17 March 2011

A goal-based framework for software measurement

Chapter 1 describes measurement's essential role in good software engineering practice, and Chapter 2 shows how to apply a general theory of measurement to software. In this chapter, we present a conceptual framework for diverse software measurement activities that you can apply to an organization's software development practices. These practices may include development and maintenance activities, plus experiments and case studies that evaluate new techniques and tools.

The framework presented here is based on two principle activites: classifying the entities to be examined and using measurement goals to identify relevant metrics., We show how such a goal-based framework supports evaluations of software products, processes, and software development oranizations. We also look at measurement validation: both the process of insuring that we are measuring what we say we are, so that we satisfy the representation condition introduced in Chapter 2, as well as demonstrating the utility of a measure.

3.1. Classifying software measures

As we have seen in Chapter 2, the first obligation of any software measurement activity is identifying the entities and attributes that we want to measure. Software entities can be classified as follows:

- **Processes**: software-related activities.
- **Products**: artifacts, deliverables or documents that result from a process activity.
- **Resources**: entities required by a process activity.

A process is usually associated with some time-scale. Process activities have a duration – they occur over time, and they may be ordered or related in some way that depends on time, so that one activity must be completed before another can begin. The timing can be explicit, as when design must be complete by October 31, or implicit, as when a flow diagram shows that design must be completed before coding can begin.

Resources and products are associated with a process. Each process activity uses resources and products, and produces products. Thus, the product of one activity may feed another activity. For example, a design document can be the product of the design activity which is then used as an input to the coding activity.

Many of the examples that we use in this book relate to the development process or maintenance process. But the concepts that we introduce apply to any process: the reuse process, the configuration management process, the testing process, and so on. In other words, measurement activities may focus on any process and need not be concerned with the

comprehensive development process. As we show later in this chapter, our choice of measurements depends on our measurement goals.

Within each class of entity, we distinguish between internal and external attributes of a product, process, or resource:

- **Internal attributes**: attributes that can be measured purely in terms of the product, process or resource itself. An internal attribute can be measured by examining the product, process or resource on its own, without considering its behavior.
- **External attributes**: attributes that can be measured only with respect to how the product, process or resource relates to its environment. Here, the behavior of the process, product or resource is important, rather than the entity itself.

To understand the difference between internal and external attributes, consider a set of software modules. Without actually executing the code, we can determine several important internal attributes: its size (perhaps in terms of lines of code or number of operands), its complexity (perhaps in terms of the number of decision points in the code), and the dependencies among modules. We may even find faults in the code as we read it: misplaced commas, improper use of a command, or failure to consider a particular case. However, there are other attributes of the code that can be measured only when the code is executed: the number of failures experienced by the user, the difficulty that the user has in navigating among the screens provided, or the length of time it takes to search the database and retrieve requested information, for instance. It is easy to see that these attributes depend on the behavior of the code, making them external attributes rather than internal. Table 3.1 provides additional examples of types of entities and attributes.

Table 3.1: Components of software measurement

ENTITIES	ATTRIBUTES	
Products	*Internal*	*External*
Requirements Use case models & scenarios	size, reuse, modularity, redundancy, functionality, syntactic correctness, ...	comprehensibility, maintainability, ...
Designs Design models	size, reuse, modularity, coupling, cohesiveness, functionality, ...	quality, complexity, maintainability, ...
Code	size, reuse, modularity, coupling, functionality, algorithmic complexity, control-flow structuredness, ...	reliability, usability, maintainability, ...
Test requirements	size, …	effectiveness, …
Test data	size, coverage,	fault finding ability
Test harness	languages supported, features	ease of use
...	...	...
Processes		
Constructing requirements	time, effort, number of requirements changes, ...	quality, cost, stability, ...
Detailed design	time, effort, number of specification faults found, ...	cost, cost-effectiveness, ...
Testing	time, effort, number of coding faults found, ...	cost, cost-effectiveness, stability, ...
...	...	...
Resources		
Personnel	age, price, ...	productivity, experience, intelligence, ...
Teams	size, communication level, structuredness, ...	productivity, quality, ...
Software	price, size, ...	usability, reliability, ...
Hardware	price, speed, memory size, ...	reliability, ...
Offices	size, temperature, light, ...	comfort, quality, ...
...	...	...

