DRASKO

Software Testing and Management

Professor Bruha

Software Engineering 3S03

Winter 2011

McMaster University

SOFTWARE ENGINEERING 3S03

TABLE OF CONTENTS & ACKNOWLEDGEMENTS

CHAPTER 7

PROGRAM TESTING

In the previous chapter, we saw how to transform a program's design specification into a programming language implementation of the design. Figure 7.1 shows that we are ready to address the problem of testing the resulting modules to see whether the programs work as intended. First, we look at *defective software* to see what types of errors can occur. Then, we discuss the *purpose* of testing. We see that testing a software project is different from the kind of testing you do for your class projects. The testing process is described in terms of who performs the tests, what steps are involved, and what types of tests can be used.

Next, we examine in depth the *testing of individual modules*. After you, as a programmer, have eliminated errors, your code can be evaluated in several ways. It can be reviewed in a program walkthrough or inspection or proven correct using formal techniques. We note the difference between *testing* and *proving* and see how tests are designed to be thorough. Finally, we discuss how to generate test data as a set of test cases.

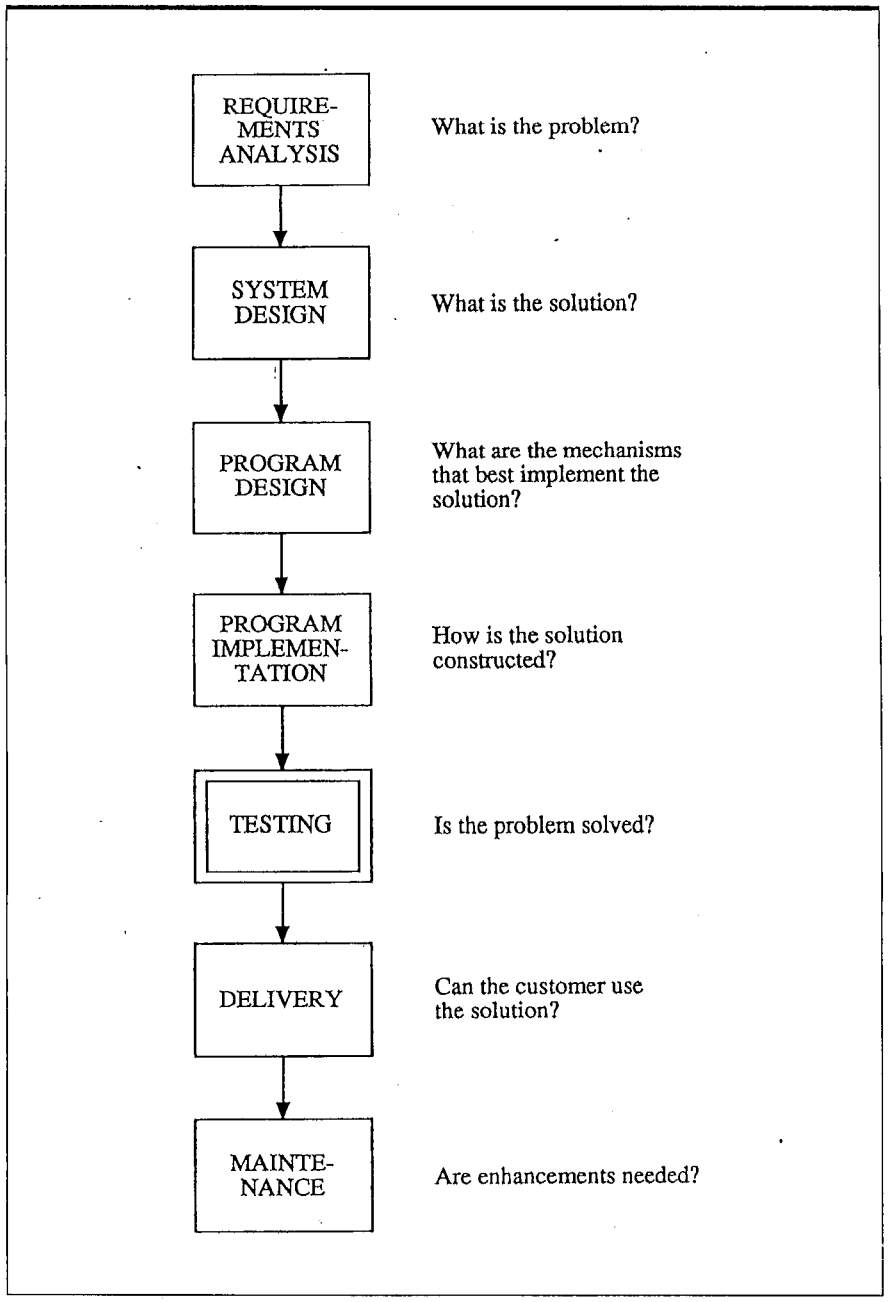
Integration testing involves the merging of tested modules, and several approaches can be taken. We detail each of these and compare their advantages and disadvantages. Once an approach is chosen, we can select from among several *automated test tools* to ensure a thorough and traceable series of tests.

In general, the testing phase of software development requires careful planning and coordination. We investigate the creation of a *test plan* to direct our testing efforts. Finally, we examine the notions of software *reliability*, *maintainability*, and *availability* to see if testing can demonstrate the quality that we intended to build in.

7.1 DEFECTIVE SOFTWARE

In an ideal situation, we, as programmers, become so good at our craft that every program we produce works properly every time we run it. Unfortunately, this ideal is not reality. We saw in chapter 1 that the inherent differences between software systems and hardware systems lead to software in which we are not 100% confident. This lack of confidence stems from several things. First, many software systems deal with large numbers of states and with complex formulae, activities, and algorithms. In addition to that, we use the tools at our disposal to implement a

Figure 7.1 The System Development Process



customer's conception of a system when the customer is sometimes uncertain of exactly what is needed. Finally, the size of a project and the number of people involved can add complexity. Thus, the presence of errors in the software we write is a function not only of the software itself but also of user and customer expectations.

Software Errors

What do we mean when we say that there is an error in our software? Usually, we mean that the software does not do what the requirements documents describe. For example, the requirements specification may state that the system must respond to a particular query only when the user is authorized to see the data. If the program responds to an unauthorized user, we say that the system is not working properly. This may be the result of any of several reasons:

1. The specification may be wrong. It may not be what the customer really wants or needs. Perhaps the customer means that several classes of authorization should be established.
2. The specification may specify something that is physically impossible, given the hardware and software prescribed by the customer.
3. The system design may be at fault. Perhaps the way in which the data base and its query language were designed makes it impossible to prevent the user from seeing the response.
4. The program design may be at fault. The module descriptions may contain an access control algorithm that does not handle this case properly.
5. The program code may be wrong. The code may implement the access algorithm improperly or incompletely.

By the time we have coded and are testing the program modules, we hope that the specifications are correct. Moreover, having used the software engineering techniques described in previous chapters, we have tried to assure that the design of the system and program modules reflects the requirements and forms a basis for a sound implementation. However, the stages of the software development cycle involve not only our computing skills but also our communication and interpersonal skills. It is entirely possible that an error in a software system can result from a misunderstanding during the development process. We say that a software system contains an **error** if it does not do what the customer expects it to do. A **failure** is an occurrence of an error somewhere in the software system. Thus, an error reflects the developer's view of the software, whereas a failure reflects the customer's view. The customer reports a failure, and the developers must find and correct the errors that resulted in that failure.

Errors are not inherent in software. Bridges, buildings, and other engineered constructions may fail because of shoddy materials or because they wear out after a period of time. However, DO-WHILE statements do not wear out after several hundred iterations, and RETURN statements do not fall off the ends of subroutines.

If a particular piece of code does not work properly and a spurious hardware failure is not the root of the problem, then we can be certain that an error is present in the code. It is for this reason that many software engineers refuse to use the term "bug" to describe a software error; calling an error a "bug" implies that the error wandered into the code from some external source over which the developers have no control.

In previous chapters, we examined the ways in which the system can be specified and designed in order to minimize the introduction of errors during those steps. In this chapter, we examine techniques that can minimize the occurrence of errors in the program code itself.

Types of Errors. After coding the program modules, we usually examine the code to spot errors and to eliminate them right away. When no obvious errors exist, we then test our program to see if we can isolate more errors. Thus, it is important that we know the kinds of errors for which to look. We can catalog the types of errors that occur in the coding of the program modules.

The first kind of error that we usually try to eliminate is an **algorithmic error**. An algorithmic error is one in which a program module's algorithm or logic does not produce the proper output for a given input because something is wrong with the processing steps. These errors are sometimes easy to spot just by reading through the program. Typical algorithmic errors include:

1. Branching too soon
2. Branching too late
3. Testing for the wrong condition
4. Forgetting to initialize variables or set loop invariants
5. Forgetting to test for a particular condition (such as when division by zero might occur)
6. Comparing variables of inappropriate data types

When checking for algorithmic errors, we may also check for **syntax errors**. Here, we want to be sure that we have properly used the constructs of the programming language. Sometimes the presence of a seemingly trivial syntax error can lead to disastrous results. For example, Myers ([MYE76]) points out that the first United States space mission to Venus failed because of a missing comma in a FORTRAN DO loop. Fortunately, compilers catch many of the syntax errors.

Computation and precision errors occur when the implementation of a formula is wrong or does not compute the result to the required degree of accuracy. For example, combining integer and fixed- or floating-point variables in an expression may produce unexpected results. Sometimes, improper use of floating-point data or ordering of operations may result in less than acceptable precision.

When the documentation describes the program's function but does not match what the program actually does, we say that the program has **documentation errors**. Often the documentation is derived from the program design and provides a very clear description of what the programmer would *like* the program to do, but the implementation of those functions is in error. Such errors can lead to a proliferation

of errors later in the life of the program since many of us tend to believe the documentation when examining the code to make modifications.

The requirements specification usually details the number of users and devices and the need for communication in a system. Thus, the system design often tailors the system characteristics to handle no more than the maximum load described by the requirements. These characteristics are carried through to the program design as parameters for the lengths of queues, the size of buffers, the dimensions of tables, and so on. **Stress** or **overload errors** occur when these data structures are filled past their specified capacity.

Similarly, **capacity** or **boundary errors** occur when the performance of a system becomes unacceptable as the activity on the system reaches its specified limit. For instance, if the requirements specify that a system must handle thirty-two devices, the programs must be tested to monitor the performance of the system when all thirty-two devices are active. Moreover, the system should also be tested to see what happens when thirty-three devices are active, if such activity is possible. By testing and documenting the system's reaction to a configuration beyond its capacity, the test team helps the maintenance team understand the implications of increasing system capacity in the future. Capacity conditions should also be examined in relation to the number of disk accesses, the number of interrupts, the number of tasks running concurrently, and other system-related measures.

In developing real-time systems, a critical consideration is the coordination of several processes executing simultaneously or in a carefully defined sequence. **Timing** or **coordination errors** occur when the code coordinating these events is inadequate. There are two reasons why this kind of error is very hard to identify and correct. First, it is usually difficult for designers and programmers to anticipate all possible system states. Second, because so many factors are involved with timing and processing, it may be impossible to replicate an error after it has occurred.

Throughput or **performance errors** occur when the system does not perform at the speed prescribed by the requirements. These are timing errors of a different sort: time constraints are placed on the system's performance by the customer's requirements, rather than by the need for coordination.

As we saw in the design stages, care is taken to ensure that the system can recover from a variety of error conditions. **Recovery errors** can occur when an error is encountered and the system does not behave as the designers desire or as the customer requires. For example, if a power failure occurs during system processing, the system should recover in an acceptable manner. For some systems, such recovery may mean that the system will continue full processing by using a backup power system; for others, this recovery means that the system keeps a log of transactions, allowing it to continue its processing whenever the power is restored.

For many systems, a set of hardware and system software is prescribed in the requirements. The software modules are designed according to specifications in the hardware and system software documentation. For example, if a modem is to be used for communications, the modem driver program generates the commands expected by the modem and reads the commands received from the modem. However, **hardware and system software errors** can arise when the documentation for the supplied hardware and software does not match their actual operating conditions

Finally, the program modules are reviewed to guarantee that the standards and procedures imposed on the system by the development team have, in fact, been followed. **Standards and procedures errors** may not affect the running of the programs, but they may foster an environment for the creation of errors as the system is tested and modified. By failing to follow the prescribed standards, one programmer may make it difficult for another to understand the programming logic or to find the data descriptions needed to solve a problem.

Purpose of Testing

No matter how capably we write programs, it is clear from the variety of possible errors that we should check to ensure that our modules are coded correctly. Many programmers view testing as a demonstration that their programs perform properly. However, the idea of demonstrating correctness is really the reverse of what testing is all about. We **test** a program in order to demonstrate the *existence* of an error. Because our goal is to discover errors, we can consider a test successful only when an error is discovered. Once an error is found, "**debugging**" or **error correction** is the process of determining what *causes* the error and of *making changes* to the system so that the error no longer exists.

Attitudes Toward Testing. New programmers are not accustomed to viewing testing as a discovery process. As a student, you write programs according to specifications given by your instructor. After having designed a program, you write the lines of code and compile them to determine if any syntax errors are present. When submitting your program for a grade, you usually present your instructor with a program listing and some kind of *test evidence*. The evidence is often a set of input data and the corresponding output from your program; the input data may be chosen to persuade your instructor that the program functions as described in the assignment.

You may have considered your program only as a solution to a problem; you may not have considered the problem itself. If so, your test data may have been chosen to show positive results in certain cases, rather than the absence of errors. Programs written in this way are evidence of your programming skill. Psychologically, a critique of your program is a critique of your ability. Thus, testing by showing that your program works correctly is a way of demonstrating your ability to your instructor.

However, when you are developing a system for a customer, the customer is not interested in knowing that the system works properly under certain conditions but rather in knowing that the system works properly under *all* conditions. Therefore, your goal as a developer should be to eliminate as many errors in the system as possible, no matter where in the system they occur and no matter who created them. There is no room in the development process for hurt feelings as errors are discovered.

Hence, many software engineers adopt an attitude known as *egoless programming*, where program modules are viewed as the components of a larger system, not as property of those who wrote them. When an error is found, the egoless development team is concerned with determining the cause of the error and correcting it, not with placing blame on a particular developer.

Who Performs the Tests? Even when a system is developed with an egoless approach, it is sometimes difficult for us to remove our personal feelings from the testing process. Thus, we often use an independent test team to test a system. In this way, we avoid conflict between personal responsibility for errors and the need to discover as many errors as possible.

In addition, there are several other factors that justify an independent team. As we have seen earlier in this chapter, there is always the possibility that we have introduced errors when interpreting the program design, determining the program logic, writing the descriptive documentation, or implementing the algorithms. Clearly, we would not have submitted our code for testing if we did not think that the code performs according to the specifications. However, we may be too close to our program to be able to view it objectively and spot some of the more subtle errors.

Furthermore, an independent test team can participate in reviewing the modules throughout development. The team can be part of both the requirements review and the design reviews. After having tested each module individually, the team can continue to test as the system is integrated and presented to the customer for acceptance. In this way, testing can proceed concurrently with the coding of programming modules; the test team can test modules and begin to piece them together as the programming staff continues to code other modules.

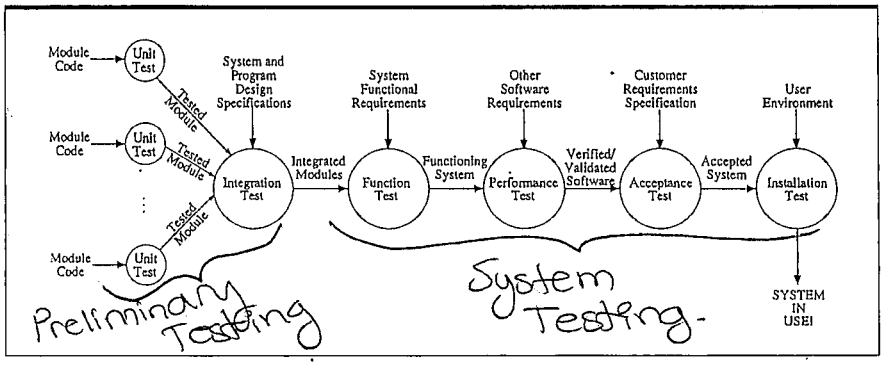
Stages of Testing. In the development of a large system, testing involves several stages. First, each program module is tested as a single program, usually isolated from the other programs in the system. Such testing, known as **module testing** or **unit testing**, verifies that the module functions properly with the types of input expected from studying the module design. Unit testing is done in a controlled environment whenever possible so that the test team can feed a predetermined set of data to the module being tested and observe what output data are produced. In addition, the test team checks the internal data structures, the logic, and the boundary conditions for the input and output data.

When collections of modules have been unit tested, the next step is to insure that the interfaces among the modules are defined and handled properly. **Integration testing** is the process of verifying that the components of a system work together as described in the program design and system design specifications.

Once we are sure that information is passed among modules according to the design prescriptions, we test the system to assure that it has the desired functionality. A **function test** evaluates the system to determine if the functions described by the requirements specification are actually performed by the integrated system. The result, then, is a functioning system.

Recall that the requirements were specified in two ways: first in the customer's terminology and again as a set of software and hardware requirements. The function test compares the system being built with the functions described in the software and hardware requirements. Then, a **performance test** compares the system with

Figure 7.2 Stages of Testing



the remainder of the software and hardware requirements. If the test is performed in the customer's actual working environment, a successful test yields a **validated system.**

When the performance test is complete, we as developers are certain that the system functions according to our understanding of the system description. The next step is to confer with the customer to make certain that the system works according to the customer's expectations. We join the customer to perform an **acceptance test** in which the system is checked against the customer's requirements description. When the acceptance test is complete, the accepted system is installed in the environment in which it will be used; a final **installation test** is performed to make sure that the system still functions as it should.

Figure 7.2 illustrates the several stages of testing. Although systems may differ in size, the type of testing described in each stage is necessary for assuring the proper performance of any system being developed.

In this chapter, we describe the preliminary testing stages: unit and integration testing. These stages deal with the testing of modules and the incorporation of the modules into a cohesive system. In chapter 8, we will investigate the remaining stages of testing, often collectively called **system testing.** In these later stages, the system is tested as one large entity, rather than as separate pieces.

Types of Testing. Before we discuss unit testing, let us consider the philosophy behind our testing. As you test a module, group of modules, subsystem, or system, your view of the object being tested can affect the way in which the testing proceeds. If you view the test object from the outside as a *closed box* whose contents are unknown, your testing consists of feeding input to the closed box and of noting what output results are produced. In this case, the test's goal is to be sure that every possible kind of input is submitted and that the output observed matches the output expected.

There are advantages and disadvantages to this kind of testing. The obvious advantage is that closed box testing is free of the constraints imposed by the internal

structure and logic of the test object. However, it is not always possible to run a complete test in this manner. For example, suppose a simple module accepts as input the three real numbers A, B, and C and produces as output the roots of the equation

$$Ax^2 + Bx + C = 0$$

or the message "NO REAL ROOTS." It is impossible to test the module by submitting to it every possible triple of real numbers (A, B, C). In this case, the test team may be able to choose representative test data to show that all possible combinations are handled properly. For instance, test data may be chosen so that the discriminant, $B^2 - 4AC$, is in each of three classes: positive, zero, or negative. However, if a test in each of the three classes reveals no error, we have no guarantee that the module is error-free. The module may still fail for a particular case.

For some test objects, it is impossible for the test team to generate a set of representative test cases that demonstrate correct functionality for all cases. Recall from chapter 6 the example of the module that accepted an adjusted gross income as input and produced the amount of federal income tax owed as output. We might have a tax table showing expected output for certain given inputs, but we may not know in general how the tax is calculated. The algorithm for computing the tax depends on tax brackets, and both the bracket limits and associated percentages are part of the internal processing of the module. By viewing this module as a closed box, we could not choose representative test cases because we would not know enough about the brackets.

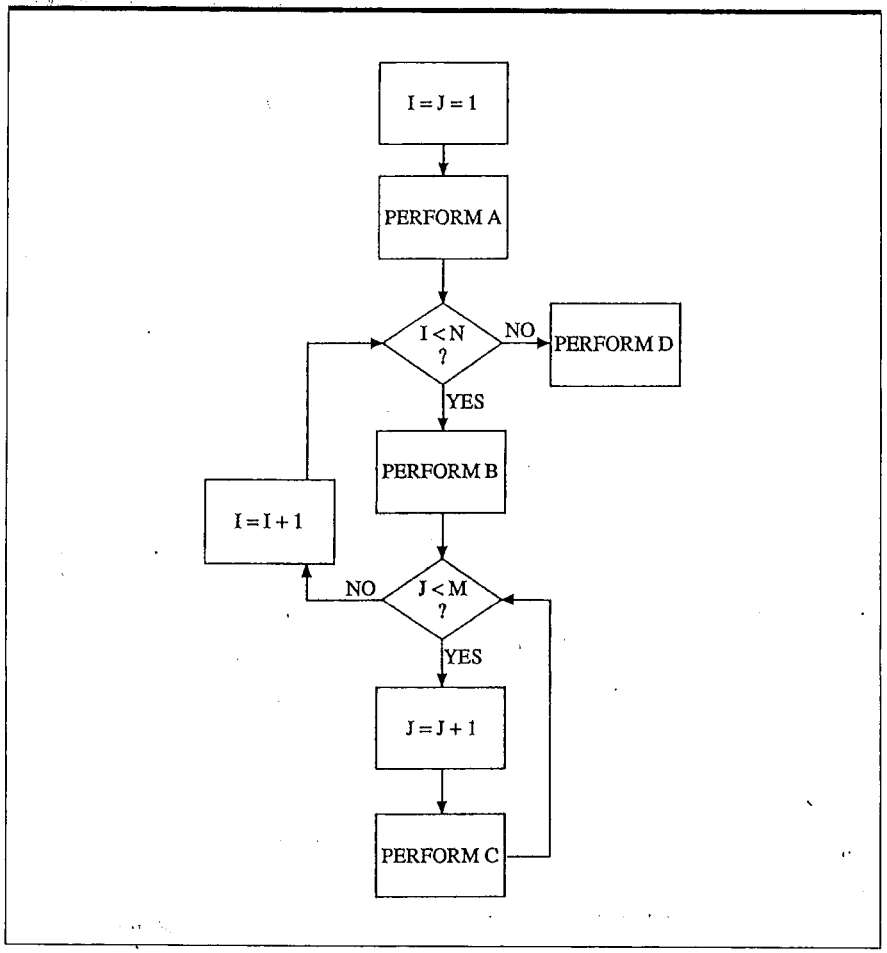
To overcome this problem, we can instead view the test object as an *open box* with whose internal structure and logic we are completely familiar. Then we can devise test cases that execute all the statements or all the control paths within the module or modules to be sure that the test object is working properly. However, as we will see later in this chapter, it may be impractical to take this approach. A module with a large number of branches and loops has many paths to check.

Even with a fairly simple logical structure, a module with substantial iteration or recursion is difficult to test thoroughly. For example, suppose a module's logic is structured so that the module loops NM times, as shown in Figure 7.3. If N and M are each equal to 100,000, a test case would have to loop 10 billion times to exercise all the logic paths.

When deciding how to test, we need not choose either open or closed box testing exclusively. We can think of closed box testing as one end of a *testing continuum* and open box testing as the other end. Any test philosophy can lie somewhere in between. The choice of test philosophy depends on a number of things, including the following:

1. The number of possible logical paths
2. The nature of the input data
3. The amount of computation involved
4. The complexity of the algorithms

Figure 7.3 Module with Iteration



7.2 UNIT TESTING

Now we examine the problem of testing individual modules after they have been coded. Given our goal of finding errors in the modules, how do we start? The process used is similar to the way you test a program assigned in class. First, you probably examine your program by reading through the code and trying to spot algorithm and syntax errors. You may even compare the code with the specifications and with the design to make sure that you have considered all necessary

cases. Next, you compile the code and eliminate any remaining syntax errors. Finally, you develop test cases to show that the input is properly converted to the output desired.

Unit testing of a program module is done in the same way. First, you and your peers examine the code for errors. When you find no more errors in this manner, the module is compiled and run with test data to search for other errors. Let us consider each step in turn.

Program Reviews

You work with a program design description to code and document each program module. Thus, the program reflects your *interpretation* of the module design, and the documentation explains *in words* what the program is supposed to do *in code*. It is helpful to ask an objective group of experts to review both your code and its documentation for errors. A second examination by such a review team often can recognize errors caused by your particular biases or interpretations.

This process, known as a **program review**, is similar to the ones in which the system and program design descriptions are reviewed prior to implementation. A team, composed of you as the programmer and three or four other technical experts, studies the program. The technical experts can be other programmers, technical writers, designers, or project supervisors. Whereas the design review teams included the customer's representatives, a program review team contains no one from the customer's organization. In addition to not understanding the technical details of a program module, customers are not concerned with the implementation at this level. They express their requirements and approve the proposed design; they are interested in implementation only when we can demonstrate that the system as a whole works according to their description.

Program Walkthroughs. There are two types of program review: a walkthrough and an inspection. In a **program walkthrough**, you present your code and the accompanying documentation to the review team, and the team comments on the correctness of the program. During a walkthrough, *you* lead the session and control the discussion. The atmosphere is informal, and the focus of attention is on the program, not the programmer. Although supervisory personnel may be present, the walk-through has no influence on your performance appraisal. This approach is consistent with the intent of testing in general: to find errors, not to correct them.

Program Inspections. A program inspection is similar to a walkthrough. Whereas a walkthrough is an informal presentation to a review team, a **program inspection** is a formal review in which the review team checks the program against a prepared list of concerns. For example, the review team may examine the definition and use of data structures and data types to see if their use is consistent with the program design and with the system standards and procedures. The team can review the algorithms and computation for correctness and efficiency. Comments accompanying the code can be compared with the code itself to insure that the comments

are accurate and complete. The interfaces between this module and others can be checked for correctness. The team may even choose to estimate the program's performance characteristics in terms of processing speed or memory usage; this estimate may be necessary for a system in which performance constraints are outlined in the system requirements.

The candidates for members of the inspection team are the same as those for a walkthrough. The composition of the team will be determined by the goals of the inspection. For example, an inspection to insure proper communication interfaces may have on its team the system designer who wrote the original communication design plans. Because the list of items to inspect determines what is reviewed, the *inspection team*—not the programmer—controls the review. Rather than having you lead the discussion (as in a walkthrough), the review team questions you using the list as a guide. As with walkthroughs, inspections are meant to criticize the program, not the programmer; the inspection results are not part of your performance evaluation.

Success of Program Reviews. You may feel uncomfortable with the idea of a review team examining your code. However, program reviews have proven to be extraordinarily successful at detecting errors. Remember that the earlier in the system development process an error is spotted, the easier and less expensive it is to correct. The same principle holds true for program errors. It is better to find an error at the module level than to discover it in a later testing phase when it can be much more difficult to determine the error's source.

Several researchers have investigated the extent to which program reviews have identified errors. Fagan ([FAG76]) performed an experiment in which 67% of the errors eventually detected in a system were found before unit testing by using code inspections. In this experiment, a second group of programmers wrote a similar program using informal walkthroughs rather than inspections. The inspection group had 38% fewer errors during the first seven months of operation than did the walkthrough group. In another Fagan experiment, of the total number of errors discovered during a system's development, 82% were found during design and code inspections. The early detection of the errors led to large savings in programmer time.

Jones ([JON77]) has done extensive studies of programmer productivity, including investigations of the nature of programmer errors and of the methods for discovering and removing the errors. Examining the history of the errors in ten million lines of program code, he found that code inspections removed as many as 85% of the total errors found. No other technique studied by Jones was as successful; in fact, none could remove even half of the errors known to be present.

Proving Programs Correct

Suppose your program module has been coded, examined by you, and reviewed by a program review team. The next step in testing is to subject the program to scrutiny in a more structured way. We want to establish somehow that the program is

correct. For the purposes of unit testing, a program is **correct** if it implements the functions specified in the program design and interfaces properly with the other modules.

One way to investigate program correctness is to view the program as a statement of logical flow. If we can rewrite the program in terms of a formal logical system (such as a series of statements and implications about data), then we can test this new expression for correctness. Although our interpretation of correctness is in terms of the design specification, we want our new expression to be correct in a formal or precise sense. For example, if we can prove that our new expression is a mathematical theorem, then the truth of the theorem may imply the correctness of the program.

A Formal Logic Proof Technique. Floyd ([FLO67]), Naur ([NAU66]), and others have developed a method to display the logic of a program as a series of **assertions**, that is, a series of statements each of which is either true or false. The conversion of the program to its logical counterpart is performed as a series of steps.

1. You begin by writing assertions to describe the input and output conditions for the program. These statements are combinations of logical variables (each of which is true or false) connected by the logical connective symbols displayed in Table 7.1.

Table 7.1: Logical Connectives

Connective	Example	Meaning
Conjunction	$x \& y$	x AND y
Disjunction	$x * y$	x OR y
Negation	$\bar{x}$	NOT x
Implication	$x \rightarrow y$	IF x THEN y
Equivalence	$x = y$	x EQUALS y
Universal quantifier	FOR ALL x (P(x))	For all x, condition P(x) is true
Existential quantifier	FOR SOME x (P(x))	For at least one x, P(x) is true

For example, suppose a module accepts as input an array T of size N. As output, the module produces an equivalent array T′ consisting of the elements of T in ascending order. We can write the input conditions as the assertion

A1: (T is an array) & (T is of size N)

Similarly, we can write the output as the assertion

Aend: (T′ is an array) & (FOR ALL i if $i < N$ then (T′(i) <= T(i+1)))
& (FOR ALL i if $i <= N$ then FOR SOME j (T′(i) = T(j))
& (T′ is of size N)

2. Next, draw a flow diagram depicting the logical flow of the module. On the diagram, denote points at which a transformation of data takes place.

   Figure 7.4 shows such a diagram for an example module in which a bubble sort is used to rearrange the array T into ascending order. In the figure, two points have been highlighted to show where data transformations take place. The point marked with a single asterisk (*) can be described as an assertion in the following way:

   [(NOT(MORE)= TRUE)) & (I < N) & (T(i) > T(i+1))] →
   [(T(i) is exchanged with T(i+1)]

   Similarly, the point marked with a double asterisk (**) can be written as

   [(NOT(MORE) = TRUE)) & (I >= N)] → [T(i) sorted]

3. From the assertions, generate a series of theorems to be proven. Begin with the input assertion, A1. If the next transformation point is denoted as A2, then the first theorem states that if A1 is true, then A2 is true. In other words, the theorem says that

   A1 → A2

   Then, proceed to the next transformation point, A3, writing that

   A2 → A3

   In this way, state theorems

   Ai → Aj

   where Ai and Aj are adjacent transformation points in the flow diagram. The last theorem states that a condition of TRUE at the last transformation point implies the truth of the output assertion:

   Ak → Aend

   Alternately, you can work backwards through the transformation points in the flow diagram. Begin at Aend and find a transformation point Ak preceding Aend. Prove that

   Ak → Aend

   and then that

   Aj → Ak

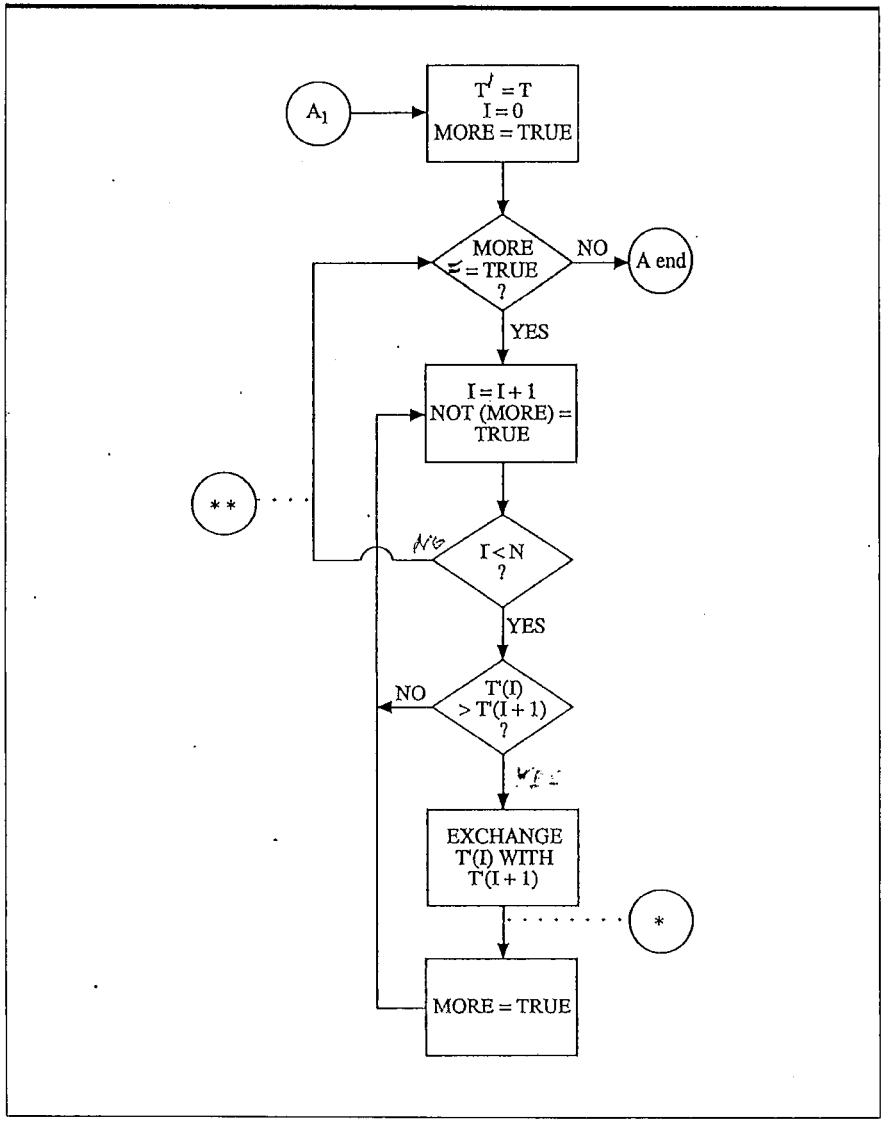
   for adjacent pairs of transformation points, and so on, until you have shown that

   A1 → A2

   The result of this approach is the same.

4. The next step is to locate the loops in the flow diagram. For each one, specify an IF-THEN assertion.

Figure 7.4 Flow Diagram for Bubble Sort



5. At this point, you have identified all possible assertions. To prove the program correct, locate all paths that begin with the input and end with the output. In other words, locate all paths that begin with A1 and end with Aend. By following each of these paths, you are following the ways in which

the program shows that the truth of the input condition leads to the truth of the output condition.

6. After identifying all paths, you must verify the truth of each one. Such verification is done by proving rigorously that the input assertion implies the output assertion according to the transformations of the path.
7. Finally, prove that the program terminates.

Advantages and Disadvantages of Logical Correctness Proofs. By constructing a program proof in the preceding manner (either in an automated way or by hand), we can discover algorithmic errors in the module. In addition, the proof technique provides us with a formal understanding of the program, because we examine the formal logical structure underlying the code. Regular use of this demonstration of correctness can force you to be much more rigorous and precise in specifying data, data structures, and algorithmic rules.

However, there is a price to be paid for such rigor. Much work is involved in setting up and carrying out the proof. For example, the code for the bubble sort module is much smaller than the logical description and proof. Thus, it may take more time to prove the code correct than to write the program itself. Moreover, larger and more complex program modules can involve enormous logic diagrams, many transformations, and a large number of paths to verify.

Nonnumerical programs may be more difficult to represent logically than numerical ones and, therefore, may be more difficult to prove in this way. Parallel processing is difficult to handle, and complex data structures may result in complex transformation statements.

Notice that the proof technique is based only on how the input assertions are transformed into the output assertions according to logical precepts. Proving the program correct in this logical sense does not mean that there are no software errors in the program code. Indeed, this technique may not spot any errors in the design, in the interfaces with other modules, in interpretation of the specification, in the syntax and semantics of the programming language, or in the documentation.

Finally, we must acknowledge that not all proofs are correct. There have been several times in the history of mathematics when a proof that had been accepted as valid for many years was later shown to have been fallacious. There is always the possibility that an especially complex or intricate proof argument is invalid.

Other Proof Techniques. The logical proof technique ignores the *structure* and *syntax* of the programming language in which the test program is implemented. In a sense, then, this technique proves that the design of the test program is correct but not necessarily the *implementation*. Other techniques take the language characteristics into account.

One such technique is known as **symbolic execution** of the program because the proof involves simulated execution of the test program using symbols instead of data variables. In symbolic execution, the test program is viewed as having an input state determined by the input data and conditions. As each line of the test program is executed, the technique checks to see whether the *state* has changed. Each state change is saved, and the execution of the program can be viewed as a *series of state changes*. Thus, each logical path through the program corresponds to an ordered series of state changes. The final state of each path should be an output state, and the program is correct if each possible input state generates the proper output state.

An example shows us how this technique works. Suppose several lines of a program to be tested read as follows:

```
A = B + C;
IF A > D THEN PERFORM TASKX
ELSE PERFORM TASKY;
```

A symbolic execution module will note that the condition (A > D) can be either true or false. Whereas the conventional execution of a program would involve specific values of A and D, the symbolic execution module records two possible states: (A > D) is FALSE, and (A > D) is TRUE. Instead of testing a large number of possible values for A and D, symbolic execution considers only two cases: when (A > D) is false and when (A > D) is true. In this way, large sets of data are divided into disjoint classes, and the program can be considered only with respect to how it reacts to each *class* of data. Considering only classes of data, represented as symbols, greatly reduces the number of cases to be considered in the proof.

However, this technique has many of the same disadvantages of logical theorem proving. Developing a proof may take longer than writing the program itself, and proof of correctness is not the same as the absence of software errors. Moreover, the technique relies on a careful tracing of changing conditions throughout the paths of the program. The technique can be automated to some extent, but large, complex test programs may still require the checking of many states and paths. It is difficult for an automated symbolic execution system to follow execution flow through loops. In addition, whenever subscripts and pointers are used in the test program, the partitioning of data into disjoint representative classes becomes more complex.

It may be useful to combine the logical flow and symbolic execution approaches to help you to implement the code correctly. First, you write assertions derived from the program design. You can test the assertions and their logical flow using the logical proof technique previously described. Then, you translate the assertions to lines of code, either manually or with some automated aid.

Automated Theorem Proving. Some software engineers have tried to automate the process of proving programs correct by developing modules that read as input

- the input data and conditions
- the output data and conditions
- the lines of code for the program to be tested

The output from the test module is either a proof of the program's correctness or a counterexample showing a set of data that the program does not correctly trans-

form to output. The test module must include information about the language in which the input program is written so that the syntax and semantic rules are accessible. Following the outlined steps, the test module identifies the paths of transformations and verifies their validity. Validity can be demonstrated in several ways. If the usual rules of inference and deduction are too cumbersome to be used, a heuristic solution can be found.

