CHAPTER TWO

Fool's Gold

Hope is a good breakfast, but it is a bad supper.

FRANCIS BACON

Software problems have persisted partly because of the bewitching appeal of a few common, ineffective practices. During the California Gold Rush in the late 1840s and early 50s, some prospectors were deceived by fool's gold—iron pyrite—a substance that has the luster and sparkle of gold. Unlike gold, iron pyrite is flaky, brittle, and virtually valueless. Experienced miners know that real gold is soft, malleable, and doesn't break under pressure. For 50 years, software developers have been succumbing to the temptation of their own fool's gold. They are drawn to flawed practices that have a seductive appeal, but the practices that make up software's fool's gold, like iron pyrite, are flaky, brittle, and virtually valueless.

Moving the Block

Looking back many centuries before the California Gold Rush, suppose that you were working on one of the ancient pyramids and were given the assignment to move an enormous stone block 10,000 meters from a river to the site of a pyramid under construction, as shown in Figure 2-1. You are given 100 days to move the block and 20 people with which to move it.

You are allowed to use any method you like to get it to its destination. Each day, you have to move the block an average of 100 meters closer to the pyramid, or you have to do something that will reduce the number of days needed to travel the remaining distance.

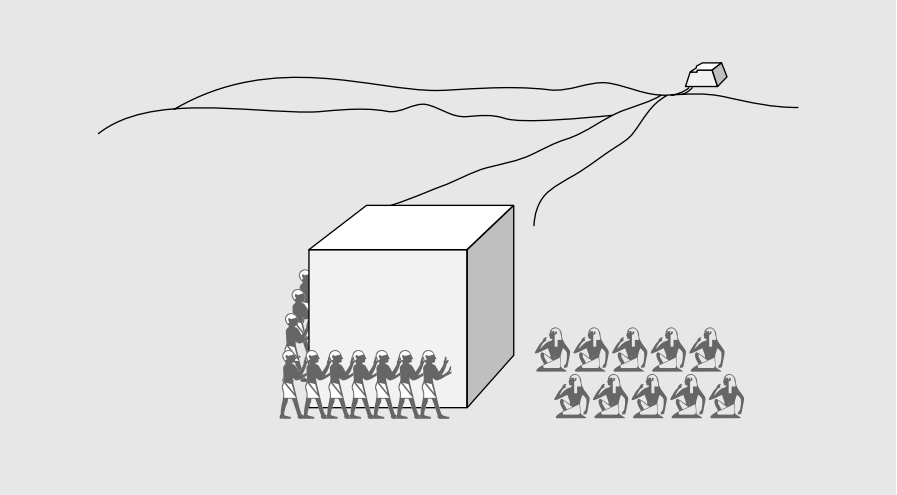

FIGURE 2-1

One way to think of a software project is as a heavy block of stone. You must either move the block one day closer to the final destination each day, or you must do something that will enable you to traverse the remaining distance in one less day.

Some block-moving teams might immediately begin pushing the block, trying to move it with brute force. With a very small block, this method might work, but with a heavy block resting directly on desert sand, this approach won't move the block very quickly, if at all. If the block moves ten meters per day, the fact that it is moving at all might be satisfying, but the team is actually falling 90 meters per day behind. "Progress" doesn't necessarily mean sufficient progress.

The smart block-moving team wouldn't jump straight into trying to move the block with brute force. They know that for all but the smallest blocks, they will need to spend time planning how to move the block before they put their muscles into it. After analyzing their assignment, they might decide to cut down a few trees and use the tree trunks as rollers, as shown in Figure 2-2. That will take a day or two, but chances are good that it will increase the speed at which they can move the block.

