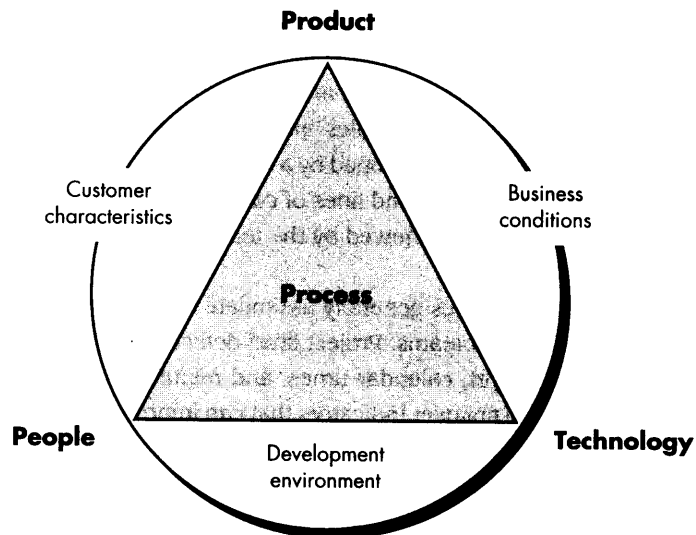


**FIGURE 22.1**

Determinants for software quality and organizational effectiveness (adapted from [PAU94])



In addition, the process triangle exists within a circle of environmental conditions that include the development environment (e.g., CASE tools), business conditions (e.g., deadlines, business rules), and customer characteristics (e.g., ease of communication and collaboration).

We measure the efficacy of a software process indirectly. That is, we derive a set of metrics based on the outcomes that can be derived from the process. Outcomes include measures of errors uncovered before release of the software, defects delivered to and reported by end-users, work products delivered (productivity), human effort expended, calendar time expended, schedule conformance, and other measures. We also derive process metrics by measuring the characteristics of specific software engineering tasks. For example, we might measure the effort and time spent performing the generic software engineering activities described in Chapter 2.

### KEY POINT

The skill and motivation of the software people doing the work are the most important factors that influence software quality.

"Software metrics let you know when to laugh and when to cry."

Tom Gilb

### What is the difference between private and public uses for software metrics?

Grady [GRA92] argues that there are "private and public" uses for different types of process data. Because it is natural that individual software engineers might be sensitive to the use of metrics collected on an individual basis, these data should be private to the individual and serve as an indicator for the individual only. Examples of *private metrics* include defect rates by individual, defect rates by software component, and errors found during development.

The "private process data" philosophy conforms well with the personal software process approach (Chapter 2) proposed by Humphrey [HUM95]. Humphrey recognizes

that software process improvement can and should begin at the individual level. Private process data can serve as an important driver as the individual software engineer works to improve.

Some process metrics are private to the software project team but public to all team members. Examples include defects reported for major software functions (that have been developed by a number of practitioners), errors found during formal technical reviews, and lines of code or function points per component or function.<sup>1</sup> These data are reviewed by the team to uncover indicators that can improve team performance.

Public metrics generally assimilate information that originally was private to individuals and teams. Project level defect rates (absolutely not attributed to an individual), effort, calendar times, and related data are collected and evaluated in an attempt to uncover indicators that can improve organizational process performance.

Software process metrics can provide significant benefit as an organization works to improve its overall level of process maturity. However, like all metrics, these can be misused, creating more problems than they solve. Grady [GRA92] suggests a “software metrics etiquette” that is appropriate for both managers and practitioners as they institute a process metrics program:

**What guidelines should be applied when we collect software metrics?**

- Use common sense and organizational sensitivity when interpreting metrics data.
- Provide regular feedback to the individuals and teams who collect measures and metrics.
- Don't use metrics to appraise individuals.
- Work with practitioners and teams to set clear goals and metrics that will be used to achieve them.
- Never use metrics to threaten individuals or teams.
- Metrics data that indicate a problem area should not be considered “negative.” These data are merely an indicator for process improvement.
- Don't obsess on a single metric to the exclusion of other important metrics.

As an organization becomes more comfortable with the collection and use of process metrics, the derivation of simple indicators gives way to a more rigorous approach called *statistical software process improvement* (SSPI). In essence, SSPI uses software failure analysis to collect information about all errors and defects<sup>2</sup> encountered as an application, system, or product is developed and used.

<sup>1</sup> Lines of code and function point metrics are discussed in Sections 22.2.1 and 22.2.2.

<sup>2</sup> In this book, an *error* is defined as some flaw in a software engineering work product that is uncovered before the software is delivered to the end-user. A *defect* is a flaw that is uncovered *after* delivery to the end-user. It should be noted that others do not make this distinction. Further discussion is presented in Chapter 26.

### 22.1.2 Project Metrics

Unlike software process metrics that are used for strategic purposes, software project metrics are tactical. That is, project metrics and the indicators derived from them are used by a project manager and a software team to adapt project workflow and technical activities.

The first application of project metrics on most software projects occurs during estimation. Metrics collected from past projects are used as a basis from which effort and time estimates are made for current software work. As a project proceeds, measures of effort and calendar time expended are compared to original estimates (and the project schedule). The project manager uses these data to monitor and control progress.

As technical work commences, other project metrics begin to have significance. Production rates represented in terms of models created, review hours, function points, and delivered source lines are measured. In addition, errors uncovered during each software engineering task are tracked. As the software evolves from requirements into design, technical metrics (Chapter 15) are collected to assess design quality and to provide indicators that will influence the approach taken to code generation and testing.

The intent of project metrics is twofold. First, these metrics are used to minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks. Second, project metrics are used to assess product quality on an ongoing basis and, when necessary, modify the technical approach to improve quality.

As quality improves, defects are minimized, and as the defect count goes down, the amount of rework required during the project is also reduced. This leads to a reduction in overall project cost.

**How should we use metrics during the project itself?**

#### SAFEHOME



#### Establishing a Metrics Approach

The scene: Doug Miller's office as the SafeHome software project is about to begin.

**Doug Miller:** Doug Miller (manager of the SafeHome project) and Vinod Raman and Jamie (members of the product software engineering team).

**Doug:**

As we start work on this project, I'd like you to collect a set of simple metrics. To start, let's define your goals.

**Vinod:** We've never done that before,

**Jamie (interrupting):** And based on the metrics management has been talking about, we'll waste a lot of time. What good are metrics anyway?

**Doug (raising his hand to stop the conversation):** Slow down and take a breath, guys. The fact that we've never done it before is all the more reason to start now, and the metrics work I'm talking about shouldn't take much time at all . . . in fact, it just might save us time.

**Vinod:** How?

**Doug:** Look, we're going to be doing a lot more in-house software engineering as our customers move

intelligent, become Web enabled, all that . . . and we need to understand the process we use to build software and improve it so we can build software better. The only way to do that is to measure.

**Jamie:** But we're under time pressure, Doug. I'm not in favor of error paper pushing . . . we need the time to do our best, not collect data.

**Doug:** Jamie, an engineer's work involves collecting data, evaluating it, and using the results to improve the product and the process. Am I wrong?

**Jamie:** No, but . . .

**Doug:** What if we hold the number of measures we collect to no more than five or six and focus on quality?

**Jamie:** No one can argue against high quality . . .

**Jamie:** True . . . but, I don't know, I still think this isn't necessary.

**Doug:** I'm going to ask you to humor me on this one. How much do you guys know about software metrics?

**Jamie (looking at Vinod):** Not much.

**Doug:** Here are some Web refs . . . spend a few hours getting up to speed.

**Jamie (smiling):** I thought you said this wouldn't take any time.

**Doug:** Time you spend learning is never wasted . . . go do it and then we'll establish some goals, ask a few questions, and define the metrics we need to collect.

## 22.2 SOFTWARE MEASUREMENT

In Chapter 15, we noted that software measurement can be categorized in two ways: (1) *direct measures* of the software process (e.g., cost and effort applied) and product (e.g., lines of code (LOC) produced, execution speed, and defects reported over some set period of time), and (2) *indirect measures* of the product that include functionality, quality, complexity, efficiency, reliability, maintainability, and many other “-abilities” discussed in Chapter 15.

**“Not everything that can be counted counts, and not everything that counts can be counted.”**

**Albert Einstein**

Project metrics can be consolidated to create process metrics that are public to the software organization as a whole. But how does an organization combine metrics that come from different individuals or projects?

To illustrate, we consider a simple example. Individuals on two different project teams record and categorize all errors that they find during the software process. Individual measures are then combined to develop team measures. Team A found 342 errors during the software process prior to release. Team B found 184 errors. All other things being equal, which team is more effective in uncovering errors throughout the process? Because we do not know the size or complexity of the projects, we cannot answer this question. However, if the measures are normalized, it is possible to create software metrics that enable comparison to broader organizational averages. Both size- and function-oriented metrics are normalized in this manner.



*Because many factors affect software work, don't use metrics to compare individuals or teams.*



opponents argue that LOC measures are programming language dependent, that when productivity is considered, they penalize well-designed but shorter programs, that they cannot easily accommodate nonprocedural languages, and that their use in estimation requires a level of detail that may be difficult to achieve (i.e., the planner must estimate the LOC to be produced long before analysis and design have been completed).

### 22.2.2 Function-Oriented Metrics

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. The most widely used function-oriented metric is the *function point* (FP). Computation of the function point is based on characteristics of the software's information domain and complexity. The mechanics of FP computation has been discussed in Chapter 15.<sup>3</sup>

The function point, like the LOC measure, is controversial. Proponents claim that FP is programming language independent, making it ideal for applications using conventional and nonprocedural languages, and that it is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach. Opponents claim that the method requires some "sleight of hand" in that computation is based on subjective rather than objective data, that counts of the information domain (and other dimensions) can be difficult to collect after the fact, and that FP has no direct physical meaning—it's just a number.

### 22.2.3 Reconciling LOC and FP Metrics

The relationship between lines of code and function points depends upon the programming language that is used to implement the software and the quality of the design. A number of studies have attempted to relate FP and LOC measures. To quote Albrecht and Gaffney [ALB83]:

The thesis of this work is that the amount of function to be provided by the application (program) can be estimated from the itemization of the major components<sup>4</sup> of data to be used or provided by it. Furthermore, this estimate of function should be correlated to both the amount of LOC to be developed and the development effort needed.

The following table<sup>5</sup> [QSM02] provides rough estimates of the average number of lines of code required to build one function point in various programming languages:

- 
- 3 See Section 15.3.1 for a detailed discussion of FP computation.
  - 4 It is important to note that "the itemization of major components" can be interpreted in a variety of ways. Software engineers who work in an object-oriented development environment use the number of classes or objects as the dominant size metric. A maintenance organization might view project size in terms of the number of engineering change orders (Chapter 27). An information systems organization might view the number of business processes affected by an application.
  - 5 Used with permission of Quantitative Software Management ([www.qsm.com](http://www.qsm.com)), copyright 2002.

**Programming  
Language****LOC per Function point**

	<b>Avg.</b>	<b>Median</b>	<b>Low</b>	<b>High</b>
Access	35	38	15	47
Ada	154	—	104	205
APS	86	83	20	184
ASP 69	62	—	32	127
Assembler	337	315	91	694
C	162	109	33	704
C++	66	53	29	178
Clipper	38	39	27	70
COBOL	77	77	14	400
Cool:Gen/IEF	38	31	10	180
Culprit	51	—	—	—
DBase iV	52	—	—	—
Easytrieve+	33	34	25	41
Excel47	46	—	31	63
Focus	43	42	32	56
FORTRAN	—	—	—	—
FoxPro	32	35	25	35
Ideal	66	52	34	203
IEF/Cool:Gen	38	31	10	180
Informix	42	31	24	57
Java	63	53	77	—
JavaScript	58	63	42	75
JCL	91	123	26	150
JSP	59	—	—	—
Lotus Notes	21	22	15	25
Mantis	71	27	22	250
Mapper	118	81	16	245
Natural	60	52	22	141
Oracle	30	35	4	217
PeopleSoft	33	32	30	40
Perl	60	—	—	—
PL/1	78	67	22	263
Powerbuilder	32	31	11	105
REXX	67	—	—	—
RPG II/III	61	49	24	155
SAS	40	41	33	49
Smalltalk	26	19	10	55
SQL	40	37	7	110
VBScript36	34	27	50	—
Visual Basic	47	42	16	158

A review of these data indicates that one LOC of C++ provides approximately 2.4 times the “functionality” (on average) as one LOC of C. Furthermore, one LOC of a Smalltalk provides at least four times the functionality of a LOC for a conventional programming language such as Ada, COBOL, or C. Using the information contained in the table, it is possible to “backfire” [JON98] existing software to estimate the number of function points, once the total number of programming language statements are known.

Function points and LOC-based metrics have been found to be relatively accurate predictors of software development effort and cost. However, to use LOC and FP for estimation (Chapter 23), a historical baseline of information must be established.

Within the context of process and project metrics, we are concerned primarily with productivity and quality—measures of software development “output” as a function of effort and time applied and measures of the “fitness for use” of the work products that are produced. For process improvement and project planning purposes, our interest is historical. What was software development productivity on past projects? What was the quality of the software that was produced? How can past productivity and quality data be extrapolated to the present? How can it help us improve the process and plan new projects more accurately?

