Chapter 3

Value of Accurate Estimates

[The common definition of estimate is] "the most optimistic prediction that has a non-zero probability of coming true." ... Accepting this definition leads irrevocably toward a method called what's-the-earliest-date-by-which-you-can't-prove-you-won't-be-finished estimating.

–Tom DeMarco

The inaccuracy of software project estimates—as muddied by unrealistic targets and unachievable commitments—has been a problem for many years. In the 1970s, Fred Brooks pointed out that "more software projects have gone awry for lack of calendar time than all other causes combined" (Brooks 1975). A decade later, Scott Costello observed that "deadline pressure is the single greatest enemy of software engineering" (Costello 1984). In the 1990s, Capers Jones reported that "excessive or irrational schedules are probably the single most destructive influence in all of software" (Jones 1994, 1997).

Tom DeMarco wrote his common definition of an estimate in 1982. Despite the successes I mentioned in the first chapter, not much has changed in the years since he wrote that definition. You might already agree that accurate estimates are valuable. This chapter details the specific benefits of accurate estimates and provides supporting data for them.

3.1 Is It Better to Overestimate or Underestimate?

Intuitively, a perfectly accurate estimate forms the ideal planning foundation for a project. If the estimates are accurate, work among different developers can be coordinated efficiently. Deliveries from one development group to another can be planned to the day, hour, or minute. We know that accurate estimates are rare, so if we're going to err, is it better to err on the side of overestimation or underestimation?

Arguments Against Overestimation

Managers and other project stakeholders sometimes fear that, if a project is overestimated, Parkinson's Law will kick in—the idea that work will expand to fill available time. If you give a developer 5 days to deliver a task that could be completed in 4 days, the developer will find something to do with the extra day. If you give a project team 6 months to complete a project that could be completed in 4 months, the project team will find a way to use up the extra 2 months. As a result, some managers consciously squeeze the estimates to try to avoid Parkinson's Law.

Another concern is Goldratt's "Student Syndrome" (Goldratt 1997). If developers are given too much time, they'll procrastinate until late in the project, at which point they'll rush to complete their work, and they probably won't finish the project on time.

A related motivation for underestimation is the desire to instill a sense of urgency in the development team. The line of reason goes like this:

> *The developers say that this project will take 6 months. I think there's some padding in their estimates and some fat that can be squeezed out of them. In addition, I'd like to have some schedule urgency on this project to force prioritizations among features. So I'm going to insist on a 3-month schedule. I don't really believe the project can be completed in 3 months, but that's what I'm going to present to the developers. If I'm right, the developers might deliver in 4 or 5 months. Worst case, the developers will deliver in the 6 months they originally estimated.*

Are these arguments compelling? To determine that, we need to examine the arguments in favor of erring on the side of overestimation.

Arguments Against Underestimation

Underestimation creates numerous problems—some obvious, some not so obvious.

Reduced effectiveness of project plans Low estimates undermine effective planning by feeding bad assumptions into plans for specific activities. They can cause planning errors in the team size, such as planning to use a team that's smaller than it should be. They can undermine the ability to coordinate among groups—if the groups aren't ready when they said they would be, other groups won't be able to integrate with their work.

If the estimation errors caused the plans to be off by only 5% or 10%, those errors wouldn't cause any significant problems. But numerous studies have found that software estimates are often inaccurate by 100% or more (Lawlis, Flowe, and Thordahl 1995; Jones 1998; Standish Group 2004; ISBSG 2005). When the planning assumptions are wrong by this magnitude, the average project's plans are based on assumptions that are so far off that the plans are virtually useless.

Statistically reduced chance of on-time completion Developers typically estimate 20% to 30% lower than their actual effort (van Genuchten 1991). Merely using their normal estimates makes the project plans optimistic. Reducing their estimates even further simply reduces the chances of on-time completion even more.

Poor technical foundation leads to worse-than-nominal results A low estimate can cause you to spend too little time on upstream activities such as requirements and design. If you don't put enough focus on requirements and design, you'll get to redo your requirements and redo your design later in the project—at greater cost than if you'd done those activities well in the first place (Boehm and Turner 2004, McConnell 2004a). This ultimately makes your project take longer than it would have taken with an accurate estimate.

Destructive late-project dynamics make the project worse than nominal Once a project gets into "late" status, project teams engage in numerous activities that they don't need to engage in during an "on-time" project. Here are some examples:

- More status meetings with upper management to discuss how to get the project back on track.
- Frequent reestimation, late in the project, to determine just when the project will be completed.
- Apologizing to key customers for missing delivery dates (including attending meetings with those customers).
- Preparing interim releases to support customer demos, trade shows, and so on. If the software were ready on time, the software itself could be used, and no interim release would be necessary.
- More discussions about which requirements absolutely must be added because the project has been underway so long.
- Fixing problems arising from quick and dirty workarounds that were implemented earlier in response to the schedule pressure.

The important characteristic of each of these activities is that they don't need to occur *at all* when a project is meeting its goals. These extra activities drain time away from productive work on the project and make it take longer than it would if it were estimated and planned accurately.

Weighing the Arguments

Goldratt's Student Syndrome can be a factor on software projects, but I've found that the most effective way to address Student Syndrome is through active task tracking and buffer management (that is, project control), similar to what Goldratt suggests, not through biasing the estimates.

As Figure 3-1 shows, the best project results come from the most accurate estimates (Symons 1991). If the estimate is too low, planning inefficiencies will drive up the actual cost and schedule of the project. If the estimate is too high, Parkinson's Law kicks in.

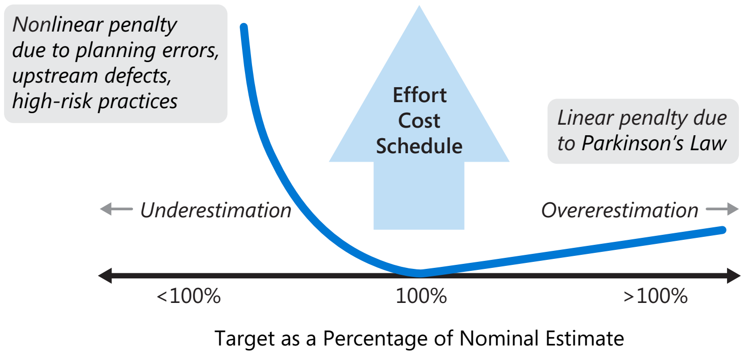
Nonlinear penalty due to planning errors, upstream defects, high-risk practices
Effort
Cost
Schedule
Linear penalty due to Parkinson's Law
Underestimation
Overerestimation
<100%
100%
>100%
Target as a Percentage of Nominal Estimate

Figure 3-1 The penalties for underestimation are more severe than the penalties for overestimation, so, if you can't estimate with complete accuracy, try to err on the side of overestimation rather than underestimation.

I believe that Parkinson's Law does apply to software projects. Work does expand to fill available time. But deliberately underestimating a project because of Parkinson's Law makes sense only if the penalty for overestimation is worse than the penalty for underestimation. In software, the penalty for overestimation is *linear and bounded*—work will expand to fill available time, but it will not expand any further. But the penalty for underestimation is *nonlinear and unbounded*—planning errors, shortchanging upstream activities, and the creation of more defects cause more damage than overestimation does, and with little ability to predict the extent of the damage ahead of time.

Tip #8 Don't intentionally underestimate. The penalty for underestimation is more severe than the penalty for overestimation. Address concerns about overestimation through planning and control, not by biasing your estimates.

3.2 Details on the Software Industry's Estimation Track Record

The software industry's estimation track record provides some interesting clues to the nature of software's estimation problems. In recent years, The Standish Group has published a biennial survey called "The Chaos Report," which describes software project outcomes. In the 2004 report, 54% of projects were delivered late, 18% failed outright, and 28% were delivered on time and within budget. Figure 3-2 shows the results for the 10 years from 1994 to 2004.

What's notable about The Standish Group's data is that it doesn't even have a category for early delivery! The best possible performance is meeting expectations "On Time/On Budget"—and the other options are all downhill from there.

