Measure Once, Cut Twice (No, Really)

Cary Millsap / Method R Corporation
cary.millsap@method-r.com

Revised 2008/04/22

“Measure Twice, Cut Once” is a reminder that careful planning yields better gratification than going too quickly into operations that can’t be undone. Sometimes, however, it’s better to Measure Once, Cut Twice. It’s one of the secrets behind how carpenters hang square cabinets in not-so-square kitchens. And it’s one of the secrets behind how developers write applications that are easy to fix when they cause performance problems in production use. The key is to know which details you can plan for directly, and which details you simply can’t know and therefore have to defend yourself against.

Introduction

Even if you’ve never picked up a saw, you’ve probably heard the old adage, “Measure twice, cut once.” It’s an encapsulation of advice born from the experience of countless mistakes made when someone has measured, cut, and then realized—d’oh!—the piece is too short, so now he has to patch or start over, consuming more time and materials than he had originally planned.

It’s good advice. Occasionally, I’ll read “19 inches” off my tape measure. I’ll set my saw for a 19-inch cut, and then I’ll measure again. This time, it says 23 inches, and after studying the problem for a while, I realize that I measured the first time to a dark spot that wasn’t really my pencil mark. “Measure twice, cut once” is good advice to remind us all that careful planning yields better gratification than going too quickly into operations that can’t be undone.

“Measure twice, cut once” is well-known far beyond the shop. Occasionally I hear it applied in the context of my job in the Oracle software market. People who say “measure twice, cut once” to me are usually conveying the idea that more planning will defend us against risk down the road.

A few years ago, I read a woodworking article—I’d love to reference it, but now I can’t find it—that broadened my perspective. It advocated the notion that to advance beyond a beginner’s level of woodworking skill, I need also to comprehend the importance of “measure once, cut twice.” …Which, of course, is completely backwards to the good advice we hear so often.

The article was very good, and practicing the ideas I learned in it definitely improved my woodworking skills. As I’ve considered those points many times since my introduction to them, they have resonated with many of the lessons I’ve learned in my professional life as well. In the sections that follow, I’ll explain (using some shop analogies for fun) the importance of “measure once, cut twice” in your life as a software professional.

Act I: The Shop

If you like to use woodworking metaphors while you’re working with software, this section might make you think twice next time. Woodworking isn’t exactly like what people who have never tried it *think* that it’s probably like. A lot of people seem to think that woodworking works like this:

1. You obtain a detailed plan showing all the pieces you need to cut.
2. You cut all the pieces.
3. You put them all together.
4. You finish them (varnish, paint, etc.).
5. You feel satisfaction as you behold your creation.

When I hear woodworking analogies in a software context, people are usually trying to make the point that without a detailed plan at the outset, a project has no hope of succeeding.

Often, though, the people who use woodworking analogies don’t realize that woodworking projects don’t really work this simple way, either. A whole host of details complicate even simple woodworking projects, just like a whole host of details complicate even your simple software projects. Here are some examples:

- It takes a lot of work to find (or make) a piece of wood that’s straight and true.
- Wood moves, even when it’s no longer alive. Its size and shape vary with the moisture content within it. Its moisture content changes with changes in climate; even the microclimate inside your home.

- Measurement devices capable of measuring distances longer than about 12 inches with great accuracy are *expensive*. (They're heavy too, and if you drop one, it's no good anymore.)
- Tape measures are inexpensive surrogates for accurate measurement devices, but they're accurate in many applications to no better than about 1/4 inch.
- Glue seams have non-zero widths, so the width of ten pieces glued together isn't exactly the sum of the widths of the individual pieces.

If you don't know all these things (and several more), then you can plan, cut, assemble, and finish, but you're not going to get great results.

In the woodshop, I don't know all the pitfalls yet. I never will. I can remember a few of the ones that have trapped me in the past, and I can create better practices to work around those, but what I need are practices I can use to prevent the mistakes that *I don't understand yet*.

That's where "measure once, cut twice" helps me.

