it is almost impossible to maintain a high level of personal performance. That is when coaching is truly essential: to provide the support, encouragement, and guidance required to maintain motivation and dedication. Without motivation and dedication, there cannot be superior performance.

Conclusions

This concludes this series of columns on large-scale work. Large-scale engineering work involves many challenges. It requires properly designed and structured organizations, sound and minimally bureaucratic decision-making processes at every level, balanced and participative management systems, well-designed and structured development processes, and the leadership and coaching support required for consistently superior work. The methods and practices required to do all of this are well known and available. The principal challenge is to do it.

Acknowledgments

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Bob Cannon, Julia Mullaney, Bill Peterson, and Alan Willett.

In Closing, an Invitation to Readers

In this particular series of columns, I have discussed some of the development issues related to large-scale projects. As I now consider the topics to address in subsequent columns, I would appreciate any of your comments and suggestions. If there are topics that you feel are particularly important and would like to see covered, please drop me a note with your comments, questions, or suggestions. Better yet, include a story or brief anecdote to illustrate your ideas. I will read your notes and consider them when planning future columns.

Thanks for your attention and please stay tuned in.

Watts S. Humphrey
watts@sei.cmu.edu

References

[Humphrey 1997]

Humphrey, Watts S. *Managing Technical People—Innovation, Teamwork, and the Software Process*. Reading, MA: Addison-Wesley, 1997.
http://www.sei.cmu.edu/library/abstracts/books/201545977.cfm

[Humphrey 2006]

Humphrey, Watts S. *TSP: Leading a Development Team*. Reading, MA: Addison-Wesley, 2006.
http://www.sei.cmu.edu/library/abstracts/books/0321349628.cfm

34 Being Your Own Boss—Part I: The Ideal Job

2007 | 3

This is the first of a series of columns on our work as developers and how we can have truly satisfying jobs. This first column in the series discusses ideal jobs, what developers like about their work, and what they find annoying and unpleasant. It then describes how the developers' and managers' views on project success differ and what this means for both developers and their organizations. In subsequent columns, I will discuss how to turn almost any job into an ideal job, some of the issues you will face in trying to do so, and the level of support you will need from your managers and peers to be successful. While there are a few basic conditions that must be satisfied before you can expect to transform any job into this ideal, it is surprising how many jobs can be rewarding when you approach them in the right way.

Job Satisfaction

In discussing job satisfaction, we must first answer the question: "Satisfaction for whom?" One would expect the characteristics of a satisfying job to be different for the person doing the job and for the person for whom it is done. However, there is one condition that is common: the job must have been a success. This, of course, leads to the next question: "How do managers and developers view project success?"

This question has been researched by Kurt Linberg [Linberg 1999]. He studied the views of developers about project success for his PhD dissertation. In this study, he asked groups of developers to identify the most successful and least successful jobs on which they had worked. Then he examined the characteristics of the most successful and least successful projects.

Successful Projects

From the developers' perspectives, the most successful projects had three common characteristics:

- The project was a technical challenge.
- The final product worked in the way that it was supposed to work.
- The team was small and performed well.

Some typical developer views about these successful jobs were that

- the product was well designed and implemented
- it was well tested
- the team members enjoyed working with each other
- they didn't feel pressured by management

In addition, as long as the project manager protected the team, conflicts with other groups or with senior management didn't seem to bother the team members. They also thought that the work was innovative, that there was a high level of trust among the team members and with the project manager, and that the team was highly motivated.

In general, the developers attributed their high levels of motivation to five things:

1. a sense of making a contribution
2. an orderly and well-managed working environment
3. frequent celebrations of even small successes
4. positive feedback from both management and marketing
5. the autonomy to do the job in the way that they thought best

It didn't seem to matter how important the project was to the organization or how well the team worked with other groups. In fact, it didn't even matter whether the job was completed on time and within its budget.

Unsuccessful Projects

Conversely, the team members viewed the least successful projects as unrewarding, as having poorly defined requirements, and as having inconsistent or even nonexistent marketing support. Typically, these worst performing projects were perceived as

- not achievable from the outset
- under excessive management pressure
- requiring unreasonable levels of overtime
- technically frustrating
- having frequent conflict among team members
- operating in a chaotic environment

In summary, the developers' views of project success were largely independent of cost or schedule performance. The team members viewed a project as successful if it was a rewarding and enjoyable experience, even if it was late and over budget. In other words, a successful project was a personally satisfying project.

Management's Views of Project Success

Linberg also found that management's views of project success or failure almost entirely concerned cost, schedule, and quality performance. Table 34-1 summarizes his findings on the definition of various levels of success or failure for both completed and cancelled projects [Linberg 1999]. This study found that developers view some projects as successful even when management does not and vice versa. Having been a manager for many years and having seen how hard developers worked to meet their schedules, I thought this conclusion contradicted everything I have seen in more than 50 years of development experience. I have worked with hundreds of teams and supervised groups of thousands of developers who worked around the clock to meet their deadlines. Every team I have ever worked with has always made an extraordinary effort to meet cost and schedule commitments, and many of them were consistently successful in doing so.

A little reflection, however, shows that Linberg's conclusions do not conflict with my experience. I never asked the teams how they viewed their projects. They were paid to meet management's goals, and when they did, management was happy and everyone got paid. So, in summary, management views a project as highly successful if a team delivers a quality product on schedule and within its committed costs, regardless of how the team feels about the job. Conversely, development teams view projects as highly successful if they are professionally challenging and technically successful and if they provide a rewarding and personally satisfying working environment.

Table 34-1: Definition of Success or Failure for Completed and Cancelled Projects

Project Outcome	Completed Projects	Cancelled Projects
Failure	A product that causes customer discontent. Not meeting user expectations	Not learning anything that can be applied to the next project. Time and money wasted with no return.
Low Success	Below average cost, effort, and schedule performance. Barely meeting user expectations.	Learning can be minimally applied to future projects. Time and money wasted for limited return.
Success	Average cost, effort, and schedule performance.	Learning can be applied to future projects. Some artifacts can be used.
High Success	Better than average cost, effort, and schedule performance.	Substantial learning can be applied to future projects. Significant number of artifacts can be used.
Exceptional Success	Meeting or exceeding all user quality, cost, effort, and schedule expectations.	A cancelled project cannot be called exceptionally successful.

Common Management and Team Goals

While no rational managers would object if developers enjoyed their work, that is not one of the managers' highest priorities. Similarly, developers universally would like to deliver quality products on schedule, and they even strive to do so, but they do it because they know it is their job. What they really want, however, is to have a rewarding experience.

While these views don't necessarily conflict, they are not mutually supportive. This situation is what is called cognitive dissonance, where our views of reality differ from what we experience. In the case of a project, the developers know what success feels like. Many of them have experienced it on sports teams where a winning performance is an exhilarating experience, the coach congratulates the players, and the fans cheer. Everybody then helps in the celebration.

But development is different from competitive sports. Developers know that the managers' kind of success is what the organization wants, and that what developers value most is not a management priority. Sometimes the team wins and nobody seems to care, and other times, it feels as if the team loses but management cheers. Cognitive dissonance is demotivating and debilitating; it is not the stuff that builds winning teams or ideal jobs.

Congruent Management and Team Goals

This raises the next question: "What would happen if the managers' and developers' views of project success reinforced and supported each other?" One team's experiences illustrate what could happen. On their last project, members of this team had put in more than 70-hour weeks, worked under enormous pressure, and managed to deliver their product to test on time. Testing, however, took longer than anyone had expected, and the product was delivered late. It was delivered, however, and at the time I met with the team, it was being installed by the users. While this job wasn't a complete failure in Linberg’s terms, it certainly wasn’t a big success from either management’s or the team’s perspective.

On the next project, however, the team members had a realistic plan; they did high quality work; they had capable leadership; they had clear, agreed-upon goals; and they believed that the team owned the project. This was their job and they were going to make it a success both for the team and for the business. They not only delivered their product to test on time, but it was of such high quality that testing was finished early. They only worked 45-hour weeks and were home for dinner with their families every night. They said it was the best project they had ever worked on, and management was full of praise.

How did they do it? The answer is that they and their management made a real effort to make this project a success for both the developers and the managers. More on how to do this in the June column.

Acknowledgments

First, I want to thank Bob Schaefer for his note. He asked about ways to handle conflicts between managers and developers and, in thinking about the answer, I got the idea for this series of columns. Also, in writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Bob Cannon, Julia Mullaney, and Bill Peterson.

In Closing, an Invitation to Readers

In this particular series of columns, I have discussed some of the development issues related to large-scale projects. As I now consider the topics to address in subsequent columns, I would appreciate any of your comments and suggestions. If there are topics that you feel are particularly important and would like to see covered, please drop me a note with your comments, questions, or suggestions. Better yet, include a story or brief anecdote to illustrate your ideas. I will read your notes and consider them when planning future columns.