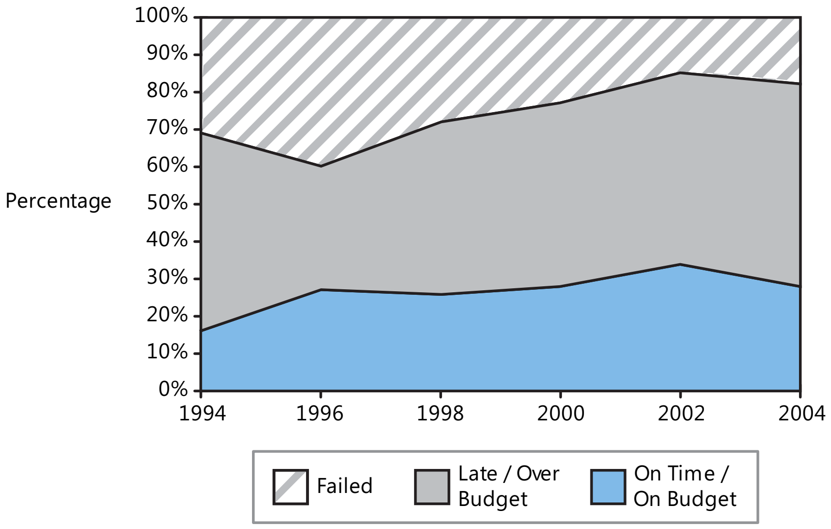

Figure 3-2 Project outcomes reported in The Standish Group's Chaos report have fluctuated year to year. About three quarters of all software projects are delivered late or fail outright.

Capers Jones presents another view of project outcomes. Jones has observed for many years that project success depends on project size. That is, larger projects struggle more than smaller projects do. Table 3-1 illustrates this point.

Table 3-1 **Project Outcomes by Project Size**

Size in Function Points (and Approximate Lines of Code)	Early	On Time	Late	Failed (Canceled)
10 FP (1,000 LOC)	11%	81%	6%	2%
100 FP (10,000 LOC)	6%	75%	12%	7%
1,000 FP (100,000 LOC)	1%	61%	18%	20%
10,000 FP (1,000,000 LOC)	<1%	28%	24%	48%
100,000 FP (10,000,000 LOC)	0%	14%	21%	65%

Source: *Estimating Software Costs* (Jones 1998).

As you can see from Jones's data, the larger a project, the less chance the project has of completing on time and the greater chance it has of failing outright.

Overall, a compelling number of studies have found results in line with the results reported by The Standish Group and Jones, that about one quarter of all projects are delivered on time; about one quarter are canceled; and about half are delivered late, over budget, or both (Lederer and Prasad 1992; Jones 1998; ISBSG 2001; Krasner 2003; Putnam and Myers 2003; Heemstra, Siskens and van der Stelt 2003; Standish Group 2004).

The reasons that projects miss their targets are manifold. Poor estimates are one reason but not the only reason. We'll discuss the reasons in depth in Chapter 4, "Where Does Estimation Error Come From?"

How Late Are the Late Projects?

The number of projects that run late or over budget is one consideration. The degree to which these projects miss their targets is another consideration. According to the first Standish Group survey, the average project schedule overrun was about 120% and the average cost overrun was about 100% (Standish Group 1994). But the estimation accuracy is probably worse than those numbers reflect. The Standish Group found that late projects routinely threw out significant amounts of functionality to achieve the schedules and budgets they eventually did meet. Of course, these projects' estimates weren't for the abbreviated versions they eventually delivered; they were for the originally specified, full-featured versions. If these late projects had delivered all of their originally specified functionality, they would have overrun their plans even more.

One Company's Experience

A more company-specific view of project outcomes is shown in the data reported by one of my clients in Figure 3-3.

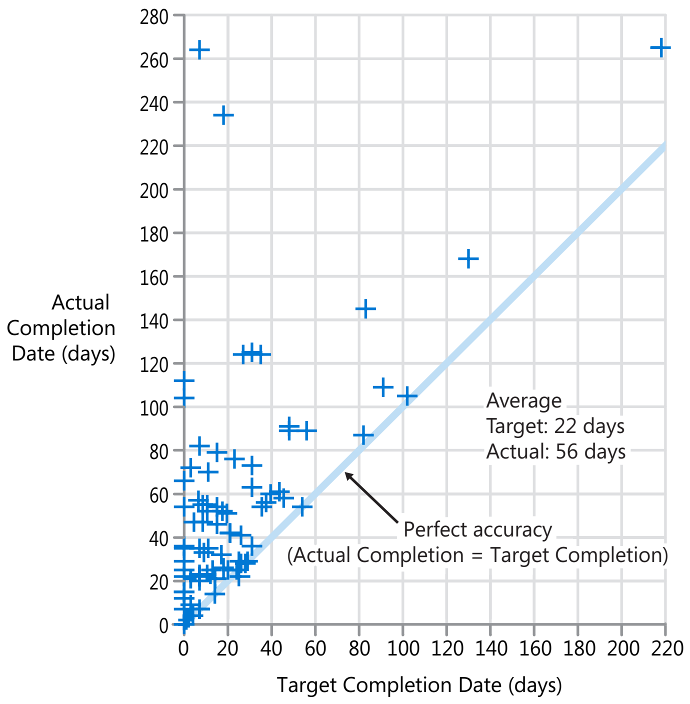

Figure 3-3 Estimation results from one organization. General industry data suggests that this company's estimates being about 100% low is typical. Data used by permission.

The points that are clustered on the "0" line on the left side of the graph represent projects for which the developers reported that they were done but which were found not to be complete when the software teams began integrating their work with other groups.

The diagonal line represents perfect scheduling accuracy. Ideally, the graph would show data points clustering tightly around the diagonal line. Instead, nearly all of the 80 data points shown are above the line and represent project overruns. One point is below the line, and a handful of points are on the line. The line illustrates DeMarco's common definition of an "estimate"—the earliest date by which you could possibly be finished.

The Software Industry's Systemic Problem

We often speak of the software industry's estimation problem as though it were a neutral estimation problem—that is, sometimes we overestimate, sometimes we underestimate, and we just can't get our estimates right.

But the software does not have a neutral estimation problem. The industry data shows clearly that *the software industry has an underestimation problem.* Before we can make our estimates more accurate, we need to start making the estimates *bigger*. That is the key challenge for many organizations.

3.3 Benefits of Accurate Estimates

Once your estimates become accurate enough that you get past worrying about large estimation errors on either the high or low side, truly accurate estimates produce additional benefits.

Improved status visibility One of the best ways to track progress is to compare planned progress with actual progress. If the planned progress is realistic (that is, based on accurate estimates), it's possible to track progress according to plan. If the planned progress is fantasy, a project typically begins to run without paying much attention to its plan and it soon becomes meaningless to compare actual progress with planned progress. Good estimates thus provide important support for project tracking.

Higher quality Accurate estimates help avoid schedule-stress-related quality problems. About 40% of all software errors have been found to be caused by stress; those errors could have been avoided by scheduling appropriately and by placing less stress on the developers (Glass 1994). When schedule pressure is extreme, about four times as many defects are reported in the released software as are reported for software developed under less extreme pressure (Jones 1994). One reason is that teams implement quick-and-dirty versions of features that absolutely must be completed in time to release the software. Excessive schedule pressure has also been found to be the most significant cause of extremely costly error-prone modules (Jones 1997).

Projects that aim from the beginning to have the lowest number of defects usually also have the shortest schedules (Jones 2000). Projects that apply pressure to create unrealistic estimates and subsequently shortchange quality are rudely awakened when they discover that they have also shortchanged cost and schedule.

Better coordination with nonsoftware functions Software projects usually need to coordinate with other business functions, including testing, document writing, marketing campaigns, sales staff training, financial projections, software support training, and so on. If the software schedule is not reliable, that can cause related functions to slip, which can cause the *entire project schedule* to slip. Better software estimates allow for tighter coordination of the whole project, including both software and nonsoftware activities.

Better budgeting Although it is almost too obvious to state, accurate estimates support accurate budgets. An organization that doesn't support accurate estimates undermines its ability to forecast the costs of its projects.

Increased credibility for the development team One of the great ironies in software development is that after a project team creates an estimate, managers, marketers, and sales staff take the estimate and turn it into an optimistic business target—over the objections of the project team. The developers then overrun the optimistic business target, at which point, managers, marketers, and sales staff blame the developers for being poor estimators! A project team that holds its ground and insists on an accurate estimate will improve its credibility within its organization.

Early risk information One of the most common wasted opportunities in software development is the failure to correctly interpret the meaning of an initial mismatch between project goals and project estimates. Consider what happens when the business sponsor says, "This project needs to be done in 4 months because we have a major trade show coming up," and the project team says, "Our best estimate is that this project will take 6 months." The most typical interaction is for the business sponsor and the project leadership to negotiate the *estimate*, and for the project team eventually to be pressured into committing to try to achieve the 4-month schedule.

Bzzzzzt! Wrong answer! The detection of a mismatch between the project goal and the project estimate should be interpreted as incredibly useful, incredibly rare, early-in-the-project risk information. The mismatch indicates a substantial chance that the project will fail to meet its business objective. Detected early, numerous corrective actions are available, and many of them are high leverage. You might redefine the scope of the project, you might increase staff, you might transfer your best staff onto the project, or you might stagger the delivery of different functionality. You might even decide the project is not worth doing after all.

