relatively simple project might require the following work tasks for the *customer communication* activity:

1. Develop list of clarification issues.
2. Meet with customer to address clarification issues.
3. Jointly develop a statement of scope.
4. Review the statement of scope with all concerned.
5. Modify the statement of scope as required.

These events might occur over a period of less than 48 hours. They represent a process decomposition that is appropriate for the small, relatively simple project.

Now, we consider a more complex project, which has a broader scope and more significant business impact. Such a project might require the following work tasks for the customer communication activity:

Adaptable process model

1. Review the customer request.
2. Plan and schedule a formal, facilitated meeting with the customer.
3. Conduct research to specify the proposed solution and existing approaches.
4. Prepare a "working document" and an agenda for the formal meeting.
5. Conduct the meeting.
6. Jointly develop mini-specs that reflect data, function, and behavioral features of the software.
7. Review each mini-spec for correctness, consistency, and lack of ambiguity.
8. Assemble the mini-specs into a scoping document.
9. Review the scoping document with all concerned.
10. Modify the scoping document as required.

Both projects perform the framework activity that we call "customer communication," but the first project team performed half as many software engineering work tasks as the second.

3.5 THE PROJECT

Quote:

"At least 7 of 10 signs of IS project failures are determined before a design is developed or a line of code is written . . ."

John Reel

In order to manage a successful software project, we must understand what can go wrong (so that problems can be avoided) and how to do it right. In an excellent paper on software projects, John Reel [REE99] defines ten signs that indicate that an information systems project is in jeopardy:

1. Software people don't understand their customer's needs.
2. The product scope is poorly defined.
3. Changes are managed poorly.

4. The chosen technology changes.
5. Business needs change [or are ill-defined].
6. Deadlines are unrealistic.
7. Users are resistant.
8. Sponsorship is lost [or was never properly obtained].
9. The project team lacks people with appropriate skills.
10. Managers [and practitioners] avoid best practices and lessons learned.

Jaded industry professionals often refer to the 90–90 rule when discussing particularly difficult software projects: The first 90 percent of a system absorbs 90 percent of the allotted effort and time. The last 10 percent takes the other 90 percent of the allotted effort and time [ZAH94]. The seeds that lead to the 90–90 rule are contained in the signs noted in the preceeding list.

But enough negativity! How does a manager act to avoid the problems just noted? Reel [REE99] suggests a five-part commonsense approach to software projects:

1. **Start on the right foot.** This is accomplished by working hard (very hard) to understand the problem that is to be solved and then setting realistic objects and expectations for everyone who will be involved in the project. It is reinforced by building the right team (Section 3.2.3) and giving the team the autonomy, authority, and technology needed to do the job.

WebRef
A broad array of resources that can help both neophyte and experienced project managers can be found at **www.pmi.org**, **www.4pm.com**, and **www.projectmanagement.com**

2. **Maintain momentum.** Many projects get off to a good start and then slowly disintegrate. To maintain momentum, the project manager must provide incentives to keep turnover of personnel to an absolute minimum, the team should emphasize quality in every task it performs, and senior management should do everything possible to stay out of the team's way.[7]
3. **Track progress.** For a software project, progress is tracked as work products (e.g., specifications, source code, sets of test cases) are produced and approved (using formal technical reviews) as part of a quality assurance activity. In addition, software process and project measures (Chapter 4) can be collected and used to assess progress against averages developed for the software development organization.
4. **Make smart decisions.** In essence, the decisions of the project manager and the software team should be to "keep it simple." Whenever possible, decide to use commercial off-the-shelf software or existing software components, decide to avoid custom interfaces when standard approaches are

7 The implication of this statement is that bureacracy is reduced to a minimum, extraneous meetings are eliminated, and dogmatic adherence to process and project rules is eliminated. The team should be allowed to do its thing.

available, decide to identify and then avoid obvious risks, and decide to allocate more time than you think is needed to complex or risky tasks (you'll need every minute).

5. **Conduct a postmortem analysis.** Establish a consistent mechanism for extracting lessons learned for each project. Evaluate the planned and actual schedules, collect and analyze software project metrics, get feedback from team members and customers, and record findings in written form.

3.6 THE W^{5}HH PRINCIPLE

In an excellent paper on software process and projects, Barry Boehm [BOE96] states: "you need an organizing principle that scales down to provide simple [project] plans for simple projects." Boehm suggests an approach that addresses project objectives, milestones and schedules, responsibilities, management and technical approaches, and required resources. He calls it the WWWWWHH principle, after a series of questions that lead to a definition of key project characteristics and the resultant project plan:

What questions need to be answered in order to develop a project plan?