What if trees aren't readily available, and the team has to spend several days hiking up river to find any trees? The hike is still probably a good

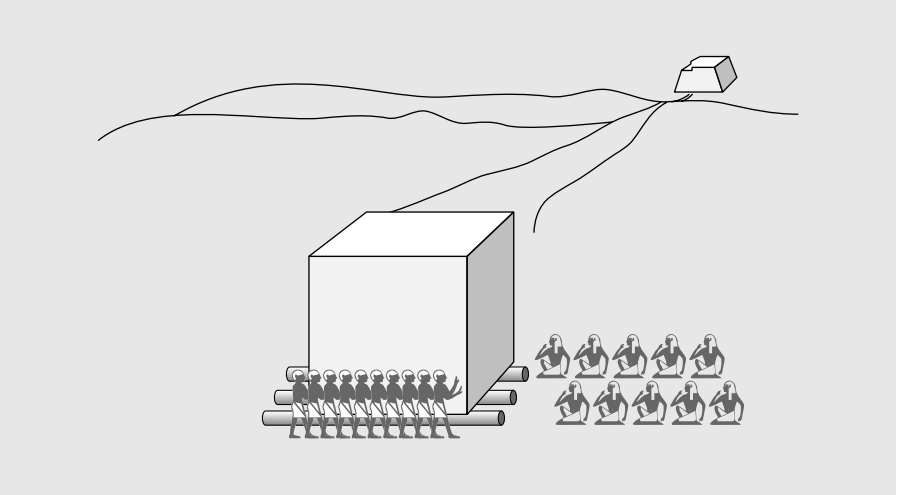

FIGURE 2-2

Whether moving a block of stone or creating computer software, the smart team takes time at the beginning of the project to plan its work so that it can work quickly and efficiently.

investment, especially since the team that begins by trying to use brute force will only move the block a fraction of the distance needed each day.

Similarly, the smart block-moving team might want to prepare the surface over which they'll be moving the block. Instead of pushing it across the sand, they might want to create a level roadway first, which will be an especially good idea if they have more than just this one block to move.

A really sophisticated block-moving team might start with the roller and road system, and eventually realize that having only the minimum number of rollers available forces them to stop work too often; they have to move the back roller to the front of the block every time they move the block forward one roller-width. By having a few extra rollers on hand and assigning some people to move the rollers from back to front, they're better able to maintain their momentum.

They might also realize that their pushing is limited by how many people can fit around the block's base. They might create a harness so that they can pull the block from the front at the same time they're pushing it

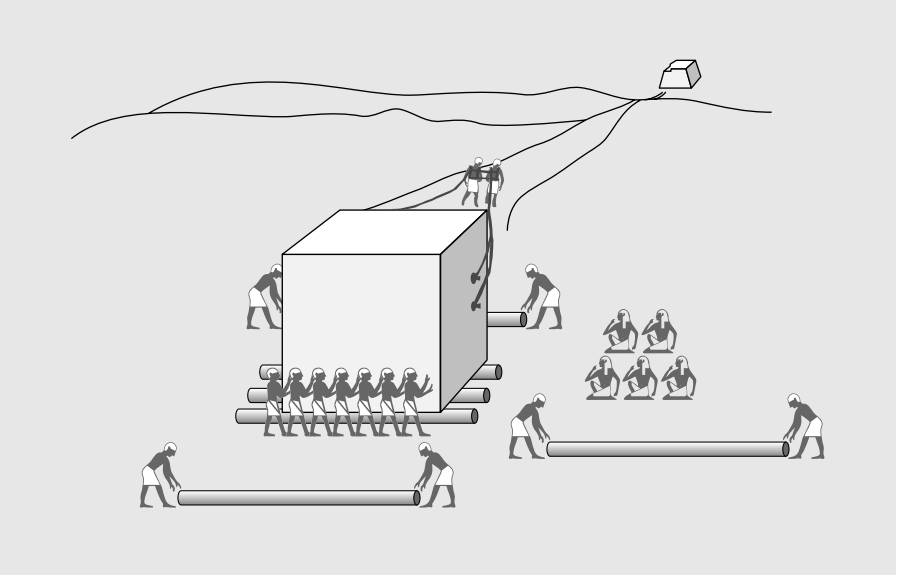

FIGURE 2-3
Smart teams continuously look for ways to work more efficiently.

from behind, as illustrated in Figure 2-3. As more people divide the work, they find that each person's work is easier, and the faster pace is actually more sustainable than the slower one.