Such a theorem-proving module is nontrivial. For example, the module must be able to verify the correct use of unary and binary operations (such as addition, subtraction, and negation) as well as the use of comparisons involving equalities and inequalities. More complex laws such as commutativity, distributivity, and associativity must be incorporated in checking the test program. Expressing the programming language as a set of postulates from which to derive the theorems is very difficult.

Suppose these difficulties can be overcome. Using trial and error to construct the theorems is far too time consuming for any but the most trivial of test programs. Thus, some human interaction is desirable to guide the automated theorem prover. Using methods frequently employed when developing an expert system, an interactive theorem prover works with you to choose transformation points and trace paths. Thus, the theorem prover does not really generate the proof; rather, it checks the proof outlined by the person working with it. Using a symbolic execution approach, several experimental programs have been developed to evaluate the code in small FORTRAN, PL/I, and LISP programs. However, there is no general-purpose automated symbolic execution system available.

Can the ideal theorem prover ever be built, assuming the existence of a machine that is fast enough and an implementation language that can handle the complexities of the problem? The ideal theorem prover would read in any program and produce as its output either a statement confirming the program's correctness or the location of an error. The theorem prover would have to determine if an arbitrary statement in the test program is executed for arbitrary input data. Unfortunately, this kind of theorem prover can never be built. It can be shown (in [PFL85], for example) that the construction of such a program is the equivalent of the *halting problem* for Turing machines. The halting problem is an unsolvable problem, which means not only that there is no solution to the problem but also that it is impossible to ever find a solution to the problem. We can make our theorem prover solvable by applying it only to programs having no branches, but with this limitation the theorem prover is no longer universal. Thus, although highly desirable, any automated theorem prover will have to approximate the ideal.

Testing Programs

Proving programs correct is a goal to which software engineers aspire; consequently, much related research is being done to develop methods and automated tools. However, in the near future, development teams are more likely to be concerned with testing their software than with proving their programs correct.

Difference Between Testing and Proving. In proving a program correct, the test team or programmer considers only the code and input and output conditions of a program. The program is viewed in terms of the classes of data and conditions described in the program design. Thus, the proof of the test program may not involve executing the program but rather understanding what is going on *inside* the program.

However, customers have a different point of view. To demonstrate to them that a program is working properly, you must show them how the program performs from *outside* the program. In this sense, testing becomes a series of experiments, the results of which become a basis for deciding how the program will behave in a given situation. Whereas a proof tells us how a program will work in the hypothetical environment described by the design and requirements, testing gives us information about how a program works in its actual operating environment.

Test Thoroughness. To perform a test, then, we decide how to demonstrate in a convincing way that the test data exhibits all possible behaviors of the program being tested. Let us look at an example to see what choices we have.

Figure 7.5 illustrates the logic flow in a program to be tested. Each statement, represented by a diamond or rectangle, has been numbered. To test this program thoroughly, we can choose test cases through one of three approaches:

1. **Statement testing**: Every statement in the program is executed at least once.
2. **Branch testing**: For every decision point in the program, each branch is chosen at least once.
3. **Path testing**: Every distinct path through the program is executed at least once.

To perform statement testing on our example program requires a test case that executes statements 1 through 7. By choosing an X larger than K that produces a positive RESULT, we can execute statements

1-2-3-4-5-6-7

in order; thus, one test case suffices.

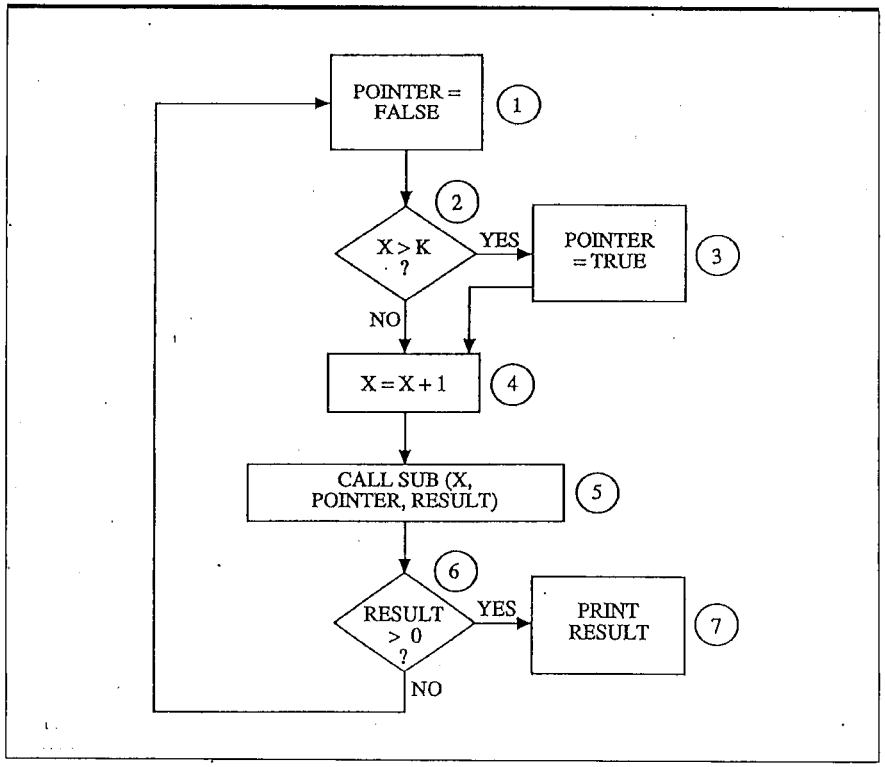
To perform branch testing, however, we must first find all decision points. The decision points are represented by diamonds in Figure 7.5. Thus, we have a decision about the relationship of X to K and another about whether or not RESULT is positive. We can choose two test cases to exercise the paths

1-2-3-4-5-6-7

and

1-2-4-5-6-1

Figure 7.5 Logic Flow in a Sample Program



and thereby traverse each possible outcome at least once. The first path uses the "yes" branch of the first decision point, while the second path uses the "no" branch. Likewise, the first path uses the "yes" branch of the second decision point, while the second path uses the "no" branch.

If we want our test to exercise each possible path throughout the program, we need more test cases. The paths

1-2-3-4-5-6-7
1-2-3-4-5-6-1
1-2-4-5-6-7
1-2-4-5-6-1

cover all the possibilities: two decision points with two choices at each branch.

In our example, statement testing requires fewer test cases than branch testing, which in turn requires fewer cases than path testing. This relationship is true in general; the more complex a program, the more path test cases required. Exercise 4 (page 337) investigates whether the structure and order of the decision points affect the number of paths through a program.

Usually, path testing is the most desirable since it tests a program more completely than the other two types. However, for complex programs, path testing can take unreasonable amounts of time. As we will see in a later section, time, resources, and other variables are considered when deciding how to test and what test cases to use.

Generating Test Data

To test a module, we choose input data and conditions, allow the program to manipulate that data, and observe the output from the program. We select the input data so that the output demonstrates something about the behavior of the program. A **test point** or **test case** is a particular choice of input data to be used in testing a program. A **test** is therefore a finite collection of test points. How do we choose test cases and define tests in order to convince ourselves and our customers that the program works correctly not only for the test cases but for all input?

Choosing Test Cases. We begin by determining the objective of our test. Then, we select test cases and define a test in order to meet that specific objective. Our objective may be to demonstrate that all statements execute properly. Or, we may want to show that every function performed by this program is done correctly. The objective determines how we classify the input in order to choose our test points.

We can view the program being tested as a closed box or as an open box, with the choice depending on our test objectives. If the program is a closed box, we supply the box with all possible input and compare the output with what is expected according to the requirements definition. However, if the program is viewed as an open box, we can examine the internal logic of the program. Our test objective can then be related to the assurance that all possible paths execute correctly.

Recall our example of a program that, given the coefficients of the variables, calculates the two roots of a quadratic equation. If our test objective is to demonstrate that the program functions properly, we might choose test points where the coefficients A, B, and C range through representative combinations of negative numbers, positive numbers, and zero. For example, we may choose combinations so that

1. A is greater than B and C
2. B is greater than C and A
3. C is greater than B and A

However, if our test acknowledges the *inner* workings of the program, we can see that the logical flow of the program depends on the value of the discriminant, $B^2 - 4AC$. Then, we can choose our test points so that we represent three cases, namely, when the discriminant is positive, negative, and zero.

Thus, keeping in mind our test objective, we can separate the possible input into classes. Ideally, these classes should meet the following criteria:

1. Every possible input belongs to one of the classes. (That is, the classes *cover* the entire set of input data.)
2. No input data set belongs to more than one class. (That is, the classes are *disjoint*.)
3. If the execution of the program demonstrates an error when a particular member of a class is used as input, then the same error can be detected by using any other member of the class as input. That is, any element of a class *represents all* elements of that class.

It is not always easy or feasible to tell if the third restriction on the classes can be met. We can loosen the third requirement so that if a data set belongs to a class and causes an error, then the probability is high that every other data set in that class causes the error.

Closed box testing suffers from uncertainty about whether the test cases selected will uncover a particular error. On the other hand, open box testing always admits the danger of paying too much attention to the code's internal processing. You may end up testing what the program *does* instead of what it *should* do.

We can combine open and closed box testing to generate test data. First, by considering the program as a closed box, we can use the *external specifications* of the program to generate initial test cases. These cases should incorporate not only the expected input data but also boundary conditions for the input and output, as well as several cases of invalid data. For instance, if the module is coded to expect an input variable that is supposed to be positive, a test case may be included for each of the following:

1. A very large positive integer
2. A positive integer
3. A positive fixed point decimal
4. A number between 0 and 1
5. 0
6. A negative number

Some of the data are purposely chosen to be improper input points; we test them to insure that the program handles erroneous input gracefully.

Next, by viewing the *internal structure* of the program, we can add other cases. For example, we can add data to test all branches of decision points and to exercise as many paths as possible through the test program. If there are loops involved, we may want test cases that pass through the loop many times, one time, and not at all.

We also examine the *implementation of algorithms*, where appropriate. For example, if we know that the program does trigonometric calculations, we may include cases that test the extremities of the trigonometric functions. Or, we may include data points that exercise functions where division by zero may occur so that we can test how the program handles such illegal divisions.

For some systems, *sequences* of test cases are needed. Sometimes a system "remembers" conditions from the previous case. For instance, recall of a previous state is needed when a system implements a finite state machine; the previous state and a current action determine the next state. Similarly, real-time systems are often interrupt-driven; tests are required for sets of cases, rather than for single ones.

7.3
INTEGRATION TESTING

When we are satisfied that individual program modules are working correctly and meet our established objectives, we combine the modules into a working system. This integration is planned and coordinated so that when an error occurs, we have some idea of what caused it. In addition, the order in which modules are tested affects our choice of test cases and the types of tools used. For large systems, some modules may be in the coding phase; others may be in the unit testing phase; and still other module collections are being tested together. Our *testing philosophy* explains why and how modules are combined to test the working system. This philosophy not only affects the timing of the integration and the order in which the program modules are coded but can also affect the cost and the thoroughness of the testing.

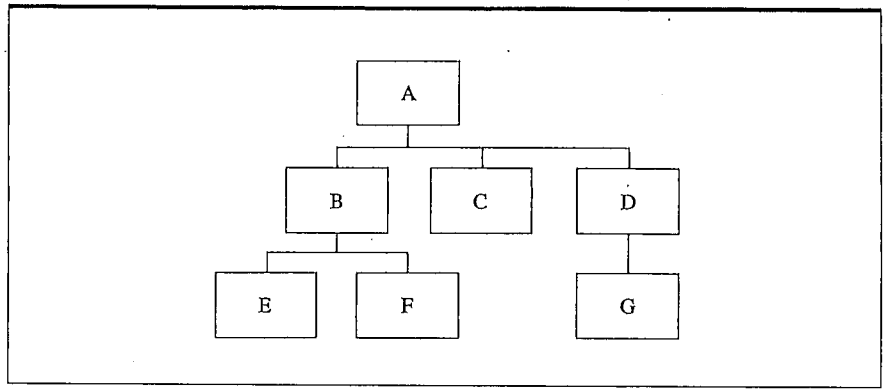
For testing, the entire system is again viewed as a hierarchy of modules, consistent with its modular design. As we have seen, each module belongs to a layer of the design. In integration testing, we begin with the modules at the highest level of the design and work down, begin at the bottom with the lowest level and work up, or use some combination of these approaches.

Bottom-up Approach

One popular approach for merging modules to test the larger system is called **bottom-up testing**. When this method is used, each module at the lowest level of the system hierarchy is tested individually. Then, the next modules to be tested are those that call the previously tested modules. This approach is followed repeatedly until all modules are included in the testing. The bottom-up method is useful when many of the low-level modules are general-purpose utility routines that are invoked often by others or when the design is object oriented.

For an example of bottom-up testing, consider the system modules illustrated in Figure 7.6. To test this system from the bottom up, we first test the modules in the lowest level: modules E, F, and G. Because we have no modules ready to call these lowest-level programs, we write special programs to aid us in the integration tests. A **module driver** is a routine that calls a particular module and passes a test case to

Figure 7.6 Sample Modular Hierarchy



it. A module driver is not difficult to code, since it rarely requires complex processing. However, care is taken to be sure that the driver's interface with the test module is defined properly. Sometimes, test data can be supplied automatically in a special-purpose language that facilitates the defining of the test data. This automated test data definition program is called a **module tester**.

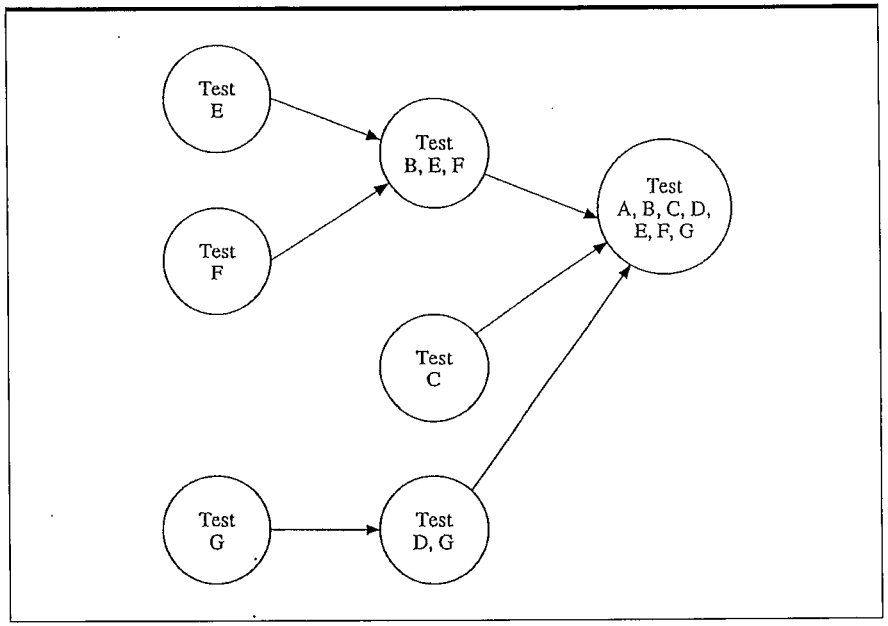
In our example, we need a module driver for each of modules E, F, and G. When we are satisfied that E, F, and G work correctly, we move to the next higher level. Unlike the modules in the lowest level, the modules in this next level are not tested by themselves. Instead, they are combined with the modules that they call (which have already been tested). Thus, the next step is to test B, E, and F together. If an error occurs, we know that its cause is probably in module B or in the interface between B and E or B and F, since E and F functioned properly on their own. Had we tested B, E, and F without having tested E and F separately, we might not have been able to isolate the error's cause as easily.

Similarly, we test module D with module G. Because C calls no module, we test it by itself. Finally, we test all modules of the system together. Figure 7.7 shows us the sequence of the tests and their dependencies on one another.

A frequent complaint about bottom-up testing in a functionally decomposed system is that the top-level modules in a system are usually the most important but are tested last. The top level directs the major activities of the system, whereas the bottom level often performs the more mundane tasks, such as input and output functions or repetitive calculations. The top levels are more general, while the lower levels are more specific. Thus, some developers feel that by testing the bottom levels first, the discovery of the major errors is postponed until the end of testing. Moreover, sometimes errors in the top levels reflect errors in design; obviously, these errors should be corrected as soon as possible in development. Finally, top-level modules often control or influence timing. It is difficult to test a system from the bottom up when much of the system's processing depends on timing.

general ← disadvantage: (Bottom-up).
spec top level tested last.

Figure 7.7 Bottom-up Testing Strategy



On the other hand, bottom-up testing is often the most sensible for object-oriented programs. Objects are combined one at a time with objects or collections of objects that have been tested previously. Messages are sent from one to another, and testing ensures that the object modules react correctly.

Top-down Approach

(2)

Many developers prefer to test using a **top-down approach**. In many ways, top-down is the reverse of bottom-up. The top level, usually one controlling module, is tested by itself. Then, all modules called by tested module(s) are combined and tested as a larger unit. This approach is reapplied until all modules are incorporated.

A module being tested may call another that is not yet tested. When this situation occurs, a special-purpose program is written to simulate the activity of the missing module. A **stub** is a program that simulates the activity of a missing module by answering to the identical calling sequence of the module and passing back output data that allows the testing process to continue. For example, if a module is called to calculate the next available address but that module is not yet integrated into the collection being tested, a stub for that module may pass back a fixed address only to allow testing to proceed. As with drivers, stubs need not be complex or logically complete.

Figure 7.8 Top-down Testing Strategy

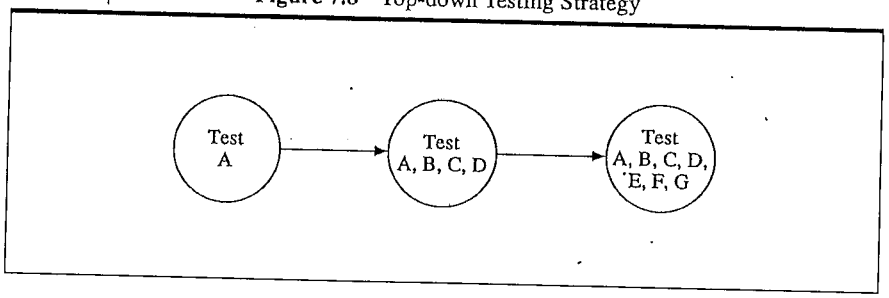


Figure 7.8 shows how top-down testing works with our example system. Only the top module is tested by itself. Once tested, it is combined with the next level, and A, B, C, and D are tested together. Stubs may be needed for modules E, F, and G at this stage of testing. Finally, the entire system is tested.

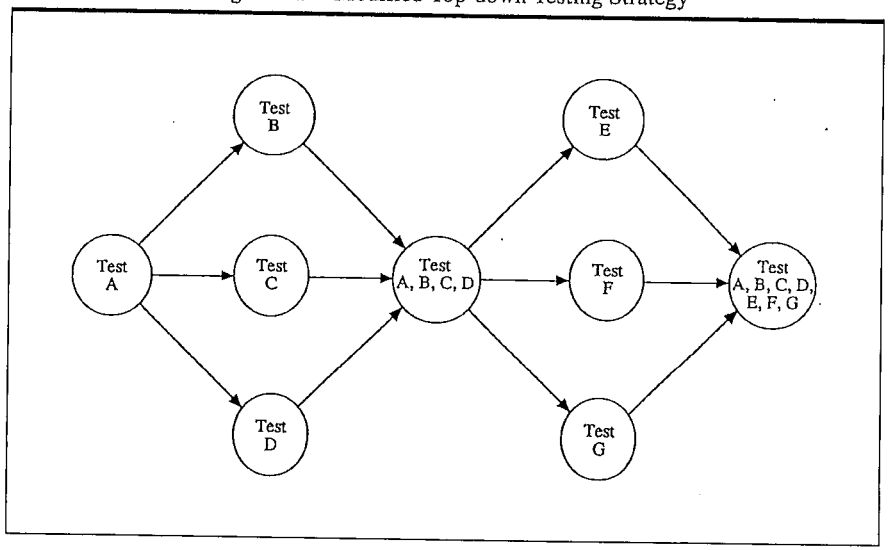
If the lowest level of modules performs the input and output operations, stubs for these modules may be almost identical to the actual programs. In this case, the integration sequence may be altered so that input and output modules are incorporated earlier in the testing sequence.

Many of the advantages of performing top-down design and coding also apply to the test process. When functions in particular modules have been localized by using top-down design, testing from the top down allows the test team to exercise one function at a time, following its command sequence from the highest levels of control down through appropriate modules. Thus, test cases can be defined in terms of the functions to be examined. Moreover, any design defects or major questions about the feasibility of the functioning of the system can be addressed at the beginning of testing, rather than at the end.

Notice, too, that driver programs are not needed in top-down testing. On the other hand, writing stubs can be difficult. Stubs must allow all possible conditions to be tested. For example, suppose module Z of a map-drawing system performs a calculation using latitude and longitude output by module Y. The design specifications state that the output from Y is always in the northern hemisphere. Since Z calls Y, when Z is part of a top-down test, Y may not yet be coded. A stub is written that generates a number between 0 and 180, allowing the testing of Z to continue. However, suppose a design change allows the output of Y to be in the southern hemisphere. The testing of the higher level may not have been checked for output in the wider range of –180 to 180. Thus, coding the stub is an important part of the testing, and its correctness may affect the validity of the test.

Modified Top-down Testing. A disadvantage to top-down testing is the possibility that a very large number of stubs may be required. This can happen when the lowest level of the system contains many general-purpose routines. One way to avoid this problem is to alter the strategy slightly. Rather than incorporate an entire level at a time, a modified top-down approach tests each level's modules individually before the merger takes place. For instance, our sample system can be tested with the modified approach by first testing module A; then testing modules B, C, and D; and then merging the four modules for a test of the first and second levels. Then, modules E, F, and G are tested by themselves. Finally, the entire system is combined for a test. Figure 7.9 shows this integration sequence.

Figure 7.9 Modified Top-down Testing Strategy

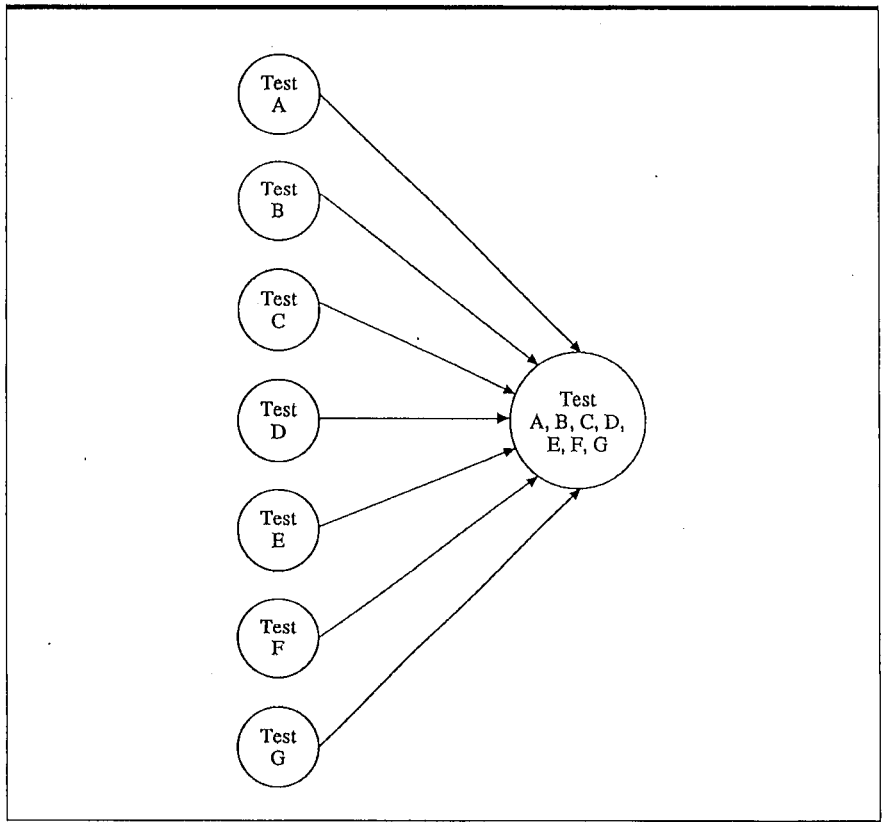


Testing each level's modules individually, however, introduces another difficulty. Both stubs and drivers are needed for each module so that each module can be tested individually. As we have seen before, the use of stubs and drivers can lead to additional problems.

Big-bang Testing

When all modules are tested in isolation, it is tempting to mix all of them together as the final system and see if it works the first time. Myers ([MYE76]) calls this **big-bang testing**. Figure 7.10 shows how modules are integrated using big-bang testing on our example system. Many programmers use the big-bang approach for small systems, but this approach is not practical for large systems. In fact, since big-bang has several major disadvantages, there is little to recommend it for *any* system. First, it requires both stubs and drivers for each module, since each module is tested by itself. Second, because all modules are merged at once, it is difficult to find the cause of any error that appears. Finally, interface errors cannot easily be distinguished from other errors. Thus, although widely used, big-bang testing is not recommended.

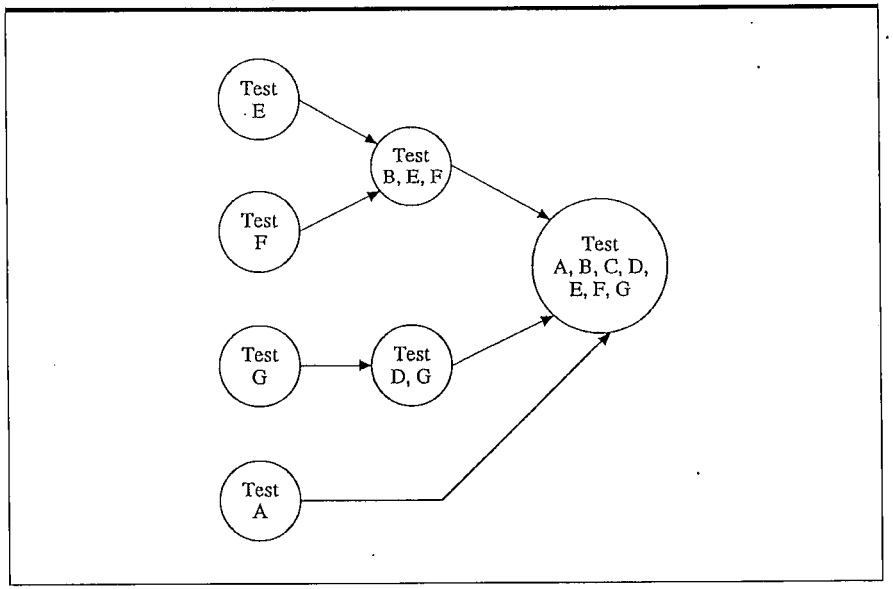
Figure 7.10 Big-bang Testing Strategy



Sandwich Testing

Myers ([MYE76]) combines a top-down strategy with a bottom-up one to form a **sandwich testing** approach. The system is viewed as having three layers, just as a sandwich does: a target layer in the middle, the levels above the target, and the levels below the target. A top-down approach is used in the top layer and a bottom-up approach in the lower layer. Testing converges on the target layer, chosen on the basis of system characteristics and the structure of the module hierarchy. For example, if the bottom layer contains many general-purpose utility programs, the target layer may be the one above, in which lie most of the modules using the utilities. This allows bottom-up testing to verify the correctness of the utilities at the beginning of testing. Then stubs for the utilities need not be written, since the actual utilities are available for use in further testing. Figure 7.11 depicts a possible integration sequence for sandwich testing our example, whose target layer is the middle level, composed of modules B, C, and D.

Figure 7.11 Sandwich Testing Strategy



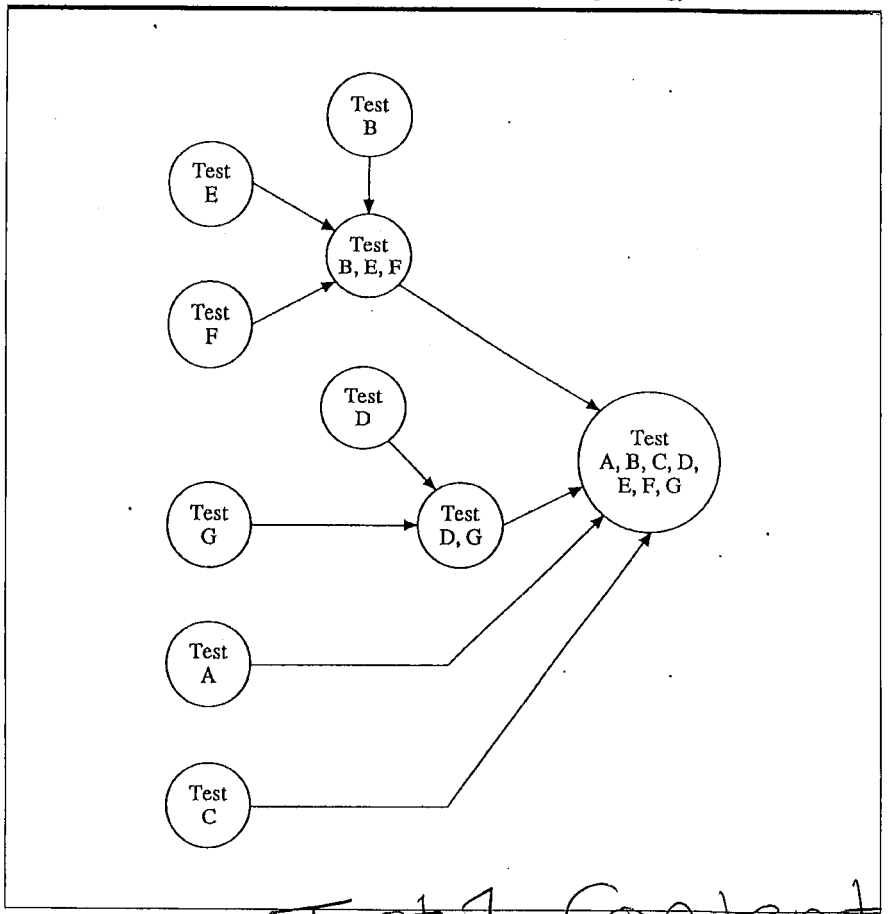
Sandwich testing allows integration testing to begin early in the testing process. It also combines the advantages of the top-down and bottom-up approaches by testing the controlling module and the utility modules at the very beginning. However, it does not test the individual modules thoroughly before integration. A variation, **modified sandwich testing**, allows upper-level modules to be tested individually before merging them with others for testing. Figure 7.12 illustrates this procedure with our example.

Comparison of Integration Strategies

The choice of integration strategy depends not only on system characteristics but also on customer expectations. For instance, the customer may want to see a working version of the system as soon as possible, so you may adopt an integration schedule that produces a basic working system early in the process. In this way, programmers are coding while others are testing so that the test and code phases can occur concurrently. Myers ([MYE76]) has composed a matrix, shown in Table 7.2, that compares the several types of testing strategies according to the characteristics of the system and the project and the expectations of the customer.

No matter what strategy is chosen, each module is merged only once for testing with the system. Furthermore, at no time is a module modified to simplify testing. Stubs and drivers are separate new programs, not temporary modifications of existing programs.

Figure 7.12 Modified Sandwich Testing Strategy



Test 1 Content
Ch. 1, 2, 3, 7.1–7.3

7.4 AUTOMATED TESTING TOOLS AND TECHNIQUES

Our discussion of unit testing described several kinds of automated tools to aid in testing program modules. Similar tools and techniques are available to help integration testing as well. They can be distinguished from one another in a very important

Table 7.2 Comparison of Integration Strategies

	Bottom-up	Top-down	Modified Top-down	Big-bang	Sandwich	Modified Sandwich
Integration	Early	Early	Early	Late	Early	Early
Time to basic working program	Late	Early	Early	Late	Early	Early
Module drivers needed	Yes	No	Yes	Yes	In part	Yes
Stubs needed	No	Yes	Yes	Yes	In part	In part
Work parallelism at beginning	Medium	Low	Medium	High	Medium	High
Ability to test particular paths	Easy	Hard	Easy	Easy	Medium	Easy
Ability to plan and control sequence	Easy	Hard	Hard	Easy	Hard	Hard

way. **Static analysis** of a program or module is performed when the program is not actually executing; **dynamic analysis** is performed while the program is running.

Static Analysis (1)

Several tools can analyze a source program before it is run. Tools that investigate the correctness of a program or set of modules can be grouped into four types:

1. *Code Analyzers*: The test modules are evaluated automatically for proper syntax. Statements can be highlighted if the syntax is wrong, if a construction is prone to error, or if an item has not been defined.
2. *Structure Checkers*: A graph is generated from the modules submitted as input. The graph depicts the hierarchy of the module or set of modules, and the automated tool checks for structural flaws.
3. *Data Analyzers*: An automated tool reviews the data structures, data declarations, and module interfaces, and then notes improper linkage between modules, conflicting data definitions, and illegal data usage.
4. *Sequence Checkers*: The sequences of events are checked; if coded in the wrong sequence, they are marked.

For example, a code analyzer can generate a symbol table to record where a variable is first defined and when it is used. Similarly, a structure checker can read a

program and determine the location of all loops, mark statements that are never executed, note the presence of branches from the middle of a loop, and so on. A data analyzer can notify us when a denominator may be set to zero; it can also check to see that subroutine arguments are passed properly. The input and output modules of a system may be submitted to a sequence checker to determine if the events are coded in the proper sequence. For example, a sequence checker can insure that all files are opened before they are modified.

Dynamic Analysis

Many times, systems are difficult to test because several parallel operations are being performed concurrently. This is especially true for real-time systems. In these cases, it is difficult to anticipate conditions and generate representative test points. Automated tools enable the test team to capture the state of events during the execution of a program by preserving a "snapshot" of conditions. These tools are sometimes called **program monitors** because they "watch" and report the behavior of the program.

A monitor can list the number of times a submodule is called or a line of code is executed. These statistics tell the testers if their test cases have statement coverage. Similarly, a monitor can report on whether a decision point has branched in all directions, thus providing information about branch coverage.

Additional information may help the test team evaluate the system's performance. Statistics can be generated about particular variables: their first value, last value, minimum value, and maximum value, for example. Breakpoints can be defined within the system so that when a variable attains or exceeds a certain value, the test tool reports the occurrence. Some tools stop when breakpoints are reached, allowing the tester to examine the contents of memory or values of specific data items; sometimes it is possible to change values as the test progresses.

For real-time systems, capturing as much information as possible about a particular state or condition during execution can be used after execution to provide additional information about the test. Control flow can be traced forward or backward from a breakpoint, and the accompanying data changes can be examined.

Other Automated Aids

In addition to analyzing code, automated tools can aid the testing itself. For example, *data bases* can be developed to track test cases. These data bases store the input data for each test case and describe the expected output. The test team can also use the data base to record the actual output. Associated with such tools are *nonprocedural languages* that are used to define and generate the test case descriptions. Several of these testing aids are sometimes combined into one automated tool. A **test harness** is an automated monitoring system that tracks test input data, passes it to the program or system being tested, and records the resulting output. A test harness can also compare actual with expected output and report any discrepancies.

7.5

THE TEST LIFE CYCLE

As we have seen, much is involved in testing a program and integrating it with other programs to build a system. The test process has a *life cycle* of its own within the development cycle. The steps in the test process include:

1. Establishing the test objectives
2. Designing the test cases
3. Writing the test cases
4. Testing the test cases
5. Executing the tests
6. Evaluating the test results

The test objective is essential in deciding what *kinds* of test cases to generate. Moreover, the design of test cases is the key to successful testing. If the test cases are not representative and do not thoroughly exercise those functions that demonstrate the correctness and validity of the system, then the remainder of the testing process is useless.

Therefore, running a test begins with a review of the test cases to verify that

1. They are correct
2. They are feasible
3. They provide the desired degree of coverage
4. They demonstrate the desired functionality

When we have validated the test cases according to these criteria, we can proceed with the actual test.

Test Plans

We use a plan to organize the activities of testing. The test plan takes into account the objectives of the testing process and incorporates any scheduling mandated by the type of testing strategy used. Let us investigate test plans in more detail.

Purpose of the Test Plan. The system development cycle requires several levels of testing, beginning with unit and integration testing and proceeding to demonstrate the functionality of the full system. The **test plan** describes the way in which we will demonstrate to our customer that the software works correctly (that is, that the software is free of technical errors and performs the functions as specified). Thus, a test plan addresses not only unit and integration testing but also

system testing. The test plan is a guide to the entire testing activity. It explains *who* does the testing, *why* the tests are performed, *how* tests are conducted, and *when* the tests are scheduled.

To develop the system's test plan, we must know the requirements, functional specifications, and the modular hierarchy of the system's software. As we develop each of these aspects of a system, we can apply what we know to choosing a test objective, defining a test strategy, and generating a set of test cases. Consequently, the test plan for a system is developed *as the system itself is developed.*

Contents of the Test Plan. A test plan begins with the *test objectives*. As we saw earlier in this chapter, there are several stages in the testing process, from unit testing through functional testing and acceptance testing to installation testing. The test plan for a system addresses each kind of testing and explains the objectives of each. Thus, a system test plan is really a series of test plans, one for each kind of testing to be performed: each stage is addressed in turn, elaborating on the objectives of each test.

Next, the test plan describes *how the tests will be administered* and what criteria will be used to judge when the test is complete. In other words, this part of the test plan explains when the tester knows that the test objectives have been met. Knowing when the test is over and that the objectives have been reached is not always easy. We have seen examples of code where it is impossible or impractical to test every possible set of input data. By choosing a subset of all possible data, we admit the possibility that an error might occur. This trade-off between completeness of testing and the realities of cost and time often involves a compromise with our objectives. Later in this chapter, we will discuss methods for estimating how many of the existing errors we have found.

Knowledge of the test objectives is a critical part of the test plan. The writers of the test plan (usually the members of the test team) work with the system and program designers to determine completion criteria for the early stages of testing and with the customer to define the criteria for the later stages.

When a test team can recognize that a test has met its objectives, we say that the test objectives are well defined. It is then that we decide how to *integrate* the program modules into a working system. At this point, we consider statement, branch, and path coverage at the module level and top-down, bottom-up, and other *integration strategies* at the integration level. The result is a plan for merging the modules into a whole, sometimes referred to separately as a **system integration plan**.

For each stage of testing, the test plan describes in detail the *methods* to be used to perform the tests. For example, unit testing may be composed of informal walkthroughs or formal inspections followed by static analysis of the code and then dynamic analysis of each module's performance. If an automated test tool or special techniques are to be used, the plan lists them and the conditions for their use. The explanation helps the test team plan its activities and schedule the tests.

A detailed list of *test cases* accompanies each test method or technique. The reasons for choosing test cases are provided to explain how the test cases address the test objectives. The test cases include:

- The *input data* to be used
- The *conditions* under which each test point is to be submitted to the module or system being tested
- A description of the *expected output*

The plan also describes the use of automated tools to generate the test cases or capture the output. If a data base is to track test points and the resultant output, the data base and its use are to be explained, too.

Thus, as you read the test plan, you have a complete picture of testing, from the overall strategy to the particular test data. By writing the test plan as you design the system, you are forced to understand the overall goals of the system. Sometimes the testing perspective forces you to ask questions about the nature of the problem and the appropriateness of the design.

