CHAPTER

23 PRODUCT METRICS

KEY CONCEPTS

A key element of any engineering process is measurement. You can use measures to better understand the attributes of the models that you create and to assess the quality of the engineered products or systems that you build. But unlike other engineering disciplines, software engineering is not grounded in the basic quantitative laws of physics. Direct measures, such as voltage, mass, velocity, or temperature, are uncommon in the software world. Because software measures and metrics are often indirect, they are open to debate. Fenton [Fen91] addresses this issue when he states:

> Measurement is the process by which numbers or symbols are assigned to the attributes of entities in the real world in such a way as to define them according to clearly defined rules. . . . In the physical sciences, medicine, economics, and more recently the social sciences, we are now able to measure attributes that were previously thought to

QUICK LOOK

What is it? By its nature, engineering is a quantitative discipline. Product metrics help software engineers gain insight into the design and construction of the software they build by focusing on specific, measurable attributes of software engineering work products.

Who does it? Software engineers use product metrics to help them build higher-quality software.

Why is it important? There will always be a qualitative element to the creation of computer software. The problem is that qualitative assessment may not be enough. You need objective criteria to help guide the design of data, architecture, interfaces, and components. When testing, you need quantitative guidance that will help in the selection of test cases and their targets. Product metrics provide a basis from which analysis, design, coding, and testing can be conducted more objectively and assessed more quantitatively.

What are the steps? The first step in the measurement process is to derive the software measures and metrics that are appropriate for the representation of software that is being considered. Next, data required to derive the formulated metrics are collected. Once computed, appropriate metrics are analyzed based on preestablished 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 requirements and design models, source code, or test cases. In some instances, it may also lead to modification of the software process itself.

What is the work product? Product metrics that are computed from data collected from the requirements 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 product 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.

> be unmeasurable. . . . Of course, such measurements are not as refined as many measurements in the physical sciences . . . , but they exist [and important decisions are made based on them]. We feel that the obligation to attempt to "measure the unmeasurable" in order to improve our understanding of particular entities is as powerful in software engineering as in any discipline.

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 product metrics for computer software are imperfect, they can provide you with a systematic way to assess quality based on a set of clearly defined rules. They also provide you with on-the-spot, rather than after-the-fact, insight. This enables you to discover and correct potential problems before they become catastrophic defects.

In this chapter, I present measures that can be used to assess the quality of the product as it is being engineered. These measures of internal product attributes provide you with a real-time indication of the efficacy of the requirements, design, and code models; the effectiveness of test cases; and the overall quality of the software to be built.

23.1 A Framework for Product Metrics

Quote:

"A science is as mature as its measurement tools."

Louis Pasteur

As I noted in the introduction, 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., [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 that guide the definition of product metrics for software.

23.1.1 Measures, Metrics, and Indicators

? What's the difference between a measure and a metric?

Although the terms *measure, measurement,* and *metrics* are often used interchangeably, it is important to note the subtle differences between them. Because *measure* can be used either as a noun or a verb, definitions of the term can become confusing. Within the software engineering context, a *measure* provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process. *Measurement* is the act of determining a measure. The *IEEE Standard Glossary of Software Engineering Terminology* [IEE93b] defines *metric* as "a quantitative measure of the degree to which a system, component, or process possesses a given attribute."

When a single data point has been collected (e.g., the number of errors uncovered within a single software component), a measure has been established. Measurement occurs as the result of the collection of one or more data points (e.g., a number of component reviews and unit tests are investigated to collect measures of the number

of errors for each). A software metric relates the individual measures in some way (e.g., the average number of errors found per review or the average number of errors found per unit test).

KEY POINT
An indicator is a metric or metrics that provide insight into the process, the product, or the project.

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

23.1.2 The Challenge of Product Metrics

Over the past four 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 the use of alternative fuels, 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.

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

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 Chapter 8). These measures and the metrics derived from them can be used as independent indicators of the quality of requirements 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 product 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 product metrics are. That is, how closely aligned are product 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 [product 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.

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

Each of the "challenges" noted here is a cause for caution, but it is no reason to dismiss product metrics.[1] Measurement is essential if quality is to be achieved.

23.1.3 Measurement Principles

Before I introduce a series of product 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 for you 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 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.
- *Interpretation.* The evaluation of metrics resulting in insight into the quality of the representation.
- *Feedback.* Recommendations derived from the interpretation of product metrics transmitted to the software team.

Software metrics will be useful only if they are characterized effectively and validated so that their worth is proven. The following principles [Let03b] are representative of many that can be proposed for metrics characterization and validation:

ADVICE

In reality, many product metrics in use today do not conform to these principles as well as they should. But that doesn't mean that they have no value—just be careful when you use them, understanding that they are intended to provide insight, not hard scientific verification.

- *A metric should have desirable mathematical properties.* That is, the metric's value should be in a meaningful range (e.g., 0 to 1, where 0 truly means absence, 1 indicates the maximum value, and 0.5 represents the "halfway point"). Also, a metric that purports to be on a rational scale should not be composed of components that are only measured on an ordinal scale.
- *When a metric represents a software characteristic that increases when positive traits occur or decreases when undesirable traits are encountered, the value of the metric should increase or decrease in the same manner.*
- *Each metric should be validated empirically in a wide variety of contexts before being published or used to make decisions.* A metric should measure the factor of interest, independently of other factors. It should "scale up" to large systems and work in a variety of programming languages and system domains.

1 Although criticism of specific metrics is common in the literature, many critiques focus on esoteric issues and miss the primary objective of metrics in the real world: to help the software engineer establish a systematic and objective way to gain insight into his or her work and to improve product quality as a result.

Although formulation, characterization, and validation are critical, collection and analysis are the activities that drive the measurement process. Roche [Roc94] suggests the following principles for these activities: (1) whenever possible, data collection and analysis should be automated; (2) valid statistical techniques should be applied to establish relationships between internal product attributes and external quality characteristics (e.g., whether the level of architectural complexity correlates with the number of defects reported in production use); and (3) interpretative guidelines and recommendations should be established for each metric.

23.1.4 Goal-Oriented Software Measurement

WebRef
A useful discussion of GQM can be found at **www.thedacs.com/GoldPractices/practices/gqma.html**.

The *Goal/Question/Metric* (GQM) paradigm has been developed by Basili and Weiss [Bas84] as a technique for identifying meaningful metrics for any part of the software process. GQM emphasizes the need to (1) establish an explicit measurement *goal* that is specific to the process activity or product characteristic that is to be assessed, (2) define a set of *questions* that must be answered in order to achieve the goal, and (3) identify well-formulated *metrics* that help to answer these questions.