Stone Blocks and Software

How does this block moving relate to software? The movement of the stone block is analogous to creating source code. If you have 100 days to complete a software project, you either need to complete 1/100 of the source code each day, or you need to do work that will allow you to complete the remaining source code faster. Because the work of creating source code is much less tangible than the work of moving a stone block, progress at the beginning of a software project can be harder to gauge. Software projects are vulnerable to a "last minute syndrome" in which the project team has little sense of urgency at the beginning of a project, frit-

ters away days on end, and works itself into a desperate frenzy by the end of the project. Thinking of a project's source code as a stone block makes it clear that you can't hope to conduct a successful project by sprinting at the end. Every day, a software project manager should ask, "Did we move the block one day closer to our destination today? If not, did we reduce our remaining work by one day?"

Another way that moving a stone block relates to software is that, eventually, no matter how much planning you do, you do have to move the block; you do have to write the source code. Source-code creation on all but the smallest projects involves an enormous amount of detail work, and it's easy to underestimate it.

Code-and-Fix Development

The problem of not focusing enough on making rollers and preparing roadways is by far the most common problem in software. About 75 percent of the software project teams begin their projects by hurling themselves against the block and trying to move it with brute force.[1] This approach is called "code-and-fix development"—jumping straight into coding without planning or designing the software first. Sometimes teams do this because the team's software developers are anxious to begin coding. Sometimes they do it because managers or customers are eager to see tangible signs of progress. Code-and-fix development is universally ineffective on all but the tiniest projects.

The problem with the code-and-fix approach, as with the brute force approach to moving the stone block, is that quick movement off the starting line doesn't necessarily translate into quick progress toward the finish line. The team that uses a more advanced approach is putting a framework in place that will allow the project to spin up to a high level of productivity and finish efficiently. It is putting rollers under the block, clearing the roadway, and preparing to focus the energy of the project team. The code-and-fix project gets the block moving early, but it doesn't move the block far enough each day and the brute force approach isn't sustainable. It typically leads to the creation of hundreds or thousands of defects early in the

project. Several studies have found that 40 to 80 percent of a typical software project's budget goes into fixing defects that were created earlier on the same project.[2]

Figure 2-4 illustrates the way that productivity erodes over time on a code-and-fix project. Little or no effort is put into planning and process management at the beginning of the project. Some small amount of effort goes into thrashing (unproductive work), but most work goes into coding. As the project moves forward, fixing defects becomes an increasingly prominent feature of the project. By the end of the project, the project that uses code-and-fix development is typically spending most of its time fixing the defects that it created earlier.

As Figure 2-4 suggests, the lucky code-and-fix projects are brought to completion while they are still eking out some small amount of coding