Thanks for your attention and please stay tuned in.

Watts S. Humphrey
watts@sei.cmu.edu

Reference

[Linberg 1999]

Linberg, Kurt R. “Software Developer Perceptions about Software Project Failure: a Case Study,” *The Journal of Systems and Software, 49* (1999) 177-192.

35 Being Your Own Boss—Part II: The Autocratic Manager

2007 | 5

This is the second in a series of columns on being your own boss. The prior column discussed ideal jobs, what developers like about their work, and what they don't like. It also described how managers' and developers' views differ about project success and what this means to developers, managers, and their organizations. In this column, I talk about autocratic managers, autocratic behavior and why it is common, and the consequences of autocratic behavior. In subsequent columns, I will discuss why an autocratic management style is not appropriate for modern development work, the alternatives to autocratic management styles, and how each developer can take charge of his or her personal working environment to turn a seemingly unpleasant job into an ideal one.

The Autocratic Boss

Autocratic bosses are common. Starting with the construction of the early pyramids in Egypt, the literature is full of examples. Bosses have historically been slave drivers, demons with whips, or guards with guns. Historically, workers may have been slaves or prisoners, but that is not true of typical working environments today. However, in development work, bosses typically have very substantial power, and workers do not. This power confers a level of authority that bosses can use in autocratic ways.

Even when the boss is charming and seemingly respectful and friendly, that boss would still be an autocrat if he or she behaved unilaterally. Autocrats do not really consider the feelings or views of their workers. While this may sound extreme, it really is not. The true test is whether the boss actually considers your needs and views in making decisions. Merely pretending to listen to your opinions does not change the facts. A typical autocrat in effect says, "Let's compromise and do it my way."

Why Is Autocratic Behavior Common?

Autocratic behavior is common because it can have substantial advantages for the autocrat. For example, autocratic decisions can be made quickly. Assuming that the autocrat is technically competent, this could be advantageous in dangerous situations like military campaigns or natural disasters. Autocratic behavior could also be effective when the autocrat-boss was better informed and more competent than the workers. This is often the case for simple and highly repetitive work. The autocrat can also get substantial personal rewards from unilateral action. Every time some autocratic command is promptly and successfully carried out, the autocrat's belief in his or her competence and infallibility is reinforced. This self-reinforcing characteristic of power led Lord Acton to say over 100 years ago that "power corrupts, and absolute power corrupts absolutely."

While this conclusion has been widely recognized and quoted, there is another similar conclusion that is not as well known: "Petty powers are most corrupting." Philip Zimbardo found this to be the case in simulated-prison experiments at Stanford University in the 1970s [Gladwell 2000]. Zimbardo enlisted volunteers to act like prisoners and guards in a simulated prison. They had built a facility just like a prison in the laboratory basement and actually locked up the volunteer "prisoners."

What Zimbardo found was that, after only a couple of days, the "guards" started to behave so abusively that one of the "prisoners" actually had a severe emotional breakdown. He had to stop the planned two-week experiment in only six days. He concluded that ordinary people will often act in authoritarian ways when they are given even menial jobs that have some minor but absolute powers.

Zimbardo's conclusion can be seen in the behavior of many bureaucrats: they often tend to use their limited powers in arbitrary and highly authoritarian ways. This is what makes some clerks, guards, or other support people insist on strictly interpreting the rules although that interpretation would make no logical sense in the specific case at hand. This kind of strict adherence to work rules makes organizations unresponsive and hard to work with. It can also be expensive and demotivating.

Why People Are Autocratic

There are four reasons for people to behave autocratically.

1. There is a crisis.
2. There is a power vacuum, and they feel they must take charge.
3. They act autocratically because that is the way such jobs have always been done.
4. They get emotional benefits from being autocratic.

The Crisis—In times of crisis, rapid decisions are often required, and a single authority is usually thought to be most effective. Most people understand and accept that this is the best way to behave in such cases.

The Power Vacuum—A power vacuum can occur in a crisis when the leader is either killed or otherwise unavailable and someone takes over. Such cases can arise in almost any situation, and it may be seen in combat when the commanding officer is killed, and a sergeant or even a private takes charge. While this situation need not lead to autocratic behavior, it generally does when nobody else seems willing to or able to make the needed decisions.

Force of Habit—While power vacuums can occur in development, the force-of-habit case is more typical. Development managers have generally managed in this way so pretty much everyone does. Furthermore, since it seems so natural and expected for the boss to make all of the decisions, force of habit tends to create power vacuums. Nobody but the designated boss feels able to make decisions.

Emotional Reinforcement—Emotional reinforcement presents an entirely different situation. When the person has clear authority and resists either suggestions or appeals to common sense, it is generally a good idea to keep quiet and do what you are told. If this person is your boss, it may not be clear whether he or she would accept suggestions or not. This is when you may have to conduct tentative tests to see how your suggestions are received. While I have only seen a couple of truly autocratic managers in 50+ years of development experience, they do exist, and they can be both unpleasant and threatening to work for, particularly if, like me, you like to make your own decisions.

The Consequences of Autocratic Behavior

While autocratic environments are not pleasant places to work, they can be reasonably efficient when the boss is both competent and fully capable of directing the work. Even in these cases, however, it has long been known that autocratic management styles demotivate the workers and produce less than optimum workplace performance.

One reason for this is explained by James Surowiecki in his book *The Wisdom of Crowds* [Surowiecki 2004]. He cites many examples of how groups typically make much better decisions than even expert individuals. This is particularly important for development groups and is the reason that autocratic behavior is ineffective and often even counterproductive. This is true regardless of how pleasant and friendly the autocrat is; the problem is unilateral behavior.

An Example

Since it is often hard to recognize that you are working in an autocratic environment, an example may help. In one case, a development team was starting a new project and management told the team leader that the job had to be completed in nine months. The team leader then produced an overall plan with key milestones for

- design-specification sign-off
- detailed design complete
- code complete
- functional testing
- system testing
- customer-acceptance testing

He then met with the entire development team and reviewed his proposed plan. Over the next couple of hours, he answered all of the developers' questions and made some minor additions and adjustments to the plan. In the end, even though none of the developers felt that the nine-month schedule could be met, the team agreed to the plan pretty much as the team leader had originally proposed it. This team leader then told management that his team now had a plan to deliver the finished product in the desired nine months.

The question is: "Is this autocratic behavior?" Most developers would say no. This is how most of their projects have always been run. They believe that they could have spoken up and made changes to the plan if they had wanted to. Furthermore, most of them also felt that the team leader knew more about planning than they did, and they may even have believed that he or she had produced a better plan than they could have. Finally, since they had to meet management's nine-month schedule and this plan did, they didn't have anything better to suggest. The team leader would also likely argue that he or she was not being autocratic. After all, there was a full team review of the plan and all of the team's suggestions and changes were incorporated.

While it is true that this style is nothing like that of the despot or slave driver of old, it still results in a unilateral plan that was produced with little or no team input. While minor changes were made, the plan had the original resources management allocated, it produced the product that they had specified, and it met management's suggested end date. Such plans commonly have three important characteristics.

1. They do not guide the developers in doing the job.
2. They do not have the full commitment of the team members.
3. They are rarely met.

The next question, of course is: "If this is autocratic management, how else could you produce a plan?" The answer is to truly involve the team in making its own plan. However, there are typically two objections to doing this.

1. It would take too long.
2. The developers wouldn't know how to make a plan.

Of course, if you don't mind getting inaccurate and largely useless plans, it doesn't make much difference how you produce them. However, if you want accurate and useful plans, it might make sense to consider alternatives. I address these points in subsequent columns.

Acknowledgments

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Dan Burton, David Carrington, Julia Mullaney, Bill Nichols, Bill Peterson, and Alan Willett.

In Closing, an Invitation to Readers

In these columns, I discuss development issues and how they impact the work of engineers and their organizations. However, I am most interested in addressing the issues that you feel are important. So, please drop me a note with your comments, questions, or suggestions. Better yet, include a story or brief anecdote to illustrate your ideas. I will read your notes and consider them when planning future columns.

Thanks for your attention and please stay tuned in.

Watts S. Humphrey
watts@sei.cmu.edu

References

[Gladwell 2000]

Malcolm Gladwell. *The Tipping Point: How Little Things Can Make a Big Difference*. New York: Little, Brown and Co., 2000.

[Surowiecki 2004]

Surowiecki, James. *The Wisdom of Crowds: Why the Many Are Smarter than the Few and How Collective Wisdom Shapes Business, Economies, Societies, and Nations.* Doubleday: New York, 2004.

36 Being Your Own Boss—Part III: Knowledge Work

2007 | 6

This is the third in a series of columns on being your own boss. The first two columns discussed ideal jobs and autocratic management. In this column, I discuss knowledge work—what it is, and why it is so important. I also discuss why the management of knowledge work is fundamentally different from the way management is typically performed today and the consequences of today's common management styles. While I will ultimately talk about what you and I, the knowledge workers, can do about the way we are managed, I am not yet ready to discuss that in this column.