But if this mismatch is allowed to persist, the options that will be available for corrective action will be far fewer and will be much lower leverage. The options will generally consist of "overrun the schedule and budget" or "cut painful amounts of functionality."

Tip #9 Recognize a mismatch between a project's business target and a project's estimate for what it is: valuable risk information that the project might not be successful. Take corrective action early, when it can do some good.

3.4 Value of Predictability Compared with Other Desirable Project Attributes

Software organizations and individual software projects try to achieve numerous objectives for their projects. Here are some of the goals they strive for:

- **Schedule** Shortest possible schedule for the desired functionality at the desired quality level
- **Cost** Minimum cost to deliver the desired functionality in the desired time
- **Functionality** Maximum feature richness for the time and money available

Projects will prioritize these generic goals as well as more specific goals differently. Agile development tends to focus on the goals of flexibility, repeatability, robustness, sustainability, and visibility (Cockburn 2001, McConnell 2002). The SEI's CMM tends to focus on the goals of efficiency, improvability, predictability, repeatability, and visibility.

In my discussions with executives, I've frequently asked, "What is more important to you: the ability to change your mind about features, or the ability to know cost, schedule, and functionality in advance?" At least 8 times out of 10, executives respond "The ability to know cost, schedule, and functionality in advance"—in other words, *predictability*. Other software experts have made the same observation (Moseman 2002, Putnam and Myers 2003).

I often follow up by saying, "Suppose I could offer you project results similar to either Option #1 or Option #2 in Figure 3-4. Let's suppose Option #1 means that I can deliver a project with an expected duration of 4 months, but it might be 1 month early and it might be as many as 4 months late. Let's suppose Option #2 means that I can deliver a project with an expected duration of 5 months (rather than 4), and I can guarantee that it will be completed within a week of that date. Which would you prefer?"

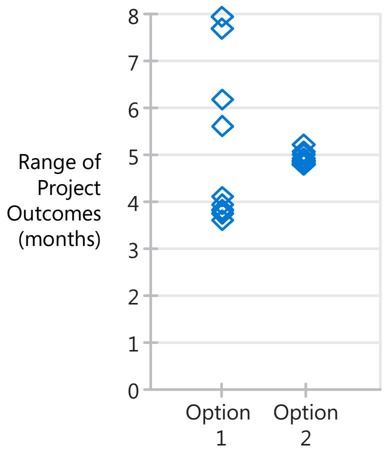

Figure 3-4 When given the option of a shorter average schedule with higher variability or a longer average schedule with lower variability, most businesses will choose the second option.

In my experience, nearly all executives will choose Option #2. The shorter schedule offered by Option #1 won't do the business any good because the business can't depend on it. Because the overrun could easily be as large as 4 months, the business has to plan on an 8-month schedule rather than a 4-month schedule. Or it delays making any plans at all until the software is actually ready. In comparison, the guaranteed 5-month schedule of Option #2 looks much better.

Over the years, the software industry has focused on time to market, cost, and flexibility. Each of these goals is desirable, but what top executives usually value most is predictability. Businesses need to make commitments to customers, investors, suppliers, the marketplace, and other stakeholders. These commitments are all supported by predictability.

None of this proves that predictability is the top priority for your business, but it suggests that you shouldn't make assumptions about your business's priorities.

Tip #10

Many businesses value predictability more than development time, cost, or flexibility. Be sure you understand what your business values the most.

3.5 Problems with Common Estimation Techniques

Considering the widespread poor results from software estimation, it shouldn't be a surprise that the techniques used to produce the estimates are not effective. These techniques should be carefully examined and thrown out!

Albert Lederer and Jayesh Prasad found that the most commonly used estimation technique was comparing a new project with a similar past project, based solely on personal memory. This technique was not found to be correlated with accurate estimates. The common techniques of "intuition" and "guessing" were found to be correlated with cost and schedule overruns (Lederer and Prasad 1992). Numerous other researchers have found that guessing, intuition, unstructured expert judgment, use of informal analogies, and similar techniques are the dominant strategies used for about 60 to 85% of all estimates (Hihn and Habib-Agahi 1991, Heemstra and Kusters 1991, Paynter 1996, Jørgensen 2002, Kitchenham et al. 2002).

Chapter 5, "Estimate Influences," presents a more detailed examination of sources of estimation error, and the rest of this book provides alternatives to these common techniques.

Additional Resources

Goldratt, Eliyahu M. *Critical Chain*. Great Barrington, MA: The North River Press, 1997. Goldratt describes an approach to dealing with Student Syndrome as well as an approach to buffer management that addresses Parkinson's Law.

Putnam, Lawrence H. and Ware Myers. *Five Core Metrics*. New York, NY: Dorset House, 2003. Chapter 4 contains an extended discussion of the importance of predictability compared to other project objectives.

Chapter 4

Where Does Estimation Error Come From?

There's no point in being exact about something if you don't even know what you're talking about.

–John von Neumann

A University of Washington Computer Science Department project was in serious estimation trouble. The project was months late and $20.5 million over budget. The causes ranged from design problems and miscommunications to last-minute changes and numerous errors. The university argued that the plans for the project weren't adequate. But this wasn't an ordinary software project. In fact, it wasn't a software project at all; it was the creation of the university's new Computer Science and Engineering Building (Sanchez 1998).

Software estimation presents challenges because estimation itself presents challenges. The Seattle Mariners' new baseball stadium was estimated in 1995 to cost $250 million. It was finally completed in 1999 at a cost of $517 million–an estimation error of more than 100% (Withers 1999). The most massive cost overrun in recent times was probably Boston's Big Dig highway construction project. Originally estimated to cost $2.6 billion, costs eventually totaled about $15 billion–an estimation error of more than 400% (Associated Press 2003).

Of course, the software world has its own dramatic estimation problems. The Irish Personnel, Payroll and Related Systems (PPARS) system was cancelled after it overran its €8.8 million system by €140 million (The Irish Times 2005). The FBI's Virtual Case File (VCF) project was shelved in March 2005 after costing $170 million and delivering only one-tenth of its planned capability (Arnone 2005). The software contractor for VCF complained that the FBI went through 5 different CIOs and 10 different project managers, not to mention 36 contract changes (Knorr 2005). Background chaos like that is not unusual in projects that have experienced estimation problems.

A chapter on sources of estimation error might just as well be titled "Classic Mistakes in Software Estimation." Merely avoiding the problems identified in this chapter will get you halfway to creating accurate estimates.

Estimation error creeps into estimates from four generic sources:

- Inaccurate information about the project being estimated

- Inaccurate information about the capabilities of the organization that will perform the project
- Too much chaos in the project to support accurate estimation (that is, trying to estimate a moving target)
- Inaccuracies arising from the estimation process itself

This chapter describes each source of estimation error in detail.

4.1 Sources of Estimation Uncertainty

How much does a new house cost? It depends on the house. How much does a Web site cost? It depends on the Web site. Until each specific feature is understood in detail, it's impossible to estimate the cost of a software project accurately. It isn't possible to estimate the amount of work required to build something when that "something" has not been defined.

Software development is a process of gradual refinement. You start with a general product concept (the vision of the software you intend to build), and you refine that concept based on the product and project goals. Sometimes your goal is to estimate the budget and schedule needed to deliver a specific amount of functionality. Other times your goal is to estimate how much functionality can be built in a predetermined amount of time under a fixed budget. Many projects navigate under a happy medium of some flexibility in budget, schedule, and features. In any of these cases the different ways the software could ultimately take shape will produce widely different combinations of cost, schedule, and feature set.

Suppose you're developing an order-entry system and you haven't yet pinned down the requirements for entering telephone numbers. Some of the uncertainties that could affect a software estimate from the requirements activity through release include the following:

- When telephone numbers are entered, will the customer want a Telephone Number Checker to check whether the numbers are valid?
- If the customer wants the Telephone Number Checker, will the customer want the cheap or expensive version of the Telephone Number Checker? (There are typically 2-hour, 2-day, and 2-week versions of any particular feature—for example, U.S.-only versus international phone numbers.)
- If you implement the cheap version of the Telephone Number Checker, will the customer later want the expensive version after all?
- Can you use an off-the-shelf Telephone Number Checker, or are there design constraints that require you to develop your own?

- How will the Telephone Number Checker be designed? (Typically there is at least a factor of 10 difference in design complexity among different designs for the same feature.)
- How long will it take to code the Telephone Number Checker? (There can be a factor of 10 difference—or more—in the time that different developers need to code the same feature.)
- Do the Telephone Number Checker and the Address Checker interact? How long will it take to integrate the Telephone Number Checker and the Address Checker?
- What will the quality level of the Telephone Number Checker be? (Depending on the care taken during implementation, there can be a factor of 10 difference in the number of defects contained in the original implementation.)
- How long will it take to debug and correct mistakes made in the implementation of the Telephone Number Checker? (Individual performance among different programmers with the same level of experience varies by at least a factor of 10 in debugging and correcting the same problems.)

