4

Software Quality Metrics Overview

Software metrics can be classified into three categories: product metrics, process metrics, and project metrics. Product metrics describe the characteristics of the product such as size, complexity, design features, performance, and quality level. Process metrics can be used to improve software development and maintenance. Examples include the effectiveness of defect removal during development, the pattern of testing defect arrival, and the response time of the fix process. Project metrics describe the project characteristics and execution. Examples include the number of software developers, the staffing pattern over the life cycle of the software, cost, schedule, and productivity. Some metrics belong to multiple categories. For example, the in-process quality metrics of a project are both process metrics and project metrics.

Software quality metrics are a subset of software metrics that focus on the quality aspects of the product, process, and project. In general, software quality metrics are more closely associated with process and product metrics than with project metrics. Nonetheless, the project parameters such as the number of developers and their skill levels, the schedule, the size, and the organization structure certainly affect the quality of the product. Software quality metrics can be divided further into end-product quality metrics and in-process quality metrics. The essence of software quality engineering is to investigate the relationships among in-process metrics, project characteristics, and end-product quality, and, based on the findings, to engineer improvements in both process and product quality. Moreover, we should view quality from the entire software life-cycle perspective and, in this regard, we should include

metrics that measure the quality level of the maintenance process as another category of software quality metrics. In this chapter we discuss several metrics in each of three groups of software quality metrics: product quality, in-process quality, and maintenance quality. In the last sections we also describe the key metrics used by several major software developers and discuss software metrics data collection.

4.1 Product Quality Metrics

As discussed in Chapter 1, the de facto definition of software quality consists of two levels: intrinsic product quality and customer satisfaction. The metrics we discuss here cover both levels:

- Mean time to failure
- Defect density
- Customer problems
- Customer satisfaction.

Intrinsic product quality is usually measured by the number of "bugs" (functional defects) in the software or by how long the software can run before encountering a "crash." In operational definitions, the two metrics are defect density (rate) and mean time to failure (MTTF). The MTTF metric is most often used with safety-critical systems such as the airline traffic control systems, avionics, and weapons. For instance, the U.S. government mandates that its air traffic control system cannot be unavailable for more than three seconds per year. In civilian airliners, the probability of certain catastrophic failures must be no worse than 10^{-9} per hour (Littlewood and Strigini, 1992). The defect density metric, in contrast, is used in many commercial software systems.

The two metrics are correlated but are different enough to merit close attention. First, one measures the *time* between failures, the other measures the *defects* relative to the software size (lines of code, function points, etc.). Second, although it is difficult to separate defects and failures in actual measurements and data tracking, failures and defects (or faults) have different meanings. According to the IEEE/American National Standards Institute (ANSI) standard (982.2):

- An error is a human mistake that results in incorrect software.
- The resulting fault is an accidental condition that causes a unit of the system to fail to function as required.
- A defect is an anomaly in a product.
- A failure occurs when a functional unit of a software-related system can no longer perform its required function or cannot perform it within specified limits.

From these definitions, the difference between a fault and a defect is unclear. For practical purposes, there is no difference between the two terms. Indeed, in many development organizations the two terms are used synonymously. In this book we also use the two terms interchangeably.

Simply put, when an error occurs during the development process, a fault or a defect is injected in the software. In operational mode, failures are caused by faults or defects, or failures are materializations of faults. Sometimes a fault causes more than one failure situation and, on the other hand, some faults do not materialize until the software has been executed for a long time with some particular scenarios. Therefore, defect and failure do not have a one-to-one correspondence.

Third, the defects that cause higher failure rates are usually discovered and removed early. The probability of failure associated with a latent defect is called its size, or "bug size." For special-purpose software systems such as the air traffic control systems or the space shuttle control systems, the operations profile and scenarios are better defined and, therefore, the time to failure metric is appropriate. For general-purpose computer systems or commercial-use software, for which there is no typical user profile of the software, the MTTF metric is more difficult to implement and may not be representative of all customers.

Fourth, gathering data about time between failures is very expensive. It requires recording the occurrence time of each software failure. It is sometimes quite difficult to record the time for all the failures observed during testing or operation. To be useful, time between failures data also requires a high degree of accuracy. This is perhaps the reason the MTTF metric is not widely used by commercial developers.

Finally, the defect rate metric (or the volume of defects) has another appeal to commercial software development organizations. The defect rate of a product or the expected number of defects over a certain time period is important for cost and resource estimates of the maintenance phase of the software life cycle.

Regardless of their differences and similarities, MTTF and defect density are the two key metrics for intrinsic product quality. Accordingly, there are two main types of software reliability growth models—the time between failures models and the defect count (defect rate) models. We discuss the two types of models and provide several examples of each type in Chapter 8.

4.1.1 The Defect Density Metric

Although seemingly straightforward, comparing the defect rates of software products involves many issues. In this section we try to articulate the major points. To define a rate, we first have to operationalize the numerator and the denominator, and specify the time frame. As discussed in Chapter 3, the general concept of defect rate is the number of defects over the opportunities for error (OFE) during a specific time frame. We have just discussed the definitions of software defect and failure. Because

failures are defects materialized, we can use the number of unique causes of observed failures to approximate the number of defects in the software. The denominator is the size of the software, usually expressed in thousand lines of code (KLOC) or in the number of function points. In terms of time frames, various operational definitions are used for the life of product (LOP), ranging from one year to many years after the software product's release to the general market. In our experience with operating systems, usually more than 95% of the defects are found within four years of the software's release. For application software, most defects are normally found within two years of its release.

Lines of Code

The lines of code (LOC) metric is anything but simple. The major problem comes from the ambiguity of the operational definition, the actual counting. In the early days of Assembler programming, in which one physical line was the same as one instruction, the LOC definition was clear. With the availability of high-level languages the one-to-one correspondence broke down. Differences between physical lines and instruction statements (or logical lines of code) and differences among languages contribute to the huge variations in counting LOCs. Even within the same language, the methods and algorithms used by different counting tools can cause significant differences in the final counts. Jones (1986) describes several variations:

- ☐ Count only executable lines.
- ☐ Count executable lines plus data definitions.
- ☐ Count executable lines, data definitions, and comments.
- ☐ Count executable lines, data definitions, comments, and job control language.
- ☐ Count lines as physical lines on an input screen.
- ☐ Count lines as terminated by logical delimiters.

To illustrate the variations in LOC count practices, let us look at a few examples by authors of software metrics. In Boehm's well-known book *Software Engineering Economics* (1981), the LOC counting method counts lines as physical lines and includes executable lines, data definitions, and comments. In *Software Engineering Metrics and Models* by Conte et al. (1986), LOC is defined as follows:

> A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements. (p. 35)

Thus their method is to count physical lines including prologues and data definitions (declarations) but not comments. In *Programming Productivity* by Jones

(1986), the source instruction (or logical lines of code) method is used. The method used by IBM Rochester is also to count source instructions including executable lines and data definitions but excluding comments and program prologues.

The resultant differences in program size between counting physical lines and counting instruction statements are difficult to assess. It is not even known which method will result in a larger number. In some languages such as BASIC, PASCAL, and C, several instruction statements can be entered on one physical line. On the other hand, instruction statements and data declarations might span several physical lines, especially when the programming style aims for easy maintenance, which is not necessarily done by the original code owner. Languages that have a fixed column format such as FORTRAN may have the physical-lines-to-source-instructions ratio closest to one. According to Jones (1992), the difference between counts of physical lines and counts including instruction statements can be as large as 500%; and the average difference is about 200%, with logical statements outnumbering physical lines. In contrast, for COBOL the difference is about 200% in the opposite direction, with physical lines outnumbering instruction statements.

There are strengths and weaknesses of physical LOC and logical LOC (Jones, 2000). In general, logical statements are a somewhat more rational choice for quality data. When any data on size of program products and their quality are presented, the method for LOC counting should be described. At the minimum, in any publication of quality when LOC data is involved, the author should state whether the LOC counting method is based on physical LOC or logical LOC.

Furthermore, as discussed in Chapter 3, some companies may use the straight LOC count (whatever LOC counting method is used) as the denominator for calculating defect rate, whereas others may use the normalized count (normalized to Assembler-equivalent LOC based on some conversion ratios) for the denominator. Therefore, industrywide standards should include the conversion ratios from high-level language to Assembler. So far, very little research on this topic has been published. The conversion ratios published by Jones (1986) are the most well known in the industry. As more and more high-level languages become available for software development, more research will be needed in this area.

When straight LOC count data is used, size and defect rate comparisons across languages are often invalid. Extreme caution should be exercised when comparing the defect rates of two products if the operational definitions (counting) of LOC, defects, and time frame are not identical. Indeed, we do not recommend such comparisons. We recommend comparison against one's own history for the sake of measuring improvement over time.

Note: The LOC discussions in this section are in the context of defect rate calculation. For productivity studies, the problems with using LOC are more severe. A basic problem is that the amount of LOC in a softare program is negatively correlated with design efficiency. The purpose of software is to provide certain functionality for

solving some specific problems or to perform certain tasks. Efficient design provides the functionality with lower implementation effort and fewer LOCs. Therefore, using LOC data to measure software productivity is like using the weight of an airplane to measure its speed and capability. In addition to the level of languages issue, LOC data do not reflect noncoding work such as the creation of requirements, specifications, and user manuals. The LOC results are so misleading in productivity studies that Jones states "using lines of code for productivity studies involving multiple languages and full life cycle activities should be viewed as professional malpractice" (2000, p. 72). For detailed discussions of LOC and function point metrics, see Jones's work (1986, 1992, 1994, 1997, 2000).