Why is the system being developed? The answer to this question enables all parties to assess the validity of business reasons for the software work. Stated in another way, does the business purpose justify the expenditure of people, time, and money?

What will be done, by when? The answers to these questions help the team to establish a project schedule by identifying key project tasks and the milestones that are required by the customer.

Who is responsible for a function? Earlier in this chapter, we noted that the role and responsibility of each member of the software team must be defined. The answer to this question helps accomplish this.

Where are they organizationally located? Not all roles and responsibilities reside within the software team itself. The customer, users, and other stakeholders also have responsibilities.

How will the job be done technically and managerially? Once product scope is established, a management and technical strategy for the project must be defined.

How much of each resource is needed? The answer to this question is derived by developing estimates (Chapter 5) based on answers to earlier questions.

Software Project Plan

Boehm's W^{5}HH principle is applicable regardless of the size or complexity of a software project. The questions noted provide an excellent planning outline for the project manager and the software team.

can be used either as a noun or a verb, definitions of the term can become confusing. Within the software engineering context, a measure provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process. Measurement is the act of determining a measure. The *IEEE Standard Glossary of Software Engineering Terms* [IEE93] defines *metric* as "a quantitative measure of the degree to which a system, component, or process possesses a given attribute."

Quote:

"Not everything that can be counted counts, and not everything that counts can be counted."

Albert Einstein

When a single data point has been collected (e.g., the number of errors uncovered in the review of a single module), a measure has been established. Measurement occurs as the result of the collection of one or more data points (e.g., a number of module reviews are investigated to collect measures of the number of errors for each). A software metric relates the individual measures in some way (e.g., the average number of errors found per review or the average number of errors found per person-hour expended on reviews.[1]

A software engineer collects measures and develops metrics so that indicators will be obtained. An *indicator* is a metric or combination of metrics that provide insight into the software process, a software project, or the product itself [RAG95]. An indicator provides insight that enables the project manager or software engineers to adjust the process, the project, or the process to make things better.

For example, four software teams are working on a large software project. Each team must conduct design reviews but is allowed to select the type of review that it will use. Upon examination of the metric, errors found per person-hour expended, the project manager notices that the two teams using more formal review methods exhibit an errors found per person-hour expended that is 40 percent higher than the other teams. Assuming all other parameters equal, this provides the project manager with an indicator that formal review methods may provide a higher return on time investment than another, less formal review approach. She may decide to suggest that all teams use the more formal approach. The metric provides the manager with insight. And insight leads to informed decision making.

4.2 METRICS IN THE PROCESS AND PROJECT DOMAINS

Measurement is commonplace in the engineering world. We measure power consumption, weight, physical dimensions, temperature, voltage, signal-to-noise ratio . . . the list is almost endless. Unfortunately, measurement is far less common in the software engineering world. We have trouble agreeing on what to measure and trouble evaluating measures that are collected.

1 This assumes that another measure, person-hours expended, is collected for each review.

WebRef
A comprehensive software metrics guidebook can be downloaded from **www.ivv.nasa.gov/SWG/resources/NASA-GB-001-94.pdf**

Metrics should be collected so that process and product indicators can be ascertained. *Process indicators* enable a software engineering organization to gain insight into the efficacy of an existing process (i.e., the paradigm, software engineering tasks, work products, and milestones). They enable managers and practitioners to assess what works and what doesn't. Process metrics are collected across all projects and over long periods of time. Their intent is to provide indicators that lead to long-term software process improvement.

Project indicators enable a software project manager to (1) assess the status of an ongoing project, (2) track potential risks, (3) uncover problem areas before they go "critical," (4) adjust work flow or tasks, and (5) evaluate the project team's ability to control quality of software work products.

In some cases, the same software metrics can be used to determine project and then process indicators. In fact, measures that are collected by a project team and converted into metrics for use during a project can also be transmitted to those with responsibility for software process improvement. For this reason, many of the same metrics are used in both the process and project domain.

4.2.1 Process Metrics and Software Process Improvement

(The only rational way to improve any process is to measure specific attributes of the process, develop a set of meaningful metrics based on these attributes, and then use the metrics to provide indicators that will lead to a strategy for improvement.) But before we discuss software metrics and their impact on software process improvement, it is important to note that process is only one of a number of "controllable factors in improving software quality and organizational performance [PAU94]."