As you can see just from this short list of uncertainties, potential differences in how a single feature is specified, designed, and implemented can introduce cumulative differences of a hundredfold or more in implementation time for any given feature. When you combine these uncertainties across hundreds or thousands of features in a large feature set, you end up with significant uncertainty in the project itself.

4.2 The Cone of Uncertainty

Software development consists of making literally thousands of decisions about all the feature-related issues described in the previous section. Uncertainty in a software estimate results from uncertainty in how the decisions will be resolved. As you make a greater percentage of those decisions, you reduce the estimation uncertainty.

As a result of this process of resolving decisions, researchers have found that project estimates are subject to predictable amounts of uncertainty at various stages. The Cone of Uncertainty in Figure 4-1 shows how estimates become more accurate as a project progresses. (The following discussion initially describes a sequential development approach for ease of explanation. The end of this section will explain how to apply the concepts to iterative projects.)

The horizontal axis contains common project milestones, such as Initial Concept, Approved Product Definition, Requirements Complete, and so on. Because of its origins, this terminology sounds somewhat product-oriented. “Product Definition” just refers to the agreed-upon vision for the software, or the *software concept*, and applies equally to Web services, internal business systems, and most other kinds of software projects.

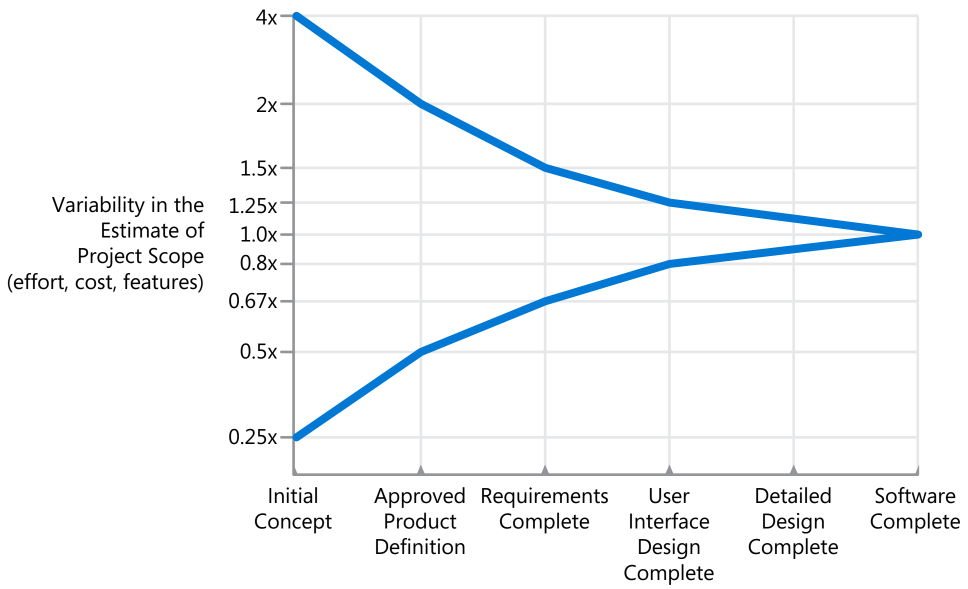

Figure 4-1 The Cone of Uncertainty based on common project milestones.

The vertical axis contains the degree of error that has been found in estimates created by skilled estimators at various points in the project. The estimates could be for how much a particular feature set will cost and how much effort will be required to deliver that feature set, or it could be for how many features can be delivered for a particular amount of effort or schedule. This book uses the generic term *scope* to refer to project size in effort, cost, features, or some combination thereof.

As you can see from the graph, estimates created very early in the project are subject to a high degree of error. Estimates created at Initial Concept time can be inaccurate by a factor of 4*x* on the high side or 4*x* on the low side (also expressed as 0.25*x*, which is just 1 divided by 4). The total range from high estimate to low estimate is 4*x* divided by 0.25*x*, or 16*x*!

One question that managers and customers ask is, "If I give you another week to work on your estimate, can you refine it so that it contains less uncertainty?" That's a reasonable request, but unfortunately it's not possible to deliver on that request. Research by Luiz Laranjeira suggests that the accuracy of the software estimate depends on the level of refinement of the software's definition (Laranjeira 1990). The more refined the definition, the more accurate the estimate. The reason the estimate contains variability is that the software project itself contains variability. The only way to reduce the variability in the estimate is to reduce the variability in the project.

One misleading implication of this common depiction of the Cone of Uncertainty is that it looks like the Cone takes forever to narrow—as if you can't have very good estimation accuracy until you're nearly done with the project. Fortunately, that impression is created because the milestones on the horizontal axis are equally spaced, and we naturally assume that the horizontal axis is calendar time.

In reality, the milestones listed tend to be front-loaded in the project's schedule. When the Cone is redrawn on a calendar-time basis, it looks like Figure 4-2.

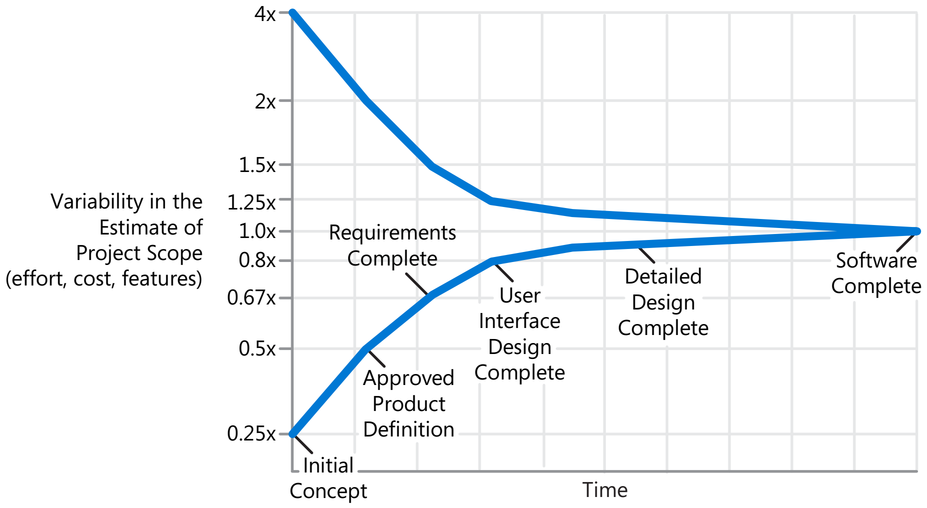

Figure 4-2 The Cone of Uncertainty based on calendar time. The Cone narrows much more quickly than would appear from the previous depiction in Figure 4-1.

As you can see from this version of the Cone, estimation accuracy improves rapidly for the first 30% of the project, improving from ±4*x* to ±1.25*x*.

Can You Beat the Cone?

An important—and difficult—concept is that the Cone of Uncertainty represents the *best-case accuracy* that is possible to have in software estimates at different points in a project. The Cone represents the error in estimates created by skilled estimators. It's easily possible to do worse. It isn't possible to be more accurate; it's only possible to be more lucky.

Tip #11 Consider the effect of the Cone of Uncertainty on the accuracy of your estimate. Your estimate cannot have more accuracy than is possible at your project's current position within the Cone.

The Cone Doesn't Narrow Itself

Another way in which the Cone of Uncertainty represents a best-case estimate is that if the project is not well controlled, or if the estimators aren't very skilled, estimates can fail to improve. Figure 4-3 shows what happens when the project doesn't focus on reducing variability—the uncertainty isn't a Cone, but rather a Cloud that persists to the end of the project. The issue isn't really that the estimates don't converge; the issue is that the project itself doesn't converge—that is, it doesn't drive out enough variability to support more accurate estimates.

