Chapter 5

Estimate Influences

How much is 68 + 73?

ENGINEER: "It's 141." Short and sweet.

MATHEMATICIAN: "68 + 73 = 73 + 68 by the commutative law of addition." True, but not very helpful.

ACCOUNTANT: "Normally it's 141, but what are you going to use it for?"

–Barry W. Boehm and Richard E. Fairley

Influences on a software project can be sliced and diced in several ways. Understanding these influences helps improve estimation accuracy and helps improve understanding of software project dynamics overall.

Project size is easily the most significant determinant of effort, cost, and schedule. The kind of software you're developing comes in second, and personnel factors are a close third. The programming language and environment you use are not first-tier influences on the project outcome, but they are a first-tier influence on the estimate. This chapter presents these first-tier influences in decreasing order of significance and concludes with a discussion of second-tier influences.

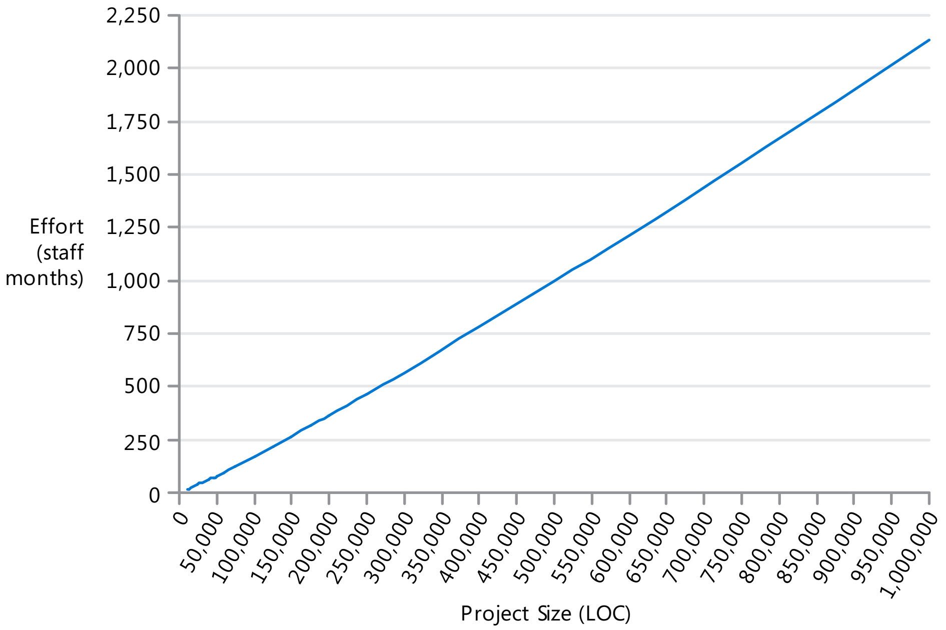
5.1 Project Size

The largest driver in a software estimate is the size of the software being built, because there is more variation in the size than in any other factor. Figure 5-1 shows the way that effort grows in an average business-systems project as project size increases from 25,000 lines of code to 1,000,000 lines of code. The figure expresses size in lines of code (LOC), but the dynamic would be the same whether you measured size in function points, number of requirements, number of Web pages, or any other measure that expressed the same range of sizes.

As the diagram shows, a system consisting of 1,000,000 LOC requires dramatically more effort than a system consisting of only 100,000 LOC.

These comments about software size being the largest cost driver might seem obvious, yet organizations routinely violate this fundamental fact in two ways:

- Costs, effort, and schedule are estimated without knowing how big the software will be.
- Costs, effort, and schedule are not adjusted when the size of the software is consciously increased (that is, in response to change requests).



Source: Computed using data from the Cocomo II estimation model, assuming nominal diseconomy of scale (Boehm, et al 2000).

Figure 5-1 Growth in effort for a typical business-systems project. The specific numbers are meaningful only for the average business-systems project. The general dynamic applies to software projects of all kinds.

Tip #24 Invest an appropriate amount of effort assessing the size of the software that will be built. The size of the software is the single most significant contributor to project effort and schedule.

Why Is This Book Discussing Size in Lines of Code?

People new to estimation sometimes have questions about whether lines of code are really a meaningful way to measure software size. One issue is that many modern programming environments are not as lines-of-code-oriented as older environments were. Another issue is that a lot of software development work—such as requirements, design, and testing—doesn't produce lines of code. If you're interested in seeing how these issues affect the usefulness of measuring size in lines of code, see Section 18.1, "Challenges with Estimating Size."

Diseconomies of Scale

People naturally assume that a system that is 10 times as large as another system will require something like 10 times as much effort to build. But the effort for a

1,000,000-LOC system is more than 10 times as large as the effort for a 100,000-LOC system, as is the effort for a 100,000-LOC system compared to the effort for a 10,000-LOC system.

The basic issue is that, in software, larger projects require coordination among larger groups of people, which requires more communication (Brooks 1995). As project size increases, the number of communication paths among different people increases as a *squared* function of the number of people on the project.[1] Figure 5-2 illustrates this dynamic.

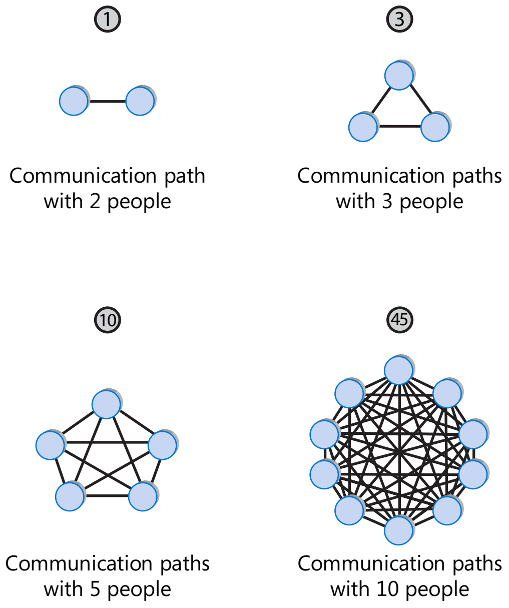

1
Communication path with 2 people
3
Communication paths with 3 people
6
Communication paths with 4 people
10
Communication paths with 5 people
45
Communication paths with 10 people

Figure 5-2 The number of communication paths on a project increases proportionally to the square of the number of people on the team.