The key point is that how we are managed is up to us. Once we have acquired the requisite knowledge and skill, we really will have the ability to manage our own projects, even if our organizations and our immediate managers do not know how to manage knowledge work. However, to be successful at doing this, we will have to show our managers how to manage us. Understanding how to do that, of course, is the key, and that is what I am talking about in this series of columns.

The Typical Team Plan

In part II, I mentioned the way development plans are typically made today. The manager sits down by himself or herself and lists half a dozen or so checkpoints, gives them more or less arbitrary dates that in aggregate meet management's desired delivery date, and then reviews this "plan" with the team. After the team has commented on the plan, and the manager has made a few minor changes, this becomes the official team plan. The team is then committed to delivering the product on the date specified in this plan. The fact that none of the developers think that this plan has a reasonable delivery date or that they can meet the schedule is beside the point; this is the "official team plan."

There are three things wrong with such plans. First, the team does not own the plan; it is the team leader's plan. The members are not committed to it, and they do not believe they can meet it. Second, plans like this do not guide the work, and they do not provide a realistic estimate of how long the work will take. Third, development work is knowledge work, and plans that are produced in this way are not suitable for guiding knowledge workers. In addressing these points, I start at the back and work forward.

In this column, I discuss knowledge work, what it is, why it is important, and how to manage it. In subsequent columns, I will address the characteristics of plans that can truly guide knowledge work and help developers accurately commit to how long their projects will really take. After this important preparatory material, there will still remain the problem of getting people to actually work in this way. This is also a topic for future columns. In case there is any suspense, however, and for those of you who don't know me and my work, all of the columns in this series are leading up to a discussion of how the Team Software Process (TSP) can guide you as you strive to become your own boss.

Knowledge Work: Definition

Knowledge work is work with minds instead of hands. While your hands may be involved, the true product of knowledge work is concepts, ideas, and designs and not the devices, machines, or things that may ultimately be produced from these knowledge products. The knowledge worker is most productive when he or she is creating, refining, recording, or elaborating ideas, concepts, and representations. This is creative work, and it requires a very special kind of management. Peter Drucker originated the concept of knowledge work, and he discussed it extensively in his final books and papers [Drucker 1999]. He describes what knowledge work is and how to prepare for it, and his books and papers are well worth reading.

Managing Knowledge Work

My objective, however, is to describe how knowledge workers can take control of their own work. This is not a subject that Drucker has addressed. However, Drucker did succinctly state the management principle for knowledge work, which is that managers can't manage knowledge work; the knowledge workers must manage it themselves. I add to this the point that, for knowledge workers to manage themselves, they must know how to manage. While all of this may sound very logical, there are some key questions. First, why must knowledge workers manage themselves? Second, why don't they manage themselves today? And third, what is wrong with having managers manage knowledge workers?

First Question: Why Must Knowledge Workers Manage Themselves?

First, the reason that knowledge workers must manage themselves is that nobody else can. We all think in our own way and often we don't think much about how we think; we just do it. While this is a natural and comfortable way to work when you can, most of us work for organizations, and we are paid to produce results on committed schedules. However, to meet our schedule commitments, we must work to achievable schedules.

Since the essence of management is using the available resources to produce defined and committed results, the manager must know a great deal about the work, the available resources, and how the work is to be done. This means that whoever manages knowledge work must be able to project how long the work will take. To do this, however, that manager must also understand something about the products to be produced, how the knowledge workers intend to produce them, and how long similar work has taken them in the past. Finally, this requires that these managers have some historical data on how long it took these knowledge workers to do similar work in the past.

The reason that nobody but the knowledge workers themselves can manage knowledge workers is that their work is highly individual. If you ask designers how they design, for example, and assuming that they have thought about this question, they will tell you that they leap from one abstraction level to another, and frequently dive from the highest level all the way down to code and back [Curtis 1999].

When you couple this dynamism with the fact that you can't watch people and tell how they are thinking, it is obvious that nobody but the knowledge workers themselves can know what they are doing, how they are doing it, what they have accomplished, or how long it took. Since you must know all of these things to manage any kind of sophisticated work, this means that nobody but the knowledge workers themselves can manage knowledge work.

Second Question: Why Don't Knowledge Workers Manage Themselves?

This gets us to the second question: "Why don't knowledge workers manage themselves today?" The principal reason is that few knowledge workers know how to manage themselves. For example, the things that software developers must be able to do to manage themselves are the four things I just listed above. They must know what they are doing, understand how they are doing it, be able to measure what they have accomplished, and gather data on how long it took. While knowing how to do all of this is not terribly difficult, it is not obvious. In fact, explaining how to do this is the principal reason that I wrote several of my books. The most relevant one for today's software professionals is *PSP: A Self-Improvement Process for Software Engineers* [Humphrey 2006].

The basic requirements for self management are pretty straightforward, and they only require that the knowledge workers be able to convince their management that they can manage their own work. Then, of course, they must actually manage it properly and do it so consistently and well that management will continue to trust them to manage themselves in the future. This, however, is the key: trust. If your managers do not trust you to manage your own work, they must manage you. They may not want to, but their continued success as managers depends on the results that you produce.

Since your manager's performance depends on your performance, and since the performance of software groups has historically been so poor, managers do not trust software professionals to manage themselves. To overcome this problem, all we have to do is to convince management that we can manage ourselves and then perform that self management so well that management will continue to trust us. Since software groups have been unable to do this in the past, they have not been trusted to manage themselves. This situation will continue until we software professionals do something about it. For more information on how to do this, read the rest of this series of columns.

Third Question: What's Wrong with Having Managers Manage Knowledge Workers?

This now gets us to the third question: "What is wrong with having managers manage knowledge workers?" Here the answer has two parts. The first is that even though the managers don't know how to manage the knowledge workers, that doesn't mean they won't try. Unfortunately, however, when they do try, they do a bad job. That is why we get vague, inaccurate, and highly misleading plans and why software projects so frequently miss their schedule commitments. Furthermore, since the managers can't produce precise or detailed plans, and since there is no way that they could know how long the various knowledge-working tasks would take, there is no way to measure and track progress against these plans.

Also, as noted above, when the managers make the knowledge-workers' plans, the knowledge workers do not own these plans, and they have no real commitment to them. So, in summary, the reasons that managers should not manage, or even try to manage, knowledge workers are the following.

First, the managers cannot possibly do a competent job of managing knowledge work.

Second, when managers try to manage knowledge work, the plans they produce are so inaccurate and misleading that neither the workers nor their managers can measure or track project progress, and the work is unpredictable and usually late.

Third, when knowledge workers work with such incompetent plans and plan management, they are unhappy and are most likely to do poor work.

That is why the managers should not manage knowledge work. However, the problem is that, if the knowledge workers do not manage themselves, the managers must.

In the next column, I will start to address the question of what software developers and other knowledge workers must to do get control over their own work. Basically, this is the issue of building trust and credibility with management. Unfortunately, credibility and trust are hard to build but very easy to lose. All you have to do is to blow one schedule. This, of course, leads us to a circular problem: the managers blame the developers for missing their schedules and the developers blame the managers for giving them schedules that they can't meet. Breaking this circle is not a trivial problem, which is why so few knowledge workers are actually able to manage their own work.

Acknowledgments

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Judi Brodman, David Carrington, and Bob Schaefer.

In Closing, an Invitation to Readers

In these columns, I discuss development issues and how they impact the work of engineers and their organizations. However, I am most interested in addressing the issues that you feel are important. So, please drop me a note with your comments, questions, or suggestions. Better yet, include a story or brief anecdote to illustrate your ideas. I will read your notes and consider them when planning future columns.

Thanks for your attention and please stay tuned in.

Watts S. Humphrey
watts@sei.cmu.edu

References

[Curtis 1999]
Curtis, Bill; Krasner, Herb; and Iscoe, Neil. "A Field Study of the Software Design Process for Large Systems," *Communications of the ACM, 31*, 11 (November 1988).

[Drucker 1999]
Drucker, Peter F. *Management Challenges for the 21st Century*, New York: Harper Collins, 1999.

[Humphrey 2006]

Humphrey, Watts S. *PSP: A Self-Improvement Process for Software Engineers*. Reading, MA.: Addison Wesley, 2006. http://www.sei.cmu.edu/library/abstracts/books/0321305493.cfm

37 Being Your Own Boss—Part IV: Being a Victim

2007 | 8

This is the fourth in this series of columns on being your own boss. In the first two columns, I discussed ideal jobs and autocratic management. Then, in part III, I talked about knowledge work and why only the knowledge workers can competently manage their own work. The key question remaining at the end of part III was: How can we take charge of our own work? That is the topic I cover in this and the final columns of this series.