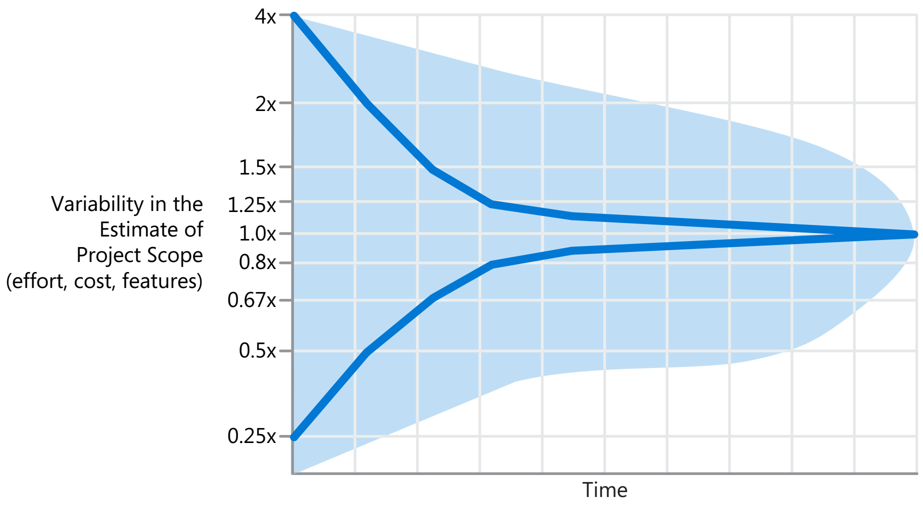

Figure 4-3 If a project is not well controlled or well estimated, you can end up with a Cloud of Uncertainty that contains even more estimation error than that represented by the Cone.

The Cone narrows only as you make decisions that eliminate variability. As Figure 4-4 illustrates, defining the product vision (including committing to what you will *not* do) reduces variability. Defining requirements—again, including what you are *not* going to do—eliminates variability further. Designing the user interface helps to reduce the risk of variability arising from misunderstood requirements. Of course, if the product isn't really defined, or if the product definition gets redefined later, the Cone will widen, and estimation accuracy will be poorer.

Tip #12 Don't assume that the Cone of Uncertainty will narrow itself. You must force the Cone to narrow by removing sources of variability from your project.

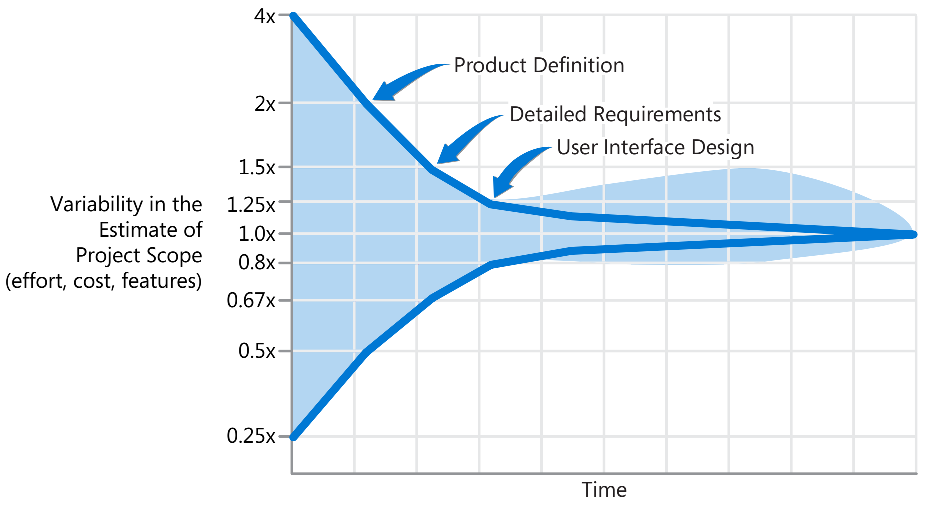

Figure 4-4 The Cone of Uncertainty doesn't narrow itself. You narrow the Cone by making decisions that remove sources of variability from the project. Some of these decisions are about what the project will deliver; some are about what the project will *not* deliver. If these decisions change later, the Cone will widen.

Accounting for the Cone of Uncertainty in Software Estimates

Studies of software estimates have found that estimators who start with single-point estimates and create ranges based on their original single-point numbers do not usually adjust their minimum and maximum values sufficiently to account for the uncertainty in the estimate, especially in circumstances that contain high uncertainty (Jørgensen 2002). This tendency to use ranges that are too narrow can be addressed two ways. The first is to start with a "most likely" estimate and then compute the ranges using predefined multipliers, as shown in Table 4-1.

Table 4-1 **Estimation Error by Software-Development Activity**

	Scoping Error		
Phase	**Possible Error on Low Side**	**Possible Error on High Side**	**Range of High to Low Estimates**
Initial Concept	0.25*x* (–75%)	4.0*x* (+300%)	16*x*
Approved Product Definition	0.50*x* (–50%)	2.0*x* (+100%)	4*x*
Requirements Complete	0.67*x* (–33%)	1.5*x* (+50%)	2.25*x*
User Interface Design Complete	0.80*x* (–20%)	1.25*x* (+25%)	1.6*x*
Detailed Design Complete (for sequential projects)	0.90*x* (–10%)	1.10*x* (+10%)	1.2*x*

Source: Adapted from *Software Estimation with Cocomo II* (Boehm et al. 2000).

If you use the entries from this table, recognize that at the point when you create the estimate you won't know whether the actual project outcome will fall toward the high end or the low end of your range.

Tip #13 Account for the Cone of Uncertainty by using predefined uncertainty ranges in your estimates.

A second approach is based on the finding that estimation "know-how-much" and estimation "know-how-uncertain" are two different skills. You can have one person estimate the best-case and worst-case ends of the range and a second person estimate the likelihood that the actual result will fall within that range (Jørgensen 2002).

Tip #14 Account for the Cone of Uncertainty by having one person create the "how much" part of the estimate and a different person create the "how uncertain" part of the estimate.

Relationship Between the Cone of Uncertainty and Commitment

Software organizations routinely sabotage their own projects by making commitments too early in the Cone of Uncertainty. If you commit at Initial Concept or Product Definition time, you will have a factor of 2*x* to 4*x* error in your estimates. As discussed in Chapter 1, "What Is an 'Estimate'?", a skilled project manager can navigate a project to completion if the estimate is within about 20% of the project reality. But no manager can navigate a project to a successful conclusion when the estimates are off by several hundred percent.

Meaningful commitments are not possible in the early, wide part of the Cone. Effective organizations delay their commitments until they have done the work to force the Cone to narrow. Meaningful commitments in the early-middle part of the project (about 30% of the way in) are possible and appropriate.

The Cone of Uncertainty and Iterative Development

Applying the Cone of Uncertainty to iterative projects is somewhat more involved than applying it to sequential projects is.

If you're working on a project that does a full development cycle each iteration—that is, from requirements definition through release—you'll go through a miniature Cone on each iteration. Before you do the requirements work for the iteration, you'll be at the Approved Product Definition point in the Cone, subject to 4*x* variability from high to low estimates. With short iterations (less than a month), you can move from Approved Product Definition to Requirements Complete and User Interface Design

Complete in a few days, reducing your variability from 4*x* to 1.6*x*. If your schedule is immovable, the 1.6*x* variability will apply to the specific features you can deliver in the time available, rather than to the effort or schedule. There are estimation advantages that flow from short iterations, which are discussed in Section 8.4, "Using Data from Your Current Project."

What you give up with approaches that leave requirements undefined until the beginning of each iteration is long-range predictability about the combination of cost, schedule, and features you'll deliver several iterations down the road. As Chapter 3, "Value of Accurate Estimates," discussed, your business might prioritize that flexibility highly, or it might prefer that your projects provide more predictability.

The alternative to *total* iteration is not *no* iteration. That option has been found to be almost universally ineffective. The alternatives are *less* iteration or *different* iteration.

Many development teams settle on a middle ground in which a majority of requirements are defined at the front end of the project, but design, construction, test, and release are performed in short iterations. In other words, the project moves sequentially through the User Interface Design Complete milestone (about 30% of the calendar time into the project) and then shifts to a more iterative approach from that point forward. This drives down the variability arising from the Cone to about ±25%, which allows for project control that is good enough to hit a target while still tapping into major benefits of iterative development. Project teams can leave some amount of planned time for as-yet-to-be-determined requirements at the end of the project. That introduces a little bit of variability related to the feature set, which in this case is positive variability because you'll exercise it only if you identify desirable features to implement. This middle ground supports long-range predictability of cost and schedule as well as a moderate amount of requirements flexibility.

4.3 Chaotic Development Processes

The Cone of Uncertainty represents uncertainty that is inherent even in well-run projects. Additional variability can arise from poorly run projects—that is, from avoidable project chaos.

Common examples of project chaos include the following:

- Requirements that weren't investigated very well in the first place
- Lack of end-user involvement in requirements validation
- Poor designs that lead to numerous errors in the code
- Poor coding practices that give rise to extensive bug fixing

- Inexperienced personnel
- Incomplete or unskilled project planning
- Prima donna team members
- Abandoning planning under pressure
- Developer gold-plating
- Lack of automated source code control

This is just a partial list of possible sources of chaos. For a more complete discussion, see Chapter 3, "Classic Mistakes," of my book *Rapid Development* (McConnell 1996) and on the Web at *www.stevemcconnell.com/rdenum.htm*.

