QUICK LOOK

guidelines and past data. The results of the analysis are interpreted to gain insight into the quality of the software, and the results of the interpretation lead to modification of work products arising out of analysis, design, code, or test.

What is the work product? Software metrics that are computed from data collected from the analysis and design models, source code, and test cases.

How do I ensure that I've done it right? You should establish the objectives of measurement before data collection begins, defining each technical metric in an unambiguous manner. Define only a few metrics and then use them to gain insight into the quality of a software engineering work product.

But some members of the software community continue to argue that software is unmeasurable or that attempts at measurement should be postponed until we better understand software and the attributes that should be used to describe it. This is a mistake.

Although technical metrics for computer software are not absolute, they provide us with a systematic way to assess quality based on a set of clearly defined rules. They also provide the software engineer with on-the-spot, rather than after-the-fact insight. This enables the engineer to discover and correct potential problems before they become catastrophic defects.

In Chapter 4, we discussed software metrics as they are applied at the process and project level. In this chapter, our focus shifts to measures that can be used to assess the quality of the product as it is being engineered. These measures of internal product attributes provide the software engineer with a real-time indication of the efficacy of the analysis, design, and code models; the effectiveness of test cases; and the overall quality of the software to be built.

19.1 SOFTWARE QUALITY

Quote:

"Every program does something right, it just may not be the thing that we want it to do."

author unknown

Even the most jaded software developers will agree that high-quality software is an important goal. But how do we define *quality?* In Chapter 8, we proposed a number of different ways to look at software quality and introduced a definition that stressed conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.

There is little question that the preceding definition could be modified or extended and debated endlessly. For the purposes of this book, the definition serves to emphasize three important points:

1. Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.[1]

1 It is important to note that quality extends to the technical attributes of the analysis, design, and code models. Models that exhibit high quality (in the technical sense) will lead to software that exhibits high quality from the customer's point of view.

2. Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.
3. There is a set of implicit requirements that often goes unmentioned (e.g., the desire for ease of use). If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

Software quality is a complex mix of factors that will vary across different applications and the customers who request them. In the sections that follow, software quality factors are identified and the human activities required to achieve them are described.

19.1.1 McCall's Quality Factors

KEY POINT

It's interesting to note that McCall's quality factors are as valid today as they were when they were first proposed in the 1970s. Therefore, it's reasonable to assert that the factors that affect software quality do not change.

The factors that affect software quality can be categorized in two broad groups: (1) factors that can be directly measured (e.g., defects per function-point) and (2) factors that can be measured only indirectly (e.g., usability or maintainability). In each case measurement must occur. We must compare the software (documents, programs, data) to some datum and arrive at an indication of quality.

McCall, Richards, and Walters [MCC77] propose a useful categorization of factors that affect software quality. These software quality factors, shown in Figure 19.1, focus on three important aspects of a software product: its operational characteristics, its ability to undergo change, and its adaptability to new environments.

Referring to the factors noted in Figure 19.1, McCall and his colleagues provide the following descriptions:

Correctness. The extent to which a program satisfies its specification and fulfills the customer's mission objectives.

Reliability. The extent to which a program can be expected to perform its intended function with required precision. [It should be noted that other, more complete definitions of reliability have been proposed (see Chapter 8).]

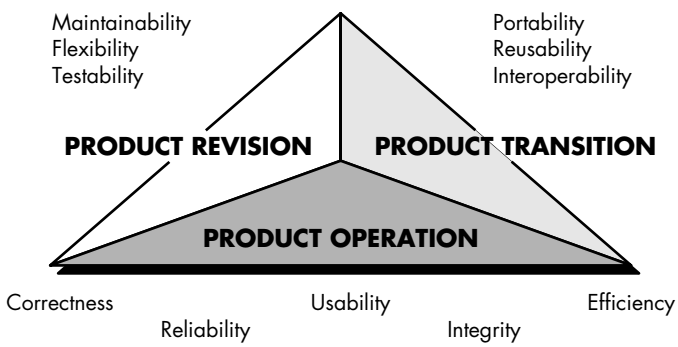


FIGURE 19.1 McCall's software quality factors

Efficiency. The amount of computing resources and code required by a program to perform its function.

Integrity. Extent to which access to software or data by unauthorized persons can be controlled.

Quote:

"A product's quality is a function of how much it changes the world for the better."

Tom DeMarco

Usability. Effort required to learn, operate, prepare input, and interpret output of a program.

Maintainability. Effort required to locate and fix an error in a program. [This is a very limited definition.]

Flexibility. Effort required to modify an operational program.

Testability. Effort required to test a program to ensure that it performs its intended function.

Portability. Effort required to transfer the program from one hardware and/or software system environment to another.

Reusability. Extent to which a program [or parts of a program] can be reused in other applications—related to the packaging and scope of the functions that the program performs.

Interoperability. Effort required to couple one system to another.

It is difficult, and in some cases impossible, to develop direct measures of these quality factors. Therefore, a set of metrics are defined and used to develop expressions for each of the factors according to the following relationship:

$$F_q = c_1 \times m_1 + c_2 \times m_2 + \ldots + c_n \times m_n$$

where F_q is a software quality factor, c_n are regression coefficients, m_n are the metrics that affect the quality factor. Unfortunately, many of the metrics defined by McCall et al. can be measured only subjectively. The metrics may be in the form of a checklist that is used to "grade" specific attributes of the software [CAV78]. The grading scheme proposed by McCall et al. is a 0 (low) to 10 (high) scale. The following metrics are used in the grading scheme:

Auditability. The ease with which conformance to standards can be checked.

Accuracy. The precision of computations and control.

Communication commonality. The degree to which standard interfaces, protocols, and bandwidth are used.

XRef

The metrics noted can be assessed during formal technical reviews discussed in Chapter 8.

Completeness. The degree to which full implementation of required function has been achieved.

Conciseness. The compactness of the program in terms of lines of code.

Consistency. The use of uniform design and documentation techniques throughout the software development project.

Data commonality. The use of standard data structures and types throughout the program.

Error tolerance. The damage that occurs when the program encounters an error.

Execution efficiency. The run-time performance of a program.

Expandability. The degree to which architectural, data, or procedural design can be extended.

Generality. The breadth of potential application of program components.

Hardware independence. The degree to which the software is decoupled from the hardware on which it operates.

Instrumentation. The degree to which the program monitors its own operation and identifies errors that do occur.

Quality Factors

Modularity. The functional independence (Chapter 13) of program components.

Operability. The ease of operation of a program.

Security. The availability of mechanisms that control or protect programs and data.

Self-documentation. The degree to which the source code provides meaningful documentation.

Simplicity. The degree to which a program can be understood without difficulty.

Software system independence. The degree to which the program is independent of nonstandard programming language features, operating system characteristics, and other environmental constraints.

Traceability. The ability to trace a design representation or actual program component back to requirements.

Training. The degree to which the software assists in enabling new users to apply the system.

The relationship between software quality factors and these metrics is shown in Figure 19.2. It should be noted that the weight given to each metric is dependent on local products and concerns.

19.1.2 FURPS

The quality factors described by McCall and his colleagues [MCC77] represent one of a number of suggested "checklists" for software quality. Hewlett-Packard [GRA87] developed a set of software quality factors that has been given the acronym FURPS—

FIGURE 19.2 Quality factors and metrics

Quality factor \ Software quality metric	Correctness	Reliability	Efficiency	Integrity	Maintainability	Flexibility	Testability	Portability	Reusability	Interoperability	Usability
Auditability				x			x				
Accuracy		x									
Communication commonality										x	
Completeness	x										
Complexity		x				x	x				
Concision			x		x	x					
Consistency	x	x			x	x					
Data commonality										x	
Error tolerance		x									
Execution efficiency			x								
Expandability						x					
Generality						x		x	x	x	
Hardware Indep.								x	x		
Instrumentation				x	x		x				
Modularity		x			x	x	x	x	x	x	
Operability			x								x
Security				x							
Self-documentation					x	x	x	x	x		
Simplicity		x			x	x	x				
System Indep.								x	x		
Traceability	x										
Training											x

(Adapted from Arthur, L. A., *Measuring Programmer Productivity and Software Quality,* Wiley-Interscience, 1985.)

functionality, usability, reliability, performance, and supportability. The FURPS quality factors draw liberally from earlier work, defining the following attributes for each of the five major factors:

- *Functionality* is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- *Usability* is assessed by considering human factors (Chapter 15), overall aesthetics, consistency, and documentation.
- *Reliability* is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.
- *Performance* is measured by processing speed, response time, resource consumption, throughput, and efficiency.