When a software product is released to the market for the first time, and when a certain LOC count method is specified, it is relatively easy to state its quality level (projected or actual). For example, statements such as the following can be made: "This product has a total of 50 KLOC; the latent defect rate for this product during the next four years is 2.0 defects per KLOC." However, when enhancements are made and subsequent versions of the product are released, the situation becomes more complicated. One needs to measure the quality of the entire product as well as the portion of the product that is new. The latter is the measurement of true development quality—the defect rate of the new and changed code. Although the defect rate for the entire product will improve from release to release due to aging, the defect rate of the new and changed code will not improve unless there is real improvement in the development process. To calculate defect rate for the new and changed code, the following must be available:

- *LOC count:* The entire software product as well as the new and changed code of the release must be available.
- *Defect tracking:* Defects must be tracked to the release origin—the portion of the code that contains the defects and at what release the portion was added, changed, or enhanced. When calculating the defect rate of the entire product, all defects are used; when calculating the defect rate for the new and changed code, only defects of the release origin of the new and changed code are included.

These tasks are enabled by the practice of change flagging. Specifically, when a new function is added or an enhancement is made to an existing function, the new and changed lines of code are flagged with a specific identification (ID) number through the use of comments. The ID is linked to the requirements number, which is usually described briefly in the module's prologue. Therefore, any changes in the program modules can be linked to a certain requirement. This linkage procedure is part of the software configuration management mechanism and is usually practiced by organizations that have an established process. If the change-flagging IDs and requirements

IDs are further linked to the release number of the product, the LOC counting tools can use the linkages to count the new and changed code in new releases. The change-flagging practice is also important to the developers who deal with problem determination and maintenance. When a defect is reported and the fault zone determined, the developer can determine in which function or enhancement pertaining to what requirements at what release origin the defect was injected.

The new and changed LOC counts can also be obtained via the delta-library method. By comparing program modules in the original library with the new versions in the current release library, the LOC count tools can determine the amount of new and changed code for the new release. This method does not involve the change-flagging method. However, change flagging remains very important for maintenance. In many software development environments, tools for automatic change flagging are also available.

Example: Lines of Code Defect Rates

At IBM Rochester, lines of code data is based on instruction statements (logical LOC) and includes executable code and data definitions but excludes comments. LOC counts are obtained for the total product and for the new and changed code of the new release. Because the LOC count is based on source instructions, the two size metrics are called *shipped source instructions* (*SSI*) and new and *changed source instructions* (*CSI*), respectively. The relationship between the SSI count and the CSI count can be expressed with the following formula:

SSI (current release) = SSI (previous release)
+ CSI (new and changed code instructions for current release)
– deleted code (usually very small)
– changed code (to avoid double count in both SSI and CSI)

Defects after the release of the product are tracked. Defects can be field defects, which are found by customers, or internal defects, which are found internally. The several postrelease defect rate metrics per thousand SSI (KSSI) or per thousand CSI (KCSI) are:

(1) Total defects per KSSI (a measure of code quality of the total product)

(2) Field defects per KSSI (a measure of defect rate in the field)

(3) Release-origin defects (field and internal) per KCSI (a measure of development quality)

(4) Release-origin field defects per KCSI (a measure of development quality per defects found by customers)

Metric (1) measures the total release code quality, and metric (3) measures the quality of the new and changed code. For the initial release where the entire product is new, the two metrics are the same. Thereafter, metric (1) is affected by aging and the improvement (or deterioration) of metric (3). Metrics (1) and (3) are process measures; their field counterparts, metrics (2) and (4) represent the customer's perspective. Given an estimated defect rate (KCSI or KSSI), software developers can minimize the impact to customers by finding and fixing the defects before customers encounter them.

Customer's Perspective

The defect rate metrics measure code quality per unit. It is useful to drive quality improvement from the development team's point of view. Good practice in software quality engineering, however, also needs to consider the customer's perspective. Assume that we are to set the defect rate goal for release-to-release improvement of one product. From the customer's point of view, the defect rate is not as relevant as the total number of defects that might affect their business. Therefore, a good defect rate target should lead to a release-to-release reduction in the total number of defects, regardless of size. If a new release is larger than its predecessors, it means the defect rate goal for the new and changed code has to be significantly better than that of the previous release in order to reduce the total number of defects.

Consider the following hypothetical example:

Initial Release of Product Y

KCSI = KSSI = 50 KLOC

Defects/KCSI = 2.0

Total number of defects = $2.0 \times 50 = 100$

Second Release

KCSI = 20

KSSI = 50 + 20 (new and changed lines of code) − 4 (assuming 20% are changed lines of code) = 66

Defect/KCSI = 1.8 (assuming 10% improvement over the first release)

Total number of additional defects = $1.8 \times 20 = 36$

Third Release

KCSI = 30

KSSI = 66 + 30 (new and changed lines of code) − 6 (assuming the same % (20%) of changed lines of code) = 90

Targeted number of additional defects (no more than previous release) = 36

Defect rate target for the new and changed lines of code: 36/30 = 1.2 defects/KCSI or lower

From the initial release to the second release the defect rate improved by 10%. However, customers experienced a 64% reduction [(100 − 36)/100] in the number of defects because the second release is smaller. The size factor works against the third release because it is much larger than the second release. Its defect rate has to be one-third (1.2/1.8) better than that of the second release for the number of new defects not to exceed that of the second release. Of course, sometimes the difference between the two defect rate targets is very large and the new defect rate target is deemed not achievable. In those situations, other actions should be planned to improve the quality of the base code or to reduce the volume of postrelease field defects (i.e., by finding them internally).

Function Points

Counting lines of code is but one way to measure size. Another one is the *function point*. Both are surrogate indicators of the opportunities for error (OFE) in the defect density metrics. In recent years the function point has been gaining acceptance in application development in terms of both productivity (e.g., function points per person-year) and quality (e.g., defects per function point). In this section we provide a concise summary of the subject.

A *function* can be defined as a collection of executable statements that performs a certain task, together with declarations of the formal parameters and local variables manipulated by those statements (Conte et al., 1986). The ultimate measure of software productivity is the number of functions a development team can produce given a certain amount of resource, regardless of the size of the software in lines of code. The defect rate metric, ideally, is indexed to the number of functions a software provides. If defects per unit of functions is low, then the software should have better quality even though the defects per KLOC value could be higher—when the functions were implemented by fewer lines of code. However, measuring functions is theoretically promising but realistically very difficult.

The function point metric, originated by Albrecht and his colleagues at IBM in the mid-1970s, however, is something of a misnomer because the technique does not measure functions explicitly (Albrecht, 1979). It does address some of the problems associated with LOC counts in size and productivity measures, especially the differences in LOC counts that result because different levels of languages are used. It is a weighted total of five major components that comprise an application:

- Number of external inputs (e.g., transaction types) × 4
- Number of external outputs (e.g., report types) × 5
- Number of logical internal files (files as the user might conceive them, not physical files) × 10
- Number of external interface files (files accessed by the application but not maintained by it) × 7
- Number of external inquiries (types of online inquiries supported) × 4

These are the average weighting factors. There are also low and high weighting factors, depending on the complexity assessment of the application in terms of the five components (Kemerer and Porter, 1992; Sprouls, 1990):

- External input: low complexity, 3; high complexity, 6
- External output: low complexity, 4; high complexity, 7
- Logical internal file: low complexity, 7; high complexity, 15
- External interface file: low complexity, 5; high complexity, 10
- External inquiry: low complexity, 3; high complexity, 6

The complexity classification of each component is based on a set of standards that define complexity in terms of objective guidelines. For instance, for the external output component, if the number of data element types is 20 or more and the number of file types referenced is 2 or more, then complexity is high. If the number of data element types is 5 or fewer and the number of file types referenced is 2 or 3, then complexity is low.

With the weighting factors, the first step is to calculate the function counts (FCs) based on the following formula:

$$\mathrm{FC} = \sum_{i=1}^{5} \sum_{j=1}^{3} w_{ij} \times x_{ij}$$

where w_{ij} are the weighting factors of the five components by complexity level (low, average, high) and x_{ij} are the numbers of each component in the application.

The second step involves a scale from 0 to 5 to assess the impact of 14 general system characteristics in terms of their likely effect on the application. The 14 characteristics are:

1. Data communications
2. Distributed functions
3. Performance
4. Heavily used configuration
5. Transaction rate
6. Online data entry
7. End-user efficiency
8. Online update
9. Complex processing
10. Reusability
11. Installation ease
12. Operational ease
13. Multiple sites
14. Facilitation of change

The scores (ranging from 0 to 5) for these characteristics are then summed, based on the following formula, to arrive at the value adjustment factor (VAF)

$$\text{VAF} = 0.65 + 0.01 \sum_{i=1}^{14} c_i$$

where c_i is the score for general system characteristic i. Finally, the number of function points is obtained by multiplying function counts and the value adjustment factor:

$$\text{FP} = \text{FC} \times \text{VAF}$$

This equation is a simplified description of the calculation of function points. One should consult the fully documented methods, such as the International Function Point User's Group Standard (IFPUG, 1999), for a complete treatment.

Over the years the function point metric has gained acceptance as a key productivity measure in the application world. In 1986 the IFPUG was established. The IFPUG counting practices committee is the de facto standards organization for function point counting methods (Jones, 1992, 2000). Classes and seminars on function points counting and applications are offered frequently by consulting firms and at software conferences. In application contract work, the function point is often used to measure the amount of work, and quality is expressed as defects per function point. In systems and real-time software, however, the function point has been slow to gain acceptance. This is perhaps due to the incorrect impression that function points work only for information systems (Jones, 2000), the inertia of the LOC-related practices, and the effort required for function points counting. Intriguingly, similar observations can be made about function point use in academic research.

There are also issues related to the function point metric. Fundamentally, the meaning of function point and the derivation algorithm and its rationale may need more research and more theoretical groundwork. There are also many variations in counting function points in the industry and several major methods other than the IFPUG standard. In 1983, Symons presented a function point variant that he termed the Mark II function point (Symons, 1991). According to Jones (2000), the Mark II function point is now widely used in the United Kingdom and to a lesser degree in Hong Kong and Canada. Some of the minor function point variants include feature points, 3D function points, and full function points. In all, based on the comprehensive software benchmark work by Jones (2000), the set of function point variants now include at least 25 functional metrics. Function point counting can be time-consuming and expensive, and accurate counting requires certified function point specialists. Nonetheless, function point metrics are apparently more robust than LOC-based data with regard to comparisons across organizations, especially studies involving multiple languages and those for productivity evaluation.

