CHAPTER 2

The Basics of Measurement

In Chapter 1, we saw how measurement pervades our world. We use measurement everyday to understand, control, and improve what we do and how we do it. In this chapter, we examine measurement in more depth, trying to apply general measurement lessons learned in daily activities to the activities we perform as part of software development.

Ordinarily, when we measure things, we do not think about the scientific principles we are applying. We measure attributes such as the length of physical objects, the timing of events, and the temperature of liquids or of the air. To do the measuring, we use both tools and principles that we now take for granted. However, these sophisticated measuring devices and techniques have been developed over time, based on the growth of understanding of the attributes we are measuring. For example, using the length of a column of mercury to capture information about temperature is a technique that was not at all obvious to the first person who wanted to know how much hotter it is in summer than in winter. As we understood more about temperature, materials, and the relationships between them, we developed a framework for describing temperature as well as tools for measuring it.

Unfortunately, we have no comparably deep understanding of software attributes. Nor do we have the associated sophisticated measurement tools. Questions that are relatively easy to answer for non-software

entities are difficult for software. For example, consider the following questions:

1. How much must we know about an attribute before it is reasonable to consider measuring it? For instance, do we know enough about "complexity" of programs to be able to measure it?
2. How do we know if we have really measured the attribute we wanted to measure? For instance, does a count of the number of "bugs" found in a system during integration testing measure the quality of the system? If not, what does the count tell us?
3. Using measurement, what meaningful statements can we make about an attribute and the entities that possess it? For instance, is it meaningful to talk about doubling a design's quality? If not, how do we compare two different designs?
4. What meaningful operations can we perform on measures? For instance, is it sensible to compute average productivity for a group of developers, or the average quality of a set of modules?

To answer these questions, we must establish the basics of a theory of measurement. We begin by examining formal measurement theory, developed as a classical discipline from the physical sciences. We see how the concepts of measurement theory apply to software, and we explore several examples to determine when measurements are meaningful and useful in decision-making. This theory tells us not only when and how to measure, but also how to analyze and depict data, and how to tie the results back to our original questions about software quality and productivity.

2.1 THE REPRESENTATIONAL THEORY OF MEASUREMENT

In any measurement activity, there are rules to be followed. The rules help us to be consistent in our measurement, as well as providing a basis for interpretation of data. Measurement theory tells us the rules, laying the groundwork for developing and reasoning about all kinds of measurement. This rule-based approach is common in many sciences. For example, recall that mathematicians learned about the world by defining axioms for a geometry. Then, by combining axioms and using their results to support or refute their observations, they expanded their understanding and the set of rules that govern the behavior of objects. In the same way, we

can use rules about measurement to codify our initial understanding, and then expand our horizons as we analyze our software.

However, just as there are several kinds of geometry (e.g., Euclidean and non-Euclidean) with each depending on the set of rules chosen, there are also several theories of measurement. In this book, we present an overview of the *representational* theory of measurement.

2.1.1 Empirical Relations

The *representational theory of measurement* seeks to formalize our intuition about the way the world works. That is, the data we obtain as measures should represent attributes of the entities we observe, and manipulation of the data should preserve relationships that we observe among the entities. Thus, our intuition is the starting point for all measurement.

Consider the way we perceive the real world. We tend to understand things by comparing them, not by assigning numbers to them. For example, Figure 2.1 illustrates how we learn about height. We observe

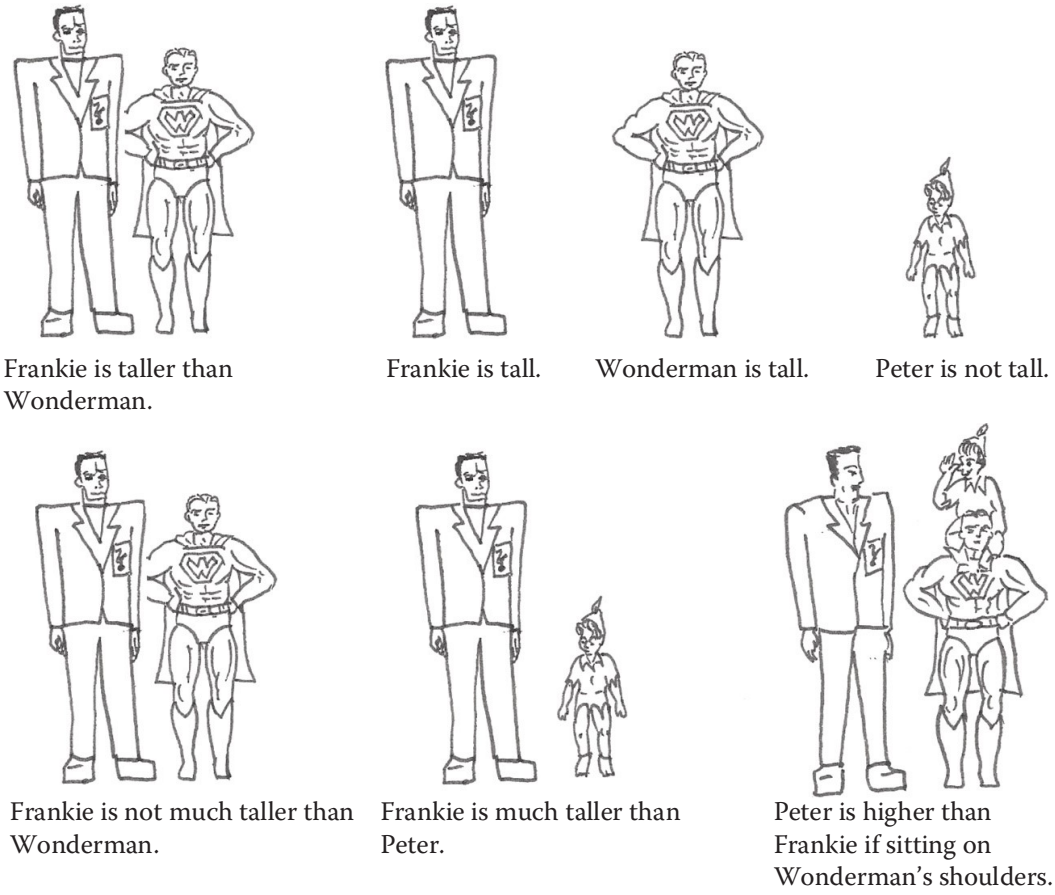

FIGURE 2.1 Some empirical relations for the attribute *height*.

that certain people are *taller than* others without actually measuring them. It is easy to see that Frankie is taller than Wonderman who in turn is taller than Peter; anyone looking at this figure would agree with this statement. However, our observation reflects a set of rules that we are imposing on the set of people. We form pairs of people and define a binary relation on them. In other words, *taller than* is a binary relation defined on the set of pairs of people. Given any two people, x and y, we can observe that

- x is taller than y, or
- y is taller than x

Therefore, we say that *taller than* is an empirical relation for height.

When the two people being compared are very close in height, we may find a difference of opinion; you may think that Jack is taller than Jill, while we are convinced that Jill is taller than Jack. Our empirical relations permit this difference by requiring only a consensus of opinion about relationships in the real world. A (binary) empirical relation is one for which there is a reasonable consensus about which pairs are in the relation.

We can define more than one empirical relation on the same set. For example, Figure 2.1 also shows the relation *much taller than*. Most of us would agree that both Frankie and Wonderman are much taller than Peter (although there is less of a consensus about this relation than *taller than*).

Empirical relations need not be binary. That is, we can define a relation on a single element of a set, or on collections of elements. Many empirical relations are unary, meaning that they are defined on individual entities. The relation *is tall* is an example of a unary relation in Figure 2.1; we can say Frankie is tall but Peter is not tall. Similarly, we can define a ternary relationship by comparing groups of three; Figure 2.1 shows how Peter sitting on Wonderman's shoulders is higher than Frankie.

We can think of these relations as mappings from the empirical, real world to a formal mathematical world. We have entities and their attributes in the real world, and we define a mathematical mapping that preserves the relationships we observe. Thus, height (that is, tallness) can be considered as a mapping from the set of people to the set of real numbers. If we can agree that Jack is *taller than* Jill, then any measure of height should assign a higher number to Jack than to Jill. As we shall see later in

TABLE 2.1 Sampling 100 Users to Express Preferences among Products A, B, C, and D

	More Functionality				More User-Friendly			
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**
A	—	80	10	80	—	45	50	44
B	20	—	5	50	55	—	52	50
C	90	95	—	96	50	48	—	51
D	20	50	4	—	54	50	49	—

this chapter, this preservation of intuition and observation is the notion behind the representation condition of measurement.

