Chapter 21

Seven Basic Steps in Software Cost Estimation

To get a reliable software cost estimate, we need to do much more than just put numbers into a formula and accept the results. This chapter provides a seven-step process for software cost estimation, which shows that a software cost estimation activity is a mini-project and should be planned, reviewed, and followed up accordingly.

The seven basic steps are

1. Establish Objectives
2. Plan for Required Data and Resources
3. Pin Down Software Requirements
4. Work Out as Much Detail as Feasible
5. Use Several Independent Techniques and Sources
6. Compare and Iterate Estimates
7. Followup

Each of these steps is discussed in more detail in Sections 21.1 through 21.7.
21.1 STEP 1: ESTABLISH OBJECTIVES

In software cost estimation, a lot of effort can be wasted in gathering information and making estimates on items that have no relevance to the need for the estimate. For example, in one situation, an extremely detailed conversion estimate was made to support a decision on whether or not to upgrade to a different make of computer. The decision required only a general estimate of conversion costs, and when it was decided not to go to a different make of computer, a great deal of hard work and careful analysis was thrown out. Thus, it is extremely important to establish the objectives of the cost estimate as the first step, and to use these objectives to drive the level of detail and effort required to perform the subsequent steps.

Objectives versus Phase, or Level of Knowledge

The main factor that helps us establish our cost-estimation objectives is our current software life-cycle phase. It largely corresponds with our level of knowledge of the software whose costs we are trying to estimate, and also to the level of commitment we will be making as a result of the cost estimate.

Figure 21-1 illustrates the accuracy within which software cost estimates can be made, as a function of the software life-cycle phase (the horizontal axis), or of the level of knowledge we have of what the software is intended to do. This level of uncertainty is illustrated in Fig. 21-1 with respect to a human-machine interface component of the software.

When we first begin to evaluate alternative concepts for a new software application, the relative range of our software cost estimates is roughly a factor of four on either the high or low side.* This range stems from the wide range of uncertainty we have at this time about the actual nature of the product. For the human-machine interface component, for example, we don't know at this time what classes of people (clerks, computer specialists, middle managers, etc.) or what classes of data (raw or pre-edited, numerical or text, digital or analog) the system will have to support. Until we pin down such uncertainties, a factor of four in either direction is not surprising as a range of estimates.

The above uncertainties are indeed pinned down once we complete the feasibility phase and settle on a particular concept of operation. At this stage, the range of our estimates diminishes to a factor of two in either direction. This range is reasonable because we still have not pinned down such issues as the specific types of user query to be supported, or the specific functions to be performed within the microprocessor in the intelligent terminal. These issues will be resolved by the time we have developed a software requirements specification, at which point, we will be able to estimate the software costs within a factor of 1.5 in either direction.

By the time we complete and validate a product design specification, we will have resolved such issues as the internal data structure of the software product and the specific techniques for handling the buffers between the terminal microprocessor

* These ranges have been determined subjectively, and are intended to represent 80% confidence limits, that is, "within a factor of four on either side, 80% of the time."
and the central processors on one side, and between the microprocessor and the display driver on the other. At this point, our software estimate should be accurate to within a factor of 1.25, the discrepancies being caused by some remaining sources of uncertainty such as the specific algorithms to be used for task scheduling, error handling, abort processing, and the like.* These will be resolved by the end of the detailed design phase, but there will still be a residual uncertainty about 10%, based on how well the programmers really understand the specifications to which they are to code. (This factor also includes such considerations as personnel turnover uncertainties during the development and test phases.)

**Estimating Implications**

The primary estimating implication of Fig. 21-1 is that we need to be consistent in defining our estimating objectives for the various components of the software product. If our understanding of the human-machine interface is at the concept-of-operation level, for example, it will generally be a waste of effort to define and estimate conversion costs at the detailed-design level. (The exception to this statement is the situation

* Within the factor of 1.25 range (80-125%), a good software manager can generally turn a software effort estimate into a self-fulfilling prophecy. See Chapter 32.
in which conversion costs are an order of magnitude larger than the human-machine 
interface software costs.)

In general, we wish to achieve a balanced set of estimating objectives: one in 
which the absolute magnitude of the uncertainty range for each component is roughly 
equal. Thus, suppose we have a human-machine interface component of roughly 
$1,000,000 defined at the requirements-spec level (say, within the range $667,000 to 
$1,500,000). If we then have a conversion component of roughly $500,000, we can 
afford to define it at the concept-of-operation level, since the resulting range 
($250,000-$1,000,000) is roughly the same magnitude as the range on the human-
machine interface component.