Example: Function Point Defect Rates

In 2000, based on a large body of empirical studies, Jones published the book *Software Assessments, Benchmarks, and Best Practices.* All metrics used throughout the book are based on function points. According to his study (1997), the average number of software defects in the U.S. is approximately 5 per function point during the entire software life cycle. This number represents the total number of defects found and measured from early software requirements throughout the life cycle of the software, including the defects reported by users in the field. Jones also estimates the defect removal efficiency of software organizations by level of the capability maturity model (CMM) developed by the Software Engineering Institute (SEI). By applying the defect removal efficiency to the overall defect rate per function point, the following defect rates for the delivered software were estimated. The time frames for these defect rates were not specified, but it appears that these defect rates are for the maintenance life of the software. The estimated defect rates per function point are as follows:

- SEI CMM Level 1: 0.75
- SEI CMM Level 2: 0.44
- SEI CMM Level 3: 0.27
- SEI CMM Level 4: 0.14
- SEI CMM Level 5: 0.05

4.1.2 Customer Problems Metric

Another product quality metric used by major developers in the software industry measures the problems customers encounter when using the product. For the defect rate metric, the numerator is the number of valid defects. However, from the customers' standpoint, all problems they encounter while using the software product, not just the valid defects, are problems with the software. Problems that are not valid defects may be usability problems, unclear documentation or information, duplicates of valid defects (defects that were reported by other customers and fixes were available but the current customers did not know of them), or even user errors. These so-called non-defect-oriented problems, together with the defect problems, constitute the total problem space of the software from the customers' perspective.

The problems metric is usually expressed in terms of problems per user month (PUM):

PUM = Total problems that customers reported (true defects and non-defect-oriented problems) for a time period
÷ Total number of license-months of the software during the period

where

Number of license-months = Number of install licenses of the software
× Number of months in the calculation period

PUM is usually calculated for each month after the software is released to the market, and also for monthly averages by year. Note that the denominator is the number of license-months instead of thousand lines of code or function point, and the numerator is all problems customers encountered. Basically, this metric relates problems to usage. Approaches to achieve a low PUM include:

- Improve the development process and reduce the product defects.
- Reduce the non-defect-oriented problems by improving all aspects of the products (such as usability, documentation), customer education, and support.
- Increase the sale (the number of installed licenses) of the product.

The first two approaches reduce the numerator of the PUM metric, and the third increases the denominator. The result of any of these courses of action will be that the PUM metric has a lower value. All three approaches make good sense for quality improvement and business goals for any organization. The PUM metric, therefore, is a good metric. The only minor drawback is that when the business is in excellent condition and the number of software licenses is rapidly increasing, the PUM metric will look extraordinarily good (low value) and, hence, the need to continue to reduce the number of customers' problems (the numerator of the metric) may be undermined. Therefore, the total number of customer problems should also be monitored and aggressive year-to-year or release-to-release improvement goals set as the number of installed licenses increases. However, unlike valid code defects, customer problems are not totally under the control of the software development organization. Therefore, it may not be feasible to set a PUM goal that the total customer problems cannot increase from release to release, especially when the sales of the software are increasing.

The key points of the defect rate metric and the customer problems metric are briefly summarized in Table 4.1. The two metrics represent two perspectives of product quality. For each metric the numerator and denominator match each other well: Defects relate to source instructions or the number of function points, and problems relate to usage of the product. If the numerator and denominator are mixed up, poor metrics will result. Such metrics could be counterproductive to an organization's quality improvement effort because they will cause confusion and wasted resources.

The customer problems metric can be regarded as an intermediate measurement between defects measurement and customer satisfaction. To reduce customer problems, one has to reduce the functional defects in the products and, in addition, improve other factors (usability, documentation, problem rediscovery, etc.). To improve

TABLE 4.1
Defect Rate and Customer Problems Metrics

	Defect Rate	Problems per User-Month (PUM)
Numerator	Valid and unique product defects	All customer problems (defects and nondefects, first time and repeated)
Denominator	Size of product (KLOC or function point)	Customer usage of the product (user-months)
Measurement perspective	Producer—software development organization	Customer
Scope	Intrinsic product quality	Intrinsic product quality plus other factors

customer satisfaction, one has to reduce defects and overall problems and, in addition, manage factors of broader scope such as timing and availability of the product, company image, services, total customer solutions, and so forth. From the software quality standpoint, the relationship of the scopes of the three metrics can be represented by the Venn diagram in Figure 4.1.

4.1.3 Customer Satisfaction Metrics

Customer satisfaction is often measured by customer survey data via the five-point scale:

- ☐ Very satisfied
- ☐ Satisfied
- ☐ Neutral
- ☐ Dissatisfied
- ☐ Very dissatisfied.

Satisfaction with the overall quality of the product and its specific dimensions is usually obtained through various methods of customer surveys. For example, the specific parameters of customer satisfaction in software monitored by IBM include the CUPRIMDSO categories (capability, functionality, usability, performance, reliability, installability, maintainability, documentation/information, service, and overall); for Hewlett-Packard they are FURPS (functionality, usability, reliability, performance, and service).

Based on the five-point-scale data, several metrics with slight variations can be constructed and used, depending on the purpose of analysis. For example:

(1) Percent of completely satisfied customers

(2) Percent of satisfied customers (satisfied and completely satisfied)

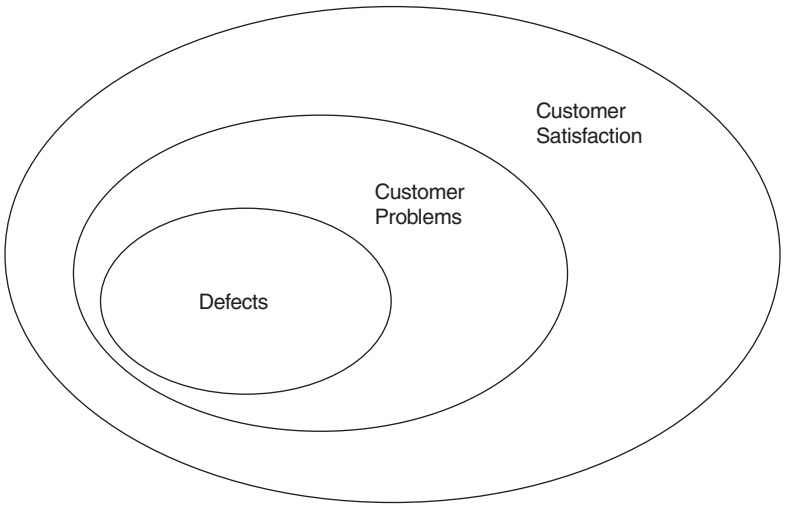

FIGURE 4.1
Scopes of Three Quality Metrics

(3) Percent of dissatisfied customers (dissatisfied and completely dissatisfied)
(4) Percent of nonsatisfied (neutral, dissatisfied, and completely dissatisfied)

Usually the second metric, percent satisfaction, is used. In practices that focus on reducing the percentage of nonsatisfaction, much like reducing product defects, metric (4) is used.

In addition to forming percentages for various satisfaction or dissatisfaction categories, the weighted index approach can be used. For instance, some companies use the *net satisfaction index* (*NSI*) to facilitate comparisons across product. The NSI has the following weighting factors:

- ☐ Completely satisfied = 100%
- ☐ Satisfied = 75%
- ☐ Neutral = 50%
- ☐ Dissatisfied = 25%
- ☐ Completely dissatisfied = 0%

NSI ranges from 0% (all customers are completely dissatisfied) to 100% (all customers are completely satisfied). If all customers are satisfied (but not completely satisfied), NSI will have a value of 75%. This weighting approach, however, may be masking the satisfaction profile of one's customer set. For example, if half of the

customers are completely satisfied and half are neutral, NSI's value is also 75%, which is equivalent to the scenario that all customers are satisfied. If satisfaction is a good indicator of product loyalty, then half completely satisfied and half neutral is certainly less positive than all satisfied. Furthermore, we are not sure of the rationale behind giving a 25% weight to those who are dissatisfied. Therefore, this example of NSI is not a good metric; it is inferior to the simple approach of calculating percentage of specific categories. If the entire satisfaction profile is desired, one can simply show the percent distribution of all categories via a histogram. A weighted index is for data summary when multiple indicators are too cumbersome to be shown. For example, if customers' purchase decisions can be expressed as a function of their satisfaction with specific dimensions of a product, then a purchase decision index could be useful. In contrast, if simple indicators can do the job, then the weighted index approach should be avoided.

4.2 In-Process Quality Metrics

Because our goal is to understand the programming process and to learn to engineer quality into the process, in-process quality metrics play an important role. In-process quality metrics are less formally defined than end-product metrics, and their practices vary greatly among software developers. On the one hand, in-process quality metrics simply means tracking defect arrival during formal machine testing for some organizations. On the other hand, some software organizations with well-established software metrics programs cover various parameters in each phase of the development cycle. In this section we briefly discuss several metrics that are basic to sound in-process quality management. In later chapters on modeling we will examine some of them in greater detail and discuss others within the context of models.

4.2.1 Defect Density During Machine Testing

Defect rate during formal machine testing (testing after code is integrated into the system library) is usually positively correlated with the defect rate in the field. Higher defect rates found during testing is an indicator that the software has experienced higher error injection during its development process, unless the higher testing defect rate is due to an extraordinary testing effort—for example, additional testing or a new testing approach that was deemed more effective in detecting defects. The rationale for the positive correlation is simple: Software defect density never follows the uniform distribution. If a piece of code or a product has higher testing defects, it is a result of more effective testing or it is because of higher latent defects in the code. Myers (1979) discusses a counterintuitive principle that the more defects found during testing, the more defects will be found later. That principle is another expres-

sion of the positive correlation between defect rates during testing and in the field or between defect rates between phases of testing.