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

Referring to Figure 4.1, process sits at the center of a triangle connecting three factors that have a profound influence on software quality and organizational performance. The skill and motivation of people has been shown [BOE81] to be the single most influential factor in quality and performance. The complexity of the product can have a substantial impact on quality and team performance. The technology (i.e., the software engineering methods) that populate the process also has an impact. 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).

How do I measure the effectiveness of a software process?

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

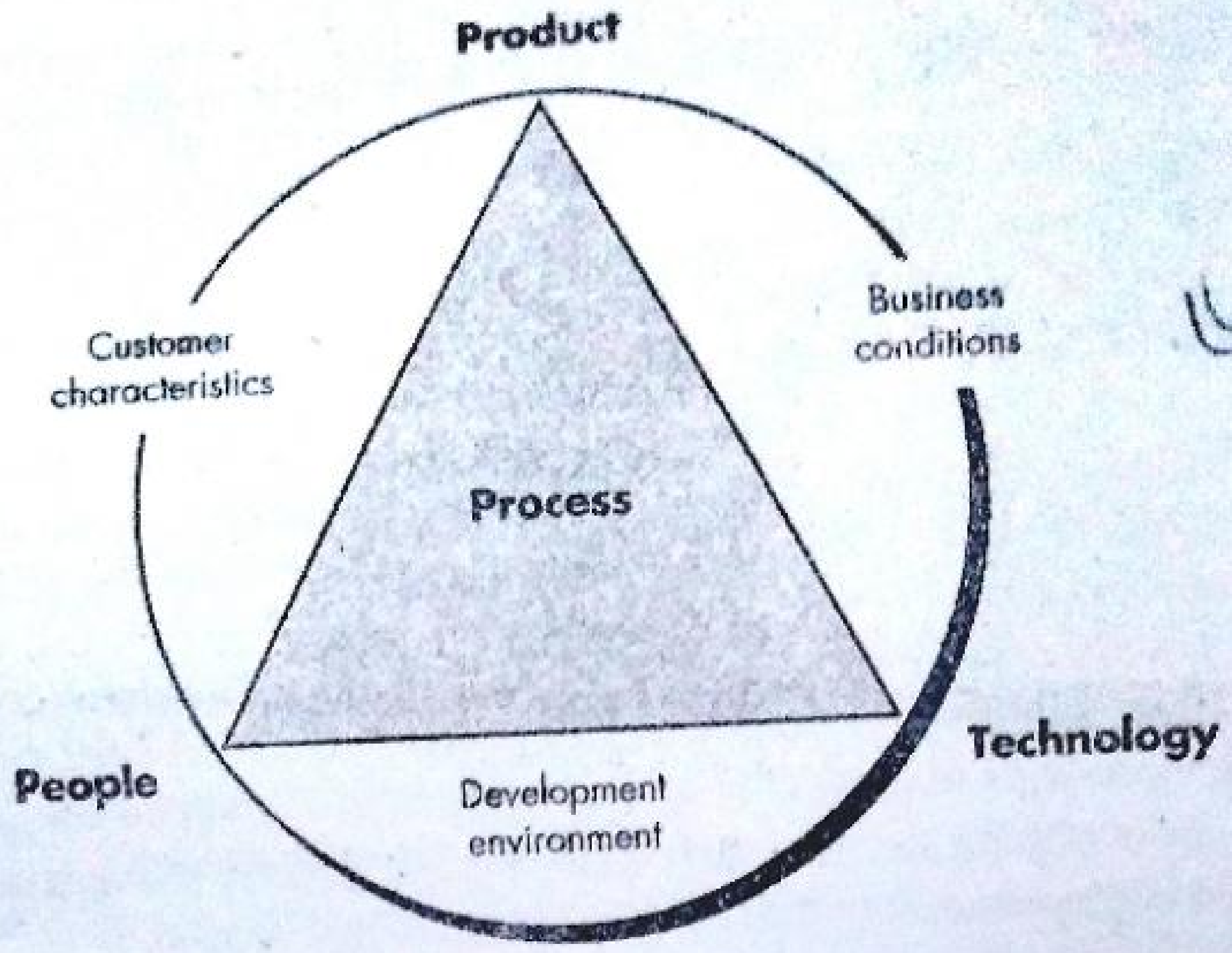

FIGURE 4.1 Determinants for software quality and organizational effectiveness (adapted from [PAU94])

performing the umbrella activities and the generic software engineering activities described in Chapter 2.

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 module), and errors found during development.