EXAMPLE 2.1

Suppose we are evaluating the four best-selling contact management programs: A, B, C, and D. We ask 100 independent computer users to rank these programs according to their functionality, and the results are shown on the left-hand portion of Table 2.1. Each cell of the table represents the percentage of respondents who preferred the row's program to the column's program; for instance, 80% rated program A as having greater functionality than B. We can use this survey to define an empirical relation *greater functionality than* for contact management programs; we say that program *x* has greater functionality than program *y* if the survey result for cell (*x*,*y*) exceeds 60%. Thus, the relation consists of the pairs (C,A), (C,B), (C,D), (A,B), and (A,D). This set of pairs tells us more than just five comparisons; for example, since C has greater functionality than A, and A in turn has greater functionality than B and D, then C has greater functionality than B and D. Note that neither pair (B,D) nor (D,B) is in the empirical relation; there is no clear consensus about which of B and D has greater functionality.

Suppose we administer a similar survey for the attribute *user-friendliness*, with the results shown on the right-hand side of Table 2.1. In this case, there is no real consensus at all. At best, we can deduce that *greater user-friendliness* is an empty empirical relation. This statement is different from saying that all the programs are equally user-friendly, since we did not specifically ask the respondents about indifference or equality. Thus, we deduce that our understanding of user-friendliness is so immature that there are no useful empirical relations.

Example 2.1 shows how we can start with simple user surveys to gain a preliminary understanding of relationships. However, as our understanding grows, we can define more sophisticated measures.

TABLE 2.2 Historical Advances in Temperature Measurement

2000 BC	Rankings, *hotter than*
1600 AD	First thermometer measuring *hotter than*
1720 AD	Fahrenheit scale
1742 AD	Celsius scale
1854 AD	Absolute zero, Kelvin scale

EXAMPLE 2.2

Table 2.2 shows that people had an initial understanding of temperature thousands of years ago. This intuition was characterized by the notion of *hotter than*. Thus, for example, by putting your hand into two different containers of liquid, you could feel if one were hotter than the other. No measurement is necessary for this determination of temperature difference. However, people needed to make finer discriminations in temperature. In 1600, the first device was constructed to capture this comparative relationship; the thermometer could consistently assign a higher number to liquids that were *hotter than* others.

Example 2.2 illustrates an important characteristic of measurement. We can begin to understand the world by using relatively unsophisticated relationships that require no measuring tools. Once we develop an initial understanding and have accumulated some data, we may need to measure in more sophisticated ways and with special tools. Analyzing the results often leads to the clarification and re-evaluation of the attribute and yet more sophisticated empirical relations. In turn, we have improved accuracy and increased understanding.

Formally, we define *measurement* as the mapping from the empirical world to the formal, relational world. Consequently, a *measure* is the number or symbol assigned to an entity by this mapping in order to characterize an attribute.

Sometimes, the empirical relations for an attribute are not yet agreed, especially when they reflect personal preference. We see this lack of consensus when we look at the ratings of wine or the preference for a design technique, for example. Here, the raters have some notion of the attribute they want to measure, but there is not always a common understanding. We may find that what is tasteless or difficult for one rater is delicious or easy for another rater. In these cases, we can still perform a subjective

Likert Scale

Give the respondent a statement with which to agree or disagree. Example:

This software program is reliable.

Strongly Agree	Agree	Neither agree nor disagree	Disagree	Strongly Disagree

Forced Ranking

Give *n* alternatives, ordered from 1 (best) to *n* (worst). Example:

Rank the following five software modules in order of maintenance difficulty, with 1 = least complex, 5 = most complex:

—	Module A
—	Module B
—	Module C
—	Module D
—	Module E

Verbal Frequency Scale

Example: How often does this program fail?

Always Often Sometimes Seldom Never

Ordinal Scale

List several ordered alternatives and have respondents select one. For example:

How often does the software fail?

1. Hourly
2. Daily
3. Weekly
4. Monthly
5. Several times a year
6. Once or twice a year
7. Never

Comparative Scale

Very superior About the same Very inferior

1 2 3 4 5 6 7 8

Numerical Scale

Unimportant Important

1 2 3 4 5 6 7 8

FIGURE 2.2 Subjective rating schemes.

assessment, but the result is not necessarily a measure, in the sense of measurement theory. For example, Figure 2.2 shows several rating formats, some of which you may have encountered in taking examinations or opinion polls. These questionnaires capture useful data. They enable us to establish the basis for empirical relations, characterizing properties so that formal measurement may be possible in the future.

2.1.2 The Rules of the Mapping

We have seen how a measure is used to *characterize* an attribute. We begin in the real world, studying an entity and trying to understand more about it. Thus, the real world is the *domain* of the mapping, and the mathematical world is the *range*. When we map the attribute to a mathematical system, we have many choices for the mapping and the range. We can use real numbers, integers, or even a set of non-numeric symbols.

EXAMPLE 2.3

To measure a person's height, it is not enough to simply specify a number. If we measure height in inches, then we are defining a mapping from the set of people into inches; if we measure height in centimeters, then we have a different mapping. Moreover, even when the domain and range are the same, the mapping definition may be different. That is, there may be many different mappings (and hence different ways of measuring) depending on the conventions we adopt. For example, we may or may not allow shoes to be worn, or we may measure people standing or sitting.

Thus, a measure must specify the domain and range as well as the rule for performing the mapping.

EXAMPLE 2.4

In some everyday situations, a measure is associated with a number, the assumptions about the mapping are well known, and our terminology is imprecise. For example, we say "Felix's age is 11," or "Felix is 11." In expressing ourselves in this way, we really mean that we are measuring age by mapping each person into years in such a way that we count only whole years since birth. But there are many different rules that we can use. For example, the Chinese measure age by counting from the time of conception;

their assumptions are therefore different, and the resulting number is different. For this reason, we must make the mapping rules explicit.

We encounter some of the same problems in measuring software. For example, many organizations measure the size of their source code in terms of the number of lines of code (LOC) in a program. But the definition of a line of code must be made clear. The US Software Engineering Institute developed a checklist to assist developers in deciding exactly what is included in a line of code (Park 1992). This standard is still used by major organizations and metrics tool providers (see, e.g., the *unified code counter* tool produced by University of Southern California in collaboration with the Aerospace Corporation (Pfeiffer, 2012). Figure 2.3 illustrates part of the checklist, showing how different choices result in different counting rules. Thus, the checklist allows you to tailor your definition of lines-of-code to your needs. We will examine the issues addressed by this checklist in more depth in Chapter 8.

Many systems consist of programs in a variety of languages. For example, the GNU/Linux distribution includes code written in at least 19 different languages (Wheeler 2002). In order to deal with code written in such a variety of languages, David Wheeler's code analysis tool uses a simple scheme for counting LOC: "a physical source line of code is a line ending in a newline or end-of-file marker, and which contains at least one non-whitespace non-comment character."

2.1.3 The Representation Condition of Measurement

We saw that, by definition, each relation in the empirical relational system corresponds via the measurement to an element in a number system. We want the properties of the measures in the number system to be the same as the corresponding elements in the real world, so that by studying the numbers, we learn about the real world. Thus, we want the mapping to preserve the relation. This rule is called the representation condition, and it is illustrated in Figure 2.4.

The *representation condition* asserts that a measurement mapping M must map entities into numbers and empirical relations into numerical relations in such a way that the empirical relations preserve and are preserved by the numerical relations. In Figure 2.4, we see that the empirical relation *taller than* is mapped to the numerical relation >. In particular, we can say that

A is *taller than* B if and only if $M(A) > M(B)$

Statement type	*Include*	*Exclude*
Executable		
Nonexecutable		
Declarations		
Compiler directives		
Comments		
On their own lines		
On lines with source code		
Banners and nonblank spacers		
Blank (empty) comments		
Blank lines		

How produced	*Include*	*Exclude*
Programmed		
Generated with source code generators		
Converted with automatic translators		
Copied or reused without change		
Modified		
Removed		

Origin	*Include*	*Exclude*
New work: no prior existence		
Prior work: taken or adapted from		
A previous version, build, or release		
Commercial, off-the-shelf software, other than libraries		
Government furnished software, other than reuse libraries		
Another product		
A vendor-supplied language support library (unmodified)		
A vendor-supplied operating system or utility (unmodified)		
A local or modified language support library or operating system		
Other commercial library		
A reuse library (software designed for reuse)		
Other software component or library		

FIGURE 2.3 Portion of US Software Engineering Institute checklist for lines-of-code count.