This simple metric of defects per KLOC or function point, therefore, is a good indicator of quality while the software is still being tested. It is especially useful to monitor subsequent releases of a product in the same development organization. Therefore, release-to-release comparisons are not contaminated by extraneous factors. The development team or the project manager can use the following scenarios to judge the release quality:

- If the defect rate during testing is the same or lower than that of the previous release (or a similar product), then ask: Does the testing for the current release deteriorate?
  - If the answer is no, the quality perspective is positive.
  - If the answer is yes, you need to do extra testing (e.g., add test cases to increase coverage, blitz test, customer testing, stress testing, etc.).
- If the defect rate during testing is substantially higher than that of the previous release (or a similar product), then ask: Did we plan for and actually improve testing effectiveness?
  - If the answer is no, the quality perspective is negative. Ironically, the only remedial approach that can be taken at this stage of the life cycle is to do more testing, which will yield even higher defect rates.
  - If the answer is yes, then the quality perspective is the same or positive.

4.2.2 Defect Arrival Pattern During Machine Testing

Overall defect density during testing is a summary indicator. The pattern of defect arrivals (or for that matter, times between failures) gives more information. Even with the same overall defect rate during testing, different patterns of defect arrivals indicate different quality levels in the field. Figure 4.2 shows two contrasting patterns for both the defect arrival rate and the cumulative defect rate. Data were plotted from 44 weeks before code-freeze until the week prior to code-freeze. The second pattern, represented by the charts on the right side, obviously indicates that testing started late, the test suite was not sufficient, and that the testing ended prematurely.

The objective is always to look for defect arrivals that stabilize at a very low level, or times between failures that are far apart, before ending the testing effort and releasing the software to the field. Such declining patterns of defect arrival during testing are indeed the basic assumption of many software reliability models. The time unit for observing the arrival pattern is usually weeks and occasionally months. For reliability models that require execution time data, the time interval is in units of CPU time.

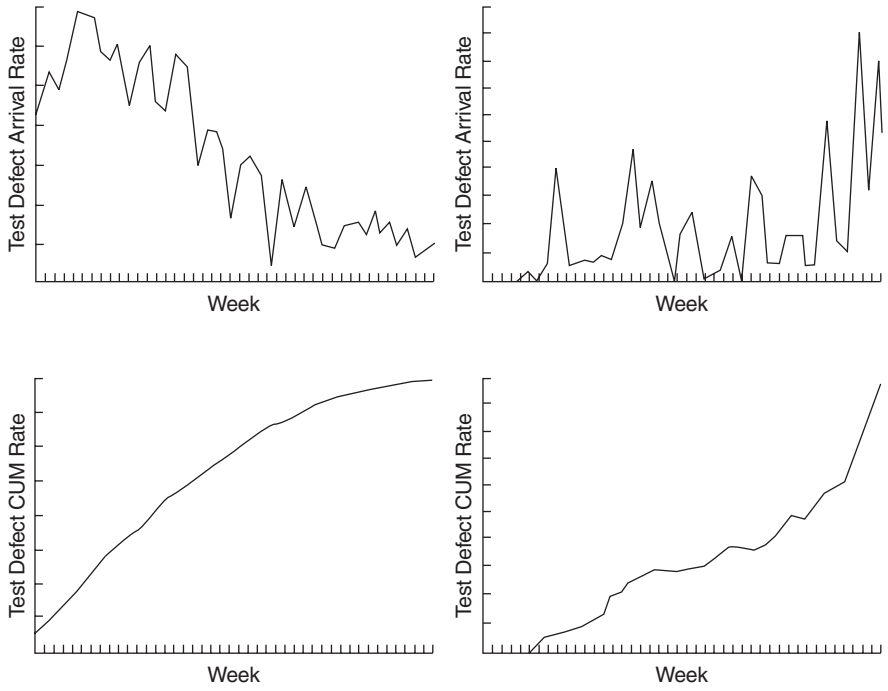

FIGURE 4.2
Two Contrasting Defect Arrival Patterns During Testing

When we talk about the defect arrival pattern during testing, there are actually three slightly different metrics, which should be looked at simultaneously:

- The defect arrivals (defects reported) during the testing phase by time interval (e.g., week). These are the raw number of arrivals, not all of which are valid defects.
- The pattern of valid defect arrivals—when problem determination is done on the reported problems. This is the true defect pattern.
- The pattern of defect backlog overtime. This metric is needed because development organizations cannot investigate and fix all reported problems immediately. This metric is a workload statement as well as a quality statement. If the defect backlog is large at the end of the development cycle and a lot of fixes have yet to be integrated into the system, the stability of the system (hence its quality) will be affected. Retesting (regression test) is needed to ensure that targeted product quality levels are reached.

4.2.3 Phase-Based Defect Removal Pattern

The phase-based defect removal pattern is an extension of the test defect density metric. In addition to testing, it requires the tracking of defects at all phases of the development cycle, including the design reviews, code inspections, and formal verifications before testing. Because a large percentage of programming defects is related to design problems, conducting formal reviews or functional verifications to enhance the defect removal capability of the process at the front end reduces error injection. The pattern of phase-based defect removal reflects the overall defect removal ability of the development process.

With regard to the metrics for the design and coding phases, in addition to defect rates, many development organizations use metrics such as inspection coverage and inspection effort for in-process quality management. Some companies even set up "model values" and "control boundaries" for various in-process quality indicators. For example, Cusumano (1992) reports the specific model values and control boundaries for metrics such as review coverage rate, review manpower rate (review work hours/number of design work hours), defect rate, and so forth, which were used by NEC's Switching Systems Division.

Figure 4.3 shows the patterns of defect removal of two development projects: project A was front-end loaded and project B was heavily testing-dependent for removing defects. In the figure, the various phases of defect removal are high-level design review (I0), low-level design review (I1), code inspection (I2), unit test (UT), component test (CT), and system test (ST). As expected, the field quality of project A outperformed project B significantly.

4.2.4 Defect Removal Effectiveness

Defect removal effectiveness (or efficiency, as used by some writers) can be defined as follows:

$$\text{DRE} = \frac{\text{Defects removed during a development phase}}{\text{Defects latent in the product}} \times 100\%$$

Because the total number of latent defects in the product at any given phase is not known, the denominator of the metric can only be approximated. It is usually estimated by:

$$\text{Defects removed during the phase} + \text{defects found later}$$

The metric can be calculated for the entire development process, for the front end (before code integration), and for each phase. It is called *early defect removal*

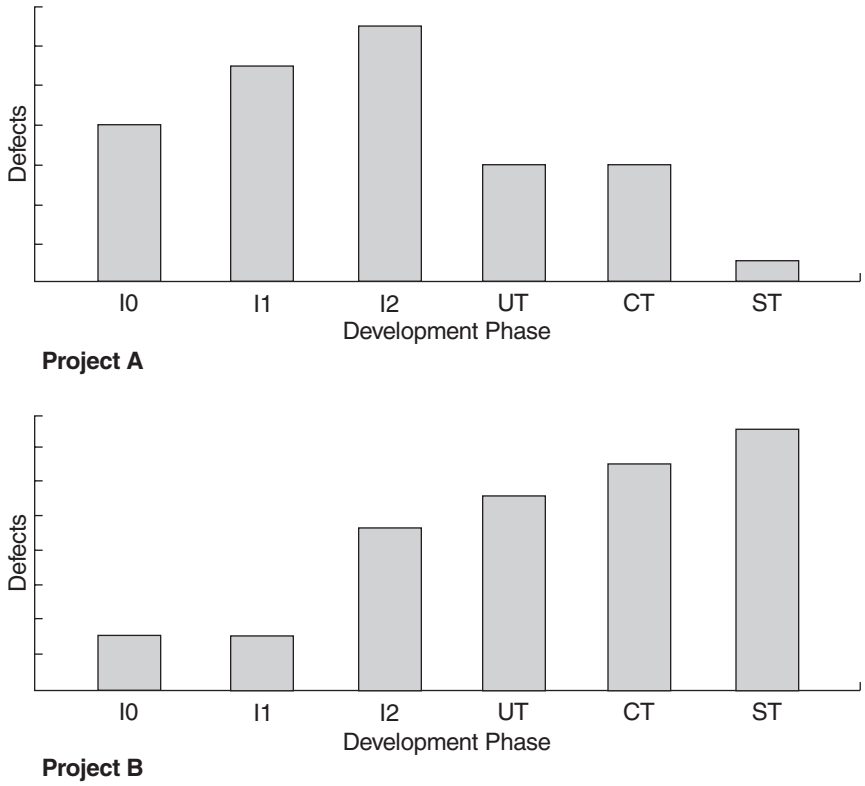

FIGURE 4.3
Defect Removal by Phase for Two Products

and *phase effectiveness* when used for the front end and for specific phases, respectively. The higher the value of the metric, the more effective the development process and the fewer the defects escape to the next phase or to the field. This metric is a key concept of the defect removal model for software development. (In Chapter 6 we give this subject a detailed treatment.) Figure 4.4 shows the DRE by phase for a real software project. The weakest phases were unit test (UT), code inspections (I2), and component test (CT). Based on this metric, action plans to improve the effectiveness of these phases were established and deployed.