A *goal definition template* [Bas94] can be used to define each measurement goal. The template takes the form:

> **Analyze** {the name of activity or attribute to be measured} **for the purpose of** {the overall objective of the analysis[2]} **with respect to** {the aspect of the activity or attribute that is considered} **from the viewpoint of** {the people who have an interest in the measurement} **in the context of** {the environment in which the measurement takes place}.

As an example, consider a goal definition template for *SafeHome*:

> **Analyze** the *SafeHome* software architecture **for the purpose of** evaluating architectural components **with respect to** the ability to make *SafeHome* more extensible **from the viewpoint of** the software engineers performing the work **in the context of** product enhancement over the next three years.

With a measurement goal explicitly defined, a set of questions is developed. Answers to these questions help the software team (or other stakeholders) to determine whether the measurement goal has been achieved. Among the questions that might be asked are:

Q_1: Are architectural components characterized in a manner that compartmentalizes function and related data?

Q_2: Is the complexity of each component within bounds that will facilitate modification and extension?

Each of these questions should be answered quantitatively, using one or more measures and metrics. For example, a metric that provides an indication of the cohesion

2 van Solingen and Berghout [Sol99] suggest that the objective is almost always "understanding, controlling or improving" the process activity or product attribute.

(Chapter 8) of an architectural component might be useful in answering Q_1. Metrics discussed later in this chapter might provide insight for Q_2. In every case, the metrics that are chosen (or derived) should conform to the measurement principles discussed in Section 23.1.3 and the measurement attributes discussed in Section 23.1.5.

23.1.5 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.
- *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 requirements 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 you with information that can lead to a higher-quality end product.

ADVICE

Experience indicates that a product 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 is unlikely that the metric will be widely adopted.

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 Section 23.2.1)—a measure of the "functionality" delivered by the software. 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

3 An equally vigorous counterargument can be made. Such is the nature of software metrics.

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.

SAFEHOME

Debating Product Metrics

The scene: Vinod's cubicle.

The players: Vinod, Jamie, and Ed—members of the *SafeHome* software engineering team who are continuing work of component-level design and test-case design.

The conversation:

Vinod: Doug [Doug Miller, software engineering manager] told me that we should all use product metrics, but he was kind of vague. He also said that he wouldn't push the matter . . . that using them was up to us.

Jamie: That's good, 'cause there's no way I have time to start measuring stuff. We're fighting to maintain the schedule as it is.

Ed: I agree with Jamie. We're up against it, here . . . no time.

Vinod: Yeah, I know, but there's probably some merit to using them.

Jamie: I'm not arguing that, Vinod, it's a time thing . . . and I for one don't have any to spare.

Vinod: But what if measuring saves you time?

Ed: Wrong, it takes time and like Jamie said . . .

Vinod: No, wait . . . what it saves us is time?

Jamie: How?

Vinod: Rework . . . that's how. If a measure we use helps us to avoid one major or even moderate problem, and that saves us from having to rework a part of the system, we save time. No?

Ed: It's possible, I suppose, but can you guarantee that some product metric will help us find a problem?

Vinod: Can you guarantee that it won't?

Jamie: So what are you proposing?"

Vinod: I think we should select a few design metrics, probably class-oriented, and use them as part of our review process for every component we develop.

Ed: I'm not real familiar with class-oriented metrics.

Vinod: I'll spend some time checking them out and make a recommendation . . . okay with you guys?

[Ed and Jamie nod without much enthusiasm.]

23.2 METRICS FOR THE REQUIREMENTS MODEL

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

Although relatively few analysis and specification metrics have appeared in the literature, it is possible to adapt metrics that are often used for project estimation and apply them in this context. These metrics examine the requirements model with the intent of predicting the "size" of the resultant system. Size is sometimes (but not always) an indicator of design complexity and is almost always an indicator of increased coding, integration, and testing effort.

23.2.1 Function-Based Metrics

WebRef
Much useful information about function points can be obtained at **www.ifpug.org** and **www.functionpoints.com**.

The *function point* (FP) *metric* can be used effectively as a means for measuring the functionality delivered by a system.[4] Using historical data, the FP metric can then be used to (1) estimate the cost or effort required to design, code, and test the software; (2) predict the number of errors that will be encountered during testing; and (3) forecast the number of components and/or the number of projected source lines in the implemented system.

Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and qualitative assessments of software complexity. Information domain values are defined in the following manner:[5]

Number of external inputs (EIs). Each *external input* originates from a user or is transmitted from another application and provides distinct application-oriented data or control information. Inputs are often used to update *internal logical files* (ILFs). Inputs should be distinguished from inquiries, which are counted separately.

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

Number of external inquiries (EQs). An *external inquiry* is defined as an online input that results in the generation of some immediate software response in the form of an online output (often retrieved from an ILF).

Number of internal logical files (ILFs). Each *internal logical file* is a logical grouping of data that resides within the application's boundary and is maintained via external inputs.

Number of external interface files (EIFs). Each *external interface file* is a logical grouping of data that resides external to the application but provides information that may be of use to the application.

Once these data have been collected, the table in Figure 23.1 is completed and a complexity value is associated with each count. Organizations that use function point methods develop criteria for determining whether a particular entry is simple, average, or complex. Nonetheless, the determination of complexity is somewhat subjective.

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

$$\text{FP} = \text{count total} \times [0.65 + 0.01 \times \Sigma (F_i)] \tag{23.1}$$

where count total is the sum of all FP entries obtained from Figure 23.1.

4 Hundreds of books, papers, and articles have been written on FP metrics. A worthwhile bibliography can be found at [IFP05].

5 In actuality, the definition of information domain values and the manner in which they are counted are a bit more complex. The interested reader should see [IFP01] for more details.

FIGURE 23.1 Computing function points

Information Domain Value	Count		Weighting factor: Simple	Average	Complex		
External Inputs (EIs)		×	3	4	6	=	
External Outputs (EOs)		×	4	5	7	=	
External Inquiries (EQs)		×	3	4	6	=	
Internal Logical Files (ILFs)		×	7	10	15	=	
External Interface Files (EIFs)		×	5	7	10	=	
Count total							

KEY POINT
Value adjustment factors are used to provide an indication of problem complexity.