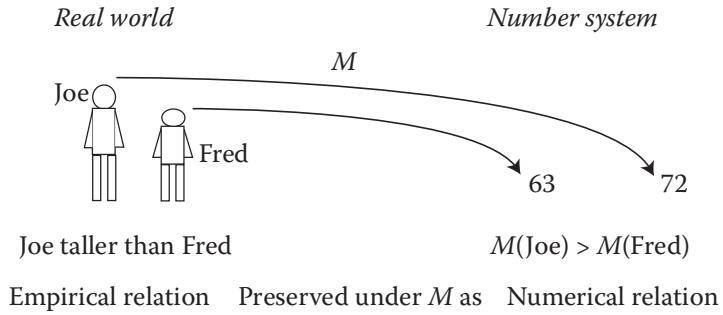

FIGURE 2.4 Representation condition.

This statement means that:

- Whenever Joe is taller than Fred, then M(Joe) must be a bigger number than M(Fred).
- We can map Jill to a higher number than Jack only if Jill is taller than Jack.

EXAMPLE 2.5

In Section 2.1.1, we noted that there can be many relations on a given set, and we mentioned several for the attribute *height*. The representation condition has implications for each of these relations. Consider these examples:

For the (binary) empirical relation *taller than*, we can have the numerical relation

$$x > y$$

Then, the representation condition requires that for any measure M,

$$A \textit{ taller than } B \text{ if and only if } M(A) > M(B)$$

For the (unary) empirical relation *is-tall*, we might have the numerical relation

$$x > 70$$

The representation condition requires that for any measure M,

$$A \textit{ is-tall} \text{ if and only if } M(A) > 70$$

For the (binary) empirical relation *much taller than,* we might have the numerical relation

$$x > y + 15$$

The representation condition requires that for any measure M,

A *much taller than B* if and only if $M(A) > M(B) + 15$

For the (ternary) empirical relation *x higher than y if sitting on z's shoulders,* we could have the numerical relation

$$0.7x + 0.8z > y$$

The representation condition requires that for any measure M,

A higher than B if sitting on C's shoulders if and only if
$0.7M(A) + 0.8M(C) > M(B)$

Consider the actual assignment of numbers M given in Figure 2.5. Wonderman is mapped to the real number 72 (i.e., M(Wonderman) = 72), Frankie to 84 (M(Frankie) = 84), and Peter to 42 (M(Peter) = 42). With this particular mapping M, the four numerical relations hold whenever the four empirical relations hold. For example,

- Frankie is taller than Wonderman, and M(Frankie) > M(Wonderman).
- Wonderman is tall, and M(Wonderman) = 72 > 70.
- Frankie is much taller than Peter, and M(Frankie) = 84 > 57 = M(Peter) + 15. Similarly Wonderman is much taller than Peter and M(Wonderman) = 72 > 57 = M(Peter) + 15.
- Peter is higher than Frankie when sitting on Wonderman's shoulders, and $0.7M$(Peter) + $0.8M$(Wonderman) = 87 > 84 = M(Frankie)

Since all the relations are preserved in this way by the mapping, we can define the mapping as a *measure* for the attribute. Thus, if we think of the

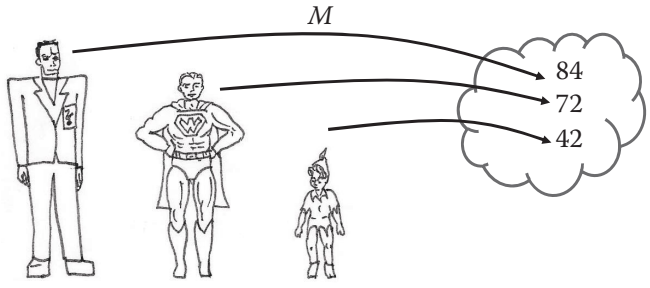

FIGURE 2.5 A measurement mapping.

measure as a measure of height, we can say that Frankie's height is 84, Peter's is 42, and Wonderman's is 72.

Not every assignment satisfies the representation condition. For instance, we could define the mapping in the following way:

$$M(\text{Wonderman}) = 72$$

$$M(\text{Frankie}) = 84$$

$$M(\text{Peter}) = 60$$

Then three of the above relations are satisfied, but *much taller than* is not. This is because *Wonderman is much taller than Peter* is not true under this mapping.

The mapping that we call a measure is sometimes called a *representation* or *homomorphism*, because the measure represents the attribute in the numerical world. Figure 2.6 summarizes the steps in the measurement process.

There are several conclusions we can draw from this discussion. First, we have seen that there may be many different measures for a given attribute. In fact, we use the notion of representation to define validity: any measure that satisfies the representation condition is a *valid* measure. Second, the richer the empirical relation system, the fewer the valid measures. We consider a relational system to be rich if it has a large number of

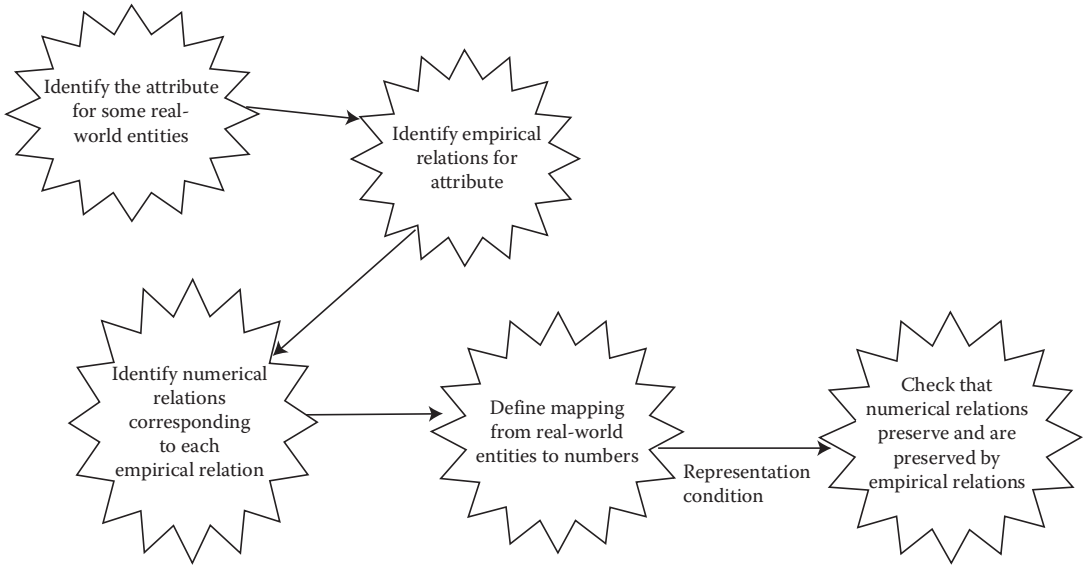

FIGURE 2.6 Key stages of formal measurement.

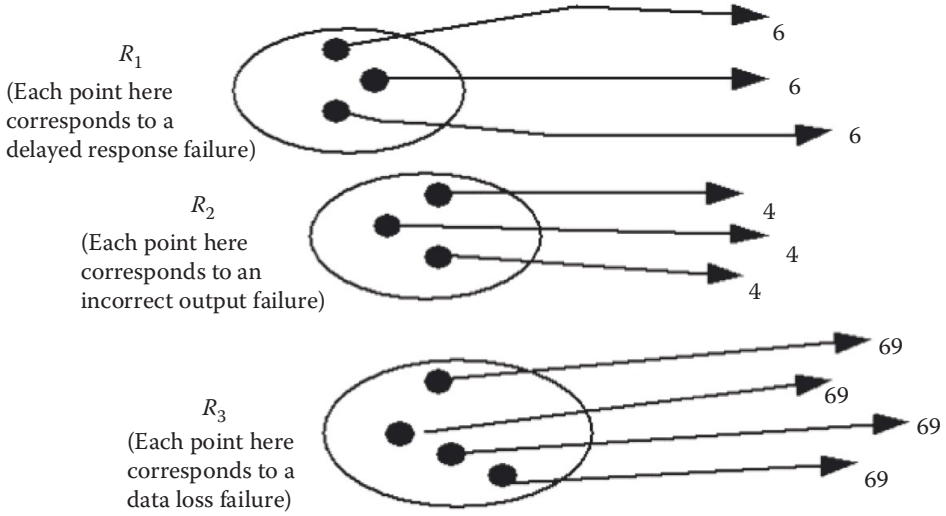

FIGURE 2.7 Measurement mapping.

relations that can be defined. But as we increase the number of empirical relations, we increase the number of conditions that a measurement mapping must satisfy in the representation condition.