Managers often want to be able to measure and predict external attributes. For example, the cost-effectiveness of an activity (such as design inspections) or the productivity of the staff can be very useful in ensuring that the quality stays high while the price stays low. Users are also interested in external attributes, since the behavior of the system affects them directly; the system's reliability, usability and portability affect maintenance and purchase decisions. However, external attributes are usually more difficult to measure than internal ones, and they are measured quite late in the development process. For example, reliability can be measured only after development is complete and the system is ready for use.

Moreover, it can be difficult to define the attributes in measurable ways that satisfy all stakeholders. For example, we all want to build and purchase systems of high quality, but we do not always agree on what we mean by quality, and it is often difficult to measure quality in a comprehensive way. Thus, we tend to define these high-level attributes in terms of other, more-concrete attributes that are well-defined and measurable. The McCall model introduced in Chapter 1 is a good example of this phenomenon, where software quality is defined as a composite of a large number of narrower, more easily measurable terms.

In many cases, developers and users focus their efforts on only one facet of a broad attribute. For example, some measure quality, an external product attribute, as the number of faults found during formal testing, an internal process attribute. Using internal attributes to make judgments about external attributes, can lead to invalid conclusions. But there is a clear need to be able to use internal attribute measurements decision-making about external attributes. One of the goals of software measurement research is to identify the relationships between internal and external attributes, as well as to find new and useful methods for directly measuring the attributes of interest.

We also need to be able to track changes in product, process, and resource attributes as systems evolve over time. For example, as a system evolves from version 2 to version 3, the design, code, regression tests, test requirements, processes, and personnel will change. Changes in key software measures can potentially identify problems such as design decay, improvements or reduction in reliability, personnel changes, etc. Measurements can also be used to determine the success or failure of the response to identified problems.

3.1.1 Processes

We often have questions about our software development activities and processes that measurement can help us to answer. We want to know how long it takes for a process to complete, how much it will cost, whether it is effective or efficient, and how it compares with other processes that we could have selected. However, only a limited number of internal process attributes can be measured directly. These measures include:

- the duration of the process or one of its activities
- the effort associated with the process or one of its activities
- the number of incidents of a specified type arising during the process or one of its activities

For example, we may be reviewing our requirements to ensure their quality before turning them over to the designers. To measure the effectiveness of the review process, we can measure the number of requirements errors found during the review as well as the number of errors found during later activities. Likewise, to determine how well we are doing integration testing, we can measure the number of faults found during integration testing, as well as those found later. And the number of personnel working on the project during a specified period can give us insight into the resources needed for the development process.

Many of these measures can be used in combination with other measures to gain a better understanding of what is happening on a project:

Example 3.1: During formal testing, we can use the indirect measure

$$\frac{cost}{number\ of\ errors}$$

as a measure of the average cost of each error found during the process.

Example 3.2: AT&T developers wanted to know the effectiveness of their software inspections. In particular, managers needed to evaluate the cost of the inspections against the benefits received. To do this, they measured the average amount of effort expended per thousand lines of code reviewed. As we will see later in this chapter, this information, combined with measures of the number of faults discovered during the inspections, allowed the managers to perform a cost-benefit analysis.

In some cases, we may want to measure properties of a process that consists of a number of distinct sub-processes:

Example 3.3: The testing process may be composed of unit testing, integration testing, system testing, and acceptance testing. Each component process can be measured to determine how effectively it contributes to overall testing. We can track the number of errors identified in each subprocess, along with the duration and cost of identifying each error, to see if each subprocess is cost-effective.

Cost is not the only process attribute that we can examine. Controllability, observability and stability are also important in managing a large project. These attributes are clearly external ones. For example, stability of the design process can depend on the particular period of time, as well as on which designers are involved. Attributes such as these may not yet be sufficiently well-understood to enable numeric measurements according to the principles described in Chapter 2, so they are often indicated by subjective ratings on an ordinal scale. However, the subjective rankings based on informal observations can form the basis for the empirical relations required for subsequent objective measurement.

We often use objective measures of internal attributes as surrogate measures of external attributes. For example, surrogate measures of the effectiveness of code maintenance can be defined as a composite measure in terms of the number of faults discovered and the number of faults corrected. In the AT&T study of Example 3.2, inspection effectiveness was measured as average faults detected per thousand lines of code inspected.

In each case, we examine the process of interest and decide what kind of information would help us to understand, control or improve the process. We will investigate process attributes in Chapter 13.