The consequence of this exponential increase in communication paths (along with some other factors) is that projects also have an exponential increase in effort as a project size increases. This is known as a *diseconomy of scale*.

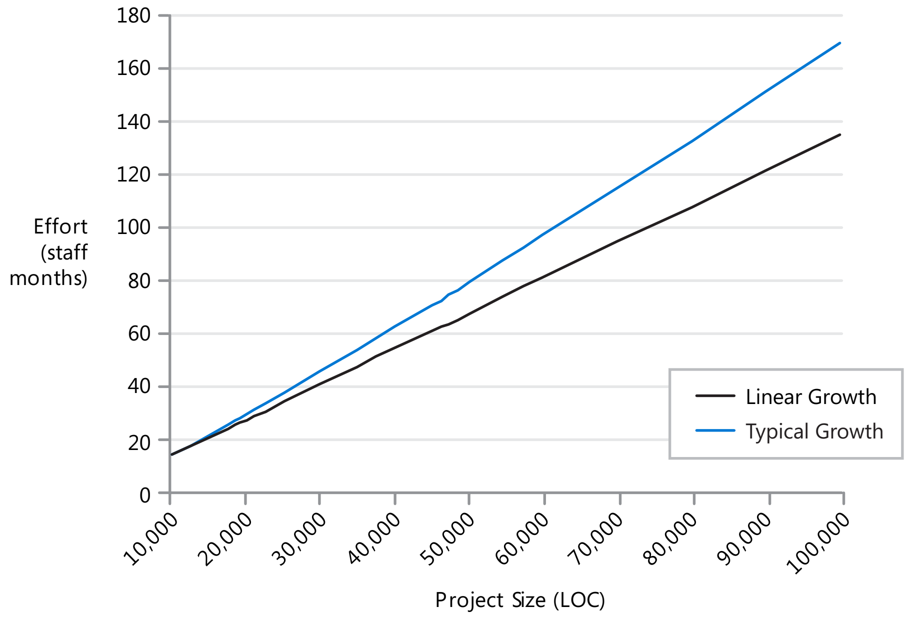
Outside software, we usually discuss *economies* of scale rather than *diseconomies* of scale. An economy of scale is something like, "If we build a larger manufacturing plant, we'll be able to reduce the cost per unit we produce." An economy of scale implies that the bigger you get, the smaller the unit cost becomes.

A diseconomy of scale is the opposite. In software, the larger the system becomes, the greater the cost of each unit. If software exhibited economies of scale, a 100,000-LOC

[1] The actual number of paths is $n \times (n - 1) / 2$, which is an n^2 function.

system would be less than 10 times as costly as a 10,000-LOC system. But the opposite is almost always the case.

Figure 5-3 illustrates a typical diseconomy of scale in software compared with the increase of effort that would be associated with linear growth.

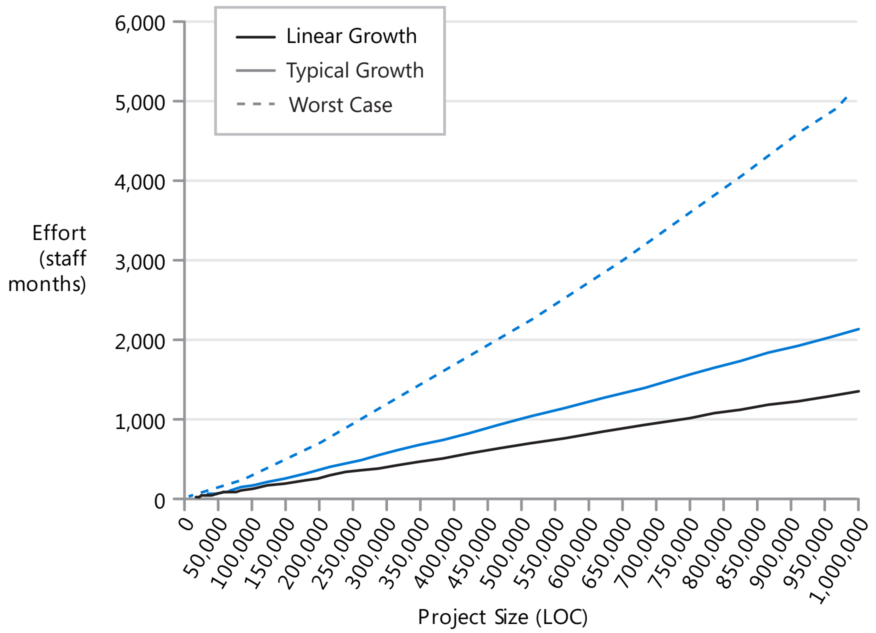


Source: Computed using data from the Cocomo II estimation model, assuming nominal diseconomy of scale (Boehm, et al 2000).

Figure 5-3 Diseconomy of scale for a typical business-systems project ranging from 10,000 to 100,000 lines of code.

As you can see from the graph, in this example, the 10,000-LOC system would require 13.5 staff months. If effort increased linearly, a 100,000-LOC system would require 135 staff months, but it actually requires 170 staff months.

As Figure 5-3 is drawn, the effect of the diseconomy of scale doesn't look very dramatic. Indeed, within the 10,000 LOC to 100,000 LOC range, the effect is usually not all that dramatic. But two factors make the effect more dramatic. One factor is greater difference in project size, and the other factor is project conditions that degrade productivity more quickly than average as project size increases. Figure 5-4 shows the range of outcomes for projects ranging from 10,000 LOC to 1,000,000 LOC. In addition to the nominal diseconomy, the graph also shows the worst-case diseconomy.



