17 March 2011

Chapter 2
The basics of measurement

In Chapter 1, we saw how measurement pervades our world. We use measurement every day, 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. 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 their are several kinds of geometry (for example, Euclidean and non-Euclidean), 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 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.

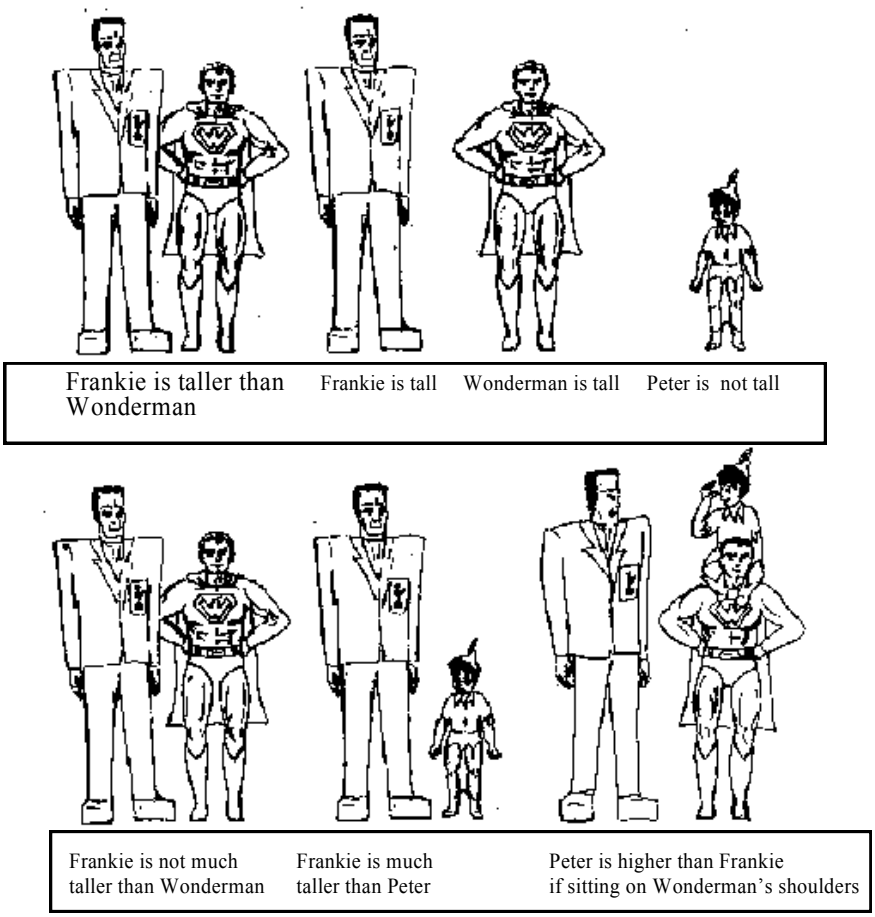

Figure 2.1: Some empirical relations for the attribute *height*

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

Table 2.1: Sampling 100 users to express preferences among products A, B, C and D

	A	*B*	*C*	*D*
A	—	80	10	80
B	20	—	5	50
C	90	95	—	96
D	20	50	4	—

More functionality

	A	*B*	*C*	*D*
A	—	45	50	44
B	55	—	52	50
C	50	48	—	51
D	54	50	49	—

More user-friendly

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 50%. Thus, the relation consists of the pairs (C,A), (C,B), (C,D), (A,B), (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.

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 which to capture this comparative relationship; the thermometer could consistently assign a higher number to liquids that were *hotter than* others.

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 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 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 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.

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

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 simply to 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 in a program. But the definition of a line of code must be made clear. The US Software Engineering Institute has developed a checklist to assist developers in deciding exactly what is included in a line of code. [Park 1992] 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 lines of code:

"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."

Statement type	*Include*	*Exclude*
Executable		
Nonexecutable		
Declarations		
Compiler directives		
Comments		
On their own lines		
On lines with source code		
Banners and non-blank 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

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 behavior 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.

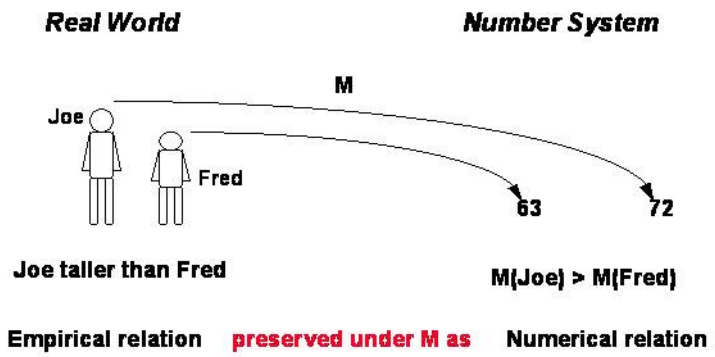

Figure 2.4: Representation condition

That is, 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

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

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: Earlier in this chapter, 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*,

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

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

$$x>70$$

The representation condition requires that for any measure *M*,

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

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

$$0.7M(A) + 0.8M(C) > M(B)$$

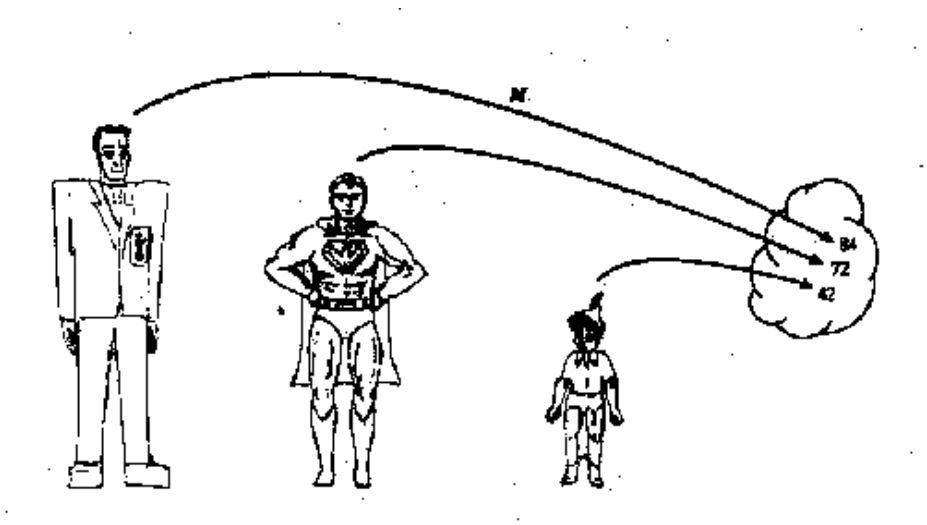

Figure 2.5: A measurement mapping

Consider the actual assignment of numbers M given in Figure 2.5. Wonderman is mapped to the real number 72 (that is, 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)

Because 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 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(Wonderman) = 72
M(Frankie) = 84
M(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.

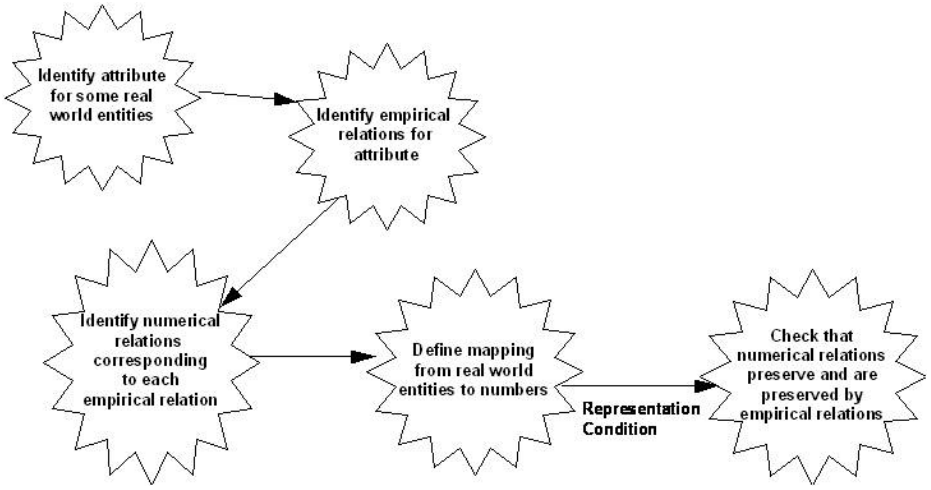

Figure 2.6: Key stages of formal measurement

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 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 only choose 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(each delayed response) = 6
M(each incorrect output) = 4
M(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*.