3.1.2 Products

Products are not restricted to the items that management is committed to deliver as the final software product. Any artifact or document produced during the software life-cycle can be measured and assessed. For example, developers often model the requirements and design using various diagrams defined in the Unified Modeling Language (UML), and they build prototypes. The purpose of models and prototypes is to help developers to understand the requirements or evaluate possible designs; these models and prototypes may be measured in

some way. Likewise, test harnesses that are constructed to assist in system testing may be measured; system size measurements should include this software if they are to be used to determine team productivity. And documents, such as the user guide or the customer specification document, can be measured for size, quality, and more.

3.1.2.1 External product attributes

There are many examples of external product attributes. Since an external product attribute depends on both product behavior and environment, each attribute measure should take these constraints into account. For example, if we are interested in measuring the reliability of code, we must consider the machine and system configuraton on which the program is run as well as the mode of operational usage. That is, someone who uses a word processing package only to type letters may find its reliability to be different from someone who uses the same package to merge tables and link to spreadsheets. Similarly, the understandability of a document depends on the experience and credentials of the person reading it; a nuclear engineer reading the specification for power plant software is likely to rate its understandability higher than a mathematician reading the same document. Or the maintainability of a system may depend on the skills of the maintainers and the tools available to them.

Usability, integrity, efficiency, testability, reusability, portability and interoperability are other external attributes that we can measure. These attributes describe not only the code but also the other documents that support the development effort. Indeed, the maintainability, reusability and even testability of specifications and designs are as important as the code itself.

3.1.2.2 Internal product attributes

Internal product attributes are sometimes easy to measure. We can determine the size of a product by measuring the number of pages it fills or the number of words it contains, for example. Since the products are concrete, we have a better understanding of attributes like size, effort and cost. Other internal product attributes are more difficult to measure, as opinions differ as to what they mean and how to measure them. For example, there are many aspects of code complexity, and no consensus about what best measures it. We will explore this issue in Chapter 9.

Because products are relatively easy to examine in an automated fashion, there is a set of commonly-measured internal attributes. For instance, you can assess specifications in terms of their length, functionality, modularity, reuse, redundancy and syntactic correctness. Formal designs and code can be measured in the same way; we can also measure attributes such as structuredness (of control and data flow, for example) as well as module coupling and cohesiveness.

Users sometimes dismiss many of these internal attributes as unimportant, since a user is interested primarily in the ultimate functionality, quality and utility of the software. However, the internal attributes can be very helpful in suggesting what we are likely to find as the external attributes.

Example 3.4: Consider a software purchase to be similar to buying an automobile. If we want to evaluate a used car, we can perform dynamic testing by actually driving the car in various conditions in order to assess external attributes such as performance and reliability. But usually we cannot make a complete dynamic assessment before we make a purchase decision, because not every type of driving condition is available (such as snowy or slick roads). Instead, we supplement our limited view of the car's performance with measures of static properties (that is, internal attributes) such as water level, oil type and level, brake fluid type and level, tire tread and wear pattern, brake wear, shock absorber response, fan belt flexibility and wear, and so on. These internal attributes provide insight into the likely external attributes; for example, uneven tire wear may indicate that the tires have been under-inflated, and the owner may have abused the car by not performing necessary maintenance. Indeed, when a car is serviced, the mechanic measures only internal attributes and makes adjustments accordingly; the car is rarely driven for any length of time to verify the conditions indicated by the internal attributes. In the same way, measures of internal software product attributes can tell us what the software's performance and reliability may be. Changes to the inspection process, for instance, may be based on measures of faults found, even though the ultimate goal of reliability is based on failures, not faults.

Just as processes may be comprised of subprocesses, there are products that are collections of subproducts.

Example 3.5: A software system design may consist of a number of UML design-level class diagrams. Average class diagram size, an attribute of the system design, can be derived by calculating the size of each diagram. We can measure size by number of classes in each diagram, and then calculate the comprehensive measure according to Table 3.2 below:

Table 3.2: Definition of example design measurements

Entity	*Entity Type*	*Attribute*	*Proposed Measure*	*Type*
Design-level clss diagram D_i	Product	Size	$Number\_of\_classes_i$	Direct
System design $\{D_1, \ldots, D_n\}$	Product	Average class diagram size	$1/n \sum Number\_of\_classes_i$	Indirect

3.1.2.3 The importance of internal attributes

Many software engineering methods proposed and developed in the last 40 years provide rules, tools, and heuristics for producing software products. Almost invariably, these methods give

structure to the products and the common wisdom is that this structure makes software products easier to understand, analyze, test, and modify. The structure involves two aspects of development:

- the development process, as certain products need to be produced at certain stages, and
- the products themselves, as the products must conform to certain structural principles.