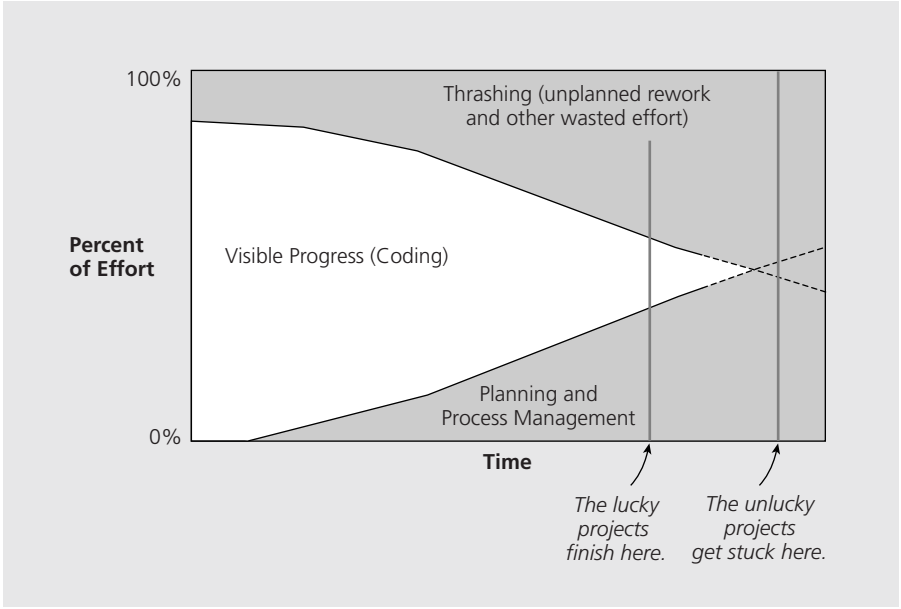

FIGURE 2-4

Using a code-and-fix approach, the lucky projects finish while they are still eking out a small amount of productive work. The unlucky projects get stuck in a zone where 100 percent of their effort is consumed by thrashing, planning, and process management.

Source: Adapted from *Software Project Survival Guide.*[3]

progress. The unlucky projects get stuck on the far right side of the diagram where 100 percent of their available effort is consumed by planning, process management, and thrashing, and they are not making any coding progress. Without enough up-front planning, the code quickly becomes flaky and brittle. Some of these projects might be rescued by taking steps to push the team back to the left enough that they can eke out a release. The remaining projects are eventually cancelled.

This gloomy picture is no exaggeration. Several studies have reported that about 25 percent of all software projects are eventually cancelled.[4] At the time the average project is cancelled, it's 100 percent over budget and is caught in an extended debug, test, and fix cycle (thrashing). The reason it's cancelled is the perception that its quality problems are insurmountable.[5]

The irony of this dynamic is that these unsuccessful projects eventually do as much planning and process management as a successful project would. They have to implement defect tracking to manage all the bugs being reported. They begin estimating more carefully as the release date approaches. Toward the end of the project, the project team might re-estimate as often as every week or even every day. They spend time managing expectations of project stakeholders, convincing them that the project will eventually be released. They may begin tracking defects and imposing standards for debugging code before it's integrated with already-debugged code. But because they begin these practices late in the project, the benefits from these practices are leveraged over only a small part of the project.

The kinds of practices they implement are different from the kinds a more effective organization would implement in a project's early stages. And many of the practices they implement wouldn't have been needed if the project had been run well from the beginning.

As Figure 2-5 illustrates, the most sophisticated organizations—those that produce the most reliable software for the least cost and with the shortest schedules—spend a relatively small percentage of their budgets on the construction part of a project. The least sophisticated organizations spend practically their whole budgets on coding and fixing bugs in their code. Their total budgets are much higher because they don't lay any groundwork for working efficiently. (I'll return to this dynamic in more detail in Chapter 14.)