EXAMPLE 2.6

Suppose we are studying the entity *software failures*, and we look at the attribute *criticality*. Our initial description distinguishes among only three types of failures:

- Delayed-response
- Incorrect output
- Data-loss

where every failure lies in exactly one failure class (based on which outcome happens first). This categorization yields an empirical relation system that consists of just three unary relations: R_1 for delayed response, R_2 for incorrect output, and R_3 for data loss. We assume every failure is in either R_1, R_2, or R_3. At this point, we cannot judge the relative criticality of these failure types; we know only that the types are different.

To find a representation for this empirical relation system in the set of real numbers, we need to choose only any three distinct numbers, and then map members from different classes into different numbers. For example, the mapping *M,* illustrated in Figure 2.7, assigns the mapping as:

$$M(\text{each delayed response}) = 6$$

$$M(\text{each incorrect output}) = 4$$

$$M(\text{each data loss}) = 69$$

This assignment is a representation, because we have numerical relations corresponding to R_1, R_2, and R_3. That is, the numerical relation corresponding to R_1 is the relation is 6; likewise, the numerical relation corresponding to R_2 is the relation is 4, and the numerical relation corresponding to R_3 is the relation is 69.

Suppose next we have formed a deeper understanding of failure criticality in a particular environment. We want to add to the above relation system a new (binary) relation, *is more critical than*. We now know that each data-loss failure is more critical than each incorrect output failure and delayed response failure; each incorrect output failure is more critical than each delayed response failure. Thus, *x more critical than y* contains all those pairs (*x,y*) of failures for which either

$$x \text{ is in } R_3 \text{ and } y \text{ is in } R_2 \text{ or } R_1, \text{ or}$$

$$x \text{ is in } R_2 \text{ and } y \text{ is in } R_1$$

To find a representation in the real numbers for this enriched empirical relation system, we now have to be much more careful with our assignment of numbers. First of all, we need a numerical relation to correspond to *more critical than*, and it is reasonable to use the binary relation >. However, it is not enough to simply map different failure types to different numbers. To preserve the new relation, we must ensure that data-loss failures are mapped to a higher number than incorrect output failures, which in turn are mapped to a higher number than delayed-response failures. One acceptable representation is the mapping:

$$M(\text{each delayed response}) = 3$$

$$M(\text{each incorrect output}) = 4$$

$$M(\text{each data-loss}) = 69$$

Note that the mapping defined initially in this example would *not* be a representation, because > does not preserve *is more critical than;* incorrect output failures were mapped to a lower number than delayed response failures.

There is nothing wrong with using the same representation in different ways, or using several representations for the same attribute. Table 2.3 illustrates a number of examples of specific measures used in software engineering. In it, we see that examples 1 and 2 in Table 2.3 give different measures of program length, while examples 9 and 10 give different measures of program reliability. Similarly, the same measure (although

TABLE 2.3 Examples of Specific Measures Used in Software Engineering

	Entity	Attribute	Measure
1	Completed project	Duration	Months from start to finish
2	Completed project	Duration	Days from start to finish
3	Program code	Length	Number of lines of code (LOC)
4	Program code	Length	Number of executable statements
5	Integration testing process	Duration	Hours from start to finish
6	Integration testing process	Rate at which faults are found	Number of faults found per KLOC (thousand LOC)
7	Test set	Efficiency	Number of faults found per number of test cases
8	Test set	Effectiveness	Number of faults found per KLOC (thousand LOC)
9	Program code	Reliability	Mean time to failure (MTTF) in CPU hours
10	Program code	Reliability	Rate of occurrence of failures (ROCOF) in CPU hours

of course not the same measurement mapping), *faults found per thousand lines of code* (*KLOC*), is used in examples 6, 7, and 8.

How good a measure is faults per KLOC? The answer depends entirely on the entity–attribute pair connected by the mapping. Intuitively, most of us would accept that faults per KLOC is a good measure of the *rate at which faults are found* for the *testing process* (example 6). However, it is not such a good measure of *efficiency* of the *tester* (example 7), because intuitively we feel that we should also take into account the difficulty of understanding and testing the program under scrutiny. This measure may be reasonable when comparing two testers of the same program, though. Faults per KLOC is not likely to be a good measure of *quality* of the *program code*; if integration testing revealed program *X* to have twice as many faults per KLOC than program *Y*, we would probably not conclude that the quality of program *Y* was twice that of program *X*.

2.2 MEASUREMENT AND MODELS

In Chapter 1, we have discussed several types of *models*: cost estimation models, quality models, capability maturity models, and more. In general, a *model* is an abstraction of reality, allowing us to strip away detail and view an entity or concept from a particular perspective. For example, cost

models permit us to examine only those project aspects that contribute to the project's final cost. Models come in many different forms: as equations, mappings, or diagrams, for instance. These show us how the component parts relate to one another, so that we can examine and understand these relationships and make judgments about them.

In this chapter, we have seen that the representation condition requires every measure to be associated with a model of how the measure maps the entities and attributes in the real world to the elements of a numerical system. These models are essential in understanding not only how the measure is derived, but also how to interpret the behavior of the numerical elements when we return to the real world. But we also need models even before we begin the measurement process.

Let us consider more carefully the role of models in measurement definition. Previous examples have made clear that if we are measuring height of people, then we must understand and declare our assumptions to ensure unambiguous measurement. For example, in measuring height, we would have to specify whether or not we allow shoes to be worn, whether or not we include hair height, and whether or not we specify a certain posture. In this sense, we are actually defining a *model* of a person, rather than the person itself, as the entity being measured. Thus, the model of the mapping should also be supplemented with a model of the mapping's domain—that is, with a model of how the entity relates to its attributes.

EXAMPLE 2.7

To measure the length of programs using LOC, we need a model of a program. The model would specify how a program differs from a subroutine, whether or not to treat separate statements on the same line as distinct LOC, whether or not to count comment lines, whether or not to count data declarations, etc. The model would also tell us what to do when we have programs written in different languages. It might also distinguish delivered operational programs from those under development, and it would tell us how to handle situations where different versions run on different platforms.

Process measures are often more difficult to define than product and resource measures, in large part because the process activities are less understood.

EXAMPLE 2.8

Suppose we want to measure the attributes of the testing process. Depending on our goals, we might measure the time or effort spent on this process, or the number of faults found during the process. To do this, we need a careful definition of what is meant by the testing process; at the very least, we must be able to identify unambiguously when the process starts and ends. A model of the testing process can show us which activities are included, when they start and stop, and what inputs and outputs are involved.

2.2.1 Defining Attributes

When measuring, there is always a danger that we focus too much on the formal, mathematical system, and not enough on the empirical one. We rush to create mappings and then manipulate numbers, without given careful thought to the relationships among entities and their attributes in the real world. Figure 2.8 presents a whimsical view of what can happen when we rush to manipulate numbers without considering their real meaning.

The dog in Figure 2.8 is clearly an exceptionally intelligent dog, but its intelligence is not reflected by the result of an IQ test. It is clearly wrong to *define* the intelligence of dogs in this way. Many people have argued that defining the intelligence of people by using IQ tests is just as problematic. What is needed is a comprehensive set of characteristics of intelligence, appropriate to the entity (so that dog intelligence will have a different set of characteristics from people intelligence) and associated by a model.

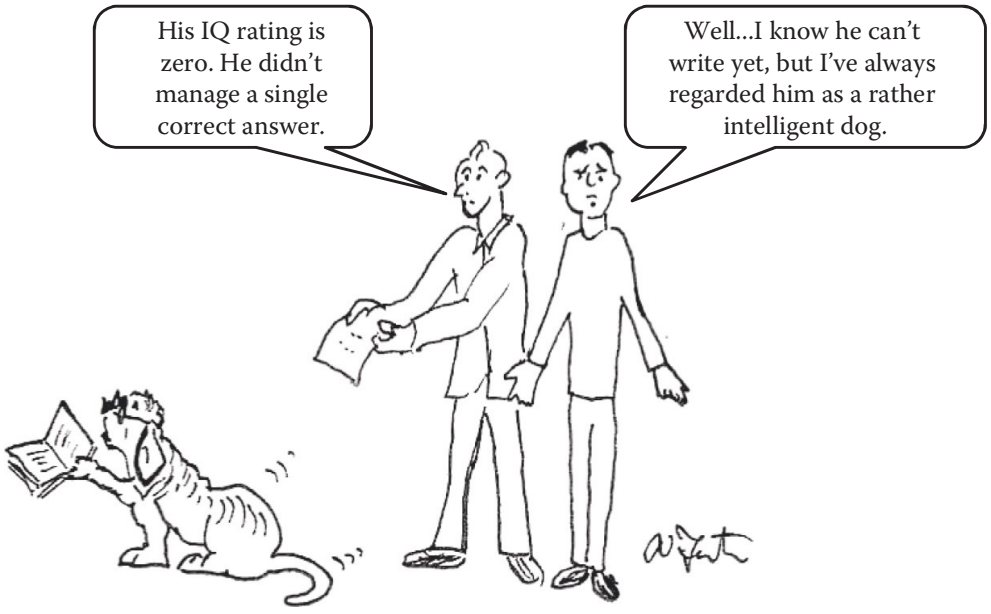

FIGURE 2.8 Using a suspect definition.