These sources of chaos share two commonalities. The first is that each introduces variability that makes accurate estimation difficult. The second is that the best way to address each of these issues is not through estimation, but through better project control.

Tip #15 Don't expect better estimation practices alone to provide more accurate estimates for chaotic projects. You can't accurately estimate an out-of-control process. As a first step, fixing the chaos is more important than improving the estimates.

4.4 Unstable Requirements

Requirements changes have often been reported as a common source of estimation problems (Lederer and Prasad 1992, Jones 1994, Stutzke 2005). In addition to all the general challenges that unstable requirements create, they present two specific estimation challenges.

The first challenge is that unstable requirements represent one specific flavor of project chaos. If requirements cannot be stabilized, the Cone of Uncertainty can't be narrowed, and estimation variability will remain high through the end of the project.

The second challenge is that requirements changes are often not tracked and the project is often not reestimated when it should be. In a well-run project, an initial set of requirements will be baselined, and cost and schedule will be estimated from that baselined set of requirements. As new requirements are added or old requirements are revised, cost and schedule estimates will be modified to reflect those changes. In practice, project managers often neglect to update their cost and schedule assumptions as their requirements change. The irony in these cases is that the estimate for the original functionality might have been accurate, but after dozens of new requirements have been piled onto the project—requirements that have been agreed to but not accounted for—the project won't have any chance of meeting its original estimates, and the project will be perceived as being late, even though everyone agreed that the feature additions were good ideas.

The estimation techniques described in this book will certainly help you estimate *better* when you have high requirements volatility, but better estimation alone cannot address problems arising from requirements instability. The more powerful responses are project control responses rather than estimation responses. If your environment doesn't allow you to stabilize requirements, consider alternative development approaches that are designed to work in high-volatility environments, such as short iterations, Scrum, Extreme Programming, DSDM (Dynamic Systems Development Method), time box development, and so on.

Tip #16 To deal with unstable requirements, consider project control strategies instead of or in addition to estimation strategies.

Estimating Requirements Growth

If you do want to estimate the effect of unstable requirements, you might consider simply incorporating an allowance for requirements growth, requirements changes, or both into your estimates. Figure 4-5 shows a revised Cone of Uncertainty that accounts for approximately 50% growth in requirements over the course of a project. (This particular Cone is for purposes of illustration only. The specific data points are not supported by the same research as the original Cone.)

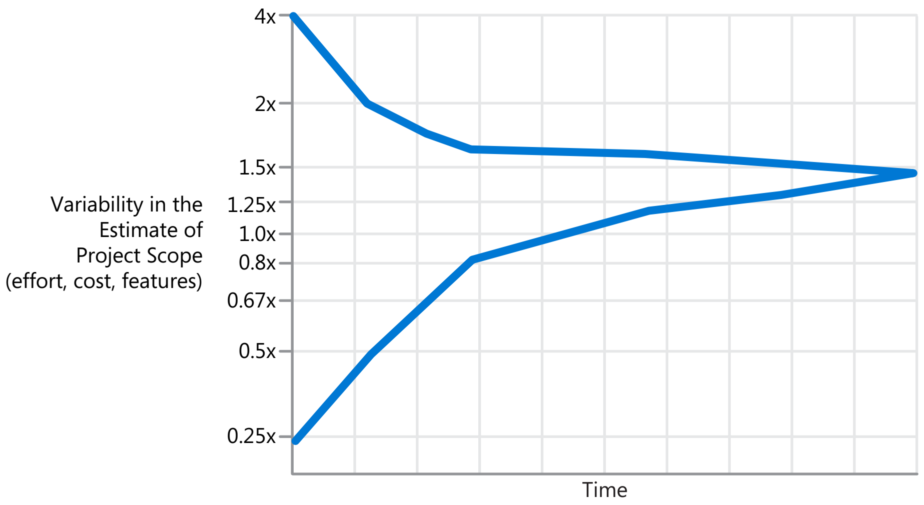

Figure 4-5 A Cone of Uncertainty that allows for requirements increases over the course of the project.

This approach has been used by leading organizations, including NASA's Software Engineering Laboratory, which plans on a 40% increase in requirements (NASA SEL 1990). A similar concept is incorporated into the Cocomo II estimation model, which includes the notion of requirements "breakage" (Boehm et al. 2000).

4.5 Omitted Activities

The previous sections described sources of error arising from the project itself. The remaining sections in this chapter turn to a discussion of errors that arise from the estimation practices.

One of the most common sources of estimation error is forgetting to include necessary tasks in the project estimates (Lederer and Prasad 1992, Coombs 2003). Researchers have found that this phenomenon applies both at the project planning level and at the individual developer level. One study found that developers tended to estimate pretty accurately the work they remembered to estimate, but they tended to overlook 20% to 30% of the necessary tasks, which led to a 20 to 30% estimation error (van Genuchten 1991).

Omitted work falls into three general categories: missing requirements, missing software-development activities, and missing non-software-development activities.

Table 4-2 lists requirements that are commonly missing from estimates.

Table 4-2 Functional and Nonfunctional Requirements Commonly Missing from Software Estimates

Functional Requirements Areas	Nonfunctional Requirements
Setup/installation program	Accuracy
Data conversion utility	Interoperability
Glue code needed to use third-party or open-source software	Modifiability
Help system	Performance
Deployment modes	Portability
Interfaces with external systems	Reliability
	Responsiveness
	Reusability
	Scalability
	Security
	Survivability
	Usability

Tip #17 Include time in your estimates for stated requirements, implied requirements, and nonfunctional requirements—that is, *all* requirements. Nothing can be built for free, and your estimates shouldn't imply that it can.

Table 4-3 lists software activities that estimators often overlook.

Table 4-3 Software-Development Activities Commonly Missing from Software Estimates

Ramp-up time for new team members	Technical support of existing systems during the project
Mentoring of new team members	Maintenance work on previous systems during the project
Management coordination/manager meetings	Defect-correction work
Cutover/deployment	Performance tuning
Data conversion	Learning new development tools
Installation	Administrative work related to defect tracking
Customization	Coordination with test (for developers)
Requirements clarifications	Coordination with developers (for test)
Maintaining the revision control system	Answering questions from quality assurance
Supporting the build	Input to user documentation and review of user documentation
Maintaining the scripts required to run the daily build	Review of technical documentation
Maintaining the automated smoke test used in conjunction with the daily build	Demonstrating software to customers or users
Installation of test builds at user location(s)	Demonstrating software at trade shows
Creation of test data	Demonstrating the software or prototypes of the software to upper management, clients, and end users
Management of beta test program	Interacting with clients or end users; supporting beta installations at client locations
Participation in technical reviews	Reviewing plans, estimates, architecture, detailed designs, stage plans, code, test cases, and so on
Integration work	
Processing change requests	
Attendance at change-control/triage meetings	
Coordinating with subcontractors	

Tip #18 Include all necessary software-development activities in your estimates, not just coding and testing.

Table 4-4 lists the non-software-development activities that are often missing from estimates.

Table 4-4 Non-Software-Development Activities Commonly Missing from Software Estimates

Vacations	Company meetings
Holidays	Department meetings
Sick days	Setting up new workstations
Training	Installing new versions of tools on workstations
Weekends	Troubleshooting hardware and software problems

Some projects deliberately plan to exclude many of the activities in Table 4-4 for a small project. That can work for a short time, but these activities tend to creep back into any project that lasts longer than a few weeks.

Tip #19 On projects that last longer than a few weeks, include allowances for overhead activities such as vacations, sick days, training days, and company meetings.

In addition to using the entries in these tables to avoid omitting parts of the software or kinds of activities from your estimates, you might also consider looking at a Work Breakdown Structure (WBS) for the standard kinds of activities to be considered. Section 10.3, "Hazards of Adding Up Best Case and Worst Case Estimates," discusses estimating with a WBS and provides a generic WBS.

4.6 Unfounded Optimism

Optimism assails software estimates from all sources. On the developer side of the project, Microsoft Vice President Chris Peters observed that "You never have to fear that estimates created by developers will be too pessimistic, because developers will always generate a too-optimistic schedule" (Cusumano and Selby 1995). In a study of 300 software projects, Michiel van Genuchten reported that developer estimates tended to contain an optimism factor of 20% to 30% (van Genuchten 1991). Although managers sometimes complain otherwise, developers don't tend to sandbag their estimates—their estimates tend to be too low!

Tip #20 Don't reduce developer estimates—they're probably too optimistic already.

Optimism applies within the management ranks as well. A study of about 100 schedule estimates within the U.S. Department of Defense found a consistent "fantasy factor" of about 1.33 (Boehm 1981). Project managers and executives might not *assume* that projects can be done 30% faster or cheaper than they can be done, but they certainly *want* the projects to be done faster and cheaper, and that is a kind of optimism in itself.

