CHAPTER 1

Measurement

What Is It and Why Do It?

Software measurement is an essential component of good software engineering. Many of the best software developers measure characteristics of their software to get some sense of whether the requirements are consistent and complete, whether the design is of high quality, and whether the code is ready to be released. Effective project managers measure attributes of processes and products to be able to tell when software will be ready for delivery and whether a budget will be exceeded. Organizations use process evaluation measurements to select software suppliers. Informed customers measure the aspects of the final product to determine if it meets the requirements and is of sufficient quality. Also, maintainers must be able to assess the current product to see what should be upgraded and improved.

This book addresses these concerns and more. The first seven chapters examine and explain the fundamentals of measurement and experimentation, providing you with a basic understanding of why we measure and how that measurement supports investigation of the use and effectiveness of software engineering tools and techniques. Chapters 8 through 11 explore software engineering measurement in further detail, with information about specific metrics and their uses. Collectively, the chapters offer broad coverage of software engineering measurement with enough depth so that you can apply appropriate metrics to your software processes, products, and resources. Even if you are a student, not yet experienced in working on projects with groups of people to solve interesting business or research

problems, this book explains how measurement can become a natural and useful part of your regular development and maintenance activities.

This chapter begins with a discussion of measurement in our everyday lives. In the first section, we explain how measurement is a common and necessary practice for understanding, controlling, and improving our environment. In this section, the readers will see why measurement requires rigor and care. In the second section, we describe the role of measurement in software engineering. In particular, we look at how measurement needs are directly related to the goals we set and the questions we must answer when developing our software. Next, we compare software engineering measurement with measurement in other engineering disciplines, and propose specific objectives for software measurement. The last section provides a roadmap to the measurement topics discussed in the remainder of the book.

1.1 MEASUREMENT IN EVERYDAY LIFE

Measurement lies at the heart of many systems that govern our lives. Economic measurements determine price and pay increases. Measurements in radar systems enable us to detect aircraft when direct vision is obscured. Medical system measurements enable doctors to diagnose specific illnesses. Measurements in atmospheric systems are the basis for weather prediction. Without measurement, technology cannot function.

But measurement is not solely the domain of professional technologists. Each of us uses it in everyday life. Price acts as a measure of value of an item in a shop, and we calculate the total bill to make sure the shopkeeper gives us correct change. We use height and size measurements to ensure that our clothing will fit properly. When making a journey, we calculate distance, choose our route, measure our speed, and predict when we will arrive at our destination (and perhaps when we need to refuel). So, measurement helps us to understand our world, interact with our surroundings, and improve our lives.

1.1.1 What Is Measurement?

These examples present a picture of the variety in how we use measurement. But there is a common thread running through each of the described activities: in every case, some aspect of a thing is assigned a descriptor that allows us to compare it with others. In the shop, we can compare the price of one item with another. In the clothing store, we contrast sizes. And on

a journey, we compare distance traveled to distance remaining. The rules for assignment and comparison are not explicit in the examples, but it is clear that we make our comparisons and calculations according to a well-defined set of rules. We can capture this notion by defining measurement formally in the following way:

> *Measurement* is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way so as to describe them according to clearly defined rules.

Thus, measurement captures information about *attributes* of *entities*. An *entity* is an object (such as a person or a room) or an event (such as a journey or the testing phase of a software project) in the real world. We want to describe the entity by identifying characteristics that are important to us in distinguishing one entity from another. An *attribute* is a feature or property of an entity. Typical attributes include the area or color (of a room), the cost (of a journey), or the elapsed time (of the testing phase). Often, we talk about entities and their attributes interchangeably, as in "It is cold today" when we really mean that the air temperature is cold today, or "she is taller than he" when we really mean "her height is greater than his height." Such loose terminology is acceptable for everyday speech, but it is unsuitable for scientific endeavors. Thus, it is wrong to say that we measure things or that we measure attributes; in fact, we measure attributes of things. It is ambiguous to say that we "measure a room," since we can measure its length, area, or temperature. It is likewise ambiguous to say that we "measure the temperature," since we measure the temperature of a specific geographical location under specific conditions. In other words, what is commonplace in common speech is unacceptable for engineers and scientists.

