Preeti Patel

Software Testing and Quality Assurance

1. Basics of Software Testing

➔ Humans, Errors & Testing

Errors are a part of our daily life. Humans make errors in their thoughts, in their actions, and in the products that might result from their actions. Errors occur almost everywhere. For example, humans make errors in speech, in medical prescription, in surgery, in driving, in observation, in sports, and certainly in love and software development. Table 1.1 provides examples of human errors. The consequences of human errors vary significantly. An error might be insignificant in that it leads to a gentle friendly smile, such as when a slip of the tongue occurs. Or, an error may lead to a catastrophe, such as when an operator fails to recognize that a relief valve on the pressurize was stuck open and this resulted in a disastrous radiation leak.

Table 1.1 Examples of errors in various fields of human endeavor.

Area	Error
Hearing	Spoken: He has a garage for repairing *foreign* cars. Heard: He has a garage for repairing *falling* cars.
Medicine	Incorrect antibiotic prescribed.
Music performance	Incorrect note played.
Numerical analysis	Incorrect algorithm for matrix inversion.
Observation	Operator fails to recognize that a relief valve is stuck open.
Software	Operator used: ≠, correct operator: >. Identifier used: new_line, correct identifier: next_line. Expression used: $a \wedge (b \vee c)$, correct expression: $(a \wedge b) \vee c$. Data conversion from 64-bit floating point to 16-bit integer not protected (resulting in a software exception).
Speech	Spoken: *waple malnut*, intent: *maple walnut*. Spoken: *We need a new refrigerator*, intent: *We need a new washing machine*.
Sports	Incorrect call by the referee in a tennis match.
Writing	Written: What kind of *pans* did you use? Intent: What kind of *pants* did you use?

Errors are a part of our daily life.

To determine whether there are any errors in our thought, actions, and the products generated, we resort to the process of *testing.* The primary goal of testing is to determine if the thoughts, actions, and products are as desired, that is, they conform to the requirements. Testing of thoughts is usually designed to determine if a concept or method has been understood satisfactorily. Testing of actions is designed to check if a skill that results in the actions has been acquired satisfactorily. Testing of a product is designed to check if the product

behaves as desired. Note that both syntax and semantic errors arise during programming. Given that most modern compilers are able to detect syntactic errors, testing focuses on semantic errors, also known as faults that cause the program under test to behave incorrectly.

The process of testing offers an opportunity to discover any errors in the product under test.

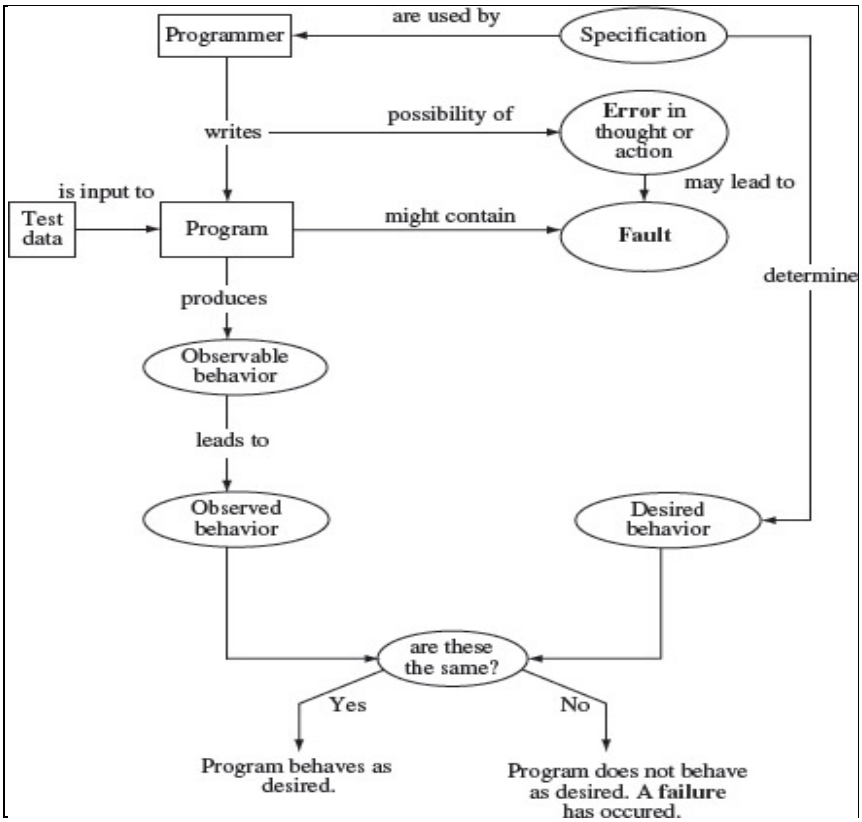
Example 1.1 An instructor administers a test to determine how well the students have understood what the instructor wanted to convey. A tennis coach administers a test to determine how well the understudy makes a serve. A software developer tests the program developed to determine if it behaves as desired. In each of these three cases there is an attempt by a tester to determine if the human thoughts, actions, and products behave as desired. Behavior that deviates from the desirable is possibly due to an error.

Example 1.2 "Deviation from the expected" may *not* be due to an error for one or more reasons. Suppose that a tester wants to test a program to sort a sequence of integers. The program can sort an input sequence in both descending and ascending orders depending on the request made. Now suppose that the tester wants to check if the program sorts an input sequence in *ascending* order. To do so, she types in an input sequence and a request to sort the sequence in *descending* order. Suppose that the program is correct and produces an output that is the input sequence in descending order.

Upon examination of the output from the program, the tester hypothesizes that the sorting program is incorrect. This is a situation where the tester made a mistake (an error) that led to her incorrect interpretation (perception) of the behavior of the program (the product).

1.1.1 Errors, faults, and failures

Figure 1.1 Errors, faults, and failures in the process of programming and testing.



There is no widely accepted and precise definition of the term "error." Figure 1.1 illustrates one class of meanings for the terms error, fault, and failure. A programmer writes a program. An *error* occurs in the process of writing a program. A *fault* is the manifestation of one or more errors. A failure occurs when a faulty piece of code is executed leading to an incorrect state that propagates to the program's output. The programmer might misinterpret the requirements and consequently write incorrect code. Upon execution, the program might display behavior that does not match with the expected behavior implying thereby that a *failure* has occurred. A fault in the program is also commonly referred to as a *bug* or a *defect.* The terms error and bug are by far the most common ways of referring to something "wrong" in the program text that might lead to a failure. In this text, we often use the terms "error" and "fault" as synonyms. Faults are sometimes referred to as *defects.*

In Figure 1.1, notice the separation of "observable" from "observed" behavior. This separation is important because it is the observed behavior that might lead one to conclude that a program has failed. Certainly, as explained earlier, this conclusion might be incorrect due to one or more reasons.

1.1.2 Test automation

Testing of complex systems embedded and otherwise, can be a human intensive task. Often one need to execute thousands of tests to ensure that, for example, a change made to a component of an application does not cause a previously correct code to malfunction. Execution of many tests can be tiring as well error prone. Hence, there is a tremendous need for automating testing tasks.

Test automation aids in reliable and faster completion of routine tasks. However, not all tasks involved in testing are prone to automation.

Most software development organizations automate test related tasks such as regression testing, GUI testing, and I/O device driver testing. Unfortunately the process of test automation cannot be generalized. For example, automating regression tests for an embedded device such as a pacemaker is quite different from that for an I/O device driver that connects to the USB port of a PC. Such lack of generalization often leads to specialized test automation tools developed in-house.

Nevertheless, there do exist general purpose tools for test automation. While such tools might not be applicable in all test environments, they are useful in many. Examples of such tools include Eggplant, Marathon, and Pounder for GUI testing; eLoadExpert, DBMonster, JMeter, Dieseltest, WAPT, LoadRunner, and Grinder for performance or load testing; and Echelon, TestTube, WinRunner, and XTest for regression testing. Despite the existence of a large number and a variety of test automation tools, large software-development organizations develop their own test automation tools primarily due to the unique nature of their test requirements.AETG is an automated test generator that can be used in a variety of applications. It uses combinatorial design techniques, which we will discuss in Chapter 6. Random testing is often used for the estimation of reliability of products with respect to specific events. For example, one might test an application using randomly generated tests to determine how frequently does it crash or hang. DART is a tool for automatically extracting an interface of a program and generating random tests. While such tools are useful in some environments, they are dependent on the programming language used and the nature of the application interface. Therefore, many organizations develop their own tools for random testing.

1.1.3 Developer and tester as two roles

In the context of software engineering, a developer is one who writes code and a tester is one who tests code. We prefer to treat developer and tester as two distinct but complementary roles. Thus, the same individual could be a developer and a tester. It is hard to imagine an individual who assumes the role of a developer but never that of a tester, and vice versa. In fact, it is safe to assume that a developer assumes two roles, that of a "developer" and

of a "tester," though at different times. Similarly, a tester also assumes the same two roles but at different times.

A developer is also a tester and vice-versa.

Certainly, within a software development organization, the primary role of an individual might be to test and hence this individual assumes the role of a "tester." Similarly, the primarily role of an individual who designs applications and writes code is that of a "developer."
A reference to "tester" in this book refers to the role someone assumes when testing a program. Such an individual could be a developer testing a class she has coded, or a tester who is testing a fully-integrated set of components. A "programmer" in this book refers to an individual who engages in software development and often assumes the role of a tester, at least temporarily. This also implies that the contents of this book are valuable not only to those whose primary role is that of a tester, but also to those who work as developers.

1.2 Software Quality

We all want high-quality software. There exist several definitions of software quality. Also, one quality attribute might be more important to a user than another. In any case, software quality is a multidimensional quantity and is measurable. So, let us look at what defines the quality of software.