Common variations on this optimism theme include the following:

- We'll be more productive on this project than we were on the last project.
- A lot of things went wrong on the last project. Not so many things will go wrong on this project.
- We started the project slowly and were climbing a steep learning curve. We learned a lot of lessons the hard way, but all the lessons we learned will allow us to finish the project much faster than we started it.

Considering that optimism is a near-universal fact of human nature, software estimates are sometimes undermined by what I think of as a Collusion of Optimists. Developers present estimates that are optimistic. Executives like the optimistic estimates because they imply that desirable business targets are achievable. Managers like the estimates because they imply that they can support upper management's objectives. And so the software project is off and running with no one ever taking a critical look at whether the estimates were well founded in the first place.

4.7 Subjectivity and Bias

Subjectivity creeps into estimates in the form of optimism, in the form of conscious bias, and in the form of unconscious bias. I differentiate between estimation *bias*, which suggests an intent to fudge an estimate in one direction or another, and estimation *subjectivity*, which simply recognizes that human judgment is influenced by human experience, both consciously and unconsciously.

As far as bias is concerned, the response of customers and managers when they discover that the estimate does not align with the business target is sometimes to apply more pressure to the estimate, to the project, and to the project team. Excessive schedule pressure occurs in 75% to 100% of large projects (Jones 1994).

As far as subjectivity is concerned, when considering different estimation techniques our natural tendency is to believe that the more "control knobs" we have on an estimate—that is, the more places there are to tweak the estimate to match our specific project—the more accurate the estimate will be.

The reality is the opposite. The more control knobs an estimate has, the more chances there are for subjectivity to creep in. The issue is not so much that estimators deliberately bias their estimates. The issue is more that the estimate gets shaded slightly higher or slightly lower with each of the subjective inputs. If the estimation technique has a large number of subjective inputs, the cumulative effect can be significant.

I've seen one example of this while teaching several hundred estimators to use the Cocomo II estimation model. Cocomo II includes 17 Effort Multipliers and 5 Scaling Factors. To create a Cocomo II estimate, the estimator must decide what adjustment is needed for each of these 22 factors. The factors adjust for whether your team is above average or below average, whether your software is more or less complex than average, and so on. In theory, these 22 control knobs should allow virtually any estimate to be fine-tuned. In practice, the control knobs seem to introduce 22 chances for error to creep into the estimate.

Figure 4-6 shows the ranges of results of about 100 groups of estimators applying Cocomo II's 17 Effort Multipliers to the same estimation problem. For each bar, the bottom of the bar represents the lowest group estimate in a session and the top of the bar represents the highest group estimate in a session. The total height of the bar represents the variation in the estimates.

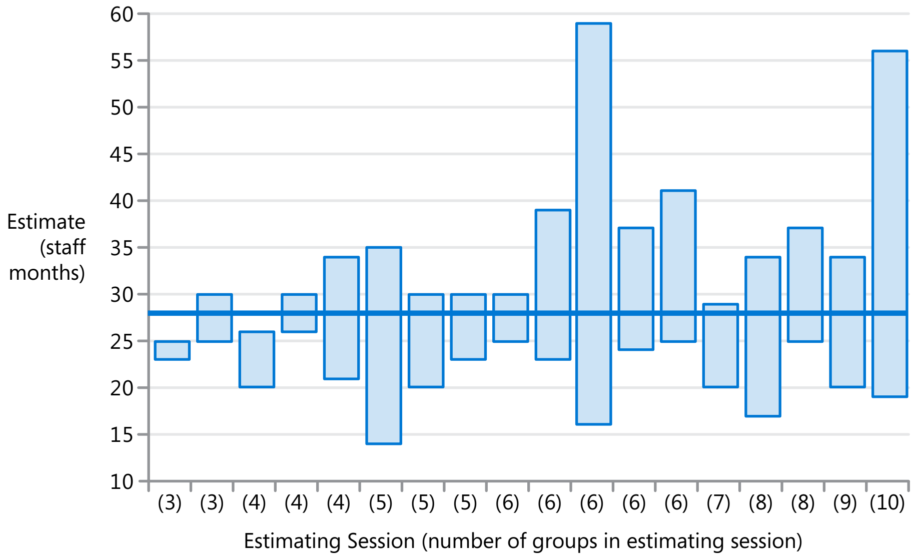

Figure 4-6 Example of variations in estimates when numerous adjustment factors are present. The more adjustment factors an estimation method provides, the more opportunity there is for subjectivity to creep into the estimate.

If the estimation technique produced consistent results, we would see a tight clustering of results along the horizontal blue bar (the average of all the estimates). But, as you can see, the variation among the estimates is enormous. The total variation from highest to lowest is a factor of 4. The average variation from the low group to the high group within any one session is a factor of 1.7.

An important aspect of this data is that *this particular estimation exercise is free of external bias*. It occurs in an estimation class in which the emphasis is on accuracy. The only bias that is affecting these estimates are the biases inherent in the estimators' experiences. In a real estimation situation, the range of results would probably be even greater because of the increased amount of external bias that would affect the estimates.

In contrast, Figure 4-7 illustrates the range of estimation outcomes with an estimation technique that includes only one place to insert a subjective judgment into the estimate—that is, one control knob. (The control knob in this case is unrelated to the Cocomo II factors.)

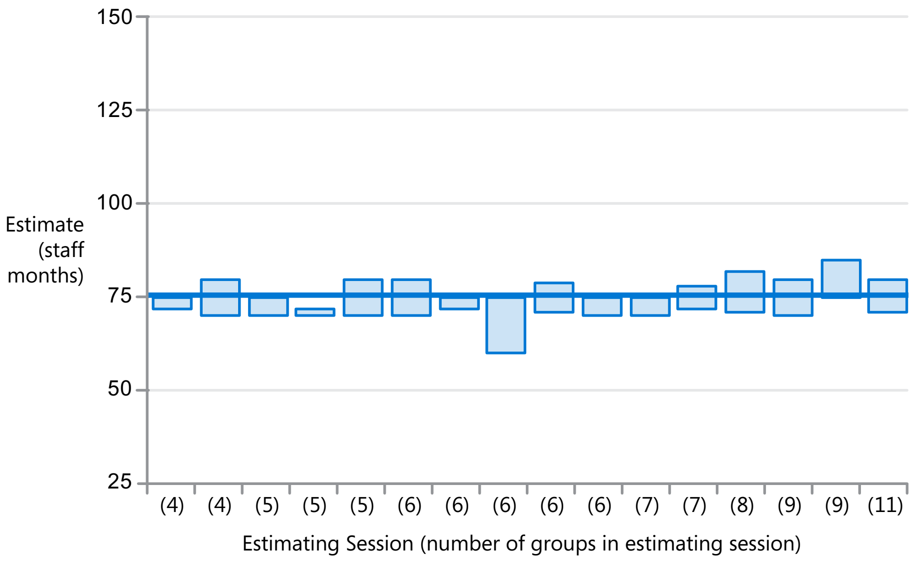

Figure 4-7 Example of low variation in estimates resulting from a small number of adjustment factors. (The scales of the two graphs are different, but they are directly comparable when you account for the difference in the average values on the two graphs.)

As you can see, the variation in these results is dramatically smaller than the variation when there are 17 control knobs present. The average variation from the low group to the high group within any one session is a factor of only 1.1.

The finding that "more control knobs isn't better" extends beyond software estimation. As forecasting guru J. Scott Armstrong states, "One of the most enduring and useful conclusions from research on forecasting is that simple methods are generally as accurate as complex methods" (Armstrong 2001).

Tip #21 Avoid having "control knobs" on your estimates. While control knobs might give you a feeling of better accuracy, they usually introduce subjectivity and degrade actual accuracy.

4.8 Off-the-Cuff Estimates

Project teams are sometimes trapped by off-the-cuff estimates. Your boss asks, for example, "How long would it take to implement print preview on the Gigacorp Web site?" You say, "I don't know. I think it might take about a week. I'll check into it." You go off to your desk, look at the design and code for the program you were asked about, notice a few things you'd forgotten when you talked to your manager, add up the changes, and decide that it would take about five weeks. You hurry over to your manager's office to update your first estimate, but the manager is in a meeting. Later that day, you catch up with your manager, and before you can open your mouth, your manager says, "Since it seemed like a small project, I went ahead and asked for approval for the print-preview function at the budget meeting this afternoon. The rest of the budget committee was excited about the new feature and can't wait to see it next week. Can you start working on it today?"