Another estimating implication is that each cost estimate should include an indica-
tion of its degree of uncertainty.

Relative versus Absolute Estimates

In other situations, even a concept-of-operation level conversion estimate may 
not be necessary. For example, suppose we are making cost estimates to support a 
make-or-buy decision, and the conversion costs will be roughly the same for each 
option. Then, for the make-or-buy decision, there is no need to have an estimate of 
conversion costs at all. Of course, at some subsequent stage, an overall life-cycle 
cost estimate will be required, including a conversion cost estimate, but at that point, 
our increased knowledge of the system will make it easier to perform the conversion 
estimate.

Here, the major concern is to make sure that our estimating objectives are consis-
tent with the needs of the decisionmaker who will use the estimate.* Thus, we are 
dealing with the same issues—of balancing the cost of obtaining information with 
the value of the information to a decisionmaker—that we discussed in Chapter 20.

Generous versus Conservative Estimates

Often, in a cost analysis to support a make-or-buy decision or a go/no-go decision 
on system development, it becomes clear early in the analysis that a particular decision 
is highly likely to result. In such a situation, we may wish to revise our objectives 
to try to demonstrate the following:

Even if we use conservative assumptions for Option A and generous assumptions 
for Option B, Option A is still the more cost-effective.

This sort of revision has two major benefits. First, it increases our confidence that 
we are making the right decision. Second, the ability to make generous or conservative 
assumptions will often simplify our cost-estimation effort. For example, we might 
assume that a potentially adaptable software component is completely adaptable (gen-
erous), or completely unadaptable (conservative), rather than perform an analysis 
to determine its adaptation adjustment factor (AAF).

From the standpoint of estimation objectives, this means that we should reexamine

* For an excellent general treatment of the use of cost information in decisionmaking, see Cost Consider-
our objectives as we proceed, and modify them when a change is advantageous (that is, we may begin with a requirements level accuracy objective, but relax it to a generous or conservative concept-of-operation level accuracy objective later in the analysis).

**Summary Guidelines**

In summary, here are the three major guidelines for establishing the objectives of a cost-estimation activity:

1. *Key the estimating objectives to the needs for decisionmaking information.* (Absolute estimates for labor or resource planning, relative estimates for either/or decisions, generous or conservative estimates to heighten confidence in the decision.)

2. *Balance the estimating accuracy objectives for the various system components of the cost estimates.* (This means that the absolute magnitude of the uncertainty range for each component should be roughly equal—assuming that such components have equal weight in the decision to be made.)

3. *Re-examine estimating objectives as the process proceeds, and modify them where appropriate.* (A further implication of this guideline is that budget commitments in the early phases should cover only the next phase. Once a validated product design is complete, a total development budget may be established without too much risk.)

**21.2 STEP 2: PLAN FOR REQUIRED DATA AND RESOURCES**

The following scenario is all too common:

*Rumpled Proposal Manager:* We've got this proposal that has to be signed off by noon so we can get it on the afternoon plane to Washington. Can you work me up a quick software cost estimate for it?

*Software Cost Estimator:* You want it when???

Typically, this scenario (and many similar ones) leads to a terribly inaccurate software cost estimate, which becomes cast into an ironclad organizational commitment (usually underpriced) affecting a lot of innocent software people who deserve a much better fate.

If we consider the software cost-estimation activity as a miniproject, then we automatically cover this problem by generating a project plan at an early stage. Table 21–1 shows a simple general form for a project plan which applies quite naturally to the cost-estimating miniproject.

The miniplan doesn't have to be a fancy, detailed document, particularly if your estimating activity is small. But even an informal early set of notes to yourself on
the why, what, when, who, where, how, how much, and whereas of your estimating activity will often save your neck, and the necks of all those software people who have to perform to your estimate.

An example software cost-estimation plan to support a feasibility study of a computer-controlled rapid transit system is shown in Fig. 21-2.

TABLE 21-1  Software Cost-Estimating Miniproject Plan

1. **Purpose**: Why is the estimate being made?

2. **Products and Schedules**: What is going to be furnished by **when**?

3. **Responsibilities**: Who is responsible for each product? Where are they going to do the job organizationally? geographically?

4. **Procedures**: How is the job going to be done? Which cost-estimation tools and techniques will be used (see Chapter 22)?

5. **Required Resources**: How much data, time, money, effort, etc. is needed to do the job?

6. **Assumptions**: Under what conditions are we promising to deliver the above estimates, given the above resources (availability of key personnel, computer time, user data)?