1.2.1 Quality attributes

There exist several measures of software quality. These can be divided into static and dynamic quality attributes. Static quality attributes refer to the actual code and related documentation. Dynamic quality attributes relate to the behavior of the application while in use. Static quality attributes include structured, maintainable, testable code as well as the availability of correct and complete documentation. You might have come across complaints such as "Product X is excellent, I like the features it offers, but its user manual stinks!" In this case, the user manual brings down the overall product quality. If you are a maintenance engineer and have been assigned the task of doing corrective maintenance on an application code, you will most likely need to understand portions of the code before you make any changes to it. This is where attributes such as code documentation, understandability, and structure come into play. A poorly-documented piece of code will be harder to understand and hence difficult to modify. Further, poorly-structured code might be harder to modify and difficult to test.

Dynamic quality attributes include software reliability, correctness, completeness, consistency, usability, and performance. Reliability refers to the probability of failure-free operation and is considered in the following section. Correctness refers to the correct operation of an application and is always with reference to some artifact. For a tester, correctness is with respect to the requirements; for a user, it is often with respect to a user manual.

Dynamic quality attributes are generally determined through multiple executions of a program. Correctness is one such attribute though one can rarely determine the correctness of a software application via testing.

Completeness refers to the availability of all features listed in the requirements, or in the user manual. Incomplete software is one that does not fully implement all features required. Of course, one usually encounters additional functionality with each new version of an application. This does not mean that a given version is incomplete because its next version has few new features. Completeness is defined with respect to a set of features that might itself be a subset of a larger set of features that are to be implemented in some future version of the application. Of course, one can easily argue that every piece of software that is correct is also complete with respect to some feature set.

Completeness refers to the availability in software of features planned and their correct implementation. Given the near impossibility of exhaustive testing, completeness is often a subjective measure.

Consistency refers to adherence to a common set of conventions and assumptions. For example, all buttons in the user interface might follow a common color coding convention. An example of inconsistency would be when a database application displays the date of birth of a person in the database; however, the date of birth is displayed in different formats, without any regard for the user's preferences, depending on which feature of the database is used.

Usability refers to the ease with which an application can be used. This is an area in itself and there exist techniques for usability testing. Psychology plays an important role in the design of techniques for usability testing. Usability testing also refers to the testing of a product by its potential users. The development organization invites a selected set of potential users and asks them to test the product. Users in turn test for ease of use, functionality as expected, performance, safety, and security. Users thus serve as an important source of tests that developers or testers within the organization might not have conceived. Usability testing is sometimes referred to as user-centric testing.

Performance refers to the time the application takes to perform a requested task. Performance is considered as a non-functional requirement. It is specified in terms such as "This task must be performed at the rate of X units of activity in one second on a machine running at speed Y, having Z gigabytes of memory." For example, the performance requirement for a compiler might be stated in terms of the minimum average time to compile of a set of numerical applications.

1.2.2 Reliability

People want software that functions correctly every time it is used. However, this happens rarely, if ever. Most software's that are used today contain faults that cause them to fail on some combination of inputs. Thus, the notion of total correctness of a program is an ideal and applies mostly to academic and textbook programs.

Correctness and reliability are two dynamic attributes of software. Reliability can be considered as a statistical measure of correctness.

Given that most software applications are defective, one would like to know how often a given piece of software might fail. This question can be answered, often with dubious accuracy, with the help of software reliability, hereafter referred to as reliability. There are several definitions of software reliability, a few are examined below.

ANSI/IEEE STD 729-1983: RELIABILITY

Software reliability is the probability of failure-free operation of software over a given time interval and under given conditions.

Reliability

Software reliability is the probability of failure-free operation of software in its intended environment.

1.3 Requirements, Behavior, and Correctness

Products, software in particular, are designed in response to requirements. Requirements specify the functions that a product is expected to perform. Once the product is ready, it is the requirements that determine the expected behavior. Of course, during the development of the product, the requirements might have changed from what was stated originally. Regardless of any change, the expected behavior of the product is determined by the tester's understanding of the requirements during testing.

Example 1.3 Here are the two requirements, each of which leads to a different program.

Requirement 1:	It is required to write a program that inputs two integers and outputs the Maximum of these.
Requirement 2:	It is required to write a program that inputs a sequence of integers and Outputs the sorted version of this sequence.

Suppose that program max is developed to satisfy Requirement 1 above. The expected output of maxwhen the input integers are 13 and 19, can be easily determined to be 19. Now

suppose that the tester wants to know if the two integers are to be input to the program on one line followed by a carriage return, or on two separate lines with a carriage return typed in after each number. The requirement as stated above fails to provide an answer to this question. This example illustrates the incompleteness of Requirement 1.

The second requirement in the above example is ambiguous. It is not clear from this requirement whether the input sequence is to be sorted in ascending or descending order. The behavior of the sortprogram, written to satisfy this requirement, will depend on the decision taken by the programmer while writing sort.

Testers are often faced with incomplete and/or ambiguous requirements. In such situations, a tester may resort to a variety of ways to determine what behavior to expect from the program under test. For example, forth above program max, one way to determine how the input should be typed in is to actually examine the program text. Another way is to ask the developer of max as to what decision was taken regarding the sequence in which the inputs are to be typed in. Yet another method is to execute max on different input sequences and determine what is acceptable to max.

Regardless of the nature of the requirements, testing requires the determination of the *expected* behavior of the program under test. The *observed* behavior of the program is compared with the expected behavior to determine if the program functions as desired.

1.3.1 Input domain

A program is considered correct if it behaves as desired on all possible test inputs. Usually, the set of all possible inputs is too large for the program to be executed on each input. For example, suppose that the max program above is to be tested on a computer in which the integers range from –32,768 to 32,767. To test max on all possible integers would require it to be executed on all pairs of integers in this range. This will require a total of 2^{32} executions of max. It will take approximately 4.3 seconds to complete all executions assuming that testing is done on a computer that will take 1 nanosecond (=10^{-9} seconds), to input a pair of integers, execute max, and check if the output is correct. Testing a program on all possible inputs is known as *exhaustive testing.*

According to one view, the input domain of a program consists of all possible inputs as derived from the program specification. According to another view, it is the set of all possible inputs that a program could be subjected, i.e., legal and illegal inputs.

A tester often needs to determine what constitutes “all possible inputs.” The first step in determining all possible inputs is to examine the requirements. If the requirements are complete and unambiguous, it should be possible to determine the set of all possible inputs. A definition is in order before we provide an example to illustrate how to determine the set of all program inputs.

Input domain
The set of all possible inputs to a program P is known as the *input domain,* or *input space,* of P.

Example 1.4 Using Requirement 1 from Example 1.3, we find the input domain of max to be the set of all pairs of integers where each element in the pair is in the range –32,768 till 32,767.

Example 1.5 Using Requirement 2 from Example 1.3, it is not possible to find the input domain for the sort program. Let us therefore assume that the requirement was modified to be the following:

Modified Requirement 2:	It is required to write a program that inputs a sequence of integers and outputs the integers in this sequence sorted in either ascending or descending order. The order of the output sequence is determined by an input request character that should be “A” when an ascending sequence is desired, and “D” otherwise. While providing input to the program, the request character is entered first followed by the sequence of integers to be sorted; the sequence is terminated with a period.

Based on the above modified requirement, the input domain for sort is a set of pairs. The first element of the pair is a character. The second element of the pair is a sequence of zero or more integers ending with a period. For example, following are three elements in the input domain of sort:

< A -3 15 12 55.>
< D 23 78.>
< A .>

The first element contains a sequence of four integers to be sorted in ascending order, the second element has a sequence to be sorted in descending order, and the third element has an empty sequence to be sorted in ascending order. We are now ready to give the definition of program correctness.

Correctness
A program is considered correct if it behaves as expected on each element of its input domain.

1.3.2 Specifying program behavior

There are several ways to define and specify program behavior. The simplest way is to specify the behavior in a natural language such as English. However, this is more likely subject to multiple interpretations than a more formally specified behavior. Here, we explain how the notion of program "state" can be used to define program behavior and how the "state transition diagram," or simply "state diagram," can be used to specify program behavior.

A collection of the current values of program variables and the location of control is considered as a state vector for that program.

The "state" of a program is the set of current values of all its variables and an indication of which statement in the program is to be executed next. One way to encode the state is by collecting the current values of program variables into a vector known as the "state vector." An indication of where the control of execution is at any instant of time can be given by using an identifier associated with the next program statement. In the case of programs in assembly language, the location of control can be specified more precisely by giving the value of the program counter.

Each variable in the program corresponds to one element of this vector. Obviously, for a large program, such as the Unix operating system, the state vector might have thousands of elements. Execution of program statements causes the program to move from one state to the next. A sequence of program states is termed as program behavior.

Example 1.6 Consider a program that inputs two integers into variables *X* and *Y,* compares these values, sets the value of *Z* to the larger of the two, displays the value of *Z* on the screen, and exits. Program P1.1 shows the program skeleton. The state vector for this program consists of four elements. The first element is the statement identifier where the execution control is currently at. The next three elements are, respectively, the values of the three variables *X, Y,* and *Z*.

Program P1.1

```
1   integer X, Y, Z;
2   input (X, Y);
3   if (X < Y)
4     {Z=Y;}
5   else
6     {Z=X;}
7   endif
8   output (Z);
9   end
```

The letter u as an element of the state vector stands for an "undefined" value. The notation $s_i \rightarrow s_j$is an abbreviation for "The program moves from state s_i to s_j." The movement

from s_i to s_j is caused by the execution of the statement whose identifier is listed as the first element of state s_i. A possible sequence of states that the max program may go through is given below.

$$[2\ u\ u\ u] \rightarrow [3\ 3\ 15\ u] \rightarrow [4\ 3\ 15\ 15] \rightarrow [5\ 3\ 15\ 15]$$

$$\rightarrow [8\ 3\ 15\ 15] \rightarrow [9\ 3\ 15\ 15]$$

Upon the start of its execution, a program is in an "initial state." A (correct) program terminates in its "final state." All other program states are termed as "intermediate states." In Example 1.6, the initial state is [2 u u u], the final state is [9 3 15 15], and there are four intermediate states as indicated.