Agile Development

With the woodworking experience I have today, I *know* not to cut everything at the beginning of my project. I cut a little, I test-fit, I trim, and then I assemble a little. Of course, I have to study ahead in my plan to know which parts I can permanently assemble—I don't want to have to cut my case apart after I've glued it because I discovered that I need to add a part inside it that I can't reach with it glued together. Then I cut a little more, test fit a little more, trim a little more, and assemble a little more.

In the software development side of my life, abiding by the same kind of construction philosophy is called *agile development*. For several years, I've been a fan of the work revealed to me through reading Kent Beck and Cynthia Andres's book *Extreme Programming Explained* [BECK2005]. Thinking about agile development values has improved many facets of my life.

Two ideas from agile development have made a particularly powerful improvement to the quality of my work, both in the shop and in my office. They map directly to the "measure once, cut twice" message of this paper. They are:

- Incremental design
- Weekly releases

In the following sections, I'll describe these two concepts and how applying them interdependently has improved the quality of my work and my life.

Incremental Design

For some projects, you really can't know your complete specification until after you have some experience *within that specific project.* Sometimes you just can't know everything you need to know up front. I've learned this over and over again, sometimes more painfully than others. Furthermore, I've grown to believe that pretending that you should (or even *can*) know everything up front is dangerous to the project.

I like Kent Beck's introduction to the notion of incremental design:

> The question is not whether or not to design, the question is *when* to design. Incremental design suggests that the most effective time to design is in the light of experience. [BECK2005, p52]

Incremental design works both in the shop and in the office. Let's look at a woodworking example, and then a computer software example.

Woodworking Example

Mom wants a custom cabinet installed to fit perfectly against her kitchen wall, where "perfection" is defined as "no gaps over 1/8 inch wide." Our first challenge is that her adjoining walls and floor aren't exactly perpendicular to each other.[1] Second, her wall is brick and mortar with a surface variance of up to 1/2 inch between the many high spots and low spots.

If you've never done a project like this before, your mind might wander to a high-tech solution. Maybe you could use digital tomography to capture an image of the walls to which you're going to be fitting your cabinets. Maybe then you could use some CAD program to transfer all those irregularities from your model into actual wood pieces that you'll use to build your cabinets.

Sounds cool. But that's not how people who *actually* do projects like this do it. The options that cabinet installers actually choose from include:

Caulk

Use caulk to fill whatever gaps exist between the wall's and baseboard's irregularities and your square cabinets. People do this, because it's easy. A small caulk bead to finish a uniform gap is just fine. But filling a half-inch wide gap with caulk is *nasty*. Over time, the caulk tends to shrink (people who do stuff like this also buy cheap caulk that shrinks). The shrinkage causes the caulk to break away from the

[1] This isn't unusual. There may be a home in the USA that has walls, floors, and ceilings that are perfectly perpendicular to each other. But if there is, I'm pretty sure I'll never be able to afford to visit it.

wall or the cabinet, leaving a gap along one side of the "caulk bridge," and it looks horrible.

Trim
: Nail down a thin flexible strip of wood trim to conceal the gap. This is what most cabinetmakers do in reasonably nice homes, and then they caulk the tiny gap that's left. But it won't work in Mom's kitchen, because there's too much irregularity in the surface of her walls. There's no way to bend a trim strip into a precise enough fit to snuggle up against Mom's walls.

Scribed trim
: *Scribe* a trim strip (or the cabinet itself) to custom-fit Mom's wall. This solution cannot be performed in your workshop—it has to be executed in Mom's kitchen, but it will produce a beautiful result with virtually no risk. Here's how you do it:

1. Affix the cabinet in its permanent position against Mom's wall.
2. Tape a trim piece to the outside wall of the cabinet so that its edge touches Mom's wall. Make sure the piece is square to the cabinet.
3. Use a scribing tool (Exhibit 1) to mark a jiggly line that exactly mimics the shape of the wall onto your trim piece.