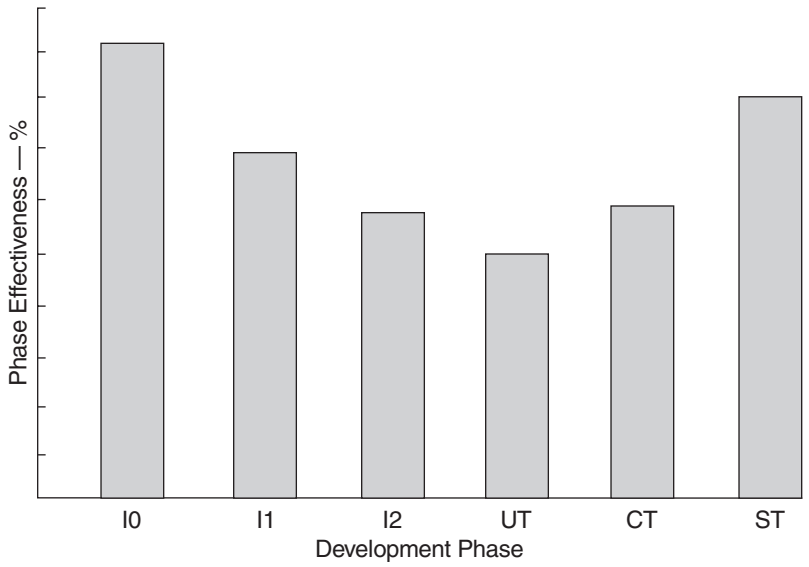

FIGURE 4.4
Phase Effectiveness of a Software Project

4.3 Metrics for Software Maintenance

When development of a software product is complete and it is released to the market, it enters the maintenance phase of its life cycle. During this phase the defect arrivals by time interval and customer problem calls (which may or may not be defects) by time interval are the de facto metrics. However, the number of defect or problem arrivals is largely determined by the development process before the maintenance phase. Not much can be done to alter the quality of the product during this phase. Therefore, these two de facto metrics, although important, do not reflect the quality of software maintenance. What can be done during the maintenance phase is to fix the defects as soon as possible and with excellent fix quality. Such actions, although still not able to improve the defect rate of the product, can improve customer satisfaction to a large extent. The following metrics are therefore very important:

- ☐ Fix backlog and backlog management index
- ☐ Fix response time and fix responsiveness
- ☐ Percent delinquent fixes
- ☐ Fix quality

4.3.1 Fix Backlog and Backlog Management Index

Fix backlog is a workload statement for software maintenance. It is related to both the rate of defect arrivals and the rate at which fixes for reported problems become available. It is a simple count of reported problems that remain at the end of each month or each week. Using it in the format of a trend chart, this metric can provide meaningful information for managing the maintenance process. Another metric to manage the backlog of open, unresolved, problems is the backlog management index (BMI).

$$\text{BMI} = \frac{\text{Number of problems closed during the month}}{\text{Number of problem arrivals during the month}} \times 100\%$$

As a ratio of number of closed, or solved, problems to number of problem arrivals during the month, if BMI is larger than 100, it means the backlog is reduced. If BMI is less than 100, then the backlog increased. With enough data points, the techniques of control charting can be used to calculate the backlog management capability of the maintenance process. More investigation and analysis should be triggered when the value of BMI exceeds the control limits. Of course, the goal is always to strive for a BMI larger than 100. A BMI trend chart or control chart should be examined together with trend charts of defect arrivals, defects fixed (closed), and the number of problems in the backlog.

Figure 4.5 is a trend chart by month of the numbers of opened and closed problems of a software product, and a pseudo-control chart for the BMI. The latest release of the product was available to customers in the month for the first data points on the two charts. This explains the rise and fall of the problem arrivals and closures. The mean BMI was 102.9%, indicating that the capability of the fix process was functioning normally. All BMI values were within the upper (UCL) and lower (LCL) control limits—the backlog management process was in control. (*Note:* We call the BMI chart a pseudo-control chart because the BMI data are autocorrelated and therefore the assumption of independence for control charts is violated. Despite not being "real" control charts in statistical terms, however, we found pseudo-control charts such as the BMI chart quite useful in software quality management. In Chapter 5 we provide more discussions and examples.)

A variation of the problem backlog index is the ratio of number of opened problems (problem backlog) to number of problem arrivals during the month. If the index is 1, that means the team maintains a backlog the same as the problem arrival rate. If the index is below 1, that means the team is fixing problems faster than the problem arrival rate. If the index is higher than 1, that means the team is losing ground in their problem-fixing capability relative to problem arrivals. Therefore, this variant index is also a statement of fix responsiveness.

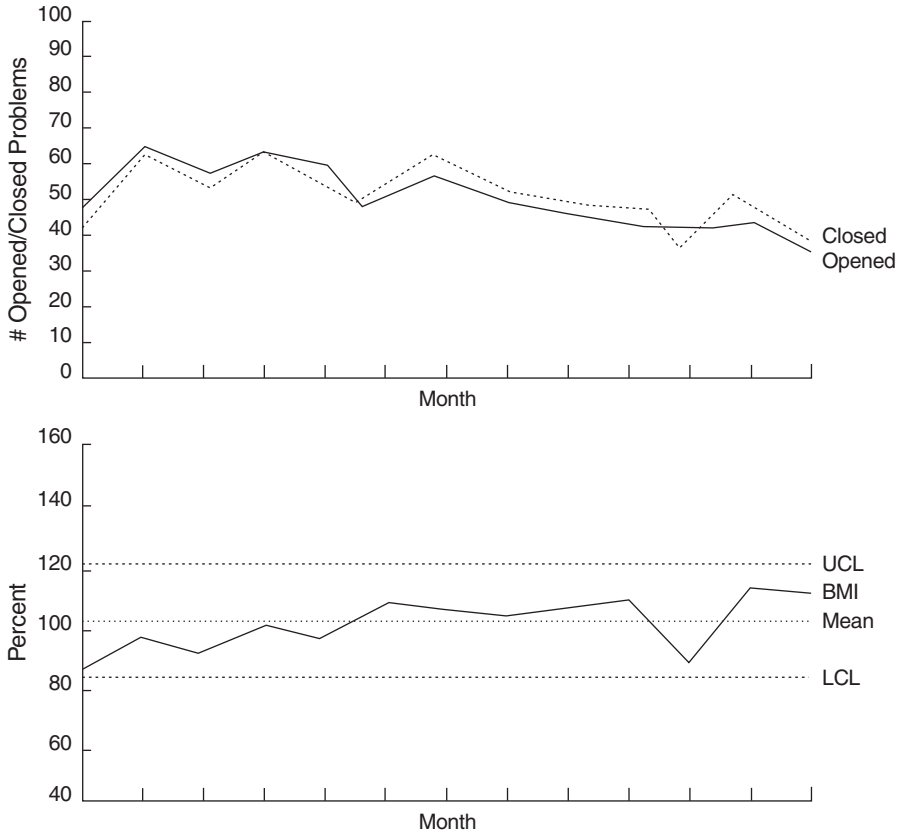

FIGURE 4.5
Opened Problems, Closed Problems, and Backlog Management Index by Month

4.3.2 Fix Response Time and Fix Responsiveness

For many software development organizations, guidelines are established on the time limit within which the fixes should be available for the reported defects. Usually the criteria are set in accordance with the severity of the problems. For the critical situations in which the customers' businesses are at risk due to defects in the software product, software developers or the software change teams work around the clock to fix the problems. For less severe defects for which circumventions are available, the required fix response time is more relaxed. The fix response time metric is usually calculated as follows for all problems as well as by severity level:

Mean time of all problems from open to closed

If there are data points with extreme values, medians should be used instead of mean. Such cases could occur for less severe problems for which customers may be satisfied with the circumvention and didn't demand a fix. Therefore, the problem may remain open for a long time in the tracking report.

In general, short fix response time leads to customer satisfaction. However, there is a subtle difference between fix responsiveness and short fix response time. From the customer's perspective, the use of averages may mask individual differences. The important elements of fix responsiveness are customer expectations, the agreed-to fix time, and the ability to meet one's commitment to the customer. For example, John takes his car to the dealer for servicing in the early morning and needs it back by noon. If the dealer promises noon but does not get the car ready until 2 o'clock, John will not be a satisfied customer. On the other hand, Julia does not need her mini van back until she gets off from work, around 6 P.M. As long as the dealer finishes servicing her van by then, Julia is a satisfied customer. If the dealer leaves a timely phone message on her answering machine at work saying that her van is ready to pick up, Julia will be even more satisfied. This type of fix responsiveness process is indeed being practiced by automobile dealers who focus on customer satisfaction.

In this writer's knowledge, the systems software development of Hewlett-Packard (HP) in California and IBM Rochester's systems software development have fix responsiveness processes similar to the process just illustrated by the automobile examples. In fact, IBM Rochester's practice originated from a benchmarking exchange with HP some years ago. The metric for IBM Rochester's fix responsiveness is operationalized as percentage of delivered fixes meeting committed dates to customers.

4.3.3 Percent Delinquent Fixes

The mean (or median) response time metric is a central tendency measure. A more sensitive metric is the percentage of delinquent fixes. For each fix, if the turnaround time greatly exceeds the required response time, then it is classified as delinquent:

$$\text{Percent delinquent fixes} = \frac{\text{Number of fixes that exceeded the response time criteria by severity level}}{\text{Number of fixes delivered in a specified time}} \times 100\%$$

This metric, however, is not a metric for real-time delinquent management because it is for closed problems only. Problems that are still open must be factored into the calculation for a real-time metric. Assuming the time unit is 1 week, we propose that the percent delinquent of problems in the active backlog be used. *Active backlog* refers to all opened problems for the week, which is the sum of the existing backlog at the

beginning of the week and new problem arrivals during the week. In other words, it contains the total number of problems to be processed for the week—the total workload. The number of delinquent problems is checked at the end of the week. Figure 4.6 shows the real-time delivery index diagrammatically.

It is important to note that the metric of percent delinquent fixes is a cohort metric. Its denominator refers to a cohort of problems (problems closed in a given period of time, or problems to be processed in a given week). The cohort concept is important because if it is operationalized as a cross-sectional measure, then invalid metrics will result. For example, we have seen practices in which at the end of each week the number of problems in backlog (problems still to be fixed) and the number of delinquent open problems were counted, and the percent delinquent problems was calculated. This cross-sectional counting approach neglects problems that were processed and closed before the end of the week, and will create a high delinquent index when significant improvement (reduction in problems backlog) is made.

4.3.4 Fix Quality

Fix quality or the number of defective fixes is another important quality metric for the maintenance phase. From the customer's perspective, it is bad enough to encounter functional defects when running a business on the software. It is even worse if the fixes turn out to be defective. A fix is defective if it did not fix the reported problem, or if it fixed the original problem but injected a new defect. For mission-critical software, defective fixes are detrimental to customer satisfaction.