When we describe entities by using attributes, we often define the attributes using numbers or symbols. Thus, price is designated as a number of dollars or pounds sterling, while height is defined in terms of inches or centimeters. Similarly, clothing size may be "small," "medium," or "large," while fuel is "regular," "premium," or "super." These numbers and symbols are abstractions that we use to reflect our perceptions of the real world. For example, in defining the numbers and symbols, we try to preserve certain relationships that we see among the entities. Thus, someone who is 6 feet in height is taller than someone who is 5 feet in height. Likewise, a "medium" T-shirt is smaller than a "large" T-shirt. This number or symbol can be very

useful and important. If we have never met Herman but are told that he is 7 feet tall, we can imagine his height in relation to ourselves without even having seen him. Moreover, because of his unusual height, we know that he will have to stoop when he enters the door of our office. Thus, we can make judgments about entities solely by knowing and analyzing their attributes.

Measurement is a process whose definition is far from clear-cut. Many different authoritative views lead to different interpretations about what constitutes measurement. To understand what measurement is, we must ask a host of questions that are difficult to answer. For example

- We have noted that color is an attribute of a room. In a room with blue walls, is "blue" a *measure* of the color of the room?
- The height of a person is a commonly understood attribute that can be measured. But what about other attributes of people, such as intelligence? Does an IQ test score adequately measure intelligence? Similarly, wine can be measured in terms of alcohol content ("proof"), but can wine quality be measured using the ratings of experts?
- The accuracy of a measure depends on the measuring instrument as well as on the definition of the measurement. For example, length can be measured accurately as long as the ruler is accurate and used properly. But some measures are not likely to be accurate, either because the measurement is imprecise or because it depends on the judgment of the person doing the measuring. For instance, the proposed measures of human intelligence or wine quality appear to have likely error margins. Is this a reason to reject them as bonafide measurements?
- Even when the measuring devices are reliable and used properly, there is margin for error in measuring the best understood physical attributes. For example, we can obtain vastly different measures for a person's height, depending on whether we make allowances for the shoes being worn or the standing posture. So how do we decide which error margins are acceptable and which are not?
- We can measure height in terms of meters, inches, or feet. These different *scales* measure the same attribute. But we can also measure height in terms of miles and kilometers—appropriate for measuring the height of a satellite above the Earth, but not for measuring the height of a person. When is a scale acceptable for the purpose to which it is put?

- Once we obtain measurements, we want to analyze them and draw conclusions about the entities from which they were derived. What kind of manipulations can we apply to the results of measurement? For example, why is it acceptable to say that Fred is twice as tall as Joe, but not acceptable to say that it is twice as hot today as it was yesterday? And why is it meaningful to calculate the mean of a set of heights (to say, e.g., that the average height of a London building is 200 m), but not the mean of the football jersey numbers of a team?

To answer these and many other questions, we examine the science of measurement in Chapter 2. This rigorous approach lays the groundwork for applying measurement concepts to software engineering problems. However, before we turn to measurement theory, we examine first the kinds of things that can be measured.

1.1.2 "What Is Not Measurable Make Measurable"

This phrase, attributable to Galileo Galilei (1564–1642), is part of the folklore of measurement scientists (Finkelstein 1982). It suggests that one of the aims of science is to find ways to measure attributes of interesting things. Implicit in Galileo's statement is the idea that measurement makes concepts more visible and therefore more understandable and controllable. Thus, as scientists, we should be creating ways to measure our world; where we can already measure, we should be making our measurements better.

In the physical sciences, medicine, economics, and even some social sciences, we can now measure attributes that were previously thought to be unmeasurable. Whether we like them or not, measures of attributes such as human intelligence, air quality, and economic inflation form the basis for important decisions that affect our everyday lives. Of course, some measurements are not as refined (in a sense to be made precise in Chapter 2) as we would like them to be; we use the physical sciences as our model for good measurement, continuing to improve measures when we can. Nevertheless, it is important to remember that the concepts of time, temperature, and speed, once unmeasurable by primitive peoples, are now not only commonplace but also easily measured by almost everyone; these measurements have become part of the fabric of our existence.

To improve the rigor of measurement in software engineering, we need not restrict the type or range of measurements we can make. Indeed, measuring the unmeasurable should improve our understanding of particular

entities and attributes, making software engineering as powerful as other engineering disciplines. Even when it is not clear how we might measure an attribute, the act of proposing such measures will open a debate that leads to greater understanding. Although some software engineers may continue to claim that important software attributes like dependability, quality, usability, and maintainability are simply not quantifiable, we prefer to try to use measurement to advance our understanding of them.