Many customers specify the contents of a test plan in their requirements documentation. For example, the Department of Defense provides a developer with automated data systems documentation standards ([DOD77]) when a system is being built. The standards explain that the test plan

> is a tool for directing the . . . testing, and contains the orderly schedule of events and list of materials necessary to effect a comprehensive test of a complete [automated data system]. Those parts of the document directed toward the staff personnel shall be presented in nontechnical language and those parts of the document directed toward the operations personnel shall be presented in suitable terminology.

We will investigate the details of this test plan example in chapter 8.

7.6

ESTIMATING SOFTWARE QUALITY

In previous chapters, we have discussed the need for quality software. In testing the software we write, we and our customers want to be sure that the promised quality is indeed there. The characteristics often discussed as measurements of overall system quality are *reliability*, *availability*, and *maintainability*. These terms are borrowed from the engineering world, where these criteria are used to judge the quality of a hardware system.

Reliability, Availability, and Maintainability

Consider the meaning of these terms as they relate to an automobile. When we think of a car as being *reliable*, we mean that the car functions properly most of the time. We realize that there may be things that break down, wear out, and need to be fixed

or replaced, but a reliable car operates for long periods of time before it requires any kind of maintenance. In general, we can say that something is reliable if it has long periods of consistent behavior between maintenance periods.

Just as reliability reflects the car's condition over a period of time, *availability* reflects the car's condition at a particular point in time. A car is available if you can use it when you need it. Your car may be ten years old and has required maintenance only twice, so we know that the car is highly reliable. However, if it happens to be in the shop when you need it, it is still not available. Thus, something can be highly reliable, but if it is not working when you need it, it may have low availability.

Suppose your car is highly reliable and available, but it was manufactured twelve years ago by a foreign manufacturer who is no longer in business. When your car needs a repair (which, admittedly, is infrequently), you have difficulty finding the needed parts. Moreover, your mechanic has difficulty repairing the car because of its unusual construction. Because of these difficulties, repairing your car takes a long time; your car has low *maintainability*. In contrast, if the repair time were short, the car would be considered highly maintainable.

The same concepts carry over to software systems. We can judge software in terms of its reliability, availability, and maintainability. We want it to function consistently and correctly over long periods of time, to be available when we need it, and to be quickly and easily repaired if it does fail. A program's performance is **successful** if the program performs according to the customer's requirements. **Software reliability** is the probability that the program is successful for a given period of time. Since we use probability for our measure, reliability is expressed as a number between 0 (not reliable at all) and 1 (completely reliable).

Another measure of reliability is the time between failures. Often represented as a function of the mean time between failures (in hours, abbreviated MTBF), the reliability R of a system is

$$R = \frac{MTBF}{1 + MTBF}$$

Thus, if the mean time between failures is very small, R is close to 0; as the mean time between failures grows large, R approaches 1.

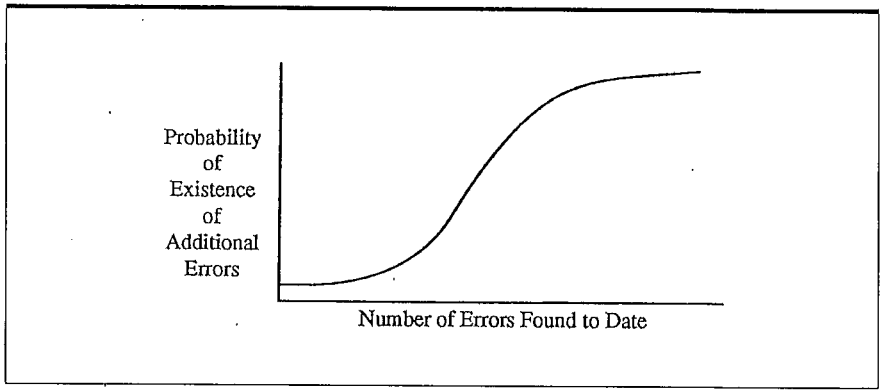
Similarly, **software availability** is the probability that a program is performing successfully according to specifications at a given point in time. If we denote the mean time (in hours) to repair a software error by MTTR, then the availability A of a system is related to MTTR and MTBF in the following way:

$$A = \frac{MTBF}{MTBF + MTTR}$$

Software maintainability is the probability that a software error can be fixed right away and is related to the mean time to repair:

$$M = \frac{1}{1 + MTTR}$$

Figure 7.13 Relationship Between Errors Found and Undetected



The remainder of this chapter examines reliability in more detail. For more information about other measures of software quality, see Shooman ([SHO83]).

Software Reliability

Software reliability cannot be added to a program at the last minute. To make software reliable, quality must be built into the system by basing the code on complete and accurate specifications followed by a well-engineered design. How do we measure reliability once we begin testing? We know that the measurement of reliability involves the mean time between failures. The fewer errors there are in a system or program, the fewer failures will result. Thus, an estimate of the *number of errors* in a program helps us determine if the program is reliable.

Estimating Number of Errors. It seems natural to assume that the software defects that are most difficult to find are also most difficult to correct. It also seems reasonable to believe that the most easily fixed errors are detected when the program is first examined, and the more difficult defects are located later in the testing process. However, Shooman and Bolsky ([SHO75]) have found that this is not the case. Sometimes it takes a great deal of time to find trivial errors, and many such errors are overlooked or don't appear until well into the testing process. Moreover, Myers ([MYE76]) reports that as the number of detected errors increases, the probability of the existence of more undetected errors increases, as shown in Figure 7.13. If there are many errors in a program, we want to find them as early as possible in the testing process. However, this graph tells us that if we find a large number of errors at the beginning of testing, we are likely to have a large number left undetected.

In addition to being contrary to our intuition, these results also make it difficult to know when to stop looking for errors during testing. We have to estimate the

number of errors remaining, not only to know when to stop our search for more errors but also to give us some degree of confidence in the programs we are producing. The number of errors also indicates the maintenance effort we can expect if errors are left to be detected after the system has been installed.

Mills ([MIL72]) developed a technique known as **error seeding** to estimate the number of errors in a program. It is based on an intentionally placed known number of errors, called *seeded errors*. A member of the test team seeds a program with errors. Then, the other team members test the program and locate as many of any kind of error as possible. The underlying assumption is that the ratio of seeded errors detected to total seeded errors should be the same as the ratio of nonseeded errors detected to the total number of nonseeded errors:

$$\frac{\text{detected seeded errors}}{\text{total seeded errors}} = \frac{\text{detected nonseeded errors}}{\text{total nonseeded errors}}$$

Thus, if 70 of 100 seeded errors are found, it is logical to assume that 70 percent of the nonseeded errors have also been found.

We can express the ratio more formally. Let S be the number of seeded errors placed in a program, and let N be the number of actual (nonseeded) errors. If n is the number of actual errors detected during testing and s the number of seeded errors detected during testing, then an estimate of the total number of actual errors is

$$N = \frac{Sn}{s}$$

Although simple and useful, this approach assumes that the seeded errors are of the *same kind* and *complexity* as the actual errors in the program. We do not know what the typical errors are before all the errors are found; consequently, it is difficult to make the seeded errors representative of the actual ones.

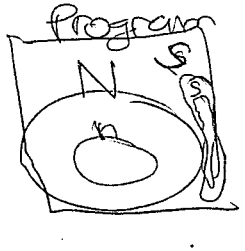
To overcome this obstacle, we can use two independent groups to test the same program. Call them Test Group 1 and Test Group 2. Let x be the number of errors detected by Test Group 1 and y be the number of errors detected by Test Group 2. Some of the errors detected by the groups will be the same; that is, some errors will be detected by both Group 1 and Group 2. Let q be the number of these errors. Finally, let n be the total number of all errors in the program. We want to estimate n.

The *effectiveness* of each group's testing can be measured by calculating the fraction of errors found by each group. Thus, the effectiveness $E(1)$ of Group 1 can be expressed as

$$E(1) = \frac{x}{N}$$

and the effectiveness $E(2)$ of Group 2 as

$$E(2) = \frac{y}{N}$$



The *effectiveness of a group* measures the group's ability to detect errors from among a set of existing errors. Thus, if a group can find half of all errors in a program, its effectiveness should be 0.5. Consider errors detected by both Group 1 and Group 2. If we assume that Group 1 is just as effective in finding errors in any part of the program as in any other part, we can look at the *ratio* of errors Group 1 found from the set of errors found by Group 2. Group 1 found q of the y errors that Group 2 found, so Group 1's effectiveness should be the same; that is,

$$E(1) = \frac{x}{N} = \frac{q}{y}$$

However, we know that $E(2)$ is y/n, so we can derive the following formula for n:

$$N = \frac{q}{E(1) * E(2)}$$

We have a known value for q, and we can use estimates of q/y for $E(1)$ and q/x for $E(2)$; this enables us to estimate n.

To see how this method works, suppose two groups test a program. Group 1 finds 25 errors. Group 2 finds 30 errors, and 15 of those errors are duplicates of those found by Group 1. Thus, we have

$$x = 25$$
$$y = 30$$
$$q = 15$$

The estimate $E(1)$ of Group 1's effectiveness is q/y, or 0.5, since Group 1 found 15 of the 30 errors found by Group 2. Similarly, the estimate $E(2)$ of Group 2's effectiveness is q/x, or 0.6. Thus, our estimate of n, the total number of errors in the program, is

$$\frac{15}{0.5 * 0.6} = 50 \text{ errors}$$

The test *strategy* defined in the test plan directs the test team in deciding when to stop testing. The strategy can use this estimating technique to tell us when testing is complete. If n is our estimate of the number of errors in a program, we can assume that our testing is thorough when we find approximately n errors. In the example above, Group 1 has found 25 errors and Group 2, 30. Since we have found 40 errors and the estimate for n is 50, testing should continue.

Confidence in the Software. We can use estimates of program errors to give us some idea of the confidence we can have in the software. Confidence is usually expressed as a percentage. We say that we have an ***n*% level of confidence** in the truth of an assertion if there is an $n/100$ probability that the assertion is true. For

example, if we say that a program is error free with a 95 percent level of confidence, then the probability that the program is error free is .95. Sometimes the reliability requirements for a system are stated in terms of a confidence level for freedom from errors.

Suppose we have seeded a program with S errors and we claim that the program has only N actual errors. We test the program until we have found all S of the seeded errors. If, as before, n is the number of actual errors detected during testing, the confidence level can be calculated as

$$C = \begin{cases} 1 & \text{if } n > N \\ \dfrac{S}{S-N+1} & \text{if } n \le N \end{cases}$$

For example, suppose we claim that a program is error free. Our claim means that N is zero. If we seed the program with ten errors and find all ten without finding an unseeded error, we can use the confidence formula with $S = 10$, $N = 0$. This tells us that C is 10/11, so we have a confidence level of 91 percent. Suppose our contract with the customer requires all programs to be tested to a level of 98 percent confidence. To achieve that level of confidence, we would need S seeded errors, where

$$\frac{S}{S-0+1} = \frac{98}{100}$$

Solving this equation, we find that S must be 49. In other words, to guarantee 98 percent confidence in an error-free program, we must seed the program with 49 errors and continue the testing process until all seeded errors are found.

The problem with this approach is that we cannot predict the level of confidence until all seeded errors are detected in the program. Richards ([RIC74]) has modified this technique so that the confidence level can be estimated using the number of detected seed errors, whether or not all have been located. In this case, C is

$$C = \begin{cases} 1 & \text{if } n > N \\ \dbinom{S}{s-1} \Big/ \dbinom{S+N+1}{N+s} & \text{if } n \le N \end{cases}$$

These estimates assume that all errors have an equal probability of being detected, an assumption which is not likely to be true. However, many other estimates take these factors into account. Such estimation techniques not only give us some idea of the confidence we may have in our programs but also provide a side benefit. Many programmers are tempted to conclude that each error is the last one. If we estimate the number of errors left or if we know how many errors we must find in order to satisfy a confidence requirement, we have incentive to keep testing for one more error.

Models of Reliability

It is often helpful to use a profile or model of error detection and removal in deciding where to test, when, and for how long. Earlier in this chapter, we looked at models that focused on the number of errors in the code. Next, we look at other aspects of reliability, including complexity and number of failures.

Module Complexity and Reliability. Some errors are more difficult to detect than others because some modules are more complex than others. Thus, modeling the *complexity* of system modules can enhance our estimation techniques. The complexity of a system and its modules depends largely on the modules' characteristics. In particular, the amount of coupling among modules and cohesion within modules can be modeled. If we number the modules of the system from 1 through n, we can define a matrix C, where each entry c_{ij} of the matrix is a measure of the coupling between module i and module j. At the same time, we define a vector D of size n, where D_i is a measure of the cohesion within module i. Using the information in the matrix and the array thus formed, we can form a new matrix, A, where entry a_{ij} is the probability that module i will have to change when module j is changed in some way. The methods for quantifying these characteristics and working with the matrices can be found in works of DeMarco ([DEM82]) and Myers ([MYE75], [MYE86]). The matrices can be useful in pointing out those modules that are most prone to contain errors. For example, we can look at the entries of matrix A that indicate a *high degree of change*. These entries correspond to modules that are probably complex and *most likely to develop new errors* when errors are detected elsewhere.

Models Based on Discovery of Failures. Much of the emphasis in testing is on discovery of errors: mistakes in code, requirements, design, and other artifacts of development. However, not every error is apparent to the user, and some errors are more severe than others. Musa ([MUS87], [MUS89a], and [MUS89b]) and others recommend that we distinguish between *errors* and *failures,* where a **failure** is a deviation from required behavior that results when code containing an error is executed. For example, in pressing a particular button, the wrong screen may be displayed. This failure may be the result of an error in the code that fails to initialize a variable. In general, errors are seen by developers, while failures are seen by users. We can model software reliability in terms of the user's viewpoint by considering the occurrence of failures.

For the same amount of testing, the *intensity of failures* (that is, the number of failures in a given period of time) is proportional to the number of faults or errors present when testing starts. This proportionality means that the number of failures is k times the number of faults for some constant k. Suppose our system performs

only two functions: function A, 90 percent of the time, and function B, 10 percent of the time. Suppose further that the software contains 100 errors, 50 associated with A and 50 with B. Then the failure intensity is

$$0.9(50)k + 0.1(50)k = 50k$$

By knowing the failure intensity of each function, we can concentrate our efforts on the function with higher intensity. If we eliminate all of the errors in function A, the new failure intensity of the system is

$$0.1(50)k = 5k$$

That is, by eliminating only half of the errors, we have reduced the failure intensity by a factor of 10. Thus, attention paid to failure intensity results in substantial increase in the quality of the system.

Many models of failure intensity have been proposed. The *Goel-Okumoto* model assumes that the number of failures in a time interval follows a statistical pattern known as a *Poisson distribution*. The average value of failures experienced is an exponential function that approaches an asymptote. The *Musa-Okumoto model* is similar, but its function describing the failures experienced by the system is logarithmic with respect to execution time. Brettschneider ([BRE89]) describes a simple model called *zero-failure testing*. It is used at Motorola and is derived from a failure rate function.

$$\text{failures to time } t = a\, e^{-b(t)}$$

for constants a and b. Let us see how such a model can be used in testing.

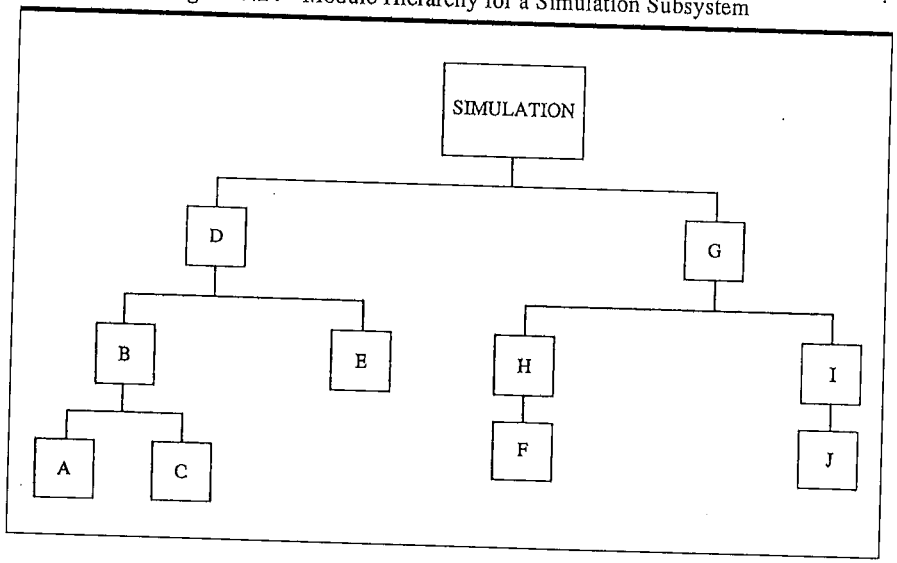
We use the reliability model to tell us how many hours we must test the system to meet our reliability goal. Thus, we need three inputs: the target projected average number of failures (*failures*), the total number of test failures detected so far (*test*), and the total test execution hours up to the last failure (*test to last failure*). The calculation for zero-failure test hours is

$$\left[\frac{\ln(\text{failures}/(0.5+\text{failures}))}{\ln((0.5+\text{failures})/(\text{test}+\text{failures}))}\right] \times (\text{test hours to last failure})$$

Suppose you are testing your 33,000-line program. Up to now, 15 failures have been detected over the total test time of 500 hours. During the last 50 hours of testing, no failures have been reported. Your goal is to guarantee no more than an average of 0.03 failure per 1000 lines of code. Based on the information you have, the projected average number of failures is 0.03 failure per 1000 times 33,000 lines of code, or 1. When the formula above is used, the number of test hours needed for these conditions is

$$\left[\frac{\ln(1/1.5)}{\ln(1.5/16)}\right] \times 450 = 77$$

Figure 7.14 Module Hierarchy for a Simulation Subsystem



Thus, you should reach the desired level of delivered failures if you can test for 77 hours after the last detected failure without discovering any more failures. Since you have already tested for 50 hours, you need only test for 27 hours more. However, if you discover a failure in that 27-hour period, you must continue your testing, recalculate, and restart the clock.

Models such as these are useful in many ways. They can guide you in your testing, so that you have some degree of confidence in declaring testing complete. In addition, the models can be used to assess the trade-offs between test time and desired quality.

7.7 TESTING THE RESOURCE TRACKING AND SIMULATION SYSTEM

We can apply the concepts introduced in this chapter to the Weaver Farm example with which we have been working. Suppose we have decided to implement and test the simulation subsystem first. If Figure 7.14 is the modular hierarchy for the subsystem, we examine the relationships among the modules and the characteristics of each module to decide on a testing strategy. For example, because the lowest-level modules are utility routines, we may want to incorporate some bottom-up

testing in our plan. When we have decided on an ordering of tests, we can depict the test cycle by drawing a diagram of the test schedule, as shown in Figure 7.15.

The test schedule tells us exactly how the modules will be unit tested and integrated with one another. If coding and testing are being done simultaneously, the test schedule tells you when your code must be written and unit tested.

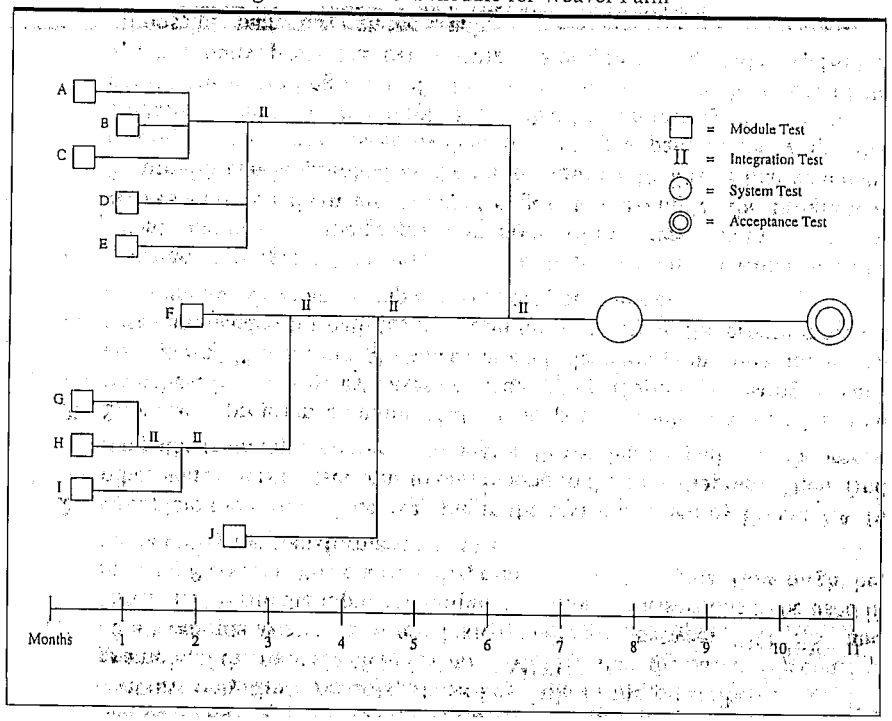
After the tests are scheduled, test cases are defined. Because the simulation system is to simulate all possible combinations of resources and timing constraints, the test team decides to use an automated tool to evaluate the test coverage. They use a tool to identify the paths in a module and report on the degree to which a test case covers all paths.

The tool notes the paths on a source code listing. As you can see from the sample below, each decision path is assigned a number.

```
STATEMENT  LISTING                                          DECISION PATHS

 1  begin
 2      incount := endpt;
 3      cr_last := false;
 4      while (iptr < incount) do
 5        begin
 6          if (ord(ch[iptr]) < ord(" ")) then                    1 - 2
 7            begin
 8              if (ch[iptr] = chr(cr))                           3 - 4
 9                then
10                  if (cr_last) then skipch                      5 - 6
11                    else cr_last := true
12              else
13                  if (ch[iptr] = chr(lf))                       7 - 8
14                    then
15                      if (cr_last)                              9 - 10
16                        then cr_last := false
17                      else skipch
18                  else
19                        skipch
20        end
21    else                                                        11
22        cr_last := false;
23        iptr := iptr+1;
24    end;
25  ch[0] := chr(incount);
26  endpt := incount;
```

Figure 7.15 Test Schedule for Weaver Farm



Accompanying the listing is more information about each decision path.

PATH NUMBER	STATEMENT NUMBER	BRANCH DESCRIPTION
1	6	PATH IS FALSE BRANCH
2	6	PATH IS TRUE BRANCH
3	8	PATH IS FALSE BRANCH
4	8	PATH IS TRUE BRANCH
5	10	PATH IS FALSE BRANCH
6	10	PATH IS TRUE BRANCH
7	13	PATH IS FALSE BRANCH
8	13	PATH IS TRUE BRANCH
9	15	PATH IS FALSE BRANCH
10	15	PATH IS TRUE BRANCH
11	21	PATH IS LOOP ESCAPE

As each test case is executed, the tool prints out a summary of the paths traversed during the test. The report for test case 6 may look like the one below.

TEST CASE	MODULE	NBR OF PATHS	THIS TEST INVO-CATIONS	THIS TEST PATHS TRAV.	THIS TEST % COVER	CUMULATIVE INVO-CATIONS	CUMULATIVE PATHS TRAV.	CUMULATIVE % COVER
6	SIMPARM	4	1	3	75.	6	4	100.
	INDATA	7	1	2	29.	6	6	89.
	DBSRCH	6	1	3	50.	5	5	83.
	SIMRUN	9	1	4	44.	6	8	88.
	OUTDATA	5	1	3	60.	6	4	80.
	FORMAT	4	0	0	00.	5	3	75.

Corresponding to this analysis, a list is printed of the decision paths not executed after six test cases:

MODULE	TEST	PATHS MISSED	TOTAL
SIMPARM	6	2	1
INDATA	6	3 4 5 6 7	5
DBSRCH	6	2 4 5	3
SIMRUN	6	2 3 4 5 9	5
OUTDATA	6	3 5	2
FORMAT	6	1 2 3 4	4

Clearly, the test tool tells the team whether it has met its test coverage objectives as specified in the test plan.

7.8

CHAPTER SUMMARY

This chapter has introduced the idea of testing a software system. Rather than viewing testing as a demonstration that a program or system works properly, the test team tries to detect errors. The testing phase of development is composed of a life cycle of its own—beginning with unit testing of each module, followed by integration of the software modules, and leading to a demonstration for the customer of the system's functionality and performance at the system's final destination.

Testing philosophies at various stages are based on test objectives set for each stage. Once the objectives are established, we select a set of test data that covers as many cases as possible. The testing of modules and their integration can include static analysis (evaluating programs before they are executed) and dynamic analysis (monitoring performance while the program is running). Many of the tools for testing programs have been automated, and tools are available for all stages of the test cycle.

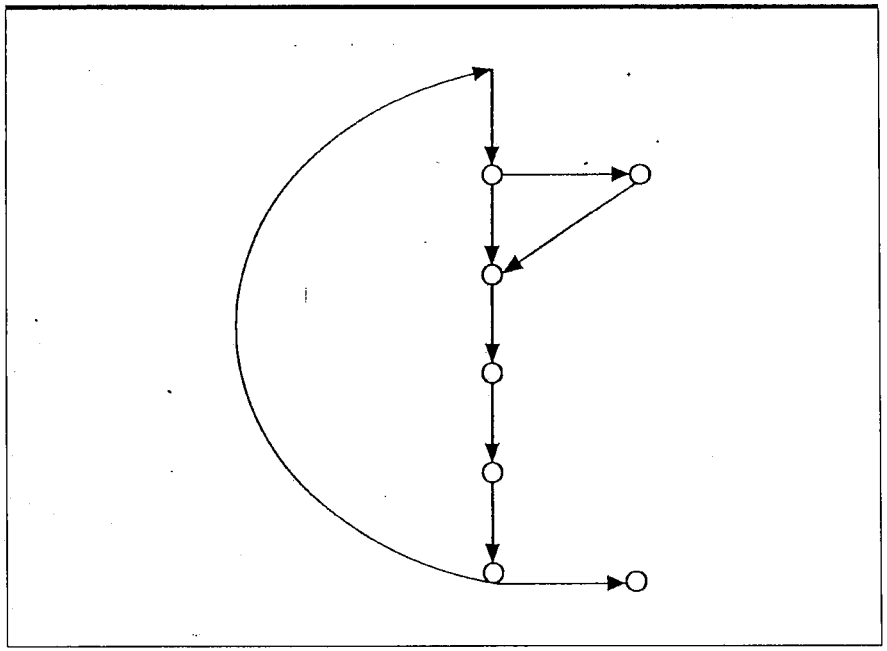
We have seen the need to measure the quality of the resulting software system. Quality can be determined by how reliable, available, and maintainable the software is. By counting the number of errors found as we test, we can estimate the total number of errors present in our programs and judge, from the number found, how many are left undetected.

In the next chapter, we will continue our discussion of testing with an investigation of the remainder of the test life cycle: function testing, performance testing, acceptance testing, and installation testing.

EXERCISES

1. Let P be a program module that reads in a list of N records and a range condition on the record key. The first seven characters of the record form the record key. The module P reads the key and produces an output file that contains only those records whose key falls in the prescribed range. For example, if the range is "JONES" to "SMITH," then the output file consists of all records whose keys are lexicographically between "JONES" and "SMITH." Write the input and output conditions as assertions to be used in proving P correct. Write a flow diagram of what P's logical flow might be, and identify the transformation points.
2. Complete the proof of the example in the text illustrated by Figure 7.4. In other words, write assertions to correspond to the flow diagram. Then, find the paths from input condition to output. Prove that the paths are theorems.
3. Suppose a program contains N decision points, each of which has two branches. How many test cases are needed to perform path testing on such a program? If there are M choices at each decision point, how many test cases are needed for path testing? Can the structure of the program reduce this number? Give an example to support your answer.
4. Consider a program flow diagram as a directed graph in which the diamonds and boxes of the program are nodes of the graph and the logic flow arrows between them are directed edges. For example, the program in Figure 7.5 can be graphed as shown in Figure 7.16. Prove that statement testing of a program is equivalent to finding a path in the graph that contains all nodes of the graph. Prove that branch testing of a program is equivalent to finding a set of paths whose union covers the edges of the graph. Finally, prove that path testing of a program is equivalent to finding all possible paths through the graph.
5. Programmable Problem: Write a program that accepts as input the nodes and edges of a directed graph and prints as output all possible paths through the graph. What are the major design considerations for your program?

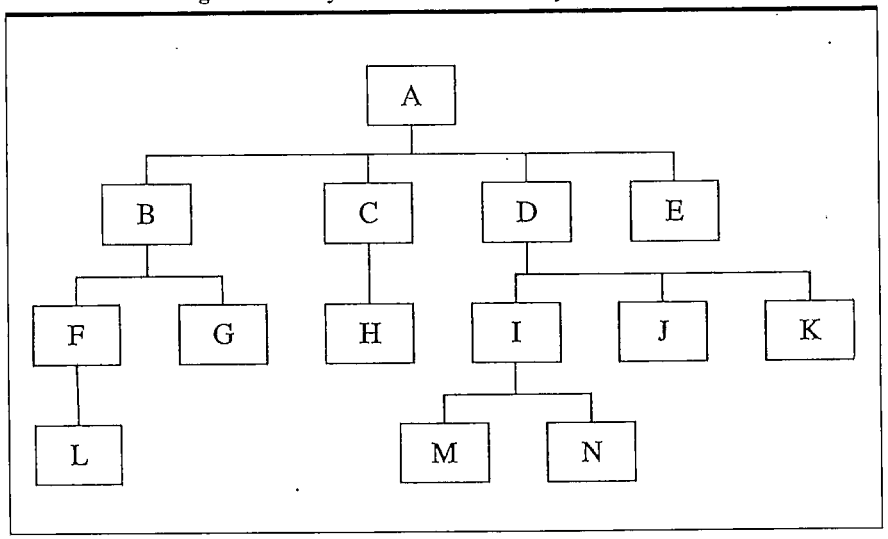
Figure 7.16 Graph of the Program in Figure 7.5



How does the complexity of the graph (in terms of the number of branches and cycles) affect the algorithm you use?

6. Figure 7.17 illustrates the hierarchy of modules in a software system. Describe the sequence of module tests for integrating the modules using a bottom-up approach; a top-down approach; a modified top-down approach; a big-bang approach; a sandwich approach; a modified sandwich approach.
7. Measurements of reliability, maintainability, and availability are often placed in the system requirements document. Give examples of systems that might be able to tolerate low levels of these qualities. Give examples of systems that can tolerate only high levels of these qualities.
8. Explain why the graph of Figure 7.13 can be interpreted to mean that if you find many errors in your code at compile time, you should throw away your code and write it again.
9. What are some possible explanations for the behavior of the graph in Figure 7.13?
10. A program is seeded with 25 errors. During testing, 18 errors are detected, 13 of which are seeded errors and 5 of which are indigenous errors. What is the Mills estimate of the number of indigenous errors remaining undetected in the program?

Figure 7.17 System Module Hierarchy for Exercise 6



11. You claim that your program is error free at a 95 percent confidence level. Your test plan calls for you to test until you find all seeded errors. With how many errors must you seed the program before testing in order to substantiate your claim? If for some reason you do not intend to find all seeded errors, how many seed errors does the Richards formula require?
12. Your testing group wants to use zero-failure testing on its 200,000-line software system. The customer requires the average number of failures to be no more than 0.01 per 1000 lines of code. You have already tested the system for 1000 hours, and you have had 250 failures. During the last 900 hours of testing, no failures have been reported. How many more hours of testing are needed?

CHAPTER 8

SYSTEM TESTING

In chapters 3, 4, 5, and 6, we discussed the use of reviews and walkthroughs to guarantee the quality of a system, and we emphasized the need for documentation to establish a common understanding of the requirements, design, and code. In the last chapter, we began to examine the testing phase of software development. We concentrated on finding errors within the program modules and in the interfaces among the modules. As shown in Figure 8.1, testing can be viewed as a two-step process. First, we must be sure that the implementation works as the programmers intend. Then, when the modules are combined into a working system, we insure that the system functions according to the prescriptions of the requirements documents.

Now we turn from a program-level view of testing to a system-level view. To test the software from a systems perspective, we verify that the software developed satisfies both our customers and ourselves. *System testing* verifies that a system solves the problem as defined by the requirements documents.

We begin our investigation of system testing by looking at the differences between system testing and program testing. We discover where system errors originate and use that information to outline the steps of system testing. As with program testing, we need test objectives and an integration plan. To tie system testing back to previous development and ahead to maintenance, we include tests to support configuration management.

Next, we discuss each step of system testing in detail: function testing, performance testing, acceptance testing, and installation testing. The test team is described in terms of its members and their roles, and we look at tools that aid test administration.

Several types of test documents are useful during testing, and we return to our Weaver Farm example to see what they look like.

8.1

PRINCIPLES OF SYSTEM TESTING

System testing is different from program testing (that is, from unit and integration testing). Let us see how.

Figure 8.1 The System Development Process

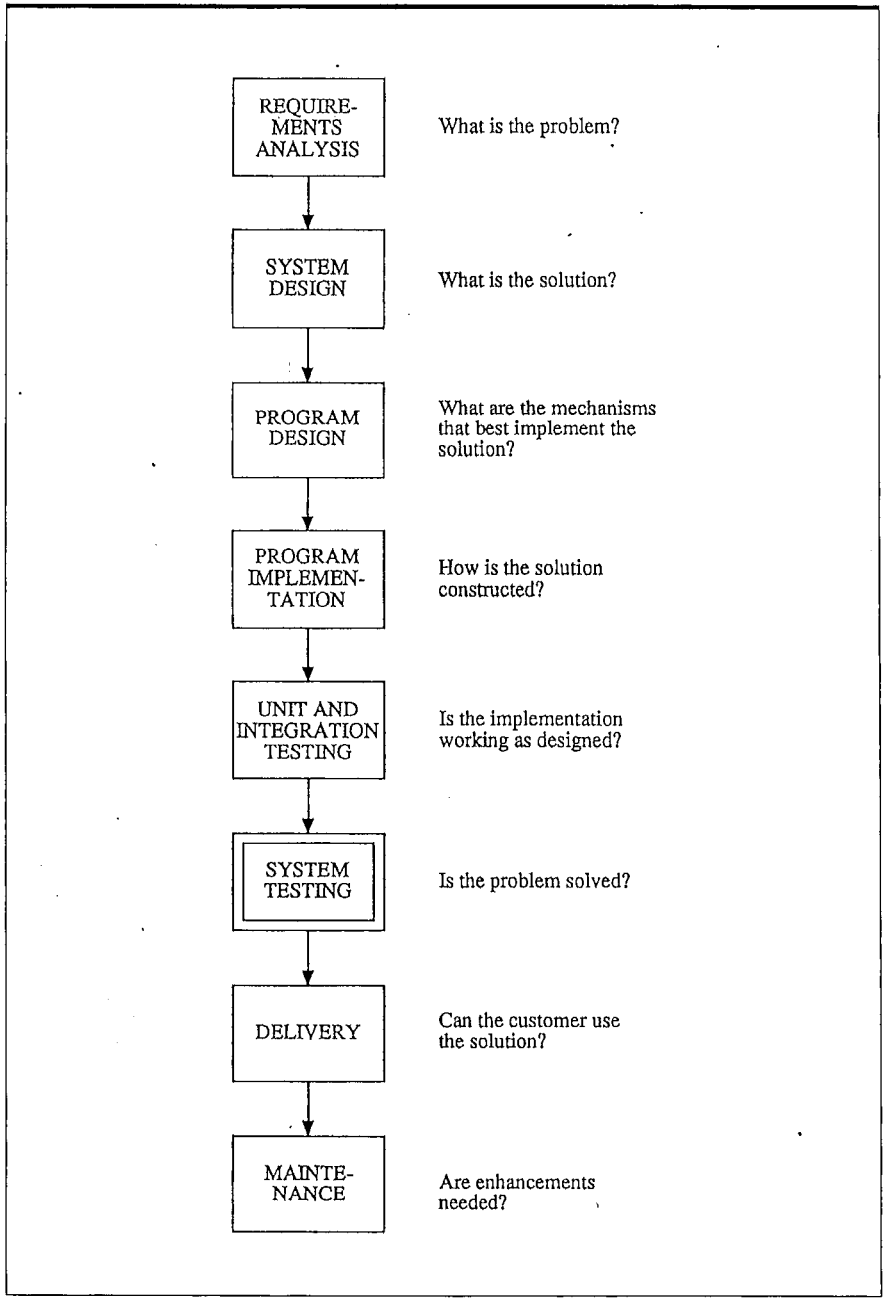
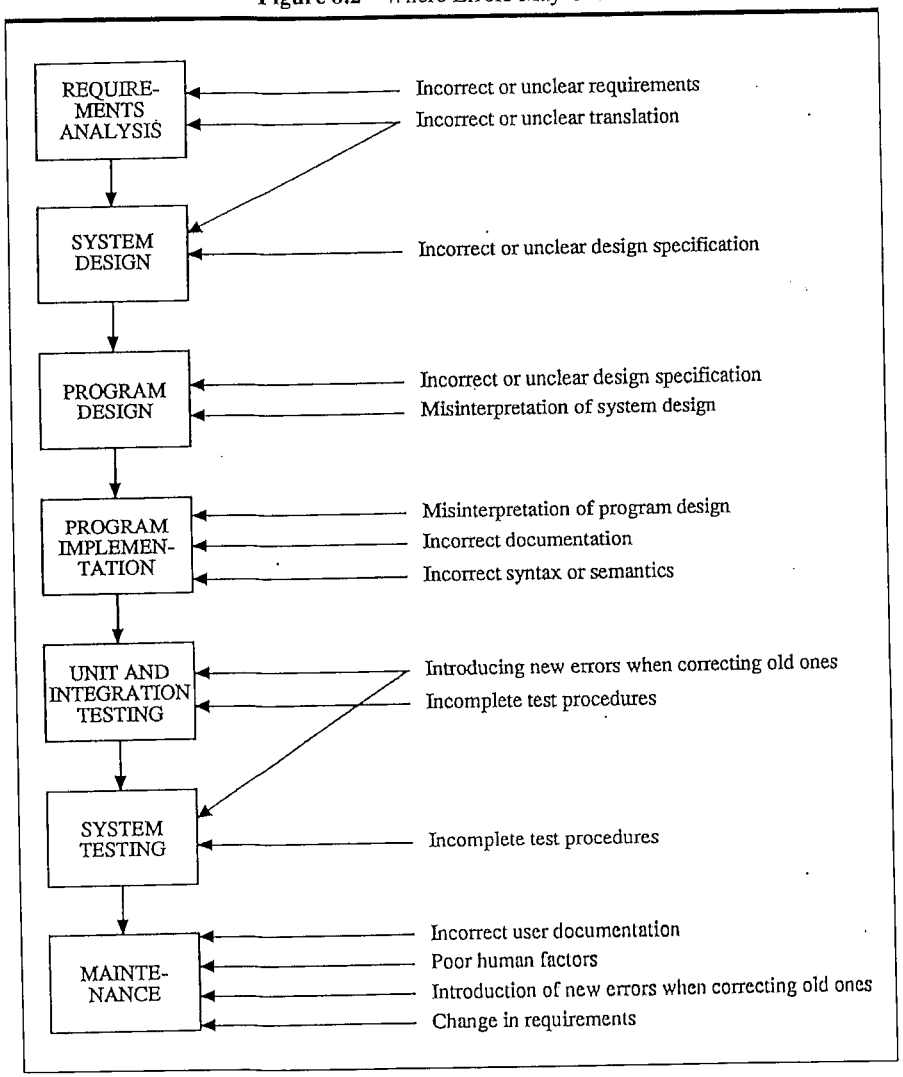


Figure 8.2 Where Errors May Occur



Where System Errors Originate

Errors can occur at any point during development. Figure 8.2 illustrates the likely causes of errors in each stage. Although we would like to find and correct errors as early as possible in development, system testing acknowledges that errors may still be present after integration testing.