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Purpose:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>To help determine the feasibility of a computer-controlled rapid transit system for the Zenith metropolitan area.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Products and Schedules:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2/1/84</td>
<td>Cost-estimation plan</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2/15/84</td>
<td>First cost model run</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2/22/84</td>
<td>Definitive cost model run</td>
<td>Expert estimates complete</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2/29/84</td>
<td>Final cost estimate report, incorporating model and expert iterations. Accuracy to within factor of 2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Responsibilities:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cost-estimation study: Z.B. Zimmerman</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cost model support: Application Software Department</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expert estimators (2): Systems Analysis Department</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Procedures:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project will use SOFTCOST model, with sensitivity analysis on high-leverage cost driver attributes.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Experts will contact BART personnel in San Francisco and Metro personnel in Washington, D.C. for comparative data.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Required Resources:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z.B. Zimmerman: 2 man-weeks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expert estimators: 3 man days each</td>
<td>Computer: &lt;200.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. Assumptions:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No major changes to system specification dated 15 January 1984.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Authors of specification available to answer sizing questions.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FIGURE 21-2  Software cost estimation plan: Zenith rapid transit system

314  THE ART OF SOFTWARE COST ESTIMATION  PART IV
21.3 STEP 3. PIN DOWN SOFTWARE REQUIREMENTS

If we don't know what products we are building, we certainly can't estimate the cost of building them very well. This means that it is important to have a set of software specifications that are as unambiguous as possible (subject to qualifications with respect to our estimating objectives).

The best way to determine to what extent a software specification is costable is to determine to what extent it is testable.

A specification is testable to the extent that one can define a clear pass/fail test for determining whether or not the developed software will satisfy the specification. In order to be testable, specifications must be specific, unambiguous, and quantitative wherever possible. Below are some examples of specifications which are not testable:

- The software shall provide interfaces with the appropriate subsystems.
- The software shall degrade gracefully under stress.
- The software shall be developed in accordance with good development standards.
- The software shall provide the necessary processing under all modes of operation.
- Computer memory utilization shall be optimized to accommodate future growth.
- The software shall provide a 99.9999% assurance of information privacy (or reliability, availability, or human safety, when these terms are undefined).
- The software shall provide accuracy sufficient to support effective flight control.
- The software shall provide real-time response to sales activity queries.

These statements are good as goals and objectives, but they are not precise enough to serve as the basis of a pass-fail acceptance test, or to serve as the basis of an accurate cost estimate. Below are some more testable versions of the last two requirements:

- The software shall compute aircraft position within the following accuracies:
  ±50 feet in the horizontal plane
  ±20 feet in the vertical plane
- The system shall respond to:
  Type A queries in ≤2 sec
  Type B queries in ≤10 sec
  Type C queries in ≤2 min
  where Type A, B, and C queries are defined in detail in the specification.

In many cases, even these versions will not be sufficiently testable without further definition. For example:

- Do the terms “±50 ft” or “≤2 sec” refer to root mean square performance, 90% confidence limits, or never-to-exceed constraints?
- Does “response” time include terminal delays, communications delays, or just the time involved in computer processing?
Thus, it will often require a good deal of added effort to eliminate the vagueness and ambiguity in a specification and make it testable. But such effort is generally well worthwhile, for the following reasons:

- It would have to be done eventually for the test phase anyway.
- Doing it early eliminates a great deal of expense, controversy, and possible bitterness in later stages.
- Doing it early means that we can generate more accurate cost estimates.

In many cases, it will be impossible or infeasible to make sure all of the software requirements are testable (see the discussion in Section 4.3). Or, it may require more effort than we need to satisfy our estimation objectives. In such cases, it is valuable to document any assumptions that were made in estimating the cost of developing the software, particularly if they were generous or conservative assumptions as discussed in Step 1.

### 21.4 STEP 4. WORK OUT AS MUCH DETAIL AS FEASIBLE

"As feasible" here means "as is consistent with our cost-estimating objectives," as discussed in Section 21.1. In general, the more detail to which we carry our estimating activities, the more accurate our estimates will be, for three main reasons:

1. The more detail we explore, the better we understand the technical aspects of the software to be developed, as indicated by Fig. 21-1 and its discussion.
2. The more pieces of software we estimate, the more we get the law of large numbers working for us to reduce the variance of the estimate. If we have one large piece of software and overestimate its cost by 20%, we are stuck with a 20% error. If we break the large piece into 10 smaller pieces, we may underestimate on most of the pieces, but overestimate on some, and on balance end up with a considerably smaller estimating error.
3. The more we think through all the functions the software must perform, the less likely we are to miss the costs of some of the more unobtrusive components of the software.

