Chapter 7

# Count, Compute, Judge

## Applicability of Techniques in This Chapter

| | Count | Compute |
|---|---|---|
| **What's estimated** | Size, Features | Size, Effort, Schedule, Features |
| **Size of project** | S M L | S M L |
| **Development stage** | Early–Late | Early–Middle |
| **Iterative or sequential** | Both | Both |
| **Accuracy possible** | High | High |

Suppose you're at a reception for the world's best software estimators. The room is packed, and you're seated in the middle of the room at a table with three other estimators. All you can see as you scan the room are wall-to-wall estimators. Suddenly, the emcee steps up to the microphone and says, "We need to know exactly how many people are in this room so that we can order dessert. Who can give me the most accurate estimate for the number of people in the room?"

The estimators at your table immediately break out into a vigorous discussion about the best way to estimate the answer. Bill, the estimator to your right, says, "I make a hobby of estimating crowds. Based on my experience, it looks to me like we've got about 335 people in the room."

The estimator sitting across the table from you, Karl, says, "This room has 11 tables across and 7 tables deep. One of my friends is a banquet planner, and she told me that they plan for 5 people per table. It looks to me like most of the tables do actually have about 5 people at them. If we multiple 11 times 7 times 5, we get 385 people. I think we should use that as our estimate."

The estimator to your left, Lucy, says, "I noticed on the way into the room that there was an occupancy limit sign that says this room can hold 485 people. This room is pretty full. I'd say 70 to 80 percent full. If we multiply those percentages by the room limit, we get 340 to 388 people. How about if we use the average of 364 people, or maybe just simplify it to 365?"

Bill says, "We have estimates of 335, 365, and 385. It seems like the right answer must be in there somewhere. I'm comfortable with 365."

"Me too," Karl says.

Everyone looks at you. You say, "I need to check something. Would you excuse me for a minute?" Lucy, Karl, and Bill give you curious looks and say, "OK."

You return a few minutes later. "Remember how we had to have our tickets scanned before we entered the room? I noticed on my way into the room that the handheld ticket scanner had a counter. So I went back and talked to the ticket taker at the front door. She said that, according to her scanner, she has scanned 407 tickets. She also said no one has left the room so far. I think we should use 407 as our estimate. What do you say?"

## 7.1 Count First

What do you think the right answer is? Is it the answer of 335, created by Bill, whose specialty is estimating crowd sizes? Is it the answer of 385, derived by Karl from a few reasonable assumptions? Is it Lucy's 365, also derived from a few reasonable assumptions? Or is the right number the 407 that was counted by the ticket scanner? *Is there any doubt in your mind that 407 is the most accurate answer?* For the record, the story ended by your table proposing the answer of 407, which turned out to be the correct number, and your table was served dessert first.

One of the secrets of this book is that you should avoid doing what we traditionally think of as estimating! If you can *count* the answer directly, you should do that first. That approach produced the most accurate answer in the story.

If you can't count the answer directly, you should count something else and then *compute* the answer by using some sort of calibration data. In the story, Karl had the historical data of knowing that the banquet was planned to have 5 people per table. He *counted* the number of tables and then computed the answer from that.

Similarly, Lucy based her estimate on the documented fact of the room's occupancy limit. She used her *judgment* to estimate the room was 70 to 80 percent full.

The least accurate estimate came from, Bill, the person who used only *judgment* to create the answer.

**Tip #30** *Count* if at all possible. *Compute* when you can't count. Use *judgment* alone only as a last resort.

## 7.2 What to Count

Software projects produce numerous things that you can count. Early in the development life cycle, you can count marketing requirements, features, use cases, and stories, among other things.

In the middle of the project, you can count at a finer level of granularity—engineering requirements, Function Points, change requests, Web pages, reports, dialog boxes, screens, and database tables, just to name a few.

Late in the project, you can count at an even finer level of detail—code already written, defects reported, classes, and tasks, as well as all the detailed items you were counting earlier in the project.

You can decide what to count based on a few goals.

***Find something to count that's highly correlated with the size of the software you're estimating*** If your features are fixed and you're estimating cost and schedule, the biggest influence on a project estimate is the size of the software. When you look for something to count, look for something that will be a strong indicator of the software's size. Number of marketing requirements, number of engineering requirements, and Function Points are all examples of countable quantities that are strongly associated with final system size.

In different environments, different quantities are the most accurate indicators of project size. In one environment, the best indicator might be the number of Web pages. In another environment, the best indicator might be the number of marketing requirements, test cases, stories, or configuration settings. The trick is to find something that's a relevant indicator of size in your environment.

**Tip #31** Look for something you can count that is a meaningful measure of the scope of work in your environment.

***Find something to count that's available sooner rather than later in the development cycle*** The sooner you can find something meaningful to count, the sooner you'll be able to provide long-range predictability. The count of lines of code for a project is often a great indicator of project effort, but the code won't be available to count until the very end of the project. Function Points are strongly associated with ultimate project size, but they aren't available until you have detailed requirements. If you can find something you can count earlier, you can use that to create an estimate earlier. For example, you might create a rough estimate based on a count of marketing requirements and then tighten up the estimate later based on a Function Point count.

***Find something to count that will produce a statistically meaningful average*** Find something that will produce a count of 20 or more. Statistically, you need a sample of at least 20 items for the average to be meaningful. Twenty is not a magic number, but it's a good guideline for statistical validity.

***Understand what you're counting*** For your count to serve as an accurate basis for estimation, you need to be sure the same assumptions apply to the count that your historical data is based on and to the count that you're using for your estimate. If you're counting marketing requirements, be sure that what you counted as a "marketing requirement" for your historical data is similar to what you count as a "marketing requirement" for your estimate. If your historical data indicates that a past project team in your company delivered 7 user stories per week, be sure your assumptions about team size, programmer experience, development technology, and other factors are similar in the project you're estimating.

***Find something you can count with minimal effort*** All other things being equal, you'd rather count something that requires the least effort. In the story at the beginning of the chapter, the count of people in the room was readily available from the ticket scanner. If you had to go around to each table and count people manually, you might decide it wasn't worth the effort.

One of the insights from the Cocomo II project is that a size estimation measure called Object Points is about as strongly correlated with effort as the Function Points measure is but requires only about half as much effort to count. Thus, Object Points are seen as an effective alternative to Function Points for estimation in the wide part of the Cone of Uncertainty (Boehm et al 2000).

## 7.3 Use Computation to Convert Counts to Estimates

If you collect historical data related to counts, you can convert the counts to something useful, such as estimated effort. Table 7-1 lists examples of quantities you might count and the data you would need to compute an estimate from the count.

**Table 7-1 Examples of Quantities That Can Be Counted for Estimation Purposes**

| Quantity to Count | Historical Data Needed to Convert the Count to an Estimate |
|---|---|
| Marketing requirements | ■ Average effort hours per requirement for development<br>■ Average effort hours per requirement for independent testing<br>■ Average effort hours per requirement for documentation<br>■ Average effort hours per requirement to create engineering requirements from marketing requirements |

Table 7-1 Examples of Quantities That Can Be Counted for Estimation Purposes

| Quantity to Count | Historical Data Needed to Convert the Count to an Estimate |
|---|---|
| Features | ■ Average effort hours per feature for development and/or testing |
| Use cases | ■ Average total effort hours per use case<br>■ Average number of use cases that can be delivered in a particular amount of calendar time |
| Stories | ■ Average total effort hours per story<br>■ Average number of stories that can be delivered in a particular amount of calendar time |
| Engineering requirements | ■ Average number of engineering requirements that can be formally inspected per hour<br>■ Average effort hours per requirement for development/test/documentation |
| Function Points | ■ Average development/test/documentation effort per Function Point<br>■ Average lines of code in the target language per Function Point |
| Change requests | ■ Average development/test/documentation effort per change request (depending on variability of the change requests, the data might be decomposed into average effort per small, medium, and large change request) |
| Web pages | ■ Average effort per Web page for user interface work<br>■ Average whole-project effort per Web page (less reliable, but can be an interesting data point) |
| Reports | ■ Average effort per report for report work |
| Dialog boxes | ■ Average effort per dialog for user interface work |
| Database tables | ■ Average effort per table for database work<br>■ Average whole-project effort per table (less reliable, but can be an interesting data point) |
| Classes | ■ Average effort hours per class for development<br>■ Average effort hours to formally inspect a class<br>■ Average effort hours per class for testing |
| Defects found | ■ Average effort hours per defect to fix<br>■ Average effort hours per defect to regression test<br>■ Average number of defects that can be corrected in a particular amount of calendar time |
| Configuration settings | ■ Average effort per configuration setting |

Table 7-1 Examples of Quantities That Can Be Counted for Estimation Purposes

| Quantity to Count | Historical Data Needed to Convert the Count to an Estimate |
|---|---|
| Lines of code already written | ■ Average number of defects per line of code<br>■ Average lines of code that can be formally inspected per hour<br>■ Average new lines of code from one release to the next |
| Test cases already written | ■ Average amount of release-stage effort per test case |

**Tip #32** Collect historical data that allows you to compute an estimate from a count.

***Example of counting defects late in a project*** Once you have the kind of data described in the table, you can use that data as a more solid basis for creating estimates than expert judgment. If you know that you have 400 open defects, and you know that the 250 defects you've fixed so far have averaged 2 hours per defect, you know that you have about 400 x 2 equals 800 hours of work to fix the open defects.

***Example of estimation by counting Web pages*** If your data says that so far your project has taken an average of 40 hours to design, code, and test each Web page with dynamic content, and you have 12 Web pages left, you know that you have something like 12 x 40 equals 480 hours of work left on the remaining Web pages.

The important point in these examples is that *there is no judgment in these estimates*. You count, and then you compute. This process helps keep the estimates free from bias that would otherwise degrade their accuracy. For counts that you already have available—such as number of defects—such estimates also require very low effort.

**Tip #33** Don't discount the power of simple, coarse estimation models such as average effort per defect, average effort per Web page, average effort per story, and average effort per use case.

## 7.4 Use Judgment Only as a Last Resort

So-called expert judgment is the least accurate means of estimation. Estimates seem to be the most accurate if they can be tied to something concrete. In the story told at the beginning of this chapter, the worst estimate was the one created by the expert who used judgment alone. Tying the estimate to the room occupancy limit was a little better, although it was subject to more error because that approach required a judgment about how full the room was as a percentage of maximum occupancy, which is an opportunity for subjectivity or bias to contaminate the estimate.

Historical data combined with computation is remarkably free from the biases that can undermine more judgment-based estimates. Avoid the temptation to tweak computed estimates to conform to your expert judgment. When I wrote the second edition of *Code Complete* (McConnell 2004a), I had a team that formally inspected the entire first edition—all 900 pages of it. During our first inspection meeting, our inspection rate averaged 3 minutes per page. Realizing that 3 minutes per page implied 45 hours of inspection meetings, I commented after the first meeting that I thought we were just beginning to gel as a team, and, in my judgment, we would speed up in future meetings. I suggested using a working number of 2 or 2.5 minutes per page instead of 3 minutes to plan future meetings. The project manager responded that, because we had only one meeting's worth of data, we should use that meeting's number of 3 minutes per page as a guide for planning the next few meetings. We could adjust our plans later based on different data from later meetings, if we needed to.

Nine hundred pages later, how many minutes per page do you think we averaged for the entire book? If you guessed 3 minutes per page, you're right!