Software Developers Cry a Lot

When I ask software developers about the issues they face and the problems that typically cause their projects to fail, I hear lots of complaints. There are many reasons why our projects fail, and we typically blame somebody else for the failures. Management gave us an impossible schedule, the customer changed the requirements, the organization has an impossible bureaucracy, or there are too many meetings and distractions. This is victim talk. Have you ever heard a first class surgeon, a top flight scientist, or a winning ball player talk like this?

Winners win; they don’t complain. It is the perpetual losers that complain about how unfair life is and how somebody else is always to blame for their failures. While it is true that software development is a challenging business and that we almost always face tight schedules and changing requirements, these problems can be managed. However, they can be managed only if you know how to manage them.

Taking Charge

You might wonder why more software professionals don’t manage themselves, and why essentially all software developers act like victims. These smart and capable people seem willing to spend their lives behaving like losers. The principal reason is that nobody ever showed them how to break out of their victim trap. The second and almost as important reason is that, even though self-management methods are not that difficult, they are not obvious.

The alternative to being a victim is to take charge. There are two parts to doing this. The first is the hard part: actually taking control of your own work. Then, once you know how to manage yourself, the second part is much easier: convincing management to let you manage yourself. The rest of this column discusses the first part, and in the next column, I describe how to convince management to let you control your own work.

Managing Yourself

There are five principal steps to managing yourself, but they are not necessarily followed in linear order. Since all of these steps are interdependent, you must treat the entire process as cyclic. The principal reason to follow a cyclic strategy is that you will almost certainly make mistakes the first time, and with a cyclic strategy, you can quickly identify and correct these errors. However, you will be surprised at how quickly you will learn and how, assuming that you have a little guidance, you will do a good job of managing yourself the very first time you try. If you stick with the self-management process, your performance will quickly improve. In brief, the five basic steps to self management are described in the following paragraphs.

Step One: Establishing Process Goals

It is important to be clear about your goals and to make sure that these goals are ones that you are willing to strive to meet. With a little reflection, the importance of goals is obvious: if you haven't decided where you want to go, you are unlikely to get there. Examples of process goals are "meeting commitments" or "building high-quality products." If you don't feel that goals like these are truly important, you will not likely strive to meet them. And then, of course, you will almost certainly not do so. The other part of establishing goals is to make sure that they are consistent with the problems you are trying to solve. That means when trying to demonstrate to management that you can manage yourself, your goals must relate to the self-management problem.

Step Two: Define Your Process Principles

It is also important to be clear about the principles you will follow when using the self-management process. For example, one key principle is that it is always faster and cheaper to do the job right the first time than it is to build a poor-quality product and spend a lot of time fixing it. Another way to state this principle is that high-quality software products are always cheaper to develop and test than poor-quality products. Since few software developers truly believe this principle, it may not be one that you are willing to follow. Once you actually do follow it, however, you will be surprised to find that it is indeed true.

Another important principle is that, to meet commitments, you must plan and track your own work. An extension of this principle is that, to make accurate plans, you must know how long similar work has taken you in the past. This principle also relates to the first principle because poor-quality work is inherently unpredictable. While few software professionals will find this principle obvious, most experienced developers know that unplanned software work is rarely if ever delivered on time. The reason is that the key to making commitments is to only make commitments you can likely meet. And, of course, the key to making commitments that you can likely meet is to accurately estimate how long the committed work will take.

While there are many possible principles, it is important to be clear about the principles you intend to follow when you define your work processes, practices, and plans. Then, of course, you must consistently follow these principles while doing the work.

Step Three: Define Your Process and Plan

To consistently do competent work, you must define and document the process and plan you will use to do the work. While doing this, start the process with a planning step and conclude with a postmortem. Also define the measures you will use to track your work as you do it. This is essential so that you can analyze your performance during the postmortem and then have the data you need to make progressively better estimates in the future. Since you will need this information for the very next cycle of the job, you must start gathering it right away.

The process you define is not some vague document to be put on the shelf; it is a working document to use in doing the work. That means that it must be an operational process. That is, it must be sufficiently detailed and precise to guide your work. Only then can you measure and estimate the work, track your process, and know if you are on schedule every day. Furthermore, the only way to consistently meet commitments is to work every day to recover from whatever problems you had the previous day. To do this, you must produce a plan with estimates and schedules for every step. For more information on defining operational processes, see my PSP book [Humphrey 1998].

The second part of step three is to actually produce a plan for the work. Once you have a little historical data and an operational process, the planning task is straightforward [Humphrey 1998]. If you don't have historical data, you will have to make some guesses. Usually, however, once you have used a defined and measured process to write even a few small programs, you can accurately tell how long such work has taken in the past. Then you can use these measures to guide the estimates for the next job. However, if you don't have any data and, since unmeasured recollections are generally inaccurate, you must be cautious about over-committing. Again, my PSP book can help you in doing this.

Step Four: Negotiate Your Commitments

In starting any job, use your plan to establish and negotiate your commitments. This, of course, is the bottom line. The key point to remember is that unless you make your commitments properly, you will rarely be able to meet them. That is why so many software projects get into trouble: few software professionals know how to make realistic commitments or to negotiate them with management. While negotiating commitments is not as hard as it might sound, it is not a trivial step. I will discuss it in more detail in the next column.

Step Five: Follow the Process and Plan

In doing the work, you must follow the process, gather the data, and use the data to plan and track this and all of your subsequent work. While this may sound simple, it is not easy to do, particularly the first time. That, however, is the reason that I developed the PSP process and course [Humphrey 1998]. While PSP takes a little effort to learn, it is really not difficult, particularly for competent programmers. A few people have completed PSP training on their own, but most find it is best to take a PSP course. The SEI offers such courses, but another and often more practical way to learn it is to convince your organization to have a qualified PSP instructor train you and your coworkers in PSP methods.

Conclusion

The purpose of this discussion is to explain why software professionals don't manage themselves and to convince them to do so. The simple reason that few developers do this today is that, while the methods are relatively straightforward, they are not trivial and they are not widely taught in a typical software engineering education. That is a shame, because understanding and following these principles will transform your life. Instead of being a victim, you can actually manage your own work. And that is the key to turning any job into an ideal job. The next column concludes this series on being your own boss with a discussion of how to negotiate with management.

Acknowledgments

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Eugene Miluk and Said Nurhan.

In Closing, an Invitation to Readers

In these columns, I discuss development issues and how they impact the work of engineers and their organizations. However, I am most interested in addressing the issues that you feel are important. So, please drop me a note with your comments, questions, or suggestions. Better yet, include a war story or brief anecdote to illustrate your ideas. I will read your notes and consider them when planning future columns.

Thanks for your attention and please stay tuned in.

Watts S. Humphrey
watts@sei.cmu.edu

Reference

[Humphrey 2006]

Watts S. Humphrey, *PSP: A Self-Improvement Process for Software Engineers*, Reading, MA.: Addison Wesley, 2006. http://www.sei.cmu.edu/library/abstracts/books/0321305493.cfm

38 Being Your Own Boss—Part V: Building Trust

2008 | 3

This fifth and final column on being your own boss describes how to take control of your own work. It covers the key issues you will face, the fears you must overcome, and the self-confidence and credibility you must build. While the challenge of doing all of this may seem daunting, and while it does take personal courage, it isn't really that difficult. The key is to be willing and able to take responsibility for your commitments. While you can take all of these steps by yourself, it helps to have teammates to work with you. Once you know how to manage yourself, however, you will not want to work any other way.

Starting a Job

To take charge of your own work, you must assert your position at the very beginning of every job. It is the only way that you will be able to consistently work on projects with realistic schedules and plans. Working in this way, however, requires the courage to take personal risks, a willingness to step out of the crowd, and the responsibility to own your own work.

Just about every one of us in the software business has been regularly told what to do and when to do it. This is the way our business has always been run. While there are many reasons for this state of affairs, the fundamental cause is that management does not trust us to establish our own plans or to set our own schedules. This is not surprising, however, for, with few exceptions, when software groups have been asked to make their own commitments, they have rarely met them.

So you start out with two strikes against you. In this negative environment, you must somehow convince your managers that they can trust you to do what you say. Then, of course, you must do what you promised. This is critical, because if you don't, it will be even harder to convince management the next time. To do all of this, however, you must overcome some fears and build your self confidence. Only then will you be able to convince your managers that they can trust you. The challenges are to overcome fears, to build self-confidence, and to establish trust.

Overcoming Fear

While there are many kinds of fears, the worst ones are fears of the unknown. The best way to deal with these is to consider rationally what is most likely to happen and to assess all of the realistic outcomes. For example, suppose management told you and your teammates that this new project must be done in three months. And suppose also that none of you believed there was the slightest chance that you could do the job in that time. What would happen if you told management that three months was not nearly enough time?