Exhibit 1. A scribing tool being used to scribe a trim strip for a tricky surface. (Photo: Alex Millsap)

4. Take the trim piece off the cabinet, and cut along this jiggly line with a tool called a jigsaw, which you can use to cut jiggly lines really nicely with just a little bit of practice.
5. Reattach the trim piece, this time with the jiggly edge fitted tightly against Mom's wall, and nail it to the cabinet.
6. If you want, caulk the thin gap that remains between the trim piece and the wall.

What you end up with looks something like the small piece of trim that I scribed a few years ago for my parents' bookcase, shown in Exhibit 2.

Exhibit 2. A trim strip that I scribed to fit against a baseboard beneath my parents' bookcase.

In this example, scribing is a simple, elegant solution to a problem that would be prohibitively expensive (and prohibitively risky) to solve any earlier in the project. In the woodshop, you simply make peace with the notion that you can't plan all your cuts in detail (nor can you make all your cuts in your woodshop), and you include *in the installation plan* all the time and materials that you will need for custom-fitting the cabinets into Mom's kitchen.

Computer Software Example

For years at Oracle Corporation, I had a responsibility to respond to sales reps' questions about how much hardware their prospects needed to buy so they could implement Oracle. I learned over and over again that practically no amount of modeling could accurately enough answer the questions in many circumstances. Just like in Mom's kitchen, tomography is not the right answer.

For instance, two nearly identical companies could implement Oracle General Ledger with nearly identical sets of books, but if one company had staff that wanted to run 50 more Aged Trial Balance reports per day than the other company, their hardware requirements were going to be vastly different.

I tried over time to add to my list of questions I asked, but no matter what additional information I asked for, I could never get enough information to reduce the risk of still being really, really wrong. The process became black comedy. I was eventually asking for information so complex and so subtle that no human could possibly provide it, yet the quality of my answers wasn't materially better than if I had just asked how many users there were going to be, and what color the CIO's eyes were.

I eventually figured out that the hardware sizing exercise is best executed as a *process* that's integrated into the project plan *and that necessarily extends into the operational phase of the system*. When you require it to be a task that gets executed once and for all as a project prerequisite, you're executing a plan that just can't work for some projects.[2] Much like the cabinet scribing example, to implement a complex software application, you need to be aware that certain critical information just isn't going to be available until a later stage in the project. You need to realize that fact, and you need to *plan* for it.

I learned that the right software server capacity plan includes elements like these:

- Don't overbuy early. Buy a system that's expandable so you can grow it if you need to.
- Make sure you have a good relationship with your hardware vendors, so you can adjust what you buy as you learn more precise information about what you need.
- Make sure that before you go live, you have a way to measure how much operational headspace you'll need and how much you'll have. Make learning this a part of your plan.
- Realize that capacity planning is a whole lot easier if your software helps you measure its own weight upon the system. Instrument your code in such a manner that a business predicate like "we'll grow sales by 3x next year" is translatable into a technical expression of the amount of new hardware capacity you'll have to purchase to make it happen.
- Test the system to destruction before going live. Experience what it's going to look like as it begins to fail from too much load.

Design: How Much Detail is Right?

You have to do some *design* before you can create a *plan*. How much design you do influences both the *breadth* of your plan (how many different things does it cover?) and the *depth* of your plan (how deep does it go into each thing it does cover?). What level of detail does a good plan need? The "measure twice" philosophy encourages you to believe that the more detailed your plan is, the better off you'll be. The "cut twice" philosophy encourages you to believe that sometimes a less detailed plan is better.

[2] A once-and-done, up-front capacity plan lasts longer in some projects than it will in other projects. Likewise, when cabinet installers install into smooth, nearly-square walls, they can skip the scribing step of the installation.

How can a less-detailed plan be better?

It's easy to see the benefits of detail in a plan. More detail makes it easier to know exactly how much your project is going to cost and how much time it's going to take. But only if the plan is *correct*. And that's the catch. In many projects, especially the more customized ones, some details are prohibitively expensive to obtain early in the project. If you try to plan for all the details up front, the project either becomes prohibitively expensive, or you create an incorrect plan. Either option is unacceptable.