#### 22.2.4 Object-Oriented Metrics

Conventional software project metrics (LOC or FP) can be used to estimate object-oriented software projects. However, these metrics do not provide enough granularity for the schedule and effort adjustments that are required as we iterate through an evolutionary or incremental process. Lorenz and Kidd [LOR94] suggest the following set of metrics for OO projects:



*It is not uncommon for multiple scenario scripts to mention the same functionality or data objects. Therefore, be careful when using script counts.*

**Number of scenario scripts.** A scenario script (analogous to use-cases discussed throughout Parts 2 and 3 of this book) is a detailed sequence of steps that describes the interaction between the user and the application. The number of scenario scripts is directly correlated to the size of the application and to the number of test cases that must be developed to exercise the system once it is constructed.

**Number of key classes.** *Key classes* are the “highly independent components” [LOR94] that are defined early in object-oriented analysis (Chapter 8).<sup>6</sup> Because key classes are central to the problem domain, the number of such classes is an indication of the amount of effort required to develop the software and also an indication of the potential amount of reuse to be applied during system development.



*Classes can vary in size and complexity. Therefore, it's worth considering classifying class counts by size and complexity.*

**Number of support classes.** *Support classes* are required to implement the system but are not immediately related to the problem domain. Examples might be UI classes, database access and manipulation classes, and computation classes. In addition, support classes can be developed for each of the key classes. The number of support classes is an indication of the amount of effort required to develop the software and an indication of the potential amount of reuse to be applied during system development.

**Average number of support classes per key class.** In general, key classes are known early in the project. Support classes are defined throughout. If the average number of support classes per key class were known for a given problem domain, estimating (based on total number of classes) would be much simplified. Lorenz and

<sup>6</sup> Key classes were referred to as *analysis classes* in Part 2 of this book.



Kidd suggest that applications with a GUI have between two and three times the number of support classes as key classes. Non-GUI applications have between one and two times the number of support classes as key classes.

**Number of subsystems.** A *subsystem* is an aggregation of classes that support a function that is visible to the end-user of a system. Once subsystems are identified, it is easier to lay out a reasonable schedule in which work on subsystems is partitioned among project staff.

To be used effectively in an object-oriented software engineering environment, metrics similar to those noted above must be collected along with project measures such as effort expended, errors and defects uncovered, and models or documentation pages produced. As the database grows (after a number of projects have been completed), relationships between object-oriented measures and project measures will provide metrics that can aid in project estimation.

### 22.2.5 Use-Case Oriented Metrics

It would seem reasonable to apply the use-case<sup>7</sup> as a normalization measure similar to LOC or FP. Like FP, the use-case is defined early in the software process, allowing it to be used for estimation before significant modeling and construction activities are initiated. Use-cases describe (indirectly, at least) user-visible functions and features that are basic requirements for a system. The use-case is independent of programming language. In addition, the number of use-cases is directly proportional to the size of the application in LOC and to the number of test cases that will have to be designed to fully exercise the application.

Because use-cases can be created at vastly different levels of abstraction, there is no standard size for a use-case. Without a standard measure of what a use-case is, its application as a normalization measure (e.g., effort expended per use-case) is suspect. Although a number of researchers (e.g., [SMI99]), have attempted to derive use-case metrics, much work remains to be done.

### 22.2.6 Web Engineering Project Metrics

The objective of all Web engineering projects (Part 3 of this book) is to build a Web application (WebApp) that delivers a combination of content and functionality to the end-user. Measures and metrics used for traditional software engineering projects are difficult to translate directly to WebApps. Yet, a Web engineering organization should develop a database that allows it to assess its internal productivity and quality over a number of projects. Among the measures that can be collected are:

**Number of static Web pages.** Web pages with static content (i.e., the end-user has no control over the content displayed on the page) are the most common of all WebApp features. These pages represent low relative complexity and generally

---

<sup>7</sup> Use-cases are discussed throughout Parts 2 and 3 of this book.

require less effort to construct than dynamic pages. This measure provides an indication of the overall size of the application and the effort required to develop it.

**Number of dynamic Web pages.** Web pages with dynamic content (i.e., end-user actions result in customized content displayed on the page) are essential in all e-commerce applications, search engines, financial applications, and many other WebApp categories. These pages represent higher relative complexity and require more effort to construct than static pages. This measure provides an indication of the overall size of the application and the effort required to develop it.

**Number of internal page links.** Internal page links are pointers that provide a hyperlink to some other Web page within the WebApp. This measure provides an indication of the degree of architectural coupling within the WebApp. As the number of page links increases, the effort expended on navigational design and construction also increases.

**Number of persistent data objects.** One or more persistent data objects (e.g., a database or data file) may be accessed by a WebApp. As the number of persistent data objects grows, the complexity of the WebApp also grows, and effort to implement it increases proportionally.

**Number of external systems interfaced.** WebApps must often interface with “backroom” business applications. As the requirement for interfacing grows, system complexity and development effort also increase.

**Number of static content objects.** Static content objects encompass static text-based, graphical, video, animation, and audio information that are incorporated within the WebApp. Multiple content objects may appear on a single Web page.

**Number of dynamic content objects.** Dynamic content objects are generated based on end-user actions and encompass internally generated text-based, graphical, video, animation, and audio information that are incorporated within the WebApp. Multiple content objects may appear on a single Web page.

**Number of executable functions.** An executable function (e.g., a script or applet) provides some computational service to the end-user. As the number of executable functions increases, modeling and construction effort also increase.

Each of the measures noted above can be determined at a relatively early stage of the Web engineering process.

For example, we can define a metric that reflects the degree of end-user customization that is required for the WebApp and correlate it to the effort expended on the WebE project and/or the errors uncovered as reviews and testing are conducted. To accomplish this, we define

$N_{sp}$  = number of static Web pages

$N_{dp}$  = number of dynamic Web pages

Then,

$$\text{Customization index, } C = N_{dp} / (N_{dp} + N_{sp})$$

The value of  $C$  ranges from 0 to 1. As  $C$  grows larger the level of WebApp customization becomes a significant technical issue.

Similar Web application metrics can be computed and correlated with project measures such as effort expended, errors and defects uncovered, and models or documentation pages produced. As the database grows (after a number of projects have been completed), relationships between the WebApp measures and project measures will provide indicators that can aid in project estimation.

## SOFTWARE TOOLS



### Project and Process Metrics

**Objective:** To assist in the definition, collection, evaluation, and reporting of software measures and metrics.

**Mechanics:** Each tool varies in its application, but all provide mechanisms for collecting and evaluating data that lead to the computation of software metrics.

#### Representative Tools<sup>8</sup>

*Function Point WORKBENCH*, developed by Charismatek ([www.charismatek.com.au](http://www.charismatek.com.au)), offers a wide array of FP-oriented metrics.

*MetricCenter*, developed by Distributive Software ([www.distributive.com](http://www.distributive.com)), supports automated data collection, analysis, chart formatting, report generation, and other measurement tasks.

*PSM Insight*, developed by Practical Software and Systems Measurement ([www.psmc.com](http://www.psmc.com)), assists in the creation and subsequent analysis of a project measurement database.

*SLIM tool set*, developed by QSM ([www.qsm.com](http://www.qsm.com)), provides a comprehensive set of metrics and estimation tools.

*SPR tool set*, developed by Software Productivity Research ([www.spr.com](http://www.spr.com)), offers a comprehensive collection of FP-oriented tools.

*TychoMetrics*, developed by Predicate Logic, Inc. ([www.predicate.com](http://www.predicate.com)), is a tool suite for management metrics collection and reporting.

## 22.3 METRICS FOR SOFTWARE QUALITY

The overriding goal of software engineering is to produce a high-quality system, application, or product within a timeframe that satisfies a market need. To achieve this goal, software engineers must apply effective methods coupled with modern tools within the context of a mature software process. In addition, a good software engineer (and good software engineering managers) must measure if high quality is to be realized.

Private metrics collected by individual software engineers are assimilated to provide project-level results. Although many quality measures can be collected, the

<sup>8</sup> Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

primary thrust at the project level is to measure errors and defects. Metrics derived from these measures provide an indication of the effectiveness of individual and group software quality assurance and control activities.

Metrics such as work product (e.g., requirements or design) errors per function point, errors uncovered per review hour, and errors uncovered per testing hour provide insight into the efficacy of each of the activities implied by the metric. Error data can also be used to compute the *defect removal efficiency* (DRE) for each process framework activity. DRE is discussed in Section 22.3.2.

### 22.3.1 Measuring Quality

Although there are many measures of software quality,<sup>9</sup> correctness, maintainability, integrity, and usability provide useful indicators for the project team. Gilb [GIL88] suggests definitions and measures for each.

#### WebRef

An excellent source of information on software quality and related topics (including metrics) can be found at [www.qualityworld.com](http://www.qualityworld.com)

**Correctness.** A program must operate correctly or it provides little value to its users. Correctness is the degree to which the software performs its required function. The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements. When considering the overall quality of a software product, defects are those problems reported by a user of the program after the program has been released for general use. For quality assessment purposes, defects are counted over a standard period of time, typically one year.

**Maintainability.** Software maintenance accounts for more effort than any other software engineering activity. Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements. There is no way to measure maintainability directly; therefore, we must use indirect measures. A simple time-oriented metric is *mean-time-to-change* (MTTC), the time it takes to analyze the change request, design an appropriate modification, implement the change, test it, and distribute the change to all users. On average, programs that are maintainable will have a lower MTTC (for equivalent types of changes) than programs that are not maintainable.

**Integrity.** Software integrity has become increasingly important in the age of cyber-terrorists and hackers. This attribute measures a system's ability to withstand attacks (both accidental and intentional) to its security. Attacks can be made on all three components of software: programs, data, and documents.

To measure integrity, two additional attributes must be defined: threat and security. *Threat* is the probability (which can be estimated or derived from empirical evidence) that an attack of a specific type will occur within a given time. *Security* is the probability (which can be estimated or derived from empirical evidence) that

<sup>9</sup> A detailed discussion of the factors that influence software quality and the metrics that can be used to assess software quality has been presented in Chapter 15.

the attack of a specific type will be repelled. The integrity of a system can then be defined as:

$$\text{integrity} = \Sigma [1 - (\text{threat} \times (1 - \text{security}))]$$

For example, if threat (the probability that an attack will occur) is 0.25 and security (the likelihood of repelling an attack) is 0.95, the integrity of the system is 0.99 (very high). If, on the other hand, the threat probability is 0.50 and the likelihood of repelling an attack is only 0.25, the integrity of the system is 0.63 (unacceptably low).

**Usability.** If a program is not easy to use, it is often doomed to failure, even if the functions that it performs are valuable. Usability is an attempt to quantify ease-of-use and can be measured in terms of characteristics presented in Chapter 12.

The four factors just described are only a sampling of those that have been proposed as measures for software quality. Chapter 15 considers this topic in additional detail.

### 22.3.2 Defect Removal Efficiency

A quality metric that provides benefits at both the project and process level is *defect removal efficiency* (DRE). In essence, DRE is a measure of the filtering ability of quality assurance and control activities as they are applied throughout all process framework activities.

When considered for a project as a whole, DRE is defined in the following manner:

$$\text{DRE} = E / (E + D)$$

where  $E$  is the number of errors found before delivery of the software to the end-user, and  $D$  is the number of defects found after delivery.

The ideal value for DRE is 1. That is, no defects are found in the software. Realistically,  $D$  will be greater than 0, but the value of DRE can still approach 1. As  $E$  increases (for a given value of  $D$ ), the overall value of DRE begins to approach 1. In fact, as  $E$  increases, it is likely that the final value of  $D$  will decrease (errors are filtered out before they become defects). If used as a metric that provides an indicator of the filtering ability of quality control and assurance activities, DRE encourages a software project team to institute techniques for finding as many errors as possible before delivery.

DRE can also be used within the project to assess a team's ability to find errors before they are passed to the next framework activity or software engineering task. For example, the requirements analysis task produces an analysis model that can be reviewed to find and correct errors. Those errors that are not found during the review of the analysis model are passed on to design (where they may or may not be found). When used in this context, we redefine DRE as

$$\text{DRE}_i = E_i / (E_i + E_{i+1})$$

where  $E_i$  is the number of errors found during software engineering activity  $i$  and  $E_{i+1}$  is the number of errors found during software engineering activity  $i + 1$  that are traceable to errors that were not discovered in software engineering activity  $i$ .



*If DRE is low as you move through analysis and design, spend some time improving the way you conduct formal technical reviews.*

A quality objective for a software team (or an individual software engineer) is to achieve DRE, that approaches 1. That is, errors should be filtered out before they are passed on to the next activity.

## SAFEHOME



### Establishing a Metrics Approach

**The scene:** Doug Miller's office two days after initial meeting on software metrics.

**The players:** Doug Miller (manager of the SafeHome software engineering team) and Vinod Raman and Jamie Jones, members of the product software engineering team.

**THE CONVERSATION**

**Doug:** You both had a chance to learn a little about process and project metrics?

**Vinod and Jamie:** [Both nod]

**Doug:** It's always a good idea to establish goals when you collect any metrics. What are yours?

**Vinod:** Our metric should focus on quality. In fact, our overall goal is to keep the number of errors we pass on from one software engineering activity to the next to an absolute minimum.

**Doug:** And be very sure you keep the number of defects released with the product to as close to zero as possible.

**Vinod (smiling):** Of course.

**Doug:** Use DRE as a metric, and I think we can use it for the entire project. Also, we can use it as we move

from one framework activity to the next. It'll encourage us to find errors at each step.