The model will show us how the characteristics relate. Then, we can try to define a measure for each characteristic, and use the representation condition to help us understand the relationships as well as overall intelligence.

EXAMPLE 2.9

In software development, our intuition tells us that the complexity of a program can affect the time it takes to code it, test it, and fix it; indeed, we suspect that complexity can help us to understand when a module is prone to contain faults. But there are few researchers who have built models of exactly what it means for a module to be complex. Instead, we often assume that we know what complexity is, and we measure complexity without first defining it in the real world. For example, many software developers still define program complexity as the cyclomatic number proposed by McCabe and illustrated in Figure 2.9 (McCabe 1976). This number, based on a graph-theoretic concept, counts the number of linearly independent paths through a program. We will discuss this measure (and its use in testing) in more detail in Chapter 9.

McCabe felt that the number of such paths was a key indicator not just of testability but also of complexity. Hence, he originally called this number, v, the cyclomatic complexity of a program. On the basis of empirical research, McCabe claimed that modules with high values of v were those most likely to be fault-prone and unmaintainable. He proposed a threshold value of 10 for each module; that is, any module with v greater than 10 should be redesigned to reduce v. However, the cyclomatic number presents only a partial view of complexity. It can be shown mathematically that the cyclomatic number is equal to one more than the number of decisions in a program, and there are many programs that have a large number of decisions but are easy to understand, code, and maintain. Thus, relying only on the cyclomatic number to measure actual program complexity can be misleading. A more complete model of program complexity is needed.

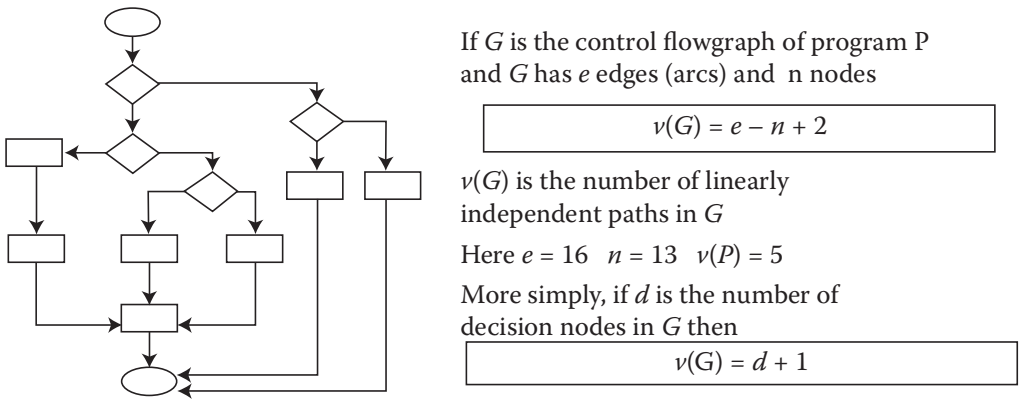

FIGURE 2.9 Computing McCabe's cyclomatic number.

Directed graphs are probably the most commonly used abstraction for modeling software designs and implementations. To develop a measure of software design attributes, we should establish relations relevant to design attributes in terms of graph models of designs. Then we can use the relations to derive and validate a measure of the attributes.

EXAMPLE 2.10

To evaluate existing software design measures, Briand and his colleagues developed relations on directed graph models of software designs, described as properties, relevant to a set of attributes including module size, module coupling, and system complexity (Briand et al. 1996).

One property of any module size measure is *module additivity*—the size of a system is the sum of the sizes of its disjoint modules. One property of any module coupling measure is that if you merge modules *m1* and *m2* to create module *M*, then $Coupling(M) \leq (Coupling(m1) + Coupling(m2))$. The coupling of *M* may be less than $Coupling(m1) + Coupling(m2)$ because *m1* and *m2* may have common intermodule relationships. Complexity properties are defined in terms of systems of modules, where complexity is defined in terms of the number of relationships between elements in a system. One complexity property is that the complexity of a system consisting of disjoint modules is the sum of the complexity of its modules.

We can use the set of properties for a software attribute as an empirical relation system to evaluate whether measures that are purported to be size, coupling, or complexity measures are really consistent with the properties. That is, we can determine if the measure satisfies the representation condition of measurement.

2.2.2 Direct and Derived Measurement

Once we have a model of the entities and attributes involved, we can define the measure in terms of them. Many of the examples we have used employ direct mappings from attribute to number, and we use the number to answer questions or assess situations. But when there are complex relationships among attributes, or when an attribute must be measured by combining several of its aspects, then we need a model of how to combine the related measures. It is for this reason that we distinguish direct measurement from derived measurement.

Direct measurement of an attribute of an entity involves no other attribute or entity. For example, *length* of a physical object can be measured without reference to any other object or attribute. On the other hand, measures of the *density* of a physical object can be derived in terms of

mass and *volume*; we then use a model to show us that the relationship between the three is

$$Density = Mass/Volume$$

Similarly, the speed of a moving object is most accurately measured using direct measures of distance and time. Thus, direct measurement forms the building blocks for our assessment, but many interesting attributes can be measured only by derived measurement.

The following direct measures are commonly used in software engineering:

- *Size* of source code (measured by LOC)
- *Schedule* of the testing process (measured by elapsed time in hours)
- *Number of defects discovered* (measured by counting defects)
- *Time* a programmer spends on a project (measured by months worked)

Table 2.4 provides examples of some derived measures that are commonly used in software engineering. The most common of all, and the most controversial, is the measure for programmer productivity, as it emphasizes size of output without taking into consideration the code's functionality or complexity. The defect detection efficiency measure is computed with respect to a specific testing or review phase; the total number of defects refers to the total number discovered during the entire product life cycle. Japanese software developers routinely compute the system spoilage measure; it indicates how much effort is wasted in fixing faults, rather than in building new code.

TABLE 2.4 Examples of Common Derived Measures Used in Software Engineering

Programmer productivity	LOC produced/person-months of effort
Module defect density	Number of defects/module size
Defect detection efficiency	Number of defects detected/total number of defects
Requirements stability	Number of initial requirements/total number of requirements
Test coverage	Number of test requirements covered/total number of test requirements
System spoilage	Effort spent fixing faults/total project effort

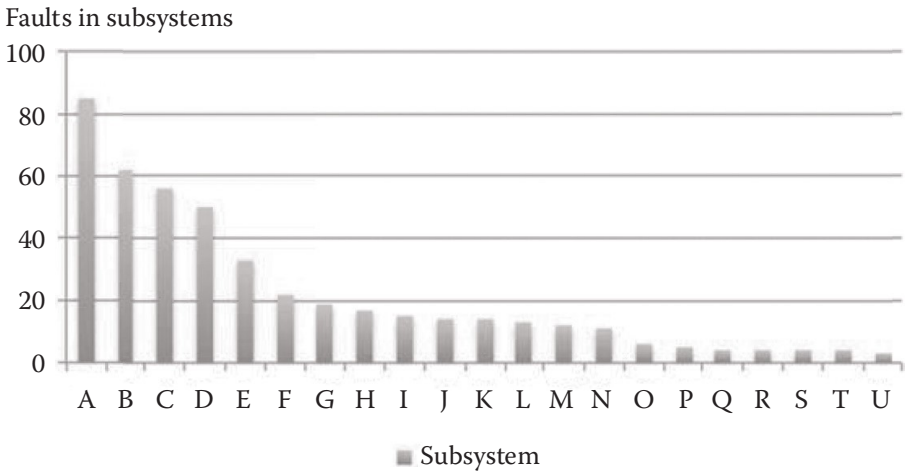

FIGURE 2.10 Using direct measurement to assess a product (from a major system made up of several subsystems).