In the examples I've described so far, tomography was prohibitively expensive. But, if you insist upon making all your cuts in your shop, then doing anything less than tomography has little hope of working. But scribing—which gives you exactly the information you need, exactly when you need it—is both effective and delightfully inexpensive. You just can't do it at the beginning of your project. You have to build the cabinets and take them over to Mom's house first. I feel the same way now about capacity planning. The cost of knowing certain details decreases dramatically during the execution of your project.

Weekly Releases

What would you rather have this Friday: four half-done features? Or two fully-done features?

If you have four half-done features, you really don't have anything you can play with or gain confidence with. You really don't have anything that can benefit from the goodness of incremental design. All you have is a bunch of ideas that are maybe closer to being finished than they were last week. And maybe not even that: a feature can retrogress relative to its specification in a week's time when your spec changes faster than your code.

I'd rather have two finished features than four half-done ones. And that's the philosophy behind *weekly releases* in agile development. Give me something I can *use*. When I *use* something, I can either enjoy it and accumulate productivity with it, or I can learn that what I *really* want is actually different from what I *thought* I wanted. Either result is valuable.

In my woodshop last year, I built a machinist's chest to hold a bunch of small tools I own (calipers, scalpels, paint brushes, …stuff like that). I spent many of my leisure hours on airplanes and in hotel rooms cooking up a very detailed plan in Microsoft Visio (Exhibit 3). That works well for me: with a printed, detailed plan right in front of me, I can focus my attention when I'm in my shop on cutting, not designing, which is good.

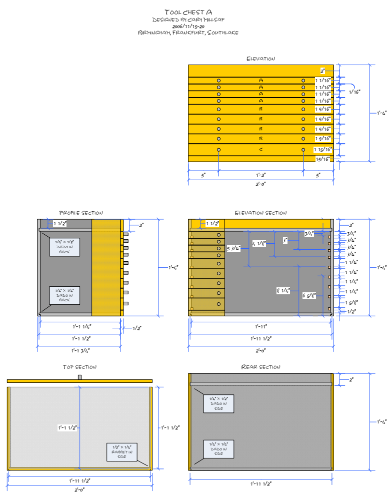
Exhibit 3. Cary Millsap's tool chest version 1.

I finished the chest, and I loaded it up with tools. The amount of stuff this thing held was phenomenal. With this new chest, I was able to condense a whole cabinet drawer, six square feet of pegboard, and about four square feet of countertop space into just *two* square feet of countertop space. Immediate benefit.

But there was one design feature that I didn't like. I used 3/8-inch square cleats to support the nine drawers. They're nice and strong, but nine 3/8-inch cleats consume 3-3/8 inches of drawer space. Since my whole chest is only 16 inches tall, that means that I'm wasting 21% of my vertical chest capacity for infrastructure used just to hold the drawers up. That started to bug me, because I still had even more stuff that I wanted to store in the chest, but all the drawers were full.

So, should I have spent more time designing the cabinet? If I had, I don't know whether I would ever have thought of a more efficient design, or if I would even have realized that I wanted one. One thing is for sure, though. If I had spent more time designing the chest, I would have spent more time in my shop without any chest at all. As it happened, my complete, fully operational, but imperfect chest radically improved my tool storage problem. *And* it inspired a design change for version 2 (Exhibit 4) that I really don't believe I would have ever thought of by looking at my paper model of version 1.

Certainly, inadequate design and planning are problems to be avoided. I have also come to view excessive up-front design work as a peril to be avoided, too.

The Perils of Excessive Up-Front Design

It is difficult to manage the balance between too much design and not enough. It seems ironic to me how expensive of a mistake too much up-front design can be.