Source: Computed using data from the Cocomo II estimation model, assuming nominal and worst-case diseconomies of scale (Boehm, et al 2000).

Figure 5-4 Diseconomy of scale for projects with greater size differences and the worst-case diseconomy of scale.

In this graph, you can see that the worst-case effort growth increases much faster than the nominal effort growth, and that the effect becomes much more pronounced at larger project sizes. Along the nominal effort growth curve, effort at 100,000 lines of code is 13 times what it is at 10,000 lines of code, rather than 10 times. At 1,000,000 LOC, effort is 160 times the 10,000-LOC effort, rather than 100 times.

The worst-case growth is much worse. Effort on the worst-case curve at 100,000 LOC is 17 times what it is at 10,000 LOC, and at 1,000,000 LOC it isn't 100 times as large–it's 300 times as large!

Table 5-1 illustrates the general relationship between project size and productivity.

Table 5-1 Relationship Between Project Size and Productivity

Project Size (in Lines of Code)	Lines of Code per Staff Year (Cocomo II Nominal in Parentheses)
10K	2,000–25,000 (3,200)
100K	1,000–20,000 (2,600)
1M	700–10,000 (2,000)
10M	300–5,000 (1,600)

Source: Derived from data in *Measures for Excellence* (Putnam and Meyers 1992), *Industrial Strength Software* (Putnam and Meyers 1997), *Software Cost Estimation with Cocomo II* (Boehm et al. 2000), and "Software Development Worldwide: The State of the Practice" (Cusumano et al. 2003).

The numbers in this table are valid only for purposes of comparison between size ranges. But the general trend the numbers show is significant. Productivity on small projects can be 2 to 3 times as high as productivity on large projects, and productivity can vary by a factor of 5 to 10 from the smallest projects to the largest.

Tip #25 Don't assume that effort scales up linearly as project size does. Effort scales up exponentially.

For software estimation, the implications of diseconomies of scale are a case of good news, bad news. The bad news is that if you have large variations in the sizes of projects you estimate, you can't just estimate a new project by applying a simple effort ratio based on the effort from previous projects. If your effort for a previous 100,000-LOC project was 170 staff months, you might figure that your productivity rate is 100,000/170, which equals 588 LOC per staff month. That might be a reasonable assumption for another project of about the same size as the old project, but if the new project is 10 times bigger, the estimate you create that way could be off by 30% to 200%.

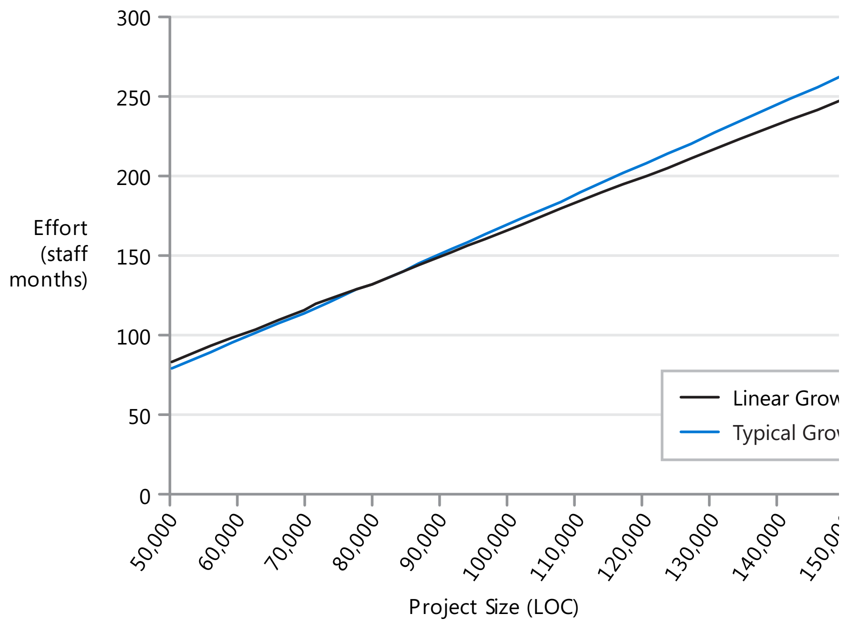
There's more bad news: There isn't a simple technique in the art of estimation that will account for a significant difference in the size of two projects. If you're estimating a project of a significantly different size than your organization has done before, you'll need to use estimation software that applies the science of estimation to compute the estimate for the new project based on the results of past projects. My company provides a free software tool called Construx® Estimate™ that will do this kind of estimate. You can download a copy at *www.construx.com/estimate*.

Tip #26 Use software estimation tools to compute the impact of diseconomies of scale.

When You Can Safely Ignore Diseconomies of Scale

After all that bad news, there is actually some good news. The majority of projects in an organization are often similar in size. If the new project you're estimating will be similar in size to your past projects, it is usually safe to use a simple effort ratio, such as lines of code per staff month, to estimate a new project. Figure 5-5 illustrates the relatively minor difference in linear versus exponential estimates that occurs within a specific size range.

If you use a ratio-based estimation approach within a restricted range of sizes, your estimates will not be subject to much error. If you used an average ratio from projects in the middle of the size range, the estimation error introduced by economies of scale would be no more than about 10%. If you work in an environment that experiences higher-than-average diseconomies of scale, the differences could be higher.



Source: Computed using data from the Cocomo II estimation model, assuming nominal disecor of scale (Boehm, et al 2000).

Figure 5-5 Differences between ratio-based estimates and estimates based on diseconomy of scale will be minimal for projects within a similar size range.