The F_i (i = 1 to 14) are *value adjustment factors* (VAF) based on responses to the following questions [Lon02]:

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

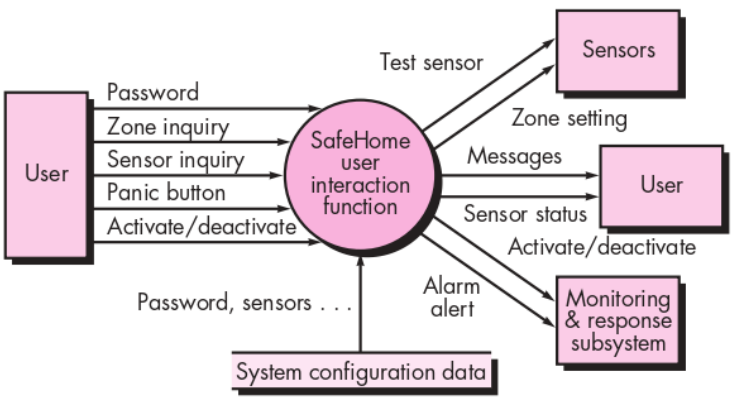
Each of these questions is answered using a scale that ranges from 0 (not important or applicable) to 5 (absolutely essential). The constant values in Equation (23.1) and the weighting factors that are applied to information domain counts are determined empirically.

WebRef
An online FP calculator can be found at **irb.cs.uni-magdeburg.de/sw-eng/us/java/fp/**.

To illustrate the use of the FP metric in this context, we consider a simple analysis model representation, illustrated in Figure 23.2. Referring to the figure, a data flow diagram (Chapter 7) for a function within the *SafeHome* software is represented.

FIGURE 23.2

A data flow model for *SafeHome* software



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 a set of key information domain measures required for computation of the function point metric. Three external inputs—**password, panic button**, and **activate/deactivate**—are shown in the figure along with two external inquiries—**zone inquiry** and **sensor inquiry**. One ILF (**system configuration file**) is shown. Two external outputs (**messages** and **sensor status**) and four EIFs (**test sensor, zone setting, activate/deactivate,** and **alarm alert**) are also present. These data, along with the appropriate complexity, are shown in Figure 23.3.

The count total shown in Figure 23.3 must be adjusted using Equation (23.1). For the purposes of this example, we assume that $\Sigma(F_i)$ is 46 (a moderately complex product). Therefore,

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

Based on the projected FP value derived from the requirements model, the project team can estimate the overall implemented size of the *SafeHome* user interaction

FIGURE 23.3

Computing function points

Information Domain Value	Count		Weighting factor: Simple	Average	Complex		
External Inputs (EIs)	3	×	3	4	6	=	9
External Outputs (EOs)	2	×	4	5	7	=	8
External Inquiries (EQs)	2	×	3	4	6	=	6
Internal Logical Files (ILFs)	1	×	7	10	15	=	7
External Interface Files (EIFs)	4	×	5	7	10	=	20
Count total							50

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

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 requirements model rather than preliminary estimates. Assume further that past projects have found an average of three errors per function point during requirements and design reviews and four errors per function point during unit and integration testing. These data can ultimately help you assess the completeness of your review and testing activities.

Uemura and his colleagues [Uem99] suggest that function points can also be computed from UML class and sequence diagrams. If you have further interest, see [Uem99] for details.

23.2.2 Metrics for Specification Quality

Davis and his colleagues [Dav93] propose a list of characteristics that can be used to assess the quality of the requirements 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. 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.

KEY POINT

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

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:

$$Q_1 = \frac{n_{ui}}{n_r}$$

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

$$Q_2 = \frac{n_u}{n_i \times n_s}$$

where n_u is the number of unique functional requirements, n_i is the number of inputs (stimuli) defined or implied by the specification, and n_s is the number of states

Quote:

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

Galileo

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, you must consider the degree to which requirements have been validated:

$$Q_3 = \frac{n_c}{n_c + n_{nv}}$$

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.

23.3 Metrics for the Design Model

KEY POINT

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

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 as indicators 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, I examine some of the more common design metrics for computer software. Each can provide you with improved insight, and all can help the design to evolve to a higher level of quality.

23.3.1 Architectural Design Metrics

Architectural design metrics focus on characteristics of the program architecture (Chapter 9) with an emphasis on the architectural structure and the effectiveness of modules or components within the architecture. 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.

For hierarchical architectures (e.g., call-and-return architectures), *structural complexity* of a module i is defined in the following manner:

$$S(i) = f^2_{\text{out}}(i)$$

where $f_{\text{out}}(i)$ is the fan-out[6] of module i.

6 *Fan-out* is defined as the number of modules immediately subordinate to module i; that is, the number of modules that are directly invoked by module i.

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

$$D(i) = \frac{v(i)}{f_{\text{out}}(i) + 1}$$

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

$$C(i) = S(i) + D(i)$$

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.

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 the call-and-return architecture in Figure 23.4, the following metrics can be defined:

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

where n is the number of nodes and a is the number of arcs. For the architecture shown in Figure 23.4,

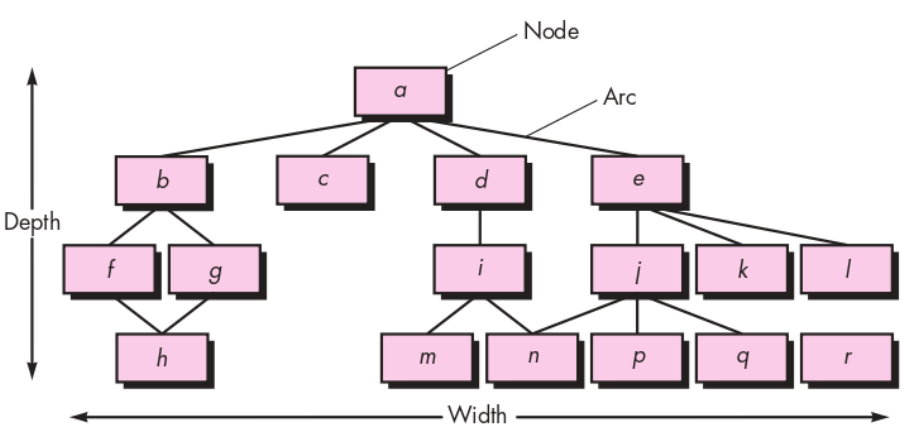
Size = 17 + 18 = 35

Depth = longest path from the root (top) node to a leaf node. For the architecture shown in Figure 23.4, depth = 4.

Width = maximum number of nodes at any one level of the architecture. For the architecture shown in Figure 23.4, width = 6.

The arc-to-node ratio, $r = a/n$, 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 23.4, $r = 18/17 = 1.06$.