Errors may have been introduced to the system early in development or recently, such as when correcting a previously found error. For example, defective software can result from errors made when determining requirements. Whether a requirement was ambiguous because the customer was unsure of a need or because we misinterpreted the customer's meaning, the result is the same: a system that does not work the way the customer wishes.

The same kind of communication mishaps can occur during system design. We may misinterpret a requirement and write an incorrect design specification. Or, we understand the requirement but may word the specification so poorly that those who subsequently read and use the design misunderstand it.

Similar events can lead to errors in the program design. Misinterpretations are common when the system design is translated into lower-level descriptions for program design specifications. Programmers are several levels removed from the initial discussions with customers about system goals and functionality. Having responsibility for one "tree" but not the "forest," programmers cannot be expected to spot design errors that have been perpetuated through the first steps of the development cycle. For this reason, requirements and design reviews are essential to assuring the quality of the resulting system.

The programmers and designers on our development team may also fail to use the proper syntax and semantics for recording their work. A compiler or assembler can catch some of these errors before a program is run, but they will not find errors where the form of a statement is correct but does not match the intention of the programmer or designer.

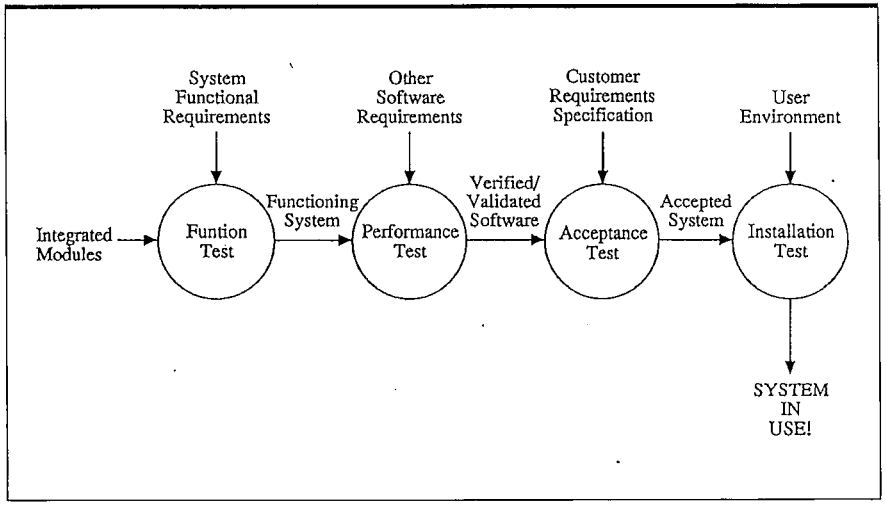
Once program module testing begins, errors may be added unintentionally in making changes to correct other errors. These errors are often very difficult to detect, because they may appear only when certain functions are exercised. If those functions have already been tested when a new error is inadvertently added, the new error may not be noticed until much later, when its source may not be clear.

For example, suppose you are testing modules A, B, and C. You test each separately. When you test all three together, you find that A passes a parameter to C incorrectly. In repairing A, you ensure that the parameter pass is correct, but you add code that sets a pointer incorrectly. Because you may not go back and test A independently again, you may not find evidence of the error until much later in testing—when it is not clear that A is the culprit.

In the same manner, errors may be introduced during maintenance. Enhancements to the original system require changes to the requirements, the system design, the program design, and the implementation itself, so that all of the errors discussed above can occur during maintenance. In addition, the system may not function properly because users do not understand how the system was designed to work. If the documentation is unclear or incorrect, an error may occur. Human factors, including user perception, play a large role in understanding the system and interpreting its messages and required input. Users who are not comfortable with the system may not exercise the functions of the system properly or to greatest advantage.

Test procedures should be thorough enough to exercise the system functions to everyone's satisfaction: user, customer, and developer. If the tests are incomplete, errors may remain undetected. As we have seen, the sooner we detect an error, the

Figure 8.3 Steps in System Testing



better; errors detected early on are easier and cheaper to fix. Thus, complete and early testing can help not only to detect errors quickly but also to isolate the causes more easily.

Figure 8.2 shows reasons for error, not evidence of them. Because testing aims to uncover as many errors as possible, it is concerned with where they may exist. Knowing how errors are created can give us clues about where to look when testing a system.

Steps in System Testing

There are several steps in testing a system:

1. Function testing
2. Performance testing
3. Acceptance testing
4. Installation testing

The steps are illustrated in Figure 8.3. Each step has a different focus, and the success of a step depends on whether its goal or objective has been met. Thus, it is helpful to review the purpose of each part of system testing.

Objectives of the Testing Steps. Initially, we test the functions performed by the system. We begin with a set of modules that have been tested individually and then together. The *function test* checks that the integrated system performs its

functional requirements as specified in the requirements. For example, a function test of a bank account package verifies that the package can correctly credit a deposit, enter a withdrawal, calculate interest, print the balance, and so on.

Once the test team is convinced that the functions work as specified, the *performance test* compares the integrated modules with the nonfunctional system requirements. These requirements include security, accuracy, speed, and reliability; they constrain the way in which the system functions are performed. For instance, a performance test of the bank account package evaluates the speed with which calculations are made, the precision of the computation, the security precautions required, and the response time to user inquiry. A *validated system* is the result of a performance test in the customer's actual working environment; if the testing is done in a simulated environment, the performance test results in a *verified system.*

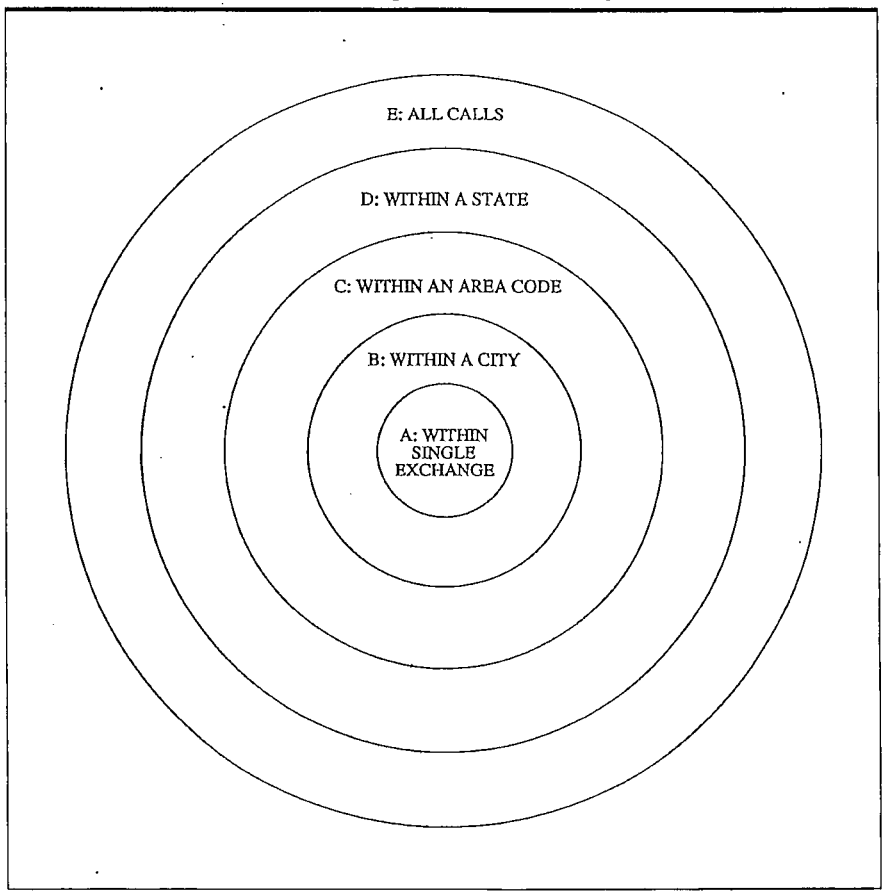
At this point, the validated or verified system operates the way the *designers* intend; it is their interpretation of the requirements specification. Next, we compare the system's performance with the *customer's* expectations by reviewing the requirements definition document. The *acceptance test* checks the system's characteristics to assure that they are in compliance with the defined requirements. If the system has not yet been installed in the user's environment, a final *installation test* is done to allow users to exercise system functions and document additional errors. For example, if a system is being designed for the Navy, all initial testing (that is, up to and including acceptance testing) may be performed at a headquarters site configured exactly as a ship might be. However, installation tests are be performed at sea on the actual ship or ships that will use the system.

Build or Integration Plan. Ideally, after program testing, you can view the collection of modules as a single entity. Then, during the first steps of system testing, the integrated collection is evaluated from a variety of perspectives, as previously described. However, large systems are sometimes unwieldy when tested as one enormous collection of modules. In fact, such systems are often candidates for phased development simply because they are much easier to build and test in smaller pieces. Thus, you may choose to perform *phased system testing*. We saw in chapter 1 that a system can be viewed as a nested set of levels or subsystems. Each level is responsible for performing at least the functions of those subsystems it contains. Similarly, we can divide the test system into a nested sequence of subsystems and perform the system test on one subsystem at a time.

The subsystem definitions are based on predetermined criteria. Usually, the basis for division is *functionality*. For example, a system that routes telephone calls might be divided into subsystems in the following way (as illustrated in Figure 8.4):

1. System A: Routing calls within a single exchange
2. System B: Routing calls within a single city
3. System C: Routing calls within a single area code
4. System D: Routing calls within a single state
5. System E: Routing any calls

Figure 8.4 Example of Nested Phone Systems



Each larger system contains all the systems preceding it. We begin our system testing by testing system A. When we have met the objectives of the function test step for system A, we proceed to a function test of system B. After a successful test of B, we continue to test systems C, D, and E in the same way. The result is a successful function test of the entire system, but incremental testing may have made error detection and correction much easier than having tested only system E. For example, an error discovered in the testing of system C is likely to be the result of something other than those modules dealing with state-related characteristics. Had the error been discovered only when testing system E, we could not eliminate the functions or modules that reside in systems E and D but not in system C.

Incremental testing requires careful planning. The test team must create a **build plan** or **integration plan** to define the subsystems to be tested and to describe how,

where, when, and by whom the tests will be conducted. Sometimes, a level or subsystem of a build plan is called a **spin** or **driver**. The spins are numbered, with the lowest level of spin called *spin 0*. For large systems, spin 0 is often a minimal system; it can even be only the operating system on a host computer.

For example, the build plan for the system depicted in Figure 8.4 may contain a schedule similar to Table 8.1.

Table 8.1 Build Plan for Phone System

Spin	Function	Test Start	Test End
0	Calls within exchange	1 Sept 90	15 Sept 90
1	Calls within city	30 Sept 90	15 Oct 90
2	Calls within area code	25 Oct 90	5 Nov 90
3	Calls within state	10 Nov 90	20 Nov 90
4	All possible calls	1 Dec 90	15 Dec 90

The build plan describes each spin by number, functional content, and testing schedule. Thus, as shown in Table 8.1, the first system to be tested will handle only those phone calls on a single exchange. The test will begin on September 1 and should end by September 15. After a successful test of spin 0, the next system to be tested has more functionality than the previous one. It is better to have a large number of spins so that the amount of change from one spin to the next is small; this gives the test team a greater amount of control over the testing process. If a test of spin N succeeds and an error arises in spin $N + 1$, then the most likely source of the error is related to the difference between spin N and spin $N + 1$: namely, the added functionality from one spin to the next. If the difference between two successive spins is small, then we have relatively few places to look for the defect's source.

The number of spins and their definitions depend primarily on our resources and those of the customer. These resources include not only hardware and software but also time and personnel availability. A minimal system is placed in the earliest spin, and subsequent spins are defined by integrating the most important or the most critical functions as early as is feasible. For example, suppose a system involves a network of computers arranged in a starlike fashion, as shown in Figure 8.5. The center of the star is a large computer that receives messages from a set of smaller computers whose main purpose is to capture and transmit data for processing. One of the major functions of the central computer is to translate and assimilate the messages from the outlying computers. Since this function is critical to the other functions of the system, it should be included in an early spin. In fact, we may begin to define the spins in the following way:

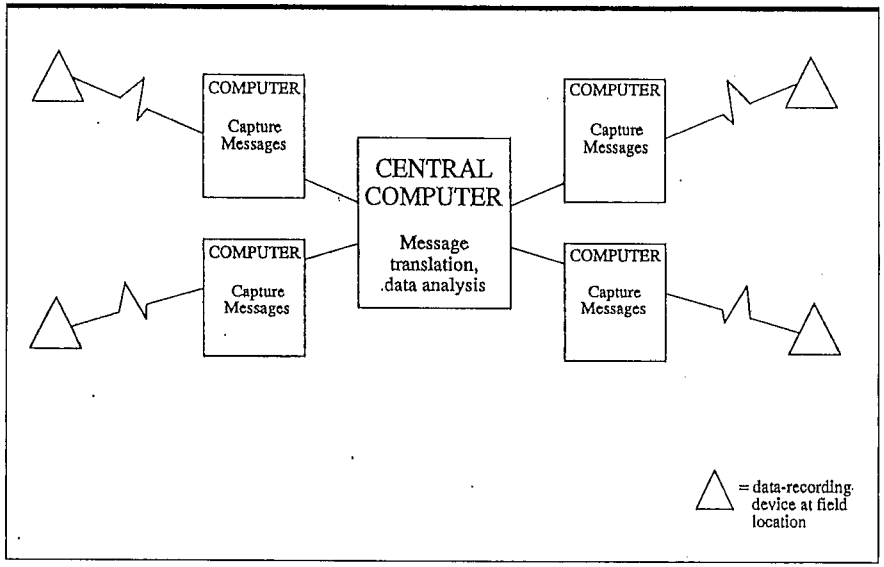
Spin 0: Test the central computer's general functions

Spin 1: Test the message translation function on the central computer

Spin 2: Test the message assimilation function on the central computer

Spin 3: Test each outlying computer in stand-alone mode

Figure 8.5 Message Processing System



Spin 4: Test each outlying computer's message-sending function

Spin 5: Test the central computer's message-receiving function

and so on.

The spin definitions also depend on the ability of the system components to operate in stand-alone mode. It may be harder to simulate a missing piece of a system than to incorporate the piece in a spin, since interdependencies among parts of the system sometimes require as much simulation code as actual code. Remember that our goal is to test the system. The time and effort needed to build and use test tools might be better spent in testing the actual system. This trade-off is similar to the one involved in choosing a test philosophy during program testing: developing many stubs and drivers may require as much time during program testing as testing the original modules they simulate.

Configuration Management

As we saw in the last chapter, the purpose of testing is to identify errors, not to correct them. However, it is natural to want to find the source of and then correct errors as soon as possible after discovery. Otherwise, the test team is unable to judge if the system is functioning properly, and the presence of some errors might halt further testing. Thus, any test plan must include a set of *guidelines* for error correction.

Defects may result from causes in any phase of system development. Correcting an error may mean modifying several lines of code, updating program documentation, revising the program or system design, or altering requirements. The configuration management team is responsible for assuring that each spin or build is correct and stable before it is released for use. Thus, configuration management ensures that changes are made accurately and promptly. Accuracy is critical because we want to avoid generating new errors while correcting existing ones. Similarly, promptness is important, because error correction is proceeding at the same time that the test team is searching for new errors. Thus, those who are trying to repair defects in the system should work with documentation that reflects the current state of the system.

Regression Testing. Regression testing identifies new errors that may be introduced as current ones are being corrected. A **regression test** is a test applied to a new version of a system to verify that it still performs the same functions in the same manner as an older version.

For example, suppose that the functional test for system C was successful, and testing is proceeding on system D, where D has all the functionality of C plus some new functions. You request that several lines of code be changed in system D to repair an error located in an earlier test; the code must be changed now so that the testing of system D can continue. If the test team is following a policy of regression testing, the testing will include these steps:

1. Insertion of your new code
2. Testing of the functions known to be affected by the new code
3. Testing of the essential functions of system C to verify that they still work properly
4. Continuation of the function testing of system D

These steps insure that adding new code has not negated the effects of the previous tests.

Often, the regression test involves the reuse of the most important test cases from the previous level's test; if you specify regression testing in your test plan, you should also explain which test cases are to be used again.

The Issue of Control. The configuration management team works closely with the test team to control all aspects of testing. Any change proposed to any part of the system is approved first by the configuration management team. The change is entered in all appropriate *documentation*, and the team notifies all who may be affected. For example, if a test results in modification of a requirement, changes are also likely to be needed in the requirements specification, the system design, the program design, the code, all relevant documentation, and even the test plan itself. Thus, altering one part of the system may affect everyone who is working on the system's development.

One method for insuring that all project members are working with the most up-to-date documentation is to keep documents online. By viewing documents on a screen and updating them immediately, we avoid the time lag usually caused by having to print and distribute new or revised pages. However, the configuration management team still maintains some degree of control to make sure that changes to documents mirror changes to design and code.

Libraries of documentation are not the only aspects of the system that must be controlled. If system development is *phased*, a production system runs in parallel with a development system. A **production system** is a version of the system that has been tested and performs according to only a subset of the customer's requirements. The next version, with more features, is developed while users operate the production system. This **development system** is built and tested; when testing is complete, the development system replaces the production system to become the new production system.

For example, suppose a power plant is automating the functions performed in the control room. The power plant operators have been trained to do everything manually and are uneasy about working with the computer, so we decide to build a phased system. The first phase is almost identical to the manual system but allows the plant operators to do some automated record keeping. The second phase adds several automated functions to the first phase, but half of the control room functions are still manual. Successive phases continue to automate selected functions, building on the previous phases until all functions are automated. By building up the automated system in this way, we allow the plant operators to slowly become accustomed to and feel comfortable with the new system.

At any point during the phased development, the plant operators are using the fully tested production system. At the same time, we are working on the next phase, testing the *development* system. When the development system is completely tested and ready for use by the plant operators, it becomes the production system (that is, it is used by the plant operators) and we move on to the next phase. We add functions to the current production or operational system, and the copy of the system we work with is our new development system.

While a system is in production, errors may occur and be reported to us. Thus, a development system often serves two purposes: it *adds the functionality* of the next phase and *corrects the errors* found in previous versions. A development system can therefore involve adding new modules as well as changing existing ones. However, this procedure allows errors to be introduced into modules that have already been tested. When we write a build plan and test plans, we should address this problem and consider the need for controlling changes implemented from one version to the next. Regression testing can make sure that the development system performs at least as well as the current production system. However, records must be kept of the exact changes made to the code from one version to the next so that we can trace problems to their source. For example, if a user on the production system reports a problem, you must know what version of code is being used. The code may differ dramatically from one version to another. If you work with the wrong listing, you may never locate the source of the error. Worse yet, you may

think you have located the source and make a change that introduces a new error while not fixing the old error.

When a new production system is ready for operation, users are trained to exercise the new functions or change the way they worked with the previous version. Should an error be discovered, we may decide to return to a previous version until the error is corrected. The configuration management team must be able to "roll back" the changes and present the users with their old system.

Similar issues arise when the final system is being maintained for the customer. We will address these issues in more depth in chapter 10.

8.2

FUNCTION TESTING

We begin system testing with function testing. While previous steps concentrated on modules and their interactions, this step of testing ignores system structure and focuses on *functionality*. Our approach from now on is more a closed box than an open one. We need not know what modules are being exercised; rather, we need to know what the system is *supposed* to do. Thus, to perform function testing, we must know the system's functional requirements.

Each function can be associated with those system components that accomplish it. For some functions, the parts may comprise the entire system. The set of module actions associated with a function is called a **thread**; hence, function testing is sometimes referred to as **thread testing**.

Logically, it should be easier to find the cause of an error in a small set of modules than in a large one. Thus, ease of testing calls for choosing carefully the order in which functions are tested. Functions may be defined in a *nested* manner, just as spins are defined in levels. For example, suppose a requirement specifies that a water monitoring system is to identify large changes in four characteristics: dissolved oxygen, temperature, acidity, and radioactivity. The requirements specification may treat change acknowledgment as one of many functions of the overall system. However, for testing we may want to view the monitoring as four separate functions:

- Acknowledging change in dissolved oxygen
- Acknowledging change in temperature
- Acknowledging change in acidity
- Acknowledging change in radioactivity

Then, we test each one individually.

Effective function tests should have a high probability of detecting an error. We use the same guidelines for function testing that we discussed in chapter 7 for unit testing. We can summarize the guidelines as:

1. Having a high probability of detecting an error
2. Using a test team independent of the designers and programmers
3. Knowing what the expected actions and output are
4. Testing both valid and invalid input
5. Never modifying the system being tested just to make testing easier
6. Knowing when the tests should stop

Function testing is performed in a carefully controlled situation. Moreover, since we are testing one function at a time, function testing can actually begin before the entire system is constructed, if need be.

Function testing compares the actual performance of the system with its requirements, so the test cases for function testing are developed from the requirements document. In the next section, we examine a method for analyzing the requirements to generate the set of test cases.

Cause-and-Effect Graphs

Testing would be easier if we could automatically generate test cases from the requirements. Work has been done at IBM ([ELM73] and [ELM74]) to convert the natural language specifications of a requirements definition document to a formal specification that can be used to enumerate test cases for functional testing. The test cases that result are not redundant; that is, one test case does not repeat the testing of functions that have already been tested by another case. In addition, the process finds incomplete and ambiguous aspects of the requirements definition if any still exist.

The process examines the semantics of the requirements definition and restates the requirements as logical relationships between inputs and outputs or between inputs and transformations. The inputs are called *causes*, and the outputs and transformations are *effects*. The result is a Boolean graph reflecting these relationships called a **cause-and-effect graph**.

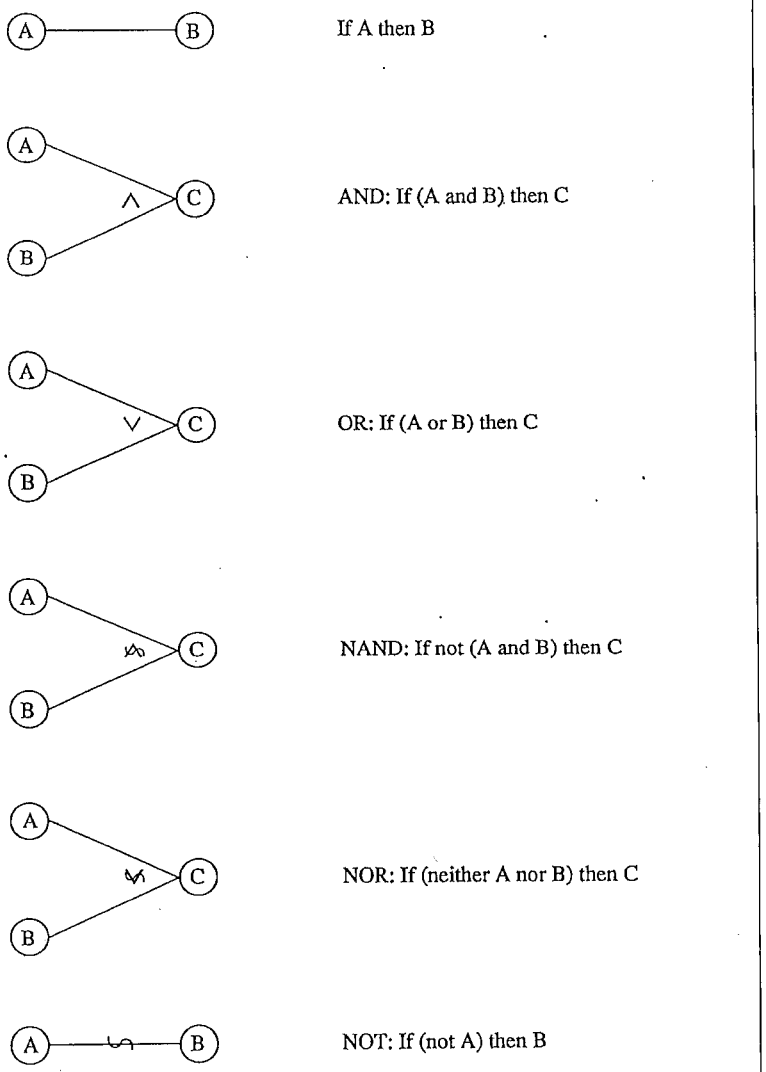
We add information to the initial graph to indicate rules of syntax and to reflect environmental constraints. Then, we convert the graph to a decision table. Each column of the decision table corresponds to a test case for functional testing.

There are several steps in the creation of a cause-and-effect graph. First, the requirements are separated so that each requirement describes a single function. Then, all causes and effects are described. The numbered causes and effects become nodes of the graph. Placing causes on the left-hand side of the drawing and effects on the right, we draw the logical relationships depicted in the graph by using the notation shown in Figure 8.6. Extra nodes can be defined to simplify the graph.

An example will show us how such a graph is built. Suppose we are testing a water level monitoring system. The requirements definition for one of the system functions reads as follows:

> The system sends a message to the dam operator about the safety of the lake level.

Figure 8.6 Cause-and-Effect Relationships



Corresponding to this requirement is a design description:

INPUT: The syntax of the function is

LEVEL(A, B)

where A is the height in feet of the water behind the dam and B is the number of inches of rain in the last twenty-four-hour period.

PROCESSING: The function calculates whether the water level is within a safe range, is too high, or is too low.

OUTPUT: The screen shows one of the following messages:

1. "LEVEL = SAFE" when result is safe or low
2. "LEVEL = HIGH" when result is high
3. "INVALID SYNTAX"

depending on the result of the calculation.

We can separate these requirements into five causes:

Cause 1. The first five characters of the command are 'LEVEL'.

Cause 2. The command contains exactly two parameters separated by a comma and enclosed in parentheses.

Cause 3. The parameters A and B are real numbers such that the water level is calculated to be LOW.

Cause 4. The parameters A and B are real numbers such that the water level is calculated to be SAFE.

Cause 5. The parameters A and B are real numbers such that the water level is calculated to be HIGH.

We can also describe three effects.

Effect E1. The message "LEVEL = SAFE" is displayed on the screen.

Effect E2. The message "LEVEL = HIGH" is displayed on the screen.

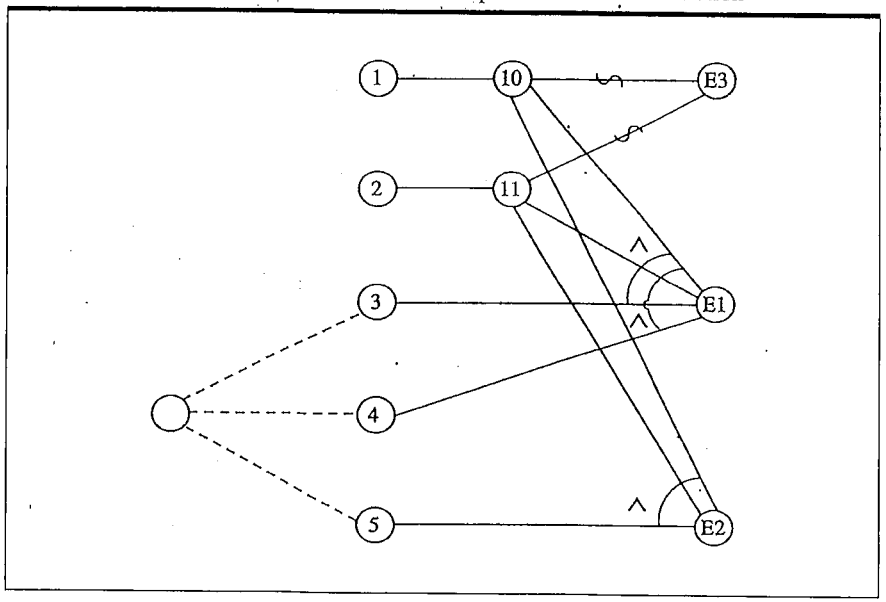
Effect E3. The message "INVALID SYNTAX" is printed out.

These become the nodes of our graph. However, the function includes a check on the parameters to be sure that they are passed properly. To reflect this, we establish two intermediate nodes:

Node 10. The command is syntactically valid.

Node 11. The operands are syntactically valid.

Figure 8.7 Cause-and-Effect Graph for the LEVEL Function



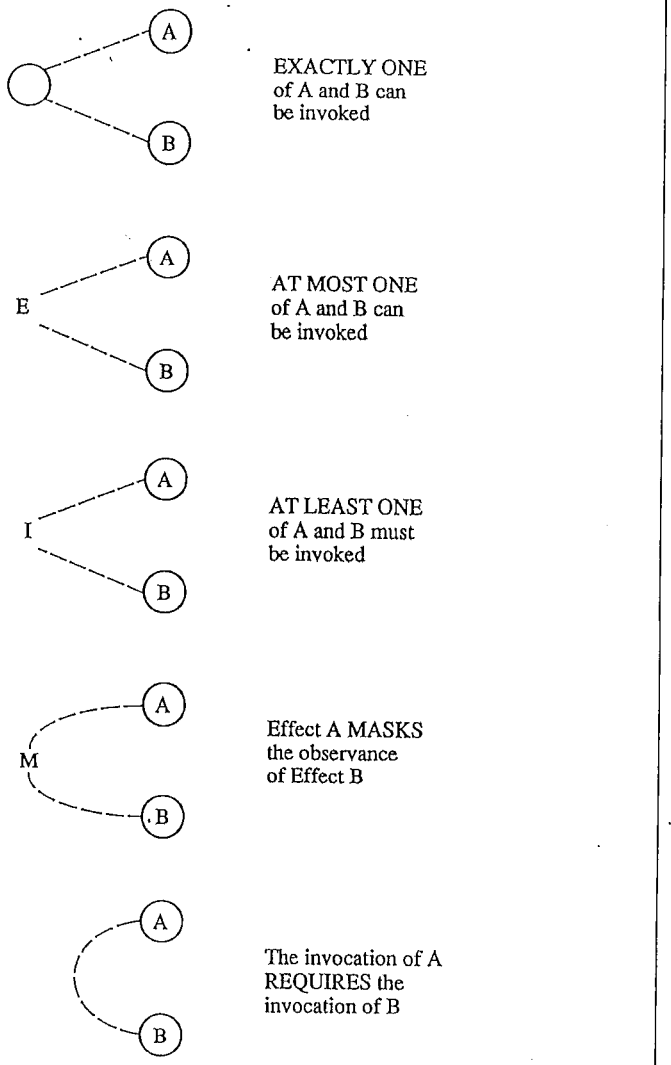
We can draw the relationships between cause and effect as shown in Figure 8.7. Notice that there are dashed lines to the left of the effects. These lines mean that exactly one effect can result. Other notations can be made on cause-and-effect graphs to provide additional information. Figure 8.8 illustrates some of the possibilities. Thus, by looking at the graph, we can tell if

- At most one of a set of conditions can be invoked
- At least one of a set of conditions must be invoked
- Exactly one of a set of conditions can be invoked
- One effect masks the observance of another effect
- Invocation of one effect requires the invocation of another

At this point, we are ready to define a decision table using the information from the cause-and-effect graph. We put a row in the table for each cause or effect. Thus, in our example, our decision table needs five rows for the causes and three rows for the effects. The columns of the decision table correspond to the test cases. We define the columns by examining each effect and listing all combinations of causes that can lead to that effect.

In our LEVEL example, we can determine the number of columns in the decision table by examining the lines flowing into the effect nodes of the cause-and-effect graph. We see in Figure 8.7 that there are two separate lines flowing into E3;

Figure 8.8 Additional Cause-and-Effect Notation



each corresponds to a column. There are four lines flowing into E1, but only two combinations yield the effect. Each of the combinations is a column in the table.

Finally, only one combination of lines results in effect E2, so we have our fifth column.

Each column of the decision table represents a set of *states* of causes and effects. We keep track of the states of other conditions when a particular combination is invoked. We indicate the condition of the cause by placing an I in the table when the cause is invoked or true, or an S if the cause is suppressed or false. If we do not care whether the cause is invoked or suppressed, we can use an X to mark the "don't-care" state. Finally, we indicate whether a particular effect is absent (A) or present (P).

For testing the LEVEL function, the five columns in Table 8.2 display the relationship between invocation of causes and the resultant effects. If causes 1 and 2 are true (that is, the command and parameters are valid), then the effect depends on whether causes 3, 4, or 5 are true. If cause 1 is true but cause 2 is false, the effect is already determined, and we don't care about the state of causes 3, 4, or 5. Similarly, if cause 1 is false, we no longer care about the states of other causes.

Note that theoretically we could have generated 32 test cases: five causes in each of two states yields 2^5 possibilities. Thus, the use of a cause-and-effect graph substantially decreases the number of test cases we must consider.

Table 8.2 Decision Table for Cause-and-Effect Graph

		Tests				
		1	2	3	4	5
Causes	1	I	I	I	I	S
	2	I	I	I	X	I
	3	I	S	S	X	X
	4	S	I	S	X	X
	5	S	S	I	X	X
Effects	E1	P	P	A	A	A
	E2	A	A	P	A	A
	E3	A	A	A	P	P

S = Suppressed I = Invoked A = Absent P = Present X = Don't Care

In general, we can reduce the number of test cases even more by using our knowledge of the causes to eliminate certain other combinations. For example, if the number of test cases is high, we may assign a priority to each combination of causes. Then, we can eliminate the combinations of low priority. Similarly, we can eliminate those combinations that are unlikely to occur or for which testing is not economically justifiable.

In addition to reducing the number of test cases to consider, cause-and-effect graphs help us predict the possible outcomes of exercising the system. At the same time, the graphs find unintended side effects for certain combinations of causes. However, cause-and-effect graphing has several limitations. The graphs are not practical for systems that include time delays, iterations, or loops where the system reacts to feedback from some of its processes to perform other processes.

8.3

PERFORMANCE TESTING

When the system performs the functions required by the requirements definition, we turn to the way in which those functions are performed. Thus, system performance is measured against the objectives set by the customer. For example, function testing may have demonstrated that a test system can calculate the mean and standard deviation of a set of numbers. *Performance testing* examines how well the calculation is done; response to user commands, accuracy of the result, and accessibility of the data are checked against the customer's performance prescriptions.

Test Case Coverage

Performance testing is based on the requirements, so the types of performance tests used are determined by the requirements. The test team chooses from among several kinds of tests.

1. **Stress tests** evaluate the system when stressed to its limits over a short period of time. If the requirements state that a system is to handle up to a specified number of devices or users, a stress test evaluates how the system performs when all of those devices or users are active simultaneously. This is especially important for systems that usually operate below maximum but are severely stressed at certain times of peak demand.
2. **Volume tests** address the handling of large amounts of data in the system. For example, the system should handle large data sets according to the requirements. In a volume test, we look at whether data structures (such as queues and stacks) have been defined to be large enough to handle all possible situations. In addition, fields, records, and files are checked to see if their sizes can accommodate all expected data. The team investigates the system's response to filled data sets, too.
3. **Configuration tests** analyze the various software and hardware configurations specified in the requirements. Sometimes a system is built to serve a variety of audiences, and the system is really a spectrum of configurations. For instance, a minimal system may be defined to serve a single user, and other configurations build on the minimal configuration to serve additional users. A configuration test evaluates all possible configurations to make sure that each satisfies the requirements.
4. **Compatibility tests** are needed when a system interfaces with other systems. We find out whether the interface functions perform according to the requirements. For instance, if the system is to communicate with a large data base system to retrieve information, a compatibility test examines the speed and accuracy of data retrieval.

5. **Regression tests** are required when the system being tested is replacing an existing system. The regression tests guarantee that the new system's performance is at least as good as that of the old. Regression tests are always used during a phased development.
6. **Security tests** insure that the security requirements are met. We test access to the system itself, to functions, and to data to see if requested restrictions have been implemented properly.
7. **Timing tests** evaluate the requirements dealing with time to respond to a user and time to perform a function. If a transaction must take place within a specified time, the test performs that transaction and verifies that the requirements are met. Timing tests are usually done in concert with stress tests to see if the timing requirements are met even when the system is extremely active.
8. **Environmental tests** look at the system's ability to perform at the installation site. If the requirements include tolerances for heat, humidity, motion, chemical presence, moisture, portability, electrical or magnetic fields, disruption of power, or any other environmental characteristic of the site, then our tests guarantee the system's proper performance under these conditions.
9. **Quality tests** evaluate the reliability, maintainability, and availability of the system. These tests address the characteristics described in chapter 7 and include requirements for mean time between failures and mean time to repair. In addition, we investigate requirements relating to detecting and correcting errors. Quality tests are sometimes very difficult to administer. For example, if a requirement specifies a long mean time between failures, it may be infeasible to let the system run long enough to verify the mean.
10. **Recovery tests** address response to the presence of errors or to the loss of data, devices, or power. We subject the system to a loss of system resources and see if it recovers properly.
11. **Maintenance tests** address the need for diagnostic tools and procedures to help in finding the source of errors. We may be required to supply diagnostic programs, memory maps, traces of transactions, diagrams of circuitry, and other aids. We verify that the aids exist and that they function properly.
12. **Documentation tests** insure that we have written the required documents. Thus, if user manuals, maintenance guides, and technical documents are needed, we verify that these materials exist and that the information they contain is consistent, accurate, and easy to read. Moreover, sometimes requirements specify the format and audience of the documentation; we evaluate the documents for compliance.
13. **Human factors tests** investigate requirements dealing with the user interface to the system. We examine display screens, messages, report formats, and other aspects that may relate to ease of use. In addition, operator and user procedures are checked to see if they conform to ease of use requirements.

Many of these tests are much more difficult to administer than the function tests. We stressed in chapter 3 the need for requirements to be explicit and detailed. The quality of the requirements is often reflected in the ease of performance testing. Unless a requirement is clear, it is hard for the test team to know when the requirement is satisfied. Indeed, it may even be difficult to know *how* to administer a test because it may not be clear when the test is successful.

8.4

ACCEPTANCE TESTING

When function and performance tests are complete, we are convinced that the system meets all requirements specified during the initial stages of system development. The next step is to ask the customer and users if they concur. Until now, we as developer have designed the test cases and administered all tests. Now the customer leads testing and defines the cases to be tested.

Types of Acceptance Tests

There are three ways the customer can evaluate the system. In a **benchmark test**, the customer prepares a set of test cases that represent typical conditions under which the system will operate when actually installed. The customer submits the test cases and evaluates the system's performance for each. Benchmark tests are performed with actual users or a special test team exercising system functions. In either case, the testers are familiar with the requirements and able to evaluate the actual performance.