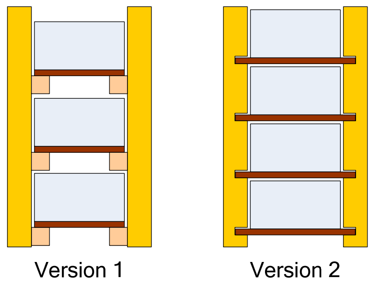

Exhibit 4. The key design difference between tool chest versions 1 and 2. Here is a front view of each chest, without drawer fronts, and with the width not drawn to scale. The v1 drawer bottoms slide atop cleats with a 3/8-inch square cross section. In v2, extra-wide drawer bottoms slide within slots cut into the case sides. The v2 design can hold more drawers in the same amount of space.

Let me return to the machinist's chest story for a moment. As I was contemplating building version 2 of the chest (exhibit 4), my biggest worry was whether the slots for the drawers would compromise the strength of the case sides. I spent a couple of weeks so concerned about this potential problem that I started designing a complicated reinforcement system. Fortunately, I stopped myself before I got too far. I decided simply to conduct a test. I cut several slots into a piece of cabinet-side material, and I tested its strength by hand. I learned conclusively that it's not a problem.

Here are some danger signs of the excessive up-front design problem:

- You have to guess what the "customer" for the new product will really want. You invent concerns that some unknown customer, somewhere in the unseeable future, might want to do X, and figuring out how to do X generates several hours of user interface design, application process design, and database schema design. In the absence of a real customer, designers tacitly begin to assume that perfection is an actual requirement.
- Meetings grow bigger and longer. You find yourself in three-hour meetings where the complexity is so bad that it's difficult even to remember all the details you cemented in place in the prior meeting. You invite more people to each meeting because it's impossible for fewer people to cover the breadth of the design. As a result, the design only gets more complicated, never simpler.
- The design gets even more complicated when you begin talking about integration issues that this project will have with other projects. It gets especially worse when you begin to design interfaces between

your product and ones you haven't even begun designing yet.

- There never seems to be anything runnable to get your hands on. The first date when you're actually going to produce something runnable is in the distant future. Before that can happen, many complex infrastructure subprojects (like database design, product installation scripts, etc.) will have to be completed. These subprojects haven't begun yet.
- Your design can ostensibly defend itself against all the contingencies your group has been able to think of, but the project somehow no longer feels like something for which anyone in the room will "feel satisfaction as you behold your creation."

If you're lucky, you can catch yourself before you invest too heavily into your journey down this road. I've found it invaluable to have two things to keep me out of the ditches:

- You need someone who can authoritatively say, "At this point, *all we need* is X. More than that is fine, eventually, but what we need *now* is just X." Most software doesn't need to be perfect. It just needs to do something better than the way people do it now.
- And you need a shared commitment to producing a sequence of "working prototypes" that inform your design process so that you can actually see (and show) periodic successes, and so you can stop guessing what you think people are going to want.

Act II: Software Performance

Application software developers have tremendous leverage over every aspect of software operational performance. That's a new idea for many Oracle practitioners, where we've been taught for years that database administrators (DBAs) are the sole guardians of application performance. But it's true: application developers have leverage over *every* aspect of software operational performance.

…It's up to them whether they seize the opportunity to use that leverage.

Scribing for Application Developers

The scribing example from cabinetry reminds me of something we also need from applications. There are some questions we simply can't answer early in the planning phases of a software development project, like these:

How fast will your code run?

How fast will it need to run?

How much load will your code exert upon the system as, for example, data volumes increase?

These are fundamental questions of capacity planning. If you don't know what your code is going to "weigh" when whoever's using the system will run it, then you have no hope of knowing how much hardware you're going to need to run it. Yet there are several reasons you won't know the answer until much later in your project. For example:

- There may be hundreds of distinct task types that your system will be required to execute. They'll possibly all have different "weights" (that is, execution durations and resource consumption profiles).
- What will be the *arrival rate* of execution requests for each task type in your system? Without this information, you won't be able to know the intensity of the workload on your system.
- What resources in your system will become bottlenecks because of peculiar usage patterns that you didn't anticipate? In the Oracle world, the bottlenecks that people seem never to expect are the *latch free* problems, the *buffer busy waits* problems, and the like.