- *Supportability* combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, maintainability—in addition, testability, compatibility, configurability (the ability to organize and control elements of the software configuration, Chapter 9), the ease with which a system can be installed, and the ease with which problems can be localized.

The FURPS quality factors and attributes just described can be used to establish quality metrics for each step in the software engineering process.

19.1.3 ISO 9126 Quality Factors

The ISO 9126 standard was developed in an attempt to identify the key quality attributes for computer software. The standard identifies six key quality attributes:

Functionality. The degree to which the software satisfies stated needs as indicated by the following subattributes: suitability, accuracy, interoperability, compliance, and security.

Reliability. The amount of time that the software is available for use as indicated by the following subattributes: maturity, fault tolerance, recoverability.

Usability. The degree to which the software is easy to use as indicated by the following subattributes: understandability, learnability, operability.

Efficiency. The degree to which the software makes optimal use of system resources as indicated by the following subattributes: time behavior, resource behavior.

Maintainability. The ease with which repair may be made to the software as indicated by the following subattributes: analyzability, changeability, stability, testability.

Portability. The ease with which the software can be transposed from one environment to another as indicated by the following subattributes: adaptability, installability, conformance, replaceability.

Quote:

"Any activity becomes creative when the doer cares about doing it right, or better."

John Updike

Like other software quality factors discussed in Sections 19.1.1 and 19.1.2, the ISO 9126 factors do not necessarily lend themselves to direct measurement. However, they do provide a worthwhile basis for indirect measures and an excellent checklist for assessing the quality of a system.

19.1.4 The Transition to a Quantitative View

In the preceding sections, a set of qualitative factors for the "measurement" of software quality was discussed. We strive to develop precise measures for software quality and are sometimes frustrated by the subjective nature of the activity. Cavano and McCall [CAV78] discuss this situation:

The determination of quality is a key factor in every day events—wine tasting contests, sporting events [e.g., gymnastics], talent contests, etc. In these situations, quality is judged in the most fundamental and direct manner: side by side comparison of objects under identical conditions and with predetermined concepts. The wine may be judged according to clarity, color, bouquet, taste, etc. However, this type of judgement is very subjective; to have any value at all, it must be made by an expert.

Subjectivity and specialization also apply to determining software quality. To help solve this problem, a more precise definition of software quality is needed as well as a way to derive quantitative measurements of software quality for objective analysis . . . Since there is no such thing as absolute knowledge, one should not expect to measure software quality exactly, for every measurement is partially imperfect. Jacob Bronkowski described this paradox of knowledge in this way: "Year by year we devise more precise instruments with which to observe nature with more fineness. And when we look at the observations we are discomfited to see that they are still fuzzy, and we feel that they are as uncertain as ever."

In the sections that follow, we examine a set of software metrics that can be applied to the quantitative assessment of software quality. In all cases, the metrics represent indirect measures; that is, we never really measure quality but rather some manifestation of quality. The complicating factor is the precise relationship between the variable that is measured and the quality of software.

19.2 A FRAMEWORK FOR TECHNICAL SOFTWARE METRICS

As we noted in the introduction to this chapter, measurement assigns numbers or symbols to attributes of entities in the real word. To accomplish this, a measurement model encompassing a consistent set of rules is required. Although the theory of measurement (e.g., [KYB84]) and its application to computer software (e.g., [DEM81], [BRI96], [ZUS97]) are topics that are beyond the scope of this book, it is worthwhile to establish a fundamental framework and a set of basic principles for the measurement of technical metrics for software.

Quote:

"Just as temperature measurement began with an index finger . . . and grew to sophisticated scales, tools and techniques, so too is software measurement maturing . . ."

Shari Pfleeger

19.2.1 The Challenge of Technical Metrics

Over the past three decades, many researchers have attempted to develop a single metric that provides a comprehensive measure of software complexity. Fenton [FEN94] characterizes this research as a search for "the impossible holy grail." Although dozens of complexity measures have been proposed [ZUS90], each takes a somewhat different view of what complexity is and what attributes of a system lead to complexity. By analogy, consider a metric for evaluating an attractive car. Some observers might emphasize body design, others might consider mechanical characteristics, still others might tout cost, or performance, or fuel economy, or the ability to recycle when the car is junked. Since any one of these characteristics may be at odds with others, it is difficult to derive a single value for "attractiveness." The same problem occurs with computer software.

WebRef

Voluminous information on technical metrics has been compiled by Horst Zuse: **irb.cs.tu-berlin.de/~zuse/**

Yet there is a need to measure and control software complexity. And if a single value of this quality metric is difficult to derive, it should be possible to develop measures of different internal program attributes (e.g., effective modularity, functional independence, and other attributes discussed in Chapters 13 through 16). These measures and the metrics derived from them can be used as independent indicators of the quality of analysis and design models. But here again, problems arise. Fenton [FEN94] notes this when he states:

> The danger of attempting to find measures which characterize so many different attributes is that inevitably the measures have to satisfy conflicting aims. This is counter to the representational theory of measurement.

Although Fenton's statement is correct, many people argue that technical measurement conducted during the early stages of the software process provides software engineers with a consistent and objective mechanism for assessing quality.

It is fair to ask, however, just how valid technical metrics are. That is, how closely aligned are technical metrics to the long-term reliability and quality of a computer-based system? Fenton [FEN91] addresses this question in the following way:

> In spite of the intuitive connections between the internal structure of software products [technical metrics] and its external product and process attributes, there have actually been very few scientific attempts to establish specific relationships. There are a number of reasons why this is so; the most commonly cited is the impracticality of conducting relevant experiments.

Each of the "challenges" noted here is a cause for caution, but it is no reason to dismiss technical metrics.[2] Measurement is essential if quality is to be achieved.

19.2.2 Measurement Principles

Before we introduce a series of technical metrics that (1) assist in the evaluation of the analysis and design models, (2) provide an indication of the complexity of procedural designs and source code, and (3) facilitate the design of more effective testing, it is important to understand basic measurement principles. Roche [ROC94] suggests a measurement process that can be characterized by five activities:

? What are the steps of an effective measurement process?

- *Formulation*. The derivation of software measures and metrics that are appropriate for the representation of the software that is being considered.
- *Collection*. The mechanism used to accumulate data required to derive the formulated metrics.
- *Analysis*. The computation of metrics and the application of mathematical tools.

2 A vast literature on software metrics (e.g., see [FEN94], [ROC94], [ZUS97] for extensive bibliographies) has been spawned, and criticism of specific metrics (including some of those presented in this chapter) is common. However, many of the critiques focus on esoteric issues and miss the primary objective of measurement in the real world: to help the engineer establish a systematic and objective way to gain insight into his or her work and to improve product quality as a result.

- *Interpretation*. The evaluation of metrics results in an effort to gain insight into the quality of the representation.
- *Feedback*. Recommendations derived from the interpretation of technical metrics transmitted to the software team.

The principles that can be associated with the formulation of technical metrics are [ROC94]

? What rules should we observe when we establish technical measures?

- The objectives of measurement should be established before data collection begins.
- Each technical metric should be defined in an unambiguous manner.
- Metrics should be derived based on a theory that is valid for the domain of application (e.g., metrics for design should draw upon basic design concepts and principles and attempt to provide an indication of the presence of an attribute that is deemed desirable).
- Metrics should be tailored to best accommodate specific products and processes [BAS84].

Although formulation is a critical starting point, collection and analysis are the activities that drive the measurement process. Roche [ROC94] suggests the following principles for these activities:

Above all, keep your early attempts at technical measurement simple. Don't obsess over the "perfect" metric because it doesn't exist.

- Whenever possible, data collection and analysis should be automated.
- Valid statistical techniques should be applied to establish relationships between internal product attributes and external quality characteristics (e.g., is the level of architectural complexity correlated with the number of defects reported in production use?).
- Interpretative guidelines and recommendations should be established for each metric.

In addition to these principles, the success of a metrics activity is tied to management support. Funding, training, and promotion must all be considered if a technical measurement program is to be established and sustained.