We can learn strategies for measuring unmeasurable attributes from the business community. Businesses often need to measure intangible attributes that you might think are unmeasurable such as customer satisfaction, future revenues, value of intellectual property, a company's reputation, etc. A key concern of the business community, including the software development business community, is risk. Businesses want to reduce the risk of product failures, late release of a product, loss of key employees, bankruptcy, etc. Thus, Douglas Hubbard provides an alternative definition of measurement:

> Measurement: A quantitatively expressed reduction of uncertainty based on one or more observations.
>
> HUBBARD 2010, P. 23

Observations that can reduce uncertainty can quantitatively measure the risk of negative events or the likelihood of positive outcomes.

EXAMPLE 1.1

Douglas Hubbard lists the kinds of statements that a business executive might want to make involving the use of measurement to reduce uncertainty, for example (Hubbard 2010):

> There is an 85% chance we will win our patent dispute.
> We are 93% certain our public image will improve after the merger.
>
> HUBBARD 2010, P. 24

He also shows that you need very little data to reduce uncertainty by applying what he calls the *Rule of Five*:

> There is a 93.75% chance that the median of a population is between the smallest and largest values in any random sample of five from the population.
>
> HUBBARD 2010, P. 30

We can apply the *Rule of Five* to measurements relevant to software engineering. Assume that you want to learn whether your organization's developers are writing class method bodies that are short, because you have heard somewhere that shorter method bodies are better. Rather than examining all methods in your code base, you could randomly select five methods and count the lines of code in their bodies. Say that these methods have bodies with 10, 15, 25, 45, and 50 lines of code. Using the *Rule of Five,* you know that there is a 93.75% chance that the median size of all method bodies in your code base is between 10 and 50 lines of code. With a sample of only five methods, you can count lines of code manually—you may not need a tool. Thus, using this method, you can easily find the median of a population with a quantified level of uncertainty. Using a larger sample can further reduce the uncertainty.

Software development involves activities and events under uncertain conditions. Requirements and user communities change unpredictably. Developers can leave a project at unpredictable times. We will examine the use of metrics in decision-making under conditions of uncertainty in Chapter 7.

Strictly speaking, we should note that there are two kinds of quantification: measurement and calculation. *Measurement* is a direct quantification, as in measuring the height of a tree or the weight of a shipment of bricks. *Calculation* is indirect, where we take measurements and combine them into a quantified item that reflects some attribute whose value we are trying to understand. For example, when the city inspectors assign a valuation to a house (from which they then decide the amount of tax owed), they calculate it by using a formula that combines a variety of factors, including the number of rooms, the type of heating and cooling, the overall floor space, and the sale prices of similar houses in comparable locations. The valuation is quantification, not a measurement, and its expression as a number makes it more useful than qualitative assessment alone. As we shall see in Chapter 2, we use *direct* and *derived* to distinguish measurement from calculation.

Sport offers us many lessons in measuring abstract attributes like quality in an objective fashion. Here, the measures used have been accepted universally, even though there is often discussion about changing or improving the measures. In the following examples, we highlight measurement concepts, showing how they may be useful in software engineering:

EXAMPLE 1.2

In the decathlon athletics event, we measure the time to run various distances as well as the length covered in various jumping activities. These measures are subsequently combined into an *overall score*, computed using a complex weighting scheme that reflects the importance of each component measure. Over the years, the weights and scoring rules have changed as the relative importance of an event or measure changes. Nevertheless, the overall score is widely accepted as a description of the athlete's all-around ability. In fact, the winner of the Olympic decathlon is generally acknowledged to be the world's finest athlete.

EXAMPLE 1.3

In European soccer leagues, a points system is used to select the best all-around team over the course of a season. Until the early 1980s, two points were awarded for each win and one point was awarded for each draw. Thereafter, the points system was changed; a win yielded three points instead of two, while a draw still yielded one point. This change was made to reflect the consensus view that the *qualitative difference* between a win and a draw was greater than that between a draw and a defeat.

EXAMPLE 1.4

There are no universally recognized measures to identify the best individual soccer players (although number of goals scored is a fairly accurate measure of quality of a striker). Although many fans and players have argued that player quality is an unmeasurable attribute, there are organizations (such as optasports.com) that provide player ratings based on a wide range of measurable attributes such as tackles, saves, or interceptions made; frequency and distance of passes (of various types), dribbles, and headers. Soccer clubs, agents, betting, and media companies pay large sums to acquire these ratings. Often clubs and players' agents use the ratings as the basis for determining player value (both from a salary and transfer price perspective).