**Tip #34** Avoid using expert judgment to tweak an estimate that has been derived through computation. Such "expert judgment" usually degrades the estimate's accuracy.

## Additional Resources

Boehm, Barry, et al. *Software Cost Estimation with Cocomo II*. Reading, MA: Addison-Wesley, 2000. Boehm provides a brief description of Object Points.

Lorenz, Mark and Jeff Kidd. *Object-Oriented Software Metrics*. Upper Saddle River, NJ: PTR Prentice Hall, 1994. Lorenz and Kidd present numerous suggestions of quantities that can be counted in object-oriented programs.

Chapter 8

# Calibration and Historical Data

## Applicability of Techniques in This Chapter

| | Calibration with Industry-Average Data | Calibration with Organizational Data | Calibration with Project-Specific Data |
|---|---|---|---|
| **What's estimated** | Size, Effort, Schedule, Features | Size, Effort, Schedule, Features | Size, Effort, Schedule, Features |
| **Size of project** | S M L | S M L | S M L |
| **Development stage** | Early–Middle | Early–Middle | Middle–Late |
| **Iterative or sequential** | Both | Both | Both |
| **Accuracy possible** | Low–Medium | Medium–High | High |

Calibration is used to convert counts to estimates—lines of code to effort, user stories to calendar time, requirements to number of test cases, and so on. Estimates always involve some sort of calibration, whether explicit or implicit. Calibration using various kinds of data makes up the second piece of the "count, then compute" approach described in Chapter 7, "Count, Compute, Judge."

Your estimates can be calibrated using any of three kinds of data:

- *Industry data*, which refers to data from other organizations that develop the same basic kind of software as the software that's being estimated
- *Historical data*, which in this book refers to data from the organization that will conduct the project being estimated
- *Project data*, which refers to data generated earlier in the same project that's being estimated

Historical data and project data are both tremendously useful and can support creation of highly accurate estimates. Industry data is a temporary backup that can be useful when you don't have historical data or project data.

## 8.1 Improved Accuracy and Other Benefits of Historical Data

The most important reason to use historical data from your own organization is that it improves estimation accuracy. The use of historical data, or "documented facts," is negatively correlated with cost and schedule overruns—that is, projects that have been estimated using historical data tend not to have overruns (Lederer and Prasad 1992).

The following sections discuss some of the reasons that historical data improves accuracy.

## Accounts for Organizational Influences

First and foremost, use of historical data accounts for a raft of organizational influences that affect project outcomes. For very small projects, individual capabilities dictate the project outcome. As project size increases, talented individuals still matter, but their efforts are either supported or undermined by organizational influences. For medium and large projects, organizational characteristics start to matter as much as or more than individual capabilities.

Here are some of the organizational influences that affect project outcomes:

- How complex is the software, what is the execution time constraint, what reliability is required, how much documentation is required, how precedented is the application—that is, how does the project stack up against the Cocomo II factors related to the kind of software being developed (as discussed in Chapter 5, "Estimate Influences")?
- Can the organization commit to stable requirements, or must the project team deal with volatile requirements throughout the project?
- Is the project manager free to remove a problem team member from the project, or do the organization's Human Resources policies make it difficult or impossible to remove a problem employee?
- Is the team free to concentrate on the current project, or are team members frequently interrupted with calls to support production releases of previous projects?
- Can the organization add team members to the new project as planned, or does it refuse to pull people off other projects before those projects have been completed?
- Does the organization support the use of effective design, construction, quality assurance, and testing practices?
- Does the organization operate in a regulated environment (for example, under FAA or FDA regulations) in which certain practices are dictated?
- Can the project manager depend on team members staying until the project is complete, or does the organization have high turnover?

Accounting for each of these influences in an estimate one by one is difficult and error-prone. But historical data adjusts for all these factors, whether you're aware of the specifics or not.

## Avoids Subjectivity and Unfounded Optimism

One way that subjectivity creeps into estimates is that project managers or estimators look at a new project, compare it with an old project, observe numerous differences between the two projects, and then conclude that the new project will go better than the old one did. They say, "We had a lot of turnover on the last project. That won't happen this time, so we'll be more productive. Also, people kept getting called back to support the previous version, and we'll make sure that that doesn't happen this time either. We also had a lot of late-breaking requirements from marketing. We'll do a better job on that, too. Plus we're working with better technology this time and newer, more effective development methods. With all those improvements, we should be able to be *way* more productive."

It's easy to identify with the optimism in these lines of reasoning. But the factors listed are controlled more by the organization than by the specific project manager, so most of these factors tend to be difficult to control for one specific project. The other factors tend to be interpreted optimistically, which introduces bias into the estimate.

With historical data, you use a simplifying assumption that the next project will go about the same as the last project did. This is a reasonable assumption. As estimation guru Lawrence Putnam says, productivity is an organizational attribute that cannot easily be varied from project to project (Putnam and Myers 1992, Putnam and Myers 2003). The same concept shows up in Extreme Programming as "Yesterday's Weather": the weather today won't always be the same as it was yesterday, but it's more likely to be like yesterday's weather than like anything else (Beck and Fowler 2001).

**Tip #35** Use historical data as the basis for your productivity assumptions. Unlike mutual fund disclosures, your organization's past performance really is your best indicator of future performance.

## Reduces Estimation Politics

One of the traps in estimation models that include a lot of control knobs is that many of the higher-leverage knobs are related to personnel. Cocomo II, for example, requires you to make assessments of your requirements analysts' and programmers' capabilities, along with several less subjective personnel factors related to experience. Cocomo requires the estimator to rate the programmers as 90th percentile, 75th percentile, 55th percentile, 35th percentile, or 15th percentile. (All these percentiles are industrywide.)

Suppose a manager takes a Cocomo II estimate into a meeting with an executive and the meeting agenda is to look for fat in the manager's estimate. It's easy to imagine the conversation going like this:

> *MANAGER: I know we had a goal of finishing this release in 12 weeks, but my estimates indicate that it will take 16 weeks. Let's walk through the estimate using this software estimation tool. Here are the assumptions I made. First, I had to calibrate the estimation model. For the "programmer capability" factor, I assumed our programmers are 35th percentile–*
>
> *EXECUTIVE: What?! No one on our staff is below average! You need to have more confidence in your staff! What kind of manager are you? Well, maybe we've got a few people who aren't quite as good as the rest, but the overall team can't be that bad. Let's assume they're at least average, right? Can you enter that into the software?*
>
> *MANAGER: Well, OK. Now, the next factor is the capability of the requirements engineers. We've never focused on recruiting good requirements engineers or developing those skills in our engineers, so I assumed they were 15th percentile–*
>
> *EXECUTIVE: Hold on! 15th percentile? These people are very talented, even if they haven't had formal training in requirements engineering. They've got to be at least average. Can we change that factor to average?*
>
> *MANAGER: I can't justify making them average. We really don't even have any staff we can call requirements specialists.*
>
> *EXECUTIVE: Fine. Let's compromise and change the factor to 35th percentile then.*
>
> *MANAGER: OK (sigh).*

In this interaction, if the manager was using the Cocomo II adjustment factors, his estimate of effort required was just reduced by 23%. If the executive had succeeded in talking the manager into rating the requirements engineers as average rather than 35th percentile, the estimate would be reduced by 39%. In either case, a single conversation would result in a significant difference.

A manager who calibrates the estimate with historical data sidesteps the whole issue of whether the programmers are above average or below average. Productivity is whatever the data says it is. It's difficult for a non-technical stakeholder to argue with a statement like this one: "We've averaged 300 to 450 delivered lines of code per staff month, so we've calibrated the model with an assumption of 400 lines of code per staff month, which we believe is a little on the optimistic side but within a prudent planning range."

Clearly, half the programmers in the industry are below average, but I rarely meet project managers or executives who believe *their* programmers are the people who are below average.

**Tip #36** Use historical data to help avoid politically charged estimation discussions arising from assumptions like "My team is below average."

## 8.2 Data to Collect

If you're not already collecting historical data, you can start with a very small set of data:

- Size (lines of code or something else you can count after the software has been released)
- Effort (staff months)
- Time (calendar months)
- Defects (classified by severity)

This small amount of data, even if you collect it only at the completion of two or three projects, will give you enough data to calibrate any of several commercial software estimation tools. It will also allow you to compute simple ratios such as lines of code per staff month.

In addition to the fact that these four kinds of data are sufficient to calibrate estimation models, most experts recommend starting small so that you understand what you're collecting (Pietrasanta 1990, NASA SEL 1995). If you don't start small, you can end up with data that's defined inconsistently across projects, which makes the data meaningless. Depending on how you define these four kinds of data, the numbers you come up with for each can vary by a factor of 2 or more.

### Issues Related to Size Measures

You can measure the size of completed projects in Function Points, stories, Web pages, database tables, and numerous other ways, but most organizations eventually settle on capturing size-related historical data in terms of lines of code. (More details on the strengths and weaknesses of using LOC measurements are discussed in Section 18.1, "Challenges with Estimating Size.")

For size in lines of code, you'll need to define several issues, including the following:

- Do you count all code or only code that's included in the released software? (For example, do you count scaffolding code, mock object code, unit test code, and system test code?)

- How do you count code that's reused from previous versions?
- How do you count open source code or third-party library code?
- Do you count blank lines and comments, or only non-blank, non-comment source lines?
- Do you count class interfaces?
- Do you count data declarations?
- How do you count lines that make up one logical line of code but that are broken across multiple lines for the sake of readability?

There isn't any industry standard on this topic, and it doesn't particularly matter how you answer these questions.[1] What does matter is that you answer these questions consistently across projects so that whatever assumptions are baked into the data you collected is *consciously* projected forward in your estimates.

## Issues Related to Effort Measures

Similar cautions apply to collecting effort data:

- Do you count time in hours, days, or some other unit?
- How many hours per day do you count? Standard 8 hours or actual hours applied to the specific project?
- Do you count unpaid overtime?
- Do you count holidays, vacation, and training?
- Do you make allowances for all-company meetings?
- What kinds of effort do you count? Testing? First-level management? Documentation? Requirements? Design? Research?
- How do you count time that's divided across multiple projects?
- How do you count time spent supporting previous releases of the same software?
- How do you count time spent supporting sales calls, trade shows, and so on?
- How do you count travel time?
- How do you count fuzzy front-end time—the time spent firming up the software concept before the project is fully defined?

[1] The closest the software industry has to a standard definition is a non-blank, non-comment, deliverable source statement that includes interfaces and data declarations. That definition still leaves a few of the questions unanswered, such as how to count code reused from previous projects.

Again, the main goal here is to define the data you're collecting well enough so that you know what you're estimating. If your data from past projects includes a high percentage of unpaid overtime and you use that historical data to estimate a future project, guess what? You've just calibrated a high percentage of overtime into your future project.

## Issues Related to Calendar Time Measures

It's surprisingly difficult in many organizations to determine how long a particular project lasted.

- When does the project start? Does it start when it gets formal budget approval? Does it start when initial discussions about the project begin? Does it start when it's fully staffed? Capers Jones reports that fewer than 1% of projects have a clearly defined starting point (Jones 1997).
- When does the project end? Does it end when the software is released to the customer? When the final release candidate is delivered to testing? What if most programmers have rolled off the project a month before the official release? Jones reports that 15% of projects have ambiguous end times (Jones 1997).