The "private process data" philosophy conforms well with the *personal software process* approach proposed by Humphrey [HUM95]. Humphrey describes the approach in the following manner:

> The personal software process (PSP) is a structured set of process descriptions, measurements, and methods that can help engineers to improve their personal performance. It provides the forms, scripts, and standards that help them estimate and plan their work. It shows them how to define processes and how to measure their quality and productivity. A fundamental PSP principle is that everyone is different and that a method that is effective for one engineer may not be suitable for another. The PSP thus helps engineers to measure and track their own work so they can find the methods that are best for them.

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 module and function.[2] These data are reviewed by the team to uncover indicators that can improve team performance.

KEY POINT

Public metrics enable an organization to make strategic changes that improve the software process and tactical changes during a software project.

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.

WebRef

SSPI and other quality related information is available through the American Society for Quality at www.asq.org

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[3] encountered as an application, system, or product is developed and used. Failure analysis works in the following manner:

1. All errors and defects are categorized by origin (e.g., flaw in specification, flaw in logic, nonconformance to standards).
2. The cost to correct each error and defect is recorded.

2 See Sections 4.3.1 and 4.3.2 for detailed discussions of LOC and function point metrics.
3 As we discuss in Chapter 8, an *error* is some flaw in a software engineering work product or deliverable that is uncovered by software engineers before the software is delivered to the end-user. A *defect* is a flaw that is uncovered after delivery to the end-user.

design quality and to provide indicators that will influence the approach taken to code generation and testing.

How should we use metrics during the project itself?

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.

Another model of software project metrics [HET93] suggests that every project should measure:

- *Inputs*—measures of the resources (e.g., people, environment) required to do the work.
- *Outputs*—measures of the deliverables or work products created during the software engineering process.
- *Results*—measures that indicate the effectiveness of the deliverables.

In actuality, this model can be applied to both process and project. In the project context, the model can be applied recursively as each framework activity occurs. Therefore the output from one activity becomes input to the next. Results metrics can be used to provide an indication of the usefulness of work products as they flow from one framework activity (or task) to the next.

4.3 SOFTWARE MEASUREMENT

Measurements in the physical world can be categorized in two ways: direct measures (e.g., the length of a bolt) and indirect measures (e.g., the "quality" of bolts produced, measured by counting rejects). Software metrics can be categorized similarly.

What is the difference between direct and indirect measures?

Direct measures of the software engineering process include cost and effort applied. Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time. *Indirect measures* of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many other "-abilities" that are discussed in Chapter 19.

The cost and effort required to build software, the number of lines of code produced, and other direct measures are relatively easy to collect, as long as specific conventions for measurement are established in advance. However, the quality and functionality of software or its efficiency or maintainability are more difficult to assess and can be measured only indirectly.

We have already partitioned the software metrics domain into process, project, and product metrics. We have also noted that product metrics that are private to an individual are often combined to develop project metrics that are public to a software team. Project metrics are then 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?

ADVICE

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

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.

4.3.1 Size-Oriented Metrics

What data should we collect to derive size-oriented metrics?

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the *size* of the software that has been produced. If a software organization maintains simple records, a table of size-oriented measures, such as the one shown in Figure 4.4, can be created. The table lists each software development project that has been completed over the past few years and corresponding measures for that project. Referring to the table entry (Figure 4.4) for project alpha: 12,100 lines of code were developed with 24 person-months of effort at a cost of $168,000. It should be noted that the effort and cost recorded in the table represent all software engineering activities (analysis, design, code, and test), not just coding. Further information for project alpha indicates that 365 pages of documentation were developed, 134 errors were recorded before the software was released, and 29 defects

line of Code [illegible]

Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
⋮	⋮	⋮	⋮	⋮	⋮		

FIGURE 4.4
Size-oriented metrics

Team Name?

were encountered after release to the customer within the first year of operation. Three people worked on the development of software for project alpha.

Metrics collection template

In order to develop metrics that can be assimilated with similar metrics from other projects, we choose lines of code as our normalization value. From the rudimentary data contained in the table, a set of simple size-oriented metrics can be developed for each project:

that are related

- Errors per KLOC (thousand lines of code).
- Defects[4] per KLOC.
- $ per LOC.
- Page of documentation per KLOC.

In addition, other interesting metrics can be computed:

- Errors per person-month.
- LOC per person-month.
- $ per page of documentation.

KEY POINT

Size-oriented metrics are widely used, but debate about their validity and applicability continues.