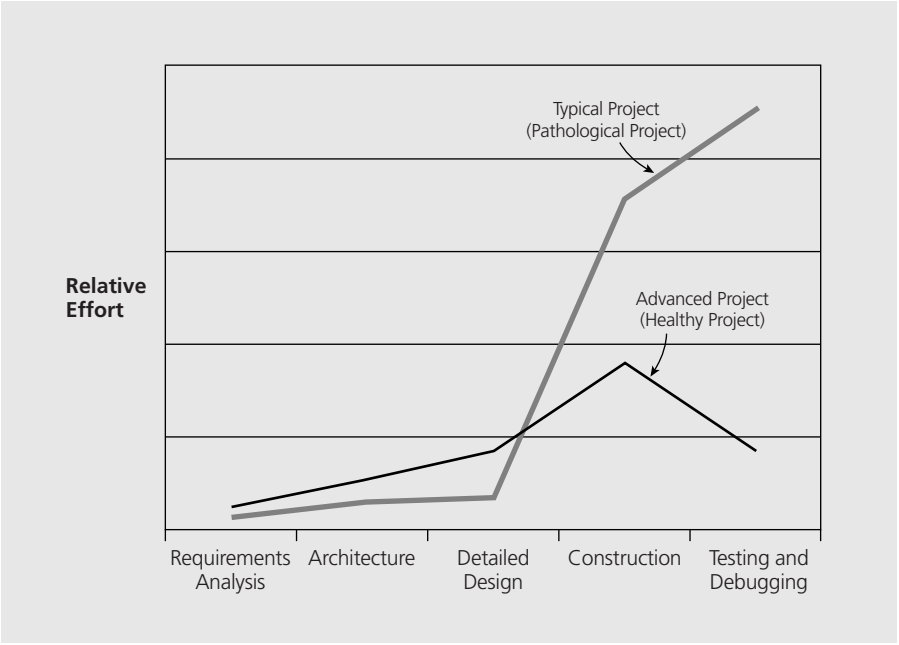

FIGURE 2-5

Advanced software development approaches require more work during the early stages of the project to eliminate an enormous amount of unnecessary work in the later stages of a project.[6]

Code-and-fix development continues to be used because it is appealing in two ways. First, it allows the project team to show signs of progress immediately—they can begin moving the stone block 10 meters per day the first day, while the more effective team is still out cutting down trees, preparing the roadway for a smooth trip, and showing no visible signs of progress on the real problem of moving the block. If managers and customers aren't very sophisticated about the dynamics of a successful project, a code-and-fix approach looks appealing. A second source of code-and-fix development's appeal is that it requires no training. In an industry in which the average level of software engineering training is low, it has been the most common method by default.

The code-and-fix approach is one form of software fool's gold. It seems attractive at first glance, but experienced software developers recognize it as having little value.

Focus on Quality

You might assume that a software project can be shortened by spending less time on testing or technical reviews. "Needless overhead!" say people with a taste for code-and-fix development. Industry experience indicates otherwise. An attempt to trade quality for cost or schedule actually results in increased cost and a longer schedule.

As Figure 2-6 illustrates, projects that remove about 95 percent of their defects prior to release are the most productive; they spend the least time

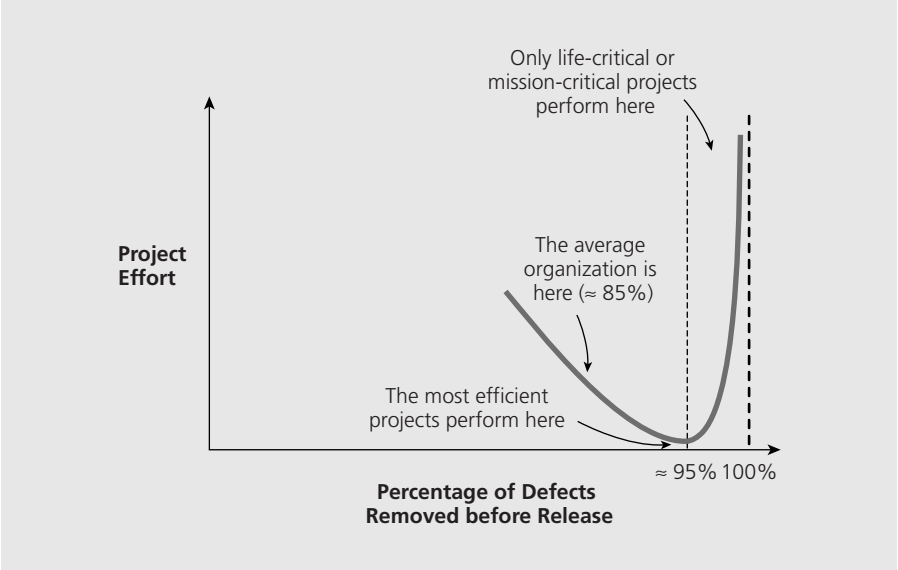

FIGURE 2-6

Up to a point, the projects that achieve the lowest defect rates also achieve the shortest schedules. Most projects can shorten their schedules by focusing on fixing defects earlier.