19.2.3 The Attributes of Effective Software Metrics

Hundreds of metrics have been proposed for computer software, but not all provide practical support to the software engineer. Some demand measurement that is too complex, others are so esoteric that few real world professionals have any hope of understanding them, and others violate the basic intuitive notions of what high-quality software really is.

Ejiogu [EJI91] defines a set of attributes that should be encompassed by effective software metrics. The derived metric and the measures that lead to it should be

How should we assess the quality of a proposed software metric?

- *Simple and computable.* It should be relatively easy to learn how to derive the metric, and its computation should not demand inordinate effort or time.
- *Empirically and intuitively persuasive.* The metric should satisfy the engineer's intuitive notions about the product attribute under consideration (e.g., a metric that measures module cohesion should increase in value as the level of cohesion increases).
- *Consistent and objective.* The metric should always yield results that are unambiguous. An independent third party should be able to derive the same metric value using the same information about the software.

Experience indicates that a technical metric will be used only if it is intuitive and easy to compute. If dozens of "counts" have to be made and complex computations are required, it's unlikely that the metric will be widely applied.

- *Consistent in its use of units and dimensions.* The mathematical computation of the metric should use measures that do not lead to bizarre combinations of units. For example, multiplying people on the project teams by programming language variables in the program results in a suspicious mix of units that are not intuitively persuasive.
- *Programming language independent.* Metrics should be based on the analysis model, the design model, or the structure of the program itself. They should not be dependent on the vagaries of programming language syntax or semantics.
- *An effective mechanism for high-quality feedback.* That is, the metric should provide a software engineer with information that can lead to a higher-quality end product.

Although most software metrics satisfy these attributes, some commonly used metrics may fail to satisfy one or two of them. An example is the function point (discussed in Chapter 4 and again in this chapter). It can be argued[3] that the consistent and objective attribute fails because an independent third party may not be able to derive the same function point value as a colleague using the same information about the software. Should we therefore reject the FP measure? The answer is: "Of course not!" FP provides useful insight and therefore provides distinct value, even if it fails to satisfy one attribute perfectly.

19.3 METRICS FOR THE ANALYSIS MODEL

XRef

Data, functional, and behavioral models are discussed in Chapters 11 and 12.

Technical work in software engineering begins with the creation of the analysis model. It is at this stage that requirements are derived and that a foundation for design is established. Therefore, technical metrics that provide insight into the quality of the analysis model are desirable.

3 Please note that an equally vigorous counterargument can be made. Such is the nature of software metrics.

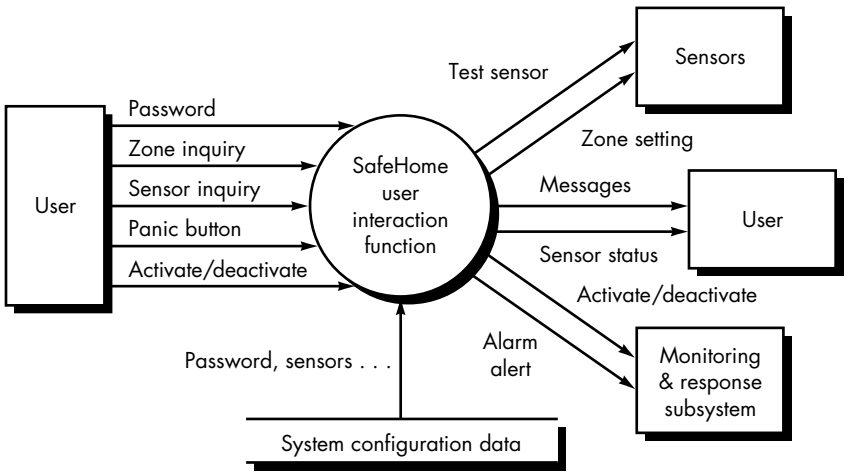


FIGURE 19.3
Part of the analysis model for SafeHome software

Although relatively few analysis and specification metrics have appeared in the literature, it is possible to adapt metrics derived for project application (Chapter 4) for use in this context. These metrics examine the analysis model with the intent of predicting the "size" of the resultant system. It is likely that size and design complexity will be directly correlated.

19.3.1 Function-Based Metrics

To be useful for technical work, measures that will assist technical decision making (e.g., errors found during unit testing) must be collected and then normalized using the FP metric.

The function point metric (Chapter 4) can be used effectively as a means for predicting the size of a system that will be derived from the analysis model. To illustrate the use of the FP metric in this context, we consider a simple analysis model representation, illustrated in Figure 19.3. Referring to the figure, a data flow diagram (Chapter 12) for a function within the *SafeHome* software[4] is represented. The function manages user interaction, accepting a user password to activate or deactivate the system, and allows inquiries on the status of security zones and various security sensors. The function displays a series of prompting messages and sends appropriate control signals to various components of the security system.

The data flow diagram is evaluated to determine the key measures required for computation of the function point metric (Chapter 4):

- number of user inputs
- number of user outputs
- number of user inquiries
- number of files
- number of external interfaces

4 *SafeHome* is a home security system that has been used as an example application in earlier chapters.

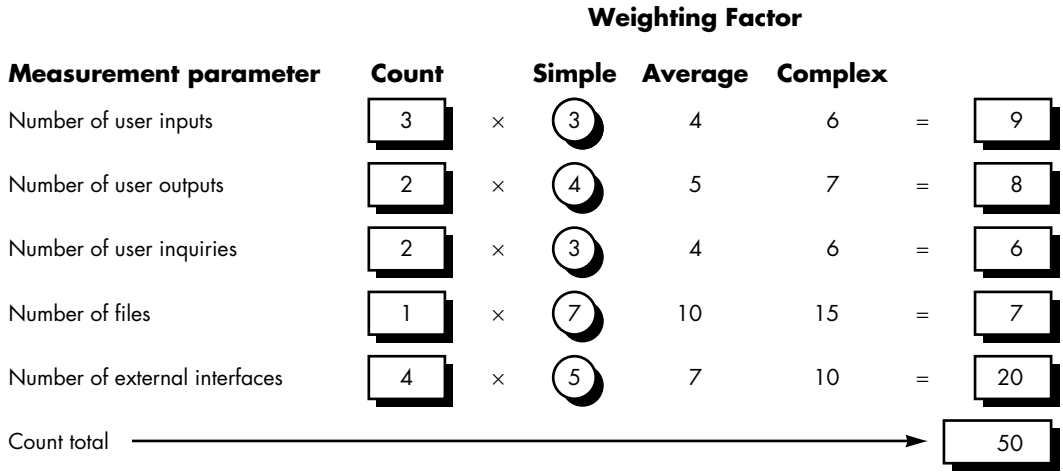


FIGURE 19.4 Computing function points for a SafeHome function

Three user inputs—**password, panic button,** and **activate/deactivate**—are shown in the figure along with two inquires—**zone inquiry** and **sensor inquiry.** One file (**system configuration file**) is shown. Two user outputs (**messages** and **sensor status**) and four external interfaces (**test sensor, zone setting, activate/deactivate,** and **alarm alert**) are also present. These data, along with the appropriate complexity, are shown in Figure 19.4.

The *count total* shown in Figure 19.4 must be adjusted using Equation (4-1):

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

where count total is the sum of all FP entries obtained from Figure 19.3 and F_i (i = 1 to 14) are "complexity adjustment values." For the purposes of this example, we assume that $\Sigma (F_i)$ is 46 (a moderately complex product). Therefore,

$$\text{FP} = 50 \times [0.65 + (0.01 \times 46)] = 56$$

Based on the projected FP value derived from the analysis model, the project team can estimate the overall implemented size of the *SafeHome* user interaction function. Assume that past data indicates that one FP translates into 60 lines of code (an object-oriented language is to be used) and that 12 FPs are produced for each person-month of effort. These historical data provide the project manager with important planning information that is based on the analysis model rather than preliminary estimates. Assume further that past projects have found an average of three errors per function point during analysis and design reviews and four errors per function point during unit and integration testing. These data can help software engineers assess the completeness of their review and testing activities.