Benchmark tests are commonly used when a customer has special requirements. Two or more development teams are asked to produce systems according to specification; one system will be chosen for purchase, based on the success of the benchmark tests. For example, a customer may ask two communications companies to install a voice and data network. Each system is benchmarked. Both systems may meet a requirement, but one may be faster or easier to use than the other. The customer decides to purchase one rather than the other based on how the systems met the benchmark criteria.

A **pilot test** installs the system on an experimental basis. Users exercise the system as if it had been permanently installed. Whereas benchmark tests include a set of *special test cases* that the users apply, pilot tests rely on the *everyday working* of the system to test all functions. The customer often prepares a suggested list of functions that each user tries to incorporate in typical daily procedures. However, a pilot test is much less formal and structured than a benchmark test.

Sometimes we test a system with users from within our own organization or company before releasing the system to the customer; we "pilot" the system before

the customer runs the real pilot test. Our in-house test is called an **alpha test** and the customer's pilot a **beta test**. This approach is common when systems are to be released to a wide variety of customers. For example, an office automation system or a new version of an operating system may be alpha tested at our own office and then beta tested using a specially selected group of customer sites. We try to choose as beta test sites customers who represent all kinds of customer organizations.

Even if a system is being developed for just one customer, a pilot test usually involves only a small subset of the customer's potential users. We choose the users for the pilot so that their activities represent those of most others who will use the system later. One location or one organization may be chosen to test the system, rather than allowing all intended users to have access.

If a new system is replacing an existing one or is part of a phased development, a third kind of testing can be used for acceptance. In **parallel testing**, the new system operates in parallel with the previous version. The users gradually become accustomed to the new system but continue to use the old one to duplicate the new system's functions. This gradual transition allows users to compare and contrast the new system with the old. It also allows skeptical users to build their confidence in the new system by comparing the results obtained by both. In a sense, parallel testing incorporates a user-administered combination of compatibility and function testing.

Results of Acceptance Tests

The type of system being tested and the preferences of the customer determine the choice of acceptance test. In fact, a combination of some or all of the approaches can be used. Tests by users sometimes find places where the customer's expectations, as stated in the requirements, do not match what we have implemented. In other words, acceptance testing is the customer's chance to verify that what was wanted is what was built. If the customer is satisfied, the system is then accepted as stated in the contract.

In reality, acceptance testing uncovers more than requirements discrepancies. The acceptance test also allows the customer to determine what is really wanted, whether specified in the requirements or not. Remember that the requirements analysis stage of development gives the customer an opportunity to explain to us what problem needs a solution. The system design is our proposed solution to the customer's problem. Until the customer and the users actually work with a system as a proposed solution, they may not really know whether the problem is indeed solved.

We have seen in previous chapters that rapid prototyping may be used to help the customer understand more about the solution before the entire system is implemented. However, prototypes are often impractical or too expensive to build. Moreover, when building large systems, there is sometimes a long time between the initial specification of requirements and the first viewing of even part of a system. During this time, the customer's needs may change in some way. For instance, the nature of the customer's business may change, affecting the nature of the original problem. Thus, changes in requirements may be needed not only because they were improperly specified at the beginning of development but also because the customer may decide that a different solution is needed.

After acceptance testing, the customer tells us which requirements are not satisfied and which must be deleted, amended, or added because of changing needs. We will see in chapter 10 how the configuration management team identifies these changes in requirements and records the resulting modifications to design, implementation, and testing.

8.5 INSTALLATION TESTING

The final round of testing involves installing the system at user sites. If acceptance testing has been performed on site, installation testing may not be needed. However, if acceptance testing conditions were not the same as actual site conditions, additional testing is necessary. To begin installation testing, we configure the system to the user environment. We attach the proper number and kind of devices to the main processor and establish communications with other systems. We allocate files and assign access to appropriate functions and data.

Unlike acceptance tests, installation tests require us to work with the customer to determine what tests are needed on site. Regression tests may be administered to verify that the system has been installed properly and works "in the field" as it did when tested previously. The test cases assure the customer that the system is complete and that all necessary files and devices are present. The tests focus on two things: *completeness* of the installed system and *verification* of any functional or nonfunctional characteristics that may be affected by site conditions. We can conduct the tests ourselves, or we can allow the customer to do so. In either case, when the customer is satisfied with the results, testing is complete and the system is formally delivered.

8.6 TEST TOOLS

Several tools are available to help us administer the variety of tests involved in system testing. Most are automated and capture data that can be of use in evaluating a system's performance.

Simulation allows us to concentrate on evaluating one part of a system while portraying the characteristics of other parts. A **simulator** presents to a system all characteristics of a device or system without actually having the device or system

available. Just as a flight simulator allows you to learn to fly without an actual airplane, a device simulator allows you to control a device even when the device is not present. Thus, if a communications controller is part of your system but is being developed by another company, you can simulate the behavior of the controller and continue your testing.

A device simulator is particularly useful if a special device is located on site but testing is being done at another location. For example, if the system you are building controls a ship's navigation system, you need not have a ship to perform the actual tests. Sometimes a device simulator is more helpful than the device itself, since the simulator can store data indicating the device's state during the various stages of a test. Then the simulator reports on its state when an error occurs, possibly helping you find the source of the error.

Simulators are also used to look like other systems with which the test system must interface. If messages are communicated or a data base is accessed, a simulator provides the necessary information for testing without duplicating the entire other system. The simulator also helps with stress and volume testing, since it can be programmed to load the system with substantial amounts of data, requests, or users.

In general, simulators give you control over the test conditions. This control allows you to perform tests that might otherwise be dangerous or impossible. For example, the test of a missile guidance system can be made much simpler and safer by using simulators.

A **monitor** is a device that captures data passing from one device or process to another. For example, an input/output monitor can be placed between two processors to store any data flowing between them. Additional information (such as interrupts) can also be stored. We can use a monitor to save a "snapshot" of the conditions before and after an error occurs. Again, the snapshot helps track down the source of the error and verifies proper performance.

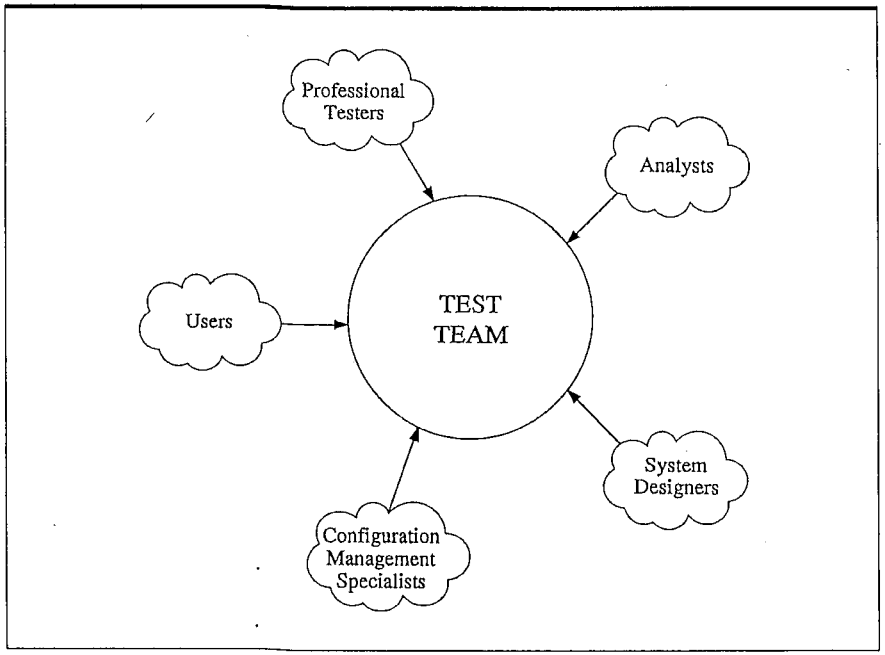
An **analyzer** goes one step beyond a monitor; it not only captures data but also evaluates them according to some prescribed criteria. For example, a *test coverage analyzer* records the number of each statement executed during a test step and notifies us if certain routines or statements are not exercised. Similarly, a *timing analyzer* works with predefined areas of memory or code and tracks the amount of time spent in each area as the system functions are performed. This tracking can be useful during performance testing when timing requirements are checked.

As we shall see later in this chapter, there are also tools for generating test cases and tracking test results.

8.7 TEST TEAM

We have primary responsibility for function and performance testing, while the customer plays a large role in acceptance and installation tests. However, the test team for all tests is drawn from both staffs, as shown in Figure 8.9. No programmers

Figure 8.9 Test Team Members



from this project are involved in system testing; they are too familiar with the structure and intention of the implementation, and they may have difficulty recognizing the differences between implementation and required function.

Professional Testers

Thus, the test team is independent of the implementation staff. Ideally, some test team members are already experienced as testers. Usually, these "professional testers" are former analysts, programmers, and designers who now devote all their time to testing systems. The testers are familiar not only with the specification of the system but also with testing methods and tools.

Professional testers *organize and run* the tests. They are involved from the beginning, designing test plans and test cases as the project progresses. The professional testers work with the configuration management team to provide documentation and other mechanisms for tying tests to the requirements, design modules, and implementation modules.

Analysts

The professional testers focus on test development, methods, and procedures. Because the testers may not be as well versed in the particulars of the requirements

as those who wrote them, the test team includes additional people who are familiar with the requirements. Analysts who were involved in the original requirements definition and specification are useful in testing because they are *familiar with the problem* as defined by the customer. Much of system testing compares the new system to its original requirements, and the analysts have a good feeling for the needs and goals of the customer. Since they have worked with the designers to define a solution, analysts have some idea of how the system should work to solve the problem.

System Designers

System designers on the test team add the perspective of *intent*. The designers understand what we proposed as a solution to the customer's problem; they know the constraints imposed on the solution by the overall design. The designers also know how the system is divided into functional subsystems and understand how the system is *supposed* to work. When designing test cases and assuring test coverage, the test team calls on the designers for help in listing all possibilities.

Configuration Management Specialists

Because tests and test cases are tied directly to requirements and design, a configuration management representative is on the test team. As errors are discovered and changes requested, the configuration management specialist arranges for the changes to be reflected in the documentation, requirements, design, implementation, or anything else affected by the change. In fact, changes to correct an error may result in modifications to other test cases or to a large part of the test plan. The configuration management specialist implements these changes and coordinates the revision of tests.

Users

Finally, the test team includes users. They are best qualified to evaluate issues dealing with appropriateness of audience, ease of use, and other human factors.

Sometimes, users have little voice in the early stages of the project. Customer representatives who participate during requirements analysis may not plan to use the system but have jobs related to those who will. For instance, the representatives may be managers of those who will use the system or technical representatives who have discovered a problem that indirectly relates to their work. However, these representatives may be so removed from the actual problem that the requirements description is inaccurate or incomplete. The customer may not be aware of the need to redefine or add requirements.

Therefore, users of the proposed system are essential if they were not present when the system requirements were first defined by the customer. A user is likely to be intimately familiar with the customer's problem because of daily exposure to it, and can be invaluable in evaluating the system to verify that it solves the problem.

8.8 TEST DOCUMENTATION

Testing can be complex and difficult. The system's *software* and *hardware* can contribute to the difficulty, as can the *procedures* involved in using the system. In addition, a distributed or real-time system requires great care in *tracing* and *timing* data and processes to draw conclusions about performance. Finally, when systems are large, the large number of people involved in development and testing can make *coordination* difficult. To control the complexity and difficulty of testing, we use complete and carefully designed test documentation.

Several types of documentation are needed. A **test plan** describes the system itself and the plan for exercising all functions and characteristics. A **test specification and evaluation** details each test and defines the criteria for evaluating each feature addressed by the test. Then, a **test description** presents the test data and procedures for the individual test components. Finally, the **test analysis report** describes the results of each test. Figure 8.10 displays the relationship of the documents to the testing process.

Test Plans

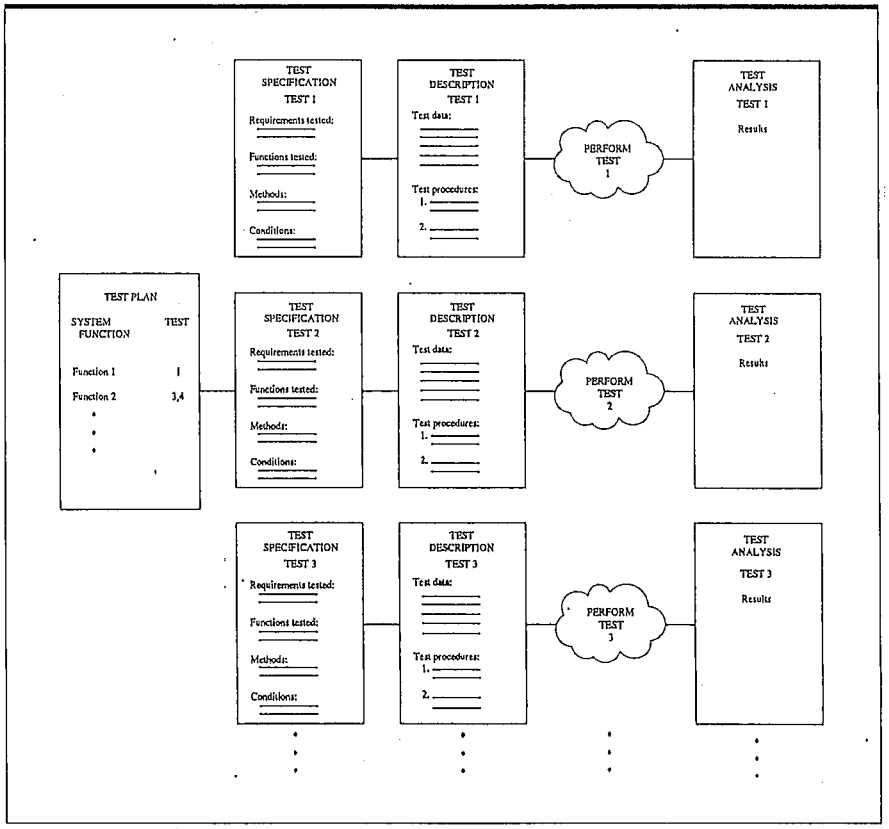
In chapter 7, we discussed the role of the test plan in laying out the pattern of testing for the entire testing stage. Now we look at how a test plan can be used to *direct* system testing.

Figure 8.11 illustrates the components of a test plan. The plan begins by stating its *objectives*. These objectives should:

1. Guide the management of testing
2. Guide the technical effort required during testing
3. Establish planning and scheduling of tests, including specification of equipment needed, organizational requirements, test methods, anticipated outcomes, and user orientation
4. Explain the nature and extent of each test
5. Explain the way in which the tests will lead to a complete evaluation of the system function and performance
6. Document test input, specific test procedures, and expected output

Next, the test plan *references* the other *major documents* produced during project development. In particular, the plan explains the relationships among the requirements documents, the design documents, the implementation documents, and the test procedures. For example, there may be a numbering scheme that ties together all documents; requirement 4.9 may be reflected in module 5.6 of the design and be tested by test procedure 8.1.

Figure 8.10 Test Documentation

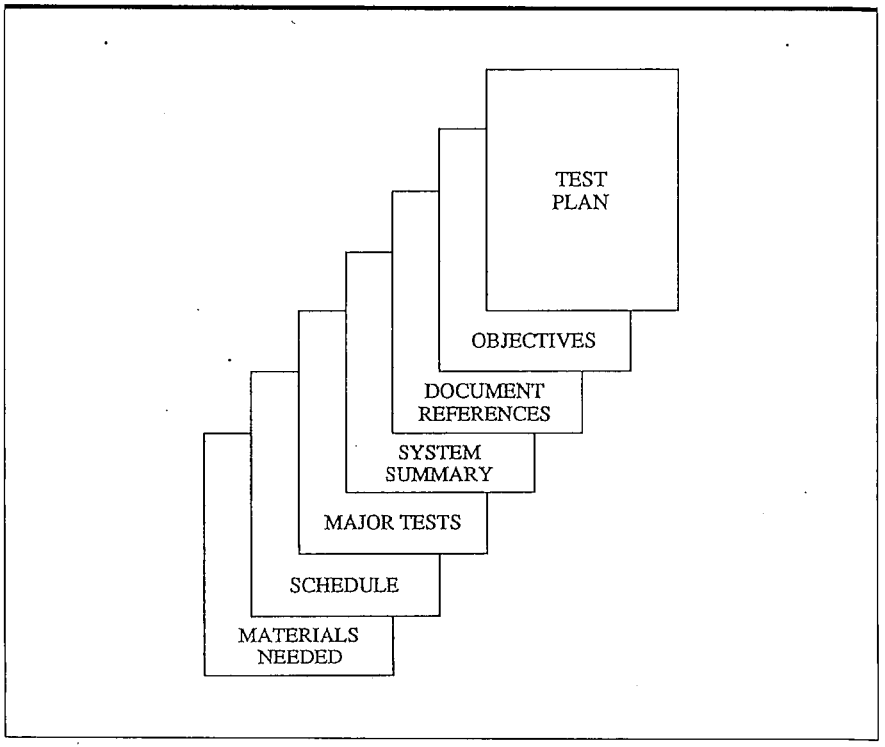


Following these preliminary sections is a *system summary*. Since a reader of the plan may not have been involved with the previous stages of development, the system summary puts the testing schedule and events in context. The summary need not be detailed; it can be a drawing depicting the major system inputs and outputs with a description of major transformations.

Once testing is placed in a system context, the plan describes the *major tests and test approaches* to be used. For example, the test plan distinguishes among function tests, performance tests, acceptance tests, and installation tests. If the function tests can be further divided by some criteria (such as subsystem tests), the test plan lays out the overall organization of the testing.

After explaining the component tests, the plan addresses the *schedule of events*. The schedule includes the place of the test as well as the time frame. Often depicted as a milestone chart or activity graph, the test schedule includes the following information:

Figure 8.11 Test Plan Components



1. The overall testing period
2. The major subdivisions of testing and their start and stop times
3. Any pretest requirements (such as orientation or familiarization with the system, user training, or generation of test data) and the time necessary for each
4. The time necessary for preparation and review of the test analysis report

If testing is to take place at several locations, the test plan includes a schedule for each location. A chart illustrates the hardware, software, and personnel necessary for the administration of the tests at each location and the duration for which each resource will be needed. Noted are special training or maintenance needs, too.

The plan identifies *test materials* in terms of deliverables (such as user or operator manuals, sample listings, or tapes) and materials supplied by the site (such as special test apparatus, data base tables, or storage media). For example, if a test is to use a data base management system to build a sample data base, the test may require the users at the site to define data elements before the arrival of the test

team. Similarly, if any security or privacy precautions are required by the test team, the personnel at the test location may be required to establish passwords or special access for the test team before the test begins.

Test Specification and Evaluation

The test plan describes an overall breakdown of testing into components that test for specific items. For example, if the system being tested has its processing distributed over several computers, the function and performance tests can be further divided into tests for each subsystem.

For each such test component, a test specification and evaluation is written. The specification begins by listing the *requirements* whose satisfaction will be demonstrated by the test. Referring to the requirements definition document, this section explains the purpose of the test.

One way to view the correspondence between the requirements and the tests is to establish a table or chart relating the two. Table 8.3 is an example of such a chart and is appropriate for inclusion in the Weaver Farm Test Plan.

Table 8.3 Test-Requirement Correspondence Chart

	Requirement		
Test	Generate and Maintain Data Base (2.4.1)	Selectively Retrieve Data (2.4.2)	Produce Specialized Reports (2.4.3)
1. Add new record	X		
2. Add field	X		
3. Change field	X		
4. Delete record	X		
5. Delete field	X		
6. Create index		X	
Retrieve record with a requested:			
7. Cell number		X	
8. Water height		X	
9. Canopy height		X	
10. Ground cover		X	
11. Percolation rate		X	
12. Print full data base			X
13. Print directory			X
14. Print keywords			X
15. Print simulation summary			X

Note that the requirements listed across the top of the chart reference the number in the requirements document; the function on the left of the chart is mandated by the requirement in whose column the X [illegible]ed.

The *system functions* involved in the test are enumerated in Table 8.3. The performance tests can be described in a similar way. Instead of listing functional requirements, the chart lists requirements relating to the speed of access, the security of the data base, and so on.

Often, the test component is a collection of smaller tests, the sum of which illustrates the satisfaction of a set of requirements. In this case, the specification shows the relationship between the smaller tests and the requirements.

Each test component is guided by a *test philosophy* and adopts a set of *methods*. However, the philosophy and methods may be constrained by other requirements and by the realities of the test situation. The specification makes these test conditions clear. Among the *conditions* may be some of the following:

1. Is the system using actual input from users or devices, or are special cases generated by a program or device?
2. Will the test cover all paths? all branches? all functions?
3. How will data be recorded?
4. Are there timing, interface, equipment, personnel, data base, or other limitations on testing?
5. If the test is a series of smaller tests, in what order are the tests performed?

If test data are to be processed before being evaluated, the *processing* is discussed. For instance, when large amounts of data are produced by a system, data reduction techniques are sometimes used on the output so that the result is more suitable for evaluation.

Accompanying each test is a way to tell when the test is *complete*. Thus, the specification is followed by a discussion of how we know when the test is over. For example, for the output collected, the plan explains what range will meet the requirement.

The *evaluation method* follows the completion criteria. For example, data produced during testing may be collected and collated manually and then inspected by the test team. Alternately, the team could use an automated tool to evaluate some of the data and then inspect summary reports or do an item-by-item comparison with the expected output.

Test Description

A test description is written for every test defined in the test specification. We use the test description document as a guide in performing the test. These documents must be detailed and clear. The test description is composed of several parts:

1. Means of control
2. Data
3. Procedures

A *general description* of the test begins the document. Then, we indicate whether the test will be initiated and controlled by manual or automatic means. For instance, data may be input manually from the keyboard, but then an automated driver may exercise the functions being tested. Alternatively, the entire process could be automated.

The *test data* can be viewed as several parts: input data, input commands, output data, and messages produced by the system. Each is described in detail. For instance, input commands are provided so that the team knows how to initiate the test, halt or suspend it, repeat or resume an unsuccessful or incomplete one, or terminate the test. Similarly, the team must interpret messages to understand the status of the system and to control testing. We explain how the team can distinguish among errors resulting from input data, from improper test procedures, or from hardware malfunction (wherever possible).

For example, the test data description for a test of a SORT routine may be the following:

INPUT DATA:

Input data is to be provided by the LIST program. This program generates randomly a list of N words of alphanumeric characters; each word is of length M. The program is invoked by calling

RUN LIST (N, M)

in your test driver. The output is placed in global data area LISTBUF. The test data sets to be used for this test are as folows:

Case 1: Use LIST with N = 5, M = 5
Case 2: Use LIST with N = 10, M = 5
Case 3: Use LIST with N = 15, M = 5
Case 4: Use LIST with N = 50, M = 10
Case 5: Use LIST with N = 100, M = 10
Case 6: Use LIST with N = 150, M = 10

INPUT COMMANDS:

The SORT routine is invoked by using the command

RUN SORT (INBUF, OUTBUF) or
RUN SORT (INBUF)

OUTPUT DATA:

If two parameters are used, the sorted list is placed in OUTBUF. Otherwise, it is placed in INBUF.

SYSTEM MESSAGES:

During the sorting process, the following message is displayed:

"Sorting . . . Please wait . . ."

Upon completion, SORT displays the following message on the screen:

"SORT completed"

To halt or terminate the test before the message is displayed, press CTRL-C on the keyboard.

Test procedures are often called a **test script** because they give us a step-by-step description of how to perform the test. A rigidly defined set of steps gives us control over the test. We can duplicate errors or conditions, if necessary. If the test is interrupted for some reason, we must be able to continue the test without having to return to the beginning. In addition, if the interruption is the result of an error, knowing the exact steps that led to the interruption allows us to try to duplicate the error and verify that it is a genuine system problem and not a random event.

For example, part of the test script for testing the "Change Field" function (listed in Table 8.3) might look like the steps in Table 8.4.

The test script steps are numbered, and data associated with each step are referenced. If we have not described them elsewhere, we explain how to prepare the data or the site for the test. For example, the equipment settings needed, the data base definitions, and the communication connections may be detailed. Next, the script explains exactly what is to happen during the test. The keys pressed, the screens displayed, the output produced, the equipment reactions, and any other manifestation of activity are reported. We explain the expected outcome or output, and we give instructions to the operator or user about what to do if the outcome differs.

Finally, the test description explains the sequence of *activities required to end the test.* These activities may involve reading or printing critical data, terminating automated procedures, or turning off pieces of equipment.

Test Analysis Report

When a test has been administered, we analyze the results to determine if the function or performance tested meets the requirements. Sometimes the mere *demonstration* of a function is enough. Most of the time, though, there are *performance constraints* on the function. For instance, it is not enough to know that a column can be sorted or summed. The speed of the calculation must be measured and compared with the corresponding constraint. Thus, a test analysis report is necessary for the following reasons:

1. It documents the results of a test.
2. If an error is discovered, the report provides information necessary to locate the source of and correct the error.
3. It provides information necessary to determine if the development project is complete.
4. It establishes confidence in the performance of the system.

The test analysis report may be read by people who were not part of the test process but who are familiar with other aspects of the system and its development. Thus, the report includes a brief *summary of the project* and its objectives and relevant references for this test. For example, the test report mentions those parts of the requirements, design, and implementation documents that describe the functions exercised in this test. The report also indicates those parts of the test plan and specification documents that deal with this test.

Table 8.4 Test Script for Change Field Function

Step *N*:	Press function key 4: Access data file.
Step *N*+1:	Screen will ask for name of data file. Type 'sys:test.text'.
Step *N*+2:	Menu will appear, reading: • Delete file • Modify file • Rename file Place cursor next to 'Modify file' and press RETURN key.
Step *N*+3:	Screen will ask for record number. Type '4017'.
Step *N*+4:	Screen will fill with data fields for record 4017: RECORD NUMBER: 4017 CELL X: 0042 CELL Y: 0036 SOIL TYPE: CLAY PERCOLATION RATE: 4 FT/HR VEGETATION: KUDZU CANOPY HEIGHT: 25 FT WATER TABLE: 12 FT CONSTRUCT: OUTHOUSE MAINTENANCE CODE: 3T/4F/9R
Step *N*+5:	Press function key 9: Modify.
Step *N*+6:	Entries on screen will be highlighted. Move cursor to VEGETATION field. Type 'GRASS' over 'KUDZU' and press RETURN key.
Step *N*+7:	Entries on screen will no longer be highlighted. VEGETATION field should now read 'GRASS'.
Step *N*+8:	Press function key 16: Return to previous screen.
Step *N*+9:	Menu will appear, reading: • Delete file • Modify file • Rename file To verify that modification has been recorded, place cursor next to 'Modify file' and press RETURN key.
Step *N*+10:	Screen will ask for record number. Type '4017'.
Step *N*+11	Screen will fill with data fields for record 4017: RECORD NUMBER: 4017 CELL X: 0042 CELL Y: 0036 SOIL TYPE: CLAY PERCOLATION RATE: 4 FT/HR VEGETATION: GRASS CANOPY HEIGHT: 25 FT WATER TABLE: 12 FT CONSTRUCT: OUTHOUSE MAINTENANCE CODE: 3T/4F/9R

Once the stage is set in this way, the test analysis report lists the *functions and performance characteristics* that were to be demonstrated and describes the actual results. The results include function, performance, and data measures, noting whether the target requirements have been met. If an error or deficiency has been discovered, the report discusses its impact. Sometimes, we evaluate the test results in terms of a measure of severity. This measure helps the test team decide whether to continue testing or wait until the error has been corrected. For example, if the error is a spurious character in the upper part of a display screen, testing can continue while we locate and correct the error. However, if an error causes the system to crash or a data file to be deleted, the test team may decide to interrupt testing.

Discrepancy Report Forms. Since the purpose of testing is to locate errors, not to correct them, the test analysis report discusses only the *existence* of errors and their impact on the system. For each error or deficiency noted in the report, a **discrepancy report form (DRF)** is completed and given to the development team. The DRF includes the following information:

1. The *state* of the system before the error occurred
2. The *evidence* of the error
3. Any *action* or procedure that appears to have led to the error's occurrence
4. A description of how the system *should* work without the error or discrepancy
5. A reference to relevant *requirements*
6. The *impact* of the error's presence on the system
7. The level of *severity*, if available

The developer can use the DRF information to decide what action to take. Customer and developer may consult to determine if the error is severe enough to interrupt testing. Then the DRF is assigned to members of the development team, who locate the *source* of the error. As we shall see in the next two chapters, the location and correction of an error may involve anything from correcting documentation to redesigning the system.

8.9

RESOURCE TRACKING AND SIMULATION SYSTEM EXAMPLE

The testing concepts introduced in this chapter can be applied to the Weaver Farm project. We continue to view the Weaver Farm system as a set of subsystems. Suppose we decide to test it one subsystem at a time. We write a build plan to describe the individual parts to be tested and the way in which the entire system will be constructed from its parts. For instance, we may build the data base management system first, the simulation system next, the reporting system third, and then have a sequence of spins that reflect the combining of the different functions.

Figure 8.12 Weaver Farm DRF

DRF Number:_____ Tester Name: _____

Date:_____ Time: _____

Test Number:_____

Script step executed when error occurred:_____

Description of error:

Activities before occurrence of error:

Expected results:

Requirements affected:

Impact of error on test:

Impact of error on system:

Security Level: (LOW) 1 2 3 4 5 (HIGH)

The function testing of the Weaver Farm system can focus on each of the subsystems in turn. For example, the subsystem that manages the data base of information about the geographical cells can be tested for a variety of functions. We have seen how the functions and requirements can be related in a table to show how each test will address the testing of which functions.

We have also seen how the tests can be specified and how test scripts detail the steps of each test. For instance, the test script in Table 8.4 asks us to update a record by typing 'GRASS' to replace 'KUDZU'. The steps read as follows:

Step N+5: Press function key 9: Modify.

Step N+6: Entries on screen will be highlighted. Move cursor to VEGETATION field. Type 'GRASS' over 'KUDZU' and press RETURN key.

Step N+7: Entries on screen will no longer be highlighted. VEGETATION field should now read 'GRASS'.

Suppose we run this test and find an error. We perform all steps through step $N + 7$, but the VEGETATION field is not updated; it still reads 'KUDZU'.

We complete a DRF to notify the others on the test team that something has gone wrong. The DRF for the Weaver Farm system is formatted as in Figure 8.12. The DRF number is assigned by the configuration management member of the test team. We describe the error by explaining what was done before the error occurred (an attempt to modify a record) and what keys were pressed. We explain that we expected the updated field to contain the new information ('GRASS'), but the old information ('KUDZU') was still displayed.

We refer to the test specification to see what requirements are affected by the error. The configuration management team can use this information to decide whether other tests will be affected by the failure of this one. For example, any other test that includes a data base update by a user may fail.

The update failure may prevent the rest of this test from being performed if the test depends on the information being placed in the field by the update process. Or, the update may be independent of the remainder of the test, so the test can continue to completion. However, the impact of the error on the *test* may be very different from the effect of the error on the *system*. For example, perhaps the field was updated in the wrong record; another possibility is that the correct record was updated on the disk but the source of the error lies in the module that retrieves the information and displays it on the screen. Errors of this nature may have a much more severe effect on the system than they do on the particular test being run. The group performing this test must rate the error on a scale from 1 (not very severe) to 5 (extremely severe); then, the test team uses this information to decide whether to continue this test, halt this test but allow others to proceed, or halt some or all of the other tests until the source of the error is located and repaired.

8.10

CHAPTER SUMMARY

In this chapter, we have taken a close look at where errors might originate. We saw that there are many places in the system development process that can lead to a

proliferation of errors later on. Next, we discussed the testing of a system as a whole, once we are convinced that the implementation modules work as the program designers intended. System testing can be thought of as a progression of steps, from testing for functional requirements (function testing) and nonfunctional requirements (performance testing) to see if the developer's implementation agrees with the customer's written requirements, to acceptance and installation testing to see if the customer is happy with the result.

Configuration management is an important part of the testing process, because it is necessary for connecting the errors discovered with the requirements and design elements from which they came. Both configuration management and regression testing are techniques that the test team can use to be sure it is in control of the testing process.

Although there are automated test tools that can help the test team to do its job, the test team still has an enormous task. It must plan its tests carefully and coordinate them with the customer, user, and all members of the development team. The blueprint for all testing is written in the form of a test plan. Each major test addressed by the plan is broken into smaller tests in the test specification, and the test description describes each small test as a step-by-step test script. Thus, the members of the test team can enter each step of testing knowing what is needed as input; what procedures should be followed; what to expect on the display screens, in data bases, or as printed output; and how to report errors or discrepancies.

Once testing is complete, the system is delivered to the customer. In the next chapter, we will investigate the steps involved in transferring responsibility for the system to the customer.

EXERCISES

1. Consider the development of a two-pass assembler. Outline the functions of the assembler. Then describe how you might test the assembler so that each function is tested thoroughly before the next function is examined. Suggest a build plan for the development, and explain how the build plan and the testing must be designed together.
2. Certification is an endorsement of the correctness of a system by some outside source. Certification is often granted by comparing the system to a predefined standard of performance. For example, the Department of Defense certifies an Ada compiler after testing the compiler against a long list of functional specifications. In the terminology of this chapter, is a test for certification a function test? a performance test? an acceptance test? an installation test? Explain why or why not for each type of test.
3. The development of a build plan must take into account the resources available to the customer and developer, including time, personnel, and money. Give examples of resource constraints that can affect the number of builds defined for system development. Explain how these constraints affect the build plan.

4. Suppose a mathematician's calculator has a function that computes the slope and intercept of a line. The requirement in the definition document reads "The calculator shall accept as input an equation of the form $Ax + By + C = 0$ and print out the slope and intercept." The system implementation of this requirement is the function LINE whose syntax is 'LINE(A, B, C)', where A and B are the coefficients of x and y and C is the constant in the equation. The result is a printout of 'D; E', where D is the slope and E the intercept. Write this requirement as a set of causes and effects, and draw the corresponding cause-and-effect graph.
5. In chapter 3, we discussed the need for requirements to be "testable." Explain why testable requirements are essential for performance testing. Use examples to support your explanation.
6. What kinds of performance tests might be required for a word processing system? for an automatic bank teller system? for an office payroll system?
7. A word processing system can be designed so that it serves one user or many. Explain how such a system can have a variety of configurations, and outline how a set of configuration tests might be designed.
8. A navigation system is to be installed in an airplane. What issues must be considered in an installation test?
9. Give an example to illustrate a situation where testing would be impossible without the use of a device simulator. Give another example of a situation where testing would be impossible without a system simulator.
10. A payroll system is designed so that there is an employee information record for each employee. Once a week, the employee record is updated with the number of hours worked by the employee that week. Once every two weeks, summary reports are printed to display the number of hours worked since the beginning of the fiscal year. Once a month, paychecks are printed for all employees. For each of the categories of performance tests discussed in this chapter, describe what kinds of tests should be administered to insure that this system performs properly.
11. Willie's Wellies, Inc. has commissioned HMP Systems to develop a computer-based system for testing the strength of its complete line of rubber footwear. Willie's has nine factories in various locations throughout the world, and each system will be configured according to the size of the factory. Explain why HMP and Willie's should conduct installation testing when acceptance testing is complete.
12. Write a test script for testing the LEVEL function described in this chapter.
13. Outline a build plan for testing the Weaver Farm system.

1 Measurement: what is it and why do it?

Software measurement, once an obscure and esoteric specialty, has become essential to good software engineering. Many of the best software developers measure characteristics of the software to get some sense of whether the requirements are consistent and complete, whether the design is of high quality, and whether the code is ready to be tested. Effective project managers measure attributes of process and product to be able to tell when the software will be ready for delivery and whether the budget will be exceeded. Informed customers measure aspects of the final product to determine if it meets the requirements and is of sufficient quality. And maintainers must be able to assess the current product to see what should be upgraded and improved.

This book addresses all of these concerns and more. The first six chapters examine and explain the fundamentals of measurement and experimentation, providing you with a basic understanding of why we measure and how that measurement supports investigation of the use and effectiveness of software-engineering tools and techniques. Chapters 7 through 12 explore software engineering measurement in great detail, with information about specific metrics and their uses. Chapters 13, 14 and 15 offer a management perspective on software measurement, explaining current practices and looking ahead toward the future of measurement and metrics. Collectively, the chapters offer broad coverage of all aspects of software-engineering measurement, plus enough depth so that you can apply the metrics to your processes, products and

resources. If you are a student, not yet experienced in working on projects with groups of people to solve interesting business or research problems, this book explains how measurement can become a natural and useful part of your regular development and maintenance activities.

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

1.1 MEASUREMENT IN EVERYDAY LIFE

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

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

1.1.1 What is measurement?

These examples present a picture of the variety in how we use measurement. But there is a common thread running through each of the described activities: in every case, some aspect of a thing is assigned a descriptor that allows us to compare it with others. In the shop, we can compare the price of one item with another. In the clothing store, we contrast sizes. And on our journey, we compare distance traveled to distance remaining. The rules for assignment and comparison are not explicit in the examples, but it is clear that we make our comparisons and calculations according to a well-defined set of rules. We can capture this notion by defining measurement formally in the following way:

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

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

When we describe entities by using attributes, we often define the attributes using numbers or symbols. Thus, price is designated as a number of dollars or pounds sterling, while height is defined in terms of inches or centimeters. Similarly, clothing size may be "small," "medium" or "large," while fuel is "regular," "premium" or "super." These numbers and symbols are abstractions that we use to reflect our perceptions of the real world. For example, in defining the numbers and symbols, we try to preserve certain relationships that we see among the entities. Thus, someone who is six feet in height is taller than someone who is five feet in height. Likewise, a "medium" T-shirt is smaller than a "large" T-shirt. This number or symbol can be very useful and important. If we have never met Herman but are told that he is seven feet tall, we can imagine his height in relation to ourselves without our ever having seen him. Moreover, because of his unusual height, we know that he will have to stoop when he enters the door of our office. Thus, we can make judgments about entities solely by knowing and analyzing their attributes.

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

- We have noted that color is an attribute of a room. In a room with blue walls, is "blue" a measure of the color of the room?
- The height of a person is a commonly understood attribute that can be measured. But what about other attributes of people, such as intelligence?

Is intelligence adequately measured by an IQ test score? Similarly, wine can be measured in terms of alcohol content ("proof"), but can wine quality be measured using the ratings of experts?

- The accuracy of a measure depends on the measuring instrument as well as on the definition of the measurement. For example, length can be measured accurately as long as the ruler is accurate and used properly. But some measures are not likely to be accurate, either because the measurement is imprecise or because it depends on the judgment of the person doing the measuring. For instance, proposed measures of human intelligence or wine quality appear to have likely error margins. Is this a reason to reject them as bona fide measurements?
- Even when the measuring devices are reliable and used properly, there is margin for error in measuring the best understood physical attributes. For example, we can obtain vastly different measures for a person's height, depending on whether we make allowances for the shoes being worn or the standing posture. So how do we decide which error margins are acceptable and which are not?
- We can measure height in terms of meters, inches or feet. These different scales measure the same attribute. But we can also measure height in terms of miles and kilometers – appropriate for measuring the height of a satellite above earth, but not for measuring the height of a person. When is a scale acceptable for the purpose to which it is put?
- Once we obtain measurements, we want to analyze them and draw conclusions about the entities from which they were derived. What kind of manipulations can we apply to the results of measurement? For example, why is it acceptable to say that Fred is twice as tall as Joe, but not acceptable to say that it is twice as hot today as it was yesterday? And why is it meaningful to calculate the mean of a set of heights (to say, for example, that the average height of a London building is 200 meters), but not the mean of the football jersey numbers of a team?