Tip #27 If you've completed previous projects that are about the same size as the project you're estimating—defined as being within a factor of 3 from largest to smallest—you can safely use a ratio-based estimating approach, such as lines of code per staff month, to estimate your new project.

Importance of Diseconomy of Scale in Software Estimation

Much of the software-estimating world's focus has been on determining the exact significance of diseconomies of scale. Although that is a significant factor, remember that the raw size is the largest contributor to the estimate. The effect of diseconomy of scale on the estimate is a second-order consideration, so put the majority of your effort into developing a good size estimate. We'll discuss how to create software size estimates more specifically in Chapter 18, "Special Issues in Estimating Size."

5.2 Kind of Software Being Developed

After project size, the kind of software you're developing is the next biggest influence on the estimate. If you're working on life-critical software, you can expect your project to require far more effort than a similarly sized business-systems project. Table 5-2 shows examples of lines of code per staff month for projects of different kinds.

Table 5-2 **Productivity Rates for Common Project Types**

	LOC/Staff Month Low-High (Nominal)		
Kind of Software	**10,000-LOC Project**	**100,000-LOC Project**	**250,000-LOC Project**
Avionics	100–1,000 **(200)**	20–300 **(50)**	20–200 **(40)**
Business Systems	800–18,000 **(3,000)**	200–7,000 **(600)**	100–5,000 **(500)**
Command and Control	200–3,000 **(500)**	50–600 **(100)**	40–500 **(80)**
Embedded Systems	100–2,000 **(300)**	30–500 **(70)**	20–400 **(60)**
Internet Systems (public)	600–10,000 **(1,500)**	100–2,000 **(300)**	100–1,500 **(200)**
Intranet Systems (internal)	1,500–18,000 **(4,000)**	300–7,000 **(800)**	200–5,000 **(600)**
Microcode	100–800 **(200)**	20–200 **(40)**	20–100 **(30)**
Process Control	500–5,000 **(1,000)**	100–1,000 **(300)**	80–900 **(200)**
Real-Time	100–1,500 **(200)**	20–300 **(50)**	20–300 **(40)**
Scientific Systems/ Engineering Research	500–7,500 **(1,000)**	100–1,500 **(300)**	80–1,000 **(200)**
Shrink wrap/Packaged Software	400–5,000 **(1,000)**	100–1,000 **(200)**	70–800 **(200)**
Systems Software/Drivers	200–5,000 **(600)**	50–1,000 **(100)**	40–800 **(90)**
Telecommunications	200–3,000 **(600)**	50–600 **(100)**	40–500 **(90)**

Source: Adapted and extended from *Measures for Excellence* (Putnam and Meyers 1992), *Industrial Strength Software* (Putnam and Meyers 1997), and *Five Core Metrics* (Putnam and Meyers 2003).

As you can see from the table, a team developing an intranet system for internal use might generate code 10 to 20 times faster than a team working on an avionics project, real-time project, or embedded systems project. The table also again illustrates the diseconomy of scale: projects of 100,000 LOC generate code far less efficiently than 10,000-LOC projects. Projects of 250,000 LOC generate code even less efficiently.

You can account for the industry in which you're working in one of three ways:

- Use the results from Table 5-2 as a starting point. If you do that, notice that the ranges in the table are large—typically a factor of 10 difference between the high and the low ends of the ranges.

- Use an estimating model such as Cocomo II, and adjust the estimating parameters to match the kind of software you develop. If you do that, remember the cautions from Chapter 4, "Where Does Estimation Error Come From?" about using too many control knobs on your estimates.
- Use historical data from your own organization, which will automatically incorporate the development factors specific to the industry you work in. This is by far the best approach, and we'll discuss the use of historical data in more detail in Chapter 8, "Calibration and Historical Data."

Tip #28 Factor the kind of software you develop into your estimate. The kind of software you're developing is the second-most significant contributor to project effort and schedule.

5.3 Personnel Factors

Personnel factors also exert significant influence on project outcomes. According to Cocomo II, on a 100,000-LOC project the combined effect of personnel factors can swing a project estimate by as much as a factor of 22! In other words, if your project ranked worst in each category shown in Figure 5-6 (shown by the gray bars), it would require 22 times as much total effort as a project that ranked best in each category (shown by the blue bars).

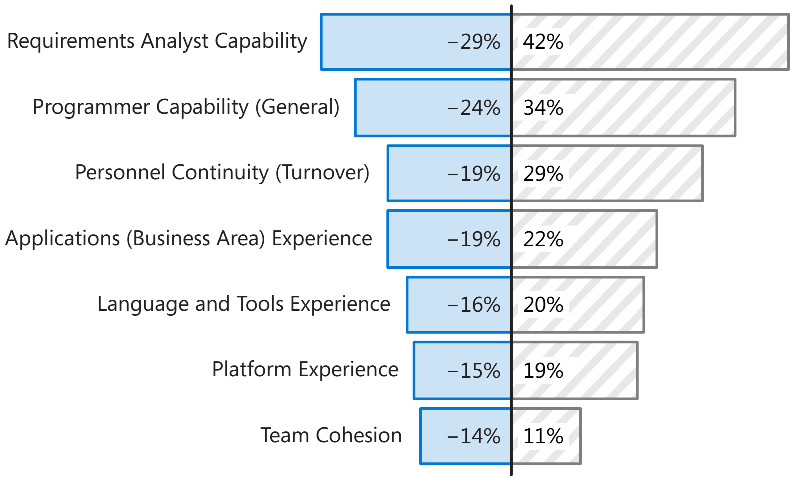


Figure 5-6 Effect of personnel factors on project effort. Depending on the strength or weakness in each factor, the project results can vary by the amount indicated—that is, a project with the worst requirements analysts would require 42% more effort than nominal, whereas a project with the best analysts would require 29% less effort than nominal.