WebRef

A useful introduction on FP has been prepared by Capers Jones and may be obtained at **www.spr.com/library/0funcmet.htm**

19.3.2 The Bang Metric

Like the function point metric, the *bang metric* can be used to develop an indication of the size of the software to be implemented as a consequence of the analysis model. Developed by DeMarco [DEM82], the bang metric is "an implementation independent indication of system size." To compute the bang metric, the software engineer must first evaluate a set of *primitives*—elements of the analysis model that are not further subdivided at the analysis level. Primitives [DEM82] are determined by evaluating the analysis model and developing counts for the following forms:[5]

Quote:

"Rather than just musing on what 'new metric' might apply . . . we should also be asking ourselves the more basic question, 'What will we do with metrics.'"

Michael Mah and Larry Putnam

Functional primitives (FuP). The number of transformations (bubbles) that appear at the lowest level of a data flow diagram (Chapter 12).

Data elements (DE). The number of attributes of a data object, data elements are not composite data and appear within the data dictionary.

Objects (OB). The number of data objects as described in Chapter 12.

Relationships (RE). The number of connections between data objects as described in Chapter 12.

States (ST). The number of user observable states in the state transition diagram (Chapter 12).

Transitions (TR). The number of state transitions in the state transition diagram (Chapter 12).

In addition to these six primitives, additional counts are determined for

Modified manual function primitives (FuPM). Functions that lie outside the system boundary but must be modified to accommodate the new system.

Input data elements (DEI). Those data elements that are input to the system.

Output data elements. (DEO). Those data elements that are output from the system.

Retained data elements. (DER). Those data elements that are retained (stored) by the system.

Data tokens (TC_i). The data tokens (data items that are not subdivided within a functional primitive) that exist at the boundary of the *i*th functional primitive (evaluated for each primitive).

Relationship connections (RE_i). The relationships that connect the *i*th object in the data model to other objects.

DeMarco [DEM82] suggests that most software can be allocated to one of two domains: *function strong* or *data strong,* depending upon the ratio RE/FuP. Function-strong

5 The acronym noted in parentheses following the primitive is used to denote the count of the particular primitive, e,g., FuP indicates the number of functional primitives present in an analysis model.

applications (often encountered in engineering and scientific applications) emphasize the transformation of data and do not generally have complex data structures. Data-strong applications (often encountered in information systems applications) tend to have complex data models.

RE/FuP < 0.7 implies a function-strong application.
0.8 < RE/FuP < 1.4 implies a hybrid application.
RE/FuP > 1.5 implies a data-strong application.

Because different analysis models will partition the model to greater or lessor degrees of refinement, DeMarco suggests that an average token count per primitive is

$$TC_{avg} = \Sigma TC_i / FuP$$

be used to control uniformity of partitioning across many different models within an application domain.

To compute the bang metric for function-strong applications, the following algorithm is used:

```
set initial value of bang = 0;
do while functional primitives remain to be evaluated
    Compute token-count around the boundary of primitive i
    Compute corrected FuP increment (CFuPI)
    Allocate primitive to class
    Assess class and note assessed weight
    Multiply CFuPI  by the assessed weight
    bang = bang + weighted CFuPI
enddo
```

The token-count is computed by determining how many separate tokens are "visible" [DEM82] within the primitive. It is possible that the number of tokens and the number of data elements will differ, if data elements can be moved from input to output without any internal transformation. The corrected CFuPI is determined from a table published by DeMarco. A much abbreviated version follows:

TC_i	CFuPI
2	1.0
5	5.8
10	16.6
15	29.3
20	43.2

The assessed weight noted in the algorithm is determined from 16 different classes of functional primitives defined by DeMarco. A weight ranging from 0.6 (simple data routing) to 2.5 (data management functions) is assigned, depending on the class of the primitive.

For data-strong applications, the bang metric is computed using the following algorithm:

```
set initial value of bang = 0;
do while objects remain to be evaluated in the data model
    compute count of relationships for object i
    compute corrected OB increment (COBI)
    bang = bang + COBI
enddo
```

The COBI is determined from a table published by DeMarco. An abbreviated version follows:

RE_i	COBI
1	1.0
3	4.0
6	9.0

Once the bang metric has been computed, past history can be used to associate it with size and effort. DeMarco suggests that an organization build its own versions of the CFuPI and COBI tables using calibration information from completed software projects.

19.3.3 Metrics for Specification Quality

By measuring characteristics of the specification, it is possible to gain quantitative insight into specificity and completeness.

Davis and his colleagues [DAV93] propose a list of characteristics that can be used to assess the quality of the analysis model and the corresponding requirements specification: *specificity* (lack of ambiguity), *completeness, correctness, understandability, verifiability, internal and external consistency, achievability, concision, traceability, modifiability, precision,* and *reusability.* In addition, the authors note that high-quality specifications are electronically stored, executable or at least interpretable, annotated by relative importance and stable, versioned, organized, cross-referenced, and specified at the right level of detail.

Although many of these characteristics appear to be qualitative in nature, Davis et al. [DAV93] suggest that each can be represented using one or more metrics.[6] For example, we assume that there are n_r requirements in a specification, such that

$$n_r = n_f + n_{nf}$$

where n_f is the number of functional requirements and n_{nf} is the number of nonfunctional (e.g., performance) requirements.

To determine the *specificity* (lack of ambiguity) of requirements, Davis et al. suggest a metric that is based on the consistency of the reviewers' interpretation of each requirement:

$$Q_1 = n_{ui}/n_r$$

6 A complete discussion of specification quality metrics is beyond the scope of this chapter. See [DAV93] for more details.

where n_{ui} is the number of requirements for which all reviewers had identical interpretations. The closer the value of Q to 1, the lower is the ambiguity of the specification.

The *completeness* of functional requirements can be determined by computing the ratio

$$Q_2 = n_u / [n_i \times n_s]$$

where n_u is the number of unique function requirements, n_i is the number of inputs (stimuli) defined or implied by the specification, and n_s is the number of states specified. The Q_2 ratio measures the percentage of necessary functions that have been specified for a system. However, it does not address nonfunctional requirements. To incorporate these into an overall metric for completeness, we must consider the degree to which requirements have been validated:

$$Q_3 = n_c / [n_c + n_{nv}]$$

where n_c is the number of requirements that have been validated as correct and n_{nv} is the number of requirements that have not yet been validated.

19.4 METRICS FOR THE DESIGN MODEL

Quote:

"Measure what is measurable, and what is not measurable, make measurable."

Galileo

It is inconceivable that the design of a new aircraft, a new computer chip, or a new office building would be conducted without defining design measures, determining metrics for various aspects of design quality, and using them to guide the manner in which the design evolves. And yet, the design of complex software-based systems often proceeds with virtually no measurement. The irony of this is that design metrics for software are available, but the vast majority of software engineers continue to be unaware of their existence.

Design metrics for computer software, like all other software metrics, are not perfect. Debate continues over their efficacy and the manner in which they should be applied. Many experts argue that further experimentation is required before design measures can be used. And yet, design without measurement is an unacceptable alternative.

In the sections that follow, we examine some of the more common design metrics for computer software. Each can provide the designer with improved insight and all can help the design to evolve to a higher level of quality.

19.4.1 Architectural Design Metrics

Architectural design metrics focus on characteristics of the program architecture (Chapter 14) with an emphasis on the architectural structure and the effectiveness of modules. These metrics are black box in the sense that they do not require any knowledge of the inner workings of a particular software component.

Card and Glass [CAR90] define three software design complexity measures: structural complexity, data complexity, and system complexity.

Structural complexity of a module *i* is defined in the following manner:

$$S(i) = f^2_{out}(i) \tag{19-1}$$

KEY POINT

Metrics can provide insight into structural, data and system complexity associated with the architectural design.

where $f_{out}(i)$ is the fan-out[7] of module *i*.

Data complexity provides an indication of the complexity in the internal interface for a module *i* and is defined as

$$D(i) = v(i)/[f_{out}(i) + 1] \tag{19-2}$$

where $v(i)$ is the number of input and output variables that are passed to and from module *i*.

Finally, *system complexity* is defined as the sum of structural and data complexity, specified as

$$C(i) = S(i) + D(i) \tag{19-3}$$

As each of these complexity values increases, the overall architectural complexity of the system also increases. This leads to a greater likelihood that integration and testing effort will also increase.

? Is there a way to assess the complexity of certain architectural models?