FIGURE 23.4
Morphology metrics



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 = total number of modules defined in the program architecture
S_2 = 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 = number of modules whose correct function depends on prior processing
S_4 = number of database items (includes data objects and all attributes that define objects)
S_5 = total number of unique database items
S_6 = number of database segments (different records or individual objects)
S_7 = 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: $D_2 = 1 - \frac{S_2}{S_1}$

Modules not dependent on prior processing: $D_3 = 1 - \frac{S_3}{S_1}$

Database size: $D_4 = 1 - \frac{S_5}{S_4}$

Database compartmentalization: $D_5 = 1 - \frac{S_6}{S_4}$

Module entrance/exit characteristic: $D_6 = 1 - \frac{S_7}{S_1}$

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

$$\text{DSQI} = \Sigma\, w_i D_i$$

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.

23.3.2 Metrics for Object-Oriented Design

There is much about object-oriented design that is subjective—an experienced designer "knows" how to characterize an OO system so that it will effectively implement customer requirements. But, as an OO design model grows in size and complexity, a more objective view of the characteristics of the design can benefit both the experienced designer (who gains additional insight) and the novice (who obtains an indication of quality that would otherwise be unavailable).

In a detailed treatment of software metrics for OO systems, Whitmire [Whi97] describes nine distinct and measurable characteristics of an OO design:

? What characteristics can be measured when we assess an OO design?

Size. Size is defined in terms of four views: population, volume, length, and functionality. *Population* is measured by taking a static count of OO entities such as classes or operations. *Volume* measures are identical to population measures but are collected dynamically—at a given instant of time. *Length* is a measure of a chain of interconnected design elements (e.g., the depth of an inheritance tree is a measure of length). *Functionality* metrics provide an indirect indication of the value delivered to the customer by an OO application.

Complexity. Like size, there are many differing views of software complexity [Zus97]. Whitmire views complexity in terms of structural characteristics by examining how classes of an OO design are interrelated to one another.

Coupling. The physical connections between elements of the OO design (e.g., the number of collaborations between classes or the number of messages passed between objects) represent coupling within an OO system.

Sufficiency. Whitmire defines *sufficiency* as "the degree to which an abstraction possesses the features required of it, or the degree to which a design component possesses features in its abstraction, from the point of view of the current application." Stated another way, we ask: "What properties does this abstraction (class) need to possess to be useful to me?" [Whi97]. In essence, a design component (e.g., a class) is *sufficient* if it fully reflects all properties of the application domain object that it is modeling—that is, that the abstraction (class) possesses the features required of it.

Completeness. The only difference between completeness and sufficiency is "the feature set against which we compare the abstraction or design component" [Whi97]. Sufficiency compares the abstraction from the point of view of the current application. *Completeness* considers multiple points of view, asking the question: "What properties are required to fully represent the problem domain object?" Because the criterion for completeness considers different points of view, it has an indirect implication about the degree to which the abstraction or design component can be reused.

Quote:

"Many of the decisions for which I had to rely on folklore and myth can now be made using quantitative data."

Scott Whitmire

Cohesion. Like its counterpart in conventional software, an OO component should be designed in a manner that has all operations working together to achieve a single, well-defined purpose. The cohesiveness of a class is determined by examining the degree to which "the set of properties it possesses is part of the problem or design domain" [Whi97].

Primitiveness. A characteristic that is similar to simplicity, primitiveness (applied to both operations and classes) is the degree to which an operation is atomic—that is, the operation cannot be constructed out of a sequence of other operations contained within a class. A class that exhibits a high degree of primitiveness encapsulates only primitive operations.

Similarity. The degree to which two or more classes are similar in terms of their structure, function, behavior, or purpose is indicated by this measure.

Volatility. As I have noted many times throughout this book, design changes can occur when requirements are modified or when modifications occur in other parts of an application, resulting in mandatory adaptation of the design component in question. Volatility of an OO design component measures the likelihood that a change will occur.

In reality, product metrics for OO systems can be applied not only to the design model, but also the requirements model. In the sections that follow, I discuss metrics that provide an indication of quality at the OO class level and the operation level. In addition, metrics applicable for project management and testing are also explored.

23.3.3 Class-Oriented Metrics—The CK Metrics Suite

The class is the fundamental unit of an OO system. Therefore, measures and metrics for an individual class, the class hierarchy, and class collaborations will be invaluable when you are required to assess OO design quality. A class encapsulates data and the function that manipulate the data. It is often the "parent" for subclasses (sometimes called children) that inherit its attributes and operations. It often collaborates with other classes. Each of these characteristics can be used as the basis for measurement.[7]

Chidamber and Kemerer have proposed one of the most widely referenced sets of OO software metrics [Chi94]. Often referred to as the *CK metrics suite,* the authors have proposed six class-based design metrics for OO systems.[8]

Weighted methods per class (WMC). Assume that n methods of complexity c_1, $c_2, \ldots, c_n$ are defined for a class **C**. The specific complexity metric that is chosen

7 It should be noted that the validity of some of the metrics discussed in this chapter is currently debated in the technical literature. Those who champion measurement theory demand a degree of formalism that some OO metrics do not provide. However, it is reasonable to state that the metrics noted provide useful insight for the software engineer.

8 Chidamber, Darcy, and Kemerer use the term *methods* rather than *operations*. Their usage of the term is reflected in this section.

(e.g., cyclomatic complexity) should be normalized so that nominal complexity for a method takes on a value of 1.0.

$$\text{WMC} = \Sigma c_i$$

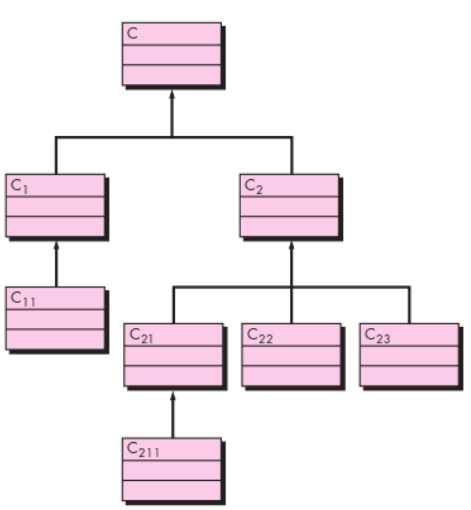
for $i = 1$ to n. The number of methods and their complexity are reasonable indicators of the amount of effort required to implement and test a class. In addition, the larger the number of methods, the more complex is the inheritance tree (all subclasses inherit the methods of their parents). Finally, as the number of methods grows for a given class, it is likely to become more and more application specific, thereby limiting potential reuse. For all of these reasons, WMC should be kept as low as is reasonable.