The magnitude of these factors from the Cocomo II model is confirmed by numerous studies since the 1960s that show 10:1 to 20:1 differences in individual and team performance (Sackman, Erikson, and Grant 1968; Weinberg and Schulman 1974; Curtis 1981; Mills 1983; Boehm, Gray, and Seewaldt 1984; DeMarco and Lister 1985; Curtis et al. 1986; Card 1987; Boehm 1987b; Boehm and Papaccio 1988; Valett and McGarry 1989; Boehm et al. 2000).

One implication of these variations among individuals is that you can't accurately estimate a project if you don't have some idea of who will be doing the work—because individual performance varies by a factor of 10 or more. Within any particular organization, however, your estimates probably won't need to account for that much variation because both top-tier and bottom-tier developers tend to migrate toward organizations that employ other people with similar skill levels (Mills 1983, DeMarco and Lister 1999).

Another implication is that the most accurate estimation approach will depend on whether you know who specifically will be doing the work that's being estimated. That issue is discussed in Chapter 16, "Flow of Software Estimates on a Well-Estimated Project."

5.4 Programming Language

The specific programming language a project uses will affect your estimates in at least four ways.

First, as Figure 5-6 suggested, the project team's experience with the specific language and tools that will be used on the project has about a 40% impact on the overall productivity rate of the project.

Second, some languages generate more functionality per line of code than others. Table 5-3 shows the amount of functionality that several languages produce relative to the C programming language.

Table 5-3 Ratio of High-Level-Language Statements to Equivalent C Code

Language	Level Relative to C
C	1 to 1
C#	1 to 2.5
C++	1 to 2.5
Cobol	1 to 1.5
Fortran 95	1 to 2
Java	1 to 2.5
Macro Assembly	2 to 1

Table 5-3 Ratio of High-Level-Language Statements to Equivalent C Code

Language	Level Relative to C
Perl	1 to 6
Smalltalk	1 to 6
SQL	1 to 10
Visual Basic	1 to 4.5

Source: Adapted from *Estimating Software Costs* (Jones 1998) and *Software Cost Estimation with Cocomo II* (Boehm 2000).

If you don't have any choice about the programming language you're using, this point is not relevant to your estimate. But if you have some leeway in choosing a programming language, you can see that using a language such as Java, C#, or Microsoft Visual Basic would tend to be more productive than using C, Cobol, or Macro Assembly.

A third factor related to languages is the richness of the tool support and environment associated with the language. According to Cocomo II, the weakest tool set and environment will increase total project effort by about 50% compared to the strongest tool set and environment (Boehm et al. 2000). Realize that the choice of programming language might determine the choice of tool set and environment.

A final factor related to programming language is that developers working in interpreted languages tend to be more productive than those working in compiled languages, perhaps as much as a factor of 2 (Jones 1986a, Prechelt 2000).

The concept of amount of functionality produced per line of code will be discussed further in Section 18.2, "Function-Point Estimation."

5.5 Other Project Influences

I've mentioned the Cocomo II estimating model several times in this chapter. As discussed in Chapter 4, I have reservations about subjectivity creeping into the use of Cocomo II's adjustment factors. However, my reservations stem from concerns about "usage failure" more than concerns about "method failure." The Cocomo II project has done a much better job than other studies of rigorously isolating the impacts of specific factors on project outcomes. Most studies combine multiple factors intentionally or unintentionally. A study might examine the impact of software process improvement, but it might not isolate the impact of switching from one programming language to another, or of consolidating staff from two locations to one location. The Cocomo II project has conducted the most statistically rigorous analysis of specific factors that I've seen. So, although I prefer other methods for estimation, I do recommend studying Cocomo II's adjustment factors to gain an understanding of the significance of different software project influences.

Table 5-4 lists the Cocomo II ratings factors for Cocomo's 17 Effort Multipliers (EMs). The Very Low column represents the amount you would adjust an effort estimate for the best (or worst) influence of that factor. For example, if a team had very low "Applications (Business Area) Experience," you would multiply your nominal Cocomo II effort estimate by 1.22. If the team had very high experience, you would multiply the estimate by 0.81 instead.

Table 5-4 Cocomo II Adjustment Factors

	Ratings						
Factor	**Very Low**	**Low**	**Nominal**	**High**	**Very High**	**Extra High**	**Influence**
Applications (Business Area) Experience	1.22	1.10	1.00	0.88	0.81		1.51
Database Size		0.90	1.00	1.14	1.28		1.42
Developed for Reuse		0.95	1.00	1.07	1.15	1.24	1.31
Extent of Documentation Required	0.81	0.91	1.00	1.11	1.23		1.52
Language and Tools Experience	1.20	1.09	1.00	0.91	0.84		1.43
Multisite Development	1.22	1.09	1.00	0.93	0.86	0.78	1.56
Personnel Continuity (turnover)	1.29	1.12	1.00	0.90	0.81		1.59
Platform Experience	1.19	1.09	1.00	0.91	0.85		1.40
Platform Volatility		0.87	1.00	1.15	1.30		1.49
Product Complexity	0.73	0.87	1.00	1.17	1.34	1.74	2.38
Programmer Capability (general)	1.34	1.15	1.00	0.88	0.76		1.76
Required Software Reliability	0.82	0.92	1.00	1.10	1.26		1.54
Requirements Analyst Capability	1.42	1.19	1.00	0.85	0.71		2.00
Storage Constraint			1.00	1.05	1.17	1.46	1.46
Time Constraint			1.00	1.11	1.29	1.63	1.63
Use of Software Tools	1.17	1.09	1.00	0.90	0.78		1.50

The Influence column on the far right of the table shows the degree of influence that each factor, in isolation, has on the overall effort estimate. The Applications (Business Area) Experience factor has an influence of 1.51, which means that a project performed by a team with very low skills in that area will require 1.51 times as much total effort as a project performed by a team with very high skills in that area. (Influence is computed by dividing the largest value by the smallest value. For example, 1.51 is 1.22/0.8.)