Source: Adapted from *Applied Software Measurement: Assuring Productivity and Quality,* 2d Ed.[7]

fixing their own defects. Beyond about 95 percent defect removal, projects have to expend extra effort to improve quality. Short of 95 percent, projects can become more efficient by removing more defects sooner. Approximately 75 percent of software projects presently fit into this category. For the projects in this category, the attempt to trade quality for cost or schedule is another example of fool's gold. It's also an example of a software project dynamic that isn't really new. IBM discovered 25 years ago that projects that focused their efforts on attaining the shortest schedules had high frequencies of cost and schedule overruns. Projects that focused on achieving low defect counts had the best schedules and the highest productivities.[8]

Some Fool's Gold Is Silver

Technologies and methodologies that are associated with extravagant productivity claims are called "silver bullets" because they are supposed to slay the werewolf of low productivity.[9] For decades, the software industry has been plagued by claims that the UmptyFratz Innovation dramatically improves development speed. In the 1960s, on-line programming was associated with this claim. In the 1970s, it was third-generation languages. In the 1980s, advocates for artificial intelligence and CASE tools made these promises. In the 1990s, object-oriented programming was lauded as the next great productivity boon. In the early 2000s, it was development in Internet time.

Suppose that a stone-block project team starts out using the brute-force method to move the stone block. After a few days, the team leader can see that progress isn't fast enough to meet the project's goals. Fortunately, he has heard of an amazing animal called an "elephant." Elephants can weigh almost 100 times as much as an adult human being and are extremely powerful. The project leader mounts an expedition to capture and bring back an elephant to help the team move the block. After a three-week safari, the team returns with a captive elephant. They harness the magnificent beast to the block and crack the whip. They hold their collective breath, waiting to see just how fast the elephant will move the block. They

may even finish ahead of schedule! As they watch, the elephant begins pulling the block forward, much faster than the team of humans had ever been able to accomplish. But then, unexpectedly, the elephant rears on its hind legs. It breaks its harness, tramples two of its handlers, and runs off at 40 kilometers per hour, never to be seen again (as shown in Figure 2-7). The stone-block team is dejected. "Maybe we should have spent more time learning how to handle the elephant before we started using him on a real project," they thought. They wasted more than 20 percent of their schedule looking for the elephant, lost two of their teammates, and are no closer to the goal than when they started.

That, in a nutshell, is Silver Bullet Syndrome.

The elephant analogy is more apt than you might think. Robert L. Glass chronicles 16 troubled projects in *Software Runaways.*[10] Four of the projects he describes expected to be breakthrough successes because of their use of silver bullet innovations. Instead, they ended up failing because of the same innovations.

A special kind of silver bullet is forged from attempts to implement organizational process improvement half-heartedly. Some organizations try to implement organizational improvement with buzzwords—TQM, QFD,