Program behavior can be modeled as a sequence of states. With every program one can associate one or more states that need to be observed to decide whether or not the program behaves according to its requirements. In some applications it is only the final state that is of interest to the tester. In other applications a sequence of states might be of interest. More complex patterns might also be needed.

A sequence of states is representative of program behavior.

Example 1.7 For the max program (P1.1), the final state is sufficient to decide if the program has successfully determined the maximum of two integers. If the numbers input to max are 3 and 15, then the correct final state is [9 3 15 15]. In fact, it is only the last element of the state vector, 15, which may be of interest to the tester.

Example 1.8 Consider a menu-driven application named myapp. Figure 1.2 shows the menu bar for this application. It allows a user to position and click the mouse on any one of a list of menu items displayed in the menu bar on the screen. This results in the "pulling down" of the menu and a list of options is displayed on the screen. One of the items on the menu bar is labeled File. When File is pulled down, it displays Open as one of several options. When the Open option is selected, by moving the cursor over it, it should be highlighted. When the mouse is released, indicating that the selection is complete, a window displaying names of files in the current directory should be displayed.

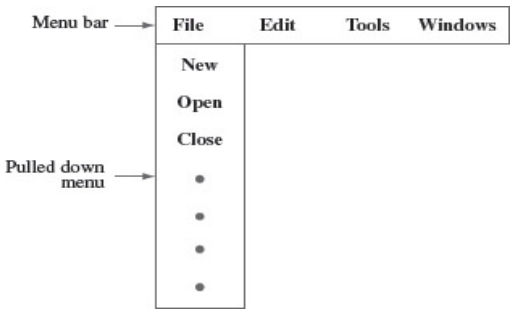
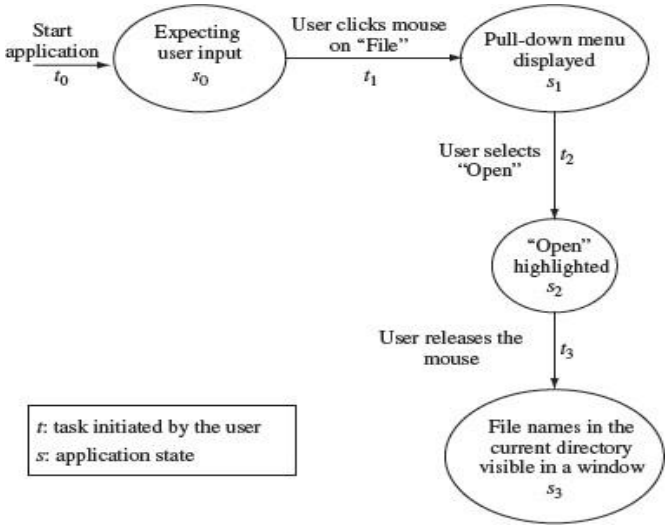


Figure 1.2 Menu bar displaying four menu items when application myapp is started.

Figure 1.3 depicts the sequence of states that myapp is expected to enter when the user actions described above are performed. When started, the application enters the initial state wherein it displays the menu bar and waits for the user to select a menu item. This state diagram depicts the expected behavior of myapp in terms of a state sequence. As shown in Figure 1.3, myapp moves from state s_0 to state s_3 after the sequence of actions t_0, t_1, t_2, and t_3 has been applied. To test myapp, the tester could apply the sequence of actions depicted in this state diagram and observe if the application enters the expected states.

Figure 1.3 A state sequence for myapp showing how the application is expected to behave when the user selects the open option under the File menu item.

As you might observe from Figure 1.3, a state sequence diagram can be used to specify the *behavioral* requirements of an application. This same specification can then be used during testing to ensure if the application conforms to the requirements.



1.3.3 Valid and invalid inputs

In the examples above, the input domains are derived from the requirements. However, due to the incompleteness of requirements, one might have to think a bit harder to determine the input domain. To illustrate why, consider the modified requirement in Example 1.5. The requirement mentions that the request characters can be "A" or "D", but it fails to answer the question "What if the user types a different character?" When using sort it is certainly possible for the user to type a character other than "A" or "D". Any character other than "A" or "D" is considered as invalid input to sort. The requirement for sort does not specify what action it should take when an invalid input is encountered.

A program ought to be tested based on representatives from the set of valid as well as invalid inputs. The latter set is used to determine the robustness of a program.

Identifying the set of invalid inputs and testing, the program against these inputs is an important part of the testing activity. Even when the requirements fail to specify the program behavior on invalid inputs, the programmer does treat these in one way or another. Testing a program against invalid inputs might reveal errors in the program.

Example 1.9 Suppose that we are testing the sort program. We execute it against the following input: < *E* 7 19 . >. The requirements in Example 1.5 are insufficient to determine the expected behavior of sort on the above input. Now suppose that upon execution on the above input, the sort program enters into an infinite loop and neither asks the user for any input nor responds to anything typed by the user. This observed behavior points to a possible error in sort.

The argument above can be extended to apply to the sequence of integers to be sorted. The requirements for the sort program do not specify how the program should behave if, instead of typing an integer, a user types in a character, such as "?". Of course, one would say, the program should inform the user that the input is invalid. But this expected behavior from sort needs to be tested. This suggests that the input domain for sort should be modified.

Example 1.10 Considering that sort may receive valid and invalid inputs, the input domain derived in Example 1.5 needs modification. The modified input domain consists of pairs of values. The first value in the pair is any ASCII character that can be typed by a user as a request character. The second element of the pair is a sequence of integers, interspersed with invalid characters, terminated by a period. Thus, for example, the following are sample elements from the modified input domain:

< A 7 19.>
< D 7 9F 19.>

In the example above, we assumed that invalid characters are possible inputs to the sort program. This, however, may not be the case in all situations. For example, it might be possible to guarantee that the inputs to sort will always be correct as determined from the

modified requirements in Example 1.5. In such a situation, the input domain need not be augmented to account for invalid inputs if the guarantee is to be believed by the tester.
In cases where the input to a program is not guaranteed to be correct, it is convenient to partition the input domain into two subdomains. One subdomain consists of inputs that are valid and the other consists of inputs that are invalid. A tester can then test the program on selected inputs from each subdomain.

➔ Correctness Vs Reliability

1.4.1 Correctness

Though correctness of a program is desirable, it is almost never the objective of testing. To establish correctness via testing would imply testing a program on all elements in the input domain. In most cases that are encountered in practice, this is impossible to accomplish. Thus, correctness is established via mathematical proofs of programs. A proof uses the formal specification of requirements and the program text to prove or disprove that the program will behave as intended. While a mathematical proof is precise, it too is subject to human errors. Even when the proof is carried out by a computer, the simplification of requirements specification and errors in tasks that are not fully automated might render the proof useless.

Program correctness is established via mathematical proofs. Program reliability is established via testing. However, neither approach is foolproof.

While correctness attempts to establish that the program is error free, testing attempts to find if there are any errors in it. Completeness of testing does not necessarily demonstrate that a program is error free. However, as testing progresses, errors might be revealed. Removal of errors from the program usually improves the chances, or the probability, of the program executing without any failure. Also, testing, debugging, and the error removal processes together increase our confidence in the correct functioning of the program under test.

Testing a program for completeness does not ensure its correctness.

Example 1.11 This example illustrates why the probability of program failure might not change upon error removal. Consider the following program that inputs two integers x and y and prints the value of $f(x, y)$ or $g(x, y)$ depending on the condition $x < y$.

```
integer x, y
input x, y
if (x<y)          ← This condition should be x≤y.
  {print f(x,y)}
else
  {print g(x,y)}
```

The above program uses two functions f and g not defined here. Let us suppose that function f produces an incorrect result whenever it is invoked with $x = y$ and that $f(x, y) \neq g(x, y)$, $x = y$. In its present form, the program fails when tested with equal input values because function g is invoked instead of function f. When the error is removed by changing the condition $x < y$ to $x \leq y$, the program fails again when the input values are the same. The latter failure is due to the error in function f. In this program, when the error in f is also removed, the program will be correct assuming that all other code is correct.

1.4.2 Reliability

The probability of a program failure is captured more formally in the term "reliability." Consider the second of the two definitions examined earlier: "The *reliability* of a program P is the probability of its successful execution on a randomly selected element from its input domain."

Reliability is a statistical attribute. According to one view, it is the probability of failure free execution of program under given conditions.

A comparison of program correctness and reliability reveals that while correctness is a binary metric, reliability is a continuous metric over a scale from 0 to 1. A program can be either

correct or incorrect; its reliability can be anywhere between 0 and 1. Intuitively, when an error is removed from a program, the reliability of the program so obtained is expected to be higher than that of the one that contains the error. As illustrated in the example above, this may not be always true. The next example illustrates how to compute program reliability in a simplistic manner.

Example 1.12 Consider a program *P* that takes a pair of integers as input. The input domain of this program is the set of all pairs of integers. Suppose now that in actual use there are only three pairs that will be input to *P.* These are

{<(0, 0) (–l, l) (l, –l)>}

The above set of three pairs is a subset of the input domain of *P* and is derived from knowledge of the actual use of *P,* and not solely from its requirements.Suppose also that each pair in the above set is equally likely to occur in practice. If it is known that *P* fails on exactly one of the three possible input pairs then the frequency with which *P* will function correctly is $\frac{2}{3}$. this number is an estimate of the probability of the successful operation of *P* and hence is the reliability of *P.*

1.4.3 Operational profiles

As per the definition above, the reliability of a program depends on how it is used. Thus, in Example 1.12, if *P* is never executed on input pair (0, 0), then the restricted input domain becomes {< (–1, 1) (1, –1) >} and the reliability of *P* is 1. This leads us to the definition of *operational profile.*

An operational profile is a statistical summary of how a program would be used in practice.