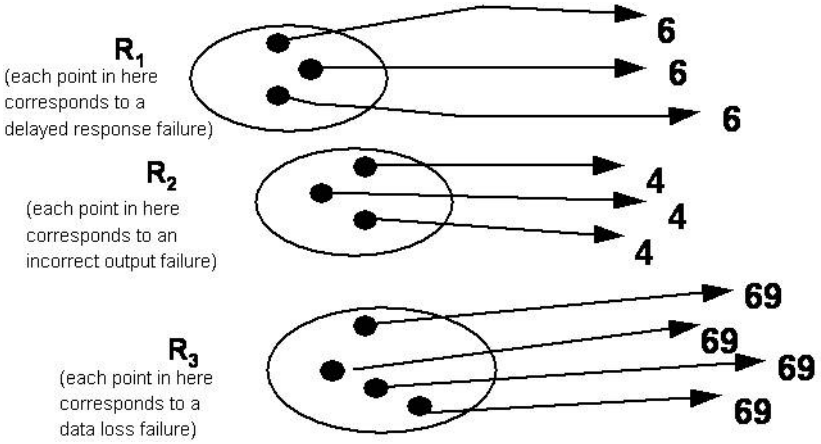

Figure 2.7: Measurement mapping

Suppose next that 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 is in R_3 and *y* is in R_2 or R_1, or
x is in R_2 and *y* 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 into 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(each delayed response) = 3
M(each incorrect output) = 4
M(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.

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	Tester	Efficiency	Number of faults found per KLOC (thousand LOC)
8	Program code	Quality	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

There is nothing wrong with using the same representation in different ways, or with 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 the table give different measures of program length, while examples 9 and 10 give different measures of program reliability. Similarly, the same measure (although 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 thousand lines of code 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 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 length of programs using lines of code, 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 lines of code, whether or not to count comment lines, whether or not to count data declarations, and so on. 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 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.

Figure 2.8: Using a suspect definition

The dog in the figure 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. 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 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.

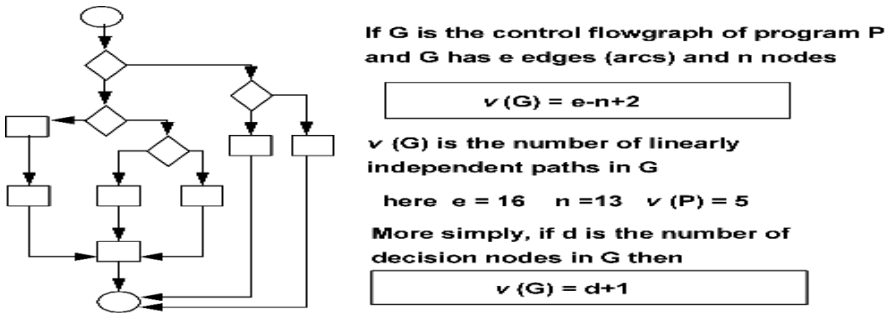

Figure 2.9: Computing McCabe's cyclomatic number

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.

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)* ≤ *(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 indirect 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 indirect.

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, *density* of a physical object can be measured only indirectly 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 indirect measurement.

The following direct measures are commonly used in software engineering:

- *Length* of source code (measured by lines of code)
- *Duration* of testing process (measured by elapsed time in hours)
- *Number of defects discovered* during the testing process (measured by counting defects)

- *Time* a programmer spends on a project (measured by months worked)

Table 2.4 provides examples of some indirect 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. The system spoilage measure is routinely computed by Japanese software developers; it indicates how much effort is wasted in fixing faults, rather than in building new code.

Table 2.4: Examples of common indirect measures used in software engineering

Programmer productivity	$\frac{\text{LOC produced}}{\text{person months of effort}}$
Module defect density	$\frac{\text{number of defects}}{\text{module size}}$
Defect detection efficiency	$\frac{\text{number of defects detected}}{\text{total number of defects}}$
Requirements stability	$\frac{\text{numb of initial requirements}}{\text{total number of requirements}}$
Test effectiveness ratio	$\frac{\text{number of items covered}}{\text{total number of items}}$
System spoilage	$\frac{\text{effort spent fixing faults}}{\text{total project effort}}$

Indirect 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.

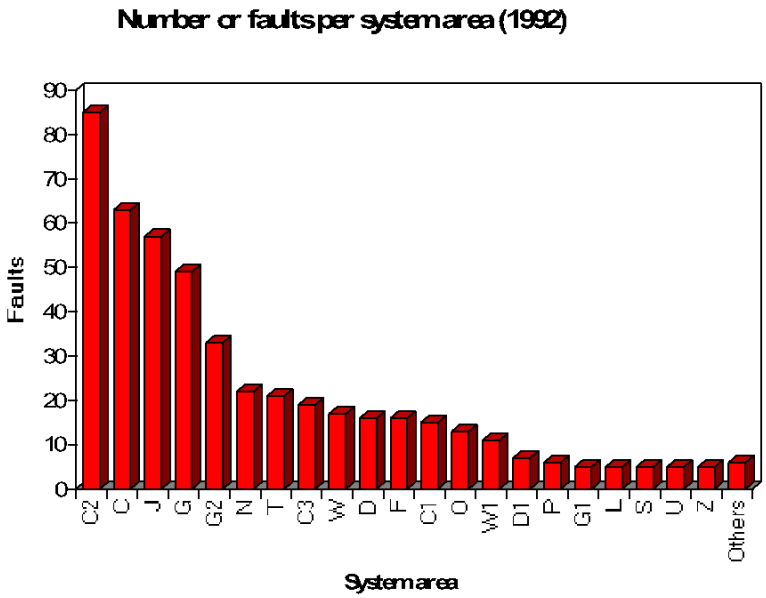

Figure 2.10: Using direct measurement to assess a product [Pfleeger, Fenton and Page 1994]

The graph shows the number of faults in each system area of a large, important software system in the United Kingdom. From the graph, it appears as if there are five system areas 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 (that is, the indirect measure defined as faults per thousand lines of code). From the indirect measurement, it is very clear that one system area is responsible for the majority of the problems. In fact, system area C2 is only 4000 lines of code out of two million, but it is a big headache for the maintainers. Here, indirect measurement helps the project team to focus their maintenance efforts more effectively.