In response, management is likely to say one of two things. First, they could say: "Three months is critical to the business, so do your best." Since you can't refuse to try, all you can do is agree and, regardless of what you think, you will then be stuck with a three-month commitment for doing the job. The second thing management could say is: "How long do you think it will take?" Then you would probably have to answer: "While I don't know how long it will take, it will certainly be longer than three months.” Then you will again get management’s first answer and be stuck with the same three-month schedule.

Of course, instead of saying you don’t know how long the job will take, you could make a guess and say it will take five months. At this point, any manager with half a brain would say: “Prove it.” And again, unless you had a really convincing story, you would be stuck with the same three-month schedule. So this fear is really not an unknown at all. If you complain about the short schedule, there are very few possible outcomes, and they all end up exactly where you started: stuck with the three-month schedule. There is, however, another approach you could take and it concerns building your self-confidence.

Building Self-Confidence

Some people can speak with assurance about things they don’t know or understand, but that skill is more common to politicians than technical people. The fundamental problem with bluffing is that the person with the most resources or power usually wins. In development work, of course, that is the manager and not the developer. This means that if you want to debate the schedule with your management, you must know what you are talking about.

This suggests that you follow a different strategy. Now, when management says that the job must be done in three months, first make sure that you understand what they want you to do and then tell management that you need to make a plan before you can discuss the schedule. At this point, management really has only two choices: They could refuse to let you make a plan or they could agree and wait for you to come back. In all of my years in development work, I have never run into a manager who refused to let the developers make a plan. While the manager may be surprised at your request, he or she will invariably agree to let you make a plan, as long as you don’t take too long.

This means that you will almost certainly be faced with the need to make a plan. In doing so, however, you must try to make a plan that does the job on the requested schedule. Then, if you can’t, you will know why and be able to explain the problem to management. Better yet, you could propose several alternate plans, each of which meets some but not all of management’s goals. Also, of course, you must know how to make a realistic and convincing plan. I have written extensively on this subject, and if you are working by yourself, the best reference is *PSP: A Self-Improvement Process for Software Engineers* [Humphrey 2005]. If, however, you are part of a development team, you should still consult this PSP reference but also look at: *TSP: Coaching Development Teams* [Humphrey 2006].

Establishing Trust

Let’s assume that you are to do the job by yourself and that you have now made a plan. Also let’s assume that, in making this plan, you followed the methods described in the PSP book. Now your job is to convince management that you have made a realistic and aggressive plan and that you are willing to commit to a delivery schedule. Also, if history is any guide, you will have found that this job is far too big for you to complete in the desired three months, and that five months would be required. To make your plan presentation to management convincing, you must describe both how you made the plan and the plan itself. The key topics to cover in this meeting are the following:

- the goals of the project—in this case, to finish in three months
- the assumptions you made—presumably about the requirements and other key project characteristics
- your concept of what the product will look like
- your estimate of the product's size and how it compares to other previously developed products
- your estimate for the time required to develop the product and the data you used to make this estimate
- some alternate plans that show how adding resources or reducing product function could shorten the schedule
- the recommended plan and schedule
- the key project risks and recommended mitigation actions

While this is a lot of work, it is a lot easier than struggling for months with an unrealistic plan and schedule. Furthermore, with a little guidance and experience, you will find that it takes surprisingly little time to make such a plan. For your own personal work, you should be able to make such a plan in a few hours to a day or two. Even teams building large products can typically make these plans in about a week. Finally, I can guarantee that, if you take these steps and make such a presentation, management will trust your plan and negotiate an aggressive but realistic schedule with you.

Maintaining Trust

Once you have produced a complete and realistic plan, and once you have convinced management to accept that plan, you can get to work. However, as you do the work, you must continue to maintain management's trust. Trust is fleeting, and after a relatively brief period, management will start to worry and begin to suspect that you are having problems. Given a little time, they will even start thinking that you cannot be trusted to meet your commitments. To counter this natural tendency, you must keep management regularly informed about your status and progress.

Again, this is not that difficult, and the PSP and TSP books can show you how to do it. I have found that, at the project level, weekly reports to your immediate managers are required. If you waited for a month between reports, management would almost certainly worry and even two weeks is probably too long. These reports should factually describe your progress and summarize any key issues and problems together with the actions you are taking to address them. By doing this, you can keep management on your side and ensure that they will be willing and able to help you if you run into serious problems.

Since problems are common in development work, and since some of these problems are likely to impact your schedule, you will occasionally find that you cannot meet an important commitment. While preparing a sound and detailed project plan will usually enable you to meet your commitments, occasionally that won't be possible. This is when frequent management reports are most important. As soon as you know that the schedule is exposed and that you are almost certain to miss the date, go to your management for help. As long as you keep them regularly informed and give them plenty of warning of problems, they will be willing to help solve the problems. Then, even if you do have to miss the schedule, management will continue to trust you to manage your own work.

Commitments in Practice

Depending of your organization’s business, use the same commitment principles but follow different implementation practices. For example, with an existing fixed-price contract, cost and schedule will be paramount and the key will be finding the alternative approaches that most closely meet the existing commitments. Since even fixed-price contracts can run late and over cost, however, make the most realistic plans that you can and face any bad news as quickly as possible. When developers merely hope things will improve, they practically never do. Then, when the bad news must be faced, there is no time to consider alternatives or prepare for the overruns. While the bearers of bad new are often shot, if you know what you are talking about and can defend your story, you will almost always come out ahead. But since there are no guarantees in this business, you must have the courage of your convictions.

Regardless of the business situation or the size and scope of the job, you will know least about the job at the beginning and learn more as development proceeds. Therefore, for any reasonably sized project, you must make progressively more detailed and better informed plans every few months. Also, since it is impossible to make detailed plans for complex development work that extend for more than three or four months, you must regularly replan anyway.

Regardless of your situation, the key is to negotiate a realistic plan and commitment at the beginning. Then, as you learn more about the work, continue refining your plan as you strive to meet that commitment. If you find that you can no longer meet the commitment, get to management right away so they can both help you meet the commitment and prepare for any possible delay.

Conclusions

This series of five articles has described how to take charge of your personal work and explained why this is the only way to have a truly satisfying and rewarding work life. The first column in the series described the different ways that managers and developers view success and it concluded that, to be truly successful, projects must be successful for both the managers and the developers. The second column pointed out that managers often seem autocratic merely because their developers don’t know how to make accurate plans or consistently meet commitments. The third column points out that the only ones who can make accurate and useful plans for knowledge-working projects like software development are the knowledge workers themselves. Part four of the series then describes the five steps required for developers to take charge of their own work, and part five discusses the challenges and risks of actually doing so. Taking charge of our own work is the key to a rewarding and satisfying career. Once you know how to do it, you won’t want to work in any other way.

Acknowledgments

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Tim Chick, Julia Mullaney, and Bob Stoddard.

In Closing, an Invitation to Readers

In these columns, I discuss development issues and how they impact the work of engineers and their organizations. However, I am most interested in addressing the issues that you feel are important. So, please drop me a note with your comments, questions, or suggestions. Better yet, include a war story or brief anecdote to illustrate your ideas. I will read your notes and consider them when planning future columns.

Thanks for your attention and please stay tuned in.

Watts S. Humphrey
watts@sei.cmu.edu

References

[Humphrey 2005]

Watts S. Humphrey. *PSP*: *A Self-Improvement Process for Software Engineers*, Reading, MA: Addison Wesley, 2005. http://www.sei.cmu.edu/library/abstracts/books/0321305493.cfm

[Humphrey 2006]

Watts S. Humphrey. *TSP: Coaching Development Teams*, Reading MA: Addison Wesley, 2006. http://www.sei.cmu.edu/library/abstracts/books/201731134.cfm

39 Crisis Management

2008 | 5

The idea for this column was suggested by Anand Paropkari and, I am ashamed to say, he made that suggestion an embarrassingly long time ago. In any event, his point was a good one. He said: "Most IT managers think that CMM and CMMI are not for them. Their perception is that the models are developed for IT vendors or DoD contractors."

I hear this all too often: "We are different." These folks seem to be saying that what worked for somebody else will not work for them. If we take this argument to its extreme, nobody could ever learn from anyone else, and we would all be stuck in whatever rat trap we happened to fall into. Then we would all have to learn by ourselves how to solve our own problems. If our forebears had all followed that strategy, we would never have progressed out of the stone age. Of course everybody is different as is every organization. But, at least in the software business, we share a great many common problems.

The Key Question

The fundamental question is whether your boss wants to improve the way the work is done. If he or she does, you probably will not have much trouble making the case for exploring an improvement effort. Then the problem would be to figure out how to get the boss' attention, what kind of improvement method to propose, and how to make the case. Actually, I have written four columns on this topic that you can access on the SEI website (http://www.sei.cmu.edu/). These columns ran from September 1999 through June 2000.