**Vinod:** I'd also like to collect the number of hours we spend on reviews.

**Jamie:** And the overall effort we spend on each software engineering task.

**Doug:** You can compute a review-to-development ratio . . . might be interesting.

**Jamie:** I'd like to track some use-case data as well. Like the amount of effort required to develop a use-case, the amount of effort required to build software to implement a use-case, and . . .

**Doug (smiling):** I thought we were going to keep this simple.

**Vinod:** We should, but once you get into this metrics stuff, there's a lot of interesting things to look at.

**Doug:** I agree, but let's walk before we run, and stick to our goal. Limit data to be collected to five or six items, and we're ready to go.

## 22.4 INTEGRATING METRICS WITHIN THE SOFTWARE PROCESS

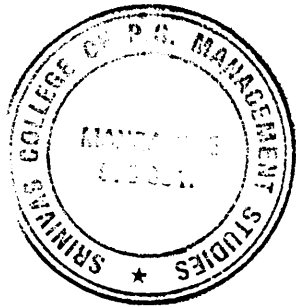
The majority of software developers still do not measure, and sadly, most have little desire to begin. As we noted earlier in this chapter, the problem is cultural. Attempting to collect measures where none had been collected in the past often precipitates resistance. "Why do we need to do this?" asks a harried project manager. "I don't see the point," complains an overworked practitioner.

In this section, we consider some arguments for software metrics and present an approach for instituting a metrics collection program within a software engineering organization. But before we begin, some words of wisdom are suggested by Grady and Caswell [GRA87]:

Some of the things we describe here will sound quite easy. Realistically, though, establishing a successful company-wide software metrics program is hard work. When we say

that you must wait at least three years before broad organizational trends are available, you get some idea of the scope of such an effort.

The caveat suggested by the authors is well worth heeding, but the benefits of measurement are so compelling that the hard work is worth it.



### 22.4.1 Arguments for Software Metrics

Why is it so important to measure the process of software engineering and the product (software) that it produces? The answer is relatively obvious. If we do not measure, there is no real way of determining whether we are improving. And if we are not improving, we are lost.

By requesting and evaluating productivity and quality measures, a software team (and their management) can establish meaningful goals for improvement of the software process. In Chapter 1 we noted that software is a strategic business issue for many companies. If the process through which it is developed can be improved, a direct impact on the bottom line can result. But to establish goals for improvement, the current status of software development must be understood. Hence, measurement is used to establish a process baseline from which improvements can be assessed.

**"We manage things by the numbers in many aspects of our lives . . . These numbers give us insight and help steer our actions."**

**Michael Mah and Larry Potam**

The day-to-day rigors of software project work leave little time for strategic thinking. Software project managers are concerned with more mundane (but equally important) issues: developing meaningful project estimates, producing higher-quality systems, getting product out the door on time. By using measurement to establish a project baseline, each of these issues becomes more manageable. We have already noted that the baseline serves as a basis for estimation. Additionally, the collection of quality metrics enables an organization to "tune" its software process to remove the "vital few" causes of defects that have the greatest impact on software development.<sup>10</sup>

### 22.4.2 Establishing a Baseline

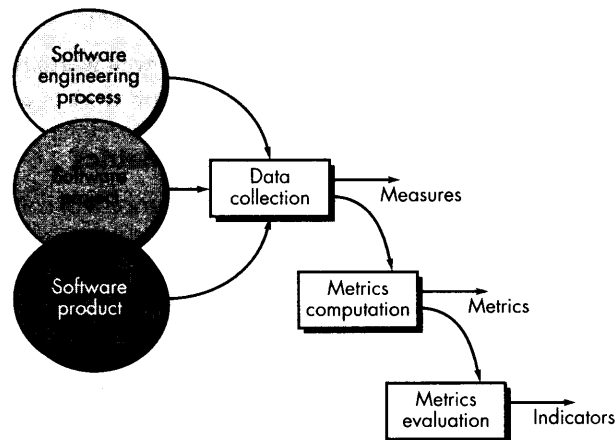
By establishing a metrics baseline, benefits can be obtained at the process, project, and product (technical) levels. Yet the information that is collected need not be fundamentally different. The same metrics can serve many masters. The metrics baseline consists of data collected from past software development projects and can be as simple as the table presented in Figure 22.2 or as complex as a comprehensive database containing dozens of project measures and the metrics derived from them.

**? What is a metrics baseline, and what benefit does it provide to a software engineer?**

<sup>10</sup> These ideas have been formalized into an approach called *statistical software quality assurance* and are discussed in detail in Chapter 26.

**FIGURE 22.3**

**Software metrics collection process**



To be an effective aid in process improvement and/or cost and effort estimation, baseline data must have the following attributes: (1) data must be reasonably accurate—“guesstimates” about past projects are to be avoided; (2) data should be collected for as many projects as possible; (3) measures must be consistent, for example, a line of code must be interpreted consistently across all projects for which data are collected; (4) applications should be similar to work that is to be estimated—it makes little sense to use a baseline for batch information systems work to estimate a real-time, embedded application.

### 22.4.3 Metrics Collection, Computation, and Evaluation

The process for establishing a metrics baseline is illustrated in Figure 22.3. Ideally, data needed to establish a baseline has been collected in an on-going manner. Sadly, this is rarely the case. Therefore, data collection requires a historical investigation of past projects to reconstruct required data. Once measures have been collected (unquestionably the most difficult step), metrics computation is possible. Depending on the breadth of measures collected, metrics can span a broad range of application-oriented metrics (e.g., LOC, FP, object-oriented, WebApp) as well as other quality- and project-oriented metrics. Finally, metrics should be evaluated and applied during estimation, technical work, project control, and process improvement. Metrics evaluation focuses on the underlying reasons for the results obtained and produces a set of indicators that guide the project or process.

#### **KEY POINT**

Baseline metrics data should be collected from a large representative sampling of past software projects.

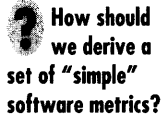
## 22.5 METRICS FOR SMALL ORGANIZATIONS

The vast majority of software development organizations have fewer than 20 software people. It is unreasonable, and in most cases unrealistic, to expect that such organizations will develop comprehensive software metrics programs. However, it





If you're just starting to collect metrics data, remember to keep it simple. If you bury yourself with data, your metrics effort will fail.



is reasonable to suggest that software organizations of all sizes measure and then use the resultant metrics to help improve their local software process and the quality and timeliness of the products they produce.

A common-sense approach to the implementation of any software process related activity is: keep it simple, customize to meet local needs, and be sure it adds value. In the paragraphs that follow, we examine how these guidelines relate to metrics for small shops.<sup>11</sup>

“Keep it simple” is a guideline that works reasonably well in many activities. But how do we derive a “simple” set of software metrics that still provides value, and how can we be sure that these simple metrics will meet the needs of a particular software organization? We begin by focusing not on measurement but rather on results. The software group is polled to define a single objective that requires improvement. For example, “reduce the time to evaluate and implement change requests.” A small organization might select the following set of easily collected measures:

- Time (hours or days) elapsed from the time a request is made until evaluation is complete,  $t_{queue}$ .
- Effort (person-hours) to perform the evaluation,  $W_{eval}$ .
- Time (hours or days) elapsed from completion of evaluation to assignment of change order to personnel,  $t_{eval}$ .
- Effort (person-hours) required to make the change,  $W_{change}$ .
- Time required (hours or days) to make the change,  $t_{change}$ .
- Errors uncovered during work to make the change,  $E_{change}$ .
- Defects uncovered after change is released to the customer base,  $D_{change}$ .

Once these measures have been collected for a number of change requests, it is possible to compute the average total elapsed time from change request to implementation of the change and the percentage of elapsed time absorbed by initial queuing, evaluation and change assignment, and change implementation. Similarly, the percentage of effort required for evaluation and implementation can be determined. These metrics can be assessed in the context of quality data,  $E_{change}$  and  $D_{change}$ . The percentages provide insight into where the change request process slows down and may lead to process improvement steps to reduce  $t_{queue}$ ,  $W_{eval}$ ,  $t_{eval}$ ,  $W_{change}$ , and/or  $E_{change}$ . In addition, the defect removal efficiency can be computed as

$$DRE = E_{change} / (E_{change} + D_{change})$$

DRE can be compared to elapsed time and total effort to determine the impact of quality assurance activities on the time and effort required to make a change.

<sup>11</sup> This discussion is equally relevant to software teams that have adopted an agile software development process (Chapter 4)

## 22.6 ESTABLISHING A SOFTWARE METRICS PROGRAM

The Software Engineering Institute has developed a comprehensive guidebook [PAR96] for establishing a “goal-driven” software metrics program. The guidebook suggests the following steps:

### WebRef

*A Guidebook for Goal  
Driven Software  
Measurement can be  
downloaded from:  
[www.sei.cmu.edu](http://www.sei.cmu.edu).*

1. Identify your business goals.
2. Identify what you want to know or learn.
3. Identify your subgoals.
4. Identify the entities and attributes related to your subgoals.
5. Formalize your measurement goals.
6. Identify quantifiable questions and the related indicators that you will use to help you achieve your measurement goals.
7. Identify the data elements that you will collect to construct the indicators that help answer your questions.
8. Define the measures to be used, and make these definitions operational.
9. Identify the actions that you will take to implement the measures.
10. Prepare a plan for implementing the measures.

A detailed discussion of these steps is best left to the SEI’s guidebook. However, a brief overview of key points is worthwhile.

### KEY POINT

The software metrics you choose should be driven by the business and technical goals you wish to accomplish.

Because software supports business functions, differentiates computer-based systems or products, or acts as a product in itself, goals defined for the business can almost always be traced downward to specific goals at the software engineering level. For example, consider a company that makes advanced home security systems which have substantial software content. Working as a team, software engineering and business managers can develop a list of prioritized business goals:

1. Improve our customers’ satisfaction with our products.
2. Make our products easier to use.
3. Reduce the time it takes us to get a new product to market.
4. Make support for our products easier.
5. Improve our overall profitability.

The software organization examines each business goal and asks: What activities do we manage or execute, and what do we want to improve within these activities? To answer these questions the SEI recommends the creation of an “entity-question list” in which all things (entities) within the software process that are managed or influenced by the software organization are noted. Examples of entities include development resources, work products, source code, test cases, change requests, software engineering tasks, and schedules. For each entity listed, software people

develop a set of questions that assess quantitative characteristics of the entity (e.g., size, cost, time to develop). The questions derived as a consequence of the creation of an entity-question list lead to the derivation of a set of subgoals that relate directly to the entities created and the activities performed as part of the software process.

Consider the fourth goal: "Make support for our products easier." The following list of questions might be derived for this goal [PAR96]:

- Do customer change requests contain the information we require to adequately evaluate the change and then implement it in a timely manner?
- How large is the change request backlog?
- Is our response time for fixing bugs acceptable, based on customer need?
- Is our change control process (Chapter 27) followed?
- Are high-priority changes implemented in a timely manner?

Based on these questions, the software organization can derive the following subgoal: *Improve the performance of the change management process*. The software process entities and attributes that are relevant to the subgoal are identified, and measurement goals associated with them are delineated.

The SEI [PAR96] provides detailed guidance for steps 6 through 10 of its goal-driven measurement approach. In essence, a process of stepwise refinement is applied in which goals are refined into questions that are further refined into entities and attributes that are then refined into metrics.



### **Establishing a Metrics Program**

The Software Productivity Center ([www.spc.ca](http://www.spc.ca)) suggests an eight-step approach for establishing a metrics program within a software organization that can be used as an alternative to the SEI approach described in Section 22.6. Their approach is summarized in this sidebar.

1. Understand the existing software process.
  - Framework activities (Chapter 2) are identified.
  - Input information for each activity is described.
  - Tasks associated with each activity are defined.
  - Quality assurance functions are noted.
  - Work products that are produced are listed.
2. Define the goals to be achieved by establishing a metrics program.
  - Examples: improve accuracy of estimation, improve product quality.
3. Identify metrics required to achieve goals.

### **INFO**

- Questions to be answered are defined; e.g., how many errors found in one framework activity can be traced to the preceding framework activity? Create measures and metrics to be collected and computed.
4. Identify the measures and metrics to be collected and computed.
  5. Establish a measurement collection process by answering these questions:
    - What is the source of the measurements?
    - Can tools be used to collect the data?
    - Who is responsible for collecting the data?
    - When are data collected and recorded?
    - How are data stored?
    - What validation mechanisms are used to ensure that the data are correct?
  6. Acquire appropriate tools to assist in collection and assessment.

7. Establish a metrics database.  
 The relative sophistication of the database is established.  
 Use of related tools (e.g., a SCM repository, Chapter 27) is explored.  
 Existing database products are evaluated.
8. Define appropriate feedback mechanisms.  
 Who requires on-going metrics information?

How is the information to be delivered?  
 What is the format of the information?

A considerably more detailed description of these eight steps can be downloaded from: <http://www.spc.ca/resources/metrics/>.

## 22.7 SUMMARY

Measurement enables managers and practitioners to improve the software process; assist in the planning, tracking, and control of a software project; and assess the quality of the product (software) that is produced. Measures of specific attributes of the process, project, and product are used to compute software metrics. These metrics can be analyzed to provide indicators that guide management and technical actions.

Process metrics enable an organization to take a strategic view by providing insight into the effectiveness of a software process. Project metrics are tactical. They enable a project manager to adapt project workflow and a technical approach in a real-time manner.