In this area, it's very helpful if the organization has well-defined project launch and project completion milestones. The main goal, again, is simply to understand the data you're collecting.

## Issues Related to Defect Measures

Finally, defect measures also vary by a factor of 2 or 3 depending on what's counted as a defect:

- Do you count all change requests as defects, or only those that are ultimately classified as defects rather than feature requests?
- Do you count multiple reports of the same defect as a single defect or as multiple defects?
- Do you count defects that are detected by developers, or only those detected by testers?
- Do you count requirements and design defects that are found prior to the beginning of system testing?
- Do you count coding defects that are found prior to the beginning of alpha or beta testing?
- Do you count defects reported by users after the software has been released?

**Tip #37** In collecting historical data to use for estimation, start small, be sure you understand what you're collecting, and collect the data consistently.

## Other Data Collection Issues

Historical data tends to be easiest to collect if it's collected while the project is underway. It's difficult to go back six months after a project has been completed and reconstruct the "fuzzy front end" of the project to determine when the project began. It's also easy to forget how much overtime people worked at the end of the project.

**Tip #38** Collect a project's historical data *as soon as possible* after the end of the project.

While it's useful to collect data at the end of a project, it's even more useful to collect snapshots of a project as it's underway. Collecting data on size, effort, and defects every 1 to 2 weeks can provide valuable insight into your project's dynamics.

For example, collecting a snapshot of reported defects can help you predict the rate at which defects will be discovered and will need to be fixed on future projects. Collecting data on effort over time can help you understand your organization's ability to mobilize staff to support a project. If one project staffs up more slowly than desired, it might be a fluke. If your historical data says that the last three projects have each staffed up at about the same rate, that suggests that you're facing an organizational influence that can't easily be changed on the next project.

**Tip #39** As a project is underway, collect historical data on a periodic basis so that you can build a data-based profile of how your projects run.

## 8.3 How to Calibrate

The ultimate goal of collecting data is to convert the data to a model that you can use for estimation. Here are some examples of models you could create:

- Our developers average *X* lines of code per staff month.
- A 3-person team can deliver *X* stories per calendar month.
- Our team is averaging *X* staff hours per use case to create the use case, and *Y* hours per use case to construct and deliver the use case.
- Our testers create test cases at a rate of *X* hours per test case.
- In our environment, we average *X* lines of code per function point in C# and *Y* lines of code per function point in Python.
- On this project so far, defect correction work has averaged *X* hours per defect.

These are just examples to illustrate the kinds of models you can build using historical data. Table 7-1 in the previous chapter listed many more examples.

One characteristic these models have in common is that they are all linear. The math works the same whether you're building a 10,000-LOC system or a 1,000,000-LOC system. But because of software's diseconomies of scale, some models will need to be adjusted for different size ranges.You could try to handle the size differentiation informally. Table 8-1 shows one example of how you might do that.

**Table 8-1 Example of Accounting for Diseconomies of Scale Informally—For Purposes of Illustration Only**

| Team Size | Average Stories Delivered per Calendar Month |
|---|---|
| 1 | 5 |
| 2–3 | 12 |
| 4–5 | 22 |
| 6–7 | 31 |
| 8 | No data for projects of this size |

This approach is valid when you have small variations in project size. To account for larger variations in project size, see Section 5.1, "Project Size," and Section 5.6, "Diseconomies of Scale Revisited."

## 8.4 Using Project Data to Refine Your Estimates

Earlier in this chapter, I pointed out that historical data is useful because it accounts for organizational influences–both recognized and unrecognized. The same idea applies to the use of historical data within a specific project (Gilb 1988, Cohn 2006). Individual projects have dynamics that will vary somewhat from the dynamics of their surrounding organizations. Using data from the project itself will account for the influences that are unique to that specific project. The sooner on a project you can begin basing your estimates on data from the project itself, the sooner your estimates will become truly accurate.

**Tip #40** Use data from your current project (project data) to create highly accurate estimates for the remainder of the project.

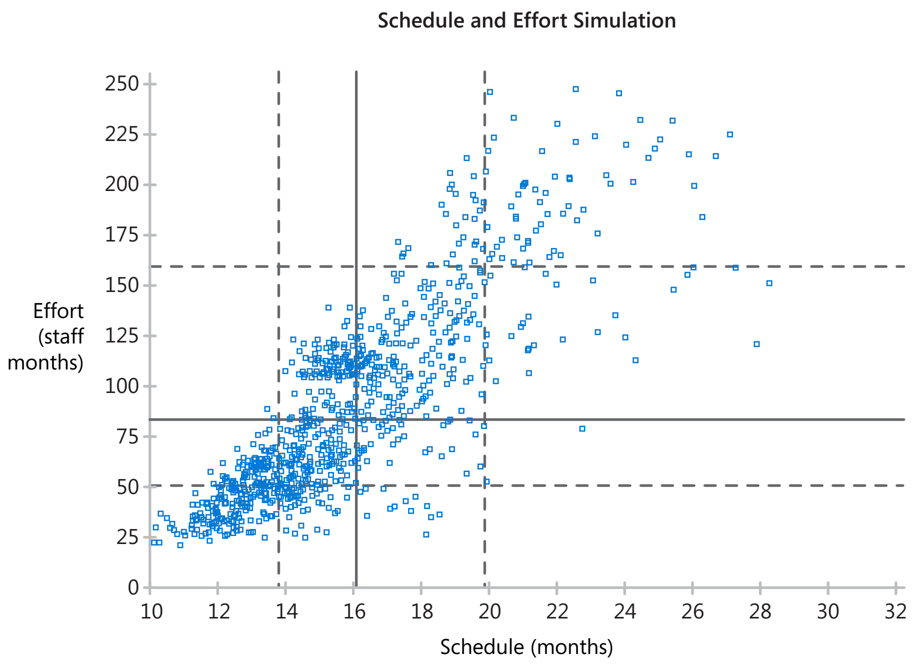
Even if you don't have historical data from past projects, you can collect data from your current project and use that as a basis for estimating the remainder of your project. Your goal should be to switch from using organizational data or industry-average data to project data as soon as you can. The more iterative your project is, the sooner you'll be able to do this.

Collecting and using data from your own project will be discussed in more detail in Section 16.4, “Estimate Refinement.” Section 12.3, “Story Points,” presents a specific example of using project data to refine your estimates.

## 8.5 Calibration with Industry Average Data

If you don’t have your own historical data, you have little choice but to use industry-average data, which is adequate but no better. As Table 5-2 illustrated, the productivity rates for different organizations within the same industries typically vary by a factor of 10. If you use the average productivity for your industry, you won’t be accounting for the possibility that your organization might be at the top end of the productivity range or at the bottom.

Figure 8-1 shows an example of an estimate created using industry-average data. Each point in the graph represents a possible project outcome created using a statistical technique known as a Monte Carlo simulation. The solid black lines represent the median effort and schedule found during the simulation. The dashed black lines represent the 25th and 75th percentiles for effort and schedule.



*Source: Estimated prepared using Construx Estimate, available at* www.construx.com/estimate.

Figure 8-1 An example of estimated outcomes for an estimate calibrated using industry-average data. Total variation in the effort estimates is about a factor of 10 (from about 25 staff months to about 250 staff months).

Figure 8-2 shows an example of a comparable estimate calibrated using historical data from one of my clients.

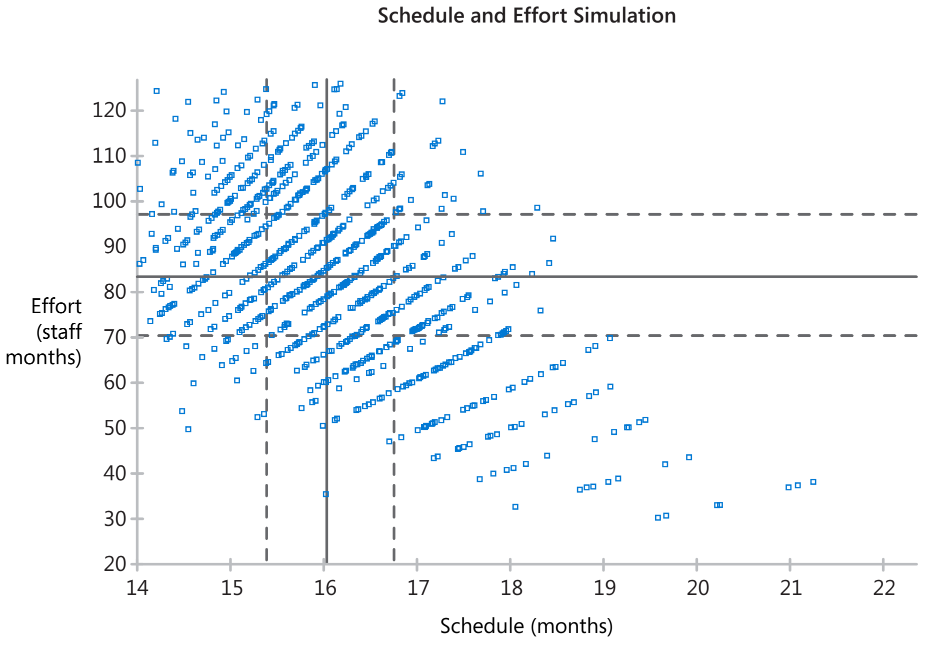


Figure 8-2 An estimate calibrated using historical productivity data. The effort estimates vary by only about a factor of 4 (from about 30 staff months to about 120 staff months).

The sizes and nominal productivity rates of the two projects are identical, but the amount of variability in the two estimates is dramatically different. Because the industry-average estimate must account for factor-of-10 differences in productivity, the standard deviation on the effort estimate created using industry-average data is about 100%! If you wanted to give your boss an estimate that ranged from 25% confident to 75% confident using industry-average data, in this case you'd need to quote a range of 50 to 160 staff months—a factor of 3 difference!

If you could use historical data instead of industry-average data, you could quote a range of 70 to 95 staff months—a factor of only 1.4 from the top end of the range to the bottom. The standard deviation in the estimate created using historical data is only about 25%.

A review of studies on estimation accuracy found that in studies in which estimation models were not calibrated to the estimation environment, expert estimates were more accurate than the models. But the studies that used models calibrated with historical data found that the models were as good as or better than expert estimates (Jørgensen 2002).

**Tip #41** Use project data or historical data rather than industry-average data to calibrate your estimates whenever possible. In addition to making your estimates more accurate, historical data will reduce variability in your estimate arising from uncertainty in the productivity assumptions.

## 8.6 Summary

If you haven't previously been exposed to the power of historical data, you can be excused for not currently having any data to use for your estimates. But now that you know how valuable historical data is, you don't have any excuse not to collect it. Be sure that when you reread this chapter next year you're not still saying, "I wish I had some historical data!"

**Tip #42** If you don't currently have historical data, begin collecting it as soon as possible.

## Additional Resources

Boehm, Barry, et al. *Software Cost Estimation with Cocomo II*. Reading, MA: Addison-Wesley, 2000. Appendix E of Boehm's book contains a checklist that's useful for precisely defining what constitutes a "line of code."

Gilb, Tom. *Principles of Software Engineering Management*. Wokingham, England: Addison-Wesley, 1988. Section 7.14 of Gilb's book describes using project-specific data to refine estimates. The description of evolutionary delivery throughout the book is based on the expectation that projects will build feedback loops allowing them to be estimated, planned, and managed in a way that allows the projects to be self-correcting.