It is easy to see parallels in software engineering. In many instances, we want an overall score that combines several measures into a "big picture" of what is going on during development or maintenance. We want to be able to tell if a software product is good or bad, based on a set of measures, each of which captures a facet of "goodness." Similarly, we want to

be able to measure an organization's ability to produce good software, or a model's ability to make good predictions about the software development process. The composite measures can be controversial, not only because of the individual measures comprising it, but also because of the weights assigned.

Likewise, controversy erupts when we try to capture qualitative information about some aspect of software engineering. Different experts have different opinions, and it is sometimes impossible to get consensus.

Finally, it is sometimes necessary to modify our environment or our practices in order to measure something new or in a new way. It may mean using a new tool (to count lines of code or evaluate code structure), adding a new step in a process (to report on effort), or using a new method (to make measurement simpler). In many cases, change is difficult for people to accept; there are management issues to be considered whenever a measurement program is implemented or changed.

1.2 MEASUREMENT IN SOFTWARE ENGINEERING

We have seen that measurement is essential to our daily lives, and measuring has become commonplace and well accepted. In this section, we examine the realm of software engineering to see why measurement is needed.

Software engineering describes the collection of techniques that apply an engineering approach to the construction and support of software products. Software engineering activities include managing, costing, planning, modeling, analyzing, specifying, designing, implementing, testing, and maintaining. By "engineering approach," we mean that each activity is understood and controlled, so that there are few surprises as the software is specified, designed, built, and maintained. Whereas computer science provides the theoretical foundations for building software, software engineering focuses on implementing the software in a controlled and scientific way.

The importance of software engineering cannot be understated, since software pervades our lives. From oven controls to automobiles, from banking transactions to air traffic control, and from sophisticated power plants to sophisticated weapons, our lives and the quality of life depend on software. For such a young profession, software engineering has usually done an admirable job of providing safe, useful, and reliable functionality. But there is room for a great deal of improvement. The literature is rife with examples of projects that have overrun their budgets and schedules.

Worse, there are too many stories about software that has put lives and businesses at risk.

Software engineers have addressed these problems by continually looking for new techniques and tools to improve process and product. Training supports these changes, so that software engineers are better prepared to apply the new approaches to development and maintenance. But methodological improvements alone do not make an engineering discipline.

1.2.1 Neglect of Measurement in Software Engineering

Engineering disciplines use methods that are underpinned by models and theories. For example, in designing electrical circuits, we appeal to theories like Ohm's law that describes the relationship between resistance, current, and voltage in the circuit. But the laws of electrical behavior have evolved by using the scientific method: stating a hypothesis, designing and running an experiment to test its validity, and analyzing the results. Underpinning the scientific process is measurement: measuring the variables to differentiate cases, measuring the changes in behavior, and measuring the causes and effects. Once the scientific method suggests the validity of a model or the truth of a theory, we continue to use measurement to apply the theory to practice. Thus, to build a circuit with a specific current and resistance, we know what voltage is required and we use instruments to measure whether we have such a voltage in a given battery.

It is difficult to imagine electrical, mechanical, and civil engineering without a central role for measurement. Indeed, science and engineering can be neither effective nor practical without measurement. But measurement has been considered a luxury in software engineering. For many development projects:

1. We fail to set measurable targets for our software products. For example, we promise that the product will be user-friendly, reliable, and maintainable without specifying clearly and objectively what these terms mean. As a result, from both is complete, we cannot tell if we have met our goals.

2. We fail to understand and quantify the component costs of software projects. For example, many projects cannot differentiate the cost of design from the cost of coding or testing. Since excessive cost is a frequent complaint from our customers, we cannot hope to control costs if we are not measuring the relative components of cost.

3. We do not quantify or predict the quality of the products we produce. Thus, we cannot tell a potential user how reliable a product will be in terms of likelihood of failure in a given period of use, or how much work will be needed to port the product to a different machine environment.

4. We allow anecdotal evidence to convince us to try yet another revolutionary new development technology, without doing a carefully controlled study to determine if the technology is efficient and effective. Promotional materials for software development tools and techniques typically include the following types of claims:

   a. "Our new technique guarantees 100% reliability."

   b. "Our tool improves productivity by 200%!!"

   c. "Build your code with half the staff in a quarter of the time."

   d. "Cuts test time by 2/3."

   These claims are generally not supported by scientific studies.