Although it would seem relatively straightforward to develop a count for the number of methods in a class, the problem is actually more complex than it seems. A consistent counting approach [Chu95] should be developed.

Depth of the inheritance tree (DIT). This metric is "the maximum length from the node to the root of the tree" [Chi94]. Referring to Figure 23.5, the value of DIT for the class hierarchy shown is 4. As DIT grows, it is likely that lower-level classes will inherit many methods. This leads to potential difficulties when attempting to predict the behavior of a class. A deep class hierarchy (DIT is large) also leads to greater design complexity. On the positive side, large DIT values imply that many methods may be reused.

Number of children (NOC). The subclasses that are immediately subordinate to a class in the class hierarchy are termed its children. Referring to Figure 23.5, class $\mathbf{C_2}$ has three children—subclasses $\mathbf{C_{21}}$, $\mathbf{C_{22}}$, and $\mathbf{C_{23}}$. As the number of children grows, reuse increases, but also, as NOC increases, the abstraction represented by the parent class can be diluted if some of the children are not appropriate members of the parent class. As NOC increases, the amount of testing (required to exercise each child in its operational context) will also increase.

FIGURE 23.5
A class hierarchy



The concepts of coupling and cohesion apply to both conventional and OO software. Keep class coupling low and class and operation cohesion high.

Coupling between object classes (CBO). The CRC model (Chapter 6) may be used to determine the value for CBO. In essence, CBO is the number of collaborations listed for a class on its CRC index card.[9] As CBO increases, it is likely that the reusability of a class will decrease. High values of CBO also complicate modifications and the testing that ensues when modifications are made. In general, the CBO values for each class should be kept as low as is reasonable. This is consistent with the general guideline to reduce coupling in conventional software.

Response for a class (RFC). The response set of a class is "a set of methods that can potentially be executed in response to a message received by an object of that class" [Chi94]. RFC is the number of methods in the response set. As RFC increases, the effort required for testing also increases because the test sequence (Chapter 19) grows. It also follows that, as RFC increases, the overall design complexity of the class increases.

Lack of cohesion in methods (LCOM). Each method within a class **C** accesses one or more attributes (also called instance variables). LCOM is the number of methods that access one or more of the same attributes.[10] If no methods access the same attributes, then LCOM = 0. To illustrate the case where LCOM ≠ 0, consider a class with six methods. Four of the methods have one or more attributes in common (i.e., they access common attributes). Therefore, LCOM = 4. If LCOM is high, methods may be coupled to one another via attributes. This increases the complexity of the class design. Although there are cases in which a high value for LCOM is justifiable, it is desirable to keep cohesion high; that is, keep LCOM low.[11]

SAFEHOME

Applying CK Metrics

The scene: Vinod's cubicle.

The players: Vinod, Jamie, Shakira, and Ed—members of the *SafeHome* software engineering team who are continuing to work on component-level design and test-case design.

The conversation:

Vinod: Did you guys get a chance to read the description of the CK metrics suite I sent you on Wednesday and make those measurements?

Shakira: Wasn't too complicated. I went back to my UML class and sequence diagrams, like you suggested, and got rough counts for DIT, RFC, and LCOM. I couldn't find the CRC model, so I didn't count CBO.

Jamie (smiling): You couldn't find the CRC model because I had it.

Shakira: That's what I love about this team, superb communication.

9 If CRC index cards are developed manually, completeness and consistency must be assessed before CBO can be determined reliably.

10 The formal definition is a bit more complex. See [Chi94] for details.

11 The LCOM metric provides useful insight in some situations, but it can be misleading in others. For example, keeping coupling encapsulated within a class increases the cohesion of the system as a whole. Therefore in at least one important sense, higher LCOM actually suggests that a class may have higher cohesion, not lower.

Vinod: I did my counts . . . did you guys develop numbers for the CK metrics?

[Jamie and Ed nod in the affirmative.]

Jamie: Since I had the CRC cards, I took a look at CBO and it looked pretty uniform across most of the classes. There was one exception, which I noted.

Ed: There are a few classes where RFC is pretty high, compared with the averages . . . maybe we should take a look at simplifying them.

Jamie: Maybe yes, maybe no. I'm still concerned about time, and I don't want to fix stuff that isn't really broken.

Vinod: I agree with that. Maybe we should look for classes that have bad numbers in at least two or more of the CK metrics. Kind of two strikes and you're modified.

Shakira (looking over Ed's list of classes with high RFC): Look, see this class, it's got a high LCOM as well as a high RFC. Two strikes?

Vinod: Yeah I think so . . . it'll be difficult to implement because of complexity and difficult to test for the same reason. Probably worth designing two separate classes to achieve the same behavior.

Jamie: You think modifying it'll save us time?

Vinod: Over the long haul, yes.

Quote:

"Analyzing OO software in order to evaluate its quality is becoming increasingly important as the [OO] paradigm continues to increase in popularity."

Rachel Harrison et al.

23.3.4 Class-Oriented Metrics—The MOOD Metrics Suite

Harrison, Counsell, and Nithi [Har98b] propose a set of metrics for object-oriented design that provide quantitative indicators for OO design characteristics. A sampling of MOOD metrics follows.

Method inheritance factor (MIF). The degree to which the class architecture of an OO system makes use of inheritance for both methods (operations) and attributes is defined as

$$\text{MIF} = \frac{\Sigma M_i\,(C_i)}{\Sigma M_a\,(C_i)}$$

where the summation occurs over $i = 1$ to TC. TC is defined as the total number of classes in the architecture, C_i is a class within the architecture, and

$$M_a(C_i) = M_d(C_i) + M_i(C_i)$$

where

$M_a(C_i)$ = number of methods that can be invoked in association with C_i
$M_d(C_i)$ = number of methods declared in the class C_i
$M_i(C_i)$ = number of methods inherited (and not overridden) in C_i

The value of MIF [the attribute inheritance factor (AIF) is defined in an analogous manner] provides an indication of the impact of inheritance on the OO software.

Coupling factor (CF). Earlier in this chapter I noted that coupling is an indication of the connections between elements of the OO design. The MOOD metrics suite defines coupling in the following way:

$$\text{CF} = \Sigma_i \Sigma_j \, is\_client\,(C_i, C_j) \frac{}{T_c^2 - T_c}$$

where the summations occur over $i = 1$ to T_c and $j = 1$ to T_c. The function

is_client = 1, if and only if a relationship exists between the client class C_c and the server class C_s, and $C_c \neq C_s$
= 0, otherwise

Although many factors affect software complexity, understandability, and maintainability, it is reasonable to conclude that as the value for CF increases, the complexity of the OO software will also increase and understandability, maintainability, and the potential for reuse may suffer as a result.