An earlier high-level architectural design metric proposed by Henry and Kafura [HEN81] also makes use of the fan-in and fan-out. The authors define a complexity metric (applicable to call and return architectures) of the form

$$\text{HKM} = \text{length}(i) \times [f_{in}(i) + f_{out}(i)]^2 \tag{19-4}$$

where length(i) is the number of programming language statements in a module *i* and $f_{in}(i)$ is the fan-in of a module *i*. Henry and Kafura extend the definitions of *fan-in* and *fan-out* presented in this book to include not only the number of module control connections (module calls) but also the number of data structures from which a module *i* retrieves (fan-in) or updates (fan-out) data. To compute HKM during design, the procedural design may be used to estimate the number of programming language statements for module *i*. Like the Card and Glass metrics noted previously, an increase in the Henry-Kafura metric leads to a greater likelihood that integration and testing effort will also increase for a module.

Fenton [FEN91] suggests a number of simple *morphology* (i.e., shape) metrics that enable different program architectures to be compared using a set of straightforward dimensions. Referring to Figure 19.5, the following metrics can be defined:

$$\text{size} = n + a$$

---

7 Recalling the discussion presented in Chapter 13, fan-out indicates the number of modules immediately subordinate to module *i;* that is, the number of modules that are directly invoked by module *i*.

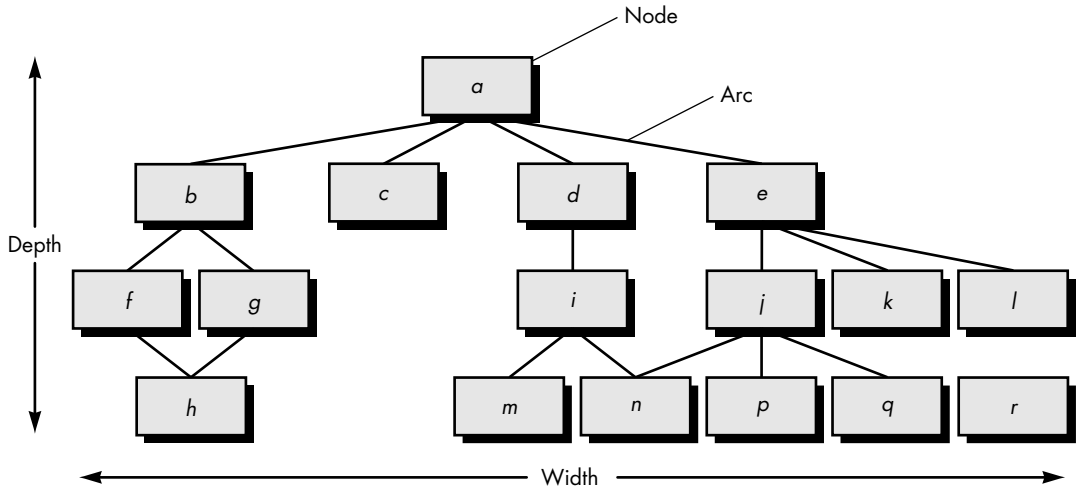


FIGURE 19.5 Morphology metrics

where n is the number of nodes and a is the number of arcs. For the architecture shown in Figure 19.5,

size = 17 + 18 = 35

depth = the longest path from the root (top) node to a leaf node. For the architecture shown in Figure 19.5, depth = 4.

width = maximum number of nodes at any one level of the architecture. For the architecture shown in Figure 19.5, width = 6.

arc-to-node ratio, $r = a/n$,

which measures the connectivity density of the architecture and may provide a simple indication of the coupling of the architecture. For the architecture shown in Figure 19.5, r = 18/17 = 1.06.

The U.S. Air Force Systems Command [USA87] has developed a number of software quality indicators that are based on measurable design characteristics of a computer program. Using concepts similar to those proposed in IEEE Std. 982.1-1988 [IEE94], the Air Force uses information obtained from data and architectural design to derive a *design structure quality index* (DSQI) that ranges from 0 to 1. The following values must be ascertained to compute the DSQI [CHA89]:

S_1 = the total number of modules defined in the program architecture.

S_2 = the number of modules whose correct function depends on the source of data input or that produce data to be used elsewhere (in general, control modules, among others, would not be counted as part of S_2).

S_3 = the number of modules whose correct function depends on prior processing.

S_4 = the number of database items (includes data objects and all attributes that define objects).

S_5 = the total number of unique database items.

S_6 = the number of database segments (different records or individual objects).

S_7 = the number of modules with a single entry and exit (exception processing is not considered to be a multiple exit).

Quote:

"Measurement can be seen as a detour. This detour is necessary because humans mostly are not able to make clear and objective decisions [without quantitative support]."

Horst Zuse

Once values S_1 through S_7 are determined for a computer program, the following intermediate values can be computed:

Program structure: D_1, where D_1 is defined as follows: If the architectural design was developed using a distinct method (e.g., data flow-oriented design or object-oriented design), then $D_1 = 1$, otherwise $D_1 = 0$.

Module independence: $D_2 = 1 - (S_2/S_1)$

Modules not dependent on prior processing: $D_3 = 1 - (S_3/S_1)$

Database size: $D_4 = 1 - (S_5/S_4)$

Database compartmentalization: $D_5 = 1 - (S_6/S_4)$

Module entrance/exit characteristic: $D_6 = 1 - (S_7/S_1)$

With these intermediate values determined, the DSQI is computed in the following manner:

$$\text{DSQI} = \Sigma\, w_i D_i \tag{19-5}$$

where $i = 1$ to 6, w_i is the relative weighting of the importance of each of the intermediate values, and $\Sigma\, w_i = 1$ (if all D_i are weighted equally, then $w_i = 0.167$).

The value of DSQI for past designs can be determined and compared to a design that is currently under development. If the DSQI is significantly lower than average, further design work and review are indicated. Similarly, if major changes are to be made to an existing design, the effect of those changes on DSQI can be calculated.

19.4.2 Component-Level Design Metrics

Component-level design metrics focus on internal characteristics of a software component and include measures of the "three Cs"—module cohesion, coupling, and complexity. These measures can help a software engineer to judge the quality of a component-level design.

The metrics presented in this section are glass box in the sense that they require knowledge of the inner working of the module under consideration. Component-level

design metrics may be applied once a procedural design has been developed. Alternatively, they may be delayed until source code is available.

It is possible to compute measures of the functional independence—coupling and cohesion—of a component and to use these to assess the quality of the design.

Cohesion metrics. Bieman and Ott [BIE94] define a collection of metrics that provide an indication of the cohesiveness (Chapter 13) of a module. The metrics are defined in terms of five concepts and measures:

Data slice. Stated simply, a data slice is a backward walk through a module that looks for data values that affect the module location at which the walk began. It should be noted that both program slices (which focus on statements and conditions) and data slices can be defined.

Data tokens. The variables defined for a module can be defined as data tokens for the module.

Glue tokens. This set of data tokens lies on one or more data slice.

Superglue tokens. These data tokens are common to every data slice in a module.

Stickiness. The relative stickiness of a glue token is directly proportional to the number of data slices that it binds.

Bieman and Ott develop metrics for *strong functional cohesion* (SFC), *weak functional cohesion* (WFC), and *adhesiveness* (the relative degree to which glue tokens bind data slices together). These metrics can be interpreted in the following manner [BIE94]:

> All of these cohesion metrics range in value between 0 and 1. They have a value of 0 when a procedure has more than one output and exhibits none of the cohesion attribute indicated by a particular metric. A procedure with no superglue tokens, no tokens that are common to all data slices, has zero strong functional cohesion—there are no data tokens that contribute to all outputs. A procedure with no glue tokens, that is no tokens common to more than one data slice (in procedures with more than one data slice), exhibits zero weak functional cohesion and zero adhesiveness—there are no data tokens that contribute to more than one output.

Strong functional cohesion and adhesiveness are encountered when the Bieman and Ott metrics take on a maximum value of 1.

A detailed discussion of the Bieman and Ott metrics is best left to the authors [BIE94]. However, to illustrate the character of these metrics, consider the metric for strong functional cohesion:

$$SFC(i) = SG\ [SA(i))/(tokens(i)] \tag{19-6}$$

where SG[SA(*i*)] denotes superglue tokens—the set of data tokens that lie on all data slices for a module *i*. As the ratio of superglue tokens to the total number of tokens in a module *i* increases toward a maximum value of 1, the functional cohesiveness of the module also increases.