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

1.1.2 *Making things measurable*

"What is not measurable make measurable".

This phrase, attributable to Galileo Galilei (1564–1642), is part of the folklore of measurement scientists (Finkelstein 1982). It suggests that one of the aims of science is to find ways to measure attributes of things in which we are interested. Implicit in this statement is the idea that measurement makes concepts more visible and therefore

more understandable and controllable. Thus, as scientists, we should be creating ways to measure our world; where we can already measure, we should be making our measurements better.

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

To improve the rigor of measurement in software engineering, we need not restrict the type or range of measurements we can make. Indeed, measuring the unmeasurable should improve our understanding of particular entities and attributes, making software engineering as powerful as other engineering disciplines. Even when it is not clear how we might measure an attribute, the act of proposing such measures will open a debate that leads to greater understanding. Although some software engineers may continue to claim that important software attributes like dependability, quality, usability and maintainability are simply not quantifiable, we prefer to try to use measurement to advance our understanding of them.

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

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

EXAMPLE 1.1: In the decathlon athletics event, we measure the time to run various distances as well as the length covered in various jumping activities. These measures are subsequently combined into an *overall score*, computed

using a complex weighting scheme that reflects the importance of each component measure. The weights are sometimes changed as the relative importance of an event or measure changes. This score is widely accepted as a description of the athlete's all-round ability. In fact, the winner of the Olympic decathlon is generally acknowledged to be the world's finest athlete.

EXAMPLE 1.2: The England soccer league points system is used to select the best all-round team over the course of a season. In 1981, the points system was changed; a win yielded three points instead of two, while a draw still yielded one point. This change was made to reflect the consensus view that the qualitative difference between a win and a draw was greater than that between a draw and a defeat.

EXAMPLE 1.3: There are no universally-recognized measures to identify the best individual soccer players (although number of goals scored is a fairly accurate measure of quality of a striker). Although many fans and players have argued that player quality is an unmeasurable attribute, this issue was addressed prior to the 1994 World Cup games in the USA. To provide an objective (measurable) means of determining the "man of the match," several new measurements were proposed:

> To help FIFA assess the best players, it will be necessary to add to the pitch markings. At ten meter intervals there will be lines both across and down the pitch. This will allow accurate pass yardage, sideways pass yardage, dribble yardage, and heading yardage to be found for each player.
>
> Translated from "Likely changes to the rules for the 1994 World Cup," *Nouveaux FIFA d'Arbitres* (FIFA Referees News), March 1990.

It was suggested that these measurements be added and weighted with the number of goals scored; tackles, saves or interceptions made; frequency and distance of passes (of various types), dribbles and headers. Notice that the proposed new measure of player quality required a change to the physical environment in which the game is played.

It is easy to see parallels in software engineering. In many instances, we want an overall score that combines several measures into a "big picture" of what is going on during development or maintenance. We want to be able to tell if a software product is good or bad, based on a set of measures, each of which captures a facet of "goodness." Similarly, we want to be able to measure an organization's ability to produce good software, or a model's ability to make good predictions about the software-development process. The composite measures can be controversial, not only because of the individual measures comprising it, but also because of the weights assigned.

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

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

1.2 MEASUREMENT IN SOFTWARE ENGINEERING

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

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

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

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

1.2.1 Neglect of measurement in software engineering

Engineering disciplines use methods that are based on models and theories. For example, in designing electrical circuits we appeal to theories like Ohm's law, which describes the relationship between resistance, current and voltage in the circuit. But the laws of electrical behavior have evolved by using the scientific method: stating a hypothesis, designing and running an experiment to test its truth, and analyzing the

results. Underpinning the scientific process is measurement: measuring the variables to differentiate cases, measuring the changes in behavior, and measuring the causes and effects. Once the scientific method suggests the validity of a model or the truth of a theory, we continue to use measurement to apply the theory to practice. Thus, to build a circuit with a specific current and resistance, we know what voltage is required and we use instruments to measure whether we have such a voltage in a given battery.

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

1. We fail to set measurable targets for our software products. For example, we promise that the product will be user-friendly, reliable and maintainable without specifying clearly and objectively what these terms mean. As a result, when the project is complete, we cannot tell if we have met our goals. This situation has prompted Tom Gilb to state (Gilb, 1988)

Gilb's Principle of Fuzzy Targets: projects without clear goals will not achieve their goals clearly.

---

2. We fail to understand and quantify the component costs of software projects. For example, most projects cannot differentiate the cost of design from the cost of coding or testing. Since excessive cost is a frequent complaint from many of our customers, we cannot hope to control costs if we are not measuring the relative components of cost.
3. We do not quantify or predict the quality of the products we produce. Thus, we cannot tell a potential user how reliable a product will be in terms of likelihood of failure in a given period of use, or how much work will be needed to port the product to a different machine environment.
4. We allow anecdotal evidence to convince us to try yet another revolutionary new development technology, without doing a carefully controlled study to determine if the technology is efficient and effective. Figure 1.1 shows examples typical of promotional materials for automated software development tools and techniques. But most of the time, these materials are not accompanied by reports of the scientific basis for the claims.

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

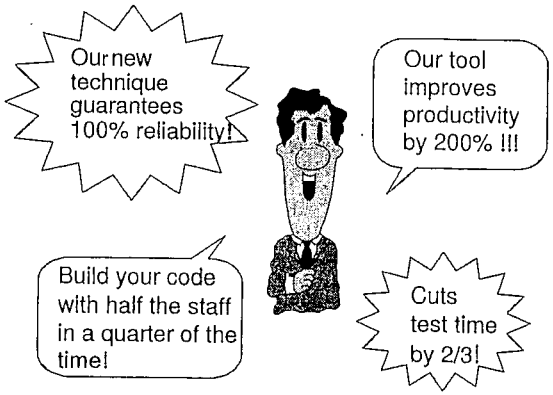


Figure 1.1: Measurement for promotion

situations. In addition, we cannot do an objective study to repeat the measurements in our own environments. Thus, the lack of measurement in software engineering is compounded by the lack of a rigorous approach.

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

1.2.2 Objectives for software measurement

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

"You cannot control what you cannot measure".

(DeMarco, 1982)

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

Below are examples of the kinds of information needed to understand and control a software development project, separated by manager and developer perspectives:

Managers

- *What does each process cost?* We can measure the time and effort involved in the various processes that comprise software production. For example, we can identify the cost of eliciting requirements, the cost of specifying the system, the cost of designing the system, and the cost of coding and testing the system. In this way, we gain understanding not only of the total project cost but also of the contribution of each activity to the whole.
- *How productive is the staff?* We can measure the time it takes for staff to specify the system, design it, code it, and test it. Then, using measures of the size of specifications, design, code, and test plans, for example, we can determine how productive the staff is at each activity. This information is useful when changes are proposed; the manager can use the productivity figures to estimate the cost and duration of the change.
- *How good is the code being developed?* By carefully recording faults, failures and changes as they occur, we can measure software quality, enabling us to compare different products, predict the effects of change, assess the effects of new practices, and set targets for process and product improvement.
- *Will the user be satisfied with the product?* We can measure functionality by determining if all of the requirements requested have actually been implemented properly. And we can measure usability, reliability, response time, and other characteristics to suggest whether our customers will be happy with both functionality and performance.
- *How can we improve?* We can measure the time it takes to perform each major development activity, and calculate its effect on quality and productivity. Then we can weigh the costs and benefits of each practice to determine if the benefit is worth the cost. Alternatively, we can try several variations of a practice and measure the results to decide which is best; for example, we can compare two design methods to see which one yields the higher quality code.

Engineers

- *Are the requirements testable?* We can analyze each requirement to determine if its satisfaction is expressed in a measurable, objective way. For example, suppose a requirement states that a system must be reliable; the requirement can be replaced by one that states that the mean time to failure must be greater than 15 elapsed hours of CPU time.
- *Have we found all the faults?* We can measure the number of faults in the specification, design, code, and test plans, and trace them back to their root causes. Using models of expected detection rates, this information can help us to decide whether inspections and testing have been effective and whether a product can be released for the next phase of development.
- *Have we met our product or process goals?* We can measure characteristics of the products and processes that tell us whether we have met standards, satisfied a requirement, or met a process goal. For example, certification may require that fewer than 20 failures have been reported per [illegible] site over a given period of time. Or a standard may mandate that no module contain more than 100 lines of code. The testing process may require that unit testing must achieve 90% statement coverage.
- *What will happen in the future?* We can measure attributes of existing products and current processes to make predictions about future ones. For example, measures of size of specifications can be used to predict size of the target system, predictions about future maintenance problems can be made from measures of structural properties of the design documents, and predictions about the reliability of software in operational use can be made by measuring reliability during testing.

1.2.3 Measurement for understanding, control and improvement

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

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

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

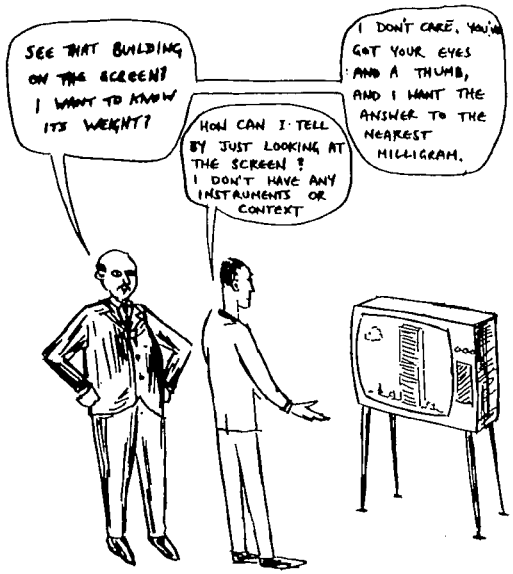


Figure 1.2: Software measurement – resource estimation

No matter how measurements are used, it is important to manage the expectations of those who will make measurement-based decisions. Users of the data should always be aware of the limited accuracy of prediction and of the margin of error in the measurements. As with any other engineering discipline, there is room in software engineering for abuse and misuse of measurement. Figure 1.2 irreverently shows how management can pressure developers to produce precise measures with inadequate models, tools and techniques.

If you are expecting measurement to provide instant, easy solutions to your software engineering problems, be aware of our corollary to DeMarco's rule:

> You can neither predict nor control what you cannot measure.

1.3 THE SCOPE OF SOFTWARE METRICS

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

- cost and effort estimation
- productivity measures and models
- data collection
- quality models and measures
- reliability models

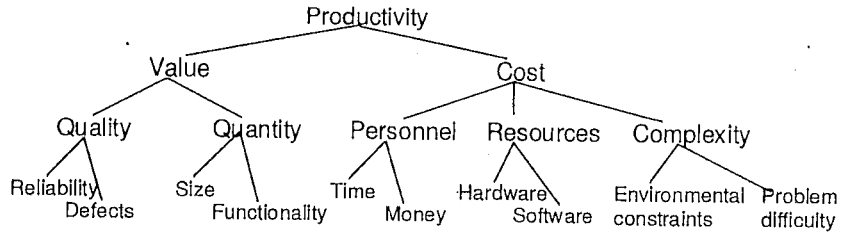


Figure 1.3: A productivity model

- performance evaluation and models
- structural and complexity metrics
- capability-maturity assessment
- management by metrics
- evaluation of methods and tools

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

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

1.3.1 Cost and effort estimation

Managers provided the original motivation for deriving and using software measures. They wanted to be able to predict project costs during early phases in the software life-cycle. As a result, numerous models for software cost and effort estimation have been proposed and used. Examples include Boehm's COCOMO model (Boehm, 1981), Putnam's SLIM model (Putnam, 1978) and Albrecht's function points model (Albrecht, 1979). These and other models often share a common approach: effort is expressed as a (pre-defined) function of one or more variables (such as size of the product, capability of the developers and level of reuse). Size is usually defined as (predicted) lines of code or number of function points (which may be derived from the product specification). Cost models and effort prediction are discussed in Chapter 12.

1.3.2 Productivity models and measures

The pressing needs of management have also resulted in numerous attempts to define measures and models for assessing staff productivity during different software processes and in different environments.

Figure 1.3 illustrates an example of the possible components that contribute to overall productivity. It shows productivity as a function of value and cost; each is then decomposed into other aspects, expressed in measurable form. This model is a

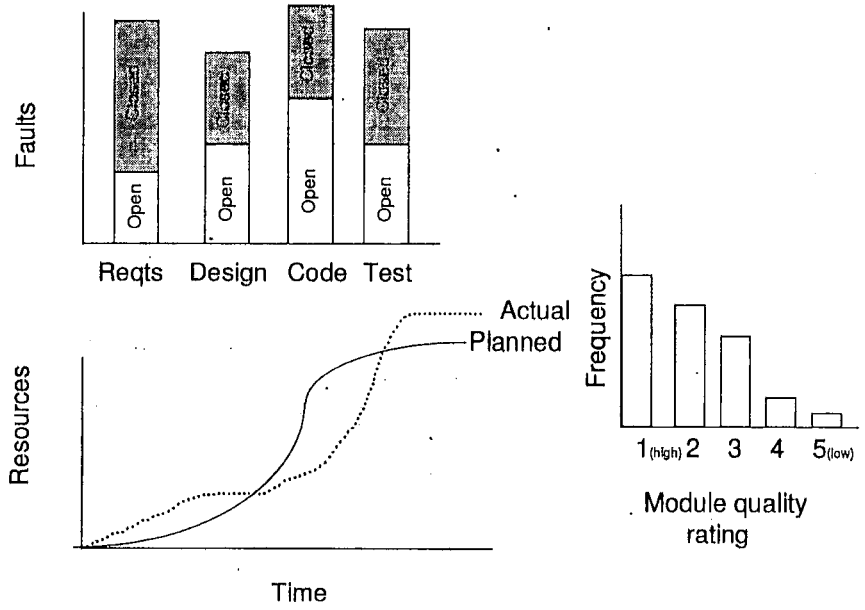


Figure 1.4: Metrics for management

significantly more comprehensive view of productivity than the traditional one, which simply divides size by effort. That is, many managers make decisions based on the rate at which lines of code are being written per person month of effort. This simpler measure can be misleading, if not dangerous (Jones, 1986). Productivity models and measures are covered mainly in Chapter 11.

1.3.3 *Data collection*

The quality of any measurement program is clearly dependent on careful data collection. But collecting data is easier said than done, especially when data must be collected across a diverse set of projects. Thus, data collection is becoming a discipline in itself, where specialists work to ensure that measures are defined unambiguously, that collection is consistent and complete, and that data integrity is not at risk. But it is acknowledged that metrics data collection must be planned and executed in a careful and sensitive manner. We will see in Chapter 14 how Hewlett-Packard and others have made public the managerial framework that helped to make its corporate metrics program a success.

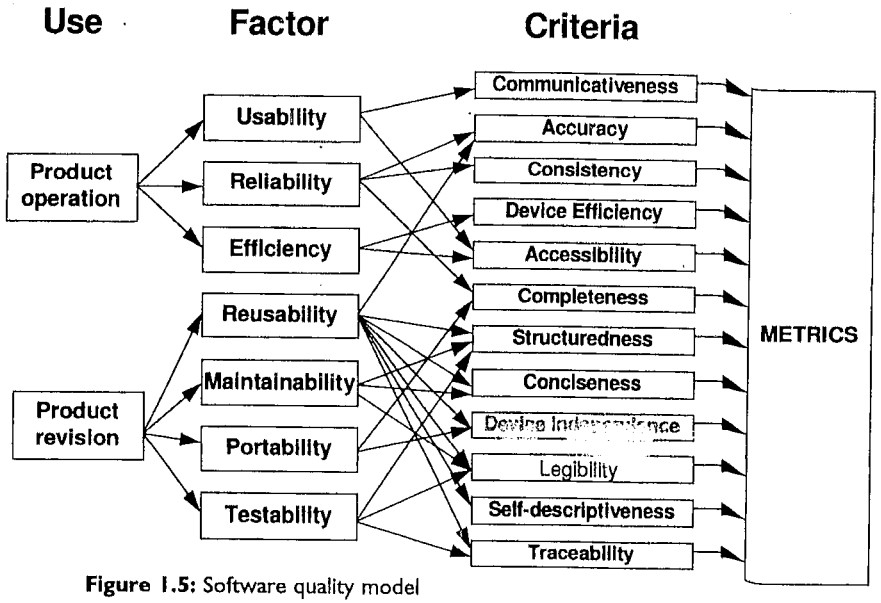


Figure 1.5: Software quality model

Figure 1.4 contains several examples of how the data collected can be distilled into simple charts and graphs that show managers the progress and problems of development. Basili and Weiss have described a general methodology for valid data collection (Basili and Weiss, 1984), while Mellor describes the data collection necessary for reliability assessment (Mellor, 1992). This material is covered in Chapters 5, 6, 10 and 11.

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

1.3.4 *Quality models and measures*

Productivity cannot be viewed in isolation. Without an accompanying assessment of product quality, speed of production is meaningless. This observation has led software engineers to develop models of quality whose measurements can be combined with those of productivity models. For example, Boehm's advanced COCOMO cost-estimation model is tied to a quality model (Boehm *et al.*, 1978). Similarly, the McCall quality model (McCall *et al.*, 1977), commonly called the FCM (Factor Criteria Metric) model, is related to productivity.

These models are usually constructed in a tree-like fashion, similar to Figure 1.5. The upper branches hold important high-level quality factors of software products, such as reliability and usability, that we would like to quantify. Each quality factor is [illegible] of lower-level criteria, such as structuredness and traceability. The criteria

are easier to understand and measure than the factors; thus, actual measures (metrics) are proposed for the criteria. The tree describes the pertinent relationships between factors and their dependent criteria, so we can measure the factors in terms of the dependent criteria measures. This notion of divide-and-conquer has been implemented as a standard approach to measuring software quality (ISO 9126). Quality models are described at length in Chapter 9.

1.3.5 *Reliability models*

Most quality models include reliability as a component factor, but the need to predict and measure reliability itself has led to a separate specialization in reliability modeling and prediction. Littlewood (1988) and others provide a rigorous and successful example of how a focus on an important product quality attribute has led to increased understanding and control of our products. This material is described in Chapter 10.

1.3.6 *Performance evaluation and models*

Performance is another aspect of quality. Work under the umbrella of performance evaluation includes externally observable system performance characteristics, such as response times and completion rates (Ferrari *et al.*, 1978, 1983; Kleinrock, 1975). Performance specialists also investigate the internal workings of a system, including the efficiency of algorithms as embodied in computational and algorithmic complexity (Garey and Johnson 1979; Harel, 1992). The latter is also concerned with the inherent complexity of problems measured in terms of efficiency of an optimal solution. This material is described in Chapter 7.

1.3.7 *Structural and complexity metrics*

Desirable quality attributes like reliability and maintainability cannot be measured until some operational version of the code is available. Yet we wish to be able to predict which parts of the software system are likely to be less reliable, more difficult to test, or require more maintenance than others, even before the system is complete. As a result, we measure structural attributes of representations of the software which are available in advance of (or without the need for) execution; then, we try to establish empirically predictive theories to support quality assurance, quality control, and quality prediction. Halstead (1977) and McCabe (1976) are two classic examples of this approach; each defines measures that are derived from suitable representations of source code. This material and more recent developments are described in Chapter 8.

1.3.8 *Management by metrics*

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

1.3.9 *Evaluation of methods and tools*

Many articles and books describe new methods and tools that may make your organization or project more productive and your products better and cheaper. But it is difficult to separate the claims from the reality. Many organizations perform experiments, run case studies or administer surveys to help them decide whether a method or tool is likely to make a positive difference in their particular situations. These investigations cannot be done without careful, controlled measurement and analysis. As we will see in Chapter 4, an evaluation's success depends on good experimental design, proper identification of the factors likely to affect the outcome, and appropriate measurement of factor attributes.

1.3.10 *Capability maturity assessment*

In the 1980s, the US Software Engineering Institute (SEI) proposed a capability maturity model (Humphrey, 1989) to measure a contractor's ability to develop quality software for the US government. This model assessed many different attributes of development, including use of tools, standard practices and more. To use the first version of the model, called a *process maturity assessment*, a contractor answered over 100 questions designed to determine the contractor's actual practices. The resulting "grade" was reported as a five-level scale, from "1" (*ad hoc* development dependent on individuals) to "5" (a development process that could be optimized based on continuous feedback).

There were many problems with the first model, as described by Bollinger and McGowan (Bollinger and McGowan, 1991), and the SEI has since revised its approach. The new model, called a *capability maturity assessment*, is based on key practices that every good contractor should be using. Other organizations, inspired by the SEI's goal, have developed other assessment models, in the hope that such evaluation will encourage improvement and enable organizations to compare and contrast candidate developers.

The notion of evaluating process maturity is very appealing, and we describe in Chapter 3 how process maturity can be useful in understanding what and when to measure. In Chapter 13, we will look at several maturity assessment techniques, including ISO 9000, examining the pros and cons of this type of evaluation.

1.4 SUMMARY

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

- The lessons of other engineering disciplines suggest that measurement must play a more significant role in software engineering.
- Software measurement is not a mainstream topic within software engineering. Rather it is a diverse collection of fringe topics (generally referred to as *software metrics*) that range from models for predicting software project costs at the specification stage to measures of program structure.
- Much software-metrics work has lacked the rigor associated with measurement in other engineering disciplines.
- General reasons for needing software-engineering measurement are not enough. Engineers must have specific, clearly stated objectives for measurement.
- We must be bold in our attempts at measurement. Just because no one has measured some attribute of interest does not mean that it cannot be measured satisfactorily.

We have set the scene for a new perspective on software metrics. To anchor a metrics program on a solid foundation of measurement theory, we turn to the next chapter. This rigorous basis will enable us to implement a scientific and effective approach to constructing, calculating and appropriately applying the metrics that we derive.

1.5 EXERCISES

1. Explain the role of measurement in determining the best players in your favorite sport.
2. How would you begin to measure the *quality* of a software product?
3. Consider some everyday measurements. What entities and attributes are being measured? What can you say about error margins in the measurements? Explain how the measuring process may affect the entity being measured.

4. In this chapter we gave examples of measurement objectives from both managers' and engineers' view points. Now consider the user's viewpoint. What measurement objectives might a software user have?
5. A commonly-used software quality measure in industry is the number of known errors per thousand lines of product source code. Compare the usefulness of this measure for developers and users. What are the possible problems with relying on this measure as the sole expression of software quality?

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 interpretating data. Measurement theory tells us the rules, laying the groundwork for developing and reasoning about all kinds of measurement. This rule-based approach is common in many sciences. For example, recall that mathematicians learned about the world by defining axioms for a geometry. Then, by combining axioms and using their results to support or refute their observations, they expanded their understanding and the set of rules that govern the behavior of objects. In the same way, we can use rules about measurement to codify our initial understanding, and then expand our horizons as we analyze our software.

However, just as there are several kinds of geometry (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

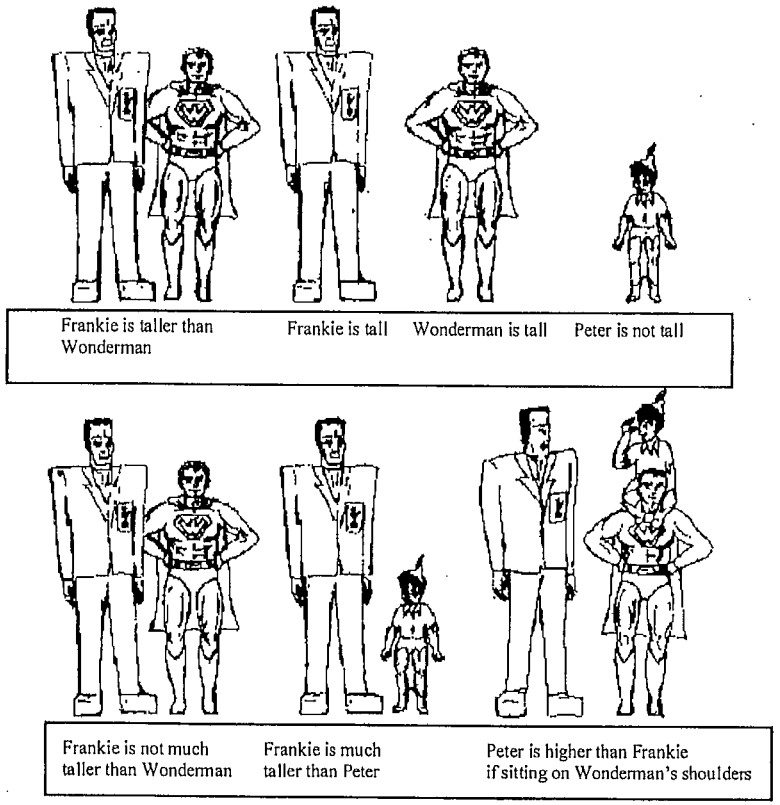


Figure 2.1: Some empirical relations for the attribute "height"

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

$H(x) > H(y)$

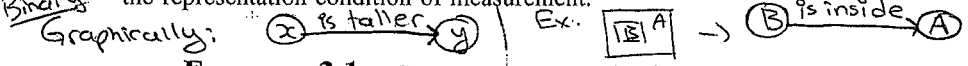
Therefore, we say that "taller than" is an **empirical relation** for height.

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

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

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

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



EXAMPLE 2.1: Suppose we are evaluating the four best-selling word-processing 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 program B. We can use this survey to define an empirical relation "greater functionality than" for word-processing programs; we say that program x has greater functionality than program y if the survey result for cell (x, y) exceeds 60% (we take 60% as representing a significant preference). 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.

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

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. Our immature understanding of user-friendliness may be the reason 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 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 a 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 upon, 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 responding to opinion polls. These questionnaires capture useful data. They enable us to establish the basis for empirical relations, characterizing properties so that formal measurement may be possible in the future.

2.1.2 *The rules of the mapping*

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

EXAMPLE 2.3: To measure a person's height, it is not enough 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, when we measure height 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

Likert scale
Give the respondent a statement with which to agree or disagree. For 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). For 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
For 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

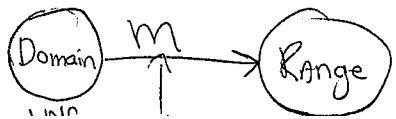
Comparative scale

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



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

Statement type	*Include?*	*Exclude?*
Executable		
Non-executable		
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		
Programmed	*Include?*	*Exclude?*
Generated with source code generators		
Converted with automatic translators		
Copied or reused without change		
Modified		
Removed		
Origin		
New work: no prior existence	*Include?*	*Exclude?*
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: Adapted from portion of US Software Engineering Institute checklist for lines-of-code count

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 7.

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

This statement implies 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:

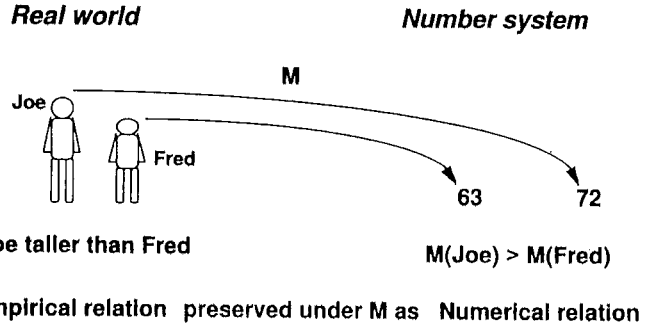


Figure 2.4: Representation condition

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

For the (ternary) empirical relation "x is 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 is higher than B if sitting on C's shoulders if and only if

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

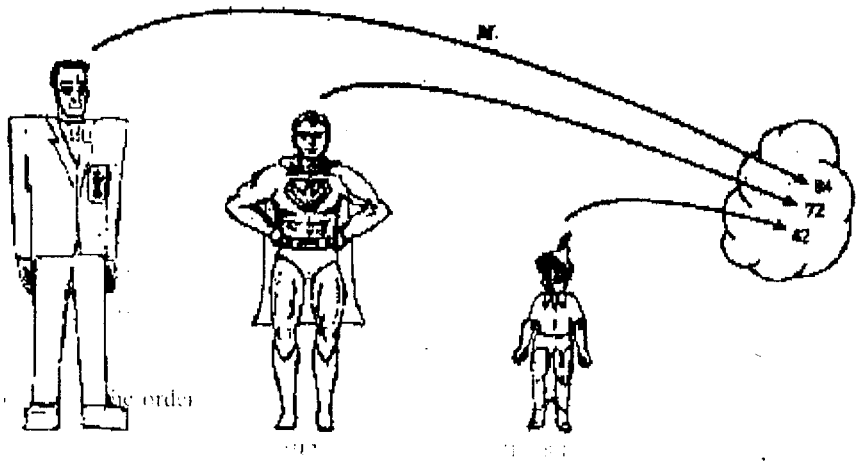


Figure 2.5: A measurement mapping

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, if we define the mapping in the following way:

$$M(\text{Wonderman}) = 72$$
$$M(\text{Frankie}) = 84$$
$$M(\text{Peter}) = 60$$

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

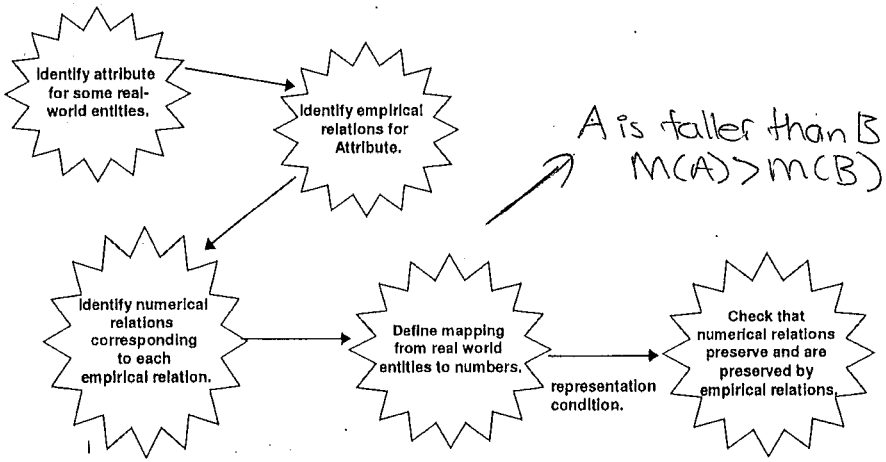


Figure 2.6: Key stages of formal measurement

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

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

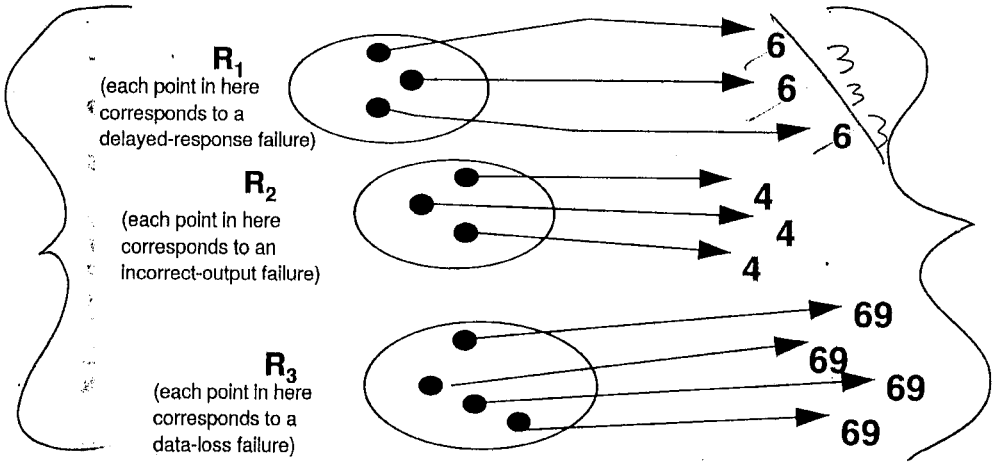


Figure 2.7: Measurement mapping

members from different classes into different numbers. For example, the mapping M, illustrated in Figure 2.7, assigns the mapping as:

$$\left.\begin{array}{c} M(\text{each delayed response}) = 6 \\ M(\text{each incorrect output}) = 4 \\ M(\text{each data loss}) = 69 \end{array}\right\}$$

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

Suppose next 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

$$\begin{array}{c} x \text{ is in } R_3 \text{ and } y \text{ is in } R_2 \text{ or } R_1, \text{ or} \\ x \text{ is in } R_2 \text{ and } y \text{ is in } R_1 \end{array}$$

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:

$$\begin{array}{c} M(\text{each delayed response}) = 3 \\ M(\text{each incorrect output}) = 4 \\ M(\text{each data loss}) = 69 \end{array}$$

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

There is nothing wrong with using the same representation in different ways, or 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.

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

How good a measure is faults per KLOC? The answer depends entirely on the entity-attribute pair connected by the mapping. Intuitively, most of us would accept that faults per KLOC is a good measure of the rate at which faults are found for the testing process (example 6). However, it is not such a good measure of efficiency of the tester (example 7), because intuitively we feel that we should also take into account the difficulty of understanding and testing the program under scrutiny. This measure may be reasonable when comparing two testers of the same program, though. Faults per KLOC is not likely to be a good measure of quality of the program code (example 8); 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 a combination of different languages. It might 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 giving 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 hurry to generate numbers without considering their real meaning.

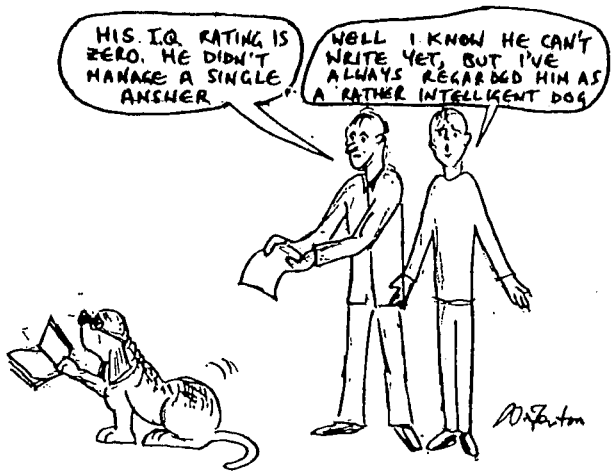


Figure 2.8: Using a suspect definition

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 8.

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,

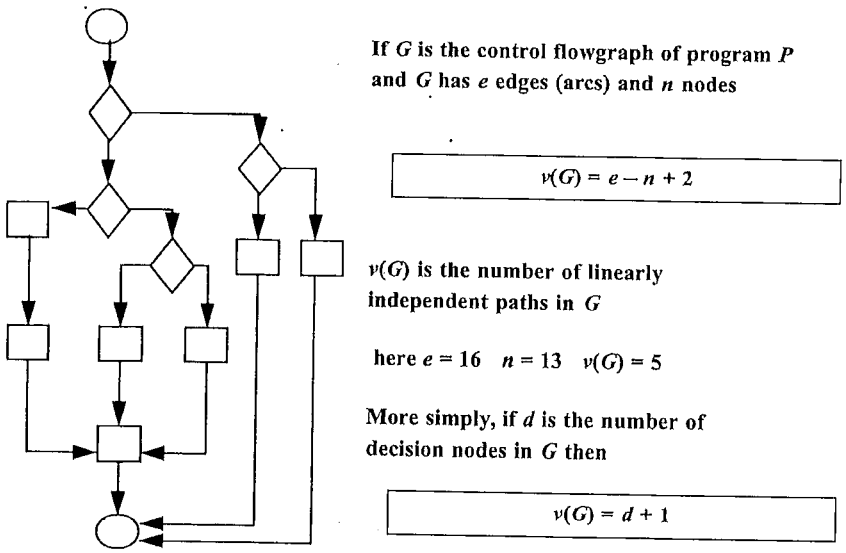


Figure 2.9: Computing McCabe's cyclomatic number

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.

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 among the three is

$$\text{density} = \frac{\text{mass}}{\text{volume}}$$

Similarly, the speed of a moving object is most accurately measured indirectly using measures of distance and time. Thus, direct measurement forms the building blocks for our assessment, but many interesting attributes are best measured 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, because 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.

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.

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	$\frac{\text{number of initial requirements}}{\text{total number of requirements}}$
Test effectiveness ratio	$\frac{\text{number of items covered}}{\text{total number of items}}$
System spoilage	$\frac{\text{effort spent fixing faults}}{\text{total project effort}}$

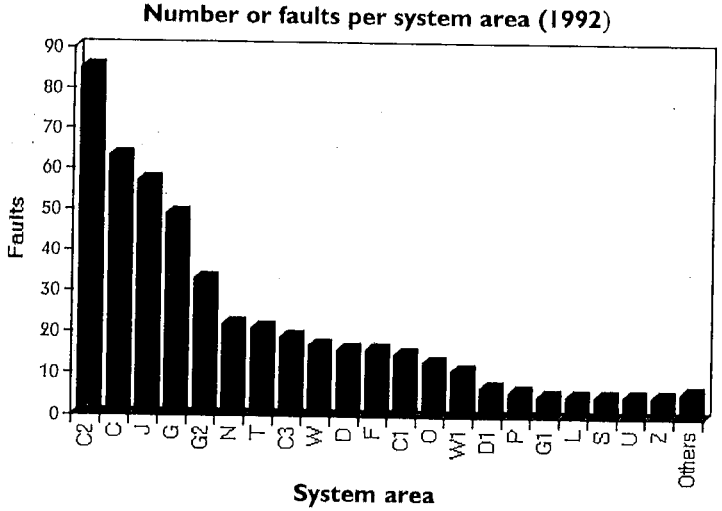


Figure 2.10: Using direct measurement to assess a product (Pfleeger, Fenton, and Page, 1994)

The graph shows the number of faults in each system area of a large, important software system in the UK. 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

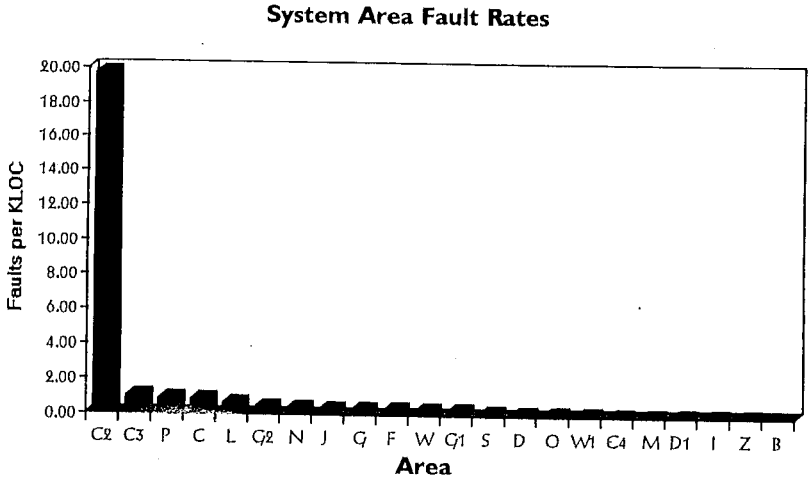


Figure 2.11: Using indirect measurement to assess a product (Pfleeger, Fenton, and Page, 1994)

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.