Harrison and her colleagues [Har98b] present a detailed analysis of MIF and CF along with other metrics and examine their validity for use in the assessment of design quality.

23.3.5 OO Metrics Proposed by Lorenz and Kidd

In their book on OO metrics, Lorenz and Kidd [Lor94] divide class-based metrics into four broad categories that each have a bearing on component-level design: size, inheritance, internals, and externals. Size-oriented metrics for an OO design class focus on counts of attributes and operations for an individual class and average values for the OO system as a whole. Inheritance-based metrics focus on the manner in which operations are reused through the class hierarchy. Metrics for class internals look at cohesion (Section 23.3.3) and code-oriented issues, and external metrics examine coupling and reuse. One example of the metrics proposed by Lorenz and Kidd is:

Class size (CS). The overall size of a class can be determined using the following measures:

- The total number of operations (both inherited and private instance operations) that are encapsulated within the class
- The number of attributes (both inherited and private instance attributes) that are encapsulated by the class

During review of the analysis model, CRC index cards will provide a reasonable indication of expected values for CS. If you encounter a class with a large number of responsibilities, consider partitioning it.

The WMC metric proposed by Chidamber and Kemerer (Section 23.3.3) is also a weighted measure of class size. As I noted earlier, large values for CS indicate that a class may have too much responsibility. This will reduce the reusability of the class and complicate implementation and testing. In general, inherited or public operations and attributes should be weighted more heavily in determining class size [Lor94]. Private operations and attributes enable specialization and are more localized in the design. Averages for the number of class attributes and operations may also be computed. The lower the average values for size, the more likely that classes within the system can be reused widely.

23.3.6 Component-Level Design Metrics

Component-level design metrics for conventional software components focus on internal characteristics of a software component and include measures of the

"three Cs"—module cohesion, coupling, and complexity. These measures can help you judge the quality of a component-level design.

Component-level design metrics may be applied once a procedural design has been developed and are "glass box" in the sense that they require knowledge of the inner workings of the module under consideration. 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 a design.

Cohesion metrics. Bieman and Ott [Bie94] define a collection of metrics that provide an indication of the cohesiveness (Chapter 8) 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). A detailed discussion of the Bieman and Ott metrics is best left to the authors [Bie94].

Coupling metrics. Module coupling provides an indication of the "connectedness" of a module to other modules, global data, and the outside environment. In Chapter 9, 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.

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:

$$m_c = \frac{k}{M}$$

where k is a proportionality constant and

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

Values for k, a , b, and c must be derived empirically.

As the value for m_c increases, the overall module coupling decreases. In order to have the coupling metric move upward as the degree of coupling increases, a revised coupling metric may be defined as

$$C = 1 - m_c$$

where the degree of coupling increases as the value of M increases.

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. A graph (Chapter 18) 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.

The most widely used (and debated) complexity metric for computer software is cyclomatic complexity, originally developed by Thomas McCabe [McC76] and discussed in detail in Chapter 18.

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

Zuse ([Zus90], [Zus97]) presents an encyclopedic discussion of no fewer than 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.

23.3.7 Operation-Oriented Metrics

Because the class is the dominant unit in OO systems, fewer metrics have been proposed for operations that reside within a class. Churcher and Shepperd [Chu95]

discuss this when they state: "Results of recent studies indicate that methods tend to be small, both in terms of number of statements and in logical complexity [Wil93], suggesting that connectivity structure of a system may be more important than the content of individual modules." However, some insights can be gained by examining average characteristics for methods (operations). Three simple metrics, proposed by Lorenz and Kidd [Lor94], are appropriate:

Average operation size (OS_{avg}). Size can be determined by counting the number of lines of code or the number of messages sent by the operation. As the number of messages sent by a single operation increases, it is likely that responsibilities have not been well allocated within a class.

Operation complexity (OC). The complexity of an operation can be computed using any of the complexity metrics proposed for conventional software [Zus90]. Because operations should be limited to a specific responsibility, the designer should strive to keep OC as low as possible.

Average number of parameters per operation (NP_{avg}). The larger the number of operation parameters, the more complex the collaboration between objects. In general, NP_{avg} should be kept as low as possible.

23.3.8 User Interface Design Metrics

Quote:

"You can learn at least one principle of user interface design by loading a dishwasher. If you crowd a lot in there, nothing gets very clean."

Author unknown

Although there is significant literature on the design of human/computer interfaces (Chapter 11), 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.

A study of Web page metrics [Ivo01] indicates that simple characteristics of the elements of the layout can also have a significant impact on the perceived quality of the GUI design. The number of words, links, graphics, colors, and fonts (among other characteristics) contained within a Web page affect the perceived complexity and quality of that page.

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 success 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."

23.4 Design Metrics for WebApps

A useful set of measures and metrics for WebApps provides quantitative answers to the following questions:

- Does the user interface promote usability?
- Are the aesthetics of the WebApp appropriate for the application domain and pleasing to the user?
- Is the content designed in a manner that imparts the most information with the least effort?
- Is navigation efficient and straightforward?
- Has the WebApp architecture been designed to accommodate the special goals and objectives of WebApp users, the structure of content and functionality, and the flow of navigation required to use the system effectively?
- Are components designed in a manner that reduces procedural complexity and enhances correctness, reliability, and performance?

Many of these metrics are applicable to all user interfaces and should be considered in conjunction with those presented in Section 23.3.8.

Today, each of these questions can be addressed only qualitatively because a validated suite of metrics that would provide quantitative answers does not yet exist.

In the paragraphs that follow, I present a representative sampling of WebApp design metrics that have been proposed in the literature. It is important to note that many of these metrics have not as yet been validated and should be used judiciously.

Interface metrics. For WebApps, the following interface measures can be considered:

Suggested Metric	*Description*
Layout appropriateness	See Section 23.3.8.
Layout complexity	Number of distinct regions[12] defined for an interface
Layout region complexity	Average number of distinct links per region
Recognition complexity	Average number of distinct items the user must look at before making a navigation or data input decision
Recognition time	Average time (in seconds) that it takes a user to select the appropriate action for a given task
Typing effort	Average number of key strokes required for a specific function
Mouse pick effort	Average number of mouse picks per function
Selection complexity	Average number of links that can be selected per page
Content acquisition time	Average number of words of text per Web page
Memory load	Average number of distinct data items that the user must remember to achieve a specific objective

12 A distinct region is an area within the layout display that accomplishes some specific set of related functions (e.g., a menu bar, a static graphical display, a content area, an animated display).