Both size- and function-oriented metrics are used throughout the industry. Size-oriented metrics use the line of code as a normalizing factor for other measures such as person-months or defects. The function point is derived from measures of the information domain and a subjective assessment of problem complexity. In addition, object-oriented metrics and Web application metrics can be used.

Software quality metrics, like productivity metrics, focus on the process, the project, and the product. By developing and analyzing a metrics baseline for quality, an organization can correct those areas of the software process that are the cause of software defects.

Measurement results in cultural change. Data collection, metrics computation, and metrics analysis are the three steps that must be implemented to begin a metrics program. In general, a goal-driven approach helps an organization focus on the right metrics for its business. By creating a metrics baseline—a database containing process and product measurements—software engineers and their managers can gain better insight into the work that they do and the product that they produce.

## REFERENCES

- [ALB83] Albrecht, A. J., and J. E. Gaffney, "Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation," *IEEE Trans. Software Engineering*, November 1983, pp. 639–648.
- [BOE81] Boehm, B., *Software Engineering Economics*, Prentice-Hall, 1981.

- [GRA87] Grady, R. B., and D. L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice-Hall, 1987.
- [GRA92] Grady, R. G., *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, 1992.
- [GIL88] Gilb, T., *Principles of Software Project Management*, Addison-Wesley, 1988.
- [HET93] Hetzel, W., *Making Software Measurement Work*, QED Publishing Group, 1993.
- [HUM95] Humphrey, W., *A Discipline for Software Engineering*, Addison-Wesley, 1995.
- [JEE93] *IEEE Software Engineering Standards*, Standard 610.12-1990, pp. 47–48.
- [JON86] Jones, C., *Programming Productivity*, McGraw-Hill, 1986.
- [JON91] Jones, C., *Applied Software Measurement*, McGraw-Hill, 1991.
- [JON98] Jones, C., *Estimating Software Costs*, McGraw-Hill, 1998.
- [LOR94] Lorenz, M., and J. Kidd, *Object-Oriented Software Metrics*, Prentice-Hall, 1994.
- [PAR96] Park, R. E., W. B. Goethert, and W. A. Florac, *Goal Driven Software Measurement—A Guidebook*, CMU/SEI-96-BH-002, Software Engineering Institute, Carnegie Mellon University, August 1996.
- [PAU94] Paulish, D., and A. Carleton, “Case Studies of Software Process Improvement Measurement,” *Computer*, vol. 27, no. 9, September 1994, pp. 50–57.
- [QSM02] “QSM Function Point Language Gearing Factors,” Version 2.0, Quantitative Software Management, 2002, <http://www.qsm.com/FPGearing.html>.
- [RAG95] Ragland, B., “Measure, Metric or Indicator: What’s the Difference?” *Crosstalk*, vol. 8, no. 3, March 1995, p. 29–30.
- [SMI99] Smith, J., “The Estimation of Effort Based on Use-Cases,” a white paper by Rational Corporation, 1999, downloaded from <http://www.rational.com/products/rup/whitepapers.jsp>.

## PROBLEMS AND POINTS TO PONDER

- 22.1.** Using the table presented in Section 22.2.3, make an argument against the use of assembler language based on the functionality delivered per statement of code. Again referring to the table, discuss why C++ would present a better alternative than C.
- 22.2.** Why should some software metrics be kept “private”? Provide examples of three metrics that should be private. Provide examples of three metrics that should be public.
- 22.3.** Describe the difference between process and project metrics in your own words.
- 22.4.** Compute the function point value for a project with the following information domain characteristics:
- Number of external inputs: 32
  - Number of external outputs: 60
  - Number of external inquiries: 24
  - Number of internal logical files: 8
  - Number of external interface files: 2
- Assume that all complexity adjustment values are average. Use the algorithm noted in Chapter 15.
- 22.5.** Present an argument against lines of code as a measure for software productivity. Will your case hold up when dozens or hundreds of projects are considered?
- 22.6.** Team A found 342 errors during the software engineering process prior to release. Team B found 184 errors. What additional measures would have to be made for projects A and B to determine which of the teams eliminated errors more efficiently? What metrics would you propose to help in making the determination? What historical data might be useful?
- 22.7.** Grady suggests an etiquette for software metrics. Can you add three more rules to those noted in Section 22.1.1?
- 22.8.** What is an indirect measure, and why are such measures common in software metrics work?

**22.9.** A software increment is delivered to end-users by a software team. The users uncover 8 defects during the first month of use. Prior to delivery, the software team found 242 errors during formal technical reviews and all testing tasks. What is the overall DRE for the project?

**22.10.** A Web engineering team has built a e-commerce WebApp that contains 145 individual pages. Of these pages, 65 are dynamic; i.e., they are internally generated based on end-user input. What is the customization index for this application?

**22.11.** The software used to control a photocopier requires 32,000 of C and 4200 lines of Smalltalk. Estimate the number of function points for the software inside the copier.

**22.12.** At the conclusion of a project that used the Unified Process (Chapter 3), it has been determined that 30 errors were found during the elaboration phase and 12 errors were found during construction phase that were traceable to errors that were not discovered in the elaboration phase. What is the DRE for these two phases?

**22.13.** A WebApp and its support environment has not been fully fortified against attack. Web engineers estimate that the likelihood of repelling an attack is only 30 percent. The system does not contain sensitive or controversial information, so the threat probability is 25 percent. What is the integrity of the WebApp?

## FURTHER READINGS AND INFORMATION SOURCES

Software process improvement (SPI) has received a significant amount of attention over the past two decades. Since measurement and software metrics are key to successfully improving the software process, many books on SPI also discuss metrics. Worthwhile sources of information on process metrics include:

Burr, A., and M. Owen, *Statistical Methods for Software Quality*, International Thomson Publishing, 1996.

El Emam, K., and N. Madhavji (eds.), *Elements of Software Process Assessment and Improvement*, IEEE Computer Society, 1999.

Florac, W. A., and A. D. Carleton, *Measuring the Software Process: Statistical Process Control for Software Process Improvement*, Addison-Wesley, 1999.

Garmus, D., and D. Herron, *Measuring the Software Process: A Practical Guide to Functional Measurements*, Prentice-Hall, 1996.

Humphrey, W., *Introduction to the Team Software Process*, Addison-Wesley/Longman, 2000.

Kan, S. H., *Metrics and Models in Software Quality Engineering*, Addison-Wesley, 1995.

McGarry and his colleagues (*Practical Software Measurement*, Addison-Wesley, 2001) present in-depth advice for assessing the software process. A worthwhile collection of papers has been edited by Haug and his colleagues (*Software Process Improvement: Metrics, Measurement, and Process Modeling*, Springer-Verlag, 2001). Florac and Carlton (*Measuring the Software Process*, Addison-Wesley, 1999) and Fenton and Pfleeger (*Software Metrics: A Rigorous and Practical Approach*, Revised, Brooks/Cole Publishers, 1998) discuss how software metrics can be used to provide the indicators necessary to improve the software process.

Putnam and Myers (*Five Core Metrics*, Dorset House, 2003) draw on a database of more than 6000 software projects to demonstrate how five core metrics—time, effort, size, reliability, and process productivity—can be used to control software projects. Maxwell (*Applied Statistics for Software Managers*, Prentice-Hall, 2003) presents techniques for analyzing software project data. Munson (*Software Engineering Measurement*, Auerbach, 2003) discusses a broad array of software engineering measurement issues. Jones (*Software Assessments, Benchmarks and Best Practices*, Addison-Wesley, 2000) describes both quantitative measurement and qualitative factors that help an organization assess its software process and practices. Garmus and Herron (*Function Point Analysis: Measurement Practices for Successful Software Projects*, Addison-Wesley, 2000) discuss process metrics with an emphasis on function point analysis.

Lorenze and Kidd [LOR94] and DeChampeax (*Object-Oriented Development Process and Metrics*, Prentice-Hall, 1996) consider the OO process and describe a set of metrics for assessing it. Whitmire (*Object-Oriented Design Measurement*, Wiley, 1997) and Henderson-Sellers (*Object-Oriented Metrics: Measures of Complexity*, Prentice-Hall, 1995) focus on technical metrics for OO work, but also consider measures and metrics that can be used at the process and product level.

Relatively little has been published on metrics for Web engineering work. However, Stern (*Web Metrics: Proven Methods for Measuring Web Site Success*, Wiley, 2002), Inan and Kean (*Measuring the Success of Your Website*, Longman, 2002), and Nobles and Grady (*Web Site Analysis and Reporting*, Premier Press, 2001) address Web metrics from a business and marketing perspective.

The latest research in the metrics area is summarized by the IEEE (*Symposium on Software Metrics*, published yearly). A wide variety of information sources on the process and project metrics is available on the Internet. An up-to-date list of World Wide Web references can be found at the SEPA Web site:

**<http://www.mhhe.com/pressman>.**

Start Quality

CHAPTER

23

Start

ESTIMATION

KEY CONCEPTS

- complexity
- estimation
- concepts
- FP-based
- LOC-based
- process-based
- reconciliation
- use-cases
- feasibility
- project planning
- resources
- scope
- software sizing

**S**oftware project management begins with a set of activities that are collectively called *project planning*. Before the project can begin, the project manager and the software team must estimate the work to be done, the resources that will be required, and the time that will elapse from start to finish. Once these activities are accomplished, the software team must establish a project schedule that defines software engineering tasks and milestones, identifies who is responsible for conducting each task, and specifies the inter-task dependencies that may have a strong bearing on progress.

In an excellent guide to “software project survival,” Steve McConnell [MCC98] presents a real-world view of project planning:

Many technical workers would rather do technical work than spend time planning. Many technical managers do not have sufficient training in technical management to feel confident that their planning will improve a project’s outcome. Since neither party wants to do planning, it often doesn’t get done.

But failure to plan is one of the most critical mistakes a project can make . . . effective planning is needed to resolve problems upstream [early in the project] at low cost, rather than downstream [late in the project] at high cost. The average project spends 80 percent of its time on rework—fixing mistakes that were made earlier in the project.

McConnell argues that every project can find the time to plan (and to adapt the plan throughout the project) simply by taking a small percentage of the time that would have been spent on rework that occurs because planning was not conducted.

QUICK LOOK

**What is it?** A real need for software has been established; stakeholders are on-board; software engineers are ready to start; and the project is about to begin. But how do you proceed? Software project planning encompasses five major activities—estimation, scheduling, risk analysis, quality management planning, and change management planning. In the context of this chapter, we consider only estimation—your attempt to determine how much money, effort, resources, and time it will take to build a specific software-based system or product.

**Who does it?** Software project managers—using information solicited from stakeholders and software engineers and software metrics data collected from past projects.

**Why is it important?** Would you build a house without knowing how much you were about to spend, the tasks you needed to perform, and the timeline for the work to be conducted? Of course not, and since most computer-based systems and products cost considerably more to build than a large house, it would seem reasonable to develop an estimate before you start creating the software.



**What are the steps?** Estimation begins with a description of the scope of the product. The problem is then decomposed into a set of smaller problems, and each of these is estimated using historical data and experience as guides. Problem complexity and risk are considered before a final estimate is made.

**What is the work product?** A simple table delineating the tasks to be performed, the functions to be implemented, and the cost, effort, and time involved for each is generated.

**How do I ensure that I've done it right?**

That's hard, because you won't really know until the project has been completed. However, if you have experience and follow a systematic approach, generate estimates using solid historical data, create estimation data points using at least two different methods, establish a realistic schedule, and continually adapt it as the project moves forward, you can feel confident that you've given it your best shot.

## 23.1 OBSERVATIONS ON ESTIMATION

Planning requires technical managers and members of the software team to make an initial commitment, even though it's likely that this "commitment" will be proven wrong. Whenever estimates are made, we look into the future and accept some degree of uncertainty as a matter of course. To quote Frederick Brooks [BRO75]:

[O]ur techniques of estimating are poorly developed. More seriously, they reflect an unvoiced assumption that is quite untrue, i.e., that all will go well . . . Because we are uncertain of our estimates, software managers often lack the courteous stubbornness to make people wait for a good product.

Although estimating is as much art as it is science, this important activity need not be conducted in a haphazard manner. Useful techniques for time and effort estimation do exist. Process and project metrics can provide historical perspective and powerful input for the generation of quantitative estimates. Past experience (of all people involved) can aid immeasurably as estimates are developed and reviewed. Because estimation lays a foundation for all other project planning activities, and project planning provides the road map for successful software engineering, we would be ill-advised to embark without it.

**"Good estimating approaches and solid historical data offer the best hope that reality will win out over impossible demands."**

**Capers Jones**

Estimation of resources, cost, and schedule for a software engineering effort requires experience, access to good historical information (metrics), and the courage to commit to quantitative predictions when qualitative information is all that exists. Estimation carries inherent risk<sup>1</sup>, and this risk leads to uncertainty.

<sup>1</sup> Systematic techniques for risk analysis are presented in Chapter 25.

The availability of historical information has a strong influence on estimation risk. By looking back, we can emulate things that worked and improve areas where problems arose. When comprehensive software metrics (Chapter 22) are available for past projects, estimates can be made with greater assurance, schedules can be established to avoid past difficulties, and overall risk is reduced.