Figure 5-7 presents another view of the impact of the Cocomo II factors, in which the factors are arranged from most significant influence to least significant influence.

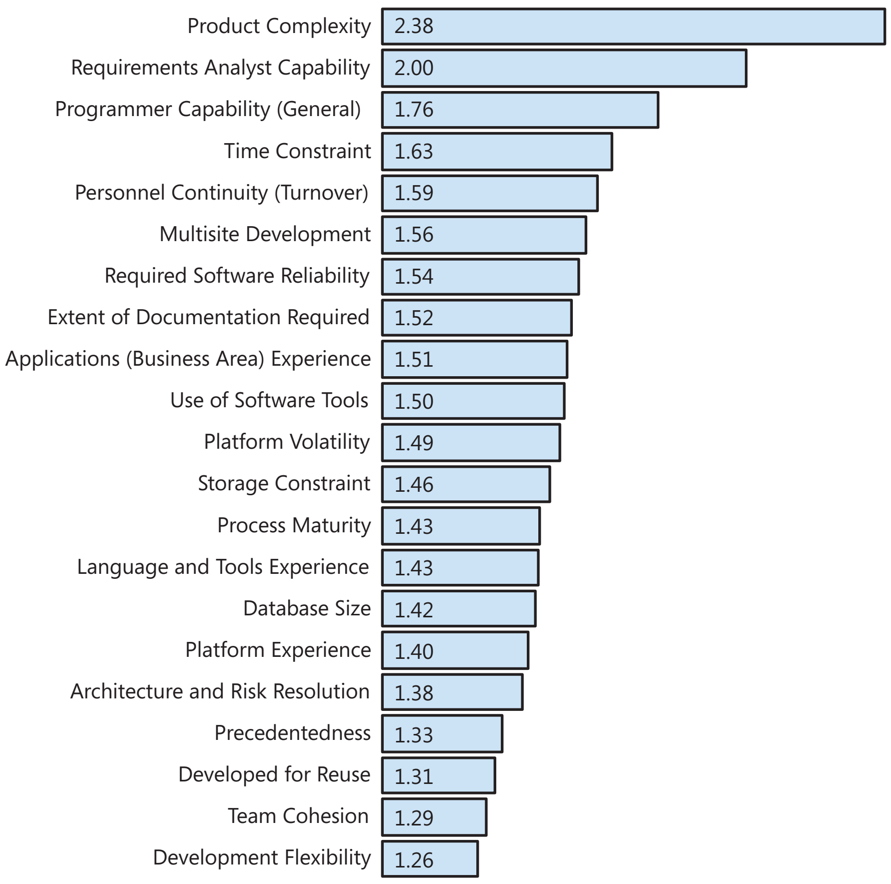


Figure 5-7 Cocomo II factors arranged in order of significance. The relative lengths of the bars represent the sensitivity of the estimate to the different factors.

Figure 5-8 shows the same factors represented in terms of their potential to increase total effort (the gray bars) versus decrease effort (the blue bars).

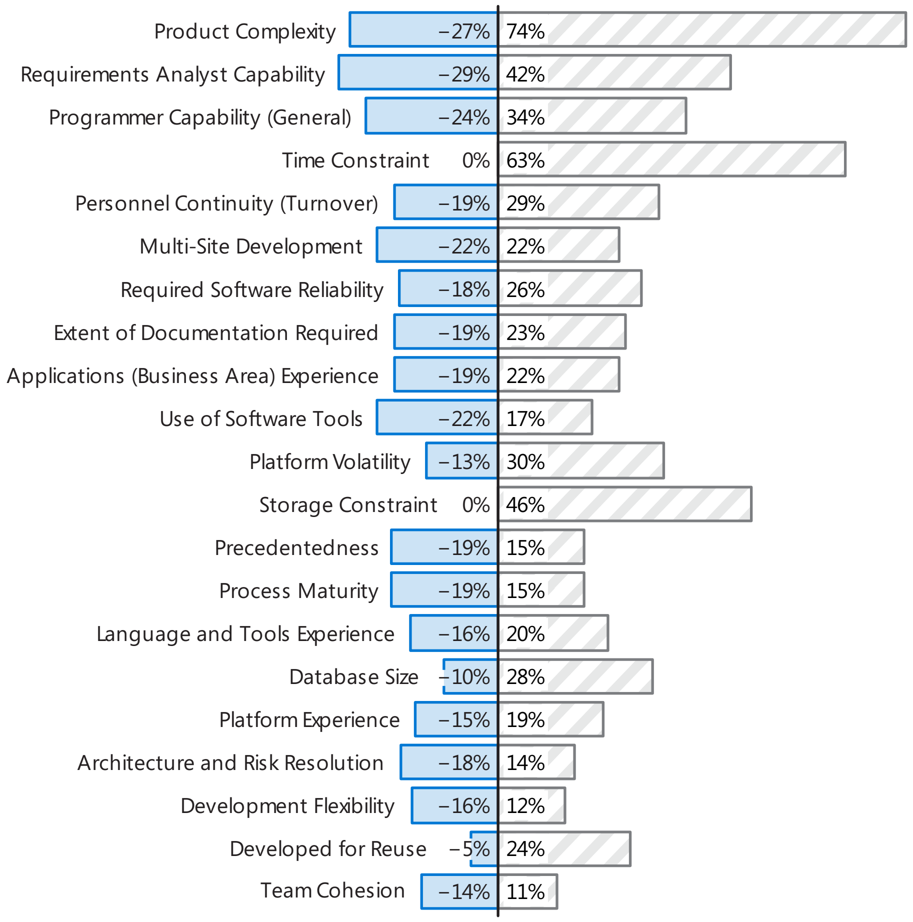


Figure 5-8 Cocomo II factors arranged by potential to increase total effort (gray bars) and potential to decrease total effort (blue bars).