Aesthetic (graphic design) metrics. By its nature, aesthetic design relies on qualitative judgment and is not generally amenable to measurement and metrics. However, Ivory and her colleagues [Ivo01] propose a set of measures that may be useful in assessing the impact of aesthetic design:

Suggested Metric	*Description*
Word count	Total number of words that appear on a page
Body text percentage	Percentage of words that are body versus display text (i.e., headers)
Emphasized body text %	Portion of body text that is emphasized (e.g., bold, capitalized)
Text positioning count	Changes in text position from flush left
Text cluster count	Text areas highlighted with color, bordered regions, rules, or lists
Link count	Total links on a page
Page size	Total bytes for the page as well as elements, graphics, and style sheets
Graphic percentage	Percentage of page bytes that are for graphics
Graphics count	Total graphics on a page (not including graphics specified in scripts, applets, and objects)
Color count	Total colors employed
Font count	Total fonts employed (i.e., face + size + bold + italic)

Content metrics. Metrics in this category focus on content complexity and on clusters of content objects that are organized into pages [Men01].

Suggested Metric	*Description*
Page wait	Average time required for a page to download at different connection speeds
Page complexity	Average number of different types of media used on page, not including text
Graphic complexity	Average number of graphics media per page
Audio complexity	Average number of audio media per page
Video complexity	Average number of video media per page
Animation complexity	Average number of animations per page
Scanned image complexity	Average number of scanned images per page

Navigation metrics. Metrics in this category address the complexity of the navigational flow [Men01]. In general, they are applicable only for static Web applications, which don't include dynamically generated links and pages.

Suggested Metric	*Description*
Page-linking complexity	Number of links per page
Connectivity	Total number of internal links, not including dynamically generated links
Connectivity density	Connectivity divided by page count

Using a subset of the metrics suggested, it may be possible to derive empirical relations that allow a WebApp development team to assess technical quality and predict effort based on projected estimates of complexity. Further work remains to be accomplished in this area.

SOFTWARE TOOLS

Technical Metrics for WebApps

Objective: To assist Web engineers in developing meaningful WebApp metrics that provide insight into the overall quality of an application.

Mechanics: Tool mechanics vary.

Representative Tools:[13] *Netmechanic Tools,* developed by Netmechanic (www.netmechanic.com), is a collection of tools that help to improve website performance, focusing on implementation-specific issues. *NIST Web Metrics* Testbed, developed by The National Institute of Standards and Technology (**zing.ncsl.nist.gov/WebTools/**) encompasses the following collection of useful tools that are available for download:

Web Static Analyzer Tool (WebSAT)—checks Web page HTML against typical usability guidelines.

Web Category Analysis Tool (WebCAT)—lets the usability engineer construct and conduct a Web category analysis.

Web Variable Instrumenter Program (WebVIP)—instruments a website to capture a log of user interaction.

Framework for Logging Usability Data (FLUD)—implements a file formatter and parser for representation of user interaction logs.

VisVIP Tool—produces a 3D visualization of user navigation paths through a website.

TreeDec—adds navigation aids to the pages of a website.

23.5 METRICS FOR SOURCE CODE

Quote:

"The human brain follows a more rigid set of rules [for developing algorithms] than it has been aware of."

Maurice Halstead

Halstead's theory of "software science" [Hal77] proposed the first analytical "laws" for computer software.[14] Halstead assigned 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. The measures are:

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

13 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category.
14 It should be noted that Halstead's "laws" have generated substantial controversy, and many believe that the underlying theory has flaws. However, experimental verification for selected programming languages has been performed (e.g., [Fel89]).

Halstead shows that length N can be estimated

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

and program volume may be defined

$$V = N \log_2 (n_1 + n_2)$$

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

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

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

$$L = \frac{2}{n_1} \times \frac{n_2}{N_2}$$

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 book. For further information, see [Zus90], [Fen91], and [Zus97].

23.6 Metrics for Testing

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.

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.

Architectural design metrics provide information on the ease or difficulty associated with integration testing (Section 23.3) 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 18. In addition, cyclomatic complexity can be used to target modules as candidates for extensive unit testing. Modules with high cyclomatic complexity are more likely to be error prone than modules whose cyclomatic complexity is lower. For this reason, you should expend above average effort to uncover errors in such modules before they are integrated in a system.

23.6.1 Halstead Metrics Applied to Testing

Testing effort can be estimated using metrics derived from Halstead measures (Section 23.5). Using the definitions for program volume V and program level PL, Halstead effort e can be computed as

$$PL = \frac{1}{(n_1/2) \times (N_2/n_2)} \tag{23.2a}$$

$$e = \frac{V}{PL} \tag{23.2b}$$

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

$$\text{Percentage of testing effort } (k) = \frac{e(k)}{\Sigma e(i)} \tag{23.3}$$

where $e(k)$ is computed for module k using Equations (23.2) and the summation in the denominator of Equation (23.3) is the sum of Halstead effort across all modules of the system.

23.6.2 Metrics for Object-Oriented Testing

The OO design metrics noted in Section 23.3 provide an indication of design quality. They also provide a general indication of the amount of testing effort required to exercise an OO system. Binder [Bin94b] suggests a broad array of design metrics that have a direct influence on the "testability" of an OO system. The metrics consider aspects of encapsulation and inheritance.

Lack of cohesion in methods (LCOM).[15] The higher the value of LCOM, the more states must be tested to ensure that methods do not generate side effects.

Percent public and protected (PAP). Public attributes are inherited from other classes and therefore are visible to those classes. Protected attributes are accessible to methods in subclasses. This metric indicates the percentage of class attributes that are public or protected. High values for PAP increase the likelihood of side effects among classes because public and protected attributes lead to high potential for coupling.[16] Tests must be designed to ensure that such side effects are uncovered.

OO testing can be quite complex. Metrics can assist you in targeting testing resources at threads, scenarios, and packages of classes that are "suspect" based on measured characteristics. Use them.

Public access to data members (PAD). This metric indicates the number of classes (or methods) that can access another class's attributes, a violation of encapsulation. High values for PAD lead to the potential for side effects among classes. Tests must be designed to ensure that such side effects are uncovered.

Number of root classes (NOR). This metric is a count of the distinct class hierarchies that are described in the design model. Test suites for each root class and the corresponding class hierarchy must be developed. As NOR increases, testing effort also increases.