In particular, product structure is usually characterized by levels of internal attributes such as modularity, coupling or cohesion.

> **Example 3.6:** One of the most widely respected books in software engineering is Brooks' *Mythical Man-Month*. [Brooks 1975, Brooks 1995] There, Brooks describes the virtues of top-down design:
>
> "A good top-down design avoids bugs in several ways. First, the clarity of structure and representation makes the precise statement of requirements and functions of the modules easier. Second, the partitioning and independence of modules avoids system bugs. Third, the suppression of detail makes flaws in the structure more apparent. Fourth, the design can be tested at each of its refinement steps, so testing can start earlier and focus on the proper level of detail at each step."
>
> Similarly, the notion of high module cohesion and low module coupling is the rationale for most design methods from structured design [Yourdon and Constantine 1979] to object-oriented design [Larman 2004]. Designs that possess these attributes are assumed to lead to more reliable and maintainable code.

Brooks, Yourdon and Constantine, and Larman assume what most other software engineers assume: that good internal structure leads to good external quality. Although intuitively appealing, the connection between internal attribute values and the resulting external attribute values has rarely been established, in part because (as we shall see in Chapter 4) it is difficult to perform controlled experiments and confirm relationships between attributes. At the same time,validating measures of attributes such as cohesion and coupling is difficult but necessary. Valid software measurement can help us to understand and confirm relationships empirically; a key to the success of this analysis is the ability to provide accurate and meaningful measures of internal product attributes.

3.1.2.4 Internal attributes and quality control and assurance

A major reason that developers want to use internal attributes to predict external ones is the need to monitor and control the products during development. For example, we want to be able to identify modules at the design stage whose profile, in terms of measures of internal attributes, shows that they are likely to be error-prone or difficult to maintain or test later on. To do this, we need to know the relationship between internal design attributes and failures. Figure 3.1 is an irreverent view of this process.

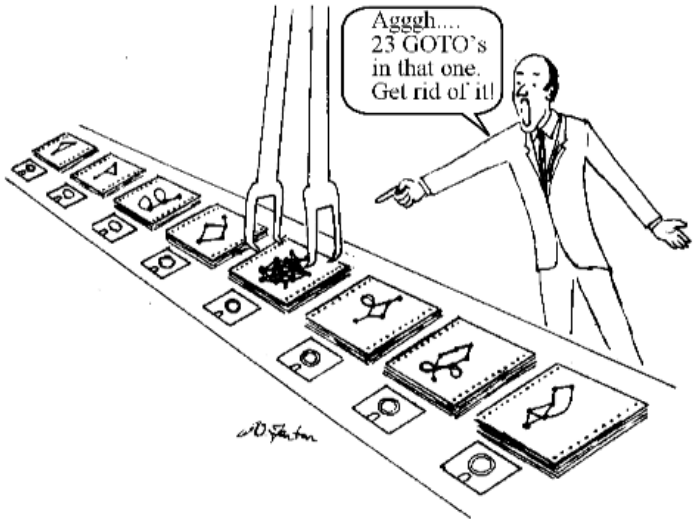

Figure 3.1: Using internal measures for quality control and assurance

3.1.2.5 Validating composite measures

"Quality" is frequently used by software engineers to describe an internal attribute of design or code. However, "quality" is multi-dimensional; it does not reflect a single aspect of a particular product.

> **Example 3.7:** Consider the "gross national product" (GNP), a measure of all the goods and services produced by a country in a given time period. Economists look at the trend in GNP over time, hoping that it will rise — that the country is becoming more productive. The GNP is a weighted combination of the values of key goods and services produced; the weights reflect the priorities and opinions of the economists defining the measure. The value of individual goods and services can be measured directly, but the GNP itself is an indirect measure.

In the same way that economists want to control the economy and make a country more productive, we want to measure and control the quality of our products, and we usually do so by measuring and controlling a number of internal (structural) product attributes. Without articulating the specific attributes that contribute to the general notion of quality, many people assume that the various cognitive and structural notions of complexity, maintainability and usability can be captured by a single number, much like GNP. This number should therefore be a powerful indicator of all the attributes which one normally associates with high-quality systems, such as high reliability and high maintainability.