When measurements are made, they are often done infrequently, inconsistently, and incompletely. The incompleteness can be frustrating to those who want to make use of the results. For example, a developer may claim that 80% of all software costs involve maintenance, or that there are on average 55 faults in every 1000 lines of software code. But often we are not told how these results were obtained, how experiments were designed and executed, which entities were measured and how, and what were the realistic error margins. Without this additional information, we remain skeptical and unable to decide whether to apply the results to our own situations. In addition, we cannot do an objective study to repeat the measurements in our own environments. Thus, the lack of measurement in software engineering is compounded by the lack of a rigorous approach.

It is clear from other engineering disciplines that measurement can be effective, if not essential, in making characteristics and relationships more visible, in assessing the magnitude of problems, and in fashioning a solution to problems. As the pace of hardware innovation has increased, the software world has been tempted to relax or abandon its engineering underpinnings and hope for revolutionary gains. But now that software, playing a key role, involves enormous investment of energy and money,

it is time for software engineering to embrace the engineering discipline that has been so successful in other areas.

1.2.2 Objectives for Software Measurement

Even when a project is not in trouble, measurement is not only useful but also necessary. After all, how can you tell if your project is healthy if you have no measures of its health? So, measurement is needed at least for assessing the status of your projects, products, processes, and resources. Since we do not always know what derails a project, it is essential that we measure and record characteristics of good projects as well as bad. We need to document trends, the magnitude of corrective action, and the resulting changes. In other words, we must control our projects, not just run them. In Chapter 3, you will see the key role that measurement plays in evaluating software development organizations and their software development processes.

There are compelling reasons to consider the measurement process scientifically, so that measurement will be a true engineering activity. Every measurement action must be motivated by a particular goal or need that is clearly defined and easily understandable. That is, it is not enough to assert that we must measure to gain control. The measurement objectives must be specific, tied to what the managers, developers, and users need to know. Thus, these objectives may differ according to the kind of personnel involved and at which level of software development and use they are generated. But it is the goals that tell us how the measurement information will be used once it is collected.

We now describe the kinds of information needed to understand and control a software development project, from both manager and developer perspectives.

1.2.2.1 Managers

- *What does each process cost?* We can measure the time and effort involved in the various processes that comprise software production. For example, we can identify the cost to elicit requirements, the cost to specify the system, the cost to design the system, and the cost to code and test the system. In this way, we gain understanding not only of the total project cost but also of the contribution of each activity to the whole.

- *How productive is the staff?* We can measure the time it takes for staff to specify the system, design it, code it, and test it. Then, using

measures of the size of specifications, design, code, and test plans, for example, we can determine how productive the staff is at each activity. This information is useful when changes are proposed; the manager can use the productivity figures to estimate the cost and duration of the change.

- *How good is the code being developed?* By carefully recording faults, failures, and changes as they occur, we can measure software quality, enabling us to compare different products, predict the effects of change, assess the effects of new practices, and set targets for process and product improvement.
- *Will the user be satisfied with the product?* We can measure functionality by determining if all of the requirements requested have actually been implemented properly. And we can measure usability, reliability, response time, and other characteristics to suggest whether our customers will be happy with both functionality and performance.
- *How can we improve?* We can measure the time it takes to perform each major development activity, and calculate its effect on quality and productivity. Then we can weigh the costs and benefits of each practice to determine if the benefit is worth the cost. Alternatively, we can try several variations of a practice and measure the results to decide which is best; for example, we can compare two design methods to see which one yields the higher-quality code.

1.2.2.2 Developers

- *Are the requirements testable?* We can analyze each requirement to determine if its satisfaction is expressed in a measurable, objective way. For example, suppose a requirement states that a web-based system must be "fast"; the requirement can be replaced by one that states that the mean response time to a set of specific of inputs must be less than 2 s for specified browsers and number of concurrent users.
- *Have we found all the faults?* We can measure the number of faults in the specification, design, code, and test plans and trace them back to their root causes. Using models of expected detection rates, we can use this information to decide whether inspections and testing have been effective and whether a product can be released for the next phase of development.

- *Have we met our product or process goals?* We can measure characteristics of the products and processes that tell us whether we have met standards, satisfied a requirement, or met a process goal. For example, certification may require that fewer than 20 failures have been reported per beta-test site over a given period of time. Or a standard may mandate that all modules must pass code inspections. The testing process may require that unit testing must achieve 90% statement coverage.
- *What will happen in the future?* We can measure attributes of existing products and current processes to make predictions about future ones. For example, measures of size of specifications can be used to predict the size of the target system, predictions about future maintenance problems can be made from measures of structural properties of the design documents, and predictions about the reliability of software in operational use can be made by measuring reliability during testing.