Fan-in (FIN). When used in the OO context, fan-in in the inheritance hierarchy is an indication of multiple inheritance. FIN > 1 indicates that a class inherits its attributes and operations from more than one root class. FIN > 1 should be avoided when possible.

Number of children (NOC) and depth of the inheritance tree (DIT).[17] As I mentioned in Chapter 19, superclass methods will have to be retested for each subclass.

15 See Section 23.3.3 for a description of LCOM.
16 Some people promote designs with none of the attributes being public or private, that is, PAP = 0. This implies that all attributes must be accessed in other classes via methods.
17 See Section 23.3.3 for a description of NOC and DIT.

23.7 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 [IEE93] 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 = number of modules in the current release
F_c = number of modules in the current release that have been changed
F_a = number of modules in the current release that have been added
F_d = number of modules from the preceding release that were deleted in the current release

The software maturity index is computed in the following manner:

$$\text{SMI} = \frac{M_T - (F_a + F_c + F_d)}{M_T}$$

As SMI approaches 1.0, the product begins to stabilize. SMI may also be used as a 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.

SOFTWARE TOOLS

Product Metrics

Objective: To assist software engineers in developing meaningful metrics that assess the work products produced during analysis and design modeling, source code generation, and testing.

Mechanics: Tools in this category span a broad array of metrics and are implemented either as a stand-alone application or (more commonly) as functionality that exists within tools for analysis and design, coding, or testing. In most cases, the metrics tool analyzes a representation of the software (e.g., a UML model or source code) and develops one or more metrics as a result.

Representative Tools:[18] *Krakatau Metrics*, developed by Power Software (**www.powersoftware.com/products**), computes complexity, Halstead, and related metrics for C/C++ and Java.

Metrics4C—developed by +1 Software Engineering (**www.plus-one.com/Metrics4C_fact_sheet.html**), computes a variety of architectural, design, and code-oriented metrics as well as project-oriented metrics.

Rational Rose, distributed by IBM (**www-304.ibm.com/jct03001c/software/awdtools/developer/rose/**), is a comprehensive tool set for UML modeling that incorporates a number of metrics analysis features.

RSM, developed by M-Squared Technologies (**msquaredtechnologies.com/m2rsm/index.html**), computes a wide variety of code-oriented metrics for C, C++, and Java.

Understand, developed by Scientific Toolworks, Inc. (**www.scitools.com**), calculates code-oriented metrics for a variety of programming languages.

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

23.8 Summary

Software metrics provide a quantitative way to assess the quality of internal product attributes, thereby enabling you to assess quality before the product is built. Metrics provide the insight necessary to create effective requirements 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 you with effective feedback.

Metrics for the requirements model focus on function, data, and behavior—the three components of the 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. User interface design metrics provide an indication of the ease with which a GUI can be used. WebApp metrics consider aspects of the user interface as well as WebApp aesthetics, content, and navigation.

Metrics for OO systems focus on measurement that can be applied to the class and the design characteristics—localization, encapsulation, information hiding, inheritance, and object abstraction techniques—that make the class unique. The CK metrics suite defines six class-oriented software metrics that focus on the class and the class hierarchy. The metrics suite also develops metrics to assess the collaborations between classes and the cohesion of methods that reside within a class. At a class-oriented level, the CK metrics suite can be augmented with metrics proposed by Lorenz and Kidd and the MOOD metrics suite.

Halstead 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 product metrics have been proposed for direct use in software testing and maintenance. However, many other product metrics can be used to guide the testing process and as a mechanism for assessing the maintainability of a computer program. A wide variety of OO metrics have been proposed to assess the testability of an OO system.

Problems and Points to Ponder

23.1. Measurement theory is an advanced topic that has a strong bearing on software metrics. Using [Zus97], [Fen91], [Zus90], or Web-based sources, 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.

23.2. Why is it that a single, all-encompassing metric cannot be developed for program complexity or program quality? Try to come up with a measure or metric from everyday life that violates the attributes of effective software metrics defined in Section 23.1.5.

23.3. A system has 12 external inputs, 24 external outputs, fields 30 different external queries, manages 4 internal logical files, and interfaces with 6 different legacy systems (6 EIFs). All of

these data are of average complexity and the overall system is relatively simple. Compute FP for the system.

23.4. Software for System X has 24 individual functional requirements and 14 nonfunctional requirements. What is the specificity of the requirements? The completeness?

23.5. 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 database items and 90 different database segments. Finally, 600 modules have single entry and exit points. Compute the DSQI for this system.

23.6. A class **X** has 12 operations. Cyclomatic complexity has been computed for all operations in the OO system, and the average value of module complexity is 4. For class **X**, the complexity for operations 1 to 12 is 5, 4, 3, 3, 6, 8, 2, 2, 5, 5, 4, 4, respectively. Compute the weighted methods per class.

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

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

23.9. 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 Readings and Information Sources

There is 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 product metrics. Lanza and her colleagues (*Object-Oriented Metrics in Practice,* Springer, 2006) discuss OO metrics and their use for assessing the quality of a design. Genero (*Metrics for Software Conceptual Models,* Imperial College Press, 2005) and Ejiogu (*Software Metrics,* BookSurge Publishing, 2005) present a wide variety of technical metrics for use cases, UML models, and other modeling representations. Hutcheson (*Software Testing Fundamentals: Methods and Metrics,* Wiley, 2003) presents a set of metrics for testing. Kan (*Metrics and Models in Software Quality Engineering,* Addison-Wesley, 2d ed., 2002), Fenton and Pfleeger (*Software Metrics: A Rigorous and Practical Approach,* Brooks-Cole Publishing, 1998), and Zuse [Zus97] have written thorough treatments of product metrics.

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 product 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.

Methods for establishing a metrics program and the underlying principles for software measurement are considered by Ebert and his colleagues (*Best Practices in Software Measurement,* Springer, 2004). Shepperd (*Foundations of Software Measurement,* Prentice-Hall, 1996) also addresses measurement theory in some detail. Current research is presented in the *Proceedings of the Symposium on Software Metrics* (IEEE, published annually).

A comprehensive summary of dozens of useful software metrics is presented in [IEE93]. 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.

Whitmire [Whi97] presents a comprehensive and mathematically sophisticated treatment of OO metrics. Lorenz and Kidd [Lor94] and Hendersen-Sellers (*Object-Oriented Metrics: Measures of Complexity,* Prentice-Hall, 1996) provide treatments that are dedicated to OO metrics.

A wide variety of information sources on software metrics is available on the Internet. An up-to-date list of World Wide Web references that are relevant to software metrics can be found at the SEPA website: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.