Coupling metrics. Module coupling provides an indication of the "connectedness" of a module to other modules, global data, and the outside environment. In Chapter 13, coupling was discussed in qualitative terms.

Dhama [DHA95] has proposed a metric for module coupling that encompasses data and control flow coupling, global coupling, and environmental coupling. The measures required to compute module coupling are defined in terms of each of the three coupling types noted previously.

WebRef

A paper, "A Software Metric System for Module Coupling," can be downloaded from **www.isse.gmu.edu/faculty/ofut/rsrch/abstracts/mj-coupling.html**

For data and control flow coupling,

d_i = number of input data parameters
c_i = number of input control parameters
d_o = number of output data parameters
c_o = number of output control parameters

For global coupling,

g_d = number of global variables used as data
g_c = number of global variables used as control

For environmental coupling,

w = number of modules called (fan-out)
r = number of modules calling the module under consideration (fan-in)

Using these measures, a module coupling indicator, m_c, is defined in the following way:

$$m_c = k/M$$

where k = 1, a proportionality constant[8] and

$$M = d_i + (a \times c_i) + d_o + (b \times c_o) + g_d + (c \times g_c) + w + r$$

where $a = b = c = 2$.

The higher the value of m_c, the lower is the overall module coupling. For example, if a module has single input and output data parameters, accesses no global data, and is called by a single module,

$$m_c = 1/(1 + 0 + 1 + 0 + 0 + + 0 + 1 + 0) = 1/3 = 0.33$$

We would expect that such a module exhibits low coupling. Hence, a value of m_c = 0.33 implies low coupling. Alternatively, if a module has five input and five output data parameters, an equal number of control parameters, accesses ten items of global data, has a fan-in of 3 and a fan-out of 4,

$$m_c = 1/[5 + (2 \times 5) + 5 + (2 \times 5) + 10 + 0 + 3 + 4] = 0.02$$

and the implied coupling would be high.

8 The author [DHA95] notes that the values of k and a, b, and c (discussed in the next equation) may be adjusted as more experimental verification occurs.

In order to have the coupling metric move upward as the degree of coupling increases (an important attribute discussed in Section 18.2.3), a revised coupling metric may be defined as

$$C = 1 - m_C$$

where the degree of coupling increases nonlinearly between a minimum value in the range 0.66 to a maximum value that approaches 1.0.

Complexity metrics. A variety of software metrics can be computed to determine the complexity of program control flow. Many of these are based on the flow graph. As we discussed in Chapter 17, a graph is a representation composed of nodes and links (also called *edges*). When the links (edges) are *directed,* the flow graph is a directed graph.

McCabe and Watson [MCC94] identify a number of important uses for complexity metrics:

> Complexity metrics can be used to predict critical information about reliability and maintainability of software systems from automatic analysis of source code [or procedural design information]. Complexity metrics also provide feedback during the software project to help control the [design activity]. During testing and maintenance, they provide detailed information about software modules to help pinpoint areas of potential instability.

Cyclomatic complexity is only one of a large number of complexity metrics.

The most widely used (and debated) complexity metric for computer software is cyclomatic complexity, originally developed by Thomas McCabe [MCC76], [MCC89] and discussed in detail in Section 17.4.2.

The McCabe metric provides a quantitative measure of testing difficulty and an indication of ultimate reliability. Experimental studies indicate distinct relationships between the McCabe metric and the number of errors existing in source code, as well as time required to find and correct such errors.

McCabe also contends that cyclomatic complexity may be used to provide a quantitative indication of maximum module size. Collecting data from a number of actual programming projects, he has found that cyclomatic complexity = 10 appears to be a practical upper limit for module size. When the cyclomatic complexity of modules exceeded this number, it became extremely difficult to adequately test a module. See Chapter 17 for a discussion of cyclomatic complexity as a guide for the design of white-box test cases.

Zuse ([ZUS90], [ZUS97]) presents an encyclopedic discussion of no fewer that 18 different categories of software complexity metrics. The author presents the basic definitions for metrics in each category (e.g., there are a number of variations on the cyclomatic complexity metric) and then analyzes and critiques each. Zuse's work is the most comprehensive published to date.

19.4.3 Interface Design Metrics

Although there is significant literature on the design of human/computer interfaces (see Chapter 15), relatively little information has been published on metrics that would provide insight into the quality and usability of the interface.

Sears [SEA93] suggests that *layout appropriateness* (LA) is a worthwhile design metric for human/computer interfaces. A typical GUI uses *layout entities*—graphic icons, text, menus, windows, and the like—to assist the user in completing tasks. To accomplish a given task using a GUI, the user must move from one layout entity to the next. The absolute and relative position of each layout entity, the frequency with which it is used, and the "cost" of the transition from one layout entity to the next all contribute to the appropriateness of the interface.

Quote:

"Beauty—the adjustment of all parts proportionately so that one cannot add or subtract or change without impairing the harmony of the whole."

Leon Alberti (1404–1472)

For a specific layout (i.e., a specific GUI design), cost can be assigned to each sequence of actions according to the following relationship:

$$\text{cost} = \Sigma\ [\text{frequency of transition}(k) \times \text{cost of transition}(k)] \tag{19-7}$$

where k is a specific transition from one layout entity to the next as a specific task is accomplished. The summation occurs across all transitions for a particular task or set of tasks required to accomplish some application function. Cost may be characterized in terms of time, processing delay, or any other reasonable value, such as the distance that a mouse must travel between layout entities. Layout appropriateness is defined as

$$\text{LA} = 100 \times [(\text{cost of LA} - \text{optimal layout})/(\text{cost of proposed layout})] \tag{19-8}$$

where LA = 100 for an optimal layout.

To compute the optimal layout for a GUI, interface real estate (the area of the screen) is divided into a grid. Each square of the grid represents a possible position for a layout entity. For a grid with N possible positions and K different layout entities to place, the number of possible layouts is represented in the following manner [SEA93]:

$$\text{number of possible layouts} = [N!/(K! \times (N - K)!] \times K! \tag{19-9}$$

Interface design metrics are fine, but above all else, be absolutely sure that your end-users like the interface and are comfortable with the interactions required.

As the number of layout positions increases, the number of possible layouts grows very large. To find the optimal (lowest cost) layout, Sears [SEA93] proposes a tree searching algorithm.

LA is used to assess different proposed GUI layouts and the sensitivity of a particular layout to changes in task descriptions (i.e., changes in the sequence and/or frequency of transitions). The interface designer can use the change in layout appropriateness, ΔLA, as a guide in choosing the best GUI layout for a particular application.

It is important to note that the selection of a GUI design can be guided with metrics such as LA, but the final arbiter should be user input based on GUI prototypes. Nielsen and Levy [NIE94] report that "one has a reasonably large chance of suc-

cess if one chooses between interface [designs] based solely on users' opinions. Users' average task performance and their subjective satisfaction with a GUI are highly correlated."

19.5 METRICS FOR SOURCE CODE

Quote:

"The Human Brain follows a more rigid set of rules [in developing algorithms] than it has been aware of."

Maurice Halstead

Halstead's theory of software science [HAL77] is one of "the best known and most thoroughly studied . . . composite measures of (software) complexity" [CUR80]. Software science proposed the first analytical "laws" for computer software.[9]

Software science assigns quantitative laws to the development of computer software, using a set of primitive measures that may be derived after code is generated or estimated once design is complete. These follow:

n_1 = the number of distinct operators that appear in a program.
n_2 = the number of distinct operands that appear in a program.
N_1 = the total number of operator occurrences.
N_2 = the total number of operand occurrences.

Halstead uses these primitive measures to develop expressions for the overall *program length, potential minimum volume* for an algorithm, the *actual volume* (number of bits required to specify a program), the *program level* (a measure of software complexity), the *language level* (a constant for a given language), and other features such as development effort, development time, and even the projected number of faults in the software.

ADVICE

Operators include all flow of control constructs, conditionals, and math operations. Operands encompass all program variables and constants.

Halstead shows that length N can be estimated

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2 \tag{19-10}$$

and program volume may be defined

$$V = N \log_2 (n_1 + n_2) \tag{19-11}$$

It should be noted that V will vary with programming language and represents the volume of information (in bits) required to specify a program.