Grady, Robert B. and Deborah L. Caswell. *Software Metrics: Establishing a Company-Wide Program*. Englewood Cliffs, NJ: Prentice Hall, 1987. This book and the following one describe Grady's experiences setting up a measurement program at Hewlett-Packard. The books contain many hard-won insights into the pitfalls of setting up a measurement program plus some interesting examples of the useful data you can ultimately obtain.

Grady, Robert B. *Practical Software Metrics for Project Management and Process Improvement*. Englewood Cliffs, NJ: PTR Prentice Hall, 1992.

Jones, Capers. *Applied Software Measurement: Assuring Productivity and Quality, 2d Ed.* New York, NY: McGraw-Hill, 1997. Chapter 3 of this book presents an excellent discussion of the sources of errors in size, effort, and quality measurements.

Putnam, Lawrence H. and Ware Myers. *Five Core Metrics*. New York, NY: Dorset House, 2003. This book presents a compelling argument for collecting data on the five core metrics of size, productivity, time, effort, and reliability.

Software Engineering Institute's *Software Engineering Measurement and Analysis (SEMA)* Web site: *www.sei.cmu.edu/sema/*. This comprehensive Web site helps organizations create data collection (measurement) practices and use the data they collect.

Chapter 9

# Individual Expert Judgment

## Applicability of Techniques in This Chapter

| | Use of Structured Process | Use of Estimation Checklist | Estimating Task Effort in Ranges | Comparing Task Estimates to Actuals |
|---|---|---|---|---|
| **What's estimated** | Effort, Schedule, Features | Effort, Schedule, Features | Size, Effort, Schedule, Features | Size, Effort, Schedule, Features |
| **Size of project** | S M L | S M L | S M L | S M L |
| **Development stage** | Early–Late | Early–Late | Early–Late | Middle–Late |
| **Iterative or sequential** | Both | Both | Both | Both |
| **Accuracy possible** | High | High | High | N/A |

Individual expert judgment is by far the most common estimation approach used in practice (Jørgensen 2002). Hihn and Habib-agahi found that 83% of estimators used "informal analogy" as their primary estimation technique (Hihn and Habib-agahi 1991). A New Zealand survey found that 86% of software organizations used "expert estimation" (Paynter 1996). Barbara Kitchenham and her colleagues found that 72% of project estimates were based on "expert opinion" (Kitchenham et al. 2002).

Expert-judgment estimates of individual tasks form the foundation for bottom-up estimation, but not all expert judgments are equal. Indeed, as Chapter 7, "Count, Compute, Judge," indicated, judgment is the most hazardous kind of estimation.

When discussing "expert judgment," we need first to ask "expert in what?" Being expert in the technology or development practices that will be employed does not make someone an expert in estimation. Magne Jørgensen reports that increased experience in the activity being estimated does not lead to increased accuracy in the estimates for the activity (Jørgensen 2002). Other studies have found that "experts" tend to use simple estimation strategies, even when their level of expertise in the subject being estimated is high (Josephs and Hahn 1995, Todd and Benbasat 2000).

This chapter describes how to ensure that, when you use expert judgment, the judgment is effective. The discussion in this chapter is closely related to the discussion in

Chapter 10, "Decomposition and Recomposition," which explains how to combine the individual estimates accurately.

## 9.1 Structured Expert Judgment

Individual expert judgment does not have to be informal or intuitive. Researchers have found significant accuracy differences between "intuitive expert judgment," which tends to be inaccurate (Lederer and Prasad 1992) and "structured expert judgment," which can produce estimates that are about as accurate as model-based estimates (Jørgensen 2002).

### Who Creates the Estimates?

For the estimation of specific tasks—such as the time needed to code and debug a particular feature or to create a specific set of test cases—the people who will actually do the work will create the most accurate estimates. Estimates prepared by people who aren't doing the work are less accurate (Lederer and Prasad 1992). In addition, separate estimators are more likely to underestimate than estimator-developers are (Lederer and Prasad 1992).

**Tip #43** To create the task-level estimates, have the people who will actually do the work create the estimates.

This guideline is for task-level estimates. If your project is still in the wide part of the Cone of Uncertainty (that is, specific tasks haven't yet been identified or assigned to individuals), the estimate should be created by an expert estimator or by the most expert development, quality assurance, and documentation staff available.

### Granularity

One of the best ways to improve the accuracy of task-level estimates is to separate large tasks into smaller tasks. When creating estimates, developers, testers, and managers tend to concentrate on the tasks that they understand and deemphasize tasks that are unfamiliar to them. The common result is that a 1-line entry on the schedule, such as "data conversion," which was supposed to take 2 weeks, instead takes 2 months because no one investigated what was actually involved.

When estimating at the task level, decompose estimates into tasks that will require no more than about 2 days of effort. Tasks larger than that will contain too many places that unexpected work can hide. Ending up with estimates that are at the 1/4 day, 1/2 day, or full day of granularity is appropriate.

## Use of Ranges

If you ask a developer to estimate a set of features, the developer will often come back with an estimate that looks like Table 9-1.

Table 9-1 **Example of Developer Single-Point Estimates**

| Feature | Estimated Days to Complete |
|---|---|
| Feature 1 | 1.5 |
| Feature 2 | 1.5 |
| Feature 3 | 2.0 |
| Feature 4 | 0.5 |
| Feature 5 | 0.5 |
| Feature 6 | 0.25 |
| Feature 7 | 2.0 |
| Feature 8 | 1.0 |
| Feature 9 | 0.75 |
| Feature 10 | 1.25 |
| **TOTAL** | **11.25** |

If you then ask the same developer to reestimate each feature's best case and worst case, the developer will often return with estimates similar to those in Table 9-2.

Table 9-2 **Example of Individual Estimation Using Best Case and Worst Case**

| | Estimated Days to Complete | |
|---|---|---|
| **Feature** | **Best Case** | **Worst Case** |
| Feature 1 | 1.25 | 2.0 |
| Feature 2 | 1.5 | 2.5 |
| Feature 3 | 2.0 | 3.0 |
| Feature 4 | 0.75 | 2.0 |
| Feature 5 | 0.5 | 1.25 |
| Feature 6 | 0.25 | 0.5 |
| Feature 7 | 1.5 | 2.5 |
| Feature 8 | 1.0 | 1.5 |
| Feature 9 | 0.5 | 1.0 |
| Feature 10 | 1.25 | 2.0 |
| **TOTAL** | 10.5[1] | 18.25 |

[1] Some statistical anomalies arise when you simply total the Best Case estimates and the Worst Case estimates. Chapter 10 discusses these in detail.

When you compare the original single-point estimates to the Best Case and Worst Case estimates, you see that the 11.25 total of the single-point estimates is much closer to the Best Case estimate of 10.5 days than to the Worst Case total of 18.25 days.

If you examine the estimate for Feature 4, you'll also notice that both the Best Case and the Worst Case estimates are higher than the original single-point estimate. Thinking through the worst case result sometimes exposes additional work that must be done even in the best case, which can raise the nominal estimate. In thinking through the worst case, I like to ask developers how long the task would take if *everything* went wrong. People's worst cases are often optimistic worst cases rather than *true* worst cases.

If you're a manager or a lead, have your developers create a set of single-point estimates. Hide those estimates from them. Then have the developers create a set of Best Case and Worst Case estimates. Have them compare their Best Case and Worst Case estimates to their original single-point estimates. This is often an eye-opening experience.

This exercise yields two benefits. First, it raises awareness that single-point estimates tend to be akin to Best Case estimates. Second, going through the process of writing down Best Case and Worst Case estimates a few times begins to engrain the habit of thinking through the worst case outcome when estimating. Once you get into the habit of considering both best case and worst case outcomes, you'll get better at factoring the full range of possible outcomes into your single-point task estimates, regardless of whether you actually write down the best and worst cases.

**Tip #44** Create both Best Case and Worst Case estimates to stimulate thinking about the full range of possible outcomes.

## Formulas

Creating the Best Case and the Worst Case estimates is just the first step. You're still left with the question of which estimate to use. Or maybe you should use the mathematical midpoint instead? The answer is none of the above. In many cases, the Worst Case is much worse than what's called the Expected Case. Taking the midpoints of the ranges could result in an unnecessarily high estimate.

A technique called the Program Evaluation and Review Technique (PERT) allows you to compute an Expected Case that might not be exactly in the middle of the range from best case to worst case (Putnam and Myers 1992, Stutzke 2005). To use PERT, you add an additional Most Likely Case to your set of cases. You can estimate the

Most Likely Case using expert judgment. You then calculate the Expected Case using this formula:

**Equation #1**

$$\text{Expected Case} = [\text{BestCase} + (4 \times \text{MostLikelyCase}) + \text{WorstCase}] / 6$$

This formula accounts for the full width of the range as well as the position of the Most Likely Case within the range. Table 9-3 shows the estimates from Table 9-2 with the addition of Most Likely Case and Expected Case. As you can see from the table, the overall estimate of 13.62 is closer to the lower end of the range than the midpoint of 14.4 would have been.

Table 9-3 **Example of Individual Estimation Using Best Case, Worst Case, and Most Likely Case**

| | Estimated Days to Complete | | | |
|---|---|---|---|---|
| **Feature** | **Best Case** | **Most Likely Case** | **Worst Case** | **Expected Case** |
| Feature 1 | 1.25 | 1.5 | 2.0 | **1.54** |
| Feature 2 | 1.5 | 1.75 | 2.5 | **1.83** |
| Feature 3 | 2.0 | 2.25 | 3.0 | **2.33** |
| Feature 4 | 0.75 | 1 | 2.0 | **1.13** |
| Feature 5 | 0.5 | 0.75 | 1.25 | **0.79** |
| Feature 6 | 0.25 | 0.5 | 0.5 | **0.46** |
| Feature 7 | 1.5 | 2 | 2.5 | **2.00** |
| Feature 8 | 1.0 | 1.25 | 1.5 | **1.25** |
| Feature 9 | 0.5 | 0.75 | 1.0 | **0.75** |
| Feature 10 | 1.25 | 1.5 | 2.0 | **1.54** |
| **TOTAL** | **10.5** | **13.25** | **18.25** | **13.62** |

As discussed in Chapter 4, "Where Does Estimation Error Come From?" people's "most likely" estimates tend to be optimistic, which can yield optimistic overall estimates when using this approach. Some estimation experts suggest altering the basic PERT formula to account for a downward bias in the estimates (Stutzke 2005). Here's the altered formula:

**Equation #2**

$$\text{Expected Case} = [\text{BestCase} + (3 \times \text{MostLikelyCase}) + (2 \times \text{WorstCase})] / 6$$

This is a reasonable short-term solution to the problem. The long-term solution to this problem is to work with people to make their Most Likely Case estimates more accurate.

## Checklists

Even experts occasionally forget to consider everything they should. Studies of forecasting in a variety of disciplines have found that simple checklists help improve accuracy by reminding people of considerations they might otherwise forget (Park 1996, Harvey 2001, Jørgensen 2002). Table 9-4 presents a checklist you might use to improve the accuracy of your estimates.