Derived measurement is often useful in making visible the interactions between direct measurements. That is, it is sometimes easier to see what is happening on a project by using combinations of measures. To see why, consider the graph in Figure 2.10.

The graph shows the (anonymized) number of faults in each subsystem of a large, important software system in the United Kingdom. From the graph, it appears as if there are five subsystems that contain the most problems for the developers maintaining this system. However, Figure 2.11 depicts the same data with one big difference: instead of using the direct measurement of faults, it shows fault density (i.e., the derived measure defined as faults per KLOC). From the derived measurement, it is very clear that one subsystem is responsible for the majority of the problems.

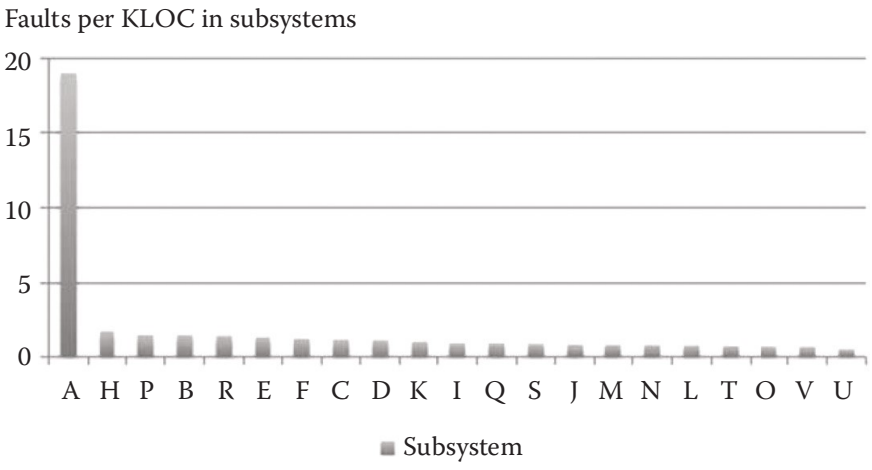

FIGURE 2.11 Using derived measurement to assess a product.

In fact, subsystem A is only 4000 LOC out of two million, but A is a big headache for the maintainers. Here, the derived measurement helps the project team to focus their maintenance efforts more effectively.

The representational theory of measurement, as described in this chapter, is initially concerned with direct measurement of attributes. Where no previous measurement has been performed, direct measurement constitutes the natural process of trying to understand entities and the attributes they possess. However, simple models of direct measurement do not preclude the possibility of more accurate subsequent measurement that will be achieved indirectly. For example, temperature can be measured as the length of a column of mercury under given pressure conditions. This measure is derived because we are examining the column, rather than the entity whose temperature we want to know.

2.2.3 Measurement for Prediction

When we talk about measuring something, we usually mean that we wish to assess some entity that already exists. This measurement for *assessment* is very helpful in understanding what exists now or what has happened in the past. However, in many circumstances, we would like to *predict* an attribute of some entity that does not yet exist. For example, suppose we are building a software system that must be highly reliable, such as the control software for an aircraft, power plant, or x-ray machine. The software construction may take some time, and we want to provide early assurance that the system will meet reliability targets. However, reliability is defined in terms of operational performance, something we clearly cannot measure before the product is finished. To provide reliability indicators *before* the system is complete, we can build a model of the factors that affect reliability, and then predict the likely reliability based on our understanding of the system while it is still under development.

Similarly, we often need to predict how much a development project will cost, or how much time and effort will be needed, so that we can allocate the appropriate resources to the project. Simply waiting for the project to end, and then measuring cost and schedule attributes are clearly not acceptable.

The distinction between measurement for assessment and prediction is not always clear-cut. For example, suppose we use a globe to determine the distance between London and Washington, DC. This derived measurement helps us to assess how far apart the cities are. However, the same activity is also involved when we want to predict the distance we will travel

on a future journey. Notice that the action we take in assessing distance involves the globe as a model of the real world, plus prediction procedures that describe how to use the model.

In general, measurement for prediction always requires some kind of *mathematical model* that relates the attributes to be predicted to some other attributes that we can measure now. The model need not be complex to be useful.

EXAMPLE 2.11

Suppose we want to predict the number of pages, *m,* that will print out as a source code program, so that we can order sufficient paper or estimate the time it will take to do the printing. We could use the very simple model

$$m = x/a$$

where x is a variable representing a measure of source-code program length in LOC, and a is a constant representing the average number of lines per page.

Project managers universally need effort prediction.

EXAMPLE 2.12

A common generic model for predicting the effort required in software projects has the form

$$E = aS^b$$

where E is effort in person-months, S is the size (in LOC) of the system to be constructed, and a and b are constants. We will examine many of these models in Chapter 11.

Sometimes, the same model is used both for assessment and prediction, as we saw with the example of the globe, above. The extent to which it applies to each situation depends on how much is known about the parameters of the model. In Example 2.11, suppose a is known to be 55 in a specific environment. If a program exists with a known x, then the derived

measure of hard copy pages computed by the given formula is not really a prediction problem (except in a very weak sense), particularly if the hard copy already exists. However, if we have only a program specification, and we wish to know roughly how many hard copy pages the final implementation will involve, then we *would* be using the model to solve a prediction problem. In this case, we need some kind of procedure for determining the unknown value of x based on our knowledge of the program specification. The same is true in Example 2.12, where invariably we need some means of determining the parameters a, b, and S based on our knowledge of the project to be developed.

These examples illustrate that the model alone is not enough to perform the required prediction. In addition, we need some means of determining the model parameters, plus a procedure to interpret the results. Therefore, we must think in terms of a *prediction system*, rather than of the model itself. A *prediction system* consists of a mathematical model together with a set of prediction procedures for determining unknown parameters and interpreting results (Littlewood 1988).

EXAMPLE 2.13

Suppose we want to predict the cost of an automobile journey from London to Bristol. The entity we want to predict is the journey and the attribute is its cost. We begin by obtaining measures (in the assessment sense) of:

- a: the distance between London and Bristol
- b: the cost per gallon of fuel
- c: the average distance we can travel on a gallon of fuel in our car

Next, we can predict the journey's cost using the formula

$$cost = ab/c$$

In fact, we are using a *prediction system* that involves:

1. A *model*: that is, the formula $cost = ab/c$.
2. A *set of procedures for determining the model parameters*: that is, how we determine the values of a, b, and c. For example, we may consult with the local automobile association, or simply ask a friend.
3. *Procedures for interpreting the results:* for example, we may use Bayesian probability to determine likely margins of error.

Using the same model will generally yield different results, if we use different prediction procedures. For instance, in Example 2.13, the model parameters supplied by a friend may be very different from those supplied by the automobile association. This notion of changing results is especially important when predicting software reliability.

EXAMPLE 2.14

A well-known reliability model is based on an exponential distribution for the time to the *i*th failure of the product. This distribution is described by the formula

$$F(t) = 1 - e^{-(N-i+1)at}$$

Here, N represents the number of faults initially residing in the program, while a represents the overall rate of occurrence of failures. There are many ways that the model parameters N and a can be estimated, including sophisticated techniques such as maximum likelihood estimation. The details of these prediction systems will be discussed in Chapter 11.

Accurate predictive measurement is always based on measurement in the assessment sense, so the need for assessment is especially critical in software engineering. Everyone wants to be able to predict key determinants of success, such as the effort needed to build a new system, or the reliability of the system in operation. However, there are no magic models. The models are dependent on high-quality measurements of past projects (as well as the current project during development and testing) if they are to support accurate predictions. Since software development is more a creative process than a manufacturing one, there is a high degree of risk when we undertake to build a new system, especially if it is very different from systems we have developed in the past. Thus, software engineering involves risk, and there are some clear parallels with gambling.

> Testing your methods on a sample of past data gets to the heart of the scientific approach to gambling. Unfortunately this implies some preliminary spadework, and most people skimp on that bit, preferring to rely on blind faith instead. (Drapkin and Forsyth 1987)

We can replace "gambling" with *software prediction*, and then heed the warning. In addition, we must recognize that the quality of our predictions

is based on several other assumptions, including the notion that the future will be like the past, and that we understand how data are distributed. For instance, many reliability models specify a particular distribution, such as Gaussian or Poisson. If our new data does not behave like the distribution in the model, our prediction is not likely to be accurate.