Operational profile

An operational profile is a numerical description of how a program is used.

In accordance with the above definition, a program might have several operational profiles depending on its users.

Example 1.13 Consider a sort program that, on any given execution, allows any one of two types of input sequences. One sequence consists of numbers only and the other consists of alphanumeric strings. One operational profile for sort is specified as follows:

Operational profile #1	
Sequence	Probability
Numbers only	0.9
Alphanumeric strings	0.1

Operational profile #2	
Sequence	Probability
Numbers only	0.1
Alphanumeric strings	0.9

Another operational profile for sort is specified as follows:

The two operational profiles above suggest significantly different uses of sort. In one case, it is used mostly for sorting sequences of numbers, and in the other case, it is used mostly for sorting alphanumeric strings.

➔ Testing & Debugging

Testing is the process of determining if a program behaves as expected. In the process, one may discover errors in the program under test. However, when testing reveals an error, the process used to determine the cause of this error and to remove it is known as *debugging.* As illustrated in Figure 1.4, testing and debugging are often used as two related activities in a cyclic manner.

Testing and debugging are two distinct though intertwined activities. Testing generally leads to debugging though both activities might not be always performed by the same individual.

1.5.1 Preparing a test plan

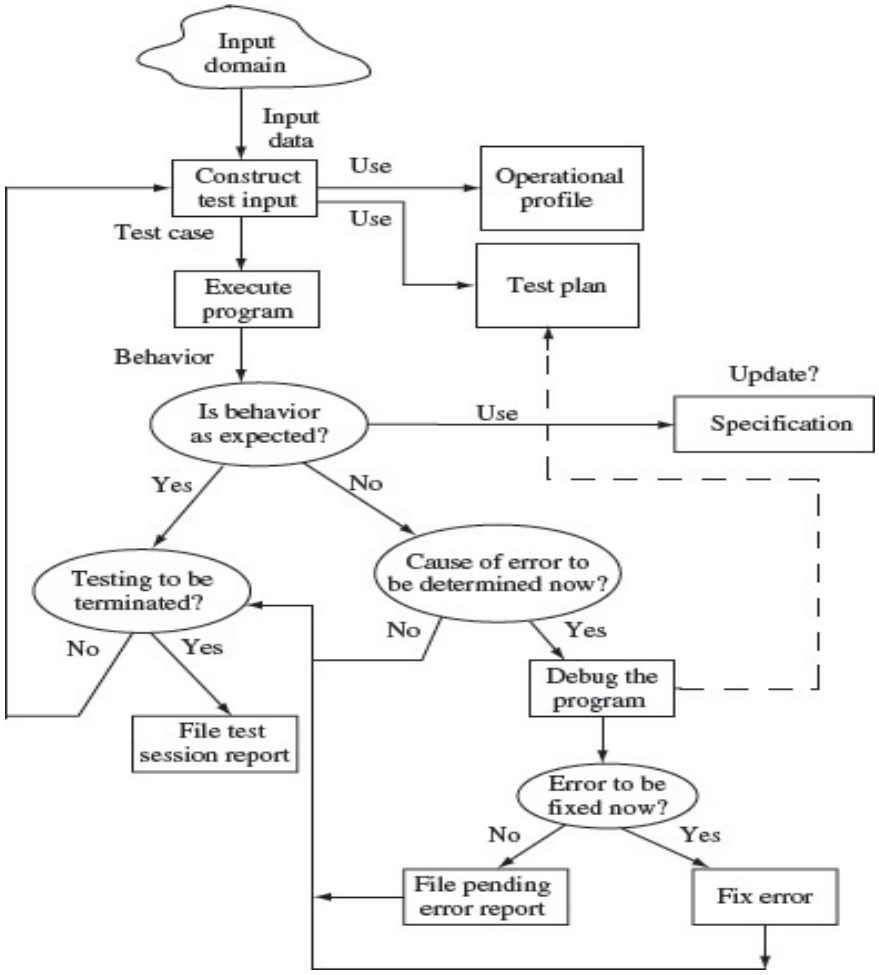


Figure 1.4 A test and debug cycle.

Test Plan for `sort`.

The `sort` program is to be tested to meet the requirements given in Example 1.5. Specifically, the following needs to be done:

1. Execute the program on at least two input sequences, one with "A" and the other with "D" as request characters.
2. Execute the program on an empty input sequence.
3. Test the program for robustness against erroneous inputs such as "R" typed in as the request character.
4. All failures of the test program should be recorded in a suitable file using the Company Failure Report Form.

Figure 1.5 A sample test plan for the sort program.

A test cycle is often guided by a *test plan.* When relatively small programs are being tested, a test plan is usually informal and in the tester's mind, or there may be no plan at all. A sample test plan for testing the sort program is shown in Figure 1.5.

The sample test plan in Figure 1.5 is often augmented by items such as the method used for testing, method for evaluating the adequacy of test cases, and method to determine if a program has failed or not.

1.5.2 Constructing test data

A *test case* is a pair of input data and the corresponding program output. The test data are a set of values: one for each input variable. A *test set* is a collection of test cases. A test set is sometimes referred to as a test suite. The notion of "one execution" of a program is rather tricky and is elaborated later in this chapter. *Test data* is an alternate term for test set.

A test case is a pair of input data and the corresponding program output; a test set is a collection of test cases.

Program requirements and the test plan help in the construction of test data. Execution of the program on test data might begin after all or a few test cases have been constructed. While testing, relatively small programs testers often generate a few test cases and execute the program against these. Based on the results obtained, the tester decides whether to continue the construction of additional test cases or to enter the debugging phase.

Example 1.14 The following test cases are generated for the sort program using the test plan in Figure 1.5.

```
Test case 1:
  Test data:        <"A" 12 -29 32.>
  Expected output: -29 12 32

Test case 2:
  Test data:        <"D" 12 -29 32.>
  Expected output: 32 12 -29

Test case 3:
  Test data:        <"A".>
  Expected output: No input to be sorted in ascending
                    order.

 Test case 4:
   Test data:       <"D".>
   Expected output: No input to be sorted in ascending
                    order.

 Test case 5:
   Test data:        <"R" 3 17.>
   Expected output: Invalid request character
   Valid characters: "A" and "D".

 Test case 6:
   Test data:       <"A" c 17.>
   Expected output: Invalid number.
```

Test cases 1 and 2 are derived in response to item 1 in the test plan; 3 and 4 are in response to item 2. Notice that we have designed two test cases in response to item 2 of the test plan even though the plan calls for only 1 test case. Notice also that the requirements for the sort program as in Example 1.5 do not indicate what should be the output of sort when there is nothing to be sorted. We therefore took an arbitrary decision while composing the "Expected output" for an input that has no numbers to be sorted. Test cases 5 and 6 are in response to item 3 in the test plan.

As is evident from the above example, one can select a variety of test sets to satisfy the test plan requirements. Questions such as "Which test set is the best?" and "Is a given test set adequate?" are answered in Part III of this book.

1.5.3 Executing the program

Execution of a program under test is the next significant step in testing. Execution of this step for the sort program is most likely a trivial exercise. However, this may not be so for large and complex programs. For example, to execute a program that controls a digital cross connect switch used in telephone networks, one may first need to follow an elaborate procedure to load the program into the switch and then yet another procedure to input the test cases to the program. Obviously, the complexity of actual program execution is dependent on the program itself.

A test harness is an aid to testing a program.

Often a tester might be able to construct a *test harness* to aid in program execution. The harness initializes any global variables, inputs a test case, and executes the program. The output generated by the program may be saved in a file for subsequent examination by a tester. The next example illustrates a simple test harness.

Example 1.15 The test harness in Figure 1.6 reads an input sequence, checks for its correctness, and then calls sort. The sorted array sorted_sequence returned by sort is printed using the print_sequence procedure. Test cases are assumed to be available in the Test pool file shown in the figure. In some cases, the tests might be generated from within the harness.

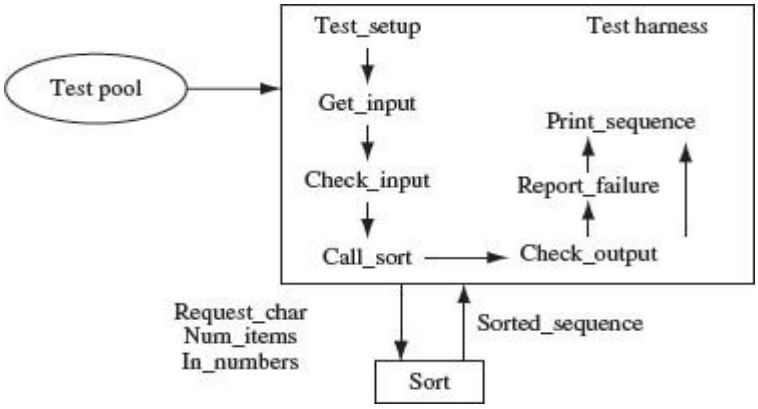


Figure 1.6 A simple test harness to test the sort program.

In preparing this test harness we assume that (a) sort is coded as a procedure, (b) the get_input procedure reads the request character and the sequence to be sorted into variables request_char, num_items, and in_numbers, and (c) the input is checked prior to calling sort by the check_input procedure.

The test_setup procedure is usually invoked first to set up the test that, in this example, includes identifying and opening the file containing tests. The check_output procedure serves as the oracle that checks if the program under test behaves correctly.