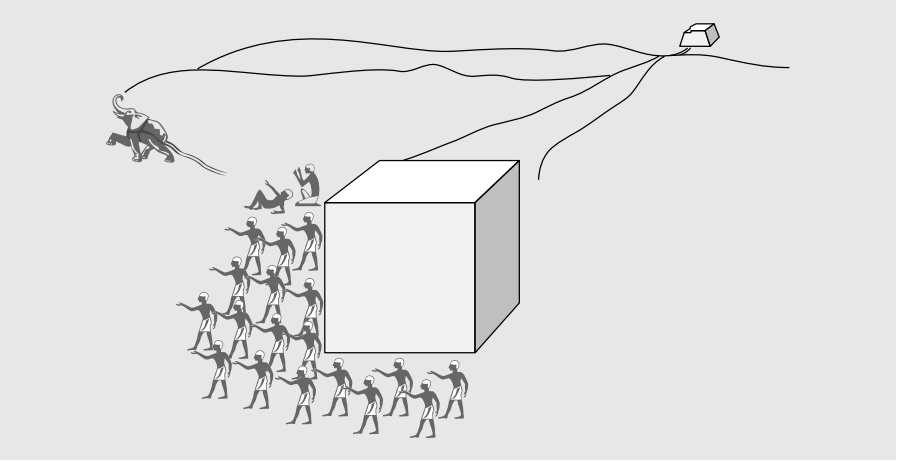

FIGURE 2-7
Silver bullet innovations often fall short of expectations.

SW-CMM, Zero Defects, Six Sigma, Continuous Improvement, Statistical Process Control—these are all valuable practices when properly applied by focusing on the substance of the practice and not just the form. But each of these practices is virtually worthless when applied as buzzwords. Some organizations cycle through them in 12-month intervals, as if ritualistically chanting the initials of a current management fad could call forth improvements in quality and productivity. A special place in low-productivity hell is reserved for these organizations. After years of Management By Buzzword (MBB), entire staffs become cynical about organizational improvement initiatives in general, which adds one more challenge to escaping from code-and-fix development.

The right innovation applied to the right project, supported by appropriate training and deployed with realistic expectations can be tremendously beneficial as a long-term strategy. But new innovations aren't magic and they aren't easy. When they are adopted with a get-rich-quick attitude, innovations become fool's gold.

Software Isn't Soft

One more kind of fool's gold is the belief that software is soft. Hardware is "hard" because it is difficult to change. Software was originally called "soft" because it was easy to change. For very small programs at the dawn of computer programming, this might have been true. As software systems have become more complex, however, this notion that software is easy to change has become one of the most pernicious ideas in software development.

Several studies have found that changing requirements—attempts to take advantage of software's supposed softness—are the most common or one of the most common sources of cost and schedule overruns.[11] They are also a major factor in project cancellations; in some cases, changes resulting from creeping requirements can destabilize a product to such a degree that it can't be finished at all.[12]

A simple example illustrates why software isn't as soft as people think. Suppose that you are asked to design a system that will initially print a set

of five reports and eventually print a set of 10 reports. You have several kinds of flexibility—softness—that you will need to be concerned about:

- Is ten an upper limit on the number of reports?
- Will the future reports be similar to the initial five reports?
- Will all of the reports always be printed?
- Will they always be printed in the same order?
- To what extent will the user be able to customize the reports?
- Will users be allowed to define their own reports?
- Will the reports be customizable and definable on the fly?
- Will the reports be translated to other languages?

No matter how carefully the software is designed, there will always be a point at which the software won't be soft. In the case of the reports, any of the following areas could turn out to be "hard":

- Defining more than ten reports
- Defining a new report that is different from the initial set of reports
- Printing a subset of the reports
- Printing the reports in a user-defined order
- Allowing the user to customize reports
- Allowing the user to define an entire custom report
- Translating the reports to another language that uses a Latin alphabet
- Translating the reports to another language that uses a non-Latin alphabet or that reads right to left

What's interesting about this example is that I can ask a whole hat full of questions about the "softness" of these reports without knowing anything whatsoever about the specific reports or even about the system within which the reports will be printed. Simply knowing that there are "some reports" raises many general questions about the different degrees of softness.

It's tempting to say that software developers should always design the software to be as flexible as possible, but flexibility is an almost infinitely variable entity, and it comes at a price. If the user really wants a standard set of five preformatted reports, always printed as a set, and always printed in the same order in the same language, the software developer should not create an elaborate utility that allows the user to generate highly customized reports. That could easily cost the customer 100 to 1,000 times as much as providing the basic functionality the user really needs. The user (or client or manager) has a responsibility to help software developers define the specific flexibility needed.

Flexibility costs money now. Limiting flexibility saves money now, but typically costs disproportionately more money later. The difficult engineering judgment is weighing the known present need against the possible future need and determining how "soft" or "hard" to make the "ware."