*"It is the mark of an instructed mind to rest satisfied with the degree of precision that the nature of the subject admits, and not to seek exactness when only an approximation of the truth is possible."*

**Advice**

Estimation risk is measured by the degree of uncertainty in the quantitative estimates established for resources, cost, and schedule. If project scope is poorly understood or project requirements are subject to change, uncertainty and estimation risk become dangerously high. The planner, and more importantly, the customer should recognize that variability in software requirements means instability in cost and schedule.

However, a project manager should not become obsessive about estimation. Modern software engineering approaches (e.g., incremental process models) take an iterative view of development. In such approaches, it is possible—although not always politically acceptable—to revisit the estimate (as more information is known) and revise it when the customer makes changes to requirements.

## 23.2 THE PROJECT PLANNING PROCESS



*The more you know, the better you estimate. Therefore, update your estimates as the project progresses.*

The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule. In addition, estimates should attempt to define best-case and worst-case scenarios so that project outcomes can be bounded. Although there is an inherent degree of uncertainty, the software team embarks on a plan that has been established as a consequence of planning tasks. Therefore, the plan must be adapted and updated as the project proceeds. In the following sections, each of the activities associated with software project planning is discussed.

### TASK SET



#### Task Set for Project Planning

1. Establish project scope
2. Determine feasibility
3. Analyze risks (Chapter 25)
4. Define required resources
  - a. Determine human resources required
  - b. Define reusable software resources
  - c. Identify environmental resources
5. Estimate cost and effort
  - a. Decompose the problem
  - b. Develop two or more estimates using size, function points, process tasks, or use-cases
  - c. Reconcile the estimates
6. Develop a project schedule (Chapter 24)
  - a. Establish a meaningful task set
  - b. Define a task network
  - c. Use scheduling tools to develop a timeline chart
  - d. Define schedule tracking mechanisms

## 23.3 SOFTWARE SCOPE AND FEASIBILITY

### KEY POINT

Although there are many reasons for uncertainty, incomplete information about problem requirements dominates.

*Software scope* describes the functions and features that are to be delivered to end-users, the data that are input and output, the “content” that is presented to users as a consequence of using the software, and the performance, constraints, interfaces, and reliability that *bound* the system. Scope is defined using one of two techniques:

1. A narrative description of software scope is developed after communication with all stakeholders.
2. A set of use-cases<sup>2</sup> is developed by end-users.

Functions described in the statement of scope (or within the use-cases) are evaluated and in some cases refined to provide more detail prior to the beginning of estimation. Because both cost and schedule estimates are functionally oriented, some degree of decomposition is often useful. Performance considerations encompass processing and response time requirements. Constraints identify limits placed on the software by external hardware, available memory, or other existing systems.

Once scope has been identified (with the concurrence of the customer), it is reasonable to ask: Can we build software to meet this scope? Is the project feasible? All too often, software engineers rush past these questions (or are pushed past them by impatient managers or customers), only to become mired in a project that is doomed from the onset. Putnam and Myers [PUT97a] address this issue when they write:

[N]ot everything imaginable is feasible, not even in software, evanescent as it may appear to outsiders. On the contrary, software feasibility has four solid dimensions: *Technology*—Is a project technically feasible? Is it within the state of the art? Can defects be reduced to a level matching the application’s needs? *Finance*—Is it financially feasible? Can development be completed at a cost the software organization, its client, or the market can afford? *Time*—Will the project’s time-to-market beat the competition? *Resources*—Does the organization have the resources needed to succeed?

Putnam and Myers correctly suggest that scoping is not enough. Once scope is understood, the software team and others must work to determine if it can be done within the dimensions just noted. This is a crucial, although often overlooked, part of the estimation process.

### ADVICE

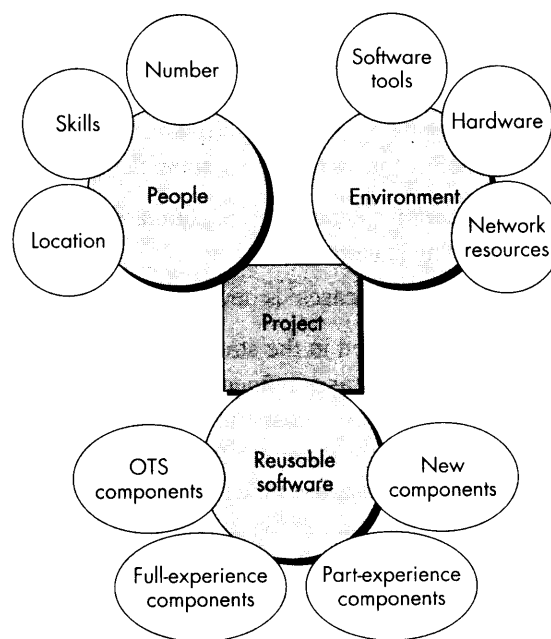
*Project feasibility is important, but a consideration of business need is even more important. It does no good to build a high-tech system or product that no one wants.*

## 23.4 RESOURCES

The second planning task is estimation of the resources required to accomplish the software development effort. Figure 23.1 depicts the three major categories of software engineering resources—people, reusable software components, and the development environment (hardware and software tools). Each resource is specified with

<sup>2</sup> Use-cases have been discussed in detail throughout Parts 2 and 3 of this book. A use-case is a scenario-based description of the user’s interaction with the software from the user’s point of view.

FIGURE 23.1

Project  
resources

four characteristics: description of the resource; a statement of availability; time when the resource will be required; duration of time that resource will be applied. The last two characteristics can be viewed as a *time window*. Availability of the resource for a specified window must be established at the earliest practical time.

#### 23.4.1 Human Resources

The planner begins by evaluating software scope and selecting the skills required to complete development. Both organizational position (e.g., manager, senior software engineer) and specialty (e.g., telecommunications, database, client/server) are specified. For relatively small projects (a few person-months), a single individual may perform all software engineering tasks, consulting with specialists as required. For larger projects, the software team may be geographically dispersed across a number of different locations. Hence, the location of each human resource is specified.

The number of people required for a software project can be determined only after an estimate of development effort (e.g., person-months) is made. Techniques for estimating effort are discussed later in this chapter.

#### 23.4.2 Reusable Software Resources

Component-based software engineering (Chapter 30) emphasizes reusability—that is, the creation and reuse of software building blocks [HOO91]. Such building blocks, often called *components*, must be cataloged for easy reference, standardized for easy application, and validated for easy integration.

Bennatan [BEN92] suggests four software resource categories that should be considered as planning proceeds:



*Never forget that integrating a variety of reusable components can be a significant challenge. The integration problem often resurfaces as various components are upgraded.*

*Off-the-shelf components.* Existing software can be acquired from a third party or has been developed internally for a past project. COTS (commercial off-the-shelf) components are purchased from a third party, are ready for use on the current project, and have been fully validated.

*Full-experience components.* Existing specifications, designs, code, or test data developed for past projects are similar to the software to be built for the current project. Members of the current software team have had full experience in the application area represented by these components. Therefore, modifications required for full-experience components will be relatively low-risk.

*Partial-experience components.* Existing specifications, designs, code, or test data developed for past projects are related to the software to be built for the current project but will require substantial modification. Members of the current software team have only limited experience in the application area represented by these components. Therefore, modifications required for partial-experience components have a fair degree of risk.

*New components.* Software components must be built by the software team specifically for the needs of the current project.

Ironically, reusable software components are often neglected during planning, only to become a paramount concern during the development phase of the software process. It is better to specify software resource requirements early. In this way technical evaluation of the alternatives can be conducted and timely acquisition can occur.

### 23.4.3 Environmental Resources

The environment that supports a software project, often called the *software engineering environment* (SEE), incorporates hardware and software. Hardware provides a platform that supports the tools (software) required to produce the work products that are an outcome of good software engineering practice.<sup>3</sup> Because most software organizations have multiple constituencies that require access to the SEE, a project planner must prescribe the time window required for hardware and software and verify that these resources will be available.

When a computer-based system (incorporating specialized hardware and software) is to be engineered, the software team may require access to hardware elements being developed by other engineering teams. For example, software for a numerical control (NC) used on a class of machine tools may require a specific machine tool (e.g., an

<sup>3</sup> Other hardware—the target environment—is the computer(s) on which the software will execute when it has been released to the end-user.

NC lathe) as part of the validation test step; a software project for advanced page-layout may need a high-quality printer at some point during development. Each hardware element must be specified by the software project planner.

Software is the most expensive element of virtually all computer-based systems. For complex, custom systems, a large cost estimation error can make the difference between profit and loss. Cost overrun can be disastrous for the developer.

*"In the face of contracting and increased competition, the ability to estimate more accurately ... has become a critical success factor for many IT groups."*



*Although software engineering effort is a dominant element of project cost, it's important to remember that other costs (e.g., development environment and tools, travel, training, office space, hardware) must also be considered.*

Software cost and effort estimation will never be an exact science.<sup>4</sup> Too many variables—human, technical, environmental, political—can affect the ultimate cost of software and effort applied to develop it. However, software project estimation can be transformed from a black art to a series of systematic steps that provide estimates with acceptable risk. To achieve reliable cost and effort estimates, a number of options arise:

1. Delay estimation until late in the project (obviously, we can achieve 100 percent accurate estimates after the project is complete!).
2. Base estimates on similar projects that have already been completed.
3. Use relatively simple decomposition techniques to generate project cost and effort estimates.
4. Use one or more empirical models for software cost and effort estimation.

Unfortunately, the first option, however attractive, is not practical. Cost estimates must be provided “up front.” However, we should recognize that the longer we wait, the more we know, and the more we know, the less likely we are to make serious errors in our estimates.

The second option can work reasonably well, if the current project is quite similar to past efforts and other project influences (e.g., the software team, the customer, business conditions, the SEE, deadlines) are roughly equivalent. Unfortunately, past experience has not always been a good indicator of future results.

The remaining options are viable approaches to software project estimation. Ideally, the techniques noted for each option should be applied in tandem; each used as a cross-check for the other. Decomposition techniques take a “divide and conquer” approach to software project estimation. By decomposing a project into major func-

<sup>4</sup> Bennatan [BEN03] reports that 40 percent of software developers continue to struggle with estimation and that software size and development time are very difficult to estimate accurately.

tions and related software engineering activities, cost and effort estimation can be performed in a stepwise fashion. Empirical estimation models can be used to complement decomposition techniques and offer a potentially valuable estimation approach in their own right. These models are discussed in Section 23.7.

Each of the viable software cost estimation options is only as good as the historical data used to seed the estimate. If no historical data exist, costing rests on a very shaky foundation. In Chapter 22, we examined the characteristics of some of the software metrics that provide the basis for historical estimation data.

## 23.6 DECOMPOSITION TECHNIQUE

Software project estimation is a form of problem solving, and in most cases, the problem to be solved (i.e., developing a cost and effort estimate for a software project) is too complex to be considered in one piece. For this reason, we decompose the problem, recharacterizing it as a set of smaller (and hopefully, more manageable) problems.

In Chapter 21, the decomposition approach was discussed from two different points of view: decomposition of the problem and decomposition of the process. Estimation uses one or both forms of partitioning. But before an estimate can be made, the project planner must understand the scope of the software to be built and generate an estimate of its “size.”

### 23.6.1 Software Sizing

The accuracy of a software project estimate is predicated on a number of things: (1) the degree to which the planner has properly estimated the size of the product to be built; (2) the ability to translate the size estimate into human effort, calendar time, and dollars (a function of the availability of reliable software metrics from past projects); (3) the degree to which the project plan reflects the abilities of the software team; and (4) the stability of product requirements and the environment that supports the software engineering effort.

In this section, we consider the *software sizing* problem. Because a project estimate is only as good as the estimate of the size of the work to be accomplished, sizing represents the project planner’s first major challenge. In the context of project planning, *size* refers to a quantifiable outcome of the software project. If a direct approach is taken, size can be measured in lines of code (LOC). If an indirect approach is chosen, size is represented as function points (FP).

Putnam and Myers [PUT92] suggest four different approaches to the sizing problem:

- *“Fuzzy logic” sizing.* To apply this approach, the planner must identify the type of application, establish its magnitude on a qualitative scale, and then refine the magnitude within the original range.
- *Function point sizing.* The planner develops estimates of the information domain characteristics discussed in Chapter 15.

#### KEY POINT

The “size” of software to be built can be estimated using a direct measure, LOC, or an indirect measure, FP.

**How do we size the software that we’re planning to build?**

- *Standard component sizing.* Software is composed of a number of different “standard components” that are generic to a particular application area. For example, the standard components for an information system are subsystems, modules, screens, reports, interactive programs, batch programs, files, LOC, and object-level instructions. The project planner estimates the number of occurrences of each standard component and then uses historical project data to determine the delivered size per standard component.
- *Change sizing.* This approach is used when a project encompasses the use of existing software that must be modified in some way as part of a project. The planner estimates the number and type (e.g., reuse, adding code, changing code, deleting code) of modifications that must be accomplished.

Putnam and Myers suggest that the results of each of these sizing approaches be combined statistically to create a *three-point* or *expected-value* estimate. This is accomplished by developing optimistic (low), most likely, and pessimistic (high) values for size and combining them using Equation (23-1) described in the next section.


### 23.6.2 Problem-Based Estimation

In Chapter 22, lines of code and function points were described as measures from which productivity metrics can be computed. LOC and FP data are used in two ways during software project estimation: (1) as an estimation variable to “size” each element of the software and (2) as baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.