The report_failure procedure is invoked in case the output from sort is incorrect. A failure might be simply reported via a message on the screen or saved in a test report file (not shown in Figure 1.6). The print_sequence procedure prints the sequence generated by the sort program. The output generated by print_sequence can also be piped into a file for subsequent examination.

1.5.4 Assessing program correctness

An important step in testing a program is the one wherein the tester determines if the observed behavior of the program under test is correct or not. This step can be further divided into two smaller steps. In the first step one observes the behavior and in the second step analyses the observed behavior to check if it is correct or not. Both these steps can be trivial for small programs, such as for max in Example 1.3, or extremely complex as in the case of a large distributed software system. The entity that performs the task of checking the correctness of the

observed behavior is known as an *oracle.* Figure 1.7 shows the relationship between the program under test and the oracle.

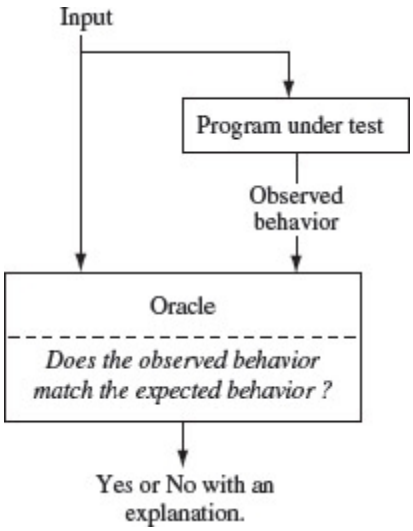


Figure 1.7 Relationship between the program under test and the oracle. The output from an oracle can be binary such as "Yes" or "No" or more complex such as an explanation as to why the oracle finds the observed behavior to be same or different from the expected behavior.

Testing	Debugging
1. Testing always starts with known conditions, uses predefined methods, and has predictable outcomes too.	1. Debugging starts from possibly un-known initial conditions and its end cannot be predicted, apart from statistically.
2. Testing can and should definitely be planned, designed, and scheduled.	2. The procedures for, and period of, debugging cannot be so constrained.
3. It proves a programmers failure.	3. It is the programmer's vindication.
4. It is a demonstration of error or apparent correctness.	4. It is always treated as a deductive process.
5. Testing as executed should strive to be predictable, dull, constrained, rigid, and inhuman.	5. Debugging demands intuitive leaps, conjectures, experimentation, and some freedom also.
6. Much of the testing can be done without design knowledge.	6. Debugging is impossible without detailed design knowledge.
7. It can often be done by an outsider.	7. It must be done by an insider.
8. Much of test execution and design can be automated.	8. Automated debugging is still a dream for programmers.
9. Testing purpose is to find bug.	9. Debugging purpose is to find cause of bug.

A tester often assumes the role of an oracle and thus serves as a "human oracle." For example, to verify if the output of a matrix multiplication program is correct or not, a tester might input two 2 × 2 matrices and check if the output produced by the program matches the results of hand calculation. As another example, consider checking the output of a text processing program. In this case, a human oracle might visually inspect the monitor screen to verify whether or not the italicize command works correctly when applied to a block of text.

An oracle is something that checks the behavior of a program. An oracle could itself be a program or a human being.

Checking program behavior by humans has several disadvantages. First, it is error prone as the human oracle might make an error in analysis. Second, it may be slower than the speed with which the program computed the results. Third, it might result in the checking of only trivial input–output behaviors. However, regardless of these disadvantages, a human oracle can often be the best available oracle.

Oracles can also be programs designed to check the behavior of other programs. For example, one might use a matrix multiplication program to check if a matrix inversion program has produced the correct output. In this case, the matrix inversion program inverts a given matrix A and generates B as the output matrix. The multiplication program can check to see if $A \times B = I$ within some bounds on the elements of the identity matrix I. Another example is an oracle that checks the validity of the output from a sort program. Assuming that the sort program sorts input numbers in ascending order, the oracle needs to check if the output of the sort program is indeed in ascending order.

Using programs as oracles has the advantage of speed, accuracy, and the ease with which complex computations can be checked. Thus, a matrix multiplication program, when used as an oracle for a matrix inversion program, can be faster, accurate, and check very large matrices when compared to the same function performed by a human oracle.

➔ Principles of Testing

The seven testing principles are as follows:

Principle 1: Testing shows presence of defects: Testing can show that defects are present, but cannot prove that there are no defects. Testing reduces the probability of undiscovered defects remaining in the software but even if no defects are found, it is not a proof of correctness. In other words, one can never assume that there are no defects or the application is 100 percent bug free even if thorough testing is done.

Principle 2: Exhaustive testing is impossible: Testing everything (all combinations of inputs and preconditions) is not feasible except for trivial cases. Instead of exhaustive testing, risk analysis and priorities should be used to focus testing efforts. For example, if we are testing a text box that accepts numbers between 0 to 100, we would test for boundary values, one less than boundary value, one more than boundary values, few random numbers, middle number, that's it and assume that if it is working fine for these numbers it will work for other numbers also. We are not testing for each number from 1 to 100.

Principle 3: Early testing: To find defects early, testing activities shall be started as early as possible in the software or system development life cycle, and shall be focused on defined objectives. If the testing team is involved right from the beginning of the requirement gathering and analysis phase they have better understanding and insight into the product and moreover the cost of quality will be much less if the defects are found as early as possible rather than later in the development life cycle.

Principle 4: Defect clustering: Testing effort shall be focused proportionally to the expected and later observed defect density of modules. A small number of modules usually contains most of the defects discovered during pre-release testing, or is responsible for most of the operational failures. The Pareto principle of 80:20 works here that is 80 percent of defects are due to 20 percent of code! This information could prove to be very helpful while testing, if we find one defect in a particular module/area there is pretty high chance of getting many more there itself.

Principle 5: Pesticide paradox: If the same kinds of tests are repeated again and again, eventually the same set of test cases will no longer be able to find any new bugs. To overcome this "Pesticide Paradox", test cases need to be regularly reviewed and revised, and the new and different tests need to be written to exercise different parts of the software or system to find potentially more defects.

Principle 6: Testing is context dependent: Testing is done differently in different contexts. For example, safety – critical software is tested differently from an e-commerce site. Very true, testing effort should be based on what is to be tested. Testing focus will depend on what is more important for that type of application.

Principle 7: Absence–of–errors fallacy: If the system built is unusable and does not fulfil the user's needs and expectations then finding and fixing defects does not help. As said, if the product does not meet user's requirements – explicitly mentioned and implicitly implied, that is if it is not fit for use, there is no point in testing, finding defects and fixing it.

➔ Test Metrics

The term "metric" refers to a standard of measurement. In software testing, there exist a variety of metrics. Figure 1.9 shows a classification of various types of metrics briefly discussed in this section. Metrics can be computed at the organizational, process, project, and product levels. Each set of measurements has its value in monitoring, planning, and control.

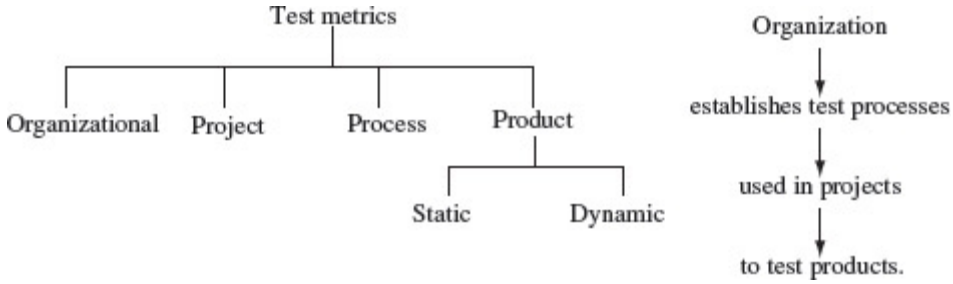


Figure 1.9 Types of metrics used in software testing and their relationships.

A test metric measures some aspect of the test process. Test metrics could be at various levels such as at the level of an organization, a project, a process or a product.

Regardless of the level at which metrics are defined and collected, there exist four general core areas that assist in the design of metrics. These are schedule, quality, resources, and size. Schedule-related metrics measure actual completion times of various activities and compare these with estimated time to completion. Quality-related metrics measure the quality of a product or a process. Resource-related metrics measure items such as cost in dollars, manpower, and tests executed. Size-related metrics measure the size of various objects such as the source code and the number of tests in a test suite.

1.6.1 Organizational metrics

Metrics at the level of an organization are useful in overall project planning and management. Some of these metrics are obtained by aggregating compatible metrics across multiple projects. Thus, for example, the number of defects reported after product release, averaged over a set of products developed and marketed by an organization, is a useful metric of product quality at the organizational level.

Computing this metric at regular intervals and overall products released over a given duration shows the quality trend across the organization. For example, one might say "The number of defects reported in the field over all products and within three months of their shipping, has dropped from 0.2 defects per KLOC (thousand lines of code) to 0.04 defects KLOC." Other organizational level metrics include testing cost per KLOC, delivery schedule slippage, and time to complete system testing.

Defect density is the number of defects per line of code. Defect density is a widely used metric at the level of a project.

Organizational level metrics allow senior management to monitor the overall strength of the organization and points to areas of weaknesses. These metrics help the senior management in setting new goals and plan for resources needed to realize these goals.

Example 1.17 The average defect density across all software projects in a company is 1.73 defects per KLOC. Senior management has found that for the next generation of software products, which they plan to bid, they need to show that product density can be reduced to 0.1 defects per KLOC. The management thus sets a new goal.

Given the time frame from now until the time to bid, the management needs to do a feasibility analysis to determine whether or not this goal can be met. If a preliminary analysis shows that it can be met, then a detailed plan needs to be worked out and put into action. For example, management might decide to train its employees in the use of new tools and techniques for defect prevention and detection using sophisticated static analysis techniques.