2.3 MEASUREMENT SCALES AND SCALE TYPES

We have seen how direct measurement of an attribute assigns a representation or mapping M from an observed (empirical) relation system to some numerical relation system. The purpose of performing the mapping is to be able to manipulate data in the numerical system and use the results to draw conclusions about the attribute in the empirical system. We do this sort of analysis all the time. For example, we use a thermometer to measure air temperature, and then we conclude that it is hotter today than yesterday; the numbers tell us about the characteristic of the air.

But not all measurement mappings are the same. And the differences among the mappings can restrict the kind of analysis we can do. To understand these differences, we introduce the notion of a *measurement scale*, and then we use the scale to help us understand which analyses are appropriate.

We refer to our measurement mapping, M, together with the empirical and numerical relation systems, as a *measurement scale*. Where the relation systems (i.e., the domain and range) are obvious from the context, we sometimes refer to M alone as the scale. There are three important questions concerning representations and scales:

1. How do we determine when one numerical relation system is preferable to another?
2. How do we know if a particular empirical relation system has a representation in a given numerical relation system?
3. What do we do when we have several different possible representations (and hence many scales) in the same numerical relation system?

Our answer to the first question is pragmatic. Recall that the formal relational system to which the scale maps need not be numeric; it can be symbolic. However, symbol manipulation may be far more unwieldy than numerical manipulation. Thus, we try to use the real numbers wherever possible, since analyzing real numbers permits us to use techniques with which we are familiar.

The second question is known as the *representation problem*, and its answer is sought not just by software engineers but also by all scientists who are concerned with measurement. The representation problem is one of the basic problems of measurement theory; it has been solved for various types of relation systems characterized by certain types of axioms. Rather than addressing it in this book, we refer the readers to the classical literature on measurement theory.

Our primary concern in this chapter is with the third question. Called the *uniqueness problem*, this question addresses our ability to determine which representation is the most suitable for measuring an attribute of interest.

In general, there are many different representations for a given empirical relation system. We have seen that the more relations there are, the fewer are the representations. This notion of shrinking representations can be best understood by a formal characterization of scale types. In this section, we classify measurement scales as one of five major types:

1. Nominal
2. Ordinal
3. Interval
4. Ratio
5. Absolute

There are other scales that can be defined (such as a logarithmic scale), but we focus only on these five, as they illustrate the range of possibilities and the issues that must be considered when measurement is done.

One relational system is said to be *richer* than another if all relations in the second are contained in the first. Using this notion, the scale types listed above are shown in increasing level of richness. That is, the richer the empirical relation system, the more restrictive the set of representations, and so the more sophisticated the scale of measurement.

The idea behind the formal definition of scale types is quite simple. If we have a satisfactory measure for an attribute with respect to an empirical relation system (i.e., it captures the empirical relations in which we are interested), we want to know what other measures exist that are also acceptable. For example, we may measure the length of physical objects by using a mapping from length to inches. But there are equally acceptable measures

in feet, meters, furlongs, miles, and more. In this example, all of the acceptable measures are very closely related, in that we can map one into another by multiplying by a suitable positive constant (such as converting inches into feet by multiplying by 1/12). A mapping from one acceptable measure to another is called an *admissible transformation*. When measuring length, the class of admissible transformations is very restrictive, in the sense that all admissible transformations are of the form

$$M' = aM$$

where M is the original measure, M' is the new one, and a is a constant.

In particular, transformations of the form

$$M' = b + aM \quad (b \neq 0)$$

or

$$M' = aM^b \quad (b \neq 1)$$

are not acceptable. Thus, the set of admissible transformations for length is smaller than the set of all possible transformations. We say that the more restrictive the class of admissible transformations, the more *sophisticated* the measurement scale.

2.3.1 Nominal Scale Type

Suppose we define classes or categories, and then place each entity in a particular class or category, based on the value of the attribute. This categorization is the basis for the most primitive form of measurement, the *nominal scale*. Thus, the nominal scale has two major characteristics:

1. The empirical relation system consists only of different classes; there is no notion of ordering among the classes.
2. Any distinct numbering or symbolic representation of the classes is an acceptable measure, but there is no notion of magnitude associated with the numbers or symbols.

In other words, nominal scale measurement places elements in a classification scheme. The classes are not ordered; even if the classes are numbered from 1 to n for identification, there is no implied ordering of the classes.

EXAMPLE 2.15

Suppose that we are investigating the set of all known software faults in our code, and we are trying to capture the *location* of the faults. Then we seek a measurement scale with faults as entities and location as the attribute. We can use a common but primitive mapping to identify the fault location: we denote a fault as *specification*, *design*, or *code*, according to where the fault was first introduced. Notice that this classification imposes no judgment about which class of faults is more severe or important than another. However, we have a clear distinction among the classes, and every fault belongs to exactly one class. This is a very simple empirical relation system. Any mapping, M, that assigns the three different classes to three different numbers satisfies the representation condition and is therefore an acceptable measure. For example, the mappings M_1 and M_2 defined by

$$M_1(x) = \begin{cases} 1, & \text{if } x \text{ is specification fault} \\ 2, & \text{if } x \text{ is design fault} \\ 3, & \text{if } x \text{ is code fault} \end{cases}$$

$$M_2(x) = \begin{cases} 101, & \text{if } x \text{ is specification fault} \\ 2.73, & \text{if } x \text{ is design fault} \\ 69, & \text{if } x \text{ is code fault} \end{cases}$$

are acceptable. In fact, any two mappings, M and M', will always be related in a special way: M' can be obtained from M by a one-to-one mapping. The mappings need not involve numbers; distinct symbols will suffice. Thus, the class of admissible transformations for a nominal scale measure is the set of all one-to-one mappings.

2.3.2 Ordinal Scale Type

We can often augment the nominal scale with information about an ordering of the classes or categories creating an *ordinal scale*. The ordering leads to analysis not possible with nominal measures. The ordinal scale has the following characteristics:

- The empirical relation system consists of classes that are ordered with respect to the attribute.
- Any mapping that preserves the ordering (i.e., any monotonic function) is acceptable.
- The numbers represent ranking only, so addition, subtraction, and other arithmetic operations have no meaning.

However, classes can be combined, as long as the combination makes sense with respect to the ordering.

EXAMPLE 2.16

Suppose our set of entities is a set of software modules, and the attribute we wish to capture quantitatively is *complexity*. Initially, we may define five distinct classes of module complexity: *trivial*, *simple*, *moderate*, *complex*, and *incomprehensible*. There is an implicit order relation of *less complex than* on these classes; that is, all trivial modules are less complex than simple modules, which are less complex than moderate modules, etc. In this case, since the measurement mapping must preserve this ordering, we cannot be as free in our choice of mapping as we could with a nominal measure. Any mapping, M, must map each distinct class to a different number, as with nominal measures. But we must also ensure that the more complex classes are mapped to bigger numbers. Therefore, M must be a monotonically increasing function. For example, each of the mappings M_1, M_2, and M_3 is a valid measure, since each satisfies the representation condition.

$$M_1(x) = \begin{cases} 1 \text{ if } x \text{ is trivial} \\ 2 \text{ if } x \text{ is simple} \\ 3 \text{ if } x \text{ is moderate} \\ 4 \text{ if } x \text{ is complex} \\ 5 \text{ if } x \text{ is incomprehensible} \end{cases} \qquad M_2(x) = \begin{cases} 1 \text{ if } x \text{ is trivial} \\ 2 \text{ if } x \text{ is simple} \\ 3 \text{ if } x \text{ is moderate} \\ 4 \text{ if } x \text{ is complex} \\ 10 \text{ if } x \text{ is incomprehensible} \end{cases}$$

$$M_3(x) = \begin{cases} 0.1 \text{ if } x \text{ is trivial} \\ 1001 \text{ if } x \text{ is simple} \\ 1002 \text{ if } x \text{ is moderate} \\ 4570 \text{ if } x \text{ is complex} \\ 4573 \text{ if } x \text{ is incomprehensible} \end{cases}$$

However, neither M_4 nor M_5 is valid:

$$M_4(x) = \begin{cases} 1 \text{ if } x \text{ is trivial} \\ 1 \text{ if } x \text{ is simple} \\ 3 \text{ if } x \text{ is moderate} \\ 4 \text{ if } x \text{ is complex} \\ 5 \text{ if } x \text{ is incomprehensible} \end{cases} \qquad M_5(x) = \begin{cases} 1 \text{ if } x \text{ is trivial} \\ 3 \text{ if } x \text{ is simple} \\ 2 \text{ if } x \text{ is moderate} \\ 4 \text{ if } x \text{ is complex} \\ 10 \text{ if } x \text{ is incomprehensible} \end{cases}$$

Since the mapping for an ordinal scale preserves the ordering of the classes, the set of ordered classes $<C_1, C_2, \ldots, C_n>$ is mapped to an

increasing series of numbers $< a_1, a_2, \ldots, a_n >$ where a_i is greater than a_j when i is greater than j. Any acceptable mapping can be transformed to any other as long as the series of a_i is mapped to another increasing series. Thus, in the ordinal scale, any two measures can be related by a monotonic mapping, so the class of admissible transformations is the set of all monotonic mappings.