**Table 9-4 Checklist for Individual Estimates**

1. Is what's being estimated clearly defined?
2. Does the estimate include all the *kinds of work* needed to complete the task?
3. Does the estimate include all the *functionality areas* needed to complete the task?
4. Is the estimate broken down into enough detail to expose hidden work?
5. Did you look at documented facts (written notes) from past work rather than estimating purely from memory?
6. Is the estimate approved by the person who will actually do the work?
7. Is the productivity assumed in the estimate similar to what has been achieved on similar assignments?
8. Does the estimate include a Best Case, Worst Case, and Most Likely Case?
9. Is the Worst Case really the worst case? Does it need to be made even worse?
10. Is the Expected Case computed appropriately from the other cases?
11. Have the assumptions in the estimate been documented?
12. Has the situation changed since the estimate was prepared?

To avoid omitting work from your estimates, you might also review the lists of overlooked activities in Section 4.5, "Omitted Activities."

**Tip #45** Use an estimation checklist to improve your individual estimates. Develop and maintain your own personal checklist to improve your estimation accuracy.

## 9.2 Compare Estimates to Actuals

Prying yourself loose from single-point/Best Case estimates is half the battle. The other half is comparing your actual results to your estimated results so that you can refine your personal estimating abilities.

Keep a list of your estimates, and fill in your actual results when you complete them. Then compute the Magnitude of Relative Error (MRE) of your estimates (Conte, Dunsmore, and Shen 1986). MRE is computed using this formula:

**Equation #3**

$$\text{MRE} = \text{AbsoluteValue} \times [(\text{ActualResult} - \text{EstimatedResult}) / \text{ActualResult}]$$

Table 9-5 shows how the MRE calculations would work out for the Best Case and Worst Case estimates presented earlier.

**Table 9-5** **Table 9-5 Example of Spreadsheet for Tracking Accuracy of Individual Estimates**

| | Estimated Days to Complete | | | | | |
|---|---|---|---|---|---|---|
| **Feature** | **Best Case** | **Worst Case** | **Expected Case** | **Actual Outcome** | **MRE** | **In Range from Best Case to Worst Case?** |
| Feature 1 | 1.25 | 2 | 1.54 | 2 | 23% | Yes |
| Feature 2 | 1.5 | 2.5 | 1.83 | 2.5 | 27% | Yes |
| Feature 3 | 2 | 3 | 2.33 | 1.25 | 87% | No |
| Feature 4 | 0.75 | 2 | 1.13 | 1.5 | 25% | Yes |
| Feature 5 | 0.5 | 1.25 | 0.79 | 1 | 21% | Yes |
| Feature 6 | 0.25 | 0.5 | 0.46 | 0.5 | 8% | Yes |
| Feature 7 | 1.5 | 2.5 | 2.00 | 3 | 33% | No |
| Feature 8 | 1 | 1.5 | 1.25 | 1.5 | 17% | Yes |
| Feature 9 | 0.5 | 1 | 0.75 | 1 | 25% | Yes |
| Feature 10 | 1.25 | 2 | 1.54 | 2 | 23% | Yes |
| **TOTAL** | **10.50** | **18.25** | **13.625** | **16.25** | | **80% Yes** |
| **Average** | | | | | **29%** | |

In this spreadsheet, the MRE is calculated for each estimate. The average MRE, shown in the bottom row, is 29% for the set of estimates. You can use this average MRE to measure the accuracy of your estimates. As your estimates improve, you should see the MRE decline. The right-most column shows how many estimates are within the best case/worst case range. You should also see the percentage of estimates that fall within the range increase over time.

**Tip #46** Compare actual performance to estimated performance so that you can improve your individual estimates over time.

When you compare your actual performance to your estimates, you should try to understand what went right, what went wrong, what you overlooked, and how to avoid making those mistakes in the future.

Another practice that sets up a feedback loop and encourages accurate estimates is a public estimation review. I've worked with companies that have their developers report on their actual results versus their estimates at a Monday morning standup meeting. This reinforces the idea that accurate estimates are an organizational priority.

Regardless of how you do it, the key principle is to set up a feedback loop based on actual results so that your estimates improve over time. To be effective, the feedback should be as timely as possible; delay reduces effectiveness of the feedback loop (Jørgensen 2002).

## Additional Resources

Jørgensen, M. "A Review of Studies on Expert Estimation of Software Development Effort." 2002. This paper presents a comprehensive review of the research on expert estimation approaches. The author draws numerous conclusions from the common research threads and presents 12 tips for achieving accurate expert estimates.

Humphrey, Watts S. *A Discipline for Software Engineering*. Reading, MA: Addison-Wesley, 1995. Humphrey lays out a detailed methodology by which developers can collect personal productivity data, compare their planned results to their actual results, and improve over time.

Stutzke, Richard D. *Estimating Software-Intensive Systems*. Upper Saddle River, NJ: Addison-Wesley, 2005. Chapter 5 of Stutzke's book discusses judgment-based estimation techniques and provides background on some of the math described in this chapter.

Chapter 10

# Decomposition and Recomposition

## Applicability of Techniques in This Chapter

| | Decomposition by Feature or Task | Decomposition by Work Breakdown Structure (WBS) | Computing Best and Worst Cases from Standard Deviation |
|---|---|---|---|
| **What's estimated** | Size, Effort, Features | Effort | Effort, Schedule |
| **Size of project** | S M L | - M L | S M L |
| **Development stage** | Early–Late (small projects); Middle–Late (medium and large projects) | Early–Middle | Early–Late (small projects); Middle–Late (medium and large projects) |
| **Iterative or sequential** | Both | Both | Both |
| **Accuracy possible** | Medium–High | Medium | Medium |

Decomposition is the practice of separating an estimate into multiple pieces, estimating each piece individually, and then recombining the individual estimates into an aggregate estimate. This estimation approach is also known as "bottom up," "micro estimation," "module build up," "by engineering procedure," and by many other names (Tockey 2005).

Decomposition is a cornerstone estimation practice—as long as you watch out for a few pitfalls. This chapter discusses the basic practice in more detail and explains how to avoid such pitfalls.

## 10.1 Calculating an Accurate Overall Expected Case

Scene: The weekly team meeting…

> *YOU: We need to create an estimate for a new project. I want to emphasize how important accurate estimation is to this group, and so I'm betting a pizza lunch that I can create a more accurate estimate for this project than you can. If you win, I'll buy the pizza. If I win, you'll buy. Any takers?*
>
> *TEAM: You're on!*
>
> *YOU: OK, let's get started.*

You look up information about a similar past project, and you find that that project took 18 staff weeks. You estimate that this project is about 20 percent larger than the past project, so you create a total estimate of 22 staff weeks.

Meanwhile, your team has created a more detailed, feature-by-feature estimate. They come back with the estimate shown in Table 10-1.

**Table 10-1 Example of Estimation by Decomposition**

| Feature | Estimated Staff Weeks to Complete |
|---|---|
| Feature 1 | 1.5 |
| Feature 2 | 4 |
| Feature 3 | 3 |
| Feature 4 | 1 |
| Feature 5 | 4 |
| Feature 6 | 6 |
| Feature 7 | 2 |
| Feature 8 | 1 |
| Feature 9 | 3 |
| Feature 10 | 1.5 |
| **TOTAL** | **27** |

> *YOU: 27 weeks? Wow, I think your estimate is high, but I guess we'll find out.*

A few weeks later…

> *YOU: Now that the project is done, we know that it took a total of 29 staff weeks. It looks like your estimate of 27 staff weeks was optimistic by 2 weeks, which is an error of 7%. My estimate of 22 staff weeks was off by 7 staff weeks, about 24%. It looks like you win, so I'm buying the pizza.*
>
> *By the way, I want to see which of you good estimators cost me the pizza. Let's take a look at which detailed estimates were the most accurate.*

You take a few minutes to compute the magnitude of relative error of each individual estimate and write the results on the whiteboard. Table 10-2 shows the results.

**Table 10-2 Example Results of Estimation by Decomposition**

| Feature | Estimated Staff Weeks to Complete | Actual Effort | Raw Error | Magnitude of Relative Error |
|---|---|---|---|---|
| Feature 1 | 1.5 | 3.0 | –1.5 | 50% |
| Feature 2 | 4.5 | 2.5 | 2.0 | 80% |
| Feature 3 | 3 | 1.5 | 1.5 | 100% |
| Feature 4 | 1 | 2.5 | –1.5 | 60% |

Table 10-2 **Example Results of Estimation by Decomposition**

| Feature | Estimated Staff Weeks to Complete | Actual Effort | Raw Error | Magnitude of Relative Error |
|---|---|---|---|---|
| Feature 5 | 4 | 4.5 | –0.5 | 11% |
| Feature 6 | 6 | 4.5 | 1.5 | 33% |
| Feature 7 | 2 | 3.0 | –1.0 | 33% |
| Feature 8 | 1 | 1.5 | –0.5 | 33% |
| Feature 9 | 3 | 2.5 | 0.5 | 20% |
| Feature 10 | 1.5 | 3.5 | –2.0 | 57% |
| **TOTAL** | **27** | **29** | **–2** | - |
| **Average** | - | - | **–7%** | **46%** |

*TEAM: Wow, that's interesting. Most of our individual estimates weren't any more accurate than yours. Our estimates were nearly all wrong by 30% to 50% or more. Our average error was 46%—which is way higher than your error. But our overall error was still only 7% and yours was 24%.*

*But the joke is on you. Even though our estimates were worse than yours, you're still buying the pizza!*

Somehow the team's estimate was more accurate than your estimate even though their individual feature estimates were worse. How is that possible?

## The Law of Large Numbers

The team's estimate benefited from a statistical property called the Law of Large Numbers. The gist of this law is that if you create one big estimate, the estimate's error tendency will be completely on the high side or completely on the low side. But if you create several smaller estimates, some of the estimation errors will be on the high side, and some will be on the low side. The errors will tend to cancel each other out to some degree. Your team underestimated in some cases, but it also overestimated in some cases, so the error in the aggregate estimate is only 7%. In your estimate, all 24% of the error was on the same side.

This approach should work in theory, and research says that it also works in practice. Lederer and Prasad found that summing task durations was negatively correlated with cost and schedule overruns (Lederer and Prasad 1992).

**Tip #47**

Decompose large estimates into small pieces so that you can take advantage of the Law of Large Numbers: the errors on the high side and the errors on the low side cancel each other out to some degree.

## How Small Should the Estimated Pieces Be?

Seen from the perspective shown in Figure 10-1, software development is a process of making larger numbers of steadily smaller decisions. At the beginning of the project, you make such decisions as “What major *areas* should this software contain?” A simple decision to include or exclude an area can significantly swing total project effort and schedule in one direction or another. As you approach top-level requirements, you make a larger number of decisions about which features should be in or out, but each of those decisions on average exerts a smaller impact on the overall project outcome. As you approach detailed requirements, you typically make hundreds of decisions, some with larger implications and some with smaller implications, but on average the impact of these decisions is far smaller than the impact of the decisions made earlier in the project.