It's tough being a software developer. Some aspects of your design simply cannot be known at compile time, but you're going to be held accountable for them at operational run-time anyway. So, how can you defend yourself?

The answer is easier to see after riding a few such projects through the tough times when there's a slow system and you're the one who has to fix it. First, let me describe one answer that *doesn't* work very well:

Bad practice: Analysis with system-wide scope
: Consulting system-wide statistics is an entrenched tradition in the Oracle market. The process begins with finding out which resource (CPU, memory, disk, this latch, that lock, etc.) the system uses most heavily. Different people make different choices once that "bottleneck resource" has been identified. Some favor upgrading the resource. Some favor looking for SQL that dominates the consumption of that resource.

I've been pretty thorough in documenting my opinion about why I believe this behavior is an antipattern (that is, a practice that is *bad*). See especially [MILL2003], [MILL2004a], [MILL2006b], and [MILL2007] for more information.

Method R

In contrast, a problem-solving process that I've found *does* work very well is a business task-focused approach:

1. Identify which misbehaving business task you want to fix next.

2. Collect detailed statistics that can explain exactly how that task spent its time, specifically while that task is misbehaving.
3. Either reduce the task's "weight" (time spent, resources consumed) or show that the task is already as efficient as you can afford to make it.
4. If you still have performance problems, then go to step 1.

These steps restate what Jeff Holt and I call Method R in [MILL2003, MILL2006a]. Method R works like a charm in situations where the expression of the performance problem suits the way step 1 is expressed. For example, here's a scenario in which Method R is the perfect way to get moving:

> It's the Payroll process PYUGEN that's killing us. Our inability to complete that process on time is causing us to miss our payroll processing deadlines, which means that we can't pay people on Fridays like we're supposed to. It's killing morale, and it's even fueling episodes of vandalism against the company. *Fix that Payroll process!*

In this case, Method R works well because it mirrors the way that system owners want to work.

In other circumstances, people steer away from Method R because they feel unable or unwilling to implement step 1. Many DBAs (remember, the *guardians* of performance at many implementations) lack intimate contact with the actual business they work for. Some DBAs are frustrated by this detachment. Others believe that detachment from business people and business processes is the way DBAs *should* work—"Just show me the system, and I'll figure out what's wrong with it. Don't waste my time making me talk to users."

People who are enthusiastic about Method R perceive the inability or unwillingness to identify a prioritized list of problem tasks as a political problem. However, sometimes the right technology can dissolve a political problem. And here's where developers enter the performance picture. Imagine a world where it's easy to see how any important task within an application is performing…

Better Measurement, Better Management

Think of a mature system you use today. Chances are good that it's not always as fast as you'd like it to be. What happens when part of the application is unacceptably slow? Does the duration of the task you're running get recorded somewhere? Is there a button on your application form that you can click to obtain some kind of relief when you're having a performance problem? Do you call someone?

Chances are good that you don't have any of these things, and you only pick up the phone when the situation is so grave that maybe it will be worth the effort to initiate a conversation about it with someone instead of doing what you would normally be doing. But software *can* keep track of its own performance—in terms that would be meaningful to its end users. And it *can* provide detailed diagnostic data when an analyst needs to execute step 2 of Method R.[3]

…If the application developer built it in.

It may be impossible for an application developer to guess how long a button click will take to respond next year, in an application he's writing today. But it's easy enough for that developer to write code into his application to record exactly when those clicks and their associated fulfillments will take place. It's easy enough for that developer to identify those click and fulfillment timings with a business task label that makes sense to a user of the system:

```
2004-06-15T12:35:25.688913 +Enter order
2004-06-15T12:35:37.289525 –Enter order
```

And it's easy enough for that developer to create an application feature that collects more detailed performance diagnostic data when it's needed—not all the time, because that might cause its own performance penalties—but when it's needed.