Size-oriented metrics are not universally accepted as the best way to measure the process of software development [JON86]. Most of the controversy swirls around the use of lines of code as a key measure. Proponents of the LOC measure claim that LOC is an "artifact" of all software development projects that can be easily counted, that many existing software estimation models use LOC or KLOC as a key input, and that a large body of literature and data predicated on LOC already exists. On the other hand, opponents argue that LOC measures are programming language dependent, that 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).

4.3.2 Function-Oriented Metrics

WebRef

Comprehensive information on function points can be obtained at **www.ifpug.org**

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. Since 'functionality' cannot be measured directly, it must be derived indirectly using other direct measures. Function-oriented metrics were first proposed by Albrecht [ALB79], who suggested a measure called the *function point*. Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity.

Function points are computed [IFP94] by completing the table shown in Figure 4.5. Five information domain characteristics are determined and counts are provided in

4 A defect occurs when quality assurance activities (e.g., formal technical reviews) fail to uncover an error in a work product produced during the software process.

FIGURE 4.5
Computing function points

			Weighting factor				
Measurement parameter	Count		Simple	Average	Complex		
Number of user inputs		×	3	4	6	=	
Number of user outputs		×	4	5	7	=	
Number of user inquiries		×	3	4	6	=	
Number of files		×	7	10	15	=	
Number of external interfaces		×	5	7	10	=	
Count total						→	

the appropriate table location. Information domain values are defined in the following manner.[5]

KEY POINT Function points are derived from direct measures of the information domain.

Number of user inputs. Each user input that provides distinct application-oriented data to the software is counted. Inputs should be distinguished from inquiries, which are counted separately.

Number of user outputs. Each user output that provides application-oriented information to the user is counted. In this context output refers to reports, screens, error messages, etc. Individual data items within a report are not counted separately.

Number of user inquiries. An inquiry is defined as an on-line input that results in the generation of some immediate software response in the form of an on-line output. Each distinct inquiry is counted.

Number of files. Each logical master file (i.e., a logical grouping of data that may be one part of a large database or a separate file) is counted.

Number of external interfaces. All machine readable interfaces (e.g., data files on storage media) that are used to transmit information to another system are counted.

Once these data have been collected, a complexity value is associated with each count. Organizations that use function point methods develop criteria for determining whether a particular entry is simple, average, or complex. Nonetheless, the determination of complexity is somewhat subjective.

To compute function points (FP), the following relationship is used:

$$FP = \text{count total} \times [0.65 + 0.01 \times \Sigma(F_i)] \qquad (4\text{-}1)$$

where count total is the sum of all FP entries obtained from Figure 4.5.

5 In actuality, the definition of information domain values and the manner in which they are counted are a bit more complex. The interested reader should see [IFP94] for details.

The F_i ($i = 1$ to 14) are "complexity adjustment values" based on responses to the following questions [ART85]:

1. Does the system require reliable backup and recovery?
2. Are data communications required?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require on-line data entry?
7. Does the on-line data entry require the input transaction to be built over multiple screens or operations?
8. Are the master files updated on-line?
9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and ease of use by the user?

Each of these questions is answered using a scale that ranges from 0 (not important or applicable) to 5 (absolutely essential). The constant values in Equation (4-1) and the weighting factors that are applied to information domain counts are determined empirically.

Once function points have been calculated, they are used in a manner analogous to LOC as a way to normalize measures for software productivity, quality, and other attributes:

- Errors per FP.
- Defects per FP.
- $ per FP.
- Pages of documentation per FP.
- FP per person-month.

KEY POINT

Extending function points are used for engineering, real-time, and control-oriented applications.

4.3.3 Extended Function Point Metrics

The function point measure was originally designed to be applied to business information systems applications. To accommodate these applications, the data dimension (the information domain values discussed previously) was emphasized to the exclusion of the functional and behavioral (control) dimensions. For this reason, the function point measure was inadequate for many engineering and embedded systems (which emphasize function and control). A number of extensions to the basic function point measure have been proposed to remedy this situation.

A function point extension called *feature points* [JON91], is a superset of the function point measure that can be applied to systems and engineering software applications.

to these questions is an emphatic "No!" The reason for this response is that many factors influence productivity, making for "apples and oranges" comparisons that can be easily misinterpreted.

Function points and LOC based metrics have been found to be relatively accurate predictors of software development effort and cost. However, in order to use LOC and FP for estimation (Chapter 5), a historical baseline of information must be established.