More likely, however, your boss' saying "We're different" is just a way of saying "Don't bother me, I'm busy." When your boss is not interested in process improvement, there are two likely reasons. He or she is either a plodder or is too preoccupied with the current crises to think about anything else.

A Plodder Example

By plodder, I don't mean someone who is lazy but rather a manager who is comfortable with the way things currently work and doesn't want to be bothered with some new improvement effort. Such people will usually give an excuse like "we tried that and it didn't work" or cite an example of an improvement effort that failed. A more sophisticated management response is to ask for a return-on-investment analysis. Since such studies typically take a lot of effort and are easy to shoot down, such requests are guaranteed to indefinitely postpone further talk of improvement.

A good example of the plodder case was the director of a laboratory where I worked many years ago. After I had explained my improvement proposal, he asked how long it would take to recoup the investment. I explained that, realistically, it would take from five to 10 years. He then answered: "But I retire in five years." I thanked him for his time, rolled up my presentation flip charts, and started looking for another job. With bosses like this, you only have two choices: outwait them or leave. I left.

The other case is much more likely. It is that your boss is so enmeshed in the current crises that he or she doesn’t have the time or energy to think about anything else. This is the most common case, and it is the one most of us need to think about, either as workers trying to convince busy bosses to make changes or as bosses who are too busy to think about anything but surviving. In either case, if you handle the situation properly, you have a good chance of actually making some improvements.

The way to deal with a busy boss is built into the Capability Maturity Model Integration CMMI) model and the improvement methods such as the Personal Software Process (PSP) and Team Software Process (TSP) that grew out of it. The best way to appreciate the CMMI strategy is to realize that the move from Level 1 to Level 2 is actually a move from crisis-driven management to plan-driven management. That is a fundamental change. It will show immediate and lasting benefits and should be top priority for any busy boss.

The Power of Planning

The great power of plan-driven management was driven home to me many years ago. I worked at IBM, and the company had announced and was starting to ship a new line of hardware products. Unfortunately, the software to support these products was late, and the schedule had slipped three times. At that point, the company fired the director of programming and gave me the job. I now had nearly 4,000 programmers working for me in 15 laboratories in the United States and Europe. They were all working to develop parts of this system, and everyone was late. The customers were irate, the marketing force was in turmoil, and everybody demanded delivery dates from me RIGHT NOW! This indeed was a crisis.

When I arrived in my new office the first day, I quickly made two decisions. First, I wanted no mail unless it came from my boss, his boss, the senior VP, or the chairman. My predecessor had spent most of every day and night just reading and answering his 3-foot stack of daily mail, and my large and overworked office staff spent all of its time just producing a 20- page mail summary. Clearly, reading and answering mail had not prevented and would not solve the current crisis.

The second thing I needed to do was to find out what was going on in the development groups. To do this, I scheduled trips to the three largest development laboratories. I found the same story at every one. Nobody had any plans and they didn’t even have a prioritized list of what they had to do. Everybody was just banging out and testing code as fast as they could. I then asked the management team at each location how they would do the job if they did it in the best way they knew. They all agreed that they would establish clear priorities lists, nail down the requirements, and develop plans.

When I asked them why they didn’t do that, they all said the same thing: “We don’t have the time.” This was clearly nonsense! The best way to do the job had to be the fastest and cheapest way. These folks, as smart and competent as they were, were clearly in panic mode and flailing.

Crisis Management is an Executive Problem

The problem was that all of these managers had an enormous list of things that they had to do. But to ship code, the only things that were essential were coding and testing. So they spent their time coding and testing, and they were then so busy that they didn't have time to do any of the other things that they knew they "had to do." While everyone believed that planning was important, it wasn't an essential prerequisite to shipping code and therefore was considered optional.

It took me a little while to realize that, even though I had only been in the job for a couple of weeks, I was responsible for this whole crisis. I then went to the senior VP and told him I was going to stop everything. He turned pale. However, after I explained what I planned to do and why, he agreed. Since all that the developers could do was respond to crises, I had to turn planning into a crisis. I then sent a directive to all of the laboratories. Henceforth, no one could ship a program or announce a program, and I would not even fund a program until I had plans on my desk. They had 60 days to produce their plans and to review them with me personally. This made planning a crisis.

At the same time, I gave a talk to each of the four marketing regions to explain to the sales force what we were doing. I said that we had cancelled all of the software schedules and that we were busily developing plans for the work. When we had the plans in 60 days, we would announce new schedules. Furthermore, these would be schedules that they could believe. Several salesmen later told me that, for the first time, they were able to explain what was going on to our customers and that, while the lack of dates was a problem, they could start selling again.

The Plan Reviews

In the plan reviews, I found lots of holes. Some plans weren't synchronized with planned release schedules, others left out quality assurance or documentation, and a few even left out integration and final testing. I didn't cut a single schedule, but I did add a 90-day cushion to all of the release end dates. As I explained to the development managers, they might be embarrassed if they missed a release date; but I could get fired.

While we gradually ate up the 90-day schedule cushion, we did not miss a single delivery date for more than two years. The change in the organization was dramatic. This group that had never before met a schedule was now consistently delivering on or ahead of plan. People now had time to think, product quality improved dramatically, and many of the developers and managers were even getting home for dinner with their families. Now, instead of being bums, the programmers were treated like the true heroes they were.

Getting Your Boss' Attention

Now that you are convinced that crisis management is a black hole, how do you convince your manager? There are three general ways to attack this problem. The first is by direction. To do this, however, you have to go over your boss' head and get higher management to issue a directive. While this will almost certainly work, it is generally very risky unless your father is company president or your uncle is a majority stockholder.

The second approach is through logic. If you work directly for the executive who has the power to resolve this problem, and if you are on good enough terms to have such a discussion, this is probably a good next step.

The third approach, however, is almost always effective. That is to identify those elements of the recurring crises that are under your personal control. Then figure out how to establish plans to resolve the crises before they blow up. You might even involve some co-workers in doing this with you and cover as many of the crisis causes as you can identify and control. Then, after you have established and used these plans long enough to demonstrate their effectiveness, go to your boss and explain what you did and why it worked. Assuming that your boss finds what you did helpful, that would provide the opening for broadening your efforts to include more of your boss' territory and, ultimately, even to start looking at a more comprehensive improvement strategy.

Getting Started

After getting your boss' attention, you need to have a concrete plan of action. The key to any improvement plan is to start with a modest initial effort but to also ensure that this effort will produce tangible positive results. For example, you could either look at a small part of CMMI such as product planning or start with two or three TSP projects [Chrissis 2007, Humphrey 2002]. Regardless of the approach you take, it is important to have management's agreement.

While you can often accomplish a lot by yourself, any CMMI improvement effort must have reasonably broad coverage to make a measurable improvement in the organization's performance. For broad coverage, you need senior management support. Similarly, while initial TSP introduction can generally be confined to a small part of the organization, you will need senior management support here as well. This is both because the professional guidance to launch even a small TSP effort costs money and also because TSP introduction calls for a change in management style. However, management style is set by senior managers, so they must be involved.

A Final Caveat

All process improvement programs generally encounter a hidden trap. The problem is that crisis management is an all-consuming activity. Everybody is working hard, people are staying late, a lot is going on, and there is a great feeling of accomplishment. This is the typical reaction of people who confuse speed with progress. Highly professional and efficient work, however, is almost invisible. The product just gets delivered on schedule and works. What is the big deal?

What this means is that, after you take the first improvement step and it is successful, most of the causes for your crises will now be prevented and the immediate rationale for the improvement effort will have gone. If you haven't built the case for the next improvement step before you reach this point, the entire improvement effort will almost certainly be truncated and you will be back where you were at the beginning.

Acknowledgments

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Bob Cannon, Harry Levinson, and Bill Peterson.

Watts S. Humphrey
watts@sei.cmu.edu

References

[Chrissis 2007]

Chrissis, Mary Beth; Konrad, Mike; & Shrum, Sandy. *CMMI: Guidelines for Process Integration and Process Improvement. 2nd Edition.* Reading, MA: Addison Wesley, 2007.
http://www.sei.cmu.edu/library/abstracts/books/0321279670.cfm

[Humphrey 2002]

Humphrey, Watts S. *Winning with Software: An Executive Strategy*. Reading, MA: Addison-Wesley, 2002. http://www.sei.cmu.edu/library/abstracts/books/0201776391.cfm

40 New Priorities

2008 | 7

This column has now run for 10 years, and I have decided that it is time to change its focus. I have enjoyed the privilege of sharing my thoughts with you but have concluded that 10 years is long enough for this particular way of discussing software issues and feel that now is the right time to present new ideas and different viewpoints. I say more about my specific ideas and plans at the end of this column.