I've found that the safest policy is not to give off-the-cuff estimates. Lederer and Prasad found that intuition and guessing in software project estimates were both correlated with cost and schedule overruns (Lederer and Prasad 1992). I've collected data on off-the-cuff estimates from 24 groups of estimators. Figure 4-8 shows the average error of estimates in these 24 groups of estimators for off-the-cuff estimates versus estimates that go through a group review process.

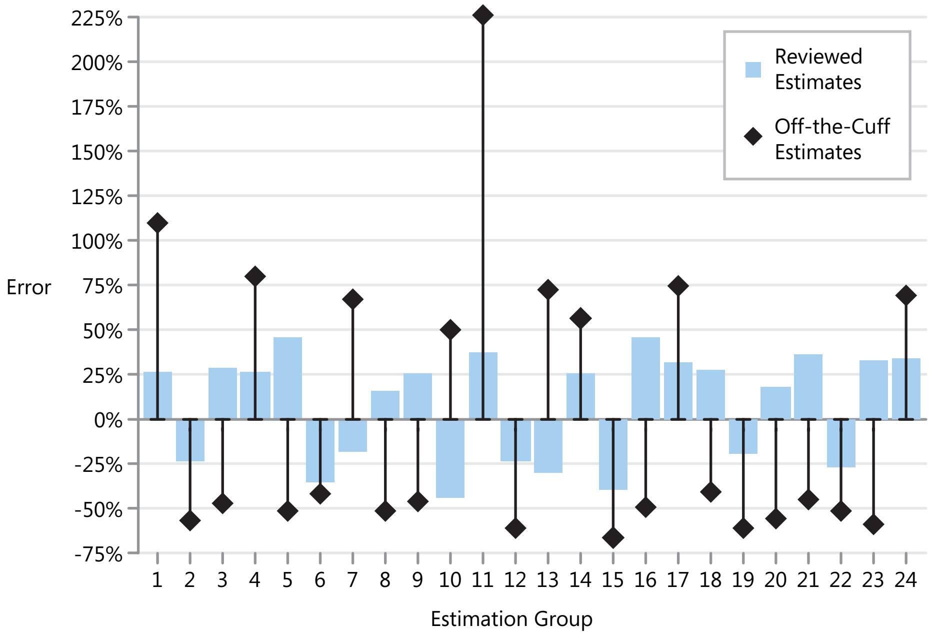

Figure 4-8 Average error from off-the-cuff estimates vs. reviewed estimates.

The average off-the-cuff estimate has a mean magnitude of relative error (MMRE[1]) of 67%, whereas the average reviewed estimate has an error of only 30%—less than half. (These are not software estimates, so the percentage errors shouldn't be applied literally to software project estimates.)

[1] MMRE is equal to AbsoluteValue [(ActualResult − EstimatedResult) / ActualResult].

One of the errors people commit when estimating solely from personal memory is that they compare the new project to their memory of how long a past project took, or how much effort it required. Unfortunately, people sometimes remember their *estimate* for the past project rather than the *actual outcome* of the past project. If they use their past estimate as the basis for a new estimate, and the past project's actual outcome was that it overran its estimate, guess what? The estimator has just calibrated a project overrun into the estimate for the new project.

While Lederer and Prasad found that guessing and intuition were positively correlated with project overruns, they also found that the use of "documented facts" was *negatively* correlated with project overruns. In other words, there is a world of difference between giving your boss an off-the-cuff answer versus saying, "I can't give you an answer off the top of my head, but let me go back to my desk, check a few notes, and get back to you in 15 minutes. Would that be OK?"

While this is a simple point, off-the-cuff estimation is one of the most common errors that project teams make (Lederer and Prasad 1992, Jørgensen 1997, Kitchenham et al. 2002). Avoiding off-the-cuff estimates is one of the most important points in this book.

Tip #22 Don't give off-the-cuff estimates. Even a 15-minute estimate will be more accurate.

What if your boss calls on a cell phone and insists on getting an estimate *right now*? Consider your performance on the estimation quiz in Chapter 2, "How Good an Estimator Are You?" Did you get 8 to 10 answers correct? If not, what are the odds that the off-the-cuff answer you give your boss on the cell phone—even an estimate padded for uncertainty—will give you a 90% chance of including the correct answer?

4.9 Unwarranted Precision

In casual conversation, people tend to treat "accuracy" and "precision" as synonyms. But for estimation purposes, the distinctions between these two terms are critical.

Accuracy refers to how close to the real value a number is. *Precision* refers merely to how exact a number is. In software estimation, this amounts to how many significant digits an estimate has. A measurement can be precise without being accurate, and it can be accurate without being precise. The single digit 3 is an accurate representation of pi to one significant digit, but it is not precise. 3.37882 is a more precise representation of pi than 3 is, but it is not any more accurate.

Airline schedules are precise to the minute, but they are not very accurate. Measuring people's heights in whole meters might be accurate, but it would not be at all precise.

Table 4-5 provides examples of numbers that are accurate, precise, or both.

Table 4-5 Examples of Accuracy and Precision

Example	Comment
pi = 3	Accurate to 1 significant digit, but not precise
pi = 3.37882	Precise to 6 significant digits, but accurate only to 1 significant digit
pi = 3.14159	Both accurate and precise, to 6 significant digits
My height = 2 meters	Accurate to 1 significant digit, but not very precise
Airline flight times	Precise to the minute, but not very accurate
"This project will take 395.7 days, ± 2 months"	Highly precise, but not accurate to the precision stated
"This project will take 1 year"	Not very precise, but could be accurate
"This project will require 7,214 staff hours"	Highly precise, but probably not accurate to the precision stated
"This project will require 4 staff years"	Not very precise, but could be accurate

For software estimation purposes, the distinction between accuracy and precision is critical. Project stakeholders make assumptions about project accuracy based on the precision with which an estimate is presented. When you present an estimate of 395.7 days, stakeholders assume the estimate is accurate to 4 significant digits! The accuracy of the estimate might be better reflected by estimating 1 year, 4 quarters, or 13 months, rather than 395.7 days. Using an estimate of 395.7 days instead of 1 year is like representing pi as 3.37882—the number is more precise, but it's really less accurate.

Tip #23 Match the number of significant digits in your estimate (its precision) to to your estimate's accuracy.

4.10 Other Sources of Error

The sources of error described in the first nine sections of this chapter are the most common and the most significant, but they are not exhaustive. Here are some of the other ways that error can creep into an estimate:

- Unfamiliar business area
- Unfamiliar technology area
- Incorrect conversion from estimated time to project time (for example, assuming the project team will focus on the project eight hours per day, five days per week)
- Misunderstanding of statistical concepts (especially adding together a set of "best case" estimates or a set of "worst case" estimates)

- Budgeting processes that undermine effective estimation (especially those that require final budget approval in the wide part of the Cone of Uncertainty)
- Having an accurate size estimate, but introducing errors when converting the size estimate to an effort estimate
- Having accurate size and effort estimates, but introducing errors when converting those to a schedule estimate
- Overstated savings from new development tools or methods
- Simplification of the estimate as it's reported up layers of management, fed into the budgeting process, and so on

These topics are all discussed in more detail in later chapters.

Additional Resources

Armstrong, J. Scott, ed. *Principles of forecasting: A handbook for researchers and practitioners*. Boston, MA: Kluwer Academic Publishers, 2001. Armstrong is one of the leading researchers in forecasting in a marketing context. Many of the observations in this book are relevant to software estimation. Armstrong has been a leading critic of overly complex estimation models.

Boehm, Barry, et al. *Software Cost Estimation with Cocomo II*. Reading, MA: Addison-Wesley, 2000. Boehm was the first to popularize the Cone of Uncertainty (he calls it a funnel curve). This book contains his most current description of the phenomenon.

Cockburn, Alistair. *Agile Software Development*. Boston, MA: Addison-Wesley, 2001. Cockburn's book introduces Agile development approaches, which are especially useful in environments characterized by highly volatile requirements.

Laranjeira, Luiz. "Software Size Estimation of Object-Oriented Systems," *IEEE Transactions on Software Engineering*, May 1990. This paper provided a theoretical-research foundation for the empirical observation of the Cone of Uncertainty.

Tockey, Steve. *Return on Software*. Boston, MA: Addison-Wesley, 2005. Chapters 21 through 23 discuss basic estimation concepts, general estimation techniques, and allowing for inaccuracy in estimates. Tockey includes a detailed discussion of how to build your own Cone of Uncertainty.

Wiegers, Karl. *More About Software Requirements: Thorny Issues and Practical Advice*. Redmond, WA: Microsoft Press, 2006.

Wiegers, Karl. *Software Requirements, Second Edition*. Redmond, WA: Microsoft Press, 2003. In these two books, Wiegers describes numerous practices that help elicit good requirements in the first place, which substantially reduces requirements volatility later in a project.