LOC and FP estimation are distinct estimation techniques. Yet both have a number of characteristics in common. The project planner begins with a bounded statement of software scope and from this statement attempts to decompose software into problem functions that can each be estimated individually. LOC or FP (the estimation variable) is then estimated for each function. Alternatively, the planner may choose another component for sizing such as classes or objects, changes, or business processes affected.

Baseline productivity metrics (e.g., LOC/pm or FP/pm<sup>5</sup>) are then applied to the appropriate estimation variable, and cost or effort for the function is derived. Function estimates are combined to produce an overall estimate for the entire project.

It is important to note, however, that there is often substantial scatter in productivity metrics for an organization, making the use of a single baseline productivity metric suspect. In general, LOC/pm or FP/pm averages should be computed by project domain. That is, projects should be grouped by team size, application area, complexity, and other relevant parameters. Local domain averages should then be computed. When a new project is estimated, it should first be allocated to a domain,

 **What do LOC- and FP-based estimation have in common?**



*When collecting productivity metrics for projects, be sure to establish a taxonomy of project types. This will enable you to compute domain-specific averages, making estimation more accurate.*

<sup>5</sup> The acronym *pm* means person-month of effort.



and then the appropriate domain average for productivity should be used in generating the estimate.

The LOC and FP estimation techniques differ in the level of detail required for decomposition and the target of the partitioning. When LOC is used as the estimation variable, decomposition is absolutely essential and is often taken to considerable levels of detail. The greater the degree of partitioning, the more likely reasonably accurate estimates of LOC can be developed.

For FP estimates, decomposition works differently. Rather than focusing on function, each of the five information domain characteristics as well as the 14 complexity adjustment values discussed in Chapter 15 are estimated. The resultant estimates can then be used to derive a FP value that can be tied to past data and used to generate an estimate.

Regardless of the estimation variable that is used, the project planner begins by estimating a range of values for each function or information domain value. Using historical data or (when all else fails) intuition, the planner estimates an optimistic, most likely, and pessimistic size value for each function or count for each information domain value. An implicit indication of the degree of uncertainty is provided when a range of values is specified.

A three-point or expected-value can then be computed. The *expected-value* for the estimation variable (size),  $S$ , can be computed as a weighted average of the optimistic ( $S_{opt}$ ), most likely ( $S_m$ ), and pessimistic ( $S_{pess}$ ) estimates. For example,

$$S = (S_{opt} + 4S_m + S_{pess})/6 \quad (23-1)$$

gives heaviest credence to the “most likely” estimate and follows a beta probability distribution. We assume that there is a very small probability the actual size result will fall outside the optimistic or pessimistic values.

Once the expected value for the estimation variable has been determined, historical LOC or FP productivity data are applied. Are the estimates correct? The only reasonable answer to this question is: We can't be sure. Any estimation technique, no matter how sophisticated, must be cross-checked with another approach. Even then, common sense and experience must prevail.

### 23.6.3 An Example of LOC-Based Estimation

As an example of LOC and FP problem-based estimation techniques, let us consider a software package to be developed for a computer-aided design application for mechanical components. The software is to execute on an engineering workstation and must interface with various peripherals including a mouse, digitizer, high-resolution color display, and laser printer. A preliminary statement of software scope can be developed:

The mechanical CAD software will accept two- and three-dimensional geometric data from an engineer. The engineer will interact and control the CAD system through a user interface that will exhibit characteristics of good human/machine interface design. All

#### KEY POINT

For FP estimates, decomposition focuses on information domain characteristics.

? How do we compute the “expected value” for software size?



*Many modern applications reside on a network or are part of a client/server architecture. Therefore, be sure that your estimates include the effort required to develop "infrastructure" software.*

geometric data and other supporting information will be maintained in a CAD database. Design analysis modules will be developed to produce the required output, which will be displayed on a variety of graphics devices. The software will be designed to control and interact with peripheral devices that include a mouse, digitizer, laser printer, and plotter.

This statement of scope is preliminary—it is not bounded. Every sentence would have to be expanded to provide concrete detail and quantitative bounding. For example, before estimation can begin, the planner must determine what "characteristics of good human/machine interface design" means or what the size and sophistication of the "CAD database" are to be.

For our purposes, we assume that further refinement has occurred and that the major software functions listed in Figure 23.2 are identified. Following the decomposition technique for LOC, an estimation table, shown in Figure 23.2, is developed. A range of LOC estimates is developed for each function. For example, the range of LOC estimates for the 3D geometric analysis function is optimistic—4600 LOC, most likely—6900 LOC, and pessimistic—8600 LOC. Applying Equation (23-1), the expected value for the 3D geometric analysis function is 6800 LOC. Other estimates are derived in a similar fashion. By summing vertically in the estimated LOC column, an estimate of 33,200 lines of code is established for the CAD system.



*Do not succumb to the temptation to use this result as your project estimate. You should derive another result using a different approach.*

A review of historical data indicates that the organizational average productivity for systems of this type is 620 LOC/pm. Based on a burdened labor rate of \$8,000 per month, the cost per line of code is approximately \$13. Based on the LOC estimate and the historical productivity data, the total estimated project cost is \$431,000 and the estimated effort is 54 person-months.<sup>6</sup>

**FIGURE 23.2**

**Estimation table for the LOC methods**

Function	Estimated LOC
User interface and control facilities (UICF)	2,300
Two-dimensional geometric analysis (2DGA)	5,300
Three-dimensional geometric analysis (3DGA)	6,800
Database management (DBM)	3,350
Computer graphics display facilities (CGDF)	4,950
Peripheral control function (PCF)	2,100
Design analysis modules (DAM)	8,400
<i>Estimated lines of code</i>	<b>33,200</b>

<sup>6</sup> Estimates are rounded to the nearest \$1,000 and person month. Further precision is unnecessary and unrealistic, given the limitation of estimation accuracy.

## SAFEHOME

**Estimating**

**The scene:** Doug Miller's office as project planning begins.

**The players:** Doug Miller (manager of the SafeHome software engineering team) and Vinod Raman, Jamie Lollar, and other members of the product software engineering team.

**The conversation:**

**Doug:** We need to develop an effort estimate for the product, and then we've got to define a micro-schedule for the first increment and a macro schedule for the remaining increments.

**Vinod (mouthing):** Okay, but we haven't defined any increments yet.

**Doug:** True, but that's why we need to estimate.

**Jamie (frowning):** You want to know how long it's going to take us?

**Doug:** Here's what I need. First, we need to functionally decompose the SafeHome software . . . at a high level . . . then we've got to estimate the number of lines of code that each function will take . . . then . . .

**Jamie:** Whoa! How are we supposed to do that?

**Vinod:** I've done it on past projects. You use user needs to determine the functionality required to implement each, guesstimate the LOC count for each piece of the function. The best approach is to have everyone do it independently and then compare results.

**Doug:** Or you can do a functional decomposition for the entire project.

**Jamie:** But that'll take forever, and we've got to get started.

**Vinod:** No . . . it can be done in a few hours . . . this morning, in fact.

**Doug:** I agree . . . we can't expect exactitude, just a ball-park idea of what the size of SafeHome will be.

**Jamie:** I think we should just estimate effort . . . that's all.

**Doug:** We'll do that too. Then use both estimates as a cross check.

**Vinod:** Let's go do it. . . .

**23.6.4 An Example of FP-Based Estimation**

Decomposition for FP-based estimation focuses on information domain values rather than software functions. Referring to the table presented in Figure 23.3, the project planner estimates external inputs, external outputs, external inquiries, internal logical files, and external interface files for the CAD software. FP are computed using the technique discussed in Chapter 15. For the purposes of this estimate, the

**FIGURE 23.3**

Estimating information domain values

Information domain value	Opt.	Likely	Pess.	Est. count	Weight	FP count
Number of external inputs	20	24	30	24	4	97
Number of external outputs	12	15	22	16	5	78
Number of external inquiries	16	22	28	22	5	88
Number of internal logical files	4	4	5	4	10	42
Number of external interface files	2	2	3	2	7	15
<b>Count total</b>						<b>320</b>

complexity weighting factor is assumed to be average. Figure 23.3 presents the results of this estimate.

Each of the complexity weighting factors is estimated and the value adjustment factor is computed as described in Chapter 15:

Factor	Value
1. Backup and recovery	4
2. Data communications	2
3. Distributed processing	0
4. Performance critical	4
5. Existing operating environment	3
6. On-line data entry	4
7. Input transaction over multiple screens	5
8. ILFs updated online	3
9. Information domain values complex	5
10. Internal processing complex	5
11. Code designed for reuse	4
12. Conversion/installation in design	3
13. Multiple installations	5
14. Application designed for change	5
<b>Value adjustment factor</b>	<b>1.17</b>

Finally, the estimated number of FP is derived:

$$FP_{\text{estimated}} = \text{count-total} \times [0.65 + 0.01 \times \Sigma (F_i)]$$

$$FP_{\text{estimated}} = 375$$

The organizational average productivity for systems of this type is 6.5 FP/pm. Based on a burdened labor rate of \$8,000 per month, the cost per FP is approximately \$1,230. Based on the FP estimate and the historical productivity data, the total estimated project cost is \$461,000 and the estimated effort is 58 person-months.



*If time permits, use finer granularity when specifying tasks in Figure 23.4. For example, break analysis into its major tasks and estimate each separately.*

### 23.6.5 Process-Based Estimation

The most common technique for estimating a project is to base the estimate on the process that will be used. That is, the process is decomposed into a relatively small set of tasks and the effort required to accomplish each task is estimated.

Like problem-based techniques, process-based estimation begins with a delineation of software functions obtained from the project scope. A series of framework activities must be performed for each function. Functions and related framework activities<sup>7</sup> may be represented as part of a table similar to the one presented in Figure 23.4.

<sup>7</sup> The framework activities chosen for this project differ somewhat from the generic activities discussed in Chapter 2. They are customer communication (CC), planning, risk analysis, engineering, and construction/release.

**FIGURE 23.4**  
Process-based  
estimation  
table

Activity	CC	Planning	Risk analysis	Engineering		Construction release		CE	Totals
Task				Analysis	Design	Code	Test		
Function									
UICF				0.50	2.50	0.40	5.00	n/a	8.40
2DGA				0.75	4.00	0.60	2.00	n/a	7.35
3DGA				0.50	4.00	1.00	3.00	n/a	8.50
CGDF				0.50	3.00	1.00	1.50	n/a	6.00
DMF				0.50	3.00	0.75	1.50	n/a	5.75
PCF				0.25	2.00	0.50	1.50	n/a	4.25
DAM				0.50	2.00	0.50	2.00	n/a	5.00
Totals	0.25	0.25	0.25	3.50	20.50	4.50	16.50		46.00
% effort	1%	1%	1%	8%	45%	10%	36%		

CC = customer communication CE = customer evaluation

Once problem functions and process activities are melded, the planner estimates the effort (e.g., person-months) that will be required to accomplish each software process activity for each software function. These data constitute the central matrix of the table in Figure 23.4. Average labor rates (i.e., cost/unit effort) are then applied to the effort estimated for each process activity. It is very likely the labor rate will vary for each task. Senior staff are heavily involved in early framework activities and are generally more expensive than junior staff involved in construction and release.

Costs and effort for each function and framework activity are computed as the last step. If process-based estimation is performed independently of LOC or FP estimation, we now have two or three estimates for cost and effort that may be compared and reconciled. If both sets of estimates show reasonable agreement, there is good reason to believe that the estimates are reliable. If, on the other hand, the results of these decomposition techniques show little agreement, further investigation and analysis must be conducted.

*"It's best to understand the background of an estimate before you use it."*

**Barry Boehm and Richard Fairley**

### 23.6.6 An Example of Process-Based Estimation

To illustrate the use of process-based estimation, we again consider the CAD software introduced in Section 23.6.3. The system configuration and all software functions remain unchanged and are indicated by project scope.

Referring to the completed process-based table shown in Figure 23.4, estimates of effort (in person-months) for each software engineering activity are provided for each CAD software function (abbreviated for brevity). The engineering and construction release activities are subdivided into the major software engineering tasks

shown. Gross estimates of effort are provided for customer communication, planning, and risk analysis. These are noted in the total row at the bottom of the table. Horizontal and vertical totals provide an indication of estimated effort required for analysis, design, code, and test. It should be noted that 53 percent of all effort is expended on front-end engineering tasks (requirements analysis and design), indicating the relative importance of this work.

Based on an average burdened labor rate of \$8,000 per month, the total estimated project cost is \$368,000, and the estimated effort is 46 person-months. If desired, labor rates could be associated with each framework activity or software engineering task and computed separately.

### 23.6.7 Estimation with Use-Cases

As we have noted throughout Parts 2 and 3 of this book, use-cases provide a software team with insight into software scope and requirements. However, developing an estimation approach with use-cases is problematic for the following reasons [SMI99]:

- Use-cases are described using many different formats and styles—there is no standard form.
- Use-cases represent an external view (the user's view) of the software and are often written at different levels of abstraction.
- Use-cases do not address the complexity of the functions and features that are described.
- Use-cases do not describe complex behavior (e.g., interactions) that involves many functions and features.

Unlike a LOC or a function point, one person's "use-case" may require months of effort while another person's use-case may be implemented in a day or two.

Although a number of investigators have considered use-cases as an estimation input, no proven estimation method has emerged to date. Smith [SMI99] suggests that use-cases can be used for estimation, but only if they are considered within the context of the "structural hierarchy" that the use-cases describe.