In reviewing the 40-some columns I have produced for *SEI Interactive* and later *news@sei* in the last 10 years and the 17 monthly columns I produce for an on-line journal (*ObjectCurrents*) run by Bob Hathaway in 1996 and 1997, I have been struck by several thoughts. The first and most obvious reaction is how little has changed. My very first column in January 1996 was titled the "The Changing World of Software." It talked about the truly abysmal state of software practice and how our customers seem to tolerate this poor performance without a whimper. What disturbs me the most about this first column is that I could have written it 30 years ago or last week. While a lot has changed for software technology, and we have learned a great deal about software processes and how effective they can be if properly designed and used, our performance as an industry is still just about as bad as ever. Further, the way we teach our computer professionals has not materially changed, at least not by enough to improve industrial performance.

After my initial column on the changing world of software, the next four were anecdotes about software issues followed by an 11-column series on the PSP PROBE estimating method. At the time I had just completed my PSP research and was planning to devote my subsequent columns to describing the various PSP methods. Unfortunately, the *ObjectCurrents* journal shut down in May of 1997, so that ended my first series of 17 columns.

About a year later, in June 1998, the SEI asked me to write a quarterly column for a new on-line publication, *SEI Interactive*, with the column to be called "Watts New." I have now completed 10 years of these columns and have covered a lot of territory. In reading them over, I find that they are all just as pertinent today as they were when I wrote them. The only significant difference is that, during much of this time, we were just developing the Personal Software Process (PSP) and Team Software Process (TSP) and gathering data on their use in industry. While I was convinced that these methods were revolutionary, I couldn't prove it, so my columns tended to be more abstract with a primary emphasis on principles and methods.

We now face a different situation. Ten years ago, the PSP and TSP methods were new and little was known about them. While the PSP and TSP are not yet widely used, their use is growing and there is now a substantial body of literature about them [Callison 2009]. In this 10-year period, I have published many technical papers, 57 columns, and 7 books. There is also a growing literature by industrial and academic users who describe their experiences and findings regarding TSP use. Now that a lot has been written about the subject and there is a growing body of evidence that they are highly effective, at least when properly used, it is time to reassess the situation and decide what is needed next.

When I retired from IBM 22 years ago, I was concerned about the poor state of the software industry, and I decided to dedicate my efforts to transforming the world of software. I made what I called an outrageous commitment to fix the world's software problems. I then joined the SEI and formed the Process Program. At that time, we faced two major challenges:

1. to determine how software work should be done
2. to get the world to do software work that way

In the intervening years, we have made a great deal of progress with the first challenge and a modest amount of headway with the second. Regarding the first challenge, we have come a long way, but it would be presumptuous to say that we now have the last word on how software work should be done. However, we do know enough to enable software organizations to consistently and predictably deliver quality software products. That is an enormous advance. With CMMI, we have shown that management has a key role to play in the software process and that, when management issues are properly addressed, organizational performance improves. As a by-product of the CMMI work, however, we have also learned that changing the practices and behavior of the managers is not enough. Unless we address the way the individual software developers and their teams work, we cannot achieve our long-term objectives.

To address these personal and team issues, we next developed the PSP and TSP, and the results have been truly extraordinary. Consider the following facts.

- Microsoft has invested about $3,000,000 to introduce TSP into its IT organization. Microsoft has trained about 1,000 software developers in PSP and now has data on more 200 TSP projects. Because of improved product quality and more predictable schedules and costs, Microsoft estimates that TSP has saved them a total of $84,000,000 to date.
- Intuit has introduced TSP into its largest division and currently does about 60% of its development work with this method. Because of improved product quality, they report that the number of customer calls to their help lines has declined by 30%. This is a reduction of 800,000 calls a year, and each call costs them $25. The total saving from this source alone is thus $20 million a year.
- Vicarious Visions, a division of Activision, introduced TSP because the quality of engineering work life had declined so much that turnover reached 17% a year. They now report that their experienced engineers, once they have worked on a TSP project, refuse to work any other way.
- Softtek, the largest software company in Latin America, reports that the engineering turnover on their TSP teams is one quarter of that on their other teams.

With results like these, you would think that everybody would be jumping onto the TSP bandwagon, but that is not the case. For example, even though Microsoft's IT organization has shown that they can save $84 million from a $3 million investment in TSP, none of Microsoft's product development groups have adopted TSP. How can this be?

Lest I sound critical of Microsoft, I assure you that they are not unique. Every major corporation that has introduced TSP has had a similar experience. Unless the introduction effort started at the top of the organization, adoption did not spread. Somehow, everybody seems blind to innovations made by other people or groups, even within the same company. What is most surprising is that this includes many of the large Department of Defense contractors and other organizations that have been rated at CMMI Maturity Level 5.

This is most surprising because one of the two principal requirements to be rated at CMMI Level 5 is the Organization Innovation and Deployment (OID) process area. This area requires that Level-5 organizations look for promising incremental and revolutionary improvement opportunities both within and outside of their organizations, and that they quantitatively evaluate these opportunities for potential use within their organizations. To quote from CMMI [Chrissis 2007]:

> *The purpose of the Identify and Analyze Innovations specific practice is to actively search for and locate innovative improvements. The search primarily involves looking outside the organization.*

In all the years that we have been working on TSP, we have yet to have any CMMI Level-5 organization come to us for the data needed to conduct such an evaluation. This can't be because they have never heard of TSP. I routinely give talks at major conferences, including the SEPG conferences in the United States and other parts of the world, and I often ask how many people have heard of TSP. While 10 years ago, very few hands went up, today more than half of the people in the audience have heard of TSP. This includes audiences in Australia, Chile, China, Hungary, India, and all over the United States. Clearly, the major software process-improvement issue faced today is the second challenge.

Getting the World to do Software Work in the Best Known Way

Based on the data we have seen to date, that means using TSP. Depending on whom you talk to, this statement will likely get reactions like the following.

- We are only at CMMI Level 2 and need to get to Level 3 before trying TSP.
- We use Agile methods so we can't adopt TSP.
- We just started to use RUP so TSP is not appropriate for us.
- We use Function Points and TSP only uses LOC, so we can't change.

These views are all based in misinformation. TSP has been used successfully by organizations at every CMMI maturity level, and TSP is used by many groups that use Agile methods, including Scrum and XP. Similarly, RUP is completely consistent with TSP, and there is no reason not to use Function Points in estimating TSP jobs. TSP was designed to be language, method, and environment independent. It adds a family of measurement, planning, tracking, and quality-management practices that are not currently used by any of the popular software methods, so it does not conflict with any of them. It merely requires that the developers and their teams do some things that they do not now do.

Based on the results we have seen to date, it is clear that TSP use will grow but that the rate of this growth will be very slow. While we certainly have to continue doing what we have been doing, when you get to be my age, you would like to see results in less than the 10 to 20 years that our current rate of progress implies. This problem, however, is not new. W.E. Deming struggled for years to convince U.S. industry to adopt his well-demonstrated quality methods. Unfortunately for the U.S. auto industry, the Japanese adopted Deming's methods first. Now, Toyota is within a hair of becoming the largest automaker in the world, and it is already the most profitable.

So is there any hope that we can accelerate the rate of TSP adoption? I believe that there are five possible avenues.

1. Convince the computer science and software engineering academic communities of the effectiveness and essential nature of TSP methods. While this community could be enormously helpful in convincing the world that TSP is the right way to go, I give this strategy very low odds. Based on what we have seen to date, the academic community changes even more slowly than industry. This does not mean that it could not change, but that it is not likely to happen soon.
2. Get the customers to demand that their software suppliers adopt TSP. While this would be nice, it is not likely to happen until a very large number of customers have experience with TSP, and it is obvious to everyone that it is the right way to go.
3. Convince the government to establish a software industrial improvement program based on TSP. In the United States, such a strategy is a non-starter, at least for now. Outside of the United States, however, the situation is quite different. When organizations are hungry for business, they are much more receptive to innovative new methods.
4. Get the U.S. government to mandate TSP use, at least as a prerequisite for DoD software development contracts. While this approach was very effective in getting CMMI adopted by the major DoD suppliers, it has a serious downside. The problem is that if the government mandated TSP as a prerequisite for software development contracts, that would be motivating organizations to do something they didn’t believe in (adopt the TSP), in order to get something they wanted (a DoD contract). While this approach can work when you have a foolproof way to determine if someone is using TSP properly, there is an enormous motivation to cheat, which could easily lead to complications.
5. Find some better way to communicate the benefits of TSP to senior corporate executives. While I do not now see a clear way to do this, I do have ideas about possible avenues to explore.

In summary, I have concluded that the most important thing to do right now is to broaden the understanding of TSP’s capabilities and benefits. The approach I have selected is to continue editing these columns but to invite members of the TSP team at the SEI to contribute the material. Also, where there are opportunities, I hope that some of the columns will be co-authored by TSP users. We have learned a great deal over the last 15-plus years, and these columns can provide a convenient and easy way to quickly communicate significant new results to the broader software community. So, in conclusion, that is why I have decided to refocus this column. Please stay tuned, and continue to drop me notes on TSP-related topics you would like to see addressed.