As an example of item 3, Fig. 21-3 shows the results of an experiment in which two teams specified and developed a small (2000-DOS) software product (actually, an interactive early version of the Detailed Cocomo model, performed as a group project in a software engineering class at USC [Boehm, 1980]). The most significant aspect of Fig. 21-3 is that the actual cost model calculations comprised only 2% of the code in one product, and 3% of the code in the other. Much of the remainder of the code was involved with the unobtrusive components of the software, such as help message processing, error processing, and moving data around. These overhead functions are often missed in software sizing and cost estimating. This is one of the...
main reasons that software costs are so often underestimated: There is a powerful tendency to focus on the highly visible mainline components of the software, and to underestimate or completely miss the unobtrusive components.

**On Software Sizing**

It would be convenient if we could provide some software sizing formulas that could say, for example:

*If we are developing an operating system which performs the following functions thoroughly, the following functions minimally, and the following functions not at all, then the estimated size of the operating system is 11 ± 2 KDSI.*

Unfortunately, quantitative software engineering has not progressed to the point that we can even begin to provide such formulas. And it is not clear that we will ever get very close to such an ideal. Having spent a good deal of time looking at sizing data and the programs they represent, and generally in going around in circles in pursuit of a simplified sizing formula, I would summarize the experience in terms of some analogies.

1. Solving the automatic software sizing problem has a good many of the aspects of solving the automatic programming problem. For example, both require a sufficiently detailed specification of the desired software to assure that some different, undesired neighboring piece of software will not be what is sized or generated. And providing this specification goes a long way toward sizing the software itself.
2. Generating a formula for sizing software has a good many of the aspects of generating a formula for sizing a novel. Both deal with a product capable of virtually unlimited levels of elaboration, and it is difficult to characterize these levels of elaboration in any way related to sizing. Just consider the problem of estimating the number of pages in a novel with

- four characters who influence each others’ lives profoundly
- 20 more or less incidental characters
- three different locations
- two years’ time span
- five detailed flashbacks

and you begin to get a better appreciation of the software sizing problem. Some further appreciation of this point can be obtained from the sizing data in the [Weinberg-Schulman, 1974] experiment discussed in Chapter 3. In the experiment, six teams were asked to develop the same program (solution of simultaneous linear equations by Gaussian elimination), but were given different objectives to optimize. The resulting programs had a 5:1 variation in size for implementing the same function, as shown below.

<table>
<thead>
<tr>
<th>Team Objective: Optimize</th>
<th>Program Size (DSI)</th>
<th>Man-Hours</th>
<th>Productivity (DSI/MH)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program size</td>
<td>33</td>
<td>30</td>
<td>1.1</td>
</tr>
<tr>
<td>Memory required</td>
<td>52</td>
<td>74</td>
<td>0.7</td>
</tr>
<tr>
<td>Program clarity</td>
<td>90</td>
<td>40</td>
<td>2.2</td>
</tr>
<tr>
<td>Execution time</td>
<td>100</td>
<td>50</td>
<td>2.0</td>
</tr>
<tr>
<td>Effort to complete</td>
<td>126</td>
<td>28</td>
<td>4.5</td>
</tr>
<tr>
<td>Output clarity</td>
<td>166</td>
<td>30</td>
<td>5.5</td>
</tr>
</tbody>
</table>

On the other hand, the software sizing problem doesn’t have all of the aspects of the automatic programming problem or the novel-sizing problem, so there is some hope of making eventual progress. In the meantime, however, there is no substitute for a detailed understanding of each software component to ensure accurate software sizing.

**PERT Sizing**

One implication of the discussion above of software sizing is that we should be careful not to make sizing appear easier than it is. One technique which unfortunately does this is the PERT sizing technique discussed in [Putnam-Fitzsimmons, 1979].

The simplest version of this technique involves estimating two quantities

\[ a = \text{The lowest possible* size of the software (say, 22 KDSI)} \]
\[ b = \text{The highest possible size of the software (say, 64 KDSI)} \]

* These formulas are based on the understanding that the low and high estimates \( a \) and \( b \) represent 3\sigma (three standard deviation) limits on the probability distribution of the actual software size. For a normal probability distribution, this means that the actual software size would lie between \( a \) and \( b \) 99.7% of the time.
Then the PERT statistical equations estimate the expected size of the software as

\[ E = \frac{a + b}{2} = 43 \text{ KDSI} \]

and the standard deviation of the estimate as

\[ \sigma = \frac{b - a}{6} = 7 \text{ KDSI} \]

This means that 68% of the time, the actual size of the software should fall between 36 and 50 KDSI (and that about 16% of the time each, the actual size should fall between the ranges 22–36 KDSI and 50–64 KDSI).