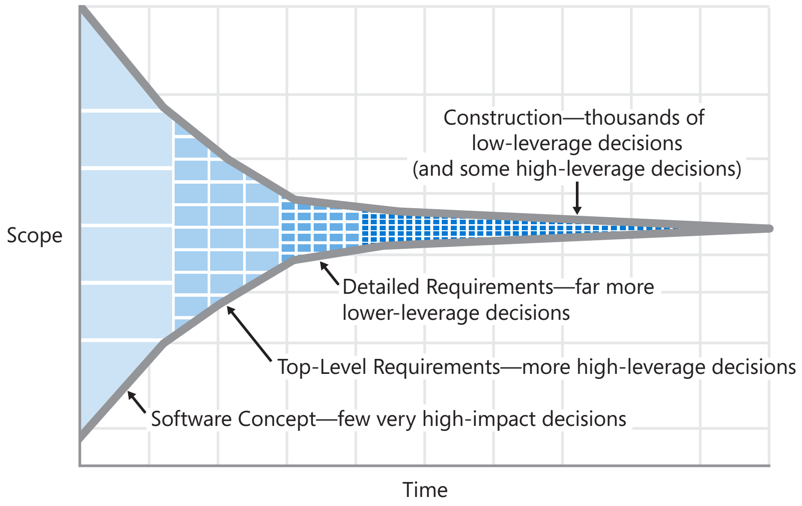

Construction—thousands of low-leverage decisions (and some high-leverage decisions)
Scope
Detailed Requirements—far more lower-leverage decisions
Top-Level Requirements—more high-leverage decisions
Software Concept—few very high-impact decisions
Time

Figure 10-1 Software projects tend to progress from large-grain focus at the beginning to fine-grain focus at the end. This progression supports increasing the use of estimation by decomposition as a project progresses.

By the time you focus on software construction, the granularity of the decisions you make is tiny: “How should I design this class interface? How should I name this variable? How should I structure this loop?” And so on. These decisions are still important, but the effect of any single decision tends to be localized compared with the big decisions that were made at the initial, software-concept level.

The implication of software development being a process of steady refinement is that the further into the project you are, the finer-grained your decomposed estimates can be. Early in the project, you might base a bottom-up estimate on feature areas. Later, you might base the estimate on marketing requirements. Still later, you might use detailed requirements or engineering requirements. In the project’s endgame, you might use developer and tester task-based estimates.

The limits on the number of items to estimate are more practical than theoretical. Very early in a project, it can be a struggle to get enough detailed information to create a decomposed estimate. Later in the project, you might have too much detail. You need 5 to 10 individual items before you get much benefit from the Law of Large Numbers, but even 5 items are better than 1.

## 10.2 Decomposition via an Activity-Based Work Breakdown Structure

Sometimes unseen work hides in the form of forgotten features. Sometimes it hides in the form of forgotten tasks. Decomposing a project via an activity-based work breakdown structure (WBS) helps you avoid forgetting tasks. It also helps fine-tune thinking about whether the project you're estimating is bigger or smaller than similar past projects. Comparing the new project to the old project in each WBS category can sharpen your assessment of which parts are bigger and which are smaller.

Table 10-3 shows a generic, activity-based WBS for a small-to-medium-sized software project. The left column lists the category of activities such as Planning, Requirements, Coding, and so on. The other columns list the kinds of work within each categories, such as Creating, Planning, Reviewing, and so on.

**Table 10-3 Generic Work Breakdown Structure for a Small-to-Medium-Sized Software Project**

| Category | Create/ Do | Plan | Manage | Review | Rework | Report Defects |
|---|---|---|---|---|---|---|
| General management | ● | ● | ● | ● | | |
| Planning | ● | | ● | ● | ● | |
| Corporate activities (meetings, vacation, holidays, and so on) | ● | | | | | |
| Hardware setup/Software setup/Maintenance | ● | ● | ● | ● | ● | ● |
| Staff preparation | ● | ● | ● | ● | | |
| Technical Processes/Practices | ● | ● | ● | ● | ● | ● |
| Requirements work | ● | ● | ● | ● | ● | ● |
| Coordinate with other projects | ● | ● | ● | ● | | |
| Change management | ● | ● | ● | ● | ● | ● |
| User-interface prototyping | ● | ● | ● | ● | ● | ● |
| Architecture work | ● | ● | ● | ● | ● | ● |
| Detailed designing | ● | ● | ● | ● | ● | ● |
| Coding | ● | ● | ● | ● | ● | ● |
| Component acquisition | ● | ● | ● | ● | ● | ● |
| Automated build | ● | ● | ● | ● | ● | ● |

Table 10-3 **Generic Work Breakdown Structure for a Small-to-Medium-Sized Software Project**

| Category | Create/ Do | Plan | Manage | Review | Rework | Report Defects |
|---|---|---|---|---|---|---|
| Integration | ● | ● | ● | ● | ● | ● |
| Manual system tests | ● | ● | ● | ● | ● | ● |
| Automated system tests | ● | ● | ● | ● | ● | ● |
| Software release (interim, alpha, beta, and final releases) | ● | ● | ● | ● | ● | ● |
| Documents (user docs, technical docs) | ● | ● | ● | ● | ● | ● |

To use the generic WBS, you combine the column descriptions with the categories—for example, Create/Do Planning, Manage Planning, Review Planning, Create/Do Requirements Work, Manage Requirements Work, Review Requirements Work, Create/Do Coding, Manage Coding, Review Coding, and so on. The dots in the table represent the most common combinations.

This WBS presents an extensive list of the kinds of activities that you might consider when creating an estimate. You will probably need to extend the list to include at least a few additional entries related to specifics of your organization's software-development approach. You might also decide to exclude some of this WBS's categories, which will be fine as long as that's a conscious decision.

**Tip #48** Use a generic software-project work breakdown structure (WBS) to avoid omitting common activities.

## 10.3 Hazards of Adding Up Best Case and Worst Case Estimates

Have you ever had the following experience? You put together a detailed task list. You carefully estimate each of the tasks on the list, thinking, "We can pull this off if we try hard enough." After you go through meticulous planning, you work hard on the first task and deliver it on time. The second task turns up some unexpected problems, but you work late and get it done on schedule. The third task turns up a few more problems, and you leave it unfinished at the end of the day, thinking you'll polish it off the next morning. By the end of the next day, you've barely finished that task, and haven't yet started the task you were supposed to do that day. By the end of the week, you're more than a full task behind schedule.

How did that happen? Were your estimates wrong, or did you just not perform very well?

## Warning: Math Ahead!

The answer lies in some of the statistical subtleties involved in combining individual estimates. *Statistical subtleties?* Yes, for better or worse, this is an area in which we must dig into the mathematics a little to understand how to avoid common problems associated with building up an estimate from decomposed task or feature estimates.

## What Went Wrong?

To see what happened in the preceding scenario, let's take another look at the case study from the beginning of the chapter. The team in the case study produced an accurate estimate. But the accuracy of their single-point estimates was unusual. A more common attempt to produce an estimate by decomposition would not produce the estimates listed in Table 10-1; it would be much more likely to produce estimates such as those shown in Table 10-4.

**Table 10-4 Example of More Typical, Error-Prone Attempt to Estimate by Decomposition**

| Feature | Estimated Staff Weeks to Complete | Actual Effort |
|---|---|---|
| Feature 1 | 1.6 | 3.0 |
| Feature 2 | 1.8 | 2.5 |
| Feature 3 | 2.0 | 1.5 |
| Feature 4 | 0.8 | 2.5 |
| Feature 5 | 3.8 | 4.5 |
| Feature 6 | 3.8 | 4.5 |
| Feature 7 | 2.2 | 3.0 |
| Feature 8 | 0.8 | 1.5 |
| Feature 9 | 1.6 | 2.5 |
| Feature 10 | 1.6 | 3.5 |
| **TOTAL** | **20.0** | **29.0** |

In this example, the accuracy of the 20-staff-week estimate obtained through a simple summation of decomposed, single-point estimates is actually worse than the aggregate estimate of 22 staff weeks that you provided earlier in the case study. How can this be?

The root cause is a combination of the "90% confident" problem that that was discussed in Chapter 1 ("What Is an 'Estimate'?") and the optimism problem discussed in Chapter 4 ("Where Does Estimation Error Come From?"). When developers are asked to provide single-point estimates, they often unconsciously present Best Case estimates. Let's say that each of the individual Best Case estimates is 25% likely, meaning that you have only a 25% chance of doing as well or better than the estimate. The odds of delivering any individual task according to a Best Case estimate are not great: only 1 in 4 (25%). But the odds of delivering *all* the tasks are vanishingly small. To deliver both the first task and the second task on time, you have to beat 1 in 4 odds for the first task and 1 in 4 odds for the second task. Statistically, those odds are multiplied together, so the odds of completing both tasks on time is only 1 in 16. To complete all 10 tasks on time you have to multiply the 1/4s 10 times, which gives you odds of only about 1 in 1,000,000, or 0.000095%. The odds of 1 in 4 might not seem so bad at the individual task level, but the combined odds kill software schedules. The statistics of combining a set of Worst Case estimates work similarly.

These statistical anomalies are another reason to create Best Case, Worst Case, Most Likely Case, and Expected Case estimates, as described in Chapter 9, "Individual Expert Judgment." Table 10-5 shows how that might work out if the developers who produced the estimates in Table 10-4 were asked to produce Best Case, Worst Case, and Most Likely Case estimates, and if the Expected Case estimates were computed from those.

**Table 10-5 Example of Estimation by Decomposition Using Best Case, Expected Case, and Worst Case Estimates**

| | Weeks to Complete | | | |
|---|---|---|---|---|
| **Feature** | **Best Case (25% Likely)** | **Most Likely Case** | **Worst Case (75% Likely)** | **Expected Case (50% Likely)** |
| Feature 1 | 1.6 | 2.0 | 3.0 | 2.10 |
| Feature 2 | 1.8 | 2.5 | 4.0 | 2.63 |
| Feature 3 | 2.0 | 3.0 | 4.2 | 3.03 |
| Feature 4 | 0.8 | 1.2 | 1.6 | 1.20 |
| Feature 5 | 3.8 | 4.5 | 5.2 | 4.50 |
| Feature 6 | 3.8 | 5.0 | 6.0 | 4.97 |
| Feature 7 | 2.2 | 2.4 | 3.4 | 2.53 |
| Feature 8 | 0.8 | 1.2 | 2.2 | 1.30 |
| Feature 9 | 1.6 | 2.5 | 3.0 | 2.43 |
| Feature 10 | 1.6 | 4.0 | 6.0 | 3.93 |
| **TOTAL** | **20.0** | **28.3** | **38.6** | **28.62** |

As usual, it turns out that the developers' single-point estimates in Table 10-4 were actually their Best Case estimates.

## 10.4 Creating Meaningful Overall Best Case and Worst Case Estimates

If you can't use the sum of the best cases and worst cases to produce overall Best Case and Worst Case estimates, what do you do? A common approximation in statistics is to assume that 1/6 of the range between a minimum and a maximum approximately equals one standard deviation. This is based on the assumption that the minimum is only 0.135% likely and the assumption that the maximum includes 99.86% of all possible values.

## Computing Aggregate Best and Worst Cases for Small Numbers of Tasks (Simple Standard Deviation Formula)

For a small number of tasks (about 10 or fewer), you can base the best and worst cases on a simple standard deviation calculation. First, you add the best cases together and add the worst cases together. Then you compute the standard deviation using this formula:

**Equation #4**

StandardDeviation = (SumOfWorstCaseEstimates − SumOfBestCaseEstimates) / 6

If you take 1/6 of the range between 20.0 and 38.6 in Table 10-5, that will be 1 standard deviation of the distribution of project outcomes for that project. One-sixth of that difference is 3.1. You can then use a table of standard deviations to compute a percentage likelihood. In a business context, this is often referred to as *percentage confident*. Table 10-6 provides the standard deviation numbers.