1.2.3 Measurement for Understanding, Control, and Improvement

The information needs of managers and developers show that measurement is important for three basic activities. First, measurement can help us to *understand* what is happening during development and maintenance. We assess the current situation, establishing baselines that help us to set goals for future behavior. In this sense, the measurements make aspects of process and product more visible, giving us a better understanding of relationships among activities and the entities they affect.

Second, the measurement allows us to *control* what is happening in our projects. Using our baselines, goals, and understanding of relationships, we predict what is likely to happen and make changes to processes and products that help us to meet our goals. For example, we may monitor the complexity of code modules, giving thorough review only to those that exceed acceptable bounds.

Third, measurement encourages us to *improve* our processes and products. For instance, we may increase the number or type of design reviews we do, based on measures of specification quality and predictions of likely design quality.

No matter how measurements are used, it is important to manage the expectations of those who will make measurement-based decisions. Users of the data should always be aware of the limited accuracy of prediction

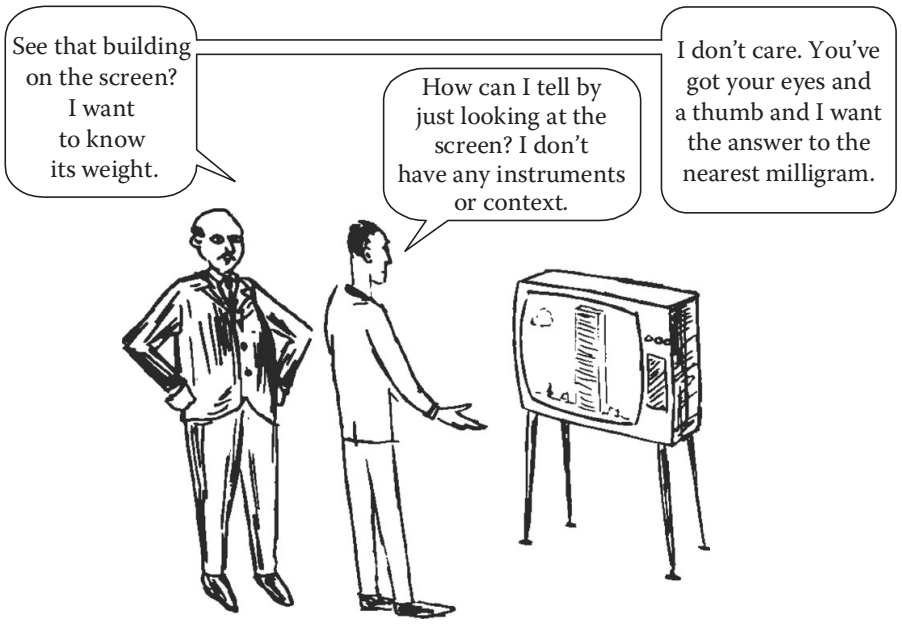

FIGURE 1.1 Software measurement—resource estimation.

and of the margin of error in the measurements. As with any other engineering discipline, there is room in software engineering for abuse and misuse of measurement. In particular, managers may pressure developers to produce precise measures with inadequate models, tools, and techniques (see Figure 1.1).

It is wrong to expect measurement to provide instant, easy solutions to your software engineering problems. Control and accurate prediction both require careful sets of measurements.

1.3 SCOPE OF SOFTWARE METRICS

Software metrics is a term that embraces many activities, all of which involve some degree of software measurement:

- Cost and effort estimation models and measures
- Data collection
- Quality models and measures
- Reliability models
- Security metrics
- Structural and complexity metrics
- Capability maturity assessment

- Management by metrics
- Evaluation of methods and tools

Each of these activities will be covered in some detail. Our theoretical foundations, to be described in Chapters 2 and 3, will enable us to consider the activities in a unified manner, rather than as diverse, unrelated topics.

The following brief introduction will give you a sense of the techniques currently in use for each facet of measurement. It provides signposts to where the material is covered in detail.

1.3.1 Cost and Effort Estimation

Managers provided the original motivation for deriving and using software measures. They wanted to be able to predict project costs during early phases in the software life cycle. As a result, numerous models for software cost and effort estimation have been proposed and used. Examples include Boehm's COCOMO II model (Boehm et al. 2000) and Albrecht's function point model (Albrecht 1979). These and other models often share a common approach: effort is expressed as a (predefined) function of one or more variables (such as size of the product, capability of the developers, and level of reuse). Size is usually defined as (predicted) lines of code or number of function points (which may be derived from the product specification). These cost and effort prediction models are discussed in Chapter 8.