Smith argues that any level of this structural hierarchy can be described by no more than 10 use-cases. Each of these use-cases would encompass no more than 30 distinct scenarios. Obviously, use-cases that describe a large system are written at a much higher level of abstraction (and represent considerably more development effort) than use-cases that are written to describe a single subsystem. Therefore, before use-cases can be used for estimation, the level within the structural hierarchy is established, the average length (in pages) of each use-case is determined, the type of software (e.g., real-time, business, engineering/scientific, embedded) is defined, and a rough architecture for the system is considered. Once these characteristics are established, empirical data may be used to establish the estimated number of LOC or

**Why is it difficult to develop an estimation technique using use-cases?**

FP per use case (for each level of the hierarchy). Historical data are then used to compute the effort required to develop the system.

To illustrate how this computation might be made, consider the following relationship:<sup>8</sup>

$$\text{LOC estimate} = N \times \text{LOC}_{\text{avg}} + [(S_a/S_h - 1) + (P_a/P_h - 1)] \times \text{LOC}_{\text{adjust}} \quad (23-2)$$

where

- $N$  = actual number of use-cases
- $\text{LOC}_{\text{avg}}$  = historical average LOC per use-case for this type of subsystem
- $\text{LOC}_{\text{adjust}}$  = represents an adjustment based on  $n$  percent of  $\text{LOC}_{\text{avg}}$  where  $n$  is defined locally and represents the difference between this project and "average" projects
- $S_a$  = actual scenarios per use-case
- $S_h$  = average scenarios per use-case for this type of subsystem
- $P_a$  = actual pages per use-case
- $P_h$  = average pages per use-case for this type of subsystem

Expression (23-2) is used to develop a rough estimate of the number of LOC based on the actual number of use-cases adjusted by the number of scenarios and the page length of the use-cases. The adjustment represents up to  $n$  percent of the historical average LOC per use case.

### 23.6.8 An Example of Use-Case Based Estimation

The CAD software introduced in Section 23.6.3 is composed of three subsystem groups:

- User interface subsystem (includes UICF).
- Engineering subsystem group (includes the 2DGA subsystem, 3DGA subsystem, and DAM subsystem).
- Infrastructure subsystem group (includes CGDF subsystem and PCF subsystem).

Six use-cases describe the user interface subsystem. Each use case is described by no more than 10 scenarios and has an average length of six pages. The engineering subsystem group is described by 10 use-cases (these are considered to be at a higher level of the structural hierarchy). Each of these use-cases has no more than 20 scenarios associated with it and has an average length of eight pages. Finally, the infrastructure subsystem group is described by five use-cases with an average of only six scenarios and an average length of five pages.

<sup>8</sup> It is important to note that Expression (23-2) is used for illustrative purposes only. Like all estimation models, it must be validated locally before it can be used with confidence.

FIGURE 23.5

## Use-case estimation

	use-cases	scenarios	pages	scenarios	pages	LOC	LOC estimate
User interface subsystem	6	10	6	12	5	560	3,366
Engineering subsystem group	10	20	8	16	8	3100	31,233
Infrastructure subsystem group	5	6	5	10	6	1650	7,970
Total LOC estimate							42,568

Using the relationship noted in Expression (23-2) with  $n = 30$  percent, the table shown in Figure 23.5 is developed. Considering the first row of the table, historical data indicate that UI software requires an average of 800 LOC per use-case when the use-case has no more than 12 scenarios and is described in less than five pages. These data conform reasonably well for the CAD system. Hence the LOC estimate for the user interface subsystem is computed using Expression (23-2). Using the same approach, estimates are made for both the engineering and infrastructure subsystem groups. Figure 23.5 summarizes the estimates and indicates that the overall size of the CAD software is estimated at 42,500 LOC.

Using 620 LOC/pm as the average productivity for systems of this type and a burdened labor rate of \$8,000 per month, the cost per line of code is approximately \$13. Based on the use-case estimate and the historical productivity data, the total estimated project cost is \$552,000 and the estimated effort is 68 person-months.

### 23.6.9 Reconciling Estimates

The estimation techniques discussed in the preceding sections result in multiple estimates which must be reconciled to produce a single estimate of effort, project duration, or cost. To illustrate this reconciliation procedure, we again consider the CAD software introduced in Section 23.6.3.

*"Complicated methods might not yield a more accurate estimate, particularly when developers can incorporate their own intuition into the estimate."*

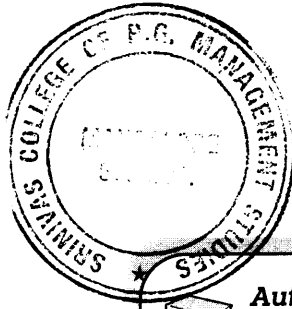
Philip Johnson et al.

Total estimated effort for the CAD software range from a low of 46 person-months (derived using a process-based estimation approach) to a high of 68 person-months (derived with use-case estimation). The average estimate (using all four approaches) is 56 person-months. The variation from the average estimate is approximately 18 percent on the low side and 21 percent on the high side.

What happens when agreement between estimates is poor? The answer to this question requires a reevaluation of information used to make the estimates. Widely divergent estimates can often be traced to one of two causes:

1. The scope of the project is not adequately understood or has been misinterpreted by the planner.






2. Productivity data used for problem-based estimation techniques is inappropriate for the application, obsolete (in that it no longer accurately reflects the software engineering organization), or has been misapplied.

The planner must determine the cause of divergence and then reconcile the estimates.

**INFO**

### Automated Estimation Techniques for Software Projects



Automated estimation tools allow the planner to estimate cost and effort and to perform “what-if” analyses for important project variables such as delivery date or staffing. Although many automated estimation tools exist (see sidebar later in this chapter), all exhibit the same general characteristics, and all perform the following six generic functions [JON96]:

1. *Sizing of project deliverables.* The “size” of one or more software work products is estimated. Work products include the external representation of software (e.g., screens, reports), the software itself (e.g., KLOC), functionality delivered (e.g., function points), and descriptive information (e.g. documents).
2. *Selecting project activities.* The appropriate process framework is selected, and the software engineering task set is specified.
3. *Predicting staffing levels.* The number of people who will be available to do the work is specified. Because the relationship between people available and work (predicted effort) is highly nonlinear, this is an important input.

4. *Predicting software effort.* Estimation tools use one or more models (Section 23.7) that relate the size of the project deliverables to the effort required to produce them.
5. *Predicting software cost.* Given the results of step 4, costs can be computed by allocating labor rates to the project activities noted in step 2.
6. *Predicting software schedules.* When effort, staffing level, and project activities are known, a draft schedule can be produced by allocating labor across software engineering activities based on recommended models for effort distribution discussed later in Chapter 24.

When different estimation tools are applied to the same project data, a relatively large variation in estimated results can be encountered. More important, predicted values sometimes are significantly different than actual values. This reinforces the notion that the output of estimation tools should be used as one “data point” from which estimates are derived—not as the only source for an estimate.

## 23.7 EMPIRICAL ESTIMATION MODELS

### KEY POINT

An estimation model reflects the population of projects from which it has been derived. Therefore, the model is domain sensitive.

An estimation model for computer software uses empirically derived formulas to predict effort as a function of LOC or FP.<sup>9</sup> Values for LOC or FP are estimated using the approach described in Sections 23.6.3 and 23.6.4. But instead of using the tables described in those sections, the resultant values for LOC or FP are plugged into the estimation model.

The empirical data that support most estimation models are derived from a limited sample of projects. For this reason, no estimation model is appropriate for all classes of software and in all development environments. Therefore, the results obtained from such models must be used judiciously.

<sup>9</sup> An empirical model using use-cases as the independent variable is suggested in Section 23.6.7. However, relatively few have appeared in the literature to date.

An estimation model should be calibrated to reflect local conditions. The model should be tested by applying data collected from completed projects, plugging the data into the model, and then comparing actual to predicted results. If agreement is poor, the model must be tuned and retested before it can be used.

### 23.7.1 The Structure of Estimation Models

A typical estimation model is derived using regression analysis on data collected from past software projects. The overall structure of such models takes the form [MAT94]

$$E = A + B \times (e_v)^C \quad (23-3)$$

where  $A$ ,  $B$ , and  $C$  are empirically derived constants,  $E$  is effort in person-months, and  $e_v$  is the estimation variable (either LOC or FP). In addition to the relationship noted in Equation (23-3), the majority of estimation models have some form of project adjustment component that enables  $E$  to be adjusted by other project characteristics (e.g., problem complexity, staff experience, development environment). Among the many LOC-oriented estimation models proposed in the literature are

$E = 5.2 \times (\text{KLOC})^{0.91}$	Walston-Felix model
$E = 5.5 + 0.73 \times (\text{KLOC})^{1.16}$	Bailey-Basili model
$E = 3.2 \times (\text{KLOC})^{1.05}$	Boehm simple model
$E = 5.288 \times (\text{KLOC})^{1.047}$	Doty model for KLOC > 9

FP-oriented models have also been proposed. These include

$E = -91.4 + 0.355 \text{ FP}$	Albrecht and Gaffney model
$E = -37 + 0.96 \text{ FP}$	Kemerer model
$E = -12.88 + 0.405 \text{ FP}$	small project regression model

A quick examination of these models indicates that each will yield a different result for the same values of LOC or FP. The implication is clear. Estimation models must be calibrated for local needs!



None of these models should be used without careful calibration to your environment.

#### WebRef

Detailed information on COCOMO II, including downloadable software, can be obtained at [sunset.usc.edu/research/COCOMOII/cocomo\\_main.html](http://sunset.usc.edu/research/COCOMOII/cocomo_main.html).

### 23.7.2 The COCOMO II Model

In his classic book on “software engineering economics,” Barry Boehm [BOE81] introduced a hierarchy of software estimation models bearing the name COCOMO, for *CONstructive COSt MOdel*. The original COCOMO model became one of the most widely used and discussed software cost estimation models in the industry. It has evolved into a more comprehensive estimation model, called COCOMO II [BOE96, BOE00]. Like its predecessor, COCOMO II is actually a hierarchy of estimation models that address the following areas:

- *Application composition model*. Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.

- *Early design stage model.* Used once requirements have been stabilized and basic software architecture has been established.
- *Post-architecture stage model.* Used during the construction of the software.

Like all estimation models for software, the COCOMO II models require sizing information. Three different sizing options are available as part of the model hierarchy: object points, function points, and lines of source code.

The COCOMO II application composition model uses object points—an indirect software measure that is computed using counts of the number of (1) screens (at the user interface), (2) reports, and (3) components likely to be required to build the application. Each object instance (e.g., a screen or report) is classified into one of three complexity levels (i.e., simple, medium, or difficult) using criteria suggested by Boehm [BOE96]. In essence, complexity is a function of the number and source of the client and server data tables that are required to generate the screen or report and the number of views or sections presented as part of the screen or report.

### What is an object point?

Once complexity is determined, the number of screens, reports, and components are weighted according to the table illustrated in Figure 23.6. The object point count is then determined by multiplying the original number of object instances by the weighting factor in the figure and summing to obtain a total object point count. When component-based development or general software reuse is to be applied, the percent of reuse (%reuse) is estimated and the object point count is adjusted:

$$\text{NOP} = (\text{object points}) \times [(100 - \% \text{reuse})/100]$$

where NOP is defined as new object points.

To derive an estimate of effort based on the computed NOP value, a “productivity rate” must be derived. Figure 23.7 presents the productivity rate

$$\text{PROD} = \text{NOP}/\text{person-month}$$

for different levels of developer experience and development environment maturity. Once the productivity rate has been determined, an estimate of project effort can be derived as

$$\text{estimated effort} = \text{NOP}/\text{PROD}$$

**FIGURE 23.6**

Complexity weighting for object types [BOE96]

Object type	Complexity weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL component			10

**FIGURE 23.7** Productivity rate for object points [BOE96]

<b>Developer's experience/capability</b>	Very low	Low	Nominal	High	Very high
<b>Environment maturity/capability</b>	Very low	Low	Nominal	High	Very high
<b>PROD</b>	4	7	13	25	50

In more advanced COCOMO II models,<sup>10</sup> a variety of scale factors, cost drivers, and adjustment procedures are required. The interested reader should see [BOE00] or visit the COCOMO II Web site.

### 23.7.3 The Software Equation

The software equation [PUT92] is a multivariable model that assumes a specific distribution of effort over the life of a software development project. The model has been derived from productivity data collected for over 4000 contemporary software projects. Based on these data, an estimation model of the form

$$E = [\text{LOC} \times B^{0.333}/P]^3 \times (1/t^4) \quad (23-4)$$

where

$E$  = effort in person-months or person-years

$t$  = project duration in months or years

$B$  = "special skills factor"<sup>11</sup>

$P$  = "productivity parameter" that reflects: overall process maturity and management practices, the extent to which good software engineering practices are used, the level of programming languages used, the state of the software environment, the skills and experience of the software team, and the complexity of the application.

Typical values might be  $P = 2000$  for development of real-time embedded software;  $P = 10,000$  for telecommunication and systems software;  $P = 28,000$  for business systems applications. The productivity parameter can be derived for local conditions using historical data collected from past development efforts.

It is important to note that the software equation has two independent parameters: (1) an estimate of size (in LOC) and (2) an indication of project duration in calendar months or years.

<sup>10</sup> As noted earlier, these models use FP and KLOC counts for the size variable.

<sup>11</sup>  $B$  increases slowly as "the need for integration, testing, quality assurance, documentation, and management skills grows" [PUT92]. For small programs (KLOC = 5 to 15),  $B = 0.16$ . For programs greater than 70 KLOC,  $B = 0.39$ .