**Table 10-6 Percentage Confident Based on Use of Standard Deviation**

| Percentage Confident | Calculation |
|---|---|
| 2% | Expected case – (2 x StandardDeviation) |
| 10% | Expected case – (1.28 x StandardDeviation) |
| 16% | Expected case – (1 x StandardDeviation) |
| 20% | Expected case – (0.84 x StandardDeviation) |
| 25% | Expected case – (0.67 x StandardDeviation) |
| 30% | Expected case – (0.52 x StandardDeviation) |
| 40% | Expected case – (0.25 x StandardDeviation) |
| 50% | Expected case |
| 60% | Expected case + (0.25 x StandardDeviation) |
| 70% | Expected case + (0.52 x StandardDeviation) |
| 75% | Expected case + (0.67 x StandardDeviation) |
| 80% | Expected case + (0.84 x StandardDeviation) |
| 84% | Expected case + (1 x StandardDeviation) |
| 90% | Expected case + (1.28 x StandardDeviation) |
| 98% | Expected case + (2 x StandardDeviation) |

Using this approach, a statistically valid 75%-likely estimate would be the *Expected case* (29 weeks) plus 0.67 x *StandardDeviation*, which is 29 + (0.67 x 3.1), which equals 31 weeks.

Why do I say the answer is 31 weeks instead of 31.1? Because the garbage in, garbage out principle applies. The underlying task estimates are not accurate to more than 2 significant digits, much less 3, so be humble about the results. In this example, presenting an estimate of 31 weeks probably overstates the accuracy of the result, and 30 might be a more meaningful number.

**Tip #49** Use the simple standard deviation formula to compute meaningful aggregate Best Case and Worst Case estimates for estimates containing 10 tasks or fewer.

## Computing Aggregate Best and Worst Cases for Large Numbers of Tasks (Complex Standard Deviation Formula)

If you have more than about 10 tasks, the formula for standard deviation in the previous section isn't valid, and you have to use a more complicated approach. A science-of-estimation approach begins by applying the standard deviation formula to each of the individual estimates (Stutzke 2005):

**Equation #5**

$$\text{IndividualStandardDeviation} = (\text{IndividualWorstCaseEstimate} - \text{IndividualBestCaseEstimate}) / 6$$

You use this formula to compute the Standard Deviation column in Table 10-7. You then go through some fairly complicated math to compute the standard deviation of the aggregate estimate.

1. Compute the standard deviation of each task or feature using the preceding formula.
2. Compute the square of each task's standard deviation, which is known as the *variance*. This is shown in the right-most column of Table 10-7.
3. Total the variances.
4. Take the square root of the total.

In the table, the sum of the variances is 1.22, and the square root of that is 1.1, so that's the standard deviation of the aggregate estimate.

Table 10-7 **Example of Complex Standard Deviation Calculations**

| | Weeks to Complete | | | |
|---|---|---|---|---|
| **Feature** | **Best Case** | **Worst Case** | **Standard Deviation** | **Variance (Standard Deviation Squared)** |
| Feature 1 | 1.6 | 3.0 | 0.233 | 0.054 |
| Feature 2 | 1.8 | 4.0 | 0.367 | 0.134 |
| Feature 3 | 2.0 | 4.2 | 0.367 | 0.134 |

Table 10-7 Example of Complex Standard Deviation Calculations

| | Weeks to Complete | | | |
|---|---|---|---|---|
| **Feature** | **Best Case** | **Worst Case** | **Standard Deviation** | **Variance (Standard Deviation Squared)** |
| Feature 4 | 0.8 | 1.6 | 0.133 | 0.018 |
| Feature 5 | 3.8 | 5.2 | 0.233 | 0.054 |
| Feature 6 | 3.8 | 6.0 | 0.367 | 0.134 |
| Feature 7 | 2.2 | 3.4 | 0.200 | 0.040 |
| Feature 8 | 0.8 | 2.2 | 0.233 | 0.054 |
| Feature 9 | 1.6 | 3.0 | 0.233 | 0.054 |
| Feature 10 | 1.6 | 6.0 | 0.733 | 0.538 |
| **TOTAL** | **20.0** | **38.6** | - | **1.22** |
| **Standard Deviation** | - | - | - | **1.1** |

**Tip #50** Use the complex standard deviation formula to compute meaningful aggregate Best Case and Worst Case estimates when you have about 10 tasks or more.

If you recall that the standard deviation produced by the preceding approach was 3.1, you'll realize that this approach produces an answer of 1.10 from the same data, which is quite a discrepancy! How can that be?

This turns out to be a case of the difference between precision and accuracy. The problem with using the formula *(WorstCaseEstimate − BestCaseEstimate) / 6*, is that, statistically speaking, you're assuming that the person who created the Best Case and Worst Case estimates included a 6 standard deviation range from best case to worst case. For that to be true, the estimation range would have to account for 99.7% of all possible outcomes. In other words, out of 1000 estimates, only 3 actual outcomes could fall outside their estimated ranges!

Of course, this is a ridiculous assumption. In the example, 2 outcomes out of 10 fell outside the estimation range. As Chapter 1 illustrated, most people's sense of 90% confident is really closer to 30% confident. With practice, people might be able to estimate an all-inclusive range 70% of the time, but estimators don't have a ghost of a chance of estimating a 99.7% confidence interval.

A realistic approach to computing standard deviation from best and worst cases is to divide each individual range by a number that's closer to 2 than 6. Statistically, dividing by 2 implies that the estimator's ranges will include the actual outcome 68% of the time, which is a goal that can be achieved with practice.

Table 10-8 lists the number you should divide by based on the percentage of your actual outcomes that are falling within your estimated ranges.

Table 10-8 Divisor to Use for the Complex Standard Deviation Calculation

| If this percentage of your actual outcomes fall within your estimation range... | ...use this number as the divisor in the standard deviation calculation for individual estimates |
|---|---|
| 10% | 0.25 |
| 20% | 0.51 |
| 30% | 0.77 |
| 40% | 1.0 |
| 50% | 1.4 |
| 60% | 1.7 |
| 70% | 2.1 |
| 80% | 2.6 |
| 90% | 3.3 |
| 99.7% | 6.0 |

You would then plug the appropriate number from this table into the complex standard deviation formula:

**Equation #6**

IndividualStandardDeviation = (IndividualWorstCaseEstimate − Individual BestCaseEstimate) / DivisorFromTable10-8

**Tip #51** Don't divide the range from best case to worst case by 6 to obtain standard deviations for individual task estimates. Choose a divisor based on the accuracy of your estimation ranges.

## Creating the Aggregate Best and Worst Case Estimates

In the case study, the team's actual results fell within its best case–worst case ranges 8 out of 10 times. Table 10-8 indicates that teams hitting the actual result 80% of the time should use a divisor of 2.6. Table 10-9 shows the results of recomputing the standard deviations, variances, and aggregate standard deviation based on dividing the ranges by 2.6 instead of 6.

Table 10-9 Example of Computing Standard Deviation Using a Divisor Other Than 6

| | Weeks to Complete | | | |
|---|---|---|---|---|
| **Feature** | **Best Case** | **Worst Case** | **Standard Deviation** | **Variance (Standard Deviation Squared)** |
| Feature 1 | 1.6 | 3.0 | 0.538 | 0.290 |
| Feature 2 | 1.8 | 4.0 | 0.846 | 0.716 |
| Feature 3 | 2.0 | 4.2 | 0.846 | 0.716 |

Table 10-9 **Example of Computing Standard Deviation Using a Divisor Other Than 6**

| | Weeks to Complete | | | |
|---|---|---|---|---|
| **Feature** | **Best Case** | **Worst Case** | **Standard Deviation** | **Variance (Standard Deviation Squared)** |
| Feature 4 | 0.8 | 1.6 | 0.308 | 0.095 |
| Feature 5 | 3.8 | 5.2 | 0.538 | 0.290 |
| Feature 6 | 3.8 | 6.0 | 0.846 | 0.716 |
| Feature 7 | 2.2 | 3.4 | 0.462 | 0.213 |
| Feature 8 | 0.8 | 2.2 | 0.538 | 0.290 |
| Feature 9 | 1.6 | 3.0 | 0.538 | 0.290 |
| Feature 10 | 1.6 | 6.0 | 1.692 | 2.864 |
| **TOTAL** | **20.0** | **38.6** | **-** | **6.48** |
| **Standard Deviation** | **-** | **-** | **-** | **2.55** |

This approach produces a standard deviation for the aggregate estimate of 2.55 weeks. To compute percentage-confident estimates, you would then use the Expected Case estimate of 28.6 weeks from Table 10-5 and the multipliers from Table 10-6. This would produce a set of percentage-confident estimates such as the ones shown in Table 10-10.

Table 10-10 **Example of Percentage-Confident Estimates Computed From Standard Deviation**

| **Percentage Confident** | **Effort Estimate** |
|---|---|
| 2% | 23.5 |
| 10% | 25.4 |
| 16% | 26.1 |
| 20% | 26.5 |
| 25% | 26.9 |
| 30% | 27.3 |
| 40% | 28.0 |
| 50% | 28.6 |
| 60% | 29.3 |
| 70% | 30.0 |
| 75% | 30.3 |
| 80% | 30.8 |
| 84% | 31.2 |
| 90% | 31.8 |
| 98% | 33.7 |

Depending on the audience for these estimates, you might heavily edit the entries in this table before you present them. In some circumstances, however, it might be quite useful to point out that although totaling the Best Case estimates yields a total of 20 staff weeks, it's only 2% likely that you'll beat 23.5 weeks and only 25% likely that you'll beat 26.9 weeks.

As always, you should consider the precision of the estimates before you present them—I would normally present 24 weeks instead of 23.5 and 27 weeks instead of 26.9.

### Cautions About Percentage Confident Estimates

One general pitfall with the approach I just described is that the Expected Case estimates need to be accurate—that is, they need to be truly 50% likely. You should underrun those estimates just as often as you overrun them. If you find that you're overrunning them more often than you're underrunning them, they aren't really 50% likely and you shouldn't use them as your expected cases. If the expected cases aren't accurate, then the sum of the expected cases won't be accurate either.

Chapter 9 provides suggestions for making the individual estimates more accurate.

**Tip #52** Focus on making your Expected Case estimates accurate. If the individual estimates are accurate, aggregation will not create problems. If the individual estimates are not accurate, aggregation will be problematic until you find a way to make them accurate.

## Additional Resources

Humphrey, Watts S. *A Discipline for Software Engineering*. Reading, MA: Addison-Wesley, 1995. Appendix A of Humphrey's book contains a short, readable summary of statistical techniques that are useful for software estimation.

Stutzke, Richard D. *Estimating Software-Intensive Systems*, Upper Saddle River, NJ: Addison-Wesley, 2005. Chapter 5 of Stutzke's book goes into more detail on some of the statistics presented in this chapter. Chapter 20 describes how to create a WBS.

Gonick, Larry and Woollcott Smith. *The Cartoon Guide to Statistics*. New York, NY: Harper Collins, 1993. Despite the silly title, this is a respectable (and fun) introduction to statistical techniques. Many readers will find the extensive illustrations help them learn the statistical concepts. Some readers might find that the focus on pictures rather than text makes the concepts harder to understand.

