15.2 A Framework for Product 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 product metrics for software.

15.2.1 Measures, Metrics, and Indicators

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* [IEE93] 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).

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.

> "A science is as mature as its measurement tools."
>
> **Louis Pasteur**

15.2.2 The Challenge of Product 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 product metrics has been compiled by Horst Zuse at **irb.cs.tuberlin.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 9 through 12). 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 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.

Each of the "challenges" noted here is a cause for caution, but it is no reason to dismiss product metrics.[3] Measurement is essential if quality is to be achieved.

15.2.3 Measurement Principles

Before we introduce a series of product metrics that (1) assist in the evaluation of 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 appropriate for the representation of the software that is being considered.

3 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.

- *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 in an effort to gain 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 [LET03] are representative of many that can be proposed for metrics characterization and validation:

- *A metric should have desirable mathematical properties.* That is, the metric's value should be in a meaningful range (e.g., zero to one, where zero truly means absence, one indicates the maximum value, and 0.5 represents the "half-way 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.

ADVICE

reality, many product trics in use today do conform to these ciples as well as should. But that n't mean that they no value—just reful when you iem, under- ng that they are ed to provide , not hard scien- rification.

Although formulation, characterization, and validation are critical, collection and analysis are the activities that drive the measurement process. Roche [ROC94] suggests the following guidelines 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 is correlated with the number of defects reported in production use); and (3) interpretative guidelines and recommendations should be established for each metric.

15.2.4 Goal-Oriented Software Measurement

Ref

iscussion of be found at hedacs.com ractices/ s/gqma.

The *Goal/Question/Metric* (GQM) paradigm was 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[4]} **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 (Chapter 9) of an architectural component might be useful in answering Q_1. Cyclomatric complexity and metrics discussed in Section 15.4.1 or 15.4.2 might provide insight for Q_2.

In actuality, there may be a number of measurement goals with related questions and metrics. In every case, the metrics that are chosen (or derived) should conform to the measurement principles discussed in Section 15.2.3 and the measurement attributes discussed in Section 15.2.5. For further information of GQM, the interested reader should see [SHE98] or [SOL99].

15.2.5 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:

- *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.
- *Consistent and objective.* The metric should always yield results that are unambiguous.
- *Consistent in the use of units and dimensions.* The mathematical computation of the metric should use measures that do not lead to bizarre combinations of units.
- *Programming language independent.* Metrics should be based on the analysis model, the design model, or the structure of the program itself.
- *An effective mechanism for high-quality feedback.* That is, the metric should lead to a higher-quality end product.

How should we assess quality of a posed tware metric?

ADVICE

ience indicates product metric used only if it is e and easy to te. If dozens of s" have to be and complex ations are , it is unlikely metric will be dopted.

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 15.3.1)—a measure of the "functionality" delivery by the software. It can be argued[5] 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.

15.2.6 The Product Metrics Landscape

Although a wide variety of metrics taxonomies have been proposed, the following outline addresses the most important metrics areas:

Metrics for the analysis model. These metrics address various aspects of the analysis model and include:

Functionality delivered—provides an indirect measure of the functionality that is packaged within the software.

System size—measures of the overall size of the system defined in terms of information available as part of the analysis model.

Specification quality—provides an indication of the specificity and completeness of a requirements specification.

Metrics for the design model. These metrics quantify design attributes in a manner that allows a software engineer to assess design quality. Metrics include:

Architectural metrics—provide an indication of the quality of the architectural design.

Component-level metrics—measure the complexity of software components and other characteristics that have a bearing on quality.

Interface design metrics—focus primarily on usability.

Specialized OO design metrics—measure characteristics of classes and their communication and collaboration characteristics.

Metrics for source code. These metrics measure the source code and can be used to assess its complexity, maintainability, and testability, among other characteristics:

Halstead metrics—controversial but nonetheless fascinating, these metrics provide unique measures of a computer program.

Complexity metrics—measure the logical complexity of source code (can also be considered to be component-level design metrics).

Length metrics—provide an indication of the size of the software.

Metrics for testing. These metrics assist in the design of effective test cases and evaluate the efficacy of testing:

Statement and branch coverage metrics—lead to the design of test cases that provide program coverage.