I've listed some observations about these factors in Table 5-5 in alphabetical order.

Table 5-5 Cocomo II Adjustment Factors

Cocomo II Factor	Influence	Observation
Applications (Business Area) Experience	1.51	Teams that aren't familiar with the project's business area need significantly more time. This shouldn't be a surprise.
Architecture and Risk Resolution	1.38*	The more actively the project attacks risks, the lower the effort and cost will be. This is one of the few Cocomo II factors that is controllable by the project manager.
Database Size	1.42	Large, complex databases require more effort project-wide. Total influence is moderate.

Table 5-5 Cocomo II Adjustment Factors

Cocomo II Factor	Influence	Observation
Developed for Reuse	1.31	Software that is developed with the goal of later reuse can increase costs as much as 31%. This doesn't say whether the initiative actually succeeds. Industry experience has been that forward-looking reuse programs often fail.
Extent of Documentation Required	1.52	Too much documentation can negatively affect the whole project. Impact is moderately high.
Language and Tools Experience	1.43	Teams that have experience with the programming language and/or tool set work moderately more productively than teams that are climbing a learning curve. This is not a surprise.
Multi-Site Development	1.56	Projects conducted by a team spread across multiple sites around the globe will take 56% more effort than projects that are conducted by a team co-located at one facility. Projects that are conducted at multiple sites, including outsourced or offshore projects, need to take this effect seriously.
Personnel Continuity (turnover)	1.59	Project turnover is expensive—in the top one-third of influential factors.
Platform Experience	1.40	Experience with the underlying technology platform affects overall project performance moderately.
Platform Volatility	1.49	If the platform is unstable, development can take moderately longer. Projects should weigh this factor in their decision about when to adopt a new technology. This is one reason that systems projects tend to take longer than applications projects.
Precedentedness	1.33*	Refers to how "precedented" (we usually say "unprecedented") the application is. Familiar systems are easier to create than unfamiliar systems.
Process Maturity	1.43*	Projects that use more sophisticated development processes take less effort than projects that use unsophisticated processes. Cocomo II uses an adaptation of the CMM process maturity model to apply this criterion to a specific project.
Product Complexity	2.38	Product complexity (software complexity) is the single most significant adjustment factor in the Cocomo II model. Product complexity is largely determined by the type of software you're building.
Programmer Capability (general)	1.76	The skill of the programmers has an impact of a factor of almost 2 on overall project results.

Table 5-5 Cocomo II Adjustment Factors

Cocomo II Factor	Influence	Observation
Required Reliability	1.54	More reliable systems take longer. This is one reason (though not the only reason) that embedded systems and life-critical systems tend to take more effort than other projects of similar sizes. In most cases, your marketplace determines how reliable your software must be. You don't usually have much latitude to change this.
Requirements Analyst Capability	2.00	The single largest personnel factor—good requirements capability—makes a factor of 2 difference in the effort for the entire project. Competency in this area has the potential to reduce a project's overall effort from nominal more than any other factor.
Requirements Flexibility	1.26*	Projects that allow the development team latitude in how they interpret requirements take less effort than projects that insist on rigid, literal interpretations of all requirements.
Storage Constraint	1.46	Working on a platform on which you're butting up against storage limitations moderately increases project effort.
Team Cohesion	1.29*	Teams with highly cooperative interactions develop software more efficiently than teams with more contentious interactions.
Time Constraint	1.63	Minimizing response time increases effort across the board. This is one reason that systems projects and real-time projects tend to consume more effort than other projects of similar sizes.
Use of Software Tools	1.50	Advanced tool sets can reduce effort significantly.

* Exact effect depends on project size. Effect listed is for a project size of 100,000 LOC. These factors are discussed in the next section.

As I hinted earlier, studying the Cocomo II adjustment factors to gain insight into your project's strengths and weaknesses is a high-leverage activity. For the estimate itself, using historical data from your past or current projects tends to be easier and more accurate than tweaking Cocomo's 22 adjustment factors.

Chapter 8 will discuss the ins and outs of collecting and using historical data.

5.6 Diseconomies of Scale Revisited

The Cocomo II adjustment factors provide an interesting viewpoint into how diseconomies of scale operate. In Figure 5-9, 5 of the factors in the figure are called *scaling factors*. These are the factors that contribute to software's diseconomies of scale. They affect projects to different degrees at different sizes. Figure 5-9 shows the same graph with these factors highlighted in blue.