Watts Humphrey
watts@sei.cmu.edu

References

[Callison 2009]

Rachel Callison & Marlene MacDonald. *A Bibliography of the Personal Software Process (PSP) and the Team Software Process (TSP)*, CMU/SEI-2009-SR-025. Carnegie Mellon University, Software Engineering Institute, 2009.
http://www.sei.cmu.edu/library/abstracts/reports/09sr025.cfm

[Chrissis 2007]

Mary Beth Chrissis, Mike Konrad, & Sandy Shrum. *CMMI Second Edition, Guidelines for Process Integration and Product Improvement*. Reading, MA: Addison Wesley, 2007.
http://www.sei.cmu.edu/library/abstracts/books/0321279670.cfm

41 How Mexico is Doing It

2008 | 8

Watts S. Humphrey & Anita Carleton

Introduction

In five years, Ivette Garcia, the director of Mexico's digital economy hopes Watts Humphrey will be asked, "How did Mexico do it?" In her keynote address at the Third Annual Team Software Process (TSP) Symposium held in Phoenix, Arizona, Garcia announced that she hoped the Mexican software industry would soon compete for a larger share of the U.S. software outsourcing market. How would Mexico accomplish this? "You need to differentiate yourself to compete. Mexico plans to differentiate itself through its largest competitive advantage—the TSP," says Garcia [Garcia 2008].

Background

Although the International Data Corporation (IDC) estimates that the IT services outsourcing global market reached $310 billion last year, only about 8% of that is done from offshore destinations, with India being the undisputed leader. McKinsey estimates that by 2010, the global IT outsourcing market will reach $1.1 trillion. The increase in the share of this market served from offshore destinations could reach 15%. That means that the offshore outsourcing market could reach $165 billion in the next four years.

Although Mexico is the United States' second largest trade partner, the Mexican software industry does not yet compete effectively for a share of the U.S. software outsourcing market. For example, in 2007 India sold $3 billion of software services to the United States compared to $900 million for Mexico. However, as the market continues to grow, no single nation will be able to satisfy the market need. This provides an opportunity to increase Mexico's participation in this growing market. The Mexican strategy to accomplish this is for Mexican organizations to improve quality, productivity, and delivery schedules.

Mexican Government Launches National Initiative

The Mexican government, through an initiative called PROSOFT, has launched an aggressive program to build its national reputation. This initiative integrates government, industry, and academia to develop competitive human capital, strengthen local industry, enhance process capabilities, improve quality assurance, and promote strategic alliances with foreign companies. A key to this program is the introduction of TSP. TSP provides teams and their management with precise operational guidance on how to implement CMMI high-maturity practices. The Personal Software Process (PSP) provides the development team members with the skills and practices needed to be productive and effective TSP team members [Humphrey 2002, Humphrey 2005].

Due to the global competition in information technology, Mexico has to differentiate its supply, offering quality in the development of IT products and services in less time and with higher added value. "We know that in order to develop consistent quality software, we need to have high maturity processes. The most popular process maturity model internationally is CMMI. But this model is complex to implement—especially in small enterprises," says Garcia.

The strategy to increase the software industry's maturity in Mexico has to consider not only the enterprises' processes, but also the improvement of the basic element that supports the industry: the people. Building high performance knowledge workers is the focus of PSP and building high-performance working teams is the focus of TSP.

As a whole, the software industry needs to improve cost and schedule management, cycle-time, and product quality. Improving performance in these areas and developing the workforce capability are important PROSOFT goals. Previous reports [Davis 2003] document the success of TSP in producing high quality products on time and within budget. TSP operationally implements high performing development processes. These processes are managed by trained individuals and teams.

PROSOFT has been able to make a number of significant advancements. Mexico now has 120 software development centers with certifications in several quality process models (e.g., CMM, CMMI, ISO, MoPoSoft). In 2002, only four centers had certification. Additionally, there are now 23 academic clusters and 17 industry integrators in the IT sector and 121 universities focusing on improving professional IT education.

Some Early Results

Based on some initial results from training and implementation data for projects and individuals that have adopted TSP in Phase I of the Mexican TSP initiative, the results show that TSP teams are delivering high quality (low defect) software on schedule, while improving productivity. These data can be used for benchmarking, lessons learned, and other guidance to those currently using the TSP or considering participation in the future [Nichols 2008].

Some early results from pilot projects show that the pilot TSP teams delivered their products on average on or within two weeks of the committed date. This compares favorably with industry data that show over half of all software projects were more than 100% late or were cancelled. Key to schedule success was overall high product quality—several TSP projects had no defects in system or acceptance test.

The TSP is implemented by a highly motivated development staff and management. Given the opportunity to speak for themselves, developers say they prefer the work environment of a TSP team. Management likes depth of the data and the reliability of status reports. Low worker attrition, a relative strength of Mexico, is not only maintained, but enhanced. One company survey of employees found the TSP pilot team to have the highest job satisfaction in the plant.

During the initial TSP roll out phases, a number of challenges surfaced:

- the up-front cost in both time and money
- management acceptance and support of self-directed teams
- appropriate use of the detailed data

While these problems are not unique to Mexico, they do need to be addressed to roll out this program on an organizational and national scale. A particularly important issue for Mexico is the number of small and medium sized enterprises that cannot afford the initial training. In the outsourcing market, small short-term projects must be staffed and launched on short notice. New PSP training courses have been developed by the SEI to reduce the time and cost required to launch teams.

Next Steps

TSP is beginning to show some promising results and benefits for Mexican companies. Rolling out on a national level, however, is not only challenging, but unprecedented. In addition to the practical problems of the rollout, national success depends on visibility and recognition of the accomplishments. Next steps include:

- training PSP developers in the universities
- training Mexican university professors to deliver PSP and TSP classes
- developing TSP as a cost effective way to implement CMMI
- certifying and recognizing companies that effectively use TSP

"I want the world to recognize the benefits of working with Mexico. If you are going to work with someone, you need to trust them. That is why we need TSP/PSP. So people will have trust in the quality of our products and services," stated Garcia. Mexico has launched an unprecedented and far reaching program to change an entire industry and has committed significant national, state, academic, and industrial resources to doing this on an aggressive schedule. If Mexico is successful, other countries are likely to follow its lead. We should all stay tuned to Mexico's progress as it pursues this impressive and aggressive national strategy.

References

[Davis 2003]

Davis, Noopur & Mullaney, Julia. *The Team Software Process (TSP) in Practice: A Summary of Recent Results*," CMU/SEI-2003-TR-014. Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/library/abstracts/reports/03tr014.cfm

[Garcia 2008]

Garcia, Ivette. "Prosoft 2.0 A National Program to Develop The IT Industry." Keynote Address at the Third Annual Team Software Process (TSP) Symposium held in Phoenix, Arizona, September 2008.

[Humphrey 2002]

Humphrey, Watts S. *Winning with Software: An Executive Strategy*, Reading, MA: Addison-Wesley, 2002. http://www.sei.cmu.edu/library/abstracts/books/0201776391.cfm

[Humphrey 2005]

Humphrey, Watts S. *PSP: A Self-Improvement Process for Software Engineers*, Reading, MA: Addison-Wesley, 2005. http://www.sei.cmu.edu/library/abstracts/books/0321305493.cfm

[Nichols 2008]

Nichols, William and Salazar, Rafael. “Deploying TSP on a National Scale: An Experience Report from Pilot Projects in Mexico,” CMU/SEI-2008-TR-026. Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/library/abstracts/reports/08tr026.cfm

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.			
1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE November 2009		3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE The Watts New? Collection: Columns by the SEI's Watts Humphrey			5. FUNDING NUMBERS FA8721-05-C-0003
6. AUTHOR(S) Watts S. Humphrey			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213			8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2009-SR-024
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116			10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES			
12A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS			12B DISTRIBUTION CODE
13. ABSTRACT (MAXIMUM 200 WORDS) Since June 1998, Watts Humphrey has taken readers of *news@sei* and its predecessor *SEI Interactive* on a process-improvement journey, step by step, in his column Watts New. The column has explored the problem of setting impossible dates for project completion, planning as a team using TSP, the importance of removing software defects, applying discipline to software development, approaching managers about a process improvement effort, and making a persuasive case for implementing it. After 11 years, Watts is taking a well-deserved retirement from writing the quarterly column. But you can still enjoy vintage Watts New columns, including all of the above topics, in the *news@sei* archives (http://www.sei.cmu.edu/library/abstracts/news-at-sei/) or in the Watts New Collection.			
14. SUBJECT TERMS project management, TSP, team software process, PSP, personal software process, software defects, disciplined software development, process improvement			15. NUMBER OF PAGE S 238
16. PRICE CODE			
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18
298-102