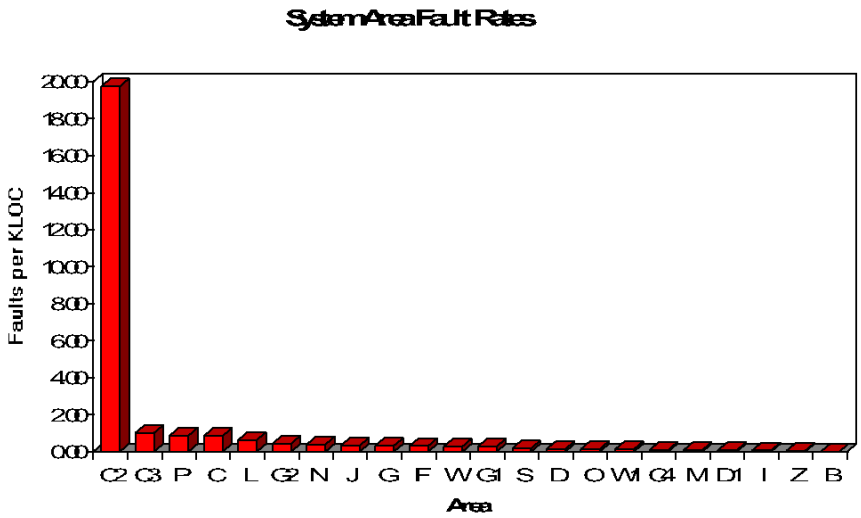

Figure 2.11: Using indirect measurement to assess a product [Pfleeger, Fenton and Page 1994]

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 indirect 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 indirect 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 lines of code, and a is a constant representing the average number of lines per page.

Effort prediction is universally needed by project managers.

> **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 lines of code) 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 indirect 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, and
- *c*: the average distance we can travel on a gallon of fuel in our car

Next, we can predict the journey's cost by 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 *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, 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.

Remember that a model defines an association between attributes. It is possible to have a useful prediction system based on models in which we have not determined the full functional form of the relationship:

Example 2.15: In a major study of software design measures at STC in 1989, Kitchenham and her colleagues confirmed an association between large values of certain program structure and size measurements with large values of measures of fault-proneness, change-proneness, and subjective complexity. [Kitchenham and Pickard 1990] As a result of their study, a special type of prediction system was constructed, having two components:

- A procedure for defining what is meant by a "large" value for each measure.
- A statistical technique for confirming that programs with large values of size and structure measures were more likely either to have a large number of faults, to have a large number of changes, or to be regarded as complex, compared with programs that did not have large values.

The STC researchers concluded that this procedure can be used to assist project managers in reducing potential project risks by identifying a group of programs or modules that are likely to benefit from additional scrutiny, such as inspection, re-design or additional testing.

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. Because 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 (that is, 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 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 address it in this book, we refer you 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 best be understood by a formal characterization of scale types. In this section, we classify measurement scales as one of five major types:

- nominal
- ordinal
- interval
- ratio
- 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 (that is, 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 to 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

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

or

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

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

Supose 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:

- The empirical relation system consists only of different classes; there is no notion of ordering among the classes.
- 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 2.16: 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

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 (that is, 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 2.17: 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, and so on. 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 also must 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.

$$M1(x) = \begin{cases} 1 \text{ if } x \text{ is trivial} \\ 2 \text{ if } x \text{ is simple} \\ 3 \text{ if } x \text{ is moderate} \\ 4 \text{ if } x \text{ is complex} \\ 5 \text{ if } x \text{ is incomprehensible} \end{cases} \quad M2(x) = \begin{cases} 1 \text{ if } x \text{ is trivial} \\ 2 \text{ if } x \text{ is simple} \\ 3 \text{ if } x \text{ is moderate} \\ 4 \text{ if } x \text{ is complex} \\ 10 \text{ if } x \text{ is incomprehensible} \end{cases} \quad M3(x) = \begin{cases} 0.1 \text{ if } x \text{ is trivial} \\ 1001 \text{ if } x \text{ is simple} \\ 1002 \text{ if } x \text{ is moderate} \\ 4570 \text{ if } x \text{ is complex} \\ 4573 \text{ if } x \text{ is incomprehensible} \end{cases}$$

However, neither M_4 nor M_5 is valid:

$$M4(x) = \begin{cases} 1 \text{ if } x \text{ is trivial} \\ 1 \text{ if } x \text{ is simple} \\ 3 \text{ if } x \text{ is moderate} \\ 4 \text{ if } x \text{ is complex} \\ 5 \text{ if } x \text{ is incomprehensible} \end{cases} \qquad M5(x) = \begin{cases} 1 \text{ if } x \text{ is trivial} \\ 3 \text{ if } x \text{ is simple} \\ 2 \text{ if } x \text{ is moderate} \\ 4 \text{ if } x \text{ is complex} \\ 10 \text{ if } x \text{ is incomprehensible} \end{cases}$$

Because 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

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 2.18: We can measure air temperature on a Fahrenheit or Celsius scale. Thus, we may say that it is usually 20 degrees Celsius on a summer's day in London, while it may be 30 degrees 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 degrees in London increases the heat in the same way that moving from 30 to 31 degrees does in Washington. However, we cannot say that it is two-thirds 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-degree Fahrenheit day in Washington is twice as hot as a 45-degree Fahrenheit day in London.

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

Example 2.19: Recall the five categories of complexity described in Example 2.17. 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:

$$M1(x) = \begin{cases} 1 \text{ if } x \text{ is trivial} \\ 2 \text{ if } x \text{ is simple} \\ 3 \text{ if } x \text{ is moderate} \\ 4 \text{ if } x \text{ is complex} \\ 5 \text{ if } x \text{ is incomprehensible} \end{cases} \quad M2(x) = \begin{cases} 0 \text{ if } x \text{ is trivial} \\ 2 \text{ if } x \text{ is simple} \\ 4 \text{ if } x \text{ is moderate} \\ 6 \text{ if } x \text{ is complex} \\ 8 \text{ if } x \text{ is incomprehensible} \end{cases} \quad M3(x) = \begin{cases} 3.1 \text{ if } x \text{ is trivial} \\ 5.1 \text{ if } x \text{ is simple} \\ 7.1 \text{ if } x \text{ is moderate} \\ 9.1 \text{ if } x \text{ is complex} \\ 11.1 \text{ if } x \text{ is incomprehensible} \end{cases}$$

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 2.18, we can transform Celsius to Fahrenheit by using the transformation

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

Likewise, in Example 2.19, we can transform M_1 to M_3 by using the formula

$$M_3 = 2M_1 + 1.1$$

Example 2.20: 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 every day by measuring the year with respect to an event (that is, by saying "1998 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, 1988. But the contract manager is measuring time in years from the day that the funds were received from the customer: January 1, 1989. 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:

$$M = 12\,M' + 9$$

2.3.4 Ratio scale

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 2.21: 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 to another, we can use a transformation of the form $M = aM'$ where a is a constant. Thus, to convert feet to 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**, as it preserves the ratios in *M'* as they are transformed to *M*.

Example 2.22: 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 lines of code, thousands of lines of code, the number of characters contained in the program, the number of executable statements, and more. Suppose *M* is the

measure of program length in lines of code, 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

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.
- 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.

Because 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 2.23: We saw in Example 2.22 that the number of lines of code (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.

Table 2.5 summarizes the key elements distinguishing the measurement scale types discussed in this chapter. This table is similar to those found in other texts on measurement. However, most texts do not point out the possible risk of mis-interpretation of the Examples column. Since scale types are defined with respect to the set of admissible transformations, we should never give examples of attributes without specifying the empirical relation system that characterizes an attribute. We have seen that as we enrich the relation system for an attribute by preserving more information with the measurement mapping, so we may arrive at a more restrictive (and hence different) scale type. Thus, when Table 2.5 says that the attributes length, time interval, and (absolute) temperature are on the ratio scale, what it really means is that we have developed sufficiently refined empirical relation systems to allow ratio scale measures for these attributes.