These formulas are based on the assumption of a normal distribution of sizes between the two extremes \( a \) and \( b \). However, anyone familiar with current software practice will recognize that if the upper limit \( b \) is 64 KDSI because this is the maximum amount of code that will fit in the machine, there is much more than a 16% chance that the final size of the software will be between 50 KDSI and 64 KDSI.

A somewhat better PERT sizing technique discussed in [Putnam–Fitzsimmons, 1979] is one based on a beta distribution and on the separate estimation of individual software components. Here, three sizing quantities are generated for each component:

- \( a_i \) = The lowest possible size of the software component
- \( m_i \) = The most likely size of the component
- \( b_i \) = The highest possible size of the component

The PERT equations estimate the expected size \( E_i \) and standard deviation \( \sigma_i \) of each component as

\[ E_i = \frac{a_i + 4m_i + b_i}{6} \quad \sigma_i = \frac{b_i - a_i}{6} \]

The estimated total software size \( E \) and standard deviation \( \sigma E \) are then

\[ E = \sum_{i=1}^{n} E_i, \quad \sigma E = \left( \sum_{i=1}^{n} \sigma_i^2 \right)^{1/2} \]

For example, suppose we are estimating the size of the software to be developed for a 64K-word microprocessor point-of-sale terminal. The individual estimates and resulting overall estimates are shown in Table 21–2.

This PERT sizing technique is somewhat better in that it requires more thought to break up the software into components and to estimate most likely sizes for each component as well as upper and lower limits. Again, however, the calculation of
\( \sigma E \) is highly misleading, as it assumes that the estimates are unbiased toward either underestimation or overestimation. Current experience, however, indicates that “most likely” estimates tend to cluster more toward the lower limit than the upper limit, while actual product sizes tend to cluster more toward the upper limit, imparting a significant underestimation bias to PERT results.

In this example, the estimated \( \sigma E \) implies that there is a 68% chance that the actual size of the microprocessor point-of-sale software will be between 35.4K and 42.6K words, and that sizes between 49.8K and 64K words would only occur 0.15% of the time. Again, experience to date would lead us to expect these larger sizes to be a much more frequent occurrence than this.

**Why Do People Underestimate Software Size?**

The software undersizing problem is our most critical road block to accurate software cost estimation. Software cost models, like other computer models, are “garbage in—garbage out” devices: put a too-small sizing estimate in, and you will get a too-small cost estimate out.

Our discussion of sizing formulas above should convince us that there are no magic formulas that we can use to overcome the software undersizing problem. In the absence of any such formula, it is important to understand the major sources of the software undersizing problem, for it is only from understanding them that we will be able to overcome them.

Current experience indicates that there are three main reasons why people underestimate software size. These are:

1. **People are basically optimistic and desire to please.** Everybody would like the software to be small and easy. High estimates lead to confrontation situations, which people generally prefer to avoid. This phenomenon is not limited to software sizing. Figure 21-4, from [Augustine, 1979], displays the estimated completion times versus the actual completion times for about 100 recent projects.

   *Most people tend to follow a geometric progression in making “most likely” estimates, rather than an arithmetic progression or something even more pessimistic. Thus, given low and high limits of 16K and 64K, people are more likely to choose the geometric mean of 32K as their “most likely” estimate, rather than the arithmetic mean of 40K, and very rarely choose a “most likely” estimate of 48K or higher.*
FIGURE 21-4  Accuracy of projecting accomplishment date for major milestones

official schedule estimates within the Department of Defense, showing a fairly consistent “fantasy factor” of about 1.33.