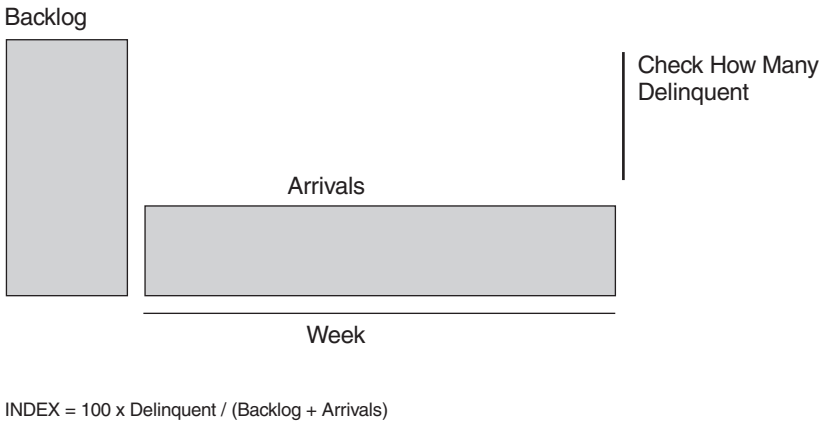

FIGURE 4.6
Real-Time Delinquency Index

The metric of percent defective fixes is simply the percentage of all fixes in a time interval (e.g., 1 month) that are defective.

A defective fix can be recorded in two ways: Record it in the month it was discovered or record it in the month the fix was delivered. The first is a customer measure, the second is a process measure. The difference between the two dates is the latent period of the defective fix. It is meaningful to keep track of the latency data and other information such as the number of customers who were affected by the defective fix. Usually the longer the latency, the more customers are affected because there is more time for customers to apply that defective fix to their software system.

There is an argument against using percentage for defective fixes. If the number of defects, and therefore the fixes, is large, then the small value of the percentage metric will show an optimistic picture, although the number of defective fixes could be quite large. This metric, therefore, should be a straight count of the number of defective fixes. The quality goal for the maintenance process, of course, is zero defective fixes without delinquency.

4.4 Examples of Metrics Programs

4.4.1 Motorola

Motorola's software metrics program is well articulated by Daskalantonakis (1992). By following the Goal/Question/Metric paradigm of Basili and Weiss (1984), goals were identified, questions were formulated in quantifiable terms, and metrics were established. The goals and measurement areas identified by the Motorola Quality Policy for Software Development (QPSD) are listed in the following.

Goals

- ☐ Goal 1: Improve project planning.
- ☐ Goal 2: Increase defect containment.
- ☐ Goal 3: Increase software reliability.
- ☐ Goal 4: Decrease software defect density.
- ☐ Goal 5: Improve customer service.
- ☐ Goal 6: Reduce the cost of nonconformance.
- ☐ Goal 7: Increase software productivity.

Measurement Areas

- ☐ Delivered defects and delivered defects per size
- ☐ Total effectiveness throughout the process
- ☐ Adherence to schedule
- ☐ Accuracy of estimates
- ☐ Number of open customer problems

- ☐ Time that problems remain open
- ☐ Cost of nonconformance
- ☐ Software reliability

For each goal the questions to be asked and the corresponding metrics were also formulated. In the following, we list the questions and metrics for each goal:[1]

Goal 1: Improve Project Planning

Question 1.1: What was the accuracy of estimating the actual value of project schedule?

Metric 1.1 : Schedule Estimation Accuracy (SEA)

$$\text{SEA} = \frac{\text{Actual project duration}}{\text{Estimated project duration}}$$

Question 1.2: What was the accuracy of estimating the actual value of project effort?

Metric 1.2 : Effort Estimation Accuracy (EEA)

$$\text{EEA} = \frac{\text{Actual project effort}}{\text{Estimated project effort}}$$

Goal 2: Increase Defect Containment

Question 2.1: What is the currently known effectiveness of the defect detection process prior to release?

Metric 2.1: Total Defect Containment Effectiveness (TDCE)

$$\text{TDCE} = \frac{\text{Number of prerelease defects}}{\text{Number of prerelease defects} + \text{Number of postrelease defects}}$$

Question 2.2: What is the currently known containment effectiveness of faults introduced during each constructive phase of software development for a particular software product?

Metric 2.2: Phase Containment Effectiveness for phase *i* (PCEi)

1. *Source:* From Daskalantonakis, M. K., "A Practical View of Software Measurement and Implementation Experiences Within Motorola (1001–1004)," *IEEE Transactions on Software Engineering,* Vol. 18, No. 11 (November 1992): 998–1010. Copyright © IEEE, 1992. Used with permission to reprint.

$$\text{PCEi} = \frac{\text{Number of phase } i \text{ errors}}{\text{Number of phase } i \text{ errors} + \text{Number of phase } i \text{ defects}}$$

Note: From Daskalantonakis's definition of *error* and *defect*, it appears that Motorola's use of the two terms differs from what was discussed earlier in this chapter. To understand the preceding metric, consider Daskalantonakis's definitions:

- *Error:* A problem found during the review of the phase where it was introduced.
- *Defect:* A problem found later than the review of the phase where it was introduced.
- *Fault:* Both errors and defects are considered faults.

Goal 3: Increase Software Reliability

Question 3.1: What is the rate of software failures, and how does it change over time?

Metric 3.1: Failure Rate (FR)

$$\text{FR} = \frac{\text{Number of failures}}{\text{Execution time}}$$

Goal 4: Decrease Software Defect Density

Question 4.1: What is the normalized number of in-process faults, and how does it compare with the number of in-process defects?

Metric 4.1a: In-process Faults (IPF)

$$\text{IPF} = \frac{\text{In-process faults caused by incremental software development}}{\text{Assembly-equivalent delta source size}}$$

Metric 4.1b: In-process Defects (IPD)

$$\text{IPF} = \frac{\text{In-process defects caused by incremental software development}}{\text{Assembly-equivalent delta source size}}$$

Question 4.2: What is the currently known defect content of software delivered to customers, normalized by Assembly-equivalent size?

Metric 4.2a: Total Released Defects (TRD) total

$$\text{TRD total} = \frac{\text{Number of released defects}}{\text{Assembly-equivalent total source size}}$$

Metric 4.2b: Total Released Defects (TRD) delta

$$\text{TRD delta} = \frac{\text{Number of released defects caused by incremental software development}}{\text{Assembly-equivalent total source size}}$$

Question 4.3: What is the currently known customer-found defect content of software delivered to customers, normalized by Assembly-equivalent source size?

Metric 4.3a: Customer-Found Defects (CFD) total

$$\text{CFD total} = \frac{\text{Number of customer -found defects}}{\text{Assembly-equivalent total source size}}$$

Metric 4.3b: Customer-Found Defects (CFD) delta

$$\text{CFD delta} = \frac{\text{Number of customer -found defects caused by incremental software development}}{\text{Assembly-equivalent total source size}}$$

Goal 5: Improve Customer Service

Question 5.1 What is the number of new problems opened during the month?

Metric 5.1: New Open Problems (NOP)

NOP = Total new postrelease problems opened during the month

Question 5.2 What is the total number of open problems at the end of the month?

Metric 5.2: Total Open Problems (TOP)

TOP = Total postrelease problems that remain open at the end of the month

Question 5.3: What is the mean age of open problems at the end of the month?

Metric 5.3: Mean Age of Open Problems (AOP)

AOP = (Total time postrelease problems remaining open at the end of the month have been open)/(Number of open post release problems remaining open at the end of the month)

Question 5.4: What is the mean age of the problems that were closed during the month?

Metric 5.4: Mean Age of Closed Problems (ACP)

ACP = (Total time postrelease problems closed within the month were open)/(Number of open postrelease problems closed within the month)

Goal 6: Reduce the Cost of Nonconformance

Question 6.1: What was the cost to fix postrelease problems during the month?

Metric 6.1: Cost of Fixing Problems (CFP)

CFP = Dollar cost associated with fixing postrelease problems within the month

Goal 7: Increase Software Productivity

Question 7.1: What was the productivity of software development projects (based on source size)?

Metric 7.1a: Software Productivity total (SP total)

$$\text{SP total} = \frac{\text{Assembly-equivalent total source size}}{\text{Software development effort}}$$

Metric 7.1b: Software Productivity delta (SP delta)

$$\text{SP delta} = \frac{\text{Assembly-equivalent delta source size}}{\text{Software development effort}}$$

From the preceding goals one can see that metrics 3.1, 4.2a, 4.2b, 4.3a, and 4.3b are metrics for end-product quality, metrics 5.1 through 5.4 are metrics for software maintenance, and metrics 2.1, 2.2, 4.1a, and 4.1b are in-process quality metrics. The others are for scheduling, estimation, and productivity.

In addition to the preceding metrics, which are defined by the Motorola Software Engineering Process Group (SEPG), Daskalantonakis describes in-process metrics that can be used for schedule, project, and quality control. Without getting into too many details, we list these additional in-process metrics in the following. [For details and other information about Motorola's software metrics program, see Daskalantonakis's original article (1992).] Items 1 through 4 are for project status/control and items 5 through 7 are really in-process quality metrics that can provide information about the status of the project and lead to possible actions for further quality improvement.

1. *Life-cycle phase and schedule tracking metric:* Track schedule based on life-cycle phase and compare actual to plan.

2. *Cost/earned value tracking metric:* Track actual cumulative cost of the project versus budgeted cost, and actual cost of the project so far, with continuous update throughout the project.
3. *Requirements tracking metric:* Track the number of requirements change at the project level.
4. *Design tracking metric:* Track the number of requirements implemented in design versus the number of requirements written.
5. *Fault-type tracking metric:* Track causes of faults.
6. *Remaining defect metrics:* Track faults per month for the project and use Rayleigh curve to project the number of faults in the months ahead during development.
7. *Review effectiveness metric:* Track error density by stages of review and use control chart methods to flag the exceptionally high or low data points.