#### WebRef

Information on software cost estimation tools that have evolved from the software equation can be found at [www.qsm.com](http://www.qsm.com).

To simplify the estimation process and use a more common form for their estimation model, Putnam and Myers [PUT92] suggest a set of equations derived from the software equation. Minimum development time is defined as

$$t_{\min} = 8.14 (LOC/P)^{0.43} \text{ in months for } t_{\min} > 6 \text{ months} \quad (23-5a)$$

$$E = 180 Bt^3 \text{ in person-months for } E \geq 20 \text{ person-months} \quad (23-5b)$$

Note that  $t$  in Equation (23-5b) is represented in years.

Using Equations (23-5) with  $P = 12,000$  (the recommended value for scientific software) for the CAD software discussed earlier in this chapter,

$$t_{\min} = 8.14 (33200/12000)^{0.43}$$

$$t_{\min} = 12.6 \text{ calendar months}$$

$$E = 180 \times 0.28 \times (1.05)^3$$

$$E = 58 \text{ person-months}$$

The results of the software equation correspond favorably with the estimates developed in Section 23.6. Like the COCOMO model noted in the preceding section, the software equation has evolved over the past decade. Further discussion of an extended version of this estimation approach can be found in [PUT97b].

## 23.8 ESTIMATION FOR OBJECT-ORIENTED PROJECTS

It is worthwhile to supplement conventional software cost estimation methods with an approach that has been designed explicitly for OO software. Lorenz and Kidd [LOR94] suggest the following approach:

1. Develop estimates using effort decomposition, FP analysis, and any other method that is applicable for conventional applications.
2. Using object-oriented analysis modeling (Chapter 8), develop use-cases and determine a count. Recognize that the number of use-cases may change as the project progresses.
3. From the analysis model, determine the number of key classes (called *analysis classes* in Chapter 8).
4. Categorize the type of interface for the application, and develop a multiplier for support classes:

Interface type	Multiplier
No GUI	2.0
Text-based user interface	2.25
GUI	2.5
Complex GUI	3.0

Multiply the number of key classes (step 3) by the multiplier to obtain an estimate for the number of support classes.

5. Multiply the total number of classes (key + support) by the average number of work-units per class. Lorenz and Kidd suggest 15 to 20 person-days per class.
6. Cross-check the class-based estimate by multiplying the average number of work-units per use-case.

## 23.9 Informal Requirements Estimation

The estimation techniques discussed in Sections 23.6, 23.7 and 23.8 can be used for any software project. However, when a software team encounters an extremely short project duration (weeks rather than months) that is likely to have a continuing stream of changes, project planning in general and estimation in particular should be abbreviated.<sup>12</sup> In the sections that follow, we examine two specialized estimation techniques.

### 23.9.1 Estimation for Agile Development

Because the requirements for an agile project (Chapter 4) are defined as a set of user scenarios (e.g., “stories” in Extreme Programming) it is possible to develop an estimation approach that is informal, yet reasonably disciplined and meaningful within the context of project planning for each software increment.

Estimation for agile projects uses a decomposition approach that encompasses the following steps:

1. Each user scenario (the equivalent of a mini-use-case created at the very start of a project by end-users or other stakeholders) is considered separately for estimation purposes.
2. The scenario is decomposed into the set of functions and the software engineering tasks that will be required to develop them.
  - 3a. Each task is estimated separately. Note: estimation can be based on historical data, an empirical model, or “experience.”
  - 3b. Alternatively, the “volume” (size) of the scenario can be estimated in LOC, FP, or some other volume-oriented measure (e.g., object points).
  - 4a. Estimates for each task are summed to create an estimate for the scenario.

**How are estimates developed when an agile process is applied?**



In the context of estimation for agile projects, “volume” is an estimate of the overall size of a user scenario in LOC or FP.

<sup>12</sup> “Abbreviated” does *not* mean eliminated. Even short duration projects must be planned, and estimation is the foundation of solid planning.

- 4b. Alternatively, the volume estimate for the scenario is translated into effort using historical data.
5. The effort estimates for all scenarios that are to be implemented for a given software increment are summed to develop the effort estimate for the increment.

Because the project duration required for the development of a software increment is quite short (typically 3–6 weeks), this estimation approach serves two purposes: (1) to ensure that the number of scenarios to be included in the increment conforms to the available resources, and (2) to establish a basis for allocating effort as the increment is developed.

### 23.9.2 Estimation for Web Engineering Projects

As we noted in Chapter 16, Web engineering projects often adopt the agile process model. A modified function point measure, coupled with the steps outlined in Section 23.9.1, can be used to develop an estimate for the WebApp.

Roetzheim [ROE00] suggests the following information domain values when adapting function points (Chapters 15 and 22) for WebApp estimation:

- *Inputs* are each input screen or form (for example, CGI or Java), each maintenance screen, and if you use a tab notebook metaphor anywhere, each tab.
- *Outputs* are each static Web page, each dynamic Web page script (for example, ASP, ISAPI, or other DHTML script), and each report (whether Web based or administrative in nature).
- *Tables* are each logical table in the database plus, if you are using XML to store data in a file, each XML object (or collection of XML attributes).
- *Interfaces* retain their definition as logical files (for example, unique record formats) into our out-of-the-system boundaries.
- *Queries* are each externally published or use a message-oriented interface. A typical example is DCOM or COM external references.

Function points (computed using the information domain values noted) are a reasonable indicator of volume for a WebApp.

Mendes and her colleagues [MEN01] suggest that the volume of a WebApp is best determined by collecting measures (called “predictor variables”) associated with the application (e.g., page count, media count, function count), its Web page characteristics (e.g., page complexity, linking complexity, graphic complexity), media characteristics (e.g., media duration), and functional characteristics (e.g., code length, reused code length). These measures can be used to develop empirical estimation models for total project effort, page authoring effort, media authoring effort, and scripting effort. However, further work remains to be done before such models can be used with confidence.

## SOFTWARE TOOLS

**Effort and Cost Estimation**

**Objective:** The objective of effort and cost estimation tools is to provide a project team with estimates of effort required, project duration, and cost in a manner that addresses the specific characteristics of the project at hand and the environment in which the project is to be built.

**Mechanics:** In general, cost estimation tools make use of a historical database derived from local projects, data collected across the industry, and an empirical model (e.g., COCOMO II) that is used to derive effort, duration and cost estimates. Characteristics of the project and the development environment are input, and the tool provides a range of estimation outputs.

**Representative Tools<sup>13</sup>**

*Costar*, developed by Softstar Systems ([www.softstarsystems.com](http://www.softstarsystems.com)), uses the COCOMO II model to develop software estimates.

*Cost Xpert*, developed by Cost Xpert Group, Inc. ([www.costxpert.com](http://www.costxpert.com)), integrates multiple estimation models and a historical project database.

*Estimate Professional*, developed by the Software Productivity Centre, Inc. ([www.spc.com](http://www.spc.com)), is based on COCOMO II and the SLIM Model.

*Knowledge Plan*, developed by Software Productivity Research ([www.spr.com](http://www.spr.com)), uses function point input as the primary driver for a complete estimation package.

*Price S*, developed by Price Systems ([www.pricystems.com](http://www.pricystems.com)), is one of the oldest and most widely used estimating tools for large-scale software development projects.

*SEER/SEM*, developed by Galorath Inc., ([www.galorath.com](http://www.galorath.com)), provides comprehensive estimation capability, sensitivity analysis, risk assessment, and other features.

*SLIM-Estimate*, developed by QSM ([www.qsm.com](http://www.qsm.com)), draws on comprehensive "industry knowledge bases" to provide a "sanity check" for estimates derived using local data.

## 23.10 THE MAKE/BUY DECISION

In many software application areas, it is often more cost effective to acquire rather than develop computer software. Software engineering managers are faced with a make/buy decision that can be further complicated by a number of acquisition options: (1) software may be purchased (or licensed) off the shelf, (2) "full-experience" or "partial-experience" software components (see Section 23.4.2) may be acquired and then modified and integrated to meet specific needs, or (3) software may be custom built by an outside contractor to meet the purchaser's specifications.

The steps involved in the acquisition of software are defined by the criticality of the software to be purchased and the end cost. In the final analysis, the make/buy decision is made based on the following conditions: (1) Will the software product be available sooner than internally developed software? (2) Will the cost of acquisition plus the cost of customization be less than the cost of developing the software internally? (3) Will the cost of outside support (e.g., a maintenance contract) be less than the cost of internal support? These conditions apply for each of the acquisition options.

<sup>13</sup> Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.



**? Is there a systematic way to sort through the options associated with the make/buy decision?**

### 23.10.1 Creating a Decision Tree

The steps just described can be augmented using statistical techniques such as *decision tree analysis* [BOE89]. For example, Figure 23.8 depicts a decision tree for a software-based system, X. In this case, the software engineering organization can (1) build system X from scratch, (2) reuse existing “partial-experience” components to construct the system, (3) buy an available software product and modify it to meet local needs, or (4) contract the software development to an outside vendor.

If the system is to be built from scratch, there is a 70 percent probability that the job will be difficult. Using the estimation techniques discussed earlier in this chapter, the project planner projects that a difficult development effort will cost \$450,000. A “simple” development effort is estimated to cost \$380,000. The *expected value* for cost, computed along any branch of the decision tree, is

$$\text{expected cost} = \sum (\text{path probability})_i \times (\text{estimated path cost}),$$

where  $i$  is the decision tree path. For the build path,

$$\text{expected cost}_{\text{build}} = 0.30 (\$380\text{K}) + 0.70 (\$450\text{K}) = \$429\text{K}$$

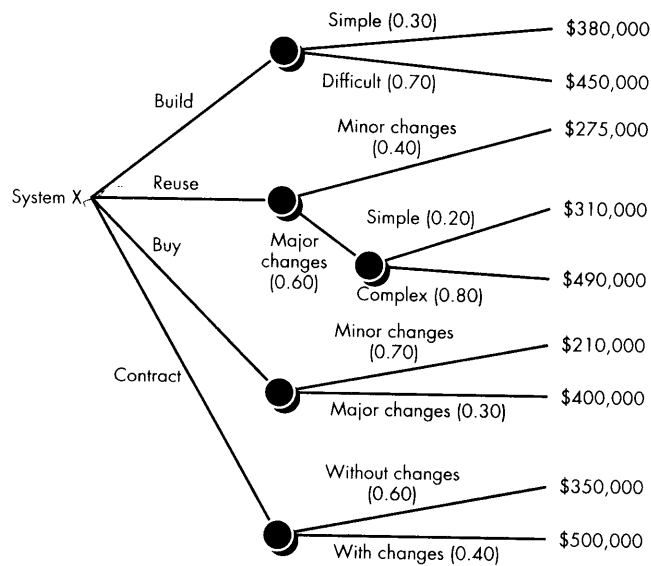
Following other paths of the decision tree, the projected costs for reuse, purchase and contract, under a variety of circumstances, are also shown. The expected costs for these paths are

$$\text{expected cost}_{\text{reuse}} = 0.40 (\$275\text{K}) + 0.60 [0.20 (\$310\text{K}) + 0.80 (\$490\text{K})] = \$382\text{K}$$

$$\text{expected cost}_{\text{buy}} = 0.70 (\$210\text{K}) + 0.30 (\$400\text{K}) = \$267\text{K}$$

$$\text{expected cost}_{\text{contract}} = 0.60 (\$350\text{K}) + 0.40 (\$500\text{K}) = \$410\text{K}$$

**FIGURE 23.8**  
A decision tree to support the make/buy decision



Based on the probability and projected costs that have been noted in Figure 23.8, the lowest expected cost is the “buy” option.

It is important to note, however, that many criteria—not just cost—must be considered during the decision-making process. Availability, experience of the developer/vendor/contractor, conformance to requirements, local “politics,” and the likelihood of change are but a few of the criteria that may affect the ultimate decision to build, reuse, buy, or contract.

### 23.10.2 Outsourcing

Sooner or later, every company that develops computer software asks a fundamental question: Is there a way that we can get the software and systems we need at a lower price? The answer to this question is not a simple one, and the emotional discussions that occur in response to the question always lead to a single word: *outsourcing*.

In concept, outsourcing is extremely simple. Software engineering activities are contracted to a third party who does the work at lower cost and, hopefully, higher quality. Software work conducted within a company is reduced to a contract management activity.<sup>14</sup>

**“As a rule outsourcing requires even more skillful management than in-house development.”**

**Steve McConnell**

The decision to outsource can be either strategic or tactical. At the strategic level, business managers consider whether a significant portion of all software work can be contracted to others. At the tactical level, a project manager determines whether part or all of a project can be best accomplished by subcontracting the software work.

Regardless of the breadth of focus, the outsourcing decision is often a financial one. A detailed discussion of the financial analysis for outsourcing is beyond the scope of this book and is best left to others (e.g., [MIN95]). However, a brief review of the pros and cons of the decision is worthwhile.

On the positive side, cost savings can usually be achieved by reducing the number of software people and the facilities (e.g., computers, infrastructure) that support them. On the negative side, a company loses some control over the software that it needs. Since software is a technology that differentiates its systems, services, and products, a company runs the risk of putting the fate of its competitiveness into the hands of a third party.

<sup>14</sup> Outsourcing can be viewed more generally as any activity that leads to the acquisition of software or software components from a source outside the software engineering organization.