Theoretically, a minimum volume must exist for a particular algorithm. Halstead defines a volume ratio L as the ratio of volume of the most compact form of a program to the volume of the actual program. In actuality, L must always be less than 1. In terms of primitive measures, the volume ratio may be expressed as

$$L = 2/n_1 \times n_2/N_2 \tag{19-12}$$

---

9 It should be noted that Halstead's "laws" have generated substantial controversy and that not everyone agrees that the underlying theory is correct. However, experimental verification of Halstead's findings have been made for a number of programming languages (e.g., [FEL89]).

Halstead's work is amenable to experimental verification and a large body of research has been conducted to investigate software science. A discussion of this work is beyond the scope of this text, but it can be said that good agreement has been found between analytically predicted and experimental results. For further information, see [ZUS90], [FEN91], and [ZUS97].

19.6 METRICS FOR TESTING

Although much has been written on software metrics for testing (e.g., [HET93]), the majority of metrics proposed focus on the process of testing, not the technical characteristics of the tests themselves. In general, testers must rely on analysis, design, and code metrics to guide them in the design and execution of test cases.

Testing metrics fall into two broad categories: (1) metrics that attempt to predict the likely number of tests required at various testing levels and (2) metrics that focus on test coverage for a given component.

Function-based metrics (Section 19.3.1) can be used as a predictor for overall testing effort. Various project-level characteristics (e.g., testing effort and time, errors uncovered, number of test cases produced) for past projects can be collected and correlated with the number of FP produced by a project team. The team can then project "expected values" of these characteristics for the current project.

The bang metric can provide an indication of the number of test cases required by examining the primitive measures discussed in Section 19.3.2. The number of functional primitives (FuP), data elements (DE), objects (OB), relationships (RE), states (ST), and transitions (TR) can be used to project the number and types of black-box and white-box tests for the software. For example, the number of tests associated with the human/computer interface can be estimated by (1) examining the number of transitions (TR) contained in the state transition representation of the HCI and evaluating the tests required to exercise each transition; (2) examining the number of data objects (OB) that move across the interface, and (3) the number of data elements that are input or output.

Architectural design metrics provide information on the ease or difficulty associated with integration testing (Chapter 18) and the need for specialized testing software (e.g., stubs and drivers). Cyclomatic complexity (a component-level design metric) lies at the core of basis path testing, a test case design method presented in Chapter 17. In addition, cyclomatic complexity can be used to target modules as candidates for extensive unit testing (Chapter 18). Modules with high cyclomatic complexity are more likely to be error prone than modules whose cyclomatic complexity is lower. For this reason, the tester should expend above average effort to uncover errors in such modules before they are integrated in a system. Testing effort can also be estimated using metrics derived from Halstead measures (Section 19.5). Using the definitions for program volume, V, and program level, PL, software science effort, e, can be computed as

$$\text{PL} = 1/[(n_1/2)\bullet(N_2/n_2)] \tag{19-13a}$$

$$e = V/\text{PL} \tag{19-13b}$$

The percentage of overall testing effort to be allocated to a module k can be estimated using the following relationship:

$$\text{percentage of testing effort } (k) = e(k) / \Sigma\, e(i) \tag{19-14}$$

where $e(k)$ is computed for module k using Equations (19-13) and the summation in the denominator of Equation (19-14) is the sum of software science effort across all modules of the system.

As tests are conducted, three different measures provide an indication of testing completeness. A measure of the breath of testing provides an indication of how many requirements (of the total number of requirements) have been tested. This provides an indication of the completeness of the test plan. Depth of testing is a measure of the percentage of independent basis paths covered by testing versus the total number of basis paths in the program. A reasonably accurate estimate of the number of basis paths can be computed by adding the cyclomatic complexity of all program modules. Finally, as tests are conducted and error data are collected, fault profiles may be used to rank and categorize errors uncovered. Priority indicates the severity of the problem. Fault categories provide a description of an error so that statistical error analysis can be conducted.

19.7 METRICS FOR MAINTENANCE

All of the software metrics introduced in this chapter can be used for the development of new software and the maintenance of existing software. However, metrics designed explicitly for maintenance activities have been proposed.

IEEE Std. 982.1-1988 [IEE94] suggests a *software maturity index* (SMI) that provides an indication of the stability of a software product (based on changes that occur for each release of the product). The following information is determined:

M_T = the number of modules in the current release
F_c = the number of modules in the current release that have been changed
F_a = the number of modules in the current release that have been added
F_d = the number of modules from the preceding release that were deleted in the current release

The software maturity index is computed in the following manner:

$$\text{SMI} = [M_T - (F_a + F_c + F_d)] / M_T \tag{19-15}$$

As SMI approaches 1.0, the product begins to stabilize. SMI may also be used as metric for planning software maintenance activities. The mean time to produce a release of a software product can be correlated with SMI and empirical models for maintenance effort can be developed.

19.8 SUMMARY

Software metrics provide a quantitative way to assess the quality of internal product attributes, thereby enabling the software engineer to assess quality before the product is built. Metrics provide the insight necessary to create effective analysis and design models, solid code, and thorough tests.

To be useful in a real world context, a software metric must be simple and computable, persuasive, consistent, and objective. It should be programming language independent and provide effective feedback to the software engineer.

Metrics for the analysis model focus on function, data, and behavior—the three components of the analysis model. The function point and the bang metric each provide a quantitative means for evaluating the analysis model. Metrics for design consider architecture, component-level design, and interface design issues. Architectural design metrics consider the structural aspects of the design model. Component-level design metrics provide an indication of module quality by establishing indirect measures for cohesion, coupling, and complexity. Interface design metrics provide an indication of layout appropriateness for a GUI.

Software science provides an intriguing set of metrics at the source code level. Using the number of operators and operands present in the code, software science provides a variety of metrics that can be used to assess program quality.

Few technical metrics have been proposed for direct use in software testing and maintenance. However, many other technical metrics can be used to guide the testing process and as a mechanism for assessing the maintainability of a computer program.

REFERENCES

[BAS84] Basili, V.R. and D.M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Trans. Software Engineering,* vol. SE-10, 1984, pp. 728–738.

[BIE94] Bieman, J.M. and L.M. Ott, "Measuring Functional Cohesion," *IEEE Trans. Software Engineering,* vol. SE-20, no. 8, August 1994, pp. 308–320.

[BRI96] Briand, L.C., S. Morasca, and V.R. Basili, "Property-Based Software Engineering Measurement," *IEEE Trans. Software Engineering,* vol. SE-22, no. 1, January 1996, pp. 68–85.

[CAR90] Card, D.N. and R.L. Glass, *Measuring Software Design Quality,* Prentice-Hall, 1990.

[CAV78] Cavano, J.P. and J.A. McCall, "A Framework for the Measurement of Software Quality," *Proc. ACM Software Quality Assurance Workshop,* November 1978, pp. 133–139.

[CHA89] Charette, R.N., *Software Engineering Risk Analysis and Management,* McGraw-Hill/Intertext, 1989.

[CUR80] Curtis, W. "Management and Experimentation in Software Engineering," *Proc. IEEE,* vol. 68, no. 9, September 1980.

[DAV93] Davis, A., et al., "Identifying and Measuring Quality in a Software Requirements Specification, *Proc. First Intl. Software Metrics Symposium,* IEEE, Baltimore, MD, May 1993, pp. 141–152.

[DEM81] DeMillo, R.A. and R.J. Lipton, "Software Project Forecasting," in *Software Metrics* (A.J. Perlis, F.G. Sayward, and M. Shaw, eds.), MIT Press, 1981, pp. 77–89.

[DEM82] DeMarco, T., *Controlling Software Projects,* Yourdon Press, 1982.

[DHA95] Dhama, H., "Quantitative Models of Cohesion and Coupling in Software," *Journal of Systems and Software,* vol. 29, no. 4, April 1995.

[EJI91] Ejiogu, L., *Software Engineering with Formal Metrics,* QED Publishing, 1991.

[FEL89] Felican, L. and G. Zalateu, "Validating Halstead's Theory for Pascal Programs," *IEEE Trans. Software Engineering,* vol. SE-15, no. 2, December 1989, pp. 1630–1632.

[FEN91] Fenton, N., *Software Metrics,* Chapman and Hall, 1991.

[FEN94] Fenton, N., "Software Measurement: A Necessary Scientific Basis," *IEEE Trans. Software Engineering,* vol. SE-20, no. 3, March 1994, pp. 199–206.