2.3.3 Interval Scale Type

We have seen how the ordinal scale carries more information about the entities than does the nominal scale, since ordinal scales preserve ordering. The interval scale carries more information still, making it more powerful than nominal or ordinal. This scale captures information about the size of the intervals that separate the classes, so that we can in some sense understand the size of the jump from one class to another. Thus, an interval scale can be characterized in the following way:

- An interval scale preserves order, as with an ordinal scale.
- An interval scale preserves differences but not ratios. That is, we know the difference between any two of the ordered classes in the range of the mapping, but computing the ratio of two classes in the range does not make sense.
- Addition and subtraction are acceptable on the interval scale, but not multiplication and division.

To understand the difference between ordinal and interval measures, consider first an example from everyday life.

EXAMPLE 2.17

We can measure air temperature on a Fahrenheit or Celsius scale. Thus, we may say that it is usually 20° Celsius on a summer's day in London, while it may be 30° Celsius on the same day in Washington, DC. The interval from one degree to another is the same, and we consider each degree to be a class related to heat. That is, moving from 20° to 21° in London increases the heat in the same way that moving from 30° to 31° does in Washington. However, we cannot say that it is two-third as hot in London as Washington; neither can we say that it is 50% hotter in Washington than in London. Similarly, we cannot say that a 90° Fahrenheit day in Washington is twice as hot as a 45° Fahrenheit day in London.

There are fewer examples of interval scales in software engineering than of nominal or ordinal.

EXAMPLE 2.18

Recall the five categories of complexity described in Example 2.16. Suppose that the difference in complexity between a trivial and simple system is the same as that between a simple and moderate system. Then any interval measure of complexity must preserve these differences. Where this equal step applies to each class, we have an attribute measurable on an interval scale. The following measures have this property and satisfy the representation condition:

$$M_1(x) = \begin{cases} 1 \text{ if } x \text{ is trivial} \\ 2 \text{ if } x \text{ is simple} \\ 3 \text{ if } x \text{ is moderate} \\ 4 \text{ if } x \text{ is complex} \\ 5 \text{ if } x \text{ is incomprehensible} \end{cases} \qquad M_2(x) = \begin{cases} 0 \text{ if } x \text{ is trivial} \\ 2 \text{ if } x \text{ is simple} \\ 4 \text{ if } x \text{ is moderate} \\ 6 \text{ if } x \text{ is complex} \\ 8 \text{ if } x \text{ is incomprehensible} \end{cases}$$

$$M_3(x) = \begin{cases} 3.1 \text{ if } x \text{ is trivial} \\ 5.1 \text{ if } x \text{ is simple} \\ 7.1 \text{ if } x \text{ is moderate} \\ 9.1 \text{ if } x \text{ is complex} \\ 11.1 \text{ if } x \text{ is incomprehensible} \end{cases}$$

Suppose an attribute is measurable on an interval scale, and M and M' are mappings that satisfy the representation condition. Then we can always find numbers a and b such that

$$M = aM' + b$$

We call this type of transformation an *affine transformation*. Thus, the class of admissible transformations of an interval scale is the set of affine transformations. In Example 2.17, we can transform Celsius to Fahrenheit by using the transformation

$$F = 9/5C + 32$$

Likewise, in Example 2.18, we can transform M_1 to M_3 by using the formula

$$M_3 = 2M_1 + 1.1$$

EXAMPLE 2.19

The timing of an event's occurrence is a classic use of interval scale measurement. We can measure the timing in units of years, days, hours, or some other standard measure, where each time is noted relative to a given fixed event. We use this convention everyday by measuring the year with respect to an event (i.e., by saying "2014 AD"), or by measuring the hour from midnight. Software development projects can be measured in the same way, by referring to the project's start day. We say that we are on day 87 of the project, when we mean that we are measuring 87 days from the first day of the project. Thus, using these conventions, it is meaningless to say "Project X started twice as early as project Y" but meaningful to say "the time between project X's beginning and now is twice the time between project Y's beginning and now."

On a given project, suppose the project manager is measuring time in months from the day work started: April 1, 2013. But the contract manager is measuring time in years from the day that the funds were received from the customer: January 1, 2014. If M is the project manager's scale and M' the contract manager's scale, we can transform the contract manager's time into the project manager's by using the following admissible transformation:

$$M = 12M' + 9$$

2.3.4 Ratio Scale Type

Although the interval scale gives us more information and allows more analysis than either nominal or ordinal, we sometimes need to be able to do even more. For example, we would like to be able to say that one liquid is twice as hot as another, or that one project took twice as long as another. This need for ratios gives rise to the ratio scale, the most useful scale of measurement, and one that is common in the physical sciences. A *ratio scale* has the following characteristics:

- It is a measurement mapping that preserves ordering, the size of intervals between entities, and ratios between entities.
- There is a zero element, representing total lack of the attribute.
- The measurement mapping must start at zero and increase at equal intervals, known as units.
- All arithmetic can be meaningfully applied to the classes in the range of the mapping.

The key feature that distinguishes ratio from nominal, ordinal, and interval scales is the existence of empirical relations to capture ratios.

EXAMPLE 2.20

The length of physical objects is measurable on a ratio scale, enabling us to make statements about how one entity is twice as long as another. The zero element is theoretical, in the sense that we can think of an object as having no length at all; thus, the zero-length object exists as a limit of things that get smaller and smaller. We can measure length in inches, feet, centimeters, meters, and more, where each different measure preserves the relations about length that we observe in the real world. To convert from one length measure into another, we can use a transformation of the form $M = aM'$, where a is a constant. Thus, to convert feet into inches, we use the transformation $I = 12F$.

In general, any acceptable transformation for a ratio scale is a mapping of the form

$$M = aM'$$

where a is a positive scalar. This type of transformation is called a *ratio transformation*.

EXAMPLE 2.21

The length of software code is also measurable on a ratio scale. As with other physical objects, we have empirical relations like *twice as long*. The notion of a zero-length object exists—an empty piece of code. We can measure program length in a variety of ways, including LOC, thousands of LOC, the number of characters contained in the program, the number of executable statements, and more. Suppose M is the measure of program length in LOC, while M' captures length as number of characters. Then we can transform one to the other by computing $M' = aM$, where a is the average number of characters per line of code.

2.3.5 Absolute Scale Type

As the scales of measurement carry more information, the defining classes of admissible transformations have become increasingly restrictive. The absolute scale is the most restrictive of all. For any two measures, M and M',

there is only one admissible transformation: the identity transformation. That is, there is only one way in which the measurement can be made, so M and M' must be equal. The *absolute scale* has the following properties:

- The measurement for an absolute scale is made simply by counting the number of elements in the entity set.
- The attribute always takes the form "number of occurrences of x in the entity."
- There is only one possible measurement mapping, namely the actual count, and there is only one way to count elements.
- All arithmetic analysis of the resulting count is meaningful.

There are many examples of absolute scale in software engineering. For instance, the number of failures observed during integration testing can be measured only in one way: by counting the number of failures observed. Hence, a count of the number of failures is an absolute scale measure for the number of failures observed during integration testing. Likewise, the number of people working on a software project can be measured only in one way: by counting the number of people.

Since there is only one possible measure of an absolute attribute, the set of acceptable transformations for the absolute scale is simply the identity transformation. The uniqueness of the measure is an important difference between the ratio scale and absolute scale.

EXAMPLE 2.22

We saw in Example 2.21 that the number of LOC is a ratio scale measure of length for source code programs. A common mistake is to assume that LOC is an *absolute* scale measure of length, because it is obtained by counting. However, it is the *attribute* (as characterized by empirical relations) that determines the scale type. As we have seen, the length of programs cannot be absolute, because there are many different ways to measure it (such as LOC, thousands of LOC, number of characters, and number of bytes). It is incorrect to say that LOC is an absolute scale measure of program length. However, LOC is an absolute scale measure of the attribute "number of lines of code" of a program. For the same reason, "number of years" is a ratio scale measure of a person's age; it cannot be an absolute scale measure of age, because we can also measure age in months, hours, minutes, or seconds.