Table 2.5: Scales of measurement

Scale type	Admissible transformations (how measures M and M' must be related)	Examples
Nominal	1-1 mapping from M to M'	Labeling, classifying entities
Ordinal	Monotonic increasing function from M to M', that is, $M(x) \geq M(y)$ implies $M'(x) \geq M'(y)$	Preference, hardness, air quality, intelligence tests (raw scores)
Interval	$M' = aM+b$ $(a>0)$	Relative time, temperature (Fahrenheit, Celsius), intelligence tests (standardized scores)
Ratio	$M' = aM$ $(a>0)$	Time interval, length, temperature (Kelvin)
Absolute	$M' = M$	Counting entities

2.4. Meaningfulness in measurement

There is more than just academic interest in scale types. Understanding scale types enables us to determine when statements about measurement make sense. For instance, we have seen how it is inappropriate to compute ratios with nominal, ordinal and interval scales. In general, measures often map attributes to real numbers, and it is tempting to manipulate the real numbers in familiar ways: adding, averaging, taking logarithms, and performing sophisticated statistical analysis. But we must remember that the analysis is constrained by the scale type. We can perform only those calculations that are permissible for the given scale, reflecting the type of attribute and mapping that generated the data. In other words, knowledge of scale type tells us about limitations on the kind of mathematical manipulations that can be performed. Thus, the key question we should ask after having made our measurements is: can we deduce meaningful statements about the entities being measured?

This question is harder to answer than it first appears. To see why, consider the following statements:

1. The number of errors discovered during the integration testing of program *X* was at least 100.

2. The cost of fixing each error in program *X* is at least 100.
3. A semantic error takes twice as long to fix as a syntactic error.
4. A semantic error is twice as complex as a syntactic error.

Intuitively, statement 1 seems to make sense, but statement 2 does not; the number of errors may be specified without reference to a particular scale, but the cost of fixing an error cannot be. Statement 3 seems to make sense (even if we think it cannot possibly be true) because the ratio of time taken is the same, regardless of the scale of measurement used. (That is, if a semantic error takes twice as many minutes to repair as a syntactic error, it also takes twice as many hours, seconds or years to repair.) Statement 4 does not appear to be meaningful, and we require clarification. If "complexity" means time to understand, then the statement makes sense. But other definitions of complexity may not admit measurement on a ratio scale; in those instances, statement 4 is meaningless.

Our intuitive notion of a statement's meaningfulness involving measurement is quite distinct from the notion of the statement's *truth*. For example, the statement

The President of the United States is 125 years old.

is a meaningful statement about the age measure, even though it is clearly false. We can define "meaningfulness" in a formal way.

We say that a statement involving measurement is **meaningful** if its truth value is invariant of transformations of allowable scales.

Example 2.24: We can examine the transformations to decide on meaningfulness. Consider these statements:

Fred is twice as tall as Jane.
This statement implies that the measures are at least on the ratio scale, because it uses scalar multiplication as an admissible transformation. The statement is meaningful because no matter which measure of height we use (inches, feet, centimeters, etc.), the truth or falsity of the statement remains consistent. In other words, if the statement is true and if M and M' are different measures of height, then both the statements

$$M(\text{Fred}) = 2M(\text{Jane})$$

and

$$M'(\text{Fred}) = 2M'(\text{Jane})$$

are true. This consistency of truth is due to the relationship $M = aM'$ for some positive number a.

The temperature in Tokyo today is twice that in London.
This statement also implies ratio scale but is not meaningful, because we measure (air) temperature only on two scales, Fahrenheit and Celsius. Suppose that the temperature in Tokyo is 40C and in London 20C. Then on the Celsius scale, the statement is true. However, on the Fahrenheit scale, Tokyo is 104F while London is 68F.

The difference in temperature between Tokyo and London today is twice what it was yesterday.
This statement implies that the distance between two measures is meaningful, a condition that is part of the interval scale. The statement is meaningful, because Fahrenheit and Celsius are related by the affine transformation $F = 9/5C + 32$, ensuring that ratios of differences (as opposed to just ratios) are preserved. For example, suppose yesterday's temperatures on the Celsius scale were 35C in Tokyo and 25C (a difference of 10) in London, while today it is 40C in Tokyo and 20C in London (a difference of 20). If we transform these temperatures to the Fahrenheit scale, we find that yesterday's temperatures were 95F in Tokyo and 77F London (a difference of 18); today's are 104F in Tokyo and 68F in London (a difference of 36). Thus, the truth value of the statement is preserved with the transformation.

Failure x is twice as critical as failure y.
This statement is not meaningful, since we have only an ordinal scale for failure criticality. To see why, suppose we have four classes of failures, $class_i$, for i from 1 to 4. We can define two mappings, M and M', to be valid ordinal measures as follows:

Failure class	Mapping M	Mapping M'
$class_1$	1	3
$class_2$	3	4
$class_3$	6	5
$class_4$	7	10

Suppose y is in $class_2$ and x in $class_3$. Notice that $M(x) = 6$ and $M(y) = 3$ while $M'(x) = 5$ and $M'(y) = 4$. In this case, the statement is true under M but false under M'.

Meaningfulness is often clear when we are dealing with measures with which we are familiar. But sometimes we define new measures, and it is not as easy to tell if the statements about them are meaningful.

Example 2.25: Suppose we define a crude notion of *speed* of software programs, and we rank three programs A, B and C with respect to a single empirical binary relation *faster than*. Suppose further that the empirical relation is such that A is faster than B which is faster than C. This notion of program speed is measurable on an ordinal scale, and any mapping M in which $M(A)>M(B)>M(C)$ is an acceptable measure. Now consider the statement "Program A is faster than both programs B and C" where we mean that A is faster than B *and* A is faster than C. We can show that this statement is meaningful in the following way. Let M and M' be any two acceptable measures. Then we know that, for any pair of programs x and y, $M(x)>M(y)$ if and only if $M'(x)>M'(y)$. Using the scale M, the statement under scrutiny corresponds to

$$M(A) > M(B) \text{ and } M(A) > M(C)$$

which is true. But then

$$M'(A) > M'(B) \text{ and } M'(A) > M'(C)$$

is also true because of the relationship between M and M'.

By similar argument, we can show that the statement "Program B is faster than both programs A and C" is meaningful even though it is false.

However, consider the statement "Program A is more than twice as fast as Program *C."* This statement is not meaningful. To see why, define acceptable measures M and M' as follows:

$$M(A) = 3;\ M(B) = 2;\ M(C) = 1$$
$$M(A) = 3;\ M(B) = 2.5;\ M(C) = 2$$

Using scale M the statement is true, since $3 = M(A) > 2M(C) = 2$. However, using M' the statement is false. Although the statement seems meaningful given our understanding of speed, the sophistication of the notion *twice as fast* was not captured in our over-simplistic empirical relation system, and hence was not preserved by all measurement mappings.

The terminology often used in software engineering can be imprecise and misleading. Many software practitioners and researchers mistakenly think that to be meaningful, a measure must be useful, practical, worthwhile or easy to collect. These characteristics are not part of meaningfulness. Indeed, such issues are difficult to address for any measure, whether it occurs in software or in some other scientific discipline. For example, carbon-dating techniques for measuring the age of fossils may not be practical or easy to do, but the measures are certainly valid and meaningful! Thus, meaningfulness should be viewed as only one attribute of a measure.

2.4.1 Statistical operations on measures

The scale type of a measure affects the types of operations and statistical analyses that can be sensibly applied to the data. Many statistical analyses use arithmetic operators:

$$+, -, \div, \times$$

The analysis need not be sophisticated. At the very least, we would like to know something about how the whole data set is distributed. We use two basic measures to capture this information: measures of central tendency, and measures of dispersion. A **measure of central tendency**, usually called an average, tells us something about where the "middle" of the set is likely to be, while a **measure of dispersion** tells us how far the data points stray from the middle.