1.6.2 Project metrics

Project metrics relate to a specific project, for example, the I/O device testing project or a compiler project. These are useful in the monitoring and control of a specific project. The ratio of actual to planned system test effort is one project metric. Test effort could be measured in terms of the tester-man-months. At the start of the system test phase, for example, the project manager estimates the total system test effort. The ratio of actual to estimated effort is zero prior to the system test phase. This ratio builds up over time. Tracking the ratio assists the project manager in allocating testing resources.

Another project metric is the ratio of the number of successful tests to the total number of tests in the system test phase. At any time during the project, the evolution of this ratio from the start of the project could be used to estimate the time remaining to complete the system test process.

1.6.3 Process metrics

Every project uses some test process. The "big bang" approach is one process sometimes used in relatively small single person projects. Several other well-organized processes exist. The goal of a process metric is to assess the "goodness" of the process.

When a test process consists of several phases, for example, unit test, integration test, system test, etc, one can measure how many defects were found in each phase. It is well known that the later a defect is found, the costlier it is to fix. Hence, a metric that classifies defects according to the phase in which they are found assists in evaluating the process itself.

The purpose of a process metric is to assess the "goodness" of a process.

Example 1.18 In one software development project, it was found that 15% of the total defects were reported by customers, 55% of the defects prior to shipping were found during system test, 22% during integration test, and the remaining during unit test. The large number of defects found during the system test phase indicates a possibly weak integration and unit test process. The management might also want to reduce the fraction of defects reported by customers.

1.6.4 Product metrics: generic

Product metrics relate to a specific product such as a compiler for a programming language. These are useful in making decisions related to the product, for example, "Should this product be released for use by the customer?"

Product quality-related metrics abound. We introduce two types of metrics here: the cyclomatic complexity and the Halstead metrics. The cyclomatic complexity proposed by Thomas McCabe in 1976 is based on the control flow of a program. Given the CFG G (see Chapter 2.2 for details) of program P containing N nodes, E edges, and p connected procedures, the cyclomatic complexity $V(G)$ is computed as follows:

$V(G) = E - N + 2p$

The cyclomatic complexity is a measure of program complexity; higher values imply higher complexity. This is a product metric.

Note that P might consist of more than one procedure. The term p in $V(G)$ counts only procedures that are reachable from the main function. $V(G)$ is the complexity of a CFG G that

corresponds to a procedure reachable from the main procedure. Also, $V(G)$ is not the complexity of the entire program, instead it is the complexity of a procedure in P that corresponds to G (see Exercise 1.12). Larger values of $V(G)$ tend to imply higher program complexity and hence a program more difficult to understand and test than one with a smaller values. $V(G)$ is 5 or less are recommended.

The now well-known Halstead complexity measures were published by late Professor Maurice Halstead in a book titled "Elements of Software Science." Table 1.2 lists some of the software science metrics. Using program size (S) and effort (E), the following estimator has been proposed for the number of errors (B) found during a software development effort:

$B = 7.6E^{0.667}S^{0.333}$

Extensive empirical studies have been reported to validate Halstead's software science metrics. An advantage of using an estimator such as B is that it allows the management to plan for testing resources. For example, a larger value of the number of expected errors will lead to a larger number of testers and testing resources to complete the test process over a given duration. Nevertheless, modern programming languages such as Java and C++ do not lend themselves well to the application of the software science metrics. Instead, one uses specially devised metrics for object-oriented languages described next.

Measure	Notation	Definition
Operator count	N_1	Number of operators in a program
Operand count	N_2	Number of operands in a program.
Unique operators	η_1	Number of unique operators in a program
Unique operands	η_2	Number of unique operands in a program
Program vocabulary	η	$\eta_1 + \eta_2$
Program size	N	$N_1 + N_2$
Program volume	V	$N \times \log_2\eta$
Difficulty	D	$2/\eta_1 \times \eta_2/N_2$
Effort	E	$D \times V$

Table 1.2 Halstead measures of program complexity and effort.

1.6.5 Product metrics: OO software

A number of empirical studies have investigated the correlation between product complexity metric application qualities. Table 1.3 lists a sample of product metrics for object-oriented and other applications. Product reliability is a quality metric and refers to the probability of product failure for a given operational profile. As explained in Section 1.4.2, product reliability of software truly measures the probability of generating a failure causing test input. If for a given operational profile and in a given environment this probability is 0, then the program is perfectly reliable despite the possible presence of errors. Certainly, one could define other metrics to assess software reliability. A number of other product quality metrics, based on defects, are listed in Table 1.3.

The OO metrics in the table are due to Shyam Chidamber and Chris Kemerer. They measure program or design complexity. They are of direct relevance to testing in that a product with a complex design will likely require more test effort to obtain a given level of defect density than a product with less complexity.

Table 1.3 A sample of product metrics.

Metric	Meaning
Reliability	Probability of failure of a software product with respect to a given operational profile in a given environment.
Defects density	Number of defects per KLOC.
Defect severity	Distribution of defects by their level of severity
Test coverage	Fraction of testable items, e.g., basic blocks, covered. Also, a metric for test adequacy or "goodness of tests."
Cyclomatic complexity	Measures complexity of a program based on its CFG.
Weighted methods per class	$\sum_{i=1}^{n} c_i$, where c_i is the complexity of method i in the class under consideration.
Class coupling	Measures the number of classes to which a given class is coupled.
Response set	Set of all methods that can be invoked, directly and indirectly, when a message is sent to object O.
Number of children	Number of immediate descendants of a class in the class hierarchy.

1.6.6 Progress monitoring and trends

Metrics are often used for monitoring progress. This requires making measurements on a regular basis over time. Such measurements offer trends. For example, suppose that a browser has been coded, unit tested, and its components integrated. It is now in the system testing phase. One could measure the cumulative number of defects found and plot these over time. Such a plot will rise over time. Eventually, it will likely show a saturation indicating that the product is reaching a stability stage. Figure 1.10 shows a sample plot of new defects found over time.

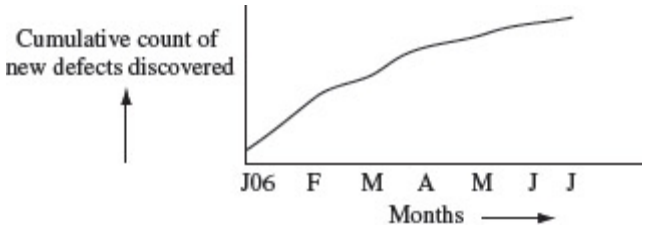


Figure 1.10 A sample plot of cumulative count of defects found over seven consecutive months in a software project.

1.6.7 Static and dynamic metrics

Static metrics are those computed without having to execute the product. Number of testable entities in an application is an example of a static product metric. Dynamic metrics require code execution. For example, the number of testable entities actually covered by a test suite is a dynamic product metric.

Product metrics could be classified as static or dynamic. Computing a dynamic metric will likely require program execution.

One could apply the notions of static and dynamic metrics to organization and project. For example, the average number of testers working on a project is a static project metric. Number of defects remaining to be fixed could be treated as a dynamic metric as it can be computed accurately only after a code change has been made and the product retested.

1.6.8 Testability

According to IEEE, testability is the "degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those

criteria have been met." Different ways to measure testability of a product can be categorized into static and dynamic testability metrics. Software complexity is one static testability metric. The more complex an application, the lower the testability, that is, the higher the effort required to test it. Dynamic metrics for testability include various code-based coverage criteria. For example, a program for which it is difficult to generate tests that satisfy the statement coverage criterion is considered to have low testability than one for which it is easier to construct such tests.

The testability of a program is the degree to which a program facilitates the establishment and measurement of some performance criteria.

High testability is a desirable goal. This is done best by knowing what needs to be tested and how, well in advance. It is recommended that features to be tested and how they are to be tested must be identified during the requirements stage. This information is then updated during the design phase and carried over to the coding phase. A testability requirement might be met through the addition of some code to a class. In more complex situations, it might require special hardware and probes in addition to special features aimed solely at meeting a testability requirement.

Example 1.19 Consider an application E required to control the operation of an elevator. E must pass a variety of tests. It must also allow a tester to perform a variety of tests. One test checks if the elevator hoists motor and the brakes are working correctly. Certainly one could do this test when the hardware is available. However, for concurrent hardware and software development, one needs a simulator for the hoist motor and brake system.

To improve the testability of E, one must include a component that allows it to communicate with a hoist motor and break simulator and display the status of the simulated hardware devices. This component must also allow a tester to input tests such as "start the motor."

Another test requirement for E is that it must allow a tester to experiment with various scheduling algorithms. This requirement can be met by adding a component to E that offers a palette of scheduling algorithms to choose from and whether or not they have been implemented. The tester selects an implemented algorithm and views the elevator movement in response to different requests. Such testing also requires a "random request generator" and a display of such requests and the elevator response.

Testability is a concern in both hardware and software designs. In hardware design, testability implies that there exist tests to detect any fault with respect to a fault model in a finished product. Thus, the aim is to verify the correctness of a finished product. Testability in software focuses on the verification of design and implementation.

2. Testing in the Software Life Cycle & Test Levels

➔ The General V-Model

The main idea behind the general V-model is that development and testing tasks are corresponding activities of equal importance. The two branches of the *V* symbolize this.

The left branch represents the development process. During development, the system is gradually being designed and finally programmed. The right branch represents the integration and testing process; the program elements are successively being assembled to form larger subsystems (integration), and their functionality is tested. Integration and testing end when the acceptance test of the entire system has been completed. Figure 3-1 shows such a V-model.