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 2.10: 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 2.11: 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 12.

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 2.10, 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 had only a program specification, and we wished to know roughly how many hard-copy pages the final implementation would 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 2.11, 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 2.12: 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 2.12, 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 2.13: 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 10.

Remember that a model defines an association among 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 2.14: 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 *et al.*, 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, redesign 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 do 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 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, since 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 levels 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 $\frac{1}{12}$). 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*

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

- 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 2.15: Suppose that we are investigating the set of all known software faults in our code, and we are trying to capture the location of the

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

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

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

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

2.3.2 *Ordinal scale*

The ordinal scale is often useful to augment the nominal scale with information about an ordering of the classes or categories. The ordering leads to analysis not possible with nominal measures. Thus, 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 2.16: Suppose our set of entities is a set of software modules, and the attribute we wish to capture quantitatively is "complexity". Initially,

we may define five distinct classes of module complexity: "trivial", "simple", "moderate", "complex" and "incomprehensible". There is an implicit order relation of "less complex than" on these classes; that is, all trivial modules are less complex than simple modules, which are less complex than moderate modules, 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.

$$M_1(x) = \begin{cases} 1 \text{ if } x \text{ is trivial} \\ 2 \text{ if } x \text{ is simple} \\ 3 \text{ if } x \text{ is moderate} \\ 4 \text{ if } x \text{ is complex} \\ 5 \text{ if } x \text{ is incomprehensible} \end{cases} \quad M_2(x) = \begin{cases} 1 \text{ if } x \text{ is trivial} \\ 2 \text{ if } x \text{ is simple} \\ 3 \text{ if } x \text{ is moderate} \\ 4 \text{ if } x \text{ is complex} \\ 10 \text{ if } x \text{ is incomprehensible} \end{cases} \quad M_3(x) = \begin{cases} 0.1 \text{ if } x \text{ is trivial} \\ 1001 \text{ if } x \text{ is simple} \\ 1002 \text{ if } x \text{ is moderate} \\ 4570 \text{ if } x \text{ is complex} \\ 4573 \text{ if } x \text{ is incomprehensible} \end{cases}$$

However, neither M_4 nor M_5 is valid:

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

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 2.17: 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 2.18: Recall the five categories of complexity described in Example 2.16. Suppose that the difference in complexity between a trivial and simple system is the same as that between a simple and moderate system. Then any interval measure of complexity must preserve these differences. Where this equal step applies to each class, we have an attribute measurable on an interval scale. The following measures have this property and satisfy the representation condition:

$$M_1(x) = \begin{cases} 1 \text{ if } x \text{ is trivial} \\ 2 \text{ if } x \text{ is simple} \\ 3 \text{ if } x \text{ is moderate} \\ 4 \text{ if } x \text{ is complex} \\ 5 \text{ if } x \text{ is incomprehensible} \end{cases} \quad M_2(x) = \begin{cases} 0 \text{ if } x \text{ is trivial} \\ 2 \text{ if } x \text{ is simple} \\ 4 \text{ if } x \text{ is moderate} \\ 6 \text{ if } x \text{ is complex} \\ 8 \text{ if } x \text{ is incomprehensible} \end{cases} \quad M_3(x) = \begin{cases} 3.1 \text{ if } x \text{ is trivial} \\ 5.1 \text{ if } x \text{ is simple} \\ 7.1 \text{ if } x \text{ is moderate} \\ 9.1 \text{ if } x \text{ is complex} \\ 11.1 \text{ if } x \text{ is incomprehensible} \end{cases}$$

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

$$M = aM' + b$$

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

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

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

$$M_3 = 2M_1 + 1.1$$

EXAMPLE 2.19: The timing of an event's occurrence is a classic use of interval scale measurement. We can measure the timing in units of years, days, hours, or some other standard measure, where each time is noted relative to a given fixed event. We use this convention 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 2.20: The length of physical objects is measurable on a ratio scale, enabling us to make statements about how one entity is twice as long as another. The zero element is theoretical, in the sense that we can think of an object as having no length at all; thus, the zero-length object exists as a limit of things that get smaller and smaller. We can measure length in inches, feet,

centimeters, meters, and more, where each different measure preserves the relations about length that we observe in the real world. To convert from one length measure to another, we can use a transformation of the form $M = aM'$ where a is a constant. Thus, to convert feet F to inches I, 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**, because it preserves the ratios in M' as they are transformed to M.

EXAMPLE 2.21: The length of software code is also measurable on a ratio scale. As with other physical objects, we have empirical relations like "twice as long". The notion of a zero-length object exists – an empty piece of code. We can measure program length in a variety of ways, including 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 the absolute scale.

EXAMPLE 2.22: We saw in Example 2.21 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 misinterpretation 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 2.23: 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 M and M' are different measures of height, then the statements

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

and

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

are either both true or both false. 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 40°C and in London 20°C. Then on the Celsius scale, the statement is true. However, on the Fahrenheit scale, Tokyo is 104°F while London is 68°F.

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 35°C in Tokyo and 25°C (a difference of 10) in London, while today it is 40°C in Tokyo and 20°C in London (a difference of 20). If we transform these temperatures to the Fahrenheit scale, we find that yesterday's temperatures were 95°F in Tokyo and 77°F London (a difference of 18); today's are 104°F in Tokyo and 68°F in London (a difference of 36). Thus, the truth value is preserved with the transformation.

Failure x is twice as critical as failure y.
This statement is not meaningful if 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 in increasing order of criticality. 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 2.24: Suppose we define a crude notion of speed of software programs, and we rank three word processing 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 overly-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 *9.1*
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 central tendency 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 2.25: 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 truth value to be preserved, the 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 2.25 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 $\{y_1,\ldots,y_m\}$ 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_js" 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

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

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 6.

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 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. Although no measurement is truly objective (because there is always some degree of subjectivity about the entities and attributes) some measures are clearly more subjective than others. 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.

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 composed. (We sometimes call these components **sub-attributes.**) For example, suppose that

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

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

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.

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.

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(Plane) < m(Train)
2. m(Plane) > m(Train)
3. m(Plane) = m(Train)

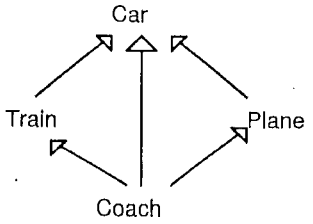


Figure 2.13: Quality relationships based on rule and collected data

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:

$$m(\text{Transport}) = (\text{Journey time, Cost per mile})$$

Then we define the actual pairs:

$$m(\text{Car})=(3,\ 1.5)$$
$$m(\text{Train})=(5,\ 2)$$
$$m(\text{Plane})=(3.5,\ 3.5)$$
$$m(\text{Coach})=(7,\ 4)$$

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

$$(x, y) \text{ superior to } (x', y') \text{ if } x<x' \text{ and } y<y'$$

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 2.26: 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.

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 9126, for measuring software quality that explicitly defines software quality as the combination of six distinct sub-attributes. We will discuss the details of this standard in Chapter 9. However, it is important to note here that the standard reflects a widely held view that no single

Table 2.11: Comparing four compilers

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

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 2.27: 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:

$$x \text{ is more complex than } z$$

and

$$z \text{ is more complex than } y$$

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,

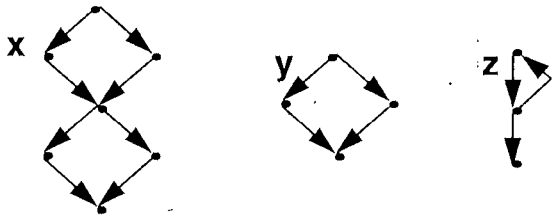


Figure 2.14: Three program flowgraphs

clearly useful in counting the number of linearly independent paths in the program flowgraph, should not be used as a comprehensive measure of complexity.

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 2.28: We want to buy a word-processing program for our home computer. It is likely that the decision will be based on a collection of attributes, such as price, reliability and usability. Since the word processor will be used primarily for letter-writing, we may give price a heavier weighting than reliability when ranking the programs. However, if we are buying a database package for use in a critical air-traffic control system application, 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 are 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 $M_1, M_2, \ldots, M_n$. That is, $M = f(M_1, M_2, \ldots, M_n)$ for some function f. We say that M' is a **rescaling** of M if there are rescalings $M_1', M_2', \ldots, M_n'$, of $M_1, M_2, \ldots, M_n$ respectively, such that $M' = f(M_1', M_2', \ldots, M_n')$.

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

EXAMPLE 2.29: 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 α_1 and every rescaling of V must be of the form $\alpha_2 V$ for some α_2. Therefore, every rescaling of d has the form

$$\frac{\alpha_1 m}{\alpha_2 V} = \frac{\alpha_1}{\alpha_2}\left(\frac{m}{V}\right) = \frac{\alpha_1}{\alpha_2} d = \alpha d \quad \left(\text{where } \alpha = \frac{\alpha_1}{\alpha_2}\right)$$

Now, we can define scale types for indirect scales in exactly the same way as for direct scales. Example 2.29 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 2.30: 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 2.31: In Example 2.11, 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 (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 2.32: Halstead developed a theory of software physics (discussed in Chapter 3), 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 2.33: 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 2.32, 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 2.16.

11 In Example 2.18, 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 admissable 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 DOS 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 five 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 Ms. (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 2.26 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 2.31 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, MA, 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, New York, 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. 311–32, 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. 677–80, 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 texts 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.

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. 215–23, 1975.

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

Campbell, N.R., *Physics: The Elements*, Cambridge University Press, Cambridge, MA, 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*, Wiley, New York, 1992.

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

3 A goal-based framework for software measurement

Chapter 1 described measurement's essential role in good software engineering practice, and Chapter 2 explained a general theory of measurement. In this chapter, we return to software engineering, presenting a conceptual framework for the diverse software-measurement activities that contribute to your organization's software practices. These practices may include not only your usual development and maintenance activities, but also any experiments and case studies you may perform as you investigate new techniques and tools.

The framework presented here is based on three principles: classifying the entities to be examined, determining relevant measurement goals, and identifying the level of maturity that your organization has reached. The first section explores the classification of measures as process, product or resource, and whether the attribute is internal or external. The next section presents the Goal–Question–Metric (GQM) paradigm to tie measurement to the overall goals of the project and process. Then, we show how process maturity and GQM can work hand-in-hand to suggest a rich set of measurements as we measure what is visible in the process. We also look at measurement validation: the process of ensuring that we are measuring what we say we are, so that we satisfy the representation condition introduced in Chapter 2. Finally, we review the major activities of software measurement to see how they relate to the framework presented.

3.1 CLASSIFYING SOFTWARE MEASURES

As we have seen in Chapter 2, the first obligation of any software-measurement activity is identifying the entities and attributes we wish to measure. In software, there are three such classes:

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

A process is usually associated with some timescale. That is, the activities in the process are ordered or related in some way that depends on time, so that one activity must be completed before another can begin. The timing can be explicit, as when design must be complete by October 31, or implicit, as when a flow diagram shows that design must be completed before coding can begin.

Resources and products are associated with the process. Each process activity has resources and products that it uses, as well as products that are produced. Thus, the product of one activity may feed another activity. For example, a design document can be the product of the design activity which is then used as input for the coding activity.

Many of the examples that we use in this book relate to the development process or the maintenance process. But the concepts that we introduce apply to any process: the reuse process, the configuration management process, the testing process, and so on. In other words, measurement activities may focus on any process and need not be concerned only with the comprehensive development process. As we shall see later in this chapter, our choice of measurements depends on our measurement goals.

Within each class of entity, we distinguish between internal and external attributes:

- **Internal attributes** of a product, process or resource are those that can be measured purely in terms of the product, process or resource itself. In other words, an internal attribute can be measured by examining the product, process or resource on its own, separate from its behavior.
- **External attributes** of a product, process or resource are those that can be measured only with respect to how the product, process or resource relates to its environment. Here, the behavior of the process, product or resource is important, rather than the entity itself.

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

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

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

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

3.1.1 *Processes*

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

Table 3.1: Components of software measurement

ENTITIES	ATTRIBUTES	
Products	*Internal*	*External*
Specifications	size, reuse, modularity, redundancy, functionality, syntactic correctness, ...	comprehensibility, maintainability, ...
Designs	size, reuse, modularity, coupling, cohesiveness, functionality, ...	quality, complexity, maintainability, ...
Code	size, reuse, modularity, coupling, functionality, algorithmic complexity, control-flow structuredness, ...	reliability, usability, maintainability, ...
Test data	size, coverage level, ...	quality, ...
...	...	...
Processes		
Constructing specification	time, effort, number of requirements changes, ...	quality, cost, stability, ...
Detailed design	time, effort, number of specification faults found, ...	cost, cost-effectiveness, ...
Testing	time, effort, number of coding faults found, ...	cost, cost-effectiveness, stability, ...
...	...	...
Resources		
Personnel	age, price, ...	productivity, experience, intelligence, ...
Teams	size, communication level, structuredness, ...	productivity, quality, ...
Software	price, size, ...	usability, reliability, ...
Hardware	price, speed, memory size, ...	reliability, ...
Offices	size, temperature, light, ...	comfort, quality, ...
...	...	...

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

For example, we may be reviewing our requirements to ensure their quality before turning them over to the designers. To measure the effectiveness of the review process, we can measure the number of requirements errors found during specification. Likewise, we can measure the number of faults found during integration testing to determine how well we are doing. And the number of personnel working on the project between May 1 and September 30 can give us insight into the resources needed for the development process.

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

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

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

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

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

In some cases, we may want to measure properties of a process which itself consists of a number of distinct processes:

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

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

We often propose objective measures of external attributes in terms of internal attributes. For example, measures of the effectiveness of code maintenance can be defined in terms of the number of faults discovered and the number of faults corrected. In the AT&T study of Example 3.2, inspection effectiveness was defined as average faults detected per thousand lines of code inspected. In each case, we examine the process of interest and decide what kind of information would help us to understand, control or improve the process. We will investigate process attributes in Chapter 12.

3.1.2 *Products*

Products are not restricted to the items that management is committed to deliver to the customer. Any artifact or document produced during the software life cycle can be measured and assessed. For example, developers often build prototypes for examination only, so that they can understand requirements or evaluate possible designs; these prototypes may be measured in some way. Likewise, test harnesses that are constructed to assist in system testing may be measured; system size measurements should include this software if they are to be used to determine team productivity. And documents, such as the user guide or the customer specification document, can be measured for size, quality, and more.

3.1.2.1 External product attributes

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

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

3.1.2.2 Internal product attributes

Internal product attributes are sometimes easy to measure. We can determine the size of a product by measuring the number of pages it fills or the number of words it contains, for example. Since the products are concrete, we have a better understanding of attributes like size, effort, and cost. Other internal product attributes are more difficult to measure, because opinions differ as to what they mean and how to measure them. For example, there are many aspects of code complexity, and no consensus about what best measures it. We will explore this issue in Chapter 8.

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

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

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

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

> EXAMPLE 3.5: A software system design document may consist of a large number of module design documents. Average module size, an attribute of the system design, can be derived by calculating the size of each module. For example, suppose the designs are represented by data-flow diagrams. We can measure size by the number of bubbles in each diagram, and then calculate the comprehensive measure according to Table 3.2.

3.1.2.3 The importance of internal attributes

Many software-engineering methods proposed and developed in the last 25 years provide rules, tools, and heuristics for producing software products. Almost invariably, these methods give structure to the products; it is claimed that this structure makes them easier to understand, analyze, and test. The structure involves two aspects of development:

Table 3.2: Definition of example design measurements

Entity	Entity Type	Attribute	Proposed Measure	Type
Module design document D_i	Product	Size	Number_of_bubbles$_i$	Direct
System design document $\{D_1, \ldots, D_n\}$	Product	Average module size	$1/n \sum_{i=1}^{n}$ Number_of_bubbles$_i$	Indirect

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

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

EXAMPLE 3.6: One of the most widely respected books in software engineering is Brooks' *Mythical Man-Month* (Brooks, 1975). There, Brooks describes the virtues of top-down design:

> A good top-down design avoids bugs in several ways. First, the clarity of structure and representation makes the precise statement of requirements and functions of the modules easier. Second, the partitioning and independence of modules avoids system bugs. Third, the suppression of detail makes flaws in the structure more apparent. Fourth, the design can be tested at each of its refinement steps, so testing can start earlier and focus on the proper level of detail at each step.

Similarly, the notion of high module cohesion and low module coupling is the rationale for most structured design methods (Yourdon and Constantine, 1979). Designs that possess these attributes are assumed to lead to more reliable and maintainable code.

Brooks and Yourdon assume what most other software engineers assume: that good internal structure leads to good external quality. Although intuitively appealing, the connection between internal attribute values and the resulting external attribute values has rarely been established, in part because (as we shall see in Chapters 4 and 15) it is sometimes difficult to perform controlled experiments and confirm relationships between attributes. At the same time, there is little understanding of measurement validation and the proper ways to demonstrate these relationships. Software measurement can help us to understand and confirm relationships empirically; a key to the success of this analysis is the ability to provide accurate and meaningful measures of internal product attributes.

3.1.2.4 Internal attributes and quality control and assurance

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

3.1.2.5 Validating composite measures

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

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

In the same way that economists want to control the economy and make a country more productive, we want to measure and control the quality of our products, and we usually do so by measuring and controlling a number of internal (structural) product

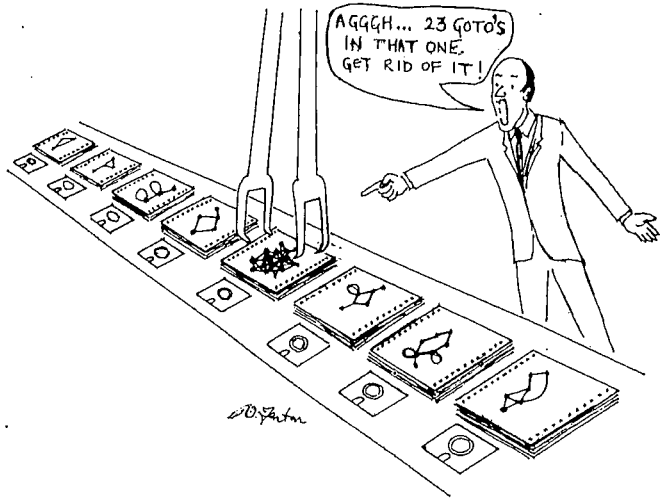


Figure 3.1: Using internal measures for quality control and assurance

Table 3.3: Examples of AT&T goals, questions and metrics (Barnard and Price, 1994)

Goal	Questions	Metrics
Plan	How much does the inspection process cost?	Average effort per KLOC Percentage of reinspections
	How much calendar time does the inspection process take?	Average effort per KLOC Total KLOC inspected
Monitor and control	What is the quality of the inspected software?	Average faults detected per KLOC Average inspection rate Average preparation rate
	To what degree did the staff conform to the procedures?	Average inspection rate Average preparation rate Average lines of code inspected Percentage of reinspections
	What is the status of the inspection process?	Total KLOC inspected
Improve	How effective is the inspection process?	Defect removal efficiency Average faults detected per KLOC Average inspection rate Average preparation rate Average lines of code inspected
	What is the productivity of the inspection process?	Average effort per fault detected Average inspection rate Average preparation rate Average lines of code inspected

Even when the metrics are expressed as an equation or relationship, the definition is not always clear and unambiguous. The tree does not tell you how to measure a line of code or a function point, only that some measure of code size is needed to answer a question about productivity. Additional work is needed to define each metric. In cases where no objective measure is available, subjective measures must be identified.

In general, typical goals are expressed in terms of productivity, quality, risk, customer satisfaction, and the like, coupled with verbs expressing the need to assess, evaluate, improve, or understand. It is important that the goals and questions be understood in terms of their audience: a productivity goal for a project manager may be different from that for a department manager or corporate director. To aid in generating the goals, questions, and metrics, Basili and Rombach provided a series of templates:

Templates for goal definition

- **Purpose:** To (characterize, evaluate, predict, motivate, etc.) the (process, product, model, metric, etc.) in order to (understand, assess, manage, engineer, learn, improve, etc.) it.

  Example: To *evaluate* the *maintenance process* in order to *improve* it.

- **Perspective:** Examine the (cost, effectiveness, correctness, defects, changes, product measures, etc.) from the viewpoint of the (developer, manager, customer, etc.)

  Example: Examine the *cost* from the viewpoint of the *manager*.

- **Environment:** The environment consists of the following: process factors, people factors, problem factors, methods, tools, constraints, ...

  Example: The maintenance staff are poorly motivated programmers who have limited access to tools.

Basili and Rombach also provide separate guidelines for defining product-related questions and process-related questions. Steps involve defining the process or product, defining the relevant attributes, and obtaining feedback related to the attributes. What constitutes a goal or question may be vague, and several levels of refinement may be required for certain goals and questions; that is, a goal may first have to be related to a series of subgoals before questions can be derived.

We can relate the GQM templates to the attribute framework introduced earlier in this chapter. A goal or question can be associated with at least one pair of entities and attributes. Thus, a goal is stated, leading to a question that should be answered so that we can tell if we have met our goal; the answer to the question requires that we measure some attribute of an entity (and possibly several attributes of an entity, or several attributes of several entities). The use of the measure is determined by the goals and questions, so that assessment, prediction, and motivation are tightly linked to the data analysis and reporting.

Thus, GQM complements the entity–attribute measurement framework. The results of a GQM analysis are a collection of measurements related by goal tree and overall model. However, GQM does not address issues of measurement scale, objectivity, or feasibility. So the GQM measures should be used with care, remembering the overall goal of providing useful data that can help to improve our processes, products and resources.

3.2.2 *Measurement and process improvement*

Measurement offers visibility into the ways in which the processes, products, resources, methods, and technologies of software development relate to one another. As we have seen, measurements can help us to answer questions about the effectiveness of techniques or tools, the productivity of development activities (such as testing or configuration management), the quality of products, and more. In addition,

measurement allows us to define a baseline for understanding the nature and impact of proposed changes. Finally, measurement allows managers and developers to monitor the effects of activities and changes on all aspects of development, so that action can be taken as early as possible to control the final outcome should actual measurements differ significantly from plans. Thus, measurement is useful for:

- understanding
- establishing a baseline
- assessing and predicting

We have seen how these goals can be interpreted using GQM. But measurement is useful only in the larger context of assessment and improvement. Choosing metrics, collecting data, analyzing the results and taking appropriate action require time and resources; these activities make sense only if they are directed at specific improvement goals.

Some development processes are more mature than others, as noted by the Software Engineering Institute's (SEI's) reports on process maturity and capability maturity (Humphrey, 1989; Paulk, 1991). While some organizations have clearly-defined processes, others are more variable, changing significantly with the people who work on the projects. The SEI has suggested that there are five levels of process maturity, ranging from *ad hoc* (the least predictable and controllable) to *repeatable*, *defined*, *managed*, and *optimizing* (the most predictable and controllable); this notion is illustrated in Figure 3.3.

The SEI distinguishes one level from another in terms of key process activities going on at each level. This and other assessment schemes will be discussed in detail in Chapter 14, where we see how measurement supports a variety of improvement models (such as ISO 9000, SPICE and Bootstrap).

But all of these models share a common goal and approach, namely that they use process visibility as a key discriminator among a set of "maturity levels." That is, the more visibility into the overall development process, the higher the maturity and the better managers and developers can understand and control their development and maintenance activities. At the lowest levels of maturity, the process is not well understood at all; as maturity increases, the process is better-understood and better-defined. At each maturity level, measurement and visibility are closely related: a developer can measure only what is visible in the process, and measurement helps to increase visibility. Thus, the five-level maturity scale such as the one employed by the SEI is a convenient context for determining what to measure first and how to plan a measurement program that grows to embrace additional aspects of development and management.

Table 3.4 presents an overview of the types of measurement suggested by each maturity level, where the selection depends on the amount of information visible and available at a given maturity level. Level 1 measurements provide a baseline for comparison as you seek to improve your processes and products; level 2 measurements focus on project management, while level 3 measures the intermediate and final

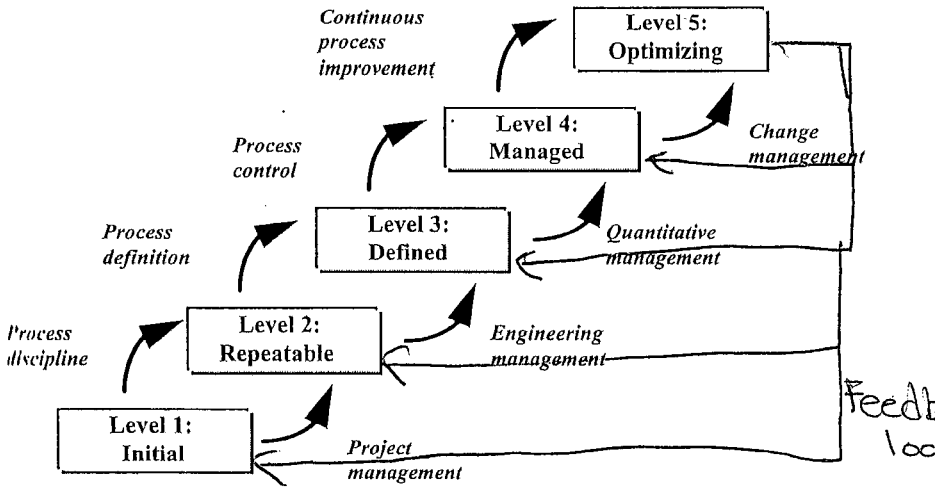


Figure 3.3: The Software Engineering Institute's levels of process maturity

products produced during development. The measurements at level 4 capture characteristics of the development process itself to allow control of the individual activities of the process. A level 5 process is mature enough and managed carefully enough to allow measurements to provide feedback for dynamically changing the process during a particular project's development.

Let us look at each maturity level in more detail.

Level 1: *ad hoc*

The initial level of process maturity is characterized by an *ad hoc* approach to the software-development process. That is, inputs to the process are ill-defined; while outputs are expected, the transition from inputs to outputs is undefined and uncontrolled. Similar projects may vary widely in their productivity and quality characteristics because of lack of adequate structure and control. For this level of

Table 3.4: Overview of process maturity and measurement (Pfleeger and McGowan, 1990)

Maturity level	Characteristics	Type of metrics to use
5. Optimizing	Improvement fed back to the process	Process plus feedback for changing the process
4. Managed	Measured process	Process plus feedback for control
3. Defined	Process defined and institutionalized	Product
2. Repeatable	Process dependent on individuals	Project management
1. Initial	Ad hoc	Baseline

process maturity, it is difficult even to write down or depict the overall process; the process is so reactive and ill-defined that visibility is nil and comprehensive measurement difficult. You may have goals relating to improved quality and productivity, but you do not know what current levels of quality and productivity are. For this level of process maturity, baseline measurements are needed to provide a starting point for measuring improvement as maturity increases. If your organization is at level 1, you should concentrate on imposing more structure and control on the process, in part to enable more meaningful measurement.

Level 2: repeatable

The second process level, called repeatable, identifies the inputs and outputs of the process, the constraints (such as budget and schedule), and the resources used to produce the final product. The process is repeatable in the same sense that a subroutine is repeatable: proper inputs produce proper outputs, but there is no visibility into how the outputs are produced. Asked to define and describe the process, you and your development team can draw no more than a diagram similar to Figure 3.4. This figure shows a repeatable process as a simplified Structured Analysis and Design Technique (SADT) diagram, with input on the left, output on the right, constraints at the top, and resources on the bottom. For example, requirements may be input to the process, with the software system as output. The control arrow represents such items as schedule and budget, standards, and management directives, while the resources arrow can include tools and staff.

Since it is possible to measure only what is visible, Figure 3.4 suggests that project management measurements make the most sense for a repeatable process. That is, since the arrows are all that is visible, we can associate measurements with each arrow in the process diagram. Thus, for a repeatable process, measures of the input might include the size and volatility of the requirements. The output may be measured in terms of system size (functional or physical), the resources as overall staff effort, and the constraints as cost and schedule in dollars and days, respectively.

Level 3: defined

The defined level of maturity (level 3) differs from level 2 in that a defined process provides visibility into the "construct the system" box. At level 3, intermediate activities are defined, and their inputs and outputs are known and understood. This additional structure means that the input to and output from the intermediate activities can be examined, measured, and assessed, since these intermediate products are well-defined and visible. Figure 3.5 shows a simple example of a defined process with three typical activities. However, different processes may be partitioned into more distinct functions or activities.

Because the activities are delineated and distinguished from one another in a defined process, measurement of product attributes should begin no later than level 3. Defects discovered in each type of product can be tracked, and you can compare the defect density of each product with planned or expected values. In particular,

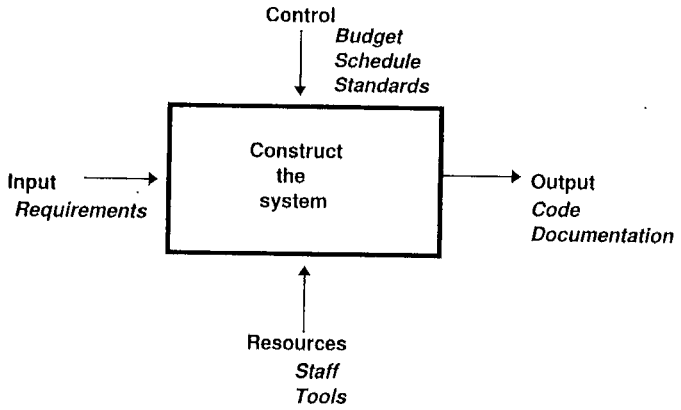


Figure 3.4: A repeatable process (level 2)

early product measures can be useful indicators of later product measures. For example, the quality of the requirements or design can be measured and used to predict the quality of the code. Such measurements use the visibility in the process to provide more control over development: if requirements quality is unsatisfactory, additional work can be expended on the requirements before the design activity begins. This early correction of problems helps not only to control quality but also to improve productivity and reduce risk.

Level 4: managed

A managed process (level 4) adds management oversight to a defined process, as shown in the example process of Figure 3.6. Here, you can use feedback from early project activities (for example, problem areas discovered in design) to set priorities for current activities (for example, redesign) and later project activities (for example, more extensive review and testing of certain code, and a changed sequence for integration). Because you can compare and contrast, the effects of changes in one activity can be tracked in the others. By level 4, the feedback determines how resources are deployed; the basic activities themselves do not change. At this level, you can evaluate the effectiveness of process activities: how effective are reviews?

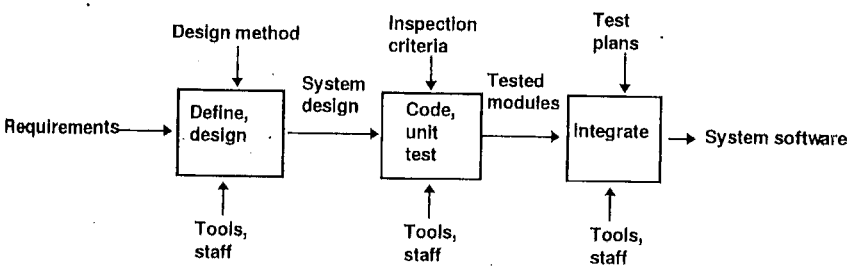


Figure 3.5: A defined process (level 3)

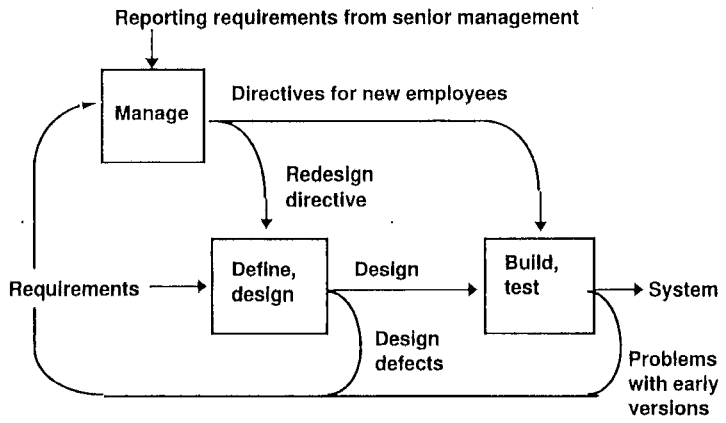


Figure 3.6: A managed process (level 4)

Configuration management? Quality assurance? Defect-driven testing? You can use the collected measures to stabilize the process, so that productivity and quality will match expectations.

A significant difference between levels 3 and 4 is that level 4 measurements reflect characteristics of the overall process and of the interaction among and across major activities. Management oversight relies on a metrics database that can provide information about such characteristics as distribution of defects, productivity and effectiveness of tasks, allocation of resources, and the likelihood that planned and actual values will match.

Level 5: optimizing

An optimizing process is the ultimate level of process maturity; it is depicted in Figure 3.7. Here, measures from activities are used to improve the process, possibly by removing and adding process activities, and changing the process structure dynamically in response to measurement feedback. Thus, the process change can affect the organization and project as well as the process. Results from one or more ongoing or completed projects may also lead to a refined, different development process for future projects. The *spiral model* (Boehm, 1988) is an example of such a dynamically changing process, responding to feedback from early activities in order to reduce risk in later ones.

This tailoring of the process to the situation is indicated in the figure by the collection of process boxes labeled $T_0, T_1, \ldots, T_n$. At time T_0, the process is as represented by box T_0. However, at time T_1, management has the option of revising or changing the overall process. For example, suppose you begin development with a standard waterfall approach. As requirements are defined and design is begun, measurements may indicate a high degree of uncertainty in the requirements. Based on this information, you may decide to change the process to one that prototypes the requirements and the design, so that some of the uncertainty can be resolved before

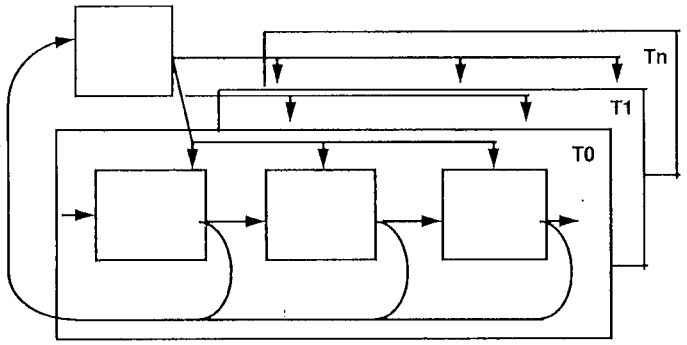


Figure 3.7: An optimizing process (level 5)

you make substantial investment in implementation of the current design. In this way, being able to optimize the process gives you maximum flexibility in development. Measurements act as sensors and monitors, and the process is not only under your control but you can change it significantly in response to warning signs.

The five categories of measurements described here – baseline, project management, product, process for feedback, process for control – are meant to be nested. That is, at a given maturity level, you can collect the measurements for that level and all levels below it. However, it is important to note that process maturity does not involve a discrete set of five possible ratings. Instead, maturity represents relative locations on a continuum from 1 to 5. An individual process is assessed or evaluated along many dimensions, and some parts of the process can be more mature or visible than others. For example, a repeatable process may not have well-defined intermediate activities, but the design activity may indeed be clearly defined and managed. The process-visibility diagrams presented here are meant only to give a general depiction of typical processes. It is essential that you begin any measurement program by examining the process model and determining what is visible. The figures and tables should not proscribe measurement simply because the overall maturity level is a particular integer; if one part of a process is more mature than the rest, measurement can enhance the visibility of that part and help to meet overall project goals, at the same time bringing the rest of the process up to a higher level of maturity. Thus, in a repeatable process with a well-defined design activity, design quality metrics may be appropriate and desirable, even though they are not generally recommended for level 2.

Moreover, it is important to remember that successful metrics programs start small and grow according to the goals and needs of a particular project (Rifkin and Cox, 1991). Your metrics program should begin by addressing the critical problems or goals of the project, viewed in terms of what is meaningful or realistic at the project's maturity level. The process maturity framework then acts as a guideline for how to expand and build a metrics program that not only takes advantage of visibility and

maturity but also enhances process improvement activities. We will discuss these issues in more detail in Chapters 13 and 14.

Next, we explain how the process maturity framework, coupled with understanding of goals, can be used to build a comprehensive measurement program.

3.2.3 *Combining GQM with process maturity*

Suppose you are using the Goal–Question–Metric paradigm to decide what your project should measure. You may have identified at least one of the following high-level goals:

- improving productivity
- improving quality
- reducing risk

Within each category, you can represent the goal's satisfaction as a set of subgoals, each of which can be examined for its implications for resources, products and process. For example, the goal of improving productivity can be represented as several subgoals affecting resources:

- ensuring adequate staff skills
- ensuring adequate managerial skills
- ensuring adequate host software engineering technology

Similarly, to improve productivity using products can mean

- identifying problems early in the life cycle
- using appropriate technology
- reusing previously built products

Next, for each subgoal, you generate questions that reflect the areas of deepest concern. For example, if "improving productivity" is your primary goal, and "ensuring adequate staff skills" is an important subgoal, you create a list of questions that you may be interested in having answered, such as:

1. Does project staffing have the right assortment of skills?
2. Do the people on the project have adequate experience?

Similarly, if you have chosen "improving quality" with a subgoal of "improving the quality of the requirements", then the related questions might include:

1. Is the set of requirements clear and understandable?
2. Is the set of requirements testable?
3. Is the set of requirements reusable?
4. Is the set of requirements maintainable?
5. Is the set of requirements correct?

However, before you identify particular measurements to help you answer these questions, it is useful to determine your process maturity level. Since process maturity suggests that you measure only what is visible, the incorporation of process maturity with GQM paints a more comprehensive picture of what measures will be most useful to you.

For example, suppose you want to answer the question "Is the set of requirements maintainable?" If you have specified a process at level 1, then the project is likely to have ill-defined requirements. Measuring requirements characteristics is difficult at this level, so you may choose to count the number of requirements and changes to those requirements to establish a baseline. If your process is at level 2, the requirements are well-defined and you can collect additional information: the type of each requirement (database requirements, interface requirements, performance requirements, and so on) and the number of changes to each type. At level 3, your visibility into the process is improved, and intermediate activities are defined, with entry and exit criteria for each activity. For this level, you can collect a richer type of measurement: measuring the traceability of each requirement to the corresponding design, code, and test components, for example, and noting the effects of each change on the related components. Thus, the goal and question analysis is the same, but the metric recommendations vary with maturity. The more mature your process, the richer your measurements. In other words, the more mature your process, the more mature your measurements.

Moreover, maturity and measurement work hand in hand. As the measurements provide additional visibility, aiding you in solving project problems, you can use this approach again to expand the measurement set and address other pressing problems. Thus, your measurement program can start small and grow carefully, driven by process and project needs.