We have measured an attribute for 13 entities, and the resulting data points in ranked order are:

2, 2, 4, 5, 5, 8, 8, 10 ,11, 11, 11, 15, 16

The **mean** of this set of data (that is, the sum divided by the number of items) is *8.3*
The **median** (that is, the value of the middle-ranked item) is *8*
The **mode** (that is, the value of the most commonly occurring item) is *11*

Figure 2.12: Different ways to compute the average of a set of numbers

Figure 2.12 shows the computation of measures of central tendency for a given set of data. Measures of dispersion include the maximum and minimum values, as well as the variance and standard deviation; these measures give us some indication of how the data are clustered around a measure of central tendency.

But even these simple analytical techniques cannot be used universally. In particular, nominal and ordinal measures do not permit computation of mean, variance and standard deviation. That is, the notion of mean is not meaningful for nominal and ordinal measures.

Example 2.26: Suppose the data points $\{x_1, \ldots, x_n\}$ represent a measure of understandability for each module in system *X*, while $\{y_1 \ldots, y_m\}$ represent the understandability values for each module in system *Y*. We would like to know which of the two systems has the higher average understandability. The statement"The average of the x_is is greater than the average of the y_js" must be meaningful; that is, the statement's truth value should be invariant with respect to the particular measure used.

Suppose we assess every module's understandability according to the following classification: trivial, simple, moderate, complex, incomprehensible. In this way, our notion of understandability is representable on an ordinal scale. From this, we can define two valid measures of understandability, *M* and *M'*, as in Table 2.6.

Table 2.6: Measures of understandability

	trivial	simple	moderate	complex	incomprehensible
M	1	2	3	4	5
M'	1	2	3	4	10

Suppose that *X* consists of exactly five modules, and the understandability of each is rated as:

x_1	trivial
x_2	simple
x_3	simple
x_4	moderate
x_5	incomprehensible

while *Y*'s seven modules have understandability

y_1	simple
y_2	moderate
y_3	moderate
y_4	moderate
y_5	complex
y_6	complex
y_7	complex

Using *M*, the mean of the *X* values is 2.6, while the mean of the *Y* values is 3.1; thus, the "average" of the *Y* values is greater than the average of the *X* values. However, using *M′*, the mean of the *X* values is 3.6, while the mean of the *Y* values is 3.1. Since the definition of meaningfulness requires the relation to be preserved, then *mean* is not a meaningful measure of central tendency for ordinal scale data.

On the other hand, the *median* (that is, the middle-ranked item) is a meaningful measure of central tendency. Using both *M* and *M′*, the median of the *Y* values (in both cases 3) is greater than the median of the *X* values (in both cases 2). Similarly, if we define *M"* as a radically different measure according to Table 2.7, the median of the *Y* values, 69, is still greater than the median of the *X* values, 3.8.

Table 2.7: Different measure M''

	trivial	simple	moderate	complex	incomprehensible
M″	0.5	3.8	69	104	500

Example 2.26 confirms that the mean cannot be used as a measure of central tendency for ordinal scale data. However, the mean is acceptable for interval and ratio scale data. To see why, let $\{X_1, \ldots, X_n\}$ and $\{Y_1, \ldots, Y_n\}$ be two sets of entities for which some attribute can be measured on a ratio scale. We must show that the statement "The mean of the X_is is greater than the mean of Y_j*s"* is meaningful. To do so, let *M* and *M′* be two measures for the attribute in question. Then we want to show that the means preserve the relation. In mathematical terms, we must demonstrate that

$$\frac{1}{n}\sum_{i=1}^{n} M\left(x_i\right) > \frac{1}{m}\sum_{j=1}^{m} M\left(y_j\right) \text{ if and only if } \frac{1}{n}\sum_{i=1}^{n} M'\left(x_i\right) > \frac{1}{m}\sum_{j=1}^{m} M'\left(y_j\right)$$

The ratio scale gives us the extra information we need to show that the assertion is valid. Thanks to the relationship between acceptable transformations for a ratio scale, we know that $M = aM'$ for some $a>0$. When we substitute aM' for *M* in the above equation, we get a statement that is clearly valid.

The same investigation can be done for any statistical technique, using scale and transformation properties to verify that a certain analysis is valid for a given scale type. Table 2.8 presents a summary of the meaningful statistics for different scale types. The entries are inclusive reading downwards. That is, every meaningful statistic

of a nominal scale type is also meaningful for an ordinal scale type, every meaningful statistic of an ordinal scale type is also meaningful for an interval scale type, and so on. We will return to the appropriateness of analysis when we discuss experimental design and analysis in Chapter 4, and again when we investigate the analysis of software measurement data in Chapter 5.

Table 2.8: Summary of measurement scales and statistics relevant to each. [Siegel and Castellan 1988]

Scale type	*Defining relations*	*Examples of appropriate statistics*	*Appropriate statistical tests*
Nominal	Equivalence	Mode Frequency	Non-parametric
Ordinal	Equivalence Greater than	Median Percentile Spearman r_S Kendall τ Kendall W	Non-parametric
Interval	Equivalence Greater than Known ratio of any intervals	Mean Standard deviation Pearson product-moment correlation Multiple product-moment correlation	Non-parametric
Ratio	Equivalence Greater than Known ratio of any intervals Known ratio of any two scale values	Geometric mean Coefficient of variation	Non-parametric and parametric

2.4.2 Objective and subjective measures

When measuring attributes of entities, we strive to keep our measurements objective. By doing so, we make sure that different people produce the same measures, regardless of whether they are measuring product, process or resource. This consistency of measurement is very important. Subjective measures depend on the environment in which they are made. The measures can vary with the person measuring, and they reflect the judgment of the measurer. What one judge considers bad, another may consider good, and it may be difficult to reach consensus on attributes such as process, product or resource quality.

Nevertheless, it is important to recognize that subjective measurements can be useful, as long as we understand their imprecision. For example, suppose we want to measure the quality of requirements before we turn the specification over to the test team, who will then define test plans from them. Any of the techniques shown in Figure 2.2 would be acceptable. For example, we may ask the test team to read and rate each requirement on a scale from 1 to 5, where '1' means "I understand this requirement completely and can write a complete test script to determine if this requirement is

met," to '5:' "I do not understand this requirement and cannot begin to write a test script." Suppose the results of this assessment look like the chart in Table 2.9.

Table 2.9: Results of requirements assessment

Requirement type	*1 (good)*	*2*	*3*	*4*	*5 (bad)*
Performance requirements	12	7	2	1	0
Database requirements	16	12	2	0	0
Interface requirements	3	4	6	7	1
Other requirements	14	10	1	0	0

Even though the measurement is subjective, the measures show us that we may have problems with our interface requirements; perhaps the interface requirements should be reviewed and rewritten before proceeding to test plan generation or even to design. It is the general picture that is important, rather than the exactness of the individual measure, so the subjectivity, although a drawback, does not prevent us from gathering useful information about the entity. We will see other examples throughout this book where measurement is far from ideal but still paints a useful picture of what is going on in the project.

2.4.3 Measurement in extended number systems

In many situations we cannot measure an attribute directly. Instead, we must measure it in terms of the more easily-understood component attributes of which it is comprised. (We sometimes call these components **sub-attributes**.) For example, suppose that we wish to assess the *quality* of the different types of transport available for traveling from our home to another city. We may not know how to measure quality directly, but we know that quality involves at least two significant sub-attributes, journey time and cost per mile. Hence, we accumulate data in Table 2.10 to describe these attributes:

Table 2.10: Transportation attributes

Option	*Journey time (hours)*	*Cost per mile (dollars)*
Car	3	1.5
Train	5	2.0
Plane	3.5	3.5
Executive coach	7	4.0

Intuitively, given two transport types, *A* and *B*, we would rank *A* superior to *B* (that is, *A* is of higher quality than *B*) if

journey time (*A*) < journey time (*B*) **AND** cost per mile (*A*) < cost per mile (*B*)

Using this rule with the data collected for each journey type, we can depict the relationships among the candidates as shown in Figure 2.13. In the figure, an arrow from transport type *B* to transport type *A* indicates the superiority of *A* to *B*. Thus, Car is superior to both Train and Plane because, in each case, the journey time is shorter

and the cost per mile is less. The figure also shows us that Car, Train and Plane are all superior to Coach.