How Fool's Gold Pans Out

In conclusion, we hold the following software truths to be self-evident (or evident after careful examination, anyway):

- The success of a software project depends on not writing source code too early in the project.
- You can't trade defect count for cost or schedule unless you're working on life-critical systems. Focus on defect count; cost and schedule will follow.
- Silver bullets are hazardous to a project's health, though software industry history suggests that vendors will continue to claim otherwise.
- Half-hearted process improvement is an especially damaging kind of silver bullet because it undermines future improvement attempts.
- Despite its name, software isn't soft, unless it's made that way in the first place, and making it soft is expensive.

The software world has had 50 years to learn these lessons. The most successful people and organizations have taken them to heart. Learning to

resist software's fool's gold consistently is one of the first steps the software industry needs to take on the road to creating a true profession of software engineering.

Notes

1. This average is based on the number of software projects at SW-CMM Level 1. See Chapter 14 for more details about this statistic.
2. Software Engineering Institute, quoted in Fishman, Charles, "They Write the Right Stuff," *Fast Company,* December 1996. Mills, Harlan D., *Software Productivity,* Boston, MA: Little, Brown, 1983, pp. 71–81. Wheeler, David, Bill Brykczynski, and Reginald Meeson, *Software Inspection: An Industry Best Practice,* Los Alamitos, CA: IEEE Computer Society Press, 1996. Jones, Capers, *Programming Productivity,* New York: McGraw-Hill, 1986. Boehm, Barry W., "Improving Software Productivity," *IEEE Computer,* September 1987, pp. 43–57.
3. McConnell, Steve, *Software Project Survival Guide,* Redmond, WA: Microsoft Press, 1997. This book contains a more in-depth description of these dynamics.
4. Johnson, Jim, "Turning Chaos into Success," *Software Magazine,* December 1999, pp. 30–39. The Standish Group, "Charting the Seas of Information Technology," Dennis, MA: The Standish Group, 1994. Jones, Capers, *Assessment and Control of Software Risks,* Englewood Cliffs, NJ: Yourdon Press, 1994.
5. Jones, Capers, *Assessment and Control of Software Risks,* Englewood Cliffs, NJ: Yourdon Press, 1994.
6. The "advanced project" profile is based on projects performed by NASA's Software Engineering Lab. The "typical project" is from project data I've compiled from my consulting work and is consistent with data reported by Capers Jones, *Patterns of Software Systems Failure and Success,* Boston, MA: International Thomson Computer Press, 1996, and other sources.
7. Jones, Capers, *Applied Software Measurement: Assuring Productivity and Quality,* 2d Ed., New York: McGraw-Hill, 1997.
8. Jones, Capers, *Programming Productivity,* New York: McGraw-Hill, 1986.
9. Brooks, Frederick P., Jr., "No Silver Bullets—Essence and Accidents of Software Engineering," *Computer,* April 1987, pp. 10–19.
10. Glass, Robert L., *Software Runaways,* Englewood Cliffs, NJ: Prentice Hall, 1998.
11. Vosburgh, J., B. Curtis, R. Wolverton, B. Albert, H. Malec, S. Hoben, and Y. Liu, "Productivity Factors and Programming Environments," *Proceedings of the 7th*

International Conference on Software Engineering, Los Alamitos, CA: IEEE Computer Society, 1984, pp. 143–152. Lederer, Albert L., and Jayesh Prasad, "Nine Management Guidelines for Better Cost Estimating," *Communications of the ACM,* February 1992, pp. 51–59. The Standish Group, "Charting the Seas of Information Technology," Dennis, MA: The Standish Group, 1994. Jones, Capers, *Assessment and Control of Software Risks,* Englewood Cliffs, NJ: Yourdon Press, 1994.

12. Jones, Capers, *Assessment and Control of Software Risks,* Englewood Cliffs, NJ: Yourdon Press, 1994.