Defect-related metrics—focus on bugs found, rather than on the tests themselves.

Testing effectiveness—provide a real-time indication of the effectiveness of tests that have been conducted.

In-process metrics—process related metrics that can be determined as testing is conducted.

In many cases, metrics for one model may be used in later software engineering activities. For example, design metrics may be used to estimate the effort required to generate source code. In addition, design metrics may be used in test planning and test case design.

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 analysis 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.

15.3.1 Function-Based Metrics

The *function point metric* (FP), first proposed by Albrecht [ALB79], can be used effectively as a means for measuring the functionality delivered by a system.[6] Using historical data, the FP 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 assessments of software complexity. Information domain values are defined in the following manner:

6 Since Albrecht's original

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 within the application and provides information to the user. In this context external output refers to reports, screens, error messages, and so on. Individual data items within a report are not counted separately.

Number of external inquiries (EQs). An *external inquiry* is defined as an on-line input that results in the generation of some immediate software response in the form of an on-line output (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 data that may be of use to the application.

Once these data have been collected, the table in Figure 15.2 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:

$$FP = \text{count total} \times [0.65 + 0.01 \times \Sigma (F_i)] \quad (15\text{-}1)$$

where count total is the sum of all FP entries obtained from Figure 15.2.

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?

FIGURE 15.2 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.

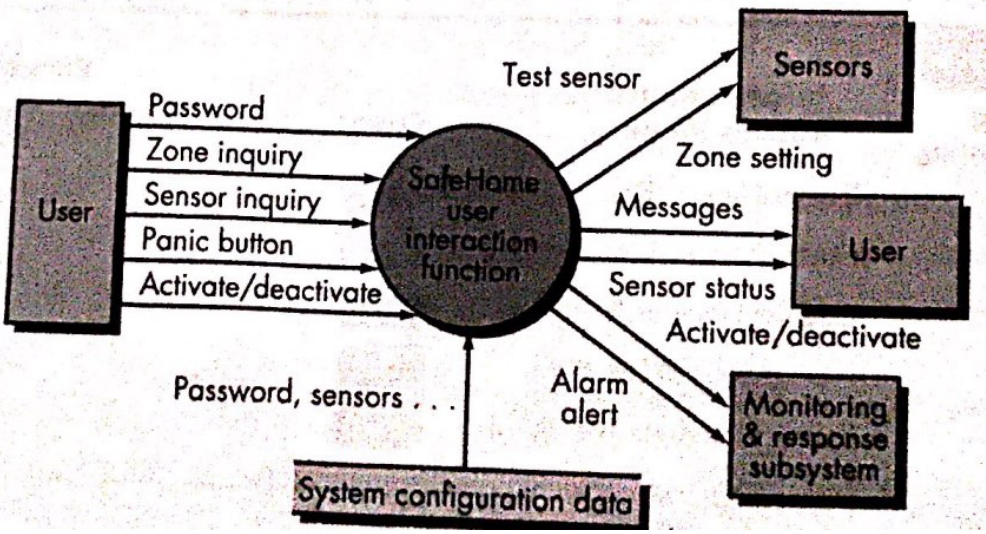
2. Are specialized data communications required to transfer information to or from the application?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require on-line data entry?
7. Does the on-line data entry require the input transaction to be built over multiple screens or operations?
8. Are the ILFs updated on-line?
9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and for 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 (15-1) and the weighting factors that are applied to information domain counts are determined empirically.

WebRef
An on-line FP calculator can be found at www.s.unimagdeburg.de/sw-eng/java/fp/.

To illustrate the use of the FP metric in this context, we consider a simple analysis model representation, illustrated in Figure 15.3. Referring to the figure, a data flow diagram (Chapter 8) for a function within the *SafeHome* software 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.

FIGURE 15.3
Data flow model for *SafeHome* software



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 inquires—**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 15.4.

The count total shown in Figure 15.4 must be adjusted using Equation (15-1):

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

where count total is the sum of all FP entries obtained from Figure 15.4 and F_i (i = 1 to 14) are value adjustment factors. 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 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.

Uemura and his colleagues [UEM99] suggest that function points can also be computed from UML class and sequence diagrams (Chapters 8 and 10). The interested reader should see [UEM99] for details.

FIGURE 15.4 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