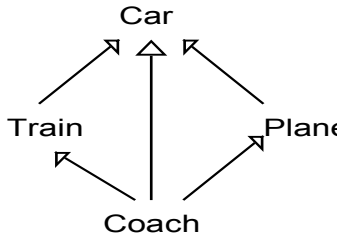

Figure 2.13: Quality relationships based on rule and collected data

Notice that in this relation Train and Plane are incomparable; that is, neither is superior to the other. Train is slower but cheaper than Plane. It would be inappropriate to force an ordering because of the different underlying attributes. We could impose an ordering only if we had additional information about the relative priorities of cost and timing. If cost is more important to us, then Train is preferable; if speed were more important, we would prefer Plane.

Now suppose we wish to use the representation condition to define a measure that characterizes the notion of journey quality given by the above relation system. It is easy to prove that there is *no* possible measure that is a single-valued real number. For suppose there were. Then Plane would be mapped to some real number m(Plane), while Train would be mapped to some real number m(Train). Exactly one of the following must then be true:

1. $m(\text{Plane}) < m(\text{Train})$
2. $m(\text{Plane}) > m(\text{Train})$
3. $m(\text{Plane}) = m(\text{Train})$

If the first statement were true, then the representation condition implies that Plane must be superior to Train. This is false, because Train is cheaper. Similarly, the second statement is false because Train is slower than Plane. But the third statement is also false, since it implies an equality relation that does not exist.

The reason we cannot find a measure satisfying the representation condition is because we are looking at too narrow a number system. When we have genuinely incomparable entities, we have a *partial order*, as opposed to what is called a *strict weak order*, so we cannot measure in the set of real numbers $\Re$. (A **strict weak order** has two properties: it is asymmetric and negatively transitive. By **asymmetric**, we mean that if the pair (x,y) is in the relation, then (y,x) is not in the relation. A relation is **negatively transitive** if, whenever (x,y) is in the relation, then for every z, either (x,z) or (z,y) is in the relation.) What we need instead is a mapping into *pairs* of real

numbers, that is, into the set $\Re \times \Re$. In the transport example, we can define a representation in the following way. First, we define a measure m that takes a transport type into a pair of elements:

m(Transport) = (Journey time, Cost per mile)

Then we define the actual pairs:

m(Car)=(3, 1.5)
m(Train)=(5, 2)
m(Plane)=(3.5, 3.5)
m(Coach)=(7,4)

The numerical binary relation over $\Re \times \Re$ that corresponds to the empirical superiority relation is defined as:

(x,y) superior to (x',y') if $x<x'$ and $y<y'$

The numerical relation preserves the empirical relation. It too is only a partial order in $\Re \times \Re$ because it contains incomparable pairs. For example, the pair (5,2) is not superior to (3.5,3.5); nor is (3.5,3.5) superior to (5,2).

Example 2.27: Suppose we wish to assess the *quality* of four different C compilers. We determine that our notion of quality is defined in terms of two sub-attributes: *speed* (average KLOC compiled per second) and *resource* (minimum Kbytes of RAM required). We collect data about each compiler, summarized in Table 2.11.

Table 2.11: Comparing four compilers

	Speed	*Resource*
A	45	200
B	30	400
C	20	300
D	10	600

Using the same sort of analysis as above, we can show that it is not possible to find a measure of this attribute in the real numbers that satisfies the representation condition.

These examples are especially relevant to software engineering. The International Standards Organization has published a standard, [ISO/IEC 25000:2005], for measuring software quality that explicitly defines *software quality* as the combination of seven distinct sub-attributes. We will discuss the details of this standard in Chapter 10. However, it is important to note here that the standard reflects a widely-held view that no single real-valued number can characterize such a broad attribute as quality. Instead, we look at n-tuples that characterize a set of n sub-attributes. The same observation can be made for *complexity* of programs.

Example 2.28: Many attempts have been made to define a single, real-valued metric to characterize program complexity. For instance, in Example 2.9, we were introduced to one of the most well-known of these metrics, the cyclomatic number. This number, originally defined by mathematicians on graphs, is the basis of an intuitive notion of program complexity. The number corresponds to an intuitive relation, *more complex than*, that allows us to compare program flowgraphs and then make judgments about the programs from which they came. That is, the cyclomatic number is a mapping from the flowgraphs into real numbers, intended to preserve the complexity relation. As we have seen in examining journey quality, if the relation *more complex than* is not a strict weak order, then cyclomatic number cannot be an ordinal-scale measure of complexity. (Indeed, a theorem of measurement theory asserts that a strict weak order is a necessary *and* sufficient condition for an ordinal scale representation in $\Re$.) We contend that no general notion of complexity can give rise to such an order. To see why, consider the graphs depicted in Figure 2.14. Flowgraph *y* represents a conditional choice structure, *x* represents a sequence of two such structures, and *z* represents a looping construct. Intuitively, it seems reasonable that graph *x* is more complex than graph *y*. If *more complex than* produced a strict weak order, we should be able to show that this relation is negatively transitive. That is, we should be able to show that for any *z*, either *x* is related to *z* or *z* is related to *y*. But neither of the following statements is obviously true:

x is more complex than *z*

and

z is more complex than *y*

Some programmers would argue that *x* is more complex than *z*, for instance, while others would say that *z* is more complex than *x*; we cannot reach consensus. In other words, some of the graphs are not comparable, so the relation is not a strict weak order and the cyclomatic number cannot be on an ordinal scale. Notice that the cyclomatic number for *x* is 3, for *y* is 2, and for *z* is 2, forcing us to conclude that *x* should be more complex than *z*. Thus, the cyclomatic number, clearly useful in counting the number of linearly independent paths in the program flowgraph, should not be used as a comprehensive measure of complexity.

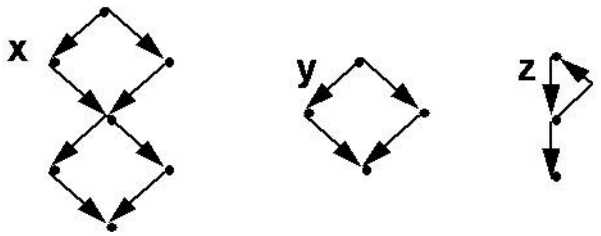

Figure 2.14: Three program flowgraphs

In spite of theoretical problems, there are many situations when we must combine sub-attributes to impose a strict ranking, and hence an ordinal scale. That is, we need

to define a single real-valued number as a measure. For example, if we are buying a coat, we may take into account the price, quality of material, fit and color. But in the end, we are forced to determine preference. Consciously or subconsciously, we must define some combination of component measures to arrive at a preference ranking.

> **Example 2.29:** We want to buy a contact management program. It is likely that the decision will be based on a collection of attributes, such as price, reliability and usability. If the program is for a single user's home PC, we may give price a heavier weighting than reliability when ranking the programs. However, if we are buying it for network use in a major organisation, it is likely that reliability would get a larger weighting than price..

Other, similar problems can arise. We may need to determine which program is safest, based on a set of criteria. Or we may wish to choose from among a variety of design techniques, based on survey data that captures developer preferences, design quality assessments, and cost of training and tools. Each of these instances presents a problem in making a decision with multiple criteria. There is an extensive literature on multi-criteria decision theory, and the measurement theory that relates to it. We discuss this type of analysis in Chapter 6, when we address data analysis techniques. However, here we must look at how to combine measures in a way that remains true to the spirit of measurement theory.