2. **People tend to have incomplete recall of previous experience.** In terms of the distribution of source code by function in Fig. 21-3, for example, people tend to have a strong recollection of the primary application software functions to be developed—the 2 to 3% of the product devoted to model calculations in Fig. 21-3—and a much weaker recollection of the large amount of user-interface and housekeeping software that must also be developed.*

3. **People are generally not familiar with the entire software job.** This factor tends to interact with the incomplete-recall factor to produce underestimates of the more obscure software products to be developed as well as of the more obscure portions of each product. A major example is a strong tendency to underestimate the size of support software, which for large operational systems is generally three to five times as large as the operational software. Some typical comparative sizes for large operational systems are given in Table 21-3. Although the sizing is in different units between projects and even displays a variability between two summaries of the same project (Safeguard), the general pattern of the results is fairly consistent. As indicated in the final column, a typical very large (500 KDSI) operational system will contain

* A similar underestimating phenomenon holds for estimating software development activity; see Section 22.7.
### TABLE 21-3 The Preponderance of Support Software on Very Large Systems

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Operational</td>
<td>653</td>
<td>789</td>
<td>276</td>
<td>20</td>
<td>280</td>
<td>200</td>
<td>100</td>
</tr>
<tr>
<td>On-site support</td>
<td>630 (1.06)</td>
<td>100+ (1.3+)</td>
<td>525 (2.72)</td>
<td>40 (2.00)</td>
<td>1190 (4.24)</td>
<td>430 (2.15)</td>
<td>150 (1.5)</td>
</tr>
<tr>
<td>(maintenance and diagnostics)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Development support</td>
<td>913 (1.40)</td>
<td>532 (1.67)</td>
<td>525 (1.90)</td>
<td>40 (2.00)</td>
<td>1190 (4.24)</td>
<td>430 (2.15)</td>
<td>150 (1.5)</td>
</tr>
<tr>
<td>(compilers, tools, utilities)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Off-site support</td>
<td>835 (1.28)</td>
<td>840 (1.05)</td>
<td>751 (2.72)</td>
<td>5 (0.25)</td>
<td>244 (0.87)</td>
<td>430 (2.15)</td>
<td>150 (1.5)</td>
</tr>
<tr>
<td>(simulation, data reduction, training)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>2378 (3.64)</td>
<td>1472+ (1.87+)</td>
<td>1276+ (4.62+)</td>
<td>75 (3.75)</td>
<td>1434 (5.12)</td>
<td>860+ (4.30+)</td>
<td>400 (5.0)</td>
</tr>
<tr>
<td>Nonoperational</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>3031 (5.64)</td>
<td>2261+ (3.64)</td>
<td>1552+ (5.75)</td>
<td>95 (5.0)</td>
<td>1747 (5.12)</td>
<td>1060+ (4.30+)</td>
<td>500 (5.0)</td>
</tr>
</tbody>
</table>

**NOTE:**
- $x$: size is $x$ times size of operational software.
- $y+$: size is at least as large as $y$.
- KDEMI: thousands of delivered, executable machine instructions.
only about 100 KDSI of mission-oriented operational software, with another 100 KDSI of hardware and system-related maintenance and diagnostic software, 150 KDSI of development support software (compilers, tools, utilities, etc.), and a final 150 KDSI of mission-oriented support software (simulation, data reduction, and training software).

In summary, then, there is no royal road to software sizing. There is no substitute for a thorough understanding of the software job to be done; a thorough understanding of our basic tendencies to underestimate software size; and a thoughtful, realistic application of this understanding to our software sizing activities.

### 21.5 STEP 5: USE SEVERAL INDEPENDENT TECHNIQUES AND SOURCES

Chapter 22 will discuss the major classes of techniques available for software cost estimating. They are

1. Algorithmic Models
2. Expert Judgment
3. Analogy
4. Parkinson
5. Price-to-Win
6. Top-Down
7. Bottom-Up

The main conclusions of Chapter 22 are

- None of the alternatives is better than the others from all aspects
- Their strengths and weaknesses are complementary

Therefore, it is important to use a combination of techniques, in order to avoid the weaknesses of any single method and to capitalize on their joint strengths. An excellent example of practical experience in surveying and using a combination of software cost estimation techniques on a large contract software procurement is given in [Lasher, 1979].

### 21.6 STEP 6: COMPARE AND ITERATE ESTIMATES

The most valuable aspect of using several independent cost-estimation techniques is the opportunity to investigate why they give different estimates. Thus, if a bottom-up technique estimated a software cost of $4 million, and a top-down technique estimated at $7 million, we could probe into the reasons for the difference by identifying the components of the cost in each and pinning down the differences in detail.

For example, we might find that the bottom-up technique had overlooked system
level activities such as integration, configuration management, and quality assurance, while the top-down technique had included these but overlooked some postprocessing software components included in the bottom-up estimate. An iteration of the two estimates may converge to a more realistic estimate of $8 million rather than some arbitrary compromise between $4 million and $7 million. Thus, it is important not just to perform independent estimates, but also to investigate why they produce different results.  

The Optimist/Pessimist Phenomenon

There are two other major reasons to iterate a cost estimate: the optimist/pessimist phenomenon, and the “tall pole in the tent” phenomenon.