Larsen, Richard J. and Morris L. Marx. *An Introduction to Mathematical Statistics and Its Applications, Third Edition*. Upper Saddle River, NJ: Prentice Hall, 2001. This book is a fairly readable, traditional introduction to mathematical statistics; at least it's readable when you consider the subject matter. It's an unavoidable fact that if you want to use statistical techniques, sooner or later you'll have to do some math!

Chapter 11

# Estimation by Analogy

### Applicability of Techniques in This Chapter

| | Estimation by Analogy |
|---|---|
| **What's estimated** | Size, Effort, Schedule, Features |
| **Size of project** | S M L |
| **Development stage** | Early–Late |
| **Iterative or sequential** | Both |
| **Accuracy possible** | Medium |

Gigacorp (a fictional corporation) was about to begin work on Triad 1.0, a companion product to its successful AccSellerator 1.0 sales-presentation software. Mike had been appointed project manager of Triad 1.0, and he needed a ballpark estimate for an upcoming sales planning meeting. He called his staff meeting to order.

"As you know, we're embarking on development of Triad 1.0," he said. "The technical work is very similar to AccSellerator 1.0. I see this project as being a little bigger overall than AccSellerator 1.0, but not much bigger."

"The database is going to be quite a bit bigger," Jennifer volunteered. "But the user interface should be about the same size."

"It will have a lot more graphs and reports than AccSellerator 1.0 had, too, but the foundation classes should be very similar; I think we'll end up with the same number of classes." Joe said.

"That all sounds right to me," Mike said. "I think this gives me enough to do a back-of-the-envelope calculation of project effort. My notes indicate that the total effort for the last system was 30 staff months. What do you think is a reasonable ballpark estimate for the effort of the new system?"

What do *you* think is a reasonable ballpark estimate for the effort of the new system?

## 11.1 Basic Approach to Estimating by Analogy

The basic approach that Mike is using in this example is estimation by analogy, which is the simple idea that you can create accurate estimates for a new project by comparing the new project to a similar past project.

I've had several hundred estimators create estimates for the Triad project. Using the approach implied in the example, their estimates have ranged from 30 to 144 staff months, with an average of 53 staff months. The standard deviation of their estimates is 24, or 46% of the average answer. That is not very good! A little bit of structure on the process helps a lot.

Here is a basic estimation by analogy process that will produce better results:

1. Get detailed size, effort, and cost results for a similar previous project. If possible, get the information decomposed by feature area, by work breakdown structure (WBS) category, or by some other decomposition scheme.
2. Compare the size of the new project piece-by-piece to the old project.
3. Build up the estimate for the new project's size as a percentage of the old project's size.
4. Create an effort estimate based on the size of the new project compared to the size of the previous project.
5. Check for consistent assumptions across the old and new projects.

**Tip #53** Estimate new projects by comparing them to similar past projects, preferably decomposing the estimate into at least five pieces.

Let's continue using the Triad case study to examine these steps.

## Step 1: Get Detailed Size, Effort, and Cost Results for a Similar Previous Project

After the first meeting, Mike asked the Triad staff to gather more specific information about the sizes of the old system and the relative amount of functionality in the old and new systems. When their work was completed, Mike asked how they had done. "Did you get the data on the project I outlined last week?" he asked.

"Sure, Mike," Jennifer replied. "AccSellerator 1.0 had 5 subsystems. They stacked up like this:

| | |
|---|---|
| Database | 5,000 lines of code (LOC) |
| User interface | 14,000 LOC |
| Graphs and reports | 9,000 LOC |
| Foundation classes | 4,500 LOC |
| Business rules | 11,000 LOC |
| **TOTAL** | **43,500 LOC** |

"We also got some general information about the number of elements in each subsystem. Here's what we found:

| | |
|---|---|
| Database | 10 tables |
| User interface | 14 Web pages |
| Graphs and reports | 10 graphs + 8 reports |
| Foundation classes | 15 classes |
| Business rules | ??? |

"We've done a fair amount of work to scope out the new system. It looks like this:

| | |
|---|---|
| Database | 14 tables |
| User interface | 19 Web pages |
| Graphs and reports | 14 graphs + 16 reports |
| Foundation classes | 15 classes |
| Business rules | ??? |

"The comparison to most of the old system is pretty straightforward, but the business rules part is a little tough," Jennifer said. "We think it's going to be more complicated than the old system, but we're not sure how to put a number on it. We've talked it over, and our feeling is that it's at least 50% more complicated than the old system."

"That's great work," Mike said. "This gives me what I need to compute an estimate for my sales meeting. I'll crunch some numbers this afternoon and run them by you before the meeting."

## Step 2: Compare the Size of the New Project to a Similar Past Project

The Triad details give us what we need to create a meaningful estimate by analogy. The Triad team has already performed Step 1, "Get detailed size, effort, cost results for a similar previous project." We can perform Step 2, "Compare the size of the new project piece-by-piece to the old project." Table 11-1 shows that detailed comparison.

**Table 11-1 Detailed Size Comparison Between AccSellerator 1.0 and Triad 1.0**

| Subsystem | Actual Size of AccSellerator 1.0 | Estimated Size of Triad 1.0 | Multiplication Factor |
|---|---|---|---|
| Database | 10 tables | 14 tables | 1.4 |
| User interface | 14 Web pages | 19 Web pages | 1.4 |
| Graphs and reports | 10 graphs + 8 reports | 14 graphs + 16 reports | 1.7 |
| Foundation classes | 15 classes | 15 classes | 1.0 |
| Business rules | ??? | ??? | 1.5 |

Writing down the numbers in columns 2 and 3 is the easy part. The tricky part is what to do in the Multiplication Factor entry in column 4. The main principle here is the Count, Compute, Judge principle. If we can find something to count, we're better off than if we insert subjective judgment.

The factors of 1.4 for database, 1.4 for user interface, and 1.0 for foundation classes seem straightforward.

The factor of 1.7 for graphs and reports is a little tricky. Should graphs be weighted the same as reports? Maybe. Graphs might require more work than reports, or vice versa. If we had access to the code base for AccSellerator 1.0, we could check whether graphs and reports should be weighted equally or whether one should be weighted more heavily than the other. In this case, we'll just assume they're weighted equally. We should document this assumption so that we can retrace our steps later, if we need to.

The business rules entry is also problematic. The team in the case study didn't find anything they could count, so our estimate is on shakier ground in that area than in the other areas. For sake of the example, we'll just accept their claim that the business rules for Triad will be about 50% more complicated than the business rules were in AccSellerator.

## Step 3: Build Up the Estimate for the New Project's Size as a Percentage of the Old Project's Size

In Step 3, we convert the size measures from the different areas to a common unit of measure, in this case, lines of code. This will allow us to perform a whole-system size comparison between AccSellerator and Triad. Table 11-2 shows how this works.

Table 11-2 **Computing Size of Triad 1.0 Based on Comparison to AccSellerator 1.0**

| Subsystem | Code Size of AccSellerator 1.0 | Multiplication Factor | Estimated Code Size of Triad 1.0 |
|---|---|---|---|
| Database | 5,000 | 1.4 | 7,000 |
| User interface | 14,000 | 1.4 | 19,600 |
| Graphs and reports | 9,000 | 1.7 | 15,300 |
| Foundation classes | 4,500 | 1.0 | 4,500 |
| Business rules | 11,000 | 1.5 | 16,500 |
| **TOTAL** | **43,500** | **-** | **62,900** |

The code sizes for AccSellerator are carried down from the information that was generated in Step 1. The multiplication factors are carried down from the work we did in Step 2. The estimated code size for Triad is simply AccSellerator's code size multiplied by the multiplication factors. The total size in lines of code becomes the basis for our effort estimate, which will in turn become the basis for schedule and cost estimates.

## Step 4: Create an Effort Estimate Based on the Size of the New Project Compared to the Previous Project

We now have enough background to compute an effort estimate, which is shown in Table 11-3.

**Table 11-3 Final Computation of Effort for Triad 1.0**

| Term | Value |
|---|---|
| Size of Triad 1.0 | 62,900 LOC |
| Size of AccSellerator 1.0 | ÷ 43,500 LOC |
| Size ratio | = 1.45 |
| Effort for AccSellerator 1.0 | × 30 staff months |
| Estimated effort for Triad 1.0 | = 44 staff months |

Dividing the size of Triad by the size of AccSellerator gives us a ratio of the sizes of the two systems. We can multiply that by AccSellerator's actual effort, and that gives us the estimate for Triad of 44 staff months.

The estimate you compute and the estimate you present are two different matters. In this computation, you ended up with a single-point estimate. When you present the estimate, you might well decide to present it as a range, as discussed in Chapter 22, "Estimate Presentation Styles."

I've had the same several hundred estimators who created the original rolled-up estimates for Triad follow this approach, and their results are more accurate and consistent. The standard deviation of their results is only 7% rather than the 46%, even with the uncertainty surrounding graphs, reports, and business rules.

## Step 5: Check for Consistent Assumptions Across the Old and New Projects

You should be checking your assumptions at each step. Some assumptions aren't completely checkable until you've computed the estimate. Look for the following major sources of inconsistency:

- Significantly different sizes between the old and new projects—that is, more than the factor of 3 difference described in Section 5.1, "Project Size." In this case, the sizes are different, but only by a factor of 1.45, which is not enough of a difference to cause any worry about diseconomies of scale.

- Different technologies (for example, one project in C# and the other in Java).
- Significantly different team members (for small teams) or team capabilities (for large teams). Small differences are OK and often unavoidable.
- Significantly different kinds of software. For example, an old system that was an internal intranet system and a new system that's a life-critical embedded system would not be comparable.

## 11.2 Comments on Uncertainty in the Triad Estimate

The information available to create the business rules estimate was pretty fuzzy. Should we fudge the business rules number upward to be conservative in our estimate? For estimation purposes, the answer is no. The focus of the estimate should be on *accuracy*, not conservatism. Once you move the estimate's focus away from accuracy, bias can creep in from many different sources and the value of the estimate will be reduced. The best estimation response to uncertainty is not to bias the estimate but to be sure that the estimate accurately expresses any underlying uncertainty. If you were completely confident in the business rules number, you might consider the effort estimate to be accurate to ±10%. Considering the uncertainty in the business rules, perhaps you would fudge the uncertainty number to something like +25%, −10%.

A better way to address the uncertainty arising from the business rules part of the estimate could be to carry a range for the business rules factor through your computations rather than using a single number. You might estimate the factor with a 50% variation (in other words, a range of 0.75 to 2.25) instead of using a single point factor of 1.5. That would produce an effort range of 38 to 49 staff months rather than the single-point estimate of 44 staff months.

One contrast between the estimate created using this approach and the estimate created using a rolled-up (undecomposed) approach is that, in the rolled-up approach, uncertainty in one area can spread to other areas. If there is a 50% uncertainty in the business rules, the estimator might apply that uncertainty to the whole estimate, rather than just to the quarter of the estimate related to business rules. If you applied that same 50% variation to the whole estimate, the estimate would range from 22 to 66 staff months rather than from 38 to 49 staff months. Identifying what specifically is uncertain and how much effect that should have on the estimate helps narrow the overall estimation range.

**Tip #54** Do not address estimation uncertainty by biasing the estimate. Address uncertainty by expressing the estimate in uncertain terms.