1.3.2 Data Collection

The quality of any measurement program is clearly dependent on careful data collection. But collecting data is easier said than done, especially when data must be collected across a diverse set of projects. Thus, data collection is becoming a discipline in itself, where specialists work to ensure that measures are defined unambiguously, that collection is consistent and complete, and that data integrity is not at risk. But it is acknowledged that metrics data collection must be planned and executed in a careful and sensitive manner. We will see in Chapter 5 how useful data can be collected. Chapter 6 shows how to analyze and display collected data in order to draw valid conclusions and make decisions.

Data collection is also essential for scientific investigation of relationships and trends. We will see in Chapter 4 how good experiments, surveys, and case studies require carefully planned data collection, as well as thorough analysis and reporting of the results.

1.3.3 Quality Models and Measures

Since software quality involves many diverse factors, software engineers have developed models of the interaction between multiple quality factors. These models are usually constructed in a tree-like fashion, similar to Figure 1.2. The upper branches hold important high-level quality factors of software products, such as reliability and usability, that we would like to quantify. Each quality factor is composed of lower-level criteria, such as modularity and data commonality. The criteria are easier to understand and measure than the factors; thus, actual measures (metrics) are proposed for the criteria. The tree describes the pertinent relationships between factors and their dependent criteria, so we can measure the factors in terms of the dependent criteria measures. This notion of divide and conquer has been implemented as a standard approach to measuring software quality (IEEE 1061-2009). Quality models are described in Chapter 10.

1.3.4 Reliability Models

Most quality models include reliability as one of their component factors. But the need to predict and measure reliability itself has led to a separate specialization in reliability modeling and prediction. Chapter 11 describes

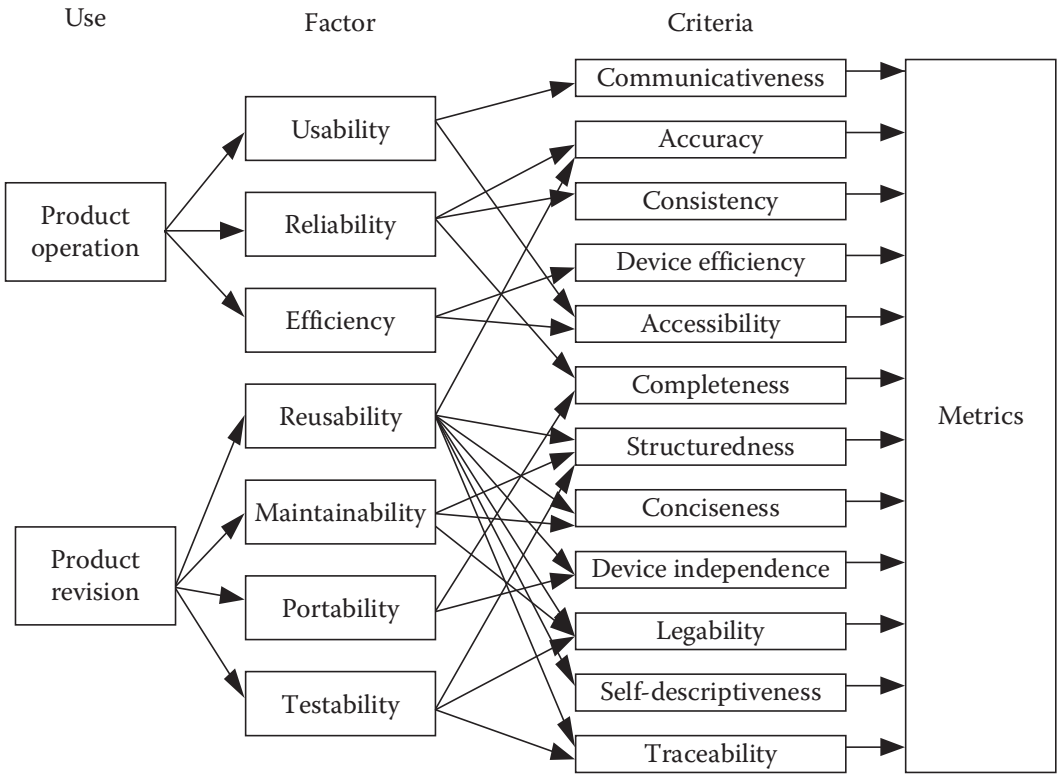

FIGURE 1.2 Software quality model.

software reliability models and measures starting with reliability theory. Chapter 11 shows how to apply reliability theory to analyze and predict software reliability.