In multicomponent estimates, one often finds very similar components with strikingly different cost-per-instruction estimates. This is often because personal differences cause some people to be highly optimistic about estimates and others to be highly pessimistic. Often, the differences are due to the person’s role and incentives. A proposal manager, who is rewarded for winning the job, is more likely to be optimistic. A project or line manager, who is rewarded for performing the job within budget, is more likely to be pessimistic.

Because of this phenomenon, it is valuable to include some overlap in the software components estimated by different people, and to calibrate the relative optimism and pessimism of different estimators.

The Tall Pole in the Tent Phenomenon

In most multicomponent estimates, there are one or two components whose costs stand out as tall poles in the tent, often containing the majority of the software costs. In such cases, it is particularly important to examine and iterate these components in greater detail than the others.

These components tend to be the larger ones in size, and are frequently overestimated with respect to complexity. First, there is a tendency for people to equate size with complexity, whereas most cost models (including COCOMO) define complexity as an inherent attribute of the code which is independent of size. Second, there is a tendency for people to rate the complexity of a component as the complexity of the hardest part in it. Large components often have a great deal of simple housekeeping code included, which easily becomes overestimated with respect to its complexity. Both of these tendencies should be checked as part of the review and iteration of the cost estimate.

---

* The need to investigate differences between estimates is the main reason why it is so important for an algorithmic cost model to be constructive. A constructive estimate makes it easy and clear to reviewers why the model gave the results it did. Otherwise, there is no way to compare algorithmic cost model estimates with other types of estimates. Constructiveness has been a paramount objective for the COCOMO model.

* Often, the estimates follow “Pareto’s Law,” which in this context is expressed as “80% of the cost is contained in 20% of the components.”
Some Useful Review Questions

In reviewing a software cost estimate upon which a development budget will be based, it is important that we obtain a clear understanding of the estimate's soundness and degree of optimism. With respect to the estimation accuracy versus phase graph in Fig. 21-1, if we accept an estimate as a relatively sound requirements specification level estimate (within a factor of 1.5) when it is actually only an early feasibility estimate (within a factor of four), we are likely to encounter some unpleasant surprises and budget renegotiations later on. We are likely to encounter similar surprises if we accept an optimistic estimate (near the lower boundary curve in Fig. 21-1) as our budgetary estimate for the project.

Here are some review questions whose answers help us to judge the relative soundness of a software cost estimate (that is, how far to the right or left in Fig. 21-1).

1. How would the estimated cost change if
   (a) there was a big change in the (human-machine, data communications, inventory system, etc.) interface?
   (b) we had to supply a more extensive (data base query, maintenance and diagnostic, trend analysis, etc.) capability?
   (c) our workload estimates were off by a factor of two?
   (d) we had to do the job on a (smaller, different, distributed, etc.) computer configuration?

2. Suppose our budget were 20% (less, greater). How would this affect the product?

3. How does the cost of the (user interface, data base management, process control, etc.) subsystem break down by component?

If the answers to such questions are vague and general ("Oh, that wouldn't impact the cost much") or are unaccompanied by any rationale ("Adding that capability would cost you $200,000"), then we need to treat the estimates as very preliminary, early figures which may change a great deal as we proceed to define the software in more detail.

Here are some review questions whose answers help us judge the relative optimism of a software cost estimate (that is, how far up or down in Fig. 21-1).

1. How do the following compare with our past experience
   (a) sizing of subsystems?
   (b) cost driver attribute ratings:
   (c) cost per instruction?
   (d) assumptions about adaptation, vendor software, key personnel, etc.?

2. Has a cost–risk analysis been performed? If so, what are the high risk items, and how well are they covered?

3. Suppose we had enough money to really do this job right. How much would we need?
21.7 STEP 7: FOLLOWUP

Once a software project is started, it is essential to gather data on its actual costs and progress and compare these to the estimates. Here are a number of reasons why this is essential.

1. Software estimating inputs are imperfect (sizing estimates, cost driver ratings). If a project finds a difference between estimated and actual costs which can be explained by improved knowledge of the cost drivers, it is important for the project manager to update the cost estimate with the new knowledge, providing a more realistic basis for continuing to manage the project.
2. Software estimating techniques are imperfect. For long-range improvements, we need to compare estimates to actuals and use the results to improve the estimating techniques.
3. Some projects do not exactly fit the estimating model (for example, in time-phasing the costs of incremental development). Both near-term project-management feedback and long-term model-improvement feedback of any estimates-versus-actuals differences are important here.
4. Software projects tend to be volatile: components are added, split up, rescoped, or combined in unforeseeable ways as the project progresses. Again, the project manager needs to identify these changes and generate a more realistic update of the estimated upcoming costs.
5. Software is an evolving field. Estimating techniques are all calibrated on previous projects, which may not have featured structured programming, automated aids, specification languages, microprocessors, or distributed data processing. For both short- and long-term purposes, it is important to sense differences due to these trends and incorporate them into improved project estimates and improved estimating techniques.