2.4.4 Indirect measurement and meaningfulness

When we measuring a complex attribute in terms of simpler sub-attributes, we are measuring indirectly. In doing so, we must adhere to the same basic rules of measurement theory that apply to direct measures. We must pay particular attention to issues of scale types and meaningfulness.

Scale types for indirect measures are similar to those for direct ones. Our concerns include the uniqueness of the representation, as well as the admissible transformations for each scale type. We call an admissible transformation a rescaling, and we define rescaling in the following way. Suppose that we measure each of n sub-attributes with measure M_i. Let M be an indirect measure involving components $M_1, M_2, ..., M_n$. That is, $M = f(M_1, M_2, ..., M_n)$ for some function f. We say that M' is a **rescaling** of M if there are rescalings $M_1', M_2', ..., M_n'$, of $M_1, M_2, ..., M_n$ respectively, such that $M' = f(M_1', M_2', ..., M_n')$.

Strictly speaking, this defines rescaling in the *wide sense*. Rescaling in the *narrow sense* requires us to verify that $M' = f(M_1, M_2, ..., M_n)$.

> **Example 2.30:** Density d is an indirect measure of mass m and volume V. The specific relationship is expressed as
>
> $$d=m/V$$

Every rescaling of d is of the form $d' = \alpha d$ (for $\alpha>0$). To see why, we must demonstrate two things: that a function of this form is a rescaling, and that every rescaling has this form. For the first part, we have to find rescalings m' and V' of m and V, respectively, such that $\alpha d = m'/V'$. Both m and V are ratio scale measures, so αm and V are acceptable rescalings of m and V respectively. Since

$$\alpha d = \alpha\left(\frac{m}{V}\right) = \frac{\alpha m}{V}$$

therefore we have a rescaling.

To show that every rescaling is of the appropriate form, notice that since m and V are ratio scale measures, every rescaling of m must be of the form $\alpha_1 m$ for some $\alpha_1$and every rescaling of V must be of the form $\alpha_2 V$ for some $\alpha_{2..}$ Therefore, every rescaling of d has the form

$$\frac{\alpha 1 m}{\alpha 2 V} = \frac{\alpha 1}{\alpha 2}\left(\frac{m}{V}\right) = \frac{\alpha 1}{\alpha 2} d = \alpha d \quad (\text{where } \alpha = \frac{\alpha 1}{\alpha 2})$$

Now, we can define scale types for indirect scales in exactly the same way as for direct scales. Example 2.30 shows us that the scale for density d is ratio, because all the admissible transformations have the form $d \rightarrow \alpha d$. In the same way, we can show that the scale type for an indirect measure M will generally be no stronger than the weakest of the scale types of the M_is. Thus, if the M_is contain a mixture of ratio, interval, and nominal scale types, then the scale type for M will at best be nominal, since it is weakest.

Example 2.31: An indirect measure of testing efficiency T is D/E, where D is the number of defects discovered and E is effort in person-months. Here D is an absolute scale measure, while E is on the ratio scale. Since absolute is stronger than ratio scale, it follows that T is a ratio scale measure. Consequently, the acceptable rescalings of T arise from rescalings of E into other measures of effort (person-days, person-years, etc.)

Many of the measures we have used in our examples are assessment measures. But indirect measures proliferate as prediction measures, too.

Example 2.32: In Example 2.12, we saw that many software resource prediction models predict effort E (in person-months) by using an equation of the form

$$E = aS^b$$

where S is a measure of software size, and a and b are constants. Some researchers have doubted the meaningfulness of these indirect effort measures. For example, DeMillo and

Lipton looked at the Walston and Felix model. Walston and Felix assert that effort can be predicted by the equation

$$E = 5.2S^{0.91}$$

where S is measured in lines of code. (See [Perlis *et al.* 1981].) DeMillo and Lipton contend that the prediction equation is an example of a meaningless measure. They assert that "both E and S are expressed as a ratio scale ... but the measurement is not invariant under the transformation $S \rightarrow \alpha S$ and so is meaningless." [DeMillo and Lipton 1981] In fact, this argument is relevant only when we consider scales defined in the *narrow* sense. In the more usual wide sense, it is easy to show that the equation is meaningful and that the scale type for effort is ratio. However, demonstrating scale type and meaningfulness is very different from asserting that the relationship is valid.

Many times, models of effort involve several levels of indirect measurement. That is, an indirect measure is defined to be a combination of other measures, both direct and indirect.

Example 2.33: Halstead developed a theory of software physics (discussed in Chapter 8), that defines attributes as combinations of counts of operators and operands. His equation for software effort, E, is

$$E = V/L$$

where V, the program volume, is on a ratio scale, but L, the estimated program level, appears to be only an ordinal scale. Thus, E cannot be a ratio scale. However, Halstead claims that E represents the number of mental discriminations necessary to implement the program, which is *necessarily* a ratio scale measure of effort. Therefore, Halstead's effort equation is not meaningful.

The unit in which the measure is expressed can affect the scale of the measure.

Example 2.34: Consider another effort measure

$$E = 2.7v + 121w + 26x + 12y + 22z - 497$$

cited by DeMillo and Lipton. [DeMillo and Lipton 1981] E is supposed to represent person-months, v is the number of program instructions, and w is a subjective complexity rating. The value of x is the number of internal documents generated on the project, while y is the number of external documents. Finally, z is the size of the program in words. DeMillo and Lipton correctly point out that, as in Example 2.33, effort should be on a ratio scale, but it cannot be ratio in this equation because w, an ordinal measure, restricts E to being ordinal. Thus, the equation is meaningless. However, E could still be an ordinal scale measure of effort if we drop the pre-condition that E expresses effort in person-months.

In this chapter, we have laid a foundation of principles on which to base valid measurement. The next chapter builds on this foundation by introducing a framework for how to choose measures, based on needs and process.

2.5. Summary

Measurement requires us to identify intuitively-understood attributes possessed by clearly-defined entities. Then, we assign numbers or symbols to the entities in a way that captures our intuitive understanding about the attribute. Thus, direct measurement of a particular attribute must be preceded by intuitive understanding of that attribute. This intuitive understanding leads to the identification of relations between entities. For example, the attribute height for the entity person gives rise to relations like *is tall*, *taller than*, and *much taller than*.

To measure the attribute, we define corresponding relations in some number system; then measurement assigns numbers to the entities in such a way that these relations are preserved. This relationship between the domain and range relationships is called the *representation condition*.

In general, there may be many ways of assigning numbers that satisfy the representation condition. The nature of different assignments determines the scale type for the attribute. There are five well-known scale types: nominal, ordinal, interval, ratio and absolute. The scale type for a measure determines what kind of statements we can meaningfully make using the measure. In particular, the scale type tells us what kind of operations we can perform. For example, we can compute means for ratio scale measures, but not for ordinal measures; we can compute medians for ordinal scale measures but not for nominal scale measures.

Many attributes of interest in software engineering are not directly measurable. This situation forces us to use vectors of measures, with rules for combining the vector elements into a larger, indirect measure. We define scale types for these in a similar way to direct measures, and hence can determine when statements and operations are meaningful.

In the next chapter, we build on this foundation to examine a framework for measurement that helps us to select appropriate measures to meet our needs.

2.6. Exercises

1. At the beginning of the chapter, we posed four questions:
    - 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?
    - 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?

- 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?
- 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?

Based on what you have learned in this chapter, answer these questions.

2. i. List, in increasing order of sophistication, the five most important measurement scale types.

ii. Suppose that the attribute "complexity" of software modules is ranked as a whole number between 1 and 5, where 1 means "trivial," 2 "simple," 3 "moderate," 4 "complex," and 5 "incomprehensible." What is the scale type for this definition of complexity? How do you know? With this measure, how could you meaningfully measure the average of a set of modules?