These things are easy enough to do, if there's motive to do them. Motive, of course, is a function of perceived benefit and perceived cost. To increase the motive to a useful level, you have to work both sides of the equation. That's what I'm trying to do today, because I'm committed to the idea that people—programmers, analysts, DBAs, users, and system owners alike—need to see performance of individual user experiences in plain terms that are easy to understand.

To make it easier for developers to add the right code to their applications without doing a lot of extra work, we've brought you the open-source ILO project [HOTS2006]. My team has also created a set of data collectors that automatically extract performance data from closed-source Oracle applications containing no performance instrumentation. We've proven that application performance instrumentation is possible even without application developer participation.

The benefit of having performance instrumentation (whether it's built-in or built-on) is the ability to answer questions like these, definitively, without guessing:

> How long does it take to run X? Does it vary by day? Time of day? Time of year? Does it always run longer when we run Y at the same time?
>
> Why does X take the time it does? Is it possible to make X any more efficient? When it runs longer than normal sometimes, what is the cause?

[3] Step 2 is, "Collect detailed statistics that can explain exactly how that task spent its time, specifically while that task is misbehaving."

How would the response time of X change if we performed some tuning operation Y upon the code? Or if we upgraded some resource Z?

How many more users of X could this system take on without violating our users' performance expectations?

I believe that applications that will answer these kinds of questions in plain, simple language are the Oracle performance management applications of the future. I believe that when task-wise instrumentation enters the mainstream of consciousness in our industry, it will improve forever the quality of performance in the applications we all use.

Summary

The following points summarize the intent of this paper:

- Some key principles of *agile development* have been of enormous value to me in my software development business, in my woodshop, and many other aspects of my life.
- In many projects—even surprisingly simple ones—understanding the benefits of *incremental design* can radically reduce project waste by permitting some design questions to go unanswered until a later phase when the answers become considerably less expensive (and less risky) to obtain.
- *Weekly releases* help keep projects on track by continually providing *something* that works and that people can get their hands on.
- Weekly releases are essential enablers of incremental design. They promote the design process from the theory world into the real world where meaningful improvements (even simplifications!) are more likely to occur.
- Likewise, too much up-front design can be deadly. Perfectionist tendencies plus an absence of a real "customer" in the design process equals disaster.
- Certain questions that developers need to answer at compile-time but cannot are easy to obtain at operational run-time, but most applications aren't programmed to obtain those answers at run-time.
- The absence of sensible task-oriented performance data in applications heightens performance analysts' motivation to use unreliable, inefficient performance improvement methods.
- When developers do instrument their code to record run-time performance data, it makes the resulting application easier to optimize, easier to repair, easier to plan for, and—ultimately—easier to use.

References and Further Reading

[BECK2005] K. Beck and C. Andres, *Extreme Programming Explained, Second Edition: Embrace Change*. Boston: Addison-Wesley, 2005.

[HOTS2006] Hotsos, "Hotsos Oracle Instrumentation Library" at *sourceforge.net/projects/hotsos-ilo*, 2006.

[MILL2003] C. Millsap and J. Holt, *Optimizing Oracle Performance*. Sebastopol CA: O'Reilly, 2003. (Chapter 1 available at www.oreilly.com/catalog/-optoraclep/chapter/index.html.

[MILL2004a] C. Millsap, "Why 'system' is a four-letter word" in *Toronto Oracle User Group* presentation, *www.hotsos.com*, 2004.

[MILL2004b] C. Millsap, "Profiling Oracle: how it works" at *www.hotsos.com*, 2004.

[MILL2005] C. Millsap, "How to make an application easy to diagnose" at *www.hotsos.com*, 2005.

[MILL2006a] C. Millsap. "Questioning Method R," at *www.hotsos.com*, 2006.

[MILL2006b] C. Millsap and G. Jósepsson, "Accountability for system performance" at *www.hotsos.com*, 2006.

[MILL2007] C. Millsap, "Why you can't see you real performance problems" at *www.hotsos.com*, 2007.