4.4.2 Hewlett-Packard

Grady and Caswell (1986) offer a good description of Hewlett-Packard's software metric program, including both the primitive metrics and computed metrics that are widely used at HP. Primitive metrics are those that are directly measurable and accountable such as control token, data token, defect, total operands, LOC, and so forth. Computed metrics are metrics that are mathematical combinations of two or more primitive metrics. The following is an excerpt of HP's computed metrics:[2]

> Average fixed defects/working day: self-explanatory.
>
> Average engineering hours/fixed defect: self-explanatory.
>
> Average reported defects/working day: self-explanatory.
>
> Bang: "A quantitative indicator of net usable function from the user's point of view" (DeMarco, 1982). There are two methods for computing Bang. Computation of Bang for function-strong systems involves counting the tokens entering and leaving the function multiplied by the weight of the function. For data-strong systems it involves counting the objects in the database weighted by the number of relationships of which the object is a member.
>
> Branches covered/total branches: When running a program, this metric indicates what percentage of the decision points were actually executed.
>
> Defects/KNCSS: Self-explanatory (KNCSS—Thousand noncomment source statements).
>
> Defects/LOD: Self-explanatory (LOD—Lines of documentation not included in program source code).
>
> Defects/testing time: Self-explanatory.

1. *Source:* Grady, R. B., and D. L. Caswell, *Software Metrics: Establishing A Company-Wide Program,* pp. 225–226. Englewood Cliffs, N.J.: Prentice-Hall. Copyright © 1986 Prentice-Hall, Inc. Used with permission to reprint.

Design weight: "Design weight is a simple sum of the module weights over the set of all modules in the design" (DeMarco, 1982). Each module weight is a function of the token count associated with the module and the expected number of decision counts which are based on the structure of data.

NCSS/engineering month: Self-explanatory.

Percent overtime: Average overtime/40 hours per week.

Phase: engineering months/total engineering months: Self-explanatory.

Of these metrics, defects/KNCSS and defects/LOD are end-product quality metrics. Defects/testing time is a statement of testing effectiveness, and branches covered/total branches is testing coverage in terms of decision points. Therefore, both are meaningful in-process quality metrics. Bang is a measurement of functions and NCSS/engineering month is a productivity measure. Design weight is an interesting measurement but its use is not clear. The other metrics are for workload, schedule, project control, and cost of defects.

As Grady and Caswell point out, this list represents the most widely used computed metrics at HP, but it may not be comprehensive. For instance, many others are discussed in other sections of their book. For example, customer satisfaction measurements in relation to software quality attributes are a key area in HP's software metrics. As mentioned earlier in this chapter, the software quality attributes defined by HP are called FURPS (functionality, usability, reliability, performance, and supportability). Goals and objectives for FURPS are set for software projects. Furthermore, to achieve the FURPS goals of the end product, measurable objectives using FURPS for each life-cycle phase are also set (Grady and Caswell, 1986, pp. 159–162).

MacLeod (1993) describes the implementation and sustenance of a software inspection program in an HP division. The metrics used include average hours per inspection, average defects per inspection, average hours per defect, and defect causes. These inspection metrics, used appropriately in the proper context (e.g., comparing the current project with previous projects), can be used to monitor the inspection phase (front end) of the software development process.

4.4.3 IBM Rochester

Because many examples of the metrics used at IBM Rochester have already been discussed or will be elaborated on later, here we give just an overview. Furthermore, we list only selected quality metrics; metrics related to project management, productivity, scheduling, costs, and resources are not included.

- Overall customer satisfaction as well as satisfaction with various quality attributes such as CUPRIMDS (capability, usability, performance, reliability, install, maintenance, documentation/information, and service).

- ☐ Postrelease defect rates such as those discussed in section 4.1.1.
- ☐ Customer problem calls per month
- ☐ Fix response time
- ☐ Number of defective fixes
- ☐ Backlog management index
- ☐ Postrelease arrival patterns for defects and problems (both defects and non-defect-oriented problems)
- ☐ Defect removal model for the software development process
- ☐ Phase effectiveness (for each phase of inspection and testing)
- ☐ Inspection coverage and effort
- ☐ Compile failures and build/integration defects
- ☐ Weekly defect arrivals and backlog during testing
- ☐ Defect severity
- ☐ Defect cause and problem component analysis
- ☐ Reliability: mean time to initial program loading (IPL) during testing
- ☐ Stress level of the system during testing as measured in level of CPU use in terms of number of CPU hours per system per day during stress testing
- ☐ Number of system crashes and hangs during stress testing and system testing
- ☐ Models for postrelease defect estimation
- ☐ Various customer feedback metrics at the end of the development cycle before the product is shipped
- ☐ S curves for project progress comparing actual to plan for each phase of development such as number of inspections conducted by week, LOC integrated by week, number of test cases attempted and succeeded by week, and so forth.

4.5 Collecting Software Engineering Data

The challenge of collecting software engineering data is to make sure that the collected data can provide useful information for project, process, and quality management and, at the same time, that the data collection process will not be a burden on development teams. Therefore, it is important to consider carefully what data to collect. The data must be based on well-defined metrics and models, which are used to drive improvements. Therefore, the goals of the data collection should be established and the questions of interest should be defined before any data is collected. Data classification schemes to be used and the level of precision must be carefully specified. The collection form or template and data fields should be pretested. The amount of data to be collected and the number of metrics to be used need not be overwhelming. It is more important that the information extracted from the data be focused, accurate, and useful than that it be plentiful. Without being metrics driven, overcollection of data could be wasteful. Overcollection of data is quite common when people start to measure software without an a priori specification of purpose, objectives, profound versus trivial issues, and metrics and models.

Gathering software engineering data can be expensive, especially if it is done as part of a research program, For example, the NASA Software Engineering Laboratory spent about 15% of their development costs on gathering and processing data on hundreds of metrics for a number of projects (Shooman, 1983). For large commercial development organizations, the relative cost of data gathering and processing should be much lower because of economy of scale and fewer metrics. However, the cost of data collection will never be insignificant. Nonetheless, data collection and analysis, which yields intelligence about the project and the development process, is vital for business success. Indeed, in many organizations, a tracking and data collection system is often an integral part of the software configuration or the project management system, without which the chance of success of large and complex projects will be reduced.

Basili and Weiss (1984) propose a data collection methodology that could be applicable anywhere. The schema consists of six steps with considerable feedback and iteration occurring at several places:

1. Establish the goal of the data collection.
2. Develop a list of questions of interest.
3. Establish data categories.
4. Design and test data collection forms.
5. Collect and validate data.
6. Analyze data.

The importance of the validation element of a data collection system or a development tracking system cannot be overemphasized.

In their study of NASA's Software Engineering Laboratory projects, Basili and Weiss (1984) found that software data are error-prone and that special validation provisions are generally needed. Validation should be performed concurrently with software development and data collection, based on interviews with those people supplying the data. In cases where data collection is part of the configuration control process and automated tools are available, data validation routines (e.g., consistency check, range limits, conditional entries, etc.) should be an integral part of the tools. Furthermore, training, clear guidelines and instructions, and an understanding of how the data are used by people who enter or collect the data enhance data accuracy significantly.

The actual collection process can take several basic formats such as reporting forms, interviews, and automatic collection using the computer system. For data collection to be efficient and effective, it should be merged with the configuration management or change control system. This is the case in most large development organizations. For example, at IBM Rochester the change control system covers the entire development process, and online tools are used for plan change

control, development items and changes, integration, and change control after integration (defect fixes). The tools capture data pertinent to schedule, resource, and project status, as well as quality indicators. In general, change control is more prevalent after the code is integrated. This is one of the reasons that in many organizations defect data are usually available for the testing phases but not for the design and coding phases.

With regard to defect data, testing defects are generally more reliable than inspection defects. During testing, a "bug" exists when a test case cannot execute or when the test results deviate from the expected outcome. During inspections, the determination of a defect is based on the judgment of the inspectors. Therefore, it is important to have a clear definition of an inspection defect. The following is an example of such a definition:

> *Inspection defect:* A problem found during the inspection process which, if not fixed, would cause one or more of the following to occur:
>
> - ☐ A defect condition in a later inspection phase
> - ☐ A defect condition during testing
> - ☐ A field defect
> - ☐ Nonconformance to requirements and specifications
> - ☐ Nonconformance to established standards such as performance, national language translation, and usability

For example, misspelled words are not counted as defects, but would be if they were found on a screen that customers use. Using nested IF-THEN-ELSE structures instead of a SELECT statement would not be counted as a defect unless some standard or performance reason dictated otherwise.

Figure 4.7 is an example of an inspection summary form. The form records the total number of inspection defects and the LOC estimate for each part (module), as well as defect data classified by defect origin and defect type. The following guideline pertains to the defect type classification by development phase:

> *Interface defect:* An interface defect is a defect in the way two separate pieces of logic communicate. These are errors in communication between:
>
> Components
>
> Products
>
> Modules and subroutines of a component
>
> User interface (e.g., messages, panels)

Examples of interface defects per development phase follow.

Product: _____ Component: _____ Release: _____

Inspection Type: _____ (RQ SD IO I1 I2 U1 U2)

Description: _____

Defect Counts

Tot for Inspection / Part Name	LOC	By Defect Origin RQ	SD	I0	I1	I2	By Defect Type LO	IF	DO	Total

Total Preparation Hrs: ____ . ____ Total Inspection Hrs: ____ . ____

Total Persons Attended: ____ Inspection Date: ____/____/____

Reinspection Required? ____ (Y/N)

Defect Types:
- DO = Documentation
- IF = Interface
- LO = Logic

Defect Origins and Inspection Types:
- RQ = Requirements
- SD = System Design
- I0 = High-Level Design
- I1 = Low-Level Design
- I2 = Code

FIGURE 4.7
An Inspection Summary Form

High-Level Design (I0)

Use of wrong parameter

Inconsistent use of function keys on user interface (e.g., screen)

Incorrect message used

Presentation of information on screen not usable

Low-Level Design (I1)

Missing required parameters (e.g., missing parameter on module)

Wrong parameters (e.g., specified incorrect parameter on module)

Intermodule interfaces: input not there, input in wrong order

Intramodule interfaces: passing values/data to subroutines

Incorrect use of common data structures

Misusing data passed to code

Code (I2)

Passing wrong values for parameters on macros, application program interfaces (APIs), modules

Setting up a common control block/area used by another piece of code incorrectly

Not issuing correct exception to caller of code

Logic defect: A logic defect is one that would cause incorrect results in the function to be performed by the logic. High-level categories of this type of defect are as follows:

Function: capability not implemented or implemented incorrectly

Assignment: initialization

Checking: validate data/values before use

Timing: management of shared/real-time resources

Data Structures: static and dynamic definition of data

Examples of logic defects per development phase follow.