Using goals to suggest a metrics program has been successful in many organizations and is well-documented in the literature (Grady and Caswell, 1987; Rifkin, 1991; Pfleeger, 1993). The GQM paradigm, in concert with process maturity, has been used as the basis for several tools that assist managers in designing measurement programs. For example, the ami project, funded by the European Community under its ESPRIT program, has a handbook and tool that suggest the use of both GQM and process maturity (Pulford *et al.*, 1995).

GQM has helped us to understand why we measure an attribute, and process maturity suggests whether we are capable of measuring it in a meaningful way. Together, they provide a context for measurement. Without such a context, measurements can be used improperly or inappropriately, giving us a false sense of comfort with our processes and products. In the next section, we look at the need for care in capturing and evaluating measures.

3.3 APPLYING THE FRAMEWORK

In Chapter 1, we introduced several diverse topics involving software measurement. In particular, we saw how the topics relate to essential software engineering practices. When divorced from any conceptual high-level view of software measurement and its objectives, many of the activities may have seemed unrelated. In this section, we revisit the topics to see how each fits into our unifying framework and to describe some of the issues that will be addressed in later chapters of this book. That is, we look at which processes, products, and resources are relevant in each case, which attributes we are measuring (and whether these are internal or external), and whether the focus is on assessment or prediction. Using the same topic headings as before, we describe these activities briefly; details will be provided in subsequent chapters.

3.3.1 Cost and effort estimation

Cost and effort estimation focuses on predicting the attributes of cost or effort for the development process. Here, the process includes activities from detailed specification through implementation. Most of the estimation techniques present a model; we examine a popular approach to see how cost and effort estimation is usually done.

EXAMPLE 3.10: Boehm's original, basic COCOMO model (described in more detail in Chapter 12) asserts that the effort required to develop a software system (measured by E in person months) is related to size (measured by S in thousands of delivered source statements) by the equation

$$E = aS^b$$

where a and b are parameters determined by the type of software system to be developed.

The model is intended for use during requirements capture, when estimates of effort are needed. Thus, to use the technique, we must determine the parameters a and b. Boehm provides three choices for these constants, dependent on the type of software system it is classified to be. We must also determine the size S of the eventual system; since the system has yet to be built, we must predict S in order to predict E.

Thus, it is more correct to view COCOMO as a prediction system rather than as a model. The model is expressed as the equation in Example 3.10, but the prediction system includes the inference procedures for calculating the model's parameters. The calculations involve a combination of calibration based on past history, assessment based on expert judgment, and the subjective rating of attributes. The underlying theory provides various means for interpreting the results based on the choice of parameters. It is ambiguous to talk of *the* COCOMO cost-estimation model, since the same data can produce results that vary according to the particular prediction procedures used. These observations apply to most cost-and-effort estimation models.

Cost models often reflect a variety of attributes, representing all of the entity types. For example, for more advanced versions of COCOMO as well as other cost-estimation approaches, the inference procedures involve numerous internal and external attributes of the requirements specification (a product), together with process and resource attributes subjectively provided by users.

Both basic COCOMO and Albrecht's function point cost-estimation model (described in detail in Chapter 12) assert that effort is an indirect measure of a single product attribute: size. In Albrecht's model, size is measured by the number of function points, derived from and measuring the specifications. Here "size" really means a more specific attribute, namely functionality. In the latest version of COCOMO, called COCOMO 2.0, size is defined in different ways, depending on when during the development process the estimate is being made (Boehm *et al.*, 1995). Early on, size is described in terms of object points that can be derived from a prototype or initial specification. As more is known about the system, function points provide the size measure. When yet more information is available, size is defined as the number of thousands of delivered source statements, an attribute of the final implemented system. Thus, to use the model for effort prediction, you must predict size at the specification phase; that is, using the model involves predicting an attribute of a future product in order to predict effort. This approach can be rather unsatisfactory, since predicting size may be just as hard as predicting effort. In addition, the number of source statements captures only a very narrow view of size, namely length. In Chapter 7, we propose several ways to address these problems.

3.3.2 Productivity measures and models

To analyze and model productivity, we must measure a resource attribute, such as personnel (either as teams or individuals) active during particular processes. The most commonly used model for productivity measurement expresses productivity as a fraction: the process output influenced by the personnel divided by personnel effort or cost during the process. In this equation, productivity is viewed as a resource attribute, captured as an indirect measure of a product-attribute measure and a process-attribute measure.

In recent years, productivity has been viewed in a different light. Inspired by Japanese notions of spoilage, where engineers measure how much effort is expended fixing things that could have been put right before delivery, some software engineers compare the cost of fault prevention with the cost of fault detection and correction.

EXAMPLE 3.11: Watts Humphrey has defined a Personal Software Process (PSP) that encourages software engineers to evaluate their individual effectiveness at producing quality code (Humphrey, 1995). As part of the PSP, Humphrey suggests that we capture time and effort information about appraisal, failure and prevention. The appraisal cost is measured as the percentage of development time spent in design and test reviews. The failure cost is the percentage of time spent in compile and test, while the prevention cost is that

time spent preventing defects before they occur (in such activities as prototyping and formal specification). The ratio of appraisal cost to failure cost then tells us the relative effort spent in early defect removal. Such notions help us to understand how to improve our productivity.

3.3.3 Data collection

Even the simplest models depend on accurate measures. Often, product measures may be extracted with a minimum of human intervention, since the products can be digitized and analyzed automatically. But automation is not usually possible for process and resource measures. Much of the work involved in data collection is therefore concerned with how to set in place rigorous procedures for gathering accurate and consistent measures of process and resource attributes.

Chillarege and his colleagues at IBM have suggested a scheme for collecting defect information that adds the rigor usually missing in such data capture. In their scheme, the defects are classified orthogonally, meaning that any defect falls in exactly one category (Chillarege *et al.*, 1992). Such consistency of capture and separation of cause allow us to analyze not only the product attributes but also process effectiveness. In Chapter 5, we shall see how quality of data collection affects the quality of data and of subsequent analysis.

3.3.4 Quality models and measures

Quality models usually involve product measurement, as their ultimate goal is predicting product quality. We saw in Chapter 1 that both cost and productivity are dependent on the quality of products output during various processes. The productivity model in Figure 1.5 explicitly considers the quality of output products; it uses an internal process attribute, defects discovered, to measure quality. The quality factors used in most quality models usually correspond to external product attributes. The criteria into which the factors are broken generally correspond to internal product or process attributes. The metrics that measure the criteria then correspond to proposed measures of the internal attributes. In each case, the terms used are general and are usually not well-defined and precise.

Gilb has suggested a different approach to quality. He breaks high-level quality attributes into lower-level attributes; his work is described in Chapter 9 (Gilb, 1988).

3.3.5 Reliability models

Reliability is a high-level, external product attribute that appears in all quality models. The accepted view of reliability is the likelihood of successful operation during a given period of time. Thus, reliability is a relevant attribute only for executable code. Many people view reliability as the single most important quality attribute to consider, so research on measuring and predicting reliability is sometimes considered a separate sub-discipline of software measurement.

In Chapter 9, we consider proposals to measure reliability using internal process measures, such as number of faults found during formal testing or mean time to failure during testing. It is in that chapter that we describe the pitfalls of such an approach.

As we have seen, many developers observe failures (and subsequent fixes) during operation or testing, and then use this information to determine the probability of the next failure (which is a random variable). Measures of reliability are defined in terms of the subsequent distributions (and are thus defined in terms of process attributes). For example, we can consider the distribution's mean or median, or the rate of occurrence of failures. But this approach presents a prediction problem: we are trying to say something about future reliability on the basis of past observations.

EXAMPLE 3.12: The *Jelinski–Moranda* model for software reliability assumes an exponential probability distribution for the time of the ith failure. The mean of this distribution (and hence the mean time to ith failure, or MTTF_i) is given by

$$\text{MTTF}_i = \frac{\alpha}{N-i+1}$$

where N is the number of faults assumed to be initially contained in the program, and $1/\alpha$ represents the size of a fault, that is, the rate at which it causes a failure. (Faults are assumed to be removed on observation of a failure, and each time the rate of occurrence of failures is reduced by $\phi = 1/\alpha$.) The unknown parameters of the model, N and α, must somehow be estimated; this estimation can be done using, for example, Maximum Likelihood Estimation after observing a number of failure times.

In this example, we cannot discuss only the Jelinski–Moranda model; we must specify a prediction system as well. In this case, the prediction system supplements the model with a statistical inference procedure for determining the model parameters, and a prediction procedure for combining the model and the parameter estimates to make statements about future reliability.

3.3.6 Performance evaluation and models

Performance evaluation is generally concerned with measuring efficiency, a product attribute. This attribute can involve time efficiency (speed of response or computation for given input) or space efficiency (memory required for given input), but it is more usually the former. Like reliability, efficiency is a product attribute that appears high in the hierarchy of most quality models. Classical performance evaluation (Ferrari, 1978; Ferrari *et al.*, 1983; Kleinrock, 1975) is concerned with measuring the external attribute of efficiency for executable products. You may be tempted to conclude that efficiency can be only an external attribute (like reliability) and that no other type of

measurement is possible. However, even when little is known about the compiler and target machine, efficiency can be predicted quite accurately by considering any representation of the source code. In particular, a high-level description of the underlying algorithm is normally enough to determine reasonably good predictors of efficiency of the implemented code; for time efficiency, the prediction is done by first determining key inputs and basic machine operations, and then calculating the number of basic operations required as a function of input size. This area of work (traditionally the staple diet of computer scientists) is called **algorithmic analysis** or **algorithmic complexity**, where complexity is synonymous with efficiency. Viewed with our framework, algorithmic efficiency is an internal attribute that can be measured and used to predict the external attribute of efficiency of executable code.

Computational complexity is closely related and also fits our software measurement framework. Here, we are interested in measuring the inherent complexity of a problem; a problem is synonymous with the requirements specification (a product in our sense), and complexity is synonymous with the efficiency of the fastest algorithm possible to solve the problem. For example, suppose our requirements specification asserts: "For an n-node network with known distances between any pair of nodes, the system must compute the shortest route through the network that visits each node."

This requirements specification seeks a general solution to the so-called traveling-salesman problem, a highly complex task because the underlying problem is known to be in a complexity class called NP-complete. We want to measure the speed of the algorithm; the notions of speed and fast algorithms will be addressed in Chapter 7. In this example, there is no known fast algorithm that will solve the problem, nor is there ever likely to be one.

3.3.7 *Structural and complexity metrics*

Many high-level quality attributes (that is, external product attributes) are notoriously difficult to measure. For reasons to be made clear in Chapter 7, we are often forced to consider measures of internal attributes of products as weaker substitutes. The Halstead measures are examples of this situation; defined on the source code, the internal measures suggest what the external measures might be. Numerous other measures have been proposed that are defined on graphical models of either the source code or design. By measuring specific internal attributes like control-flow structure, information flow and number of paths of various types, they attempt to quantify those aspects of the code that make the code difficult to understand. Unfortunately, much work in this area has been obfuscated by suggestions that such measures capture directly the external attribute of complexity. In other words, many software engineers claim that a single internal attribute representing complexity can be an accurate predictor of many external quality attributes, as well as of process attributes like cost and effort. We discuss this problem and suggest a rigorous approach to defining structural attributes in Chapter 8.

3.3.8 *Capability-maturity assessment*

As we have seen, the SEI capability-maturity model and its offshoots attempt to capture notions of process visibility: clearly an internal process attribute. But the maturity level has also been interpreted in a very different way, as a resource attribute. An SEI capability-maturity evaluation results in a profile of an organization's typical development process, as described in more detail in Chapter 14. The profile is associated with a score, placing the organization somewhere on the scale between *ad hoc* (1) and optimizing (5). This score is considered by some practitioners to be a rating of the development organization's capability, to be used in evaluating how well-prepared the developers are to build a proposed software product. That is, the maturity score is viewed as an attribute of the contractor, rather than of the contractor's process. For example, the US Air Force has proposed that contractors be rated at least level 3 on the SEI scale in order to be considered for bidding on Air Force-funded development work.

3.3.9 *Management by metrics*

Managers often use metrics to set targets for their development projects. These targets are sometimes derived from best practice, as found in a sample of existing projects.

EXAMPLE 3.13: The US Department of Defense analyzed US government and industry performance, grouping projects into low, median and best-in-class ratings. From their findings, the analysts recommended targets for Defense Department-contracted software projects, along with indicated levels of performance constituting management malpractice, shown in Table 3.5. That is, if a project demonstrates that an attribute is below the malpractice level, then something is seriously wrong with the project.

Defect-removal efficiency is a process measure, while defect density, requirements creep (that is, an increase or change from the original set of

Table 3.5: Quantitative targets for managing US defense projects (NetFocus, 1995)

Item	Target	Malpractice level
Defect removal efficiency	> 95%	< 70%
Original defect density	< 4 per function point	> 7 per function point
Slip or cost overrun in excess of risk reserve	0%	> 10%
Total requirements creep (function points or equivalent)	< 1% per month average	> 50%
Total program documentation	< 3 pages per function point	> 6 pages per function point
Staff turnover	1–3% per year	> 5% per year

requirements) and program documentation are product measures. Cost overrun and staff turnover capture resource attributes.

Notice that the measures in Example 3.13 are a mixture of measurement types. Moreover, none of the measures is external. That is, none of the measures reflects the actual product performance as experienced by the user. This situation illustrates the more general need for managers to understand likely product quality as early in the development process as possible. But, as we shall see in later chapters, unless the relationship between internal and external attributes is well-understood, such management metrics can be misleading.

3.3.10 *Evaluation of methods and tools*

Often, organizations consider investing in a new method or tool but hesitate because of uncertainty about cost, utility, or effectiveness. Sometimes the proposed tool or method is tried first on a small project, and the results are evaluated to determine if further investment and broader implementation are in order. For example, we saw in Example 3.6 that AT&T used goals and questions to decide what to measure in evaluating the effectiveness of its inspections.

EXAMPLE 3.14: The results of the AT&T inspection evaluation are summarized in Table 3.6. For the first sample project, the researchers found that 41% of the inspections were conducted at a rate faster than the recommended rate of 150 lines of code per hour. In the second project, the inspections with rates below 125 found an average of 46% more faults per KLOC than those with faster rates. This finding means either that more faults can be found when inspection rates are slower, or that finding more faults causes the inspection rate to slow. Here, the metrics used to support this analysis are primarily process metrics.

These examples show us that it takes all types of attributes and metrics – process, product and resource – to understand and evaluate software development.

Table 3.6: Code inspection statistics from AT&T (Barnard and Price, 1994)

Metric	First sample project	Second sample project
Number of inspections in sample	27	55
Total KLOC inspected	9.3	22.5
Average LOC inspected (module size)	343	409
Average preparation rate (LOC/hour)	194	121.9
Average inspection rate (LOC/hour)	172	154.8
Total faults detected (observable and non-observable) per KLOC	106	89.7
Percentage of reinspections	11	0.5

Assume P satisfies S $m(S,P) = \emptyset$

3.3.11 *A mathematician's view of metrics*

In Chapter 2, we discussed the theory of measurement, explaining that we need not use the term "metric" in our exposition. There is another, more formal, reason for using care with the term. In mathematical analysis, a *metric* has a very specific meaning: it is a rule used to describe how far apart two points are. More formally, a **metric** is a function m defined on pairs of objects x and y such that $m(x, y)$ represents the distance between x and y. Such metrics must satisfy certain properties:

- $m(x, x) = 0$ for all x: that is, the distance from a point to itself is 0;
- $m(x, y) = m(y, x)$ for all x and y: that is, the distance from x to y is the same as the distance from y to x;
- $m(x, z) < m(x, y) + m(y, z)$ for all x, y and z: that is, the distance from x to z is no larger than the distance measured by stopping through an intermediate point.

There are numerous examples where we might be interested in "mathematical" metrics in software:

EXAMPLE 3.15: Fault-tolerant techniques like N-version programming are popular for increasing the reliability of safety-critical systems. The approach involves developing N different versions of the critical software components independently. Theoretically, by having each of the N different teams solving the same problem without knowledge of what the other teams are doing, the probability that all the teams, or even of the majority, will make the same error is kept small. When the behavior of the different versions differs, a voting procedure accepts the behavior of the majority of the systems. The assumption, then, is that the correct behavior will always be chosen.

However, there may be problems in assuring genuine design independence, so we may be interested in measuring the level of diversity between two designs, algorithms or programs. We can define a metric m, where $m(P_1, P_2)$ measures the diversity between two programs P_1 and P_2. In this case, the entities being measured are products. Should we use a similar metric to measure the level of diversity between two methods applied during design, we would be measuring attributes of process entities.

EXAMPLE 3.16: We would hope that every program satisfies its specification completely, but this is rarely the case. Thus, we can view program correctness as a measure of the extent to which a program satisfies its specification, and define a metric $m(S, P)$ where the entities S (specification) and P (program) are both products.

To reconcile these mathematically precise metrics with the framework we have proposed, we can consider pairs of entities as a single entity. For example, having produced two programs satisfying the same specification, we consider the pair of programs to be a single product system, itself having a level of diversity. This approach

is consistent with a systems view of N-version programming. Where we have implemented N versions of a program, the diversity of the system may be viewed as an indirect measure of the pairwise program diversity.

3.4 SOFTWARE MEASUREMENT VALIDATION

The very large number of software measures in the literature aims to capture information about a wide range of attributes. Even when you know which entity and attribute you want to assess, there are many measures from which to choose. Finding the best measure for your purpose can be difficult, as candidates measure or predict the same attribute (such as cost, size, or complexity) in very different ways. So it is not surprising when managers are confused by measurement: they see different measures for the same thing, and sometimes the implications of one measure lead to a management decision opposite to the implications of another! One of the roots of this confusion is the lack of software measurement validation. That is, we do not always stop to ensure that the measures we use actually capture the attribute information we seek.

The formal framework presented in Chapter 2 and in this chapter leads to a formal approach for validating software measures. The validation approach depends on distinguishing measurement from prediction, as discussed in Chapter 2. That is, we must separate our concerns about two types of measuring:

- **Measures** or **measurement systems** are used to assess an existing entity by numerically characterizing one or more of its attributes.
- **Prediction systems** are used to predict some attribute of a future entity, involving a mathematical model with associated prediction procedures.

Informally, we say that a *measure* is **valid** if it accurately characterizes the attribute it claims to measure. On the other hand, a *prediction system* is **valid** if it makes accurate predictions. So not only are measures different from prediction systems, but the notion of validation is different for each. Thus, to understand why validation is important and how it should be done, we consider measures and prediction systems separately.

3.4.1 Validating prediction systems

Validating a prediction system in a given environment is the process of establishing the accuracy of the prediction system by empirical means; that is, by comparing model performance with known data in the given environment

Thus, validation of prediction systems involves experimentation and hypothesis-testing, as we shall see in Chapter 4. Rather than being a mathematical proof, validation involves confirming or refuting a hypothesis.

This type of validation is well accepted by the software engineering community. For example, researchers and practitioners use data sets to validate cost-estimation or reliability models. If you want to know whether COCOMO is valid for your type of development project, you can use data that represent that type of project and assess the accuracy of COCOMO in predicting effort and duration.

The degree of accuracy acceptable for validation depends on several things, including the person doing the assessment. We must also consider the difference between **deterministic** prediction systems (we always get the same output for a given input) and **stochastic** prediction systems (the output for a given input will vary probabilistically) with respect to a given model.

Some stochastic prediction systems are more stochastic than others. In other words, the error bounds for some systems are wider than in others. Prediction systems for software cost estimation, effort estimation, schedule estimation and reliability are very stochastic, as their margins of error are large. For example, Boehm has stated that under certain circumstances the COCOMO effort-prediction system will be accurate to within 20%; that is, the predicted effort will be within 20% of the actual effort value. An **acceptance range** for a prediction system is a statement of the maximum difference between prediction and actual value. Thus, Boehm specifies 20% as the acceptance range of COCOMO. Some project managers find this range to be too large to be useful for planning, while other project managers find 20% to be acceptable, given the other uncertainties of software development. Where no such range has been specified, you must state in advance what range is acceptable before you use a prediction system.

EXAMPLE 3.17: Sometimes the validity of a complex prediction system may not be much greater than that of a very simple one. For example, it has been shown that if the weather tomorrow in Austria is always predicted to be the same as today's weather, then the predictions are accurate 67% of the time. The use of sophisticated computer models increases this accuracy to just 70%!

In Chapter 10, we present a detailed example of how to validate software-reliability prediction systems using empirical data.

3.4.2 Validating measures

Measures used for assessment are the measures discussed in Chapter 2. We can turn to measurement theory to tell us what validation means in this context:

Validating a software measure is the process of ensuring that the measure is a proper numerical characterization of the claimed attribute by showing that the representation condition is satisfied.

EXAMPLE 3.18: We want to measure the length of a program in a valid way. Here, "program" is the entity and "length" the attribute. The measure we choose must not contradict any intuitive notions about program length.

Specifically, we need both a formal model that describes programs (to enable objectivity and repeatability) and a numerical mapping that preserves our intuitive notions of length in relations that describe the programs. For example, if we concatenate two programs P_1 and P_2, we get a program P_1; P_2 whose length is the combined lengths of P_1 and P_2. Thus, we expect any measure m of length always to satisfy the condition

$$m(P_1; P_2) = m(P_1) + m(P_2)$$

If program P_1 has a greater length than program P_2, then any measure m of length must also satisfy

$$m(P_1) > m(P_2)$$

We can measure program length by counting lines of code (in the carefully defined way we describe in Chapter 7). Since this count preserves these relationships, lines of code is a valid measure of length. We will also describe a more rigorous length measure in Chapter 7, based on a formal model of programs.

This type of validation is central to the representational theory of measurement. That is, we want to be sure that the measures we use reflect the behavior of entities in the real world. If we cannot validate the measures, then we cannot be sure that the decisions we make based on those measures will have the effects we expect. In some sense, then, we use validation to make sure that the measures are defined properly and are consistent with the entity's real-world behavior.

3.5 SOFTWARE MEASUREMENT VALIDATION IN PRACTICE

The software-engineering community has always been aware of the need for validation. As new measures are proposed, it is natural to ask whether the measure captures the attribute it claims to describe. But in the past, validation has been a relaxed process, sometimes relying on the credibility of the proposer, rather than on rigorous validation procedures. That is, if someone of stature says that measure X is a good measure of complexity, then practitioners and researchers begin to use X without question. Or software engineers adopt X after reading informal arguments about why X is probably a good measure. This situation can be remedied only by reminding the software community of the need for rigorous validation.

An additional change in attitude is needed, though. Many researchers and practitioners assume that validation of a measure (in the measurement-theory sense) is not sufficient. They expect the validation to demonstrate that the measure is itself part of a valid prediction system.

EXAMPLE 3.19: Many people use lines of code as a measure of software size. The measure has many uses: it is a general size measure in its own right, it can be used to normalize other measures (like number of faults) to enable comparison, and it is input to many compound measures such as productivity

(measured as effort divided by lines of code produced). But some software engineers claim that lines of code is not a valid software measure because it is not a good predictor of reliability or maintenance effort.

Thus, a measure must be viewed in the context in which it will be used. Validation must take into account the measurement's purpose; a measure may be valid for some uses but not for others.

3.5.1 *A more stringent requirement for validation*

As we have seen with lines of code, it is possible for a measure to serve both purposes: to measure an attribute in its own right, and to be a valuable input to a prediction system. But a measure can be one or the other without being both. We should take care not to reject as invalid a reasonable measure just because it is not a predictor. For example, there are many internal product attributes (such as size, structuredness, and modularity) that are useful and practical measures, whether or not they are also part of a prediction system. If a measure for assessment is valid, then we say that it is **valid in the narrow sense** or is **internally valid**.

However, many process attributes (such as cost), external product attributes (such as reliability) and resource attributes (such as productivity) play a dual role. We say that a measure is **valid in the wide sense** if it is both

1. internally valid; and
2. a component of a valid prediction system.

Suppose we wish to show that a particular measure is valid in the wide sense. After stating a hypothesis that proposes a specific relationship between our measure and some useful attribute, we must conduct an experiment to test the hypothesis, as described in Chapter 4. Unfortunately, in practice the experimental approach is not often taken. Instead, the measure is claimed to be valid in the wide sense by demonstrating a statistical correlation with another measure. For example, a measure of modularity might be claimed to be valid or invalid on the basis of a comparison with known development costs. This validation is claimed even though there is no demonstrated explicit relationship between modularity and development costs! In other words, our measure cannot be claimed to be a valid measure of modularity simply because of the correlation with development costs. We require a model of the relationship between modularity and development costs, showing explicitly all the factors interconnecting each. Only then can we judge whether measuring modularity is the same as measuring development cost.

This type of mistake is common in software engineering, as well as in other disciplines. Engineers forget that statistical correlation does not imply cause and effect. For example, suppose we propose to measure obesity using inches, normally a length measure. We measure the height and weight of each of a large sample of people, and we show that height correlates strongly with weight. Based on that result,

we cannot say that height is a valid measure of obesity, since height is only one of several factors that must be taken into account; heredity and body fat percentages play a role, for instance. In other words, just because height correlates with weight does not mean that height is a valid measure of weight or obesity. Likewise, there may be a statistical correlation between modularity and development costs, but that does not mean that modularity is the key factor determining development cost.

It is sometimes possible to show that a measure is valid in the wide sense without a stated hypothesis and planned experiment. Available data can be examined using techniques such as regression analysis; the measure may be consistently related (as shown by the regression formula) to another variable. However, this approach is subject to the same problems as the correlation approach. It requires a model showing how the various factors interrelate. Given our poor understanding of software, this type of validation appears fraught with difficulty. Nevertheless, many software engineers continue to use this technique as their primary approach to validation.

The many researchers who have taken this approach have not been alone in making mistakes. It is tempting to measure what is available and easy to measure, rather than to build models and capture complex relationships. Indeed, speaking generally about measurement validation, Krantz asserts:

> A recurrent temptation when we need to measure an attribute of interest is to try to avoid the difficult theoretical and empirical issues posed by fundamental measurement by substituting some easily measured physical quantity that is believed to be strongly correlated with the attribute in question: hours of deprivation in lieu of hunger; skin resistance in lieu of anxiety; milliamperes of current in lieu of aversiveness, etc. Doubtless this is a sensible thing to do when no deep analysis is available, and in all likelihood some such indirect measures will one day serve very effectively when the basic attributes are well understood, but to treat them now as objective definitions of unanalyzed concepts is a form of misplaced operationalism.
>
> Little seems possible in the way of careful analysis of an attribute until means are devised to say which of two objects or events exhibits more of the attribute. Once we are able to order the objects in an acceptable way, we need to examine them for additional structure. Then begins the search for qualitative laws satisfied by the ordering and the additional structure.
>
> (Krantz *et al.*, 1971)

Neither we nor Krantz claim that measurement and prediction are completely separate issues. On the contrary, we fully support the observation of Kyburg:

> If you have no viable theory into which *X* enters, you have very little motivation to generate a measure of *X*.
>
> (Kyburg, 1984)

However, we are convinced that our initial obligation in proposing measures is to show that they are valid in the narrow sense. Good predictive theories follow only when we have rigorous measures of specific, well-understood attributes.

3.5.2 *Validation and imprecise definition*

In Example 3.18, we noted that lines of code (LOC) is a valid measure of program length. However, LOC has not been shown convincingly to be a valid measure of complexity. Nor has it been shown to be part of an accurate prediction system for complexity. The fault lies not with the LOC measure but with the imprecise definition of complexity. Although complexity is generally described as an attribute that can affect reliability, maintainability, cost, and more, the fuzziness surrounding its definition presents a problem in complexity research. Chapter 8 explores whether complexity can ever be defined and measured precisely.

The problems with complexity do not prevent LOC from being a useful measure for purposes other than length. For example, if there is a stochastic association between a large number of lines of code and a large number of unit-testing errors, this relationship can be used in choosing a testing strategy and in reducing risks.

Many studies have demonstrated a significant correlation between LOC and the cyclomatic number. The researchers usually suggest that this correlation proves that the cyclomatic number increases with size; that is, larger code is more complex code. However, careful interpretation of the measures and their association reveals only that the number of decisions increases with code length, a far less profound conclusion. Chapter 8 contains more detailed discussion of validation for both the McCabe and Halstead measures.

3.5.3 *How not to validate*

New measures are sometimes validated by showing that they correlate with well-known, existing measures. Li and Cheung present an extensive example of this sort (Li and Cheung, 1987). This approach is appealing because:

- It is generally assumed that the well-known existing measure is valid, so that a good correlation means that the new measure must also be valid.
- This type of validation is straightforward, since the well-known measure is often easy to compute; automated tools are often available for computation.
- If the management is familiar with the existing measures, then the proposed measures have more credibility.

Well-known measures used in this way include Halstead's suite of software science measures, McCabe's cyclomatic number, and lines of code. However, although these may be valid measures of very specific attributes (such as number of decisions for cyclomatic number, and source-code program length for lines of code), they have not been shown to be valid measures of attributes such as cognitive complexity, correctness, or maintainability. Thus, we must take great care in validating by comparison with existing measures. We must verify that the qualities associated with the existing measures have been demonstrated in the past.

There is much empirical evidence to suggest that measures such as Halstead's suite and McCabe's cyclomatic number are associated with development and

maintenance effort and faults. But such correlations do not imply that the measures are good predictors of these attributes. The many claims made that Halstead and McCabe measures have been validated are annulled by studies showing that the correlations with process data are no better than using a simple measure of size, such as lines of code (Hamer and Frewin, 1982).

There is a more compelling scientific and statistical reason why we must be wary of the correlate-against-existing-measures approach. Unstructured correlation studies run the risk of identifying spurious associations. Using the 0.05 significance level, we can expect a significant but spurious correlation 1 in 20 times by chance. Thus, if you have 5 independent variables and look at the 10 possible pairwise correlations, there is a 0.5 (1 in 2) chance of getting a spurious correlation. In situations like this, if we have no hypothesis about the reason for a relationship, we can have no real confidence that the relationship is not spurious. Courtney and Gustafson examine this problem in detail (Courtney and Gustafson, 1993).

3.5.4 *Choosing appropriate prediction systems*

To help us formulate hypotheses necessary for validating the predictive capabilities of measures, we divide prediction systems into the following classes:

- **Class 1:** using internal attribute measures of early life-cycle products to predict measures of internal attributes of later life-cycle products. For example, measures of size, modularity, and reuse of a specification are used to predict size and structuredness of the final code.
- **Class 2:** using early life-cycle process attribute measures and resource-attribute measures to predict measures of attributes of later life-cycle processes and resources. For example, the number of faults found during formal design review is used to predict cost of implementation.
- **Class 3:** using internal product-attribute measures to predict process attributes. For example, measures of structuredness are used to predict time to perform some maintenance task, or number of faults found during unit testing.
- **Class 4:** using process measures to predict later process measures. For example, measures of failures during one operational period are used to predict likely failure occurrences in a subsequent operational period. In examples like this, where an external product attribute (reliability) is effectively defined in terms of process attributes (operational failures), we may also think of this class of prediction systems as using process measures to predict later external product measures.

Notice that there is no class of prediction system using internal structural attributes to predict external and process attributes. We doubt whether there will ever be such a class of generally accepted prediction system. In theory, it is always possible to construct products that appear to exhibit identical external attributes but which in fact vary greatly internally.

However, we usually assume that certain internal attributes that result from modern software-engineering techniques (such as modularization, low coupling, control and data structuredness, information hiding and reuse) will generally lead to products exhibiting a high degree of desirable external attributes like reliability and maintainability. Thus, programs and modules that have poor values of desirable internal attributes (such as large, unstructured modules) are likely (but not certain) to have more faults and take longer to produce and maintain. We return to this issue in Chapter 8.

3.6 SUMMARY

This chapter presents a framework to help us to discuss what to measure and how to use the measures appropriately. The key points to remember are the following:

- All entities of interest in software can be classified as either processes, products, or resources. Anything we may wish to measure is an identifiable attribute of these.
- Attributes are either internal or external. Although external attributes (such as reliability of products, stability of processes, or productivity of resources) tend to be the ones we are most interested in measuring, we cannot do so directly. We are generally forced to use indirect measures of internal attributes.
- The Goal–Question–Metric paradigm is a useful approach for deciding what to measure. Since managers and practitioners have little time to measure everything, the GQM approach allows them to choose those measures that relate to the most important goals or the most pressing problems.
- The GQM approach creates a hierarchy of goals to be addressed (perhaps decomposed into subgoals), questions that should be answered in order to know if the goals have been met, and measurements that must be made in order to answer the questions. The technique considers the perspective of the people needing the information and the context in which the measurements will be used.
- Process maturity must also be considered when deciding what to measure. If an entity is not visible in the development process, then it cannot be measured. Five levels of maturity, ranging from *ad hoc* to optimizing, can be associated with the types of measurements that can be made.
- GQM and process maturity must work hand-in-hand. By using GQM to decide what to measure and then assessing the visibility of the entity, software engineers can measure an increasingly richer set of attributes.
- Many misunderstandings and misapplications of software measurement would be avoided if people thought carefully about the above framework. Moreover, this framework highlights the relationships among apparently diverse software measurement activities.

- Measurement is concerned not only with assessment (of an attribute of some entity that already exists) but also with prediction (of an attribute of some future entity). We want to be able to predict attributes like the cost and effort of processes, as well as the reliability and maintainability of products. Effective prediction requires a prediction system: a model supplemented with a set of prediction procedures for determining the model parameters and applying the results.
- The validation approach described in this chapter guides you in determining precisely which entities and attributes have to be considered for measurement. In this sense, it supports a goal-oriented approach.
- Software measures and prediction systems will be neither widely used nor respected without a proper demonstration of their validity. However, commonly accepted ideas and approaches to validating software measures bear little relation to the rigorous requirements for measurement validation in other disciplines. In particular, formal validation requirements must first be addressed before we can tackle informal notions such as usefulness and practicality.
- The formal requirement for validating a measure involves demonstrating that it characterizes the stated attribute in the sense of measurement theory.
- To validate a prediction system formally, you must first decide how stochastic it is (that is, determine an acceptable error range), and then compare performance of the prediction system with known data points. This comparison will involve experiments, such as those described in Chapter 4.
- Software measures that characterize specific attributes do not have to be shown to be part of a valid prediction system in order to be valid measures. A claim that a measure is valid because it is a good predictor of some interesting attribute can be justified only by formulating a hypothesis about the relationship and then testing the hypothesis.

3.7 EXERCISES

1 For each entity listed in Table 3.1, find at least one way in which environmental considerations will influence the relevant external attributes.

2 What different product, process or resource attributes might the following measure?:

i. the number of faults found in program P using a set of test data created specifically for P;
ii. the number of faults found in program P using a standard in-house set of test data;
iii. the number of faults found in program P by programmer A during one hour.

3 Check the dictionary definitions of the following measures as a basis for validation: (i) candlepower, (ii) decibel, (iii) horsepower, (iv) lightyear, (v) span.

4 Look at the four classes of prediction systems on page 110. To which of the four classes do the following prediction systems belong?: (i) the COCOMO model (Example 3.10), (ii) the Jelinski–Moranda reliability model (Example 3.12), (iii) stochastic systems, (iv) coupling as a predictor of program errors.

5 Explain briefly the idea behind the GQM paradigm. Is it always the right approach for suggesting what to measure? Suppose you are managing a software-development project for which reliability is a major concern. A continual stream of anomalies is discovered in the software during the testing phase, and you suspect that the software will not be of sufficient quality by the shipping deadline. Construct a GQM tree that helps you to make an informed decision about when to ship the software.

6 Suppose that a software producer considers software quality to consist of a number of attributes, including reliability, maintainability, and usability. Construct a simple GQM tree corresponding to the producer's goal of improving the quality of the software.

7 Your department manager asks you to help improve the maintainability of the software that your department develops. Construct a GQM tree to identify an appropriate set of measures to assist you in this task.

8 Suppose your development team has as its goal "improve effectiveness of testing". Use the GQM approach to suggest several relevant questions and measures that will enable you to determine if you have met your goal.

9 Our confidence in a prediction system depends in part on how well we feel we understand the attributes involved. After how many successful predictions would you consider the following validated?

i. a mathematical model to predict the return time of Halley's comet to the nearest hour;
ii. a timetable for a new air service between London and Paris;
iii. a system that predicts the outcome of the spinning of a roulette wheel;
iv. a chart showing how long a bricklayer will take to erect a wall of a given area.

3.8 FURTHER READING

The COCOMO model is interesting to study, because it is a well-documented example of measurement that has evolved as goals and technology have changed. The original COCOMO approach is described in:

Boehm, B.W., *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, NJ, 1981.

An overview of the revision to COCOMO is in:

Boehm, B.W., Clar, B., Horowitz, E., Westland, C., Madachy, R. and Selby, R,."Cost models for future software life cycle processes: COCOMO 2.0," *Annals of Software Engineering* 1(1), pp. 1–30.

The first paper to mention GQM is:

Basili, V.R., and Weiss, D.M., "A method for collecting valid software engineering data," *IEEE Transactions on Software Engineering*, 10(6), pp. 728–38, 1984.

More detail and examples are provided in:

Basili, V.R. and Rombach, H.D., "The TAME project: Towards improvement-oriented software environments," *IEEE Transactions on Software Engineering*, 14(6), pp. 758–73, 1988.

Despite its widespread appeal, GQM is not without its critics. These texts argue strongly against GQM on grounds of practicality.

Bache, R. and Neil, M., "Introducing metrics into industry: a perspective on GQM," in *Software Quality Assurance and Metrics: A Worldwide Perspective* (eds: N.E. Fenton, R.W. Whitty, Y. Iizuka), International Thomson Press, pp. 59–68, 1995.
Hetzel, W.C., *Making Software Measurement Work: Building an Effective Software Measurement Program*, QED Publishing Group, Wellesley, MA, 1993.

Researchers at IBM have produced a good analysis of fault categorization. The structure and its application are described in:

Chillarege, R., Bhandari, I.S., Chaar, J.K., Halliday, M.J., Moebus, D.S., Ray B.K., and Wong, M.-Y., "Orthogonal defect classification: A concept for in-process measurements," *IEEE Transactions on Software Engineering*, 18(11), pp. 943–56, 1992.

For a good example of industry needs, look at the NetFocus newsletter. This newsletter describes the derivation of a metrics "dashboard" that depicts a small number of key measures. The dashboard is to be used by project managers to "drive" a software-development project, telling the manager when the products are ready for release to the customer.

NetFocus: Software Program Managers Network, number 207, Department of the Navy (US), January 1995.

Correlation is often used to "validate" software measures. The paper by Courtney and Gustafson shows that use of correlation analysis on many metrics will inevitably throw up spurious "significant" results.

Courtney, R.E. and Gustafson D.A., "Shotgun correlations in software measures," *Software Engineering Journal*, 8(1), pp. 5–13, 1993.

Software Engineering 3S03

317031

CUSTOM PUBLISHING SOLUTIONS
Custom Courseware

This material has been produced in partnership with
Media Production Services and Titles Bookstore,
McMaster University.

*** COURSEWARE IS NON-RETURNABLE**

9 780666 317032

media

Titles McMaster University BOOKSTORE

CAW TCA Local 555