[GRA87] Grady, R.B. and D.L. Caswell, *Software Metrics: Establishing a Company-Wide Program,* Prentice-Hall, 1987.

[HAL77] Halstead, M., *Elements of Software Science,* North-Holland, 1977.

[HEN81] Henry, S. and D. Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Trans. Software Engineering,* vol. SE-7, no. 5, September 1981, pp. 510–518.

[HET93] Hetzel, B., *Making Software Measurement Work,* QED Publishing, 1993.

[IEE94] *Software Engineering Standards,* 1994 edition, IEEE, 1994.

[KYB84] Kyburg, H.E., *Theory and Measurement,* Cambridge University Press, 1984.

[MCC76] McCabe, T.J., "A Software Complexity Measure," *IEEE Trans. Software Engineering,* vol. SE-2, December 1976, pp. 308–320.

[MCC77] McCall, J., P. Richards, and G. Walters, "Factors in Software Quality," three volumes, NTIS AD-A049-014, 015, 055, November 1977.

[MCC89] McCabe, T.J. and C.W. Butler, "Design Complexity Measurement and Testing," *CACM,* vol. 32, no. 12, December 1989, pp. 1415–1425.

[MCC94] McCabe, T.J. and A.H. Watson, "Software Complexity," *Crosstalk,* vol. 7, no. 12, December 1994, pp. 5–9.

[NIE94] Nielsen, J., and J. Levy, "Measuring Usability: Preference vs. Performance," *CACM,* vol. 37, no. 4, April 1994, pp. 65–75.

[ROC94] Roche, J.M., "Software Metrics and Measurement Principles," *Software Engineering Notes,* ACM, vol. 19, no. 1, January 1994, pp. 76–85.

[SEA93] Sears, A., "Layout Appropriateness: A Metric for Evaluating User Interface Widget Layout, *IEEE Trans. Software Engineering,* vol. SE-19, no. 7, July 1993, pp. 707–719.

[USA87] *Management Quality Insight,* AFCSP 800-14 (U.S. Air Force), January 20, 1987.

[ZUS90] Zuse, H., *Software Complexity: Measures and Methods,* DeGruyter, 1990.

[ZUS97] Zuse, H., *A Framework of Software Measurement,* DeGruyter, 1997.

PROBLEMS AND POINTS TO PONDER

19.1. Measurement theory is an advanced topic that has a strong bearing on software metrics. Using [ZUS97], [FEN91], [ZUS90], [KYB84] or some other source, write a brief paper that outlines the main tenets of measurement theory. Individual project: Develop a presentation on the subject and present it to your class.

19.2. McCall's quality factors were developed during the 1970s. Almost every aspect of computing has changed dramatically since the time that they were developed, and yet, McCall's factors continue to apply to modern software. Can you draw any conclusions based on this fact?

19.3. Why is it that a single, all-encompassing metric cannot be developed for program complexity or program quality?

19.4. Review the analysis model you developed as part of Problem 12.13. Using the guidelines presented in Section 19.3.1, develop an estimate for the number of function points associated with PHTRS.

19.5. Review the analysis model you developed as part of Problem 12.13. Using the guidelines presented in Section 19.3.2, develop primitive counts for the bang metric. Is the PHTRS system function strong or data strong?

19.6. Compute the value of the bang metric using the measures you developed in Problem 19.5.

19.7. Create a complete design model for a system that is proposed by your instructor. Compute structural and data complexity using the metrics described in Section 19.4.1. Also compute the Henry-Kafura and morphology metrics for the design model.

19.8. A major information system has 1140 modules. There are 96 modules that perform control and coordination functions and 490 modules whose function depends on prior processing. The system processes approximately 220 data objects that each have an average of three attributes. There are 140 unique data base items and 90 different database segments. Finally, 600 modules have single entry and exit points. Compute the DSQI for this system.

19.9. Research Bieman and Ott's [BIE94] paper and develop a complete example that illustrates the computation of their cohesion metric. Be sure to indicate how data slices, data tokens, glue, and superglue tokens are determined.

19.10. Select five modules in an existing computer program. Using Dhama's metric described in Section 19.4.2, compute the coupling value for each module.

19.11. Develop a software tool that will compute cyclomatic complexity for a programming language module. You may choose the language.

19.12. Develop a software tool that will compute layout appropriateness for a GUI. The tool should enable you to assign the transition cost between layout entities. (Note: Recognize that the size of the potential population of layout alternatives grows very large as the number of possible grid positions grows.)

19.13. Develop a small software tool that will perform a Halstead analysis on programming language source code of your choosing.

19.14. Research the literature and write a paper on the relationship of Halstead's metric and McCabe's metric on software quality (as measured by error count). Are the data compelling? Recommend guidelines for the application of these metrics.

19.15. Research the literature for any recent papers on metrics specifically developed to assist in test case design. Present your findings to the class.

19.16. A legacy system has 940 modules. The latest release required that 90 of these modules be changed. In addition, 40 new modules were added and 12 old modules were removed. Compute the software maturity index for the system.

FURTHER READING AND INFORMATION SOURCES

There are a surprisingly large number of books that are dedicated to software metrics, although the majority focus on process and project metrics to the exclusion of technical metrics. Zuse [ZUS97] has written the most thorough treatment of technical metrics published to date.

Books by Card and Glass [CAR90], Zuse [ZUS90], Fenton {FEN91], Ejiogu [EJI91], Moeller and Paulish *(Software Metrics,* Chapman and Hall, 1993), and Hetzel [HET93] all address technical metrics in some detail. Oman and Pfleeger (*Applying Software Metrics,* IEEE Computer Society Press, 1997) have edited an anthology of important papers on software metrics. In addition, the following books are worth examining:

Conte, S.D., H.E. Dunsmore, and V.Y. Shen. *Software Engineering Metrics and Models,* Benjamin/Cummings, 1984.

Fenton, N.E. and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach,* 2nd ed., PWS Publishing Co., 1998.

Grady, R.B. *Practical Software Metrics for Project Management and Process Improvement,* Prentice-Hall, 1992.

Perlis, A., et al., *Software Metrics: An Analysis and Evaluation,* MIT Press, 1981.

Sheppard, M., *Software Engineering Metrics,* McGraw-Hill, 1992.

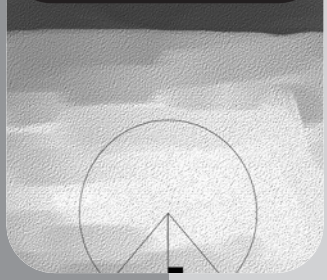
The theory of software measurement is presented by Denvir, Herman, and Whitty in an edited collection of papers (*Proceedings of the International BCS-FACS Workshop: Formal Aspects of Measurement,* Springer-Verlag, 1992). Shepperd (*Foundations of Software Measurement,* Prentice-Hall, 1996) also addresses measurement theory in some detail.

A comprehensive summary of dozens of useful software metrics is presented in [IEE94]. In general, a discussion of each metric has been distilled to the essential "primitives" (measures) required to compute the metric and the appropriate relationships to effect the computation. An appendix provides discussion and many references.

A wide variety of information sources on technical metrics and related subjects is available on the Internet. An up-to-date list of World Wide Web references that are relevant to technical metrics can be found at the SEPA Web site:
http://www.mhhe.com/engcs/compsci/pressman/resources/tech-metrics.mhtml

PART

Four



OBJECT-ORIENTED SOFTWARE ENGINEERING

In this part of *Software Engineering: A Practitioner's Approach,* we consider the technical concepts, methods, and measurements that are applicable for the analysis, design, and testing of object-oriented software. In the chapters that follow, we address the following questions:

- What basic concepts and principles are applicable to object-oriented thinking?
- How do conventional and object-oriented approaches differ?
- How should object-oriented software projects be planned and managed?
- What is object-oriented analysis and how do its various models enable a software engineer to understand classes, their relationships, and behaviors?
- What are the elements of an object-oriented design model?
- What basic concepts and principles are applicable to the software testing for object-oriented software?
- How do testing strategies and test case design methods change when object-oriented software is considered?
- What technical metrics are available for assessing the quality of object-oriented software?

Once these questions are answered, you'll understand how to analyze, design, implement, and test software using the object-oriented paradigm.