However, this approach ignores the question of whether the component internal attributes are a complete and accurate depiction of the comprehensive one, and whether the weighting is

appropriate. Just as economists question whether GNP is valid (does it capture values like happiness, beauty, and environmental quality?), we must question the validity of measures that paint what only a partial picture of the attribute of interest. In later chapters, we shall give scrutiny to conventional approaches to measuring size, quality, complexity and more.

3.1.3 Resources

The resources that we are likely to measure include any input for software production. Thus, personnel (individual or teams), materials (including office supplies), tools (both software and hardware) and methods are candidates for measurement. We measure resources to determine their magnitude (how many staff are working on this project?), their cost (how much are we paying for testing tools?), and their quality (how experienced are our designers?) These measures help us to understand and control the process by telling us how the process is using and changing inputs to outputs. For example, if we are producing poor-quality software, resource measurements may show us that the software quality is the result of using too few people or people with the wrong skills.

Cost is often measured across all types of resources, so that managers can see how the cost of the inputs affects the cost of the outputs. For instance, the division chief may want to know if a large investment in software modeling or testing tools is yielding benefits in terms of more productive staff or better quality products. Cost is often defined in terms of its components, so that managers can see which aspects of cost are having the biggest effect.

Productivity is always important, and managers are keen not only to measure it but also to understand how to improve it. Although a measure of staff, productivity is an external resource attribute, since it depends on the underlying development process. That is, a productive worker using one process may become less productive if the process changes. Productivity is usually measured as some form of the following equation:

$$\frac{amount\ of\ output}{effort\ input}$$

Notice that this resource measure combines a process measure (input) with a product measure (output). The general notion is an economic one, where businesses or markets are judged by comparing what goes in with what comes out. For software development, the measure of output is usually computed as the amount of code or functionality produced as the final product, while the input measure is the number of person-months used to specify, design, code and test the software. However, the economic analogy is incomplete for software, since the amount of software output is not related to the input in the same way as that in manufacturing. That is, most manufacturing processes involve replication, so that one car is much like another coming off the assembly line. But software development is a creation or design process, not a replication process, and the relationship between inputs and outputs is defined differently. We will explore these ideas in more depth in Chapter 13.

There are many staff attributes that we can measure, whose values may have an influence on the process or product. For example, the education, experience, age or intelligence of a developer may affect the quality of the design or code. Similarly, the size, structure and communication patterns of the development team are important.

We can also classify and analyze tools and methods. Languages are strongly-typed or not, object-oriented or not, and so on. Techniques can be rated as manual or automated, and tools can require special training or experience. These attributes of resources help us to understand how to use tools and methods in more effective ways.

3.1.4 Change and Evolution

Software that is used is continually revised. New features are added and errors are fixed. To understand and manage evolving software, we need to measure attributes of products, processes, and resources at various points in time. Processes change over time. For example, improved inspection processes may be added to improve reliability. Clearly, software products change as new features are added—modules are added and expanded. Resources also change as developers move from project to project and new tools are adopted. Measurement can help us to understand and manage evolving systems by allowing us to track the changes in attributes over time. We can track the growth of modules, as well as changes in our processes and resources. We can also track changes in external attributes to identify, for example, improvements (hopefully) in product reliability and availability. Thus, we are interested in measurement over time, and the changes and trends of measured attributes. Measuring trends is how we can demonstrate product and process improvement.

No developer has the time to measure, analyze, and track everything. It is important to focus measurement activities on those areas needing the most visibility, understanding and improvement. In the next section, we present a technique for determining which attributes to measure first.

3.2. Determining what to measure

Measurement is useful only if it helps you to understand an underlying process or one of its resultant products. Determining the appropriate attributes to measure depends on your objectives—you select specific measurements based on what information you need to meet your goals. The Goal-Question-Metric approach (GQM) for metrics selection, first suggested by Basili and his colleagues, is an effective approach. ([Basili and Weiss 1984] , [Basili and Rombach 1988], [Basili et al. 2009]) To use GQM, you first identify the overall goals of your organization. Depending on the situation, the organization can be an entire corporation, an individual department or lab, or a single project or group. Then, you generate questions whose answers you must know in order to determine if your goals are being met. Finally, you analyze each question to identify measurements you need in order to answer each question.