High-Level Design (I0)

Invalid or incorrect screen flow

High-level flow through component missing or incorrect in the review package

Function missing from macros you are implementing

Using a wrong macro to do a function that will not work (e.g., using XXXMSG to receive a message from a program message queue, instead of YYYMSG).

Missing requirements

Missing parameter/field on command/in database structure/on screen you are implementing

Wrong value on keyword (e.g., macro, command)

Wrong keyword (e.g., macro, command)

Low-Level Design (I1)

Logic does not implement I0 design

Missing or excessive function

Values in common structure not set

Propagation of authority and adoption of authority (lack of or too much)

Lack of code page conversion

Incorrect initialization

Not handling abnormal termination (conditions, cleanup, exit routines)

Lack of normal termination cleanup

Performance: too much processing in loop that could be moved outside of loop

Code (I2)

Code does not implement I1 design

Lack of initialization

Variables initialized incorrectly

Missing exception monitors

Exception monitors in wrong order

Exception monitors not active

Exception monitors active at the wrong time

Exception monitors set up wrong

Truncating of double-byte character set data incorrectly (e.g., truncating before shift in character)

Incorrect code page conversion

Lack of code page conversion

Not handling exceptions/return codes correctly

Documentation defect: A documentation defect is a defect in the description of the function (e.g., prologue of macro) that causes someone to do something wrong based on this information. For example, if a macro prologue contained an incorrect description of a parameter that caused the user of this macro to use the parameter incorrectly, this would be a documentation defect against the macro.

Examples of documentation defects per development phase follow.

High-Level Design (I0)

Incorrect information in prologue (e.g., macro)

Misspelling on user interface (e.g., screen)

Wrong wording (e.g., messages, command prompt text)
Using restricted words on user interface
Wording in messages, definition of command parameters is technically incorrect

Low-Level Design (I1)

Wrong information in prologue (e.g., macros, program, etc.)
Missing definition of inputs and outputs of module, subroutines, etc.
Insufficient documentation of logic (comments tell what but not why)

Code (I2)

Information in prologue not correct or missing
Wrong wording in messages
Second-level text of message technically incorrect
Insufficient documentation of logic (comments tell what but not why)
Incorrect documentation of logic

4.6 Summary

Software quality metrics focus on the quality aspects of the product, process, and project. They can be grouped into three categories in accordance with the software life cycle: end-product quality metrics, in-process quality metrics, and maintenance quality metrics. This chapter gives several examples for each category, summarizes the metrics programs at Motorola, Hewlett-Packard, and IBM Rochester, and discusses data collection.

- *Product quality metrics*
  - –Mean time to failure
  - –Defect density
  - –Customer-reported problems
  - –Customer satisfaction
- *In-process quality metrics*
  - –Phase-based defect removal pattern
  - –Defect removal effectiveness
  - –Defect density during formal machine testing
  - –Defect arrival pattern during formal machine testing

Recommendations for Small Organizations

To my knowledge, there is no correlation between the existence of a metrics program and the size of the organization, or between the effectiveness of using metrics in software development and team size. For small teams that are starting to implement a metrics program with minimum resources, I recommend the following:

1. Implement the defect arrival metric during the last phase of testing, augmented by a qualitative indicator of the number of critical problems (or show stoppers) and the nature of these problems. The latter is a subset of total defects. These indicators provide important information with regard to the quality of the software and its readiness to ship to customers. For example, if the defect arrival metric is trending down to a sufficiently low level and the number of show stoppers is small and becoming sparser as the test progresses, one can forecast a positive quality posture of the product in the field, and vice versa.
2. After the product is shipped, a natural metric is the number of defects coming in from the field over time. As defects are reported, the organization fixes these defects for their customers, normally in the form of individual fixes, fix packages, or frequent software upgrades, depending on the organization's maintenance strategy. The inevitable question is, how many outstanding defects need to be fixed; in other words, how big is the fix backlog. If the organization intends to compare its fix efficiency to the volume of defect arrivals, the backlog management index metric can be formed easily without additional data.
3. When a product size metric is available, the above metrics can be normalized to form other metrics such as defect density curves over time, during testing or when the product is in the field, and overall defect rate of the product, for a specific duration or for the maintenance life of the product. These normalized metrics can serve as baselines for other projects by

- *Maintenance quality metrics*
  - –Fix backlog
  - –Backlog management index
  - –Fix response time and fix responsiveness
  - –Percent delinquent fixes
  - –Defective fixes

With regard to in-process data, generally those at the back end (e.g., testing defects) are more reliable than those at the front end (e.g., design reviews and inspections). To improve data reliability, it is important to establish definitions and examples (e.g., what constitutes a defect during design reviews). Furthermore, validation must be an integral part of the data collection system and should be performed concurrently with software development and data collection.

the team, in terms of both process capacity (in-process metrics) and quality of delivered products. When enough empirical data is accumulated, correlational studies between in-process data and field data of the organization can improve the predictability and consistency of product development. If the operational definitions of the metrics are consistent, cross-organization comparisons can be made.

Of the two product size metrics discussed in this chapter, I recommend the function point (FP) over lines of code (LOC) for the many reasons discussed. However, accurate function point counting requires certified function point specialists and it can be time-consuming and expensive. If LOC data is already available, the *backfiring* method, a conversion ratio method between logical source code statements and equivalent volumes of functions points can be used. Such conversion ratios for a set of languages were published by Jones (2000; see p. 78, Table 3.5), and the average ratios (average number of source statements (LOC) per function point) by language are as follows:

Basic Assembly	320
Macro Assembly	213
C	128
FORTRAN	107
COBOL	107
Pascal	91
PL/I	80
Ada83	71
C++	53
Ada95	49
Visual Basic	32
Smalltalk	21
SQL	12

This discussion of a few simple and useful metrics as the beginning of a good metrics practice is from the quality perspective. For overall project management, it is well known that the most basic variables to measure are product size, effort (e.g., person-year), and development cycle time.

References

1. Albrecht, A. J., "Measuring Application Development Productivity," *Proceedings of the Joint IBM/SHARE/GUIDE Application Development Symposium,* October 1979, pp. 83–92.
2. Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering,* Vol. SE-10, 1984, pp. 728–738.
3. Boehm, B. W., *Software Engineering Economics,* Englewood Cliffs, N.J.: Prentice-Hall, 1981.
4. Conte, S. D., H. E. Dunsmore, and V. Y. Shen, *Software Engineering Metrics and Models,* Menlo Park, Calif.: Benjamin/Cummings, 1986.
5. Cusumano, M. A., "Objectives and Context of Software Measurement, Analysis and Control," Massachusetts Institute of Technology Sloan School of Management Working Paper 3471-92, October 1992.
6. Daskalantonakis, M. K., "A Practical View of Software Measurement and Implementation Experiences Within Motorola," *IEEE Transactions on Software Engineering,* Vol. SE-18, 1992, pp. 998–1010.

7. DeMarco, T., *Controlling Software Projects,* New York: Yourdon Press, 1982.
8. Fenton, N. E., and S. L. Pfleeger, *Software Metrics: A Rigorous Approach,* 2nd ed., Boston: International Thomson Computer Press, 1997.
9. Grady, R. B., and D. L. Caswell, *Software Metrics: Establishing a Company-Wide Program,* Englewood Cliffs, N.J.: Prentice-Hall, 1986.
10. IFPUG, IFPUG Counting Practices Manual, Release 4.1, Westerville, Ohio: International Function Point Users Group, 1999.
11. Jones, C., *Programming Productivity,* New York: McGraw-Hill, 1986.
12. Jones, C., "Critical Problems in Software Measurement," Burlington, Mass.: Software Productivity Research, 1992.
13. Jones, C., *Assessment and Control of Software Risks,* Englewood Cliffs, N. J.: Yourdon Press, 1994.
14. Jones, C., *Applied Software Measurement, Assuring Productivity and Quality,* 2nd ed., New York: McGraw-Hill, 1997.
15. Jones, C., *Software Assessments, Benchmarks, and Best Practices,* Boston: Addison-Wesley, 2000.
16. Kemerer, C. F., "Reliability of Function Point Measurement: A Field Experiment," Massachusetts Institute of Technology Sloan School of Management Working Paper 3193-90-MSA, January 1991.
17. Kemerer, C. F., and B. S. Porter, "Improving the Reliability of Function Point Measurement: An Empirical Study," *IEEE Transactions on Software Engineering,* Vol. 18, No. 11, November 1992, pp. 1011–1024.
18. Littlewood, B., and L. Strigini, "The Risks of Software," *Scientific American,* November 1992, pp. 62–75.
19. MacLeod, J. M., "Implementing and Sustaining a Software Inspection Program in an R&D Environment," *Hewlett-Packard Journal,* June 1993, pp. 60–63.
20. Myers, G. J., *The Art of Software Testing,* New York: John Wiley & Sons, 1979.
21. Oman, P., and S. L. Pfleeger (ed.), *Applying Software Metrics,* Los Alamitos: IEEE Computer Society Press, 1997.
22. Shooman, M. L., *Software Engineering: Design, Reliability, and Management,* New York: McGraw-Hill, 1983.
23. Sprouls, J., *IFPUG Function Point Counting Practices Manual, Release 3.0,* Westerville, Ohio: International Function Point Users Group, 1990.
24. Symons, C. R., *Software Sizing and Estimating: Mk II FPA (Function Point Analysis),* Chichester, United Kingdom: John Wiley & Sons, 1991.