3. We commonly use ordinal measurement scales. For example, we can use an ordinal scale to rank the understandability of programs as either *trivial*, *simple*, *moderate*, *complex* or *incomprehensible*. For each of two other common measurement scale types, give an example of a useful software measure of that type. State exactly which software entity is being measured and which attribute. State whether the entity is a product, process or resource.
4. Define measurement, and briefly summarize the representation condition for measurement.
5. For the empirical and numerical relation system of Example 2.5, determine which of the following numerical assignments satisfy the representation condition:
    i. M(Wonderman) = 100; M(Frankie) = 90; M(Peter) = 60
    ii. M(Wonderman) = 100; M(Frankie) = 120; M(Peter) = 60
    iii. M(Wonderman) = 100; M(Frankie) = 120; M(Peter) = 50
    iv. M(Wonderman) = 68; M(Frankie) = 75; M(Peter) = 40
6. For the relation systems in Example 2.6, determine which of the following mappings are representations. Explain your answers in terms of the representation condition.
    i. M(each delayed response) = 6; M(each incorrect output) = 6; M(each data-loss) = 69
    ii. M(each delayed response) = 1; M(each incorrect output) = 2; M(each data-loss) = 3
    iii. M(each delayed response) = 6; M(each incorrect output) = 3; M(each data-loss) = 2
    iv. M(each delayed response) = 0; M(each incorrect output) = 1; M(each data-loss) = 0.5
7. Suppose that we could classify every software failure as either a) syntactic, b) semantic, or c) system crash. Suppose additionally that we agree that every system crash failure is more critical than every semantic failure, which in turn is more critical than every syntactic failure. Use this information to define two different measures of the attribute of *criticality* of software failures. How are these measures related? What is the scale of each?
8. Explain why you would not conclude that the quality of program X was twice as great as program Y if integration testing revealed program X to have twice as many faults per KLOC than program Y.
9. Explain why it is wrong to assert that lines of code is a bad software measure.
10. Explain why neither M_4 nor M_5 is a valid mapping in Example 2.17.
11. In Example 2.19, determine the affine transformations from:
    i. M_1 to M_2
    ii. M_2 to M_1
    iii. M_2 to M_3
    iv. M_3 to M_2
    v. M_1 to M_3

12. Explain why duration of processes is measurable on a ratio scale. Give some example measures and the ratio transformations that relate them.
13. Determine which of the following statements are meaningful:
    i. The length of Program A is 50.
    ii. The length of Program A is 50 executable statements.
    iii. Program A took 3 months to write.
    iv. Program A is twice as long as Program B.
    v. Program A is 50 lines longer than Program B.
    vi. The cost of maintaining program A is twice that of maintaining Program B.
    vii. Program B is twice as maintainable as Program A.
    viii. Program A is more complex than Program B.

14. Formally, what do we mean when we say that a statement about measurement is meaningful? Discuss the meaningfulness of the following statements:

    "The average size of a Windows application program is about four times that of a similar Linux program."

    "Of the two Ada program analysis tools recommended in the Ada coding standard, tool A achieved a higher average usability rating than tool B." For this example, program usability was rated on a four-point scale:
    4: can be used by a non-programmer
    3: requires some knowledge of Ada
    2: usable only by someone with at least 5 years Ada programming experience
    1: totally unusable

15. Show that the mean can be used as a measure of central tendency for interval scale data.
16. Show that, for nominal scale measures, the median is not a meaningful notion of average, but the mode (that is, the most commonly occurring class of item) is meaningful.
17. Suppose that "complexity" of individual software modules is ranked (according to some specific criteria) as one of the following:

    {trivial, simple, moderate, complex, very complex, incomprehensible}

Let *M* be any measure (in the representation sense) for this notion of complexity, and let *S* be a set of modules for each of which *M* has been computed.

(I) You want to indicate the average complexity of the modules in *S*. How would you do this in a meaningful way? (Briefly explain your choice.)
(ii) Explain why it is not meaningful to compute the mean of the *M*s. (You should construct a statement involving means that you can prove is not meaningful.)
(iii) Give two examples of criteria that might be used to enable an assessor objectively to determine which of the complexity values a given module should be.
State carefully any assumptions you are making.

18. Example 2.27 defines a quality attribute for compilers. Draw a diagram to illustrate the empirical relation of quality. Explain why it is not possible to find a measure for this attribute in the set of real numbers that satisfies the representation condition. Define a

measurement mapping into an alternative number system that does satisfy the representation condition.

19. A commonly-used indirect measure of programmer productivity P is $P = L/E$, where L is the number of lines of code produced and E is effort in person-months. Show that every rescaling of P is of the form $P' = \alpha P$ (for $\alpha>0$).
20. Show that the Walston-Felix effort equation in Example 2.32 defines a ratio scale measure.
21. Construct a representation for the relations *greater functionality* and *greater user-friendliness* characterized by Table 2.1.
22. Consider the attribute, "number of bugs found," for software testing processes. Define an absolute scale measure for this attribute. Why is "number of bugs found" not an absolute scale measure of the attribute of *program correctness*?

2.7. Further reading

There is no elementary text book as such on measurement theory. The most readable book on the representational theory of measurement is:

Roberts F.S., Measurement Theory with Applications to Decision Making, Utility, and the Social Sciences, Addison-Wesley, Reading, Massachusetts, 1979.

A more formal mathematical treatment of the representational theory of measurement (only for the mathematically gifted) is

Krantz D.H., Luce R.D., Suppes P. and Tversky A., *Foundations of Measurement*, Volume 1, Academic Press, 1971.

A very good introduction to measurement using non-scientific examples, including attributes like religiosity and aspects of political ideology may be found in:

Finkelstein L., "What is not measurable, make measurable," Measurement and Control, 15, pp. 25-32, 1982.

Detailed discussions of alternative definitions of meaningfulness in measurement may be found in:

Falmagne, J.-C. and Narens, L., "Scales and meaningfulness of quantitative laws," Synthese, 55, pp. 287-325, 1983.

Roberts, F. S., "Applications of the theory of meaningfulness to psychology," Journal of Mathematical Psychology, 29, pp. 311-332, 1985.

The origin of the definition of the hierarchy of measurement scale types (nominal, ordinal, interval, and ratio) is the classic paper:

Stevens S.S., "On the theory of scale types and measurement," Science, 103, pp. 677-680, 1946.

A criticism of this basic approach appears in:

Velleman P.F. and Wilkinson L., "Nominal, ordinal, interval and ratio typologies are misleading," The American Statistician, 47(1), pp. 65-72, February 1993.

Other relevant references are:

Belton V., "A comparison of the analytic hierarchy process and a simple multi-attribute utility function," *European Journal of Operational Research*, 26, pp. 7-21, 1986.

Briand L.C., Morasca S., Basili V.R., "Property-Based Software Engineering Measurement," *IEEE Trans. Software Engineering, 22(1), pp. 68-86, 1996.*

Finkelstein L., "A review of the fundamental concepts of measurement," *Measurement*, 2(1), pp. 25-34, 1984.

Finkelstein L., "Representation by symbol systems as an extension of the concept of measurement," *Kybernetes*, Volume 4, pp. 215-223, 1975.

Sydenham P.H. (Ed.), *Handbook of Measurement Science*, Volume 1, John Wiley, New York, 1982.

Campbell N.R., *Physics: The Elements*, Cambridge University Press, Cambridge, Mass, 1920. Reprinted as *Foundations of Science: The Philosophy of Theory and Experiment*, Dover, New York, 1957.

Ellis B., *Basic Concepts of Measurement*, Cambridge University Press, 1966.

Kyburg H.E., *Theory and Measurement*, Cambridge University Press, 1984.

Vincke P., *Multicriteria Decision Aids*, John Wiley, New York, 1992.

Zuse H., *Software Complexity: Measures and Methods*. De Gruyter, Berlin, 1991.