A detailed treatment of software project planning and control followup techniques is given in Chapter 32. Techniques for ongoing project data collection and analysis are given in Appendix A. A simple followup technique, the cost-schedule-milestone chart, is illustrated below.

The Cost-Schedule-Milestone Chart

A simple but highly useful technique for comparing estimates to actuals is the cost-schedule-milestone chart. An example is shown in Fig. 21-5, using a basic, embedded-mode, 32-KDSI software project as an example. It shows the estimated number of months and man-months required to achieve four major project milestones.

- **SRR** Software requirements review (4.5 months, 18 MM)
- **PDR** Product design review (9.3 months, 60 MM)
- **UTC** Unit test completion (15.9 months, 184 MM)
- **SAR** Software acceptance review (18.5 months, 248 MM)
Starting with these estimates, the project manager can plot the actual cost and schedule associated with the achievement of these milestones. If there is a significant difference between the estimates and the actuals, there is a basis for investigating and taking corrective action.

For example, three possible points $A$, $B$, and $C$ are shown in Fig. 21-5, representing projects whose achievement of a product design review (PDR) occurs at significantly different cost-schedule points than the 9.3 months and 60 man-months estimated.

Project $A$ has taken about the right amount of effort, but has reached PDR considerably ahead of schedule. Its personnel loading is considerably higher than usual for such projects. This may be because the product is easy to segment into a number of pieces which can be usefully worked by a larger number of analysts than the usual product. In this case, the project manager can revise the estimates by simply advancing the project schedule. On the other hand, the project may have succumbed to the temptation to bring a lot of people on-board early, and the PDR milestone may not be as thoroughly satisfied as it should have been.* This is one possibility that the project manager needs to investigate before proceeding much further.

Project $B$ has taken about the right amount of time to reach PDR, but has not used the estimated amount of labor. This may be because the job has been redefined in scope to be a simpler job (less work to define, but more time consumed in renegotiation), in which case the project manager can re-estimate a lower cost to complete. Or, it may mean that the project has experienced problems in staffing up, and has not fully achieved PDR, which again the project manager needs to investigate as a possibility.

* See Table 4-1 for a summary of what a software project needs to complete by PDR.
Project C has taken considerably more schedule and labor to achieve PDR. This may be for any of the five reasons discussed at the beginning of this section. Here again, the project manager needs to find out which of these reasons have caused the discrepancy before an appropriate course can be charted for the remainder of the project.

21.8 QUESTIONS

21.1. Consider a project which cost $4 million to develop. What was the range of uncertainty of its estimated cost at SRR (requirements specification available)? At PDR (product design specification available)?

21.2. What are the three major guidelines for establishing the objectives of a software cost-estimation activity? What are the seven basic steps in software cost estimation?

21.3. Suppose we have defined a $2 million electric power distribution software component to the product design level. For an overall project cost proposal, we need an estimate of the diagnostic and support software, which is roughly in the $500,000 cost range. Based on Fig. 21-1, to what level do we need to define the diagnostic and support software to achieve a balanced overall estimate of the total product cost?

21.4. Prepare a miniproject plan for estimating the costs of developing a compiler, database management system, or some other software product with which you are familiar.

21.5. Suppose you were the project manager on the project illustrated in Fig. 21-5, and you reach your PDR milestone with an expended schedule of 6 months and an expended effort of 30 MM. What possibilities should you investigate to explain the difference between the actual and estimated schedule and budget, and what steps should you take as a project manager if a given possibility was true?

21.6. Work Question 21.5 given that the actual schedule to PDR was 9.5 months and the actual effort 110 MM.

21.7. (Research Project) Test the hypothesis that the distribution of source code by function in Fig. 21-3 is the same for all software products, by compiling comparable distributions for other products. If the hypothesis is false, prepare and test an improved hypothesis.

21.8. (Research Project) Develop and investigate improved methods of software product sizing. Note: Such investigations are more likely to be productive if restricted to a particular application domain: for example, compilers, payroll systems, software tools (refer to Table 27-9).