4.5 METRICS FOR SOFTWARE QUALITY

WebRef
An excellent source of information on software quality and related topics (including metrics) can be found at **www.qualityworld.com**

The overriding goal of software engineering is to produce a high-quality system, application, or product. 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.

The quality of a system, application, or product is only as good as the requirements that describe the problem, the design that models the solution, the code that leads to an executable program, and the tests that exercise the software to uncover errors. A good software engineer uses measurement to assess the quality of the analysis and design models, the source code, and the test cases that have been created as the software is engineered. To accomplish this real-time quality assessment, the engineer must use technical measures (Chapters 19 and 24) to evaluate quality in objective, rather than subjective ways.

XRef
A detailed discussion of software quality assurance activities is presented in Chapter 8.

The project manager must also evaluate quality as the project progresses. Private metrics collected by individual software engineers are assimilated to provide project-level results. Although many quality measures can be collected, the 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 4.5.3.

4.5.1 An Overview of Factors That Affect Quality

Over 25 years ago, McCall and Cavano [MCC78] defined a set of quality factors that were a first step toward the development of metrics for software quality. These factors assess software from three distinct points of view: (1) product operation (using it), (2) product revision (changing it), and (3) product transition (modifying it to work in a different environment; i.e., "porting" it). In their work, the authors describe the relationship between these quality factors (what they call a *framework*) and other aspects of the software engineering process:

> (First, the framework provides a mechanism for the project manager to identify what qualities are important. These qualities are attributes of the software in addition to its functional correctness and performance which have life cycle implications. Such factors as maintainability and portability have been shown in recent years to have significant life cycle cost impact.) .
>
> (Secondly, the framework provides a means for quantitatively assessing how well the development is progressing relative to the quality goals established.) .
>
> (Thirdly, the framework provides for more interaction of QA personnel throughout the development effort . .)
>
> (Lastly, . . . quality assurance personal can use indications of poor quality to help identify [better] standards to be enforced in the future.)

A detailed discussion of McCall and Cavano's framework, as well as other quality factors, is presented in Chapter 19. It is interesting to note that nearly every aspect of computing has undergone radical change as the years have passed since McCall and Cavano did their seminal work in 1978. But the attributes that provide an indication of software quality remain the same.

KEY POINT

Surprisingly, the factors that defined software quality in the 1970s are the same factors that continue to define software quality in the first decade of this century.

What does this mean? If a software organization adopts a set of quality factors as a "checklist" for assessing software quality, it is likely that software built today will still exhibit quality well into the first few decades of this century. Even as computing architectures undergo radical change (as they surely will), software that exhibits high quality in operation, transition, and revision will continue to serve its users well.

4.5.2 Measuring Quality

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

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.

Hitachi [TAJ81] has used a cost-oriented metric for maintainability called *spoilage*—the cost to correct defects encountered after the software has been released to its end-users. When the ratio of spoilage to overall project cost (for many projects) is plotted as a function of time, a manager can determine whether the overall maintainability of software produced by a software development organization is improving. Actions can then be taken in response to the insight gained from this information.

✓**Integrity.** Software integrity has become increasingly important in the age of hackers and firewalls. 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 the attack of a specific type will be repelled. The integrity of a system can then be defined as

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

where threat and security are summed over each type of attack.

✓**Usability.** The catch phrase "user-friendliness" has become ubiquitous in discussions of software products. If a program is not user-friendly, it is often doomed to failure, even if the functions that it performs are valuable. Usability is an attempt to quantify user-friendliness and can be measured in terms of four characteristics: (1) the physical and or intellectual skill required to learn the system, (2) the time required to become moderately efficient in the use of the system, (3) the net increase in productivity (over the approach that the system replaces) measured when the system is used by someone who is moderately efficient, and (4) a subjective assessment (sometimes obtained through a questionnaire) of users attitudes toward the system. Detailed discussion of this topic is contained in Chapter 15.

The four factors just described are only a sampling of those that have been proposed as measures for software quality. Chapter 19 considers this topic in additional detail.

4.5.3 Defect Removal Efficiency

A quality metric that provides benefit 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:

What is defect removal efficiency?

$$DRE = E/(E + D) \qquad (4\text{-}4)$$

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 the design task (where they may or may not be found). When used in this context, we redefine DRE as

ADVICE

Use DRE as a measure of the efficacy of your early SQA activities. If DRE is low during analysis and design, spend some time improving the way you conduct formal technical reviews.

$$DRE_i = E_i/(E_i + E_{i+1}) \qquad (4\text{-}5)$$

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.

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

4.6 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