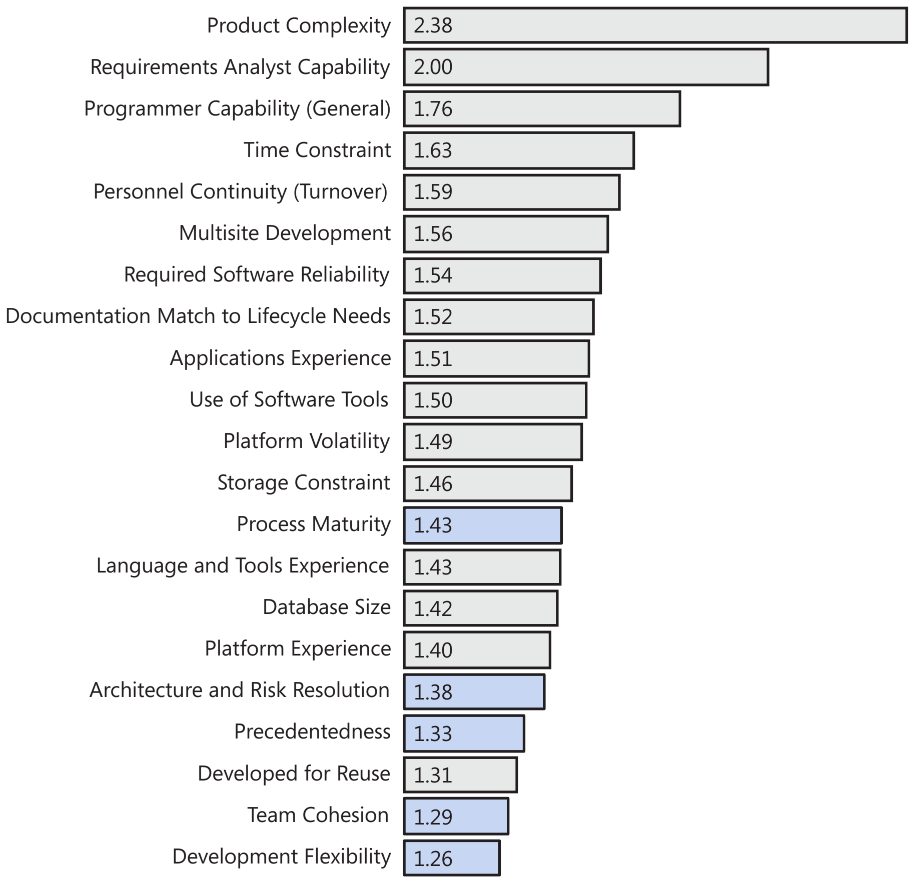


Figure 5-9 Cocomo II factors with diseconomy of scale factors highlighted in blue. Project size is 100,000 LOC.

None of the factors that contribute to software's diseconomy of scale is in the top half of factors in terms of significance. In fact, 4 of the 5 least-influential factors are scaling factors. However, because the scaling factors contribute different amounts at different project sizes, this diagram must be drawn from the point of view of a project of a specific size. The factors in Figure 5-9 are shown for a project of 100,000 lines of code. Figure 5-10 shows what happens when the factors are recalculated for a much larger project of 5,000,000 lines of code.

The scaling factors all become significant as project size increases. Although none of them was in the top half at 100,000 LOC, all the scaling factors are in the top half at 5,000,000 LOC.

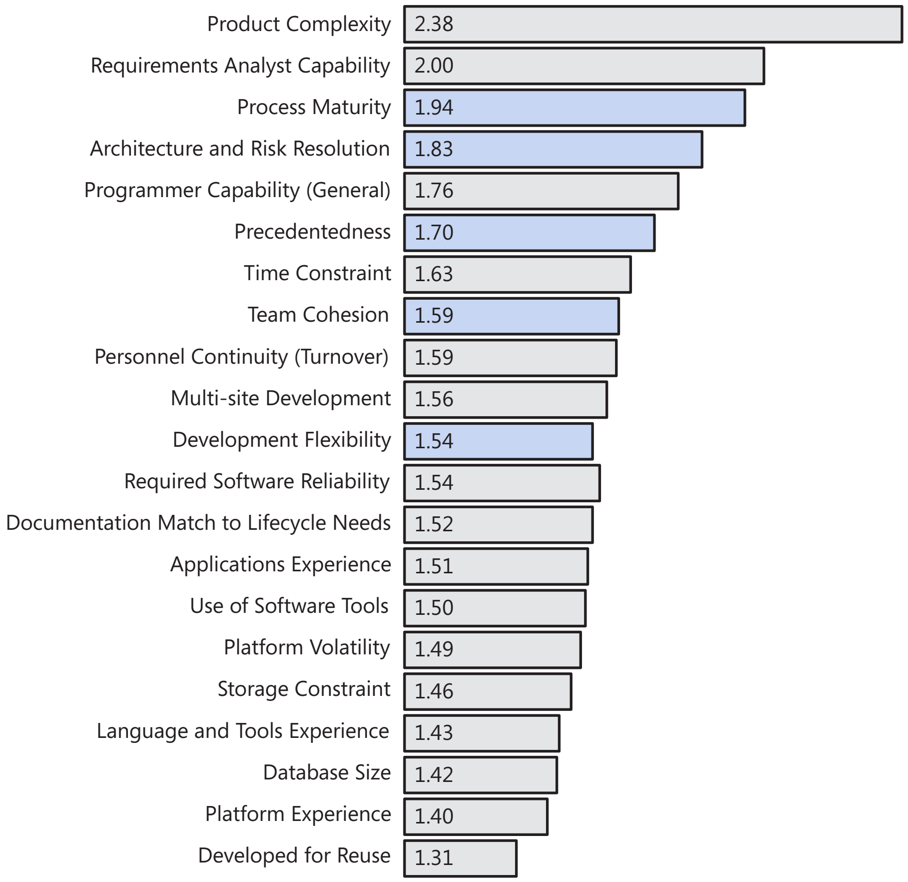


Figure 5-10 Cocomo II factors with diseconomy of scale factors highlighted in blue. Project size is 5,000,000 LOC.

What this means from an estimating point of view is that different factors need to be weighted differently at different project sizes. What this means from a project planning and control point of view is that small and medium-sized projects can succeed largely on the basis of strong individuals. Large projects still need strong individuals, but how well the project is managed (especially in terms of risk management), how mature the organization is, and how well the individuals coalesce into a team become as significant.

Additional Resources

Boehm, Barry, et al. *Software Cost Estimation with Cocomo II*. Reading, MA: Addison-Wesley, 2000. This book is the definitive description of Cocomo II. The book's size is daunting, but it describes the basic Cocomo model within the first 80 pages, including detailed definitions of the effort multipliers and scaling factors discussed in this chapter and how Cocomo II accounts for diseconomies of scale. The rest of the book describes extensions of the model.

Putnam, Lawrence H. and Ware Myers. *Measures for Excellence: Reliable Software On Time, Within Budget.* Englewood Cliffs, NJ: Yourdon Press, 1992. This book describes Putnam's estimation method including how it addresses diseconomies of scale. I like Putnam's model because it contains few control knobs and works best when it is calibrated with historical data. The book is mathematically oriented, so it can be slow going.