1.3.5 Security Metrics

As computing has become part of almost every human activity, our concerns about the security of software systems have grown. We worry that attackers will steal or corrupt data files, passwords, and our accounts. Security depends on both the internal design of a system and the nature of the attacks that originate externally. In Chapter 10, we describe some standard ways to assess security risks in terms of impact, likelihood, threats, and vulnerabilities.

1.3.6 Structural and Complexity Metrics

Desirable quality attributes like reliability and maintainability cannot be measured until some operational version of the code is available. Yet, we wish to be able to predict which parts of the software system are likely to be less reliable, more difficult to test, or require more maintenance than others, even before the system is complete. As a result, we measure structural attributes of representations of the software that are available in advance of (or without the need for) execution; then, we try to establish empirically predictive theories to support quality assurance, quality control, and quality prediction. These representations include control flow graphs that usually model code and various unified modeling language (UML) diagrams that model software designs and requirements. Structural metrics can involve the arrangement of program modules, for example, the use and properties of design patterns. These models and related metrics are described in Chapter 9.

1.3.7 Capability Maturity Assessment

The US Software Engineering Institute (SEI) has developed and maintained a software development process evaluation technique, the Capability Maturity Model Integration (CMMI®) for Development. The original objective was to measure a contractor's ability to develop quality software for the US government, but CMMI ratings are now used by many other organizations. The CMMI assesses many different attributes of development, including the use of tools, standard practices, and more. A CMMI assessment is performed by an SEI certified assessor and involves determining the answers to hundreds of questions concerning

software development practices in the organization being reviewed. The result of an assessment is a rating that is on a five-level scale, from *Level 1* (*Initial*—development is dependent on individuals) to *Level 5* (*Optimizing*—a development process that can be optimized based on quantitative process information).

The evaluation of process maturity has become much more common. Many organizations require contracted software development vendors to have CMMI certification at specified levels. In Chapter 3, we describe the CMMI evaluation mechanism and how process maturity can be useful in understanding what and when to measure, and how measurement can guide process improvements.

1.3.8 Management by Metrics

Measurement is an important part of software project management. Customers and developers alike rely on measurement-based charts and graphs to help them decide if a project is on track. Many companies and organizations define a standard set of measurements and reporting methods, so that projects can be compared and contrasted. This uniform collection and reporting is especially important when software plays a supporting role in an overall project. That is, when software is embedded in a product whose main focus is a business area other than software, the customer or ultimate user is not usually well versed in software terminology, so measurement can paint a picture of progress in general, understandable terms. For example, when a power plant designer asks a software developer to write control software, the power plant designer usually knows a lot about power generation and control, but very little about software development processes, programming languages, software testing, or computer hardware. Measurements must be presented in a way that tells both customer and developer how the project is doing. In Chapter 6, we will examine several measurement analysis and presentation techniques that are useful and understandable to all who have a stake in a project's success.

1.3.9 Evaluation of Methods and Tools

The literature is rife with descriptions of new methods and tools that may make your organization or project more productive and your products better and cheaper. But it is difficult to separate the claims from the reality. Many organizations perform experiments, run case studies, or administer surveys to help them decide whether a method or tool is likely to make a positive difference in their particular situations. These investigations

cannot be done without careful, controlled measurement and analysis. As we will see in Chapter 4, an evaluation's success depends on good experimental design, proper identification of the factors likely to affect the outcome, and appropriate measurement of factor attributes.

1.4 SUMMARY

This introductory chapter has described how measurement pervades our everyday life. We have argued that measurement is essential for good engineering in other disciplines; it should likewise become an integral part of software engineering practice. In particular

- Owing to the lessons of other engineering disciplines, measurement now plays a significant role in software engineering.
- Software measurement is a diverse collection of topics that range from models for predicting software project costs at the specification stage to measures of program structure.
- General reasons for needing software engineering measurement are not enough. Engineers must have specific, clearly stated objectives for measurement.
- We must be bold in our attempts at measurement. Just because no one has measured some attribute of interest does not mean that it cannot be measured satisfactorily.

We have set the scene for a new perspective on software metrics. Our foundation, introduced in Chapter 2, supports a scientific and effective approach to constructing, calculating, and appropriately applying the metrics that we derive.

EXERCISES

1. Explain the role of measurement in determining the best players in your favorite sport.
2. Explain how measurement is commonly used to determine the best students in a university. Describe any problems with these measurement schemes.
3. How would you begin to measure the quality of a software product?