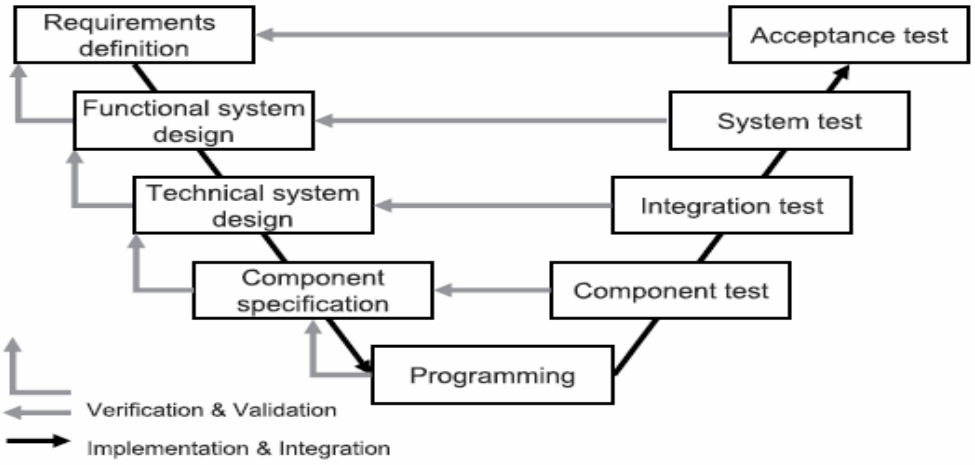


Figure 3–1 the general V-model

The constructive activities of the left branch are the activities known from the waterfall model:

■ **Requirements definition**
The needs and requirements of the customer or the future system user are gathered, specified, and approved. Thus, the purpose of the system and the desired characteristics are defined.

■ **Functional system design**
This step maps requirements onto functions and dialogues of the new system.

■ **Technical system design**
This step designs the implementation of the system. This includes the definition of interfaces to the system environment and decomposing the system into smaller, understandable subsystems (system architecture). Each subsystem can then be developed as independently as possible.

■ **Component specification**
This step defines each subsystem, including its task, behavior, inner structure, and interfaces to other subsystems.

■ **Programming**
Each specified component (module, unit, class) is coded in a programming language.

Through these construction levels, the software system is described in more and more detail. Mistakes can most easily be found at the abstraction level where they occurred.
Thus, for each specification and construction level, the right branch of the V-model defines a corresponding test level:

■ **Component test**
Verifies whether each software component correctly fulfills its specification.

■ **Integration test**

Checks if groups of components interact in the way that is specified by the technical system design.

■ **System test**
Verifies whether the system as a whole meets the specified requirements.

■ **Acceptance test**
Checks if the system meets the customer requirements, as specified in the contract and/or if the system meets user needs and expectations.

Within each test level, the tester must make sure the outcomes of development meet the requirements that are relevant or specified on this specific level of abstraction. This process of checking the development result according to their original requirements is called →validation.

Advantages:

- The process is completed once at a time.
- This model is easy to understand and simple to use.
- This Model Is Also Working Well For Smaller Projects, Where Requirements Are Understood Very Well.
- Each phase has specific deliverables and a review process.
- It covers all functional areas.
- It contains instructions and recommendations, which provide a detailed explanation of the problems involved.
- Emphasize for verification and validation of the product in early stages of product development.
- Each stage is testable
- Project management can track progress by milestones
- Easy to understand implement and use

Disadvantages:

- In this model, there is possibly of high risk and uncertainty.
- This model is not useful for long and ongoing projects.
- It is not suitable for the projects where requirements are at a moderate to high risk of changing.
- This model is not good for complex and object oriented projects.
- Does not easily handle events concurrently.
- Does not handle iterations or phases
- Does not easily handle dynamic changes in requirements
- Does not contain risk analysis or Mitigation activities

Does a product solve the intended task?

When validating, the tester judges whether a (partial) product really solves the specified task and whether it is fit or suitable for its intended use.

Is it the right system?

The tester investigates to see if the system makes sense in the context of intended product use.

Does a product fulfill its *specification?*

In addition to validation testing, the V-model requires verification3 testing. Unlike validation, →verification refers to only one single phase of the development process. Verification shall assure that the outcome of a particular development level has been achieved correctly and completely, according to its specification (the input documents for that development level).

Is the system correctly built?

Verification activities examine whether specifications are correctly implemented and whether the product meets its specification, but not whether the resulting product is suitable for its intended use.

In practice, every test contains both aspects. On higher test levels the validation part increases. To summarize, we again list the most important characteristics and ideas behind the general V-model:

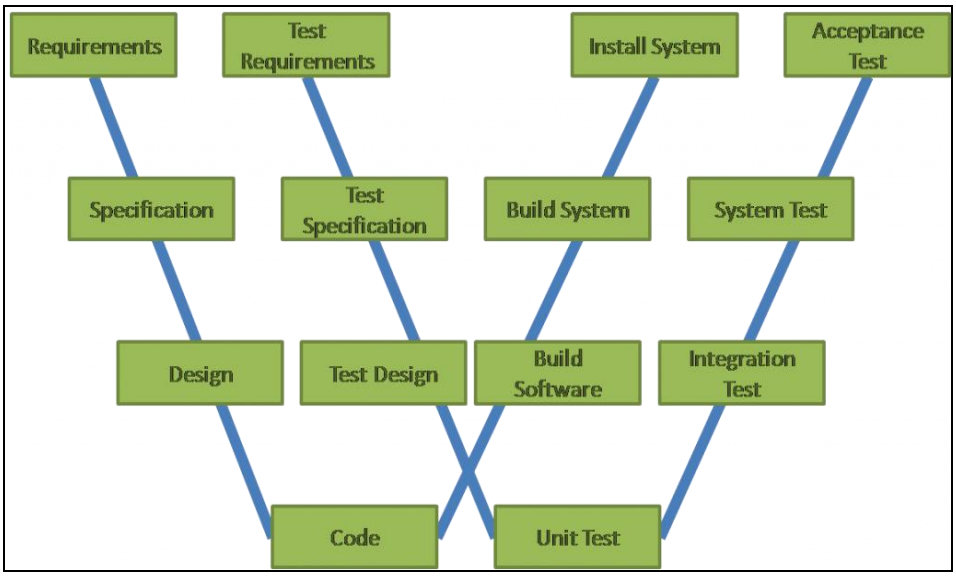
Implementation and testing activities are separated but are equally important (left side / right side).

■ The *V* illustrates the testing aspects of verification and validation.
■ we distinguish between different test levels, where each test level is testing "against" its corresponding development level.

The V-model may give the impression that testing starts relatively late, after system implementation, but this is not the case. The test levels on the right branch of the model should be interpreted as levels of test execution. Test preparation (test planning, test analysis and design) starts earlier and is performed in parallel to the development phases on the left branch4 (not explicitly shown in the V-model).

The differentiation of test levels in the V-model is more than a temporal subdivision of testing activities. It is instead defining technically very different test levels; they have different objectives and thus need different methods and tools and require personnel with different knowledge and skills. The exact contents and the process for each test level are explained in the following sections.

➔ W-Model



From the testing point of view all models are deficient in various ways. The test activities first start after the implementation. The Connection between various test stages and the basis for the test is not clear. The Tight link between tests, debug and change tasks during the phase is not clear.

In this models presented above, there usually appears an unattractive task to be carried out after coding. In order to testing on an equal footing, a second "v" dedicated to testing is integrated into the model. But "V's" put together give the "W" of the ""W-Model".

The W Model removes the vague and ambiguous lines linking the left and right legs of the V and replaces them with parallel testing activities, shadowing each of the development activities.

As the project moves down the left leg, the **testers carry out static testing**(i.e. inspections and walkthroughs) of the deliverables at each stage. Ideally prototyping and early usability testing would be included to test the system design of interactive systems at a time when it would be easy to solve problems. The emphasis would then switch to dynamic testing once the project moves into the integration leg.

There are several interesting aspects to the W Model. Firstly, it drops the arbitrary and unrealistic assumption that there should be a testing stage in the right leg for each development stage in the left leg. Each of the development stages has its testing shadow, within the same leg.

The illustration shows a typical example where there are the same number of stages in each leg, but it's possible to vary the number and the nature of the testing stages as circumstances require without violating the principles of the model.

Also, it explicitly does not require the test plan for each dynamic test stage to be based on the specification produced in the twin stage on the left hand side. There is no twin stage of course, but this does address one of the undesirable by-products of a common but unthinking adoption of the V Model; a blind insistence that test plans should be generated from these equivalent documents, and only from those documents.

A crucial advantage of the W Model is that it encourages testers to define tests that can be built into the project plan, and on which development activity will be dependent, thus making it harder for test execution to be squeezed at the end of the project.

However, starting formal test execution in parallel with the start of development must not mean token reviews and sign-offs of the documentation at the end of each stage. Commonly under the V Model, and the Waterfall, test managers receive specifications with the request to review and sign off within a few days what the developers hope is a completed document. In such circumstances test managers who detect flaws can be seen as obstructive rather than constructive. Such last minute "reviews" do not count as early testing.

➔ Component Test

3.2.1 Explanation of Terms

Within the first test level (component testing), the software units are tested systematically for the first time. The units have been implemented in the programming phase just before component testing in the V-model.

Depending on the programming language the developers used, these software units may be called by different names, such as, for example, modules and units. In object-oriented programming, they are called classes. The respective tests, therefore, are called →module tests, →unit tests (see [IEEE 1008]), and →class tests.

Generally, we speak of software units or components. Testing of a single software component is therefore called component testing.

Component testing is based on component requirements, and the component design (or detailed design). If white box test cases will be developed or white box →test coverage will be measured, the source code can also be analyzed. However, the component behavior must be compared with the component specification.

3.2.2 Test objects

Typical test objects are program modules/units or classes, (database) scripts, and other software components. The main characteristic of component testing is that the software components are tested individually and isolated from all other software components of the

system. The isolation is necessary to prevent external influences on components. If testing detects a problem, it is definitely a problem originating from the component under test itself.

The component under test may also be a unit composed of several other components. But remember that aspects internal to the components are examined, not the components' interaction with neighboring components. The latter is a task for integration tests.

Component tests may also comprise data conversion and migration components. Test objects may even be configuration data and database components.

3.2.3 Test Environment

Component testing as the lowest test level deals with test objects coming "right from the developer's desk." It is obvious that in this test level there is close cooperation with development.

Example: Testing of a class method

In the VSR subsystem *DreamCar*, the specification for calculating the price of the car states the following:

■ The starting point is base price minus discount, where base price is the general basic price of the vehicle and discount is the discount to this price granted by the dealer.

■ A price (special price) for a special model and the price for extra equipment items (extra price) shall be added.

■ If three or more extra equipment items (which are not part of the special model chosen) are chosen (extras), there is a discount of 10 percent on these particular items. If five or more special equipment items are chosen, this discount is increased to 15 percent.

■ The discount that is granted by the dealer applies only to the base price, whereas the discount on special items applies to the special items only. These discounts cannot be combined for everything.

The following C++-function calculates the total price:5

```
double calculate_price
(double baseprice, double specialprice,
double extraprice, int extras, double discount)
{
double addon_discount;
double result;
if (extras >= 3) addon_discount = 10;
else if (extras >= 5) addon_discount = 15;
else addon_discount = 0;
if (discount > addon_discount)
addon_discount = discount;
result = baseprice/100.0*(100-discount)
+ specialprice
+ extraprice/100.0*(100-addon_discount);
return result;
}
```

In order to test the price calculation, the tester uses the corresponding class interface calling the function calculate_price() with appropriate parameters and data. Then the tester records the function's reaction to the function call. That means reading and recording the return value of the previous function call. For that, a test driver is necessary. A test driver is a program that calls the component under test and then receives the test object's reaction.

For the test object calculate_price(), a very simple test driver could look like this:

```
bool test_calculate_price() {
double price;
bool test_ok = TRUE;
// testcase 01
price = calculate_price(10000.00,2000.00,1000.00,3,0);
test_ok = test_ok && (abs (price-12900.00) < 0.01);
// testcase 02
price = calculate_price(25500.00,3450.00,6000.00,6,0);
test_ok = test_ok && (abs (price-34050.00) < 0.01);
// testcase ...
```

```
// test result
return test_ok;}
```

The preceding test driver is programmed in a very simple way. Some useful extensions could be, for example, a facility to record the test data and the results, including date and time of the test, or a function that reads test cases from a table, file, or database.

To write test drivers, programming skills and knowledge of the component under test are necessary. The component's program code must be available. The tester must understand the test object (in the example, a class function) so that the call of the test object can be correctly programmed in the test driver. To write a suitable test driver, the tester must know the programming language and suitable programming tools must be available.

This is why the developers themselves usually perform the component testing. Although this is truly a component test, it may also be called developer test. The disadvantages of a programmer testing his own program were discussed in section 2.3.

Often, component testing is also confused with debugging. But debugging is not testing. Debugging is finding the cause of failures and removing them, while testing is the systematic approach for finding failures.

3.2.4 Test objectives

The test level called component test is not only characterized by the kind of test objects and the testing environment, the tester also pursues test objectives that are specific for this phase.

Testing the functionality

The most important task of component testing is to check that the entire functionality of the test object works correctly and completely as required by its specification (see →functional testing). Here, *functionality* means the input/output behavior of the test object. To check the correctness and completeness of the implementation, the component is tested with a series of test cases, where each test case covers a particular input/ output combination (partial functionality).

Example:
Test of the VSR price calculation
The test cases for the price calculation of *DreamCar* in the previous example very clearly show how the examination of the input/output behavior works. Each test case calls the test object with a particular combination of data; in this example, the price for the vehicle in combination with a different set of extra equipment items. It is then examined to see whether the test object, given this input data, calculates the correct price. For example, test case 2 checks the partial functionality of "discount with five or more special equipment items." If test case 2 is executed, we can see that the test object calculates the wrong total price. Test case 2 produces a failure. The test object does not completely meet the functional requirements.

Typical software defects found during functional component testing are incorrect calculations or missing or wrongly chosen program paths (e.g., special cases that were forgotten or misinterpreted).

Later, when the whole system is integrated, each software component must be able to cooperate with many neighboring components and exchange data with them. A component may then possibly be called or used in a wrong way, i.e., not in accordance with its specification. In such cases, the wrongly used component should not just suspend its service or cause the whole system to crash. Rather, it should be able to handle the situation in a reasonable and robust way.

Testing robustness

This is why testing for →robustness is another very important aspect of component testing. The way to do this is the same as in functional testing. However, the test focuses on items either not allowed or forgotten in the specification. The tests are function calls, test data, and special cases. Such test cases are also called →negative tests. The component's reaction should be an appropriate exception handling. If there is no such exception handling, wrong

inputs can trigger domain faults like division by zero or access to a null pointer. Such faults could lead to a program crash.

Example:
Negative test
In the price calculation example, such negative tests are function calls with negative values, values that are far too large, or wrong data types (for example, char instead of int):8

```
// testcase 20
price = calculate_price(-1000.00,0.00,0.00,0,0);
test_ok = test_ok && (ERR_CODE == INVALID_PRICE);
…
// testcase 30
price = calculate_price(”abc”,0.00,0.00,0,0);
test_ok = test_ok && (ERR_CODE == INVALID_ARGUMENT);
```

Excursion
Some interesting aspects become clear:
■ There are at least as many reasonable negative tests as positive ones.
■ The test driver must be extended in order to be able to evaluate the test object's exception handling.
■ The test object's exception handling (the analysis of ERR_CODE in the previous example) requires additional functionality. Often more than 50% of the program code deals with exception handling. Robustness has its cost.

Component testing should not only check functionality and robustness.

All the component's characteristics that have a crucial influence on its quality and that cannot be tested in higher test levels (or only with a much higher cost) should be checked during component testing. This may be nonfunctional characteristics like efficiency and maintainability.

Efficiency test

Efficiency refers to how efficiently the component uses computer resources. Here we have various aspects such as use of memory, computing time, disk or network access time, and the time required to execute the component's functions and algorithms. In contrast to most other nonfunctional tests, a test object's efficiency can be measured during the test. Suitable criteria are measured exactly (e.g., memory usage in kilobytes, response times in milliseconds). Efficiency tests are seldom performed for all the components of a system. Efficiency is usually only verified in efficiency-critical parts of the system or if efficiency requirements are explicitly stated by specifications. This happens, for example, in testing embedded software, where only limited hardware resources are available. Another example is testing real-time systems, where it must be guaranteed that the system follows given timing constraints.

Maintainability test

A maintainability test includes all the characteristics of a program that have an influence on how easy or how difficult it is to change the program or to continue developing it. Here, it is crucial that the developer fully understands the program and its context. This includes the developer of the original program who is asked to continue development after months or years as well as the programmer who takes over responsibility for a colleague's code. The following aspects are most important for testing maintainability: code structure, modularity, quality of the comments in the code, adherence to standards, understandability, and currency of the documentation.

Example:
Code that is difficult to maintain
The code in the example calculate_price() is not good enough. There are no comments, and numeric constants are not declared but are just written into the code. If such a value must be changed later, it is

not clear whether and where this value occurs in other parts of the system, nor is it clear how to find and change it.

Of course, such characteristics cannot be tested by →dynamic tests. Analysis of the program text and the specifications is necessary. →Static testing and especially reviews (see section 4.1) are the correct means for that purpose. However, it is best to include such analyses in the component test because the characteristics of a single component are examined.

3.2.5 Test Strategy

As we explained earlier, component testing is very closely related to development. The tester usually has access to the source code, which makes component testing the domain of white box testing (see section 5.2).

White box test

The tester can design test cases using her knowledge about the component's program structures, functions, and variables. Access to the program code can also be helpful for executing the tests. With the help of special tools (→debugger, see section 7.1.4), it is possible to observe program variables during test execution. This helps in checking for correct or incorrect behavior of the component. The internal state of a component cannot only be observed; it can even be manipulated with the debugger. This is especially useful for robustness tests because the tester is able to trigger special exceptional situations.

Example:
Code as test basis
Analyzing the code of calculate_price(), the following command can be recognized as a line that is relevant for testing:
if (discount > addon_discount)
addon_discount = discount;
Additional test cases that lead to fulfilling the condition (discount > addon_discount) can easily be derived from the code. The specification of the price calculation contains no information about this situation; the implemented functionality is extra: it is not supposed to be there.

In reality, however, component testing is often done as a pure black box testing, which means that the code structure is not used to design test cases. On the one hand, real software systems consist of countless elementary components; therefore, code analysis for designing test cases is probably only feasible with very few selected components.

On the other hand, the elementary components will later be integrated into larger units. Often, the tester only recognizes these larger units as units that can be tested, even in component testing. Then again, these units are already too large to make observations and interventions on the code level with reasonable effort. Therefore, integration and testing planning must answer the question of whether to test elementary parts or only larger units during component testing.

"Test first" development

Test first programming is a modern approach in component testing. The idea is to design and automate the tests first and program the desired component afterwards. This approach is very iterative. The program code is tested with the available test cases. The code is improved until it passes the tests. This is also called test-driven development.

➔ Integration Test

3.3.1 Explanation of Terms

After the component test, the second test level in the V-model is integration testing. A precondition for integration testing is that the test objects subjected to it (i.e., components) have already been tested. Defects should, if possible, already have been corrected.

Integration

Developers, testers, or special integration teams then compose groups of these components to form larger structural units and subsystems. This connecting of components is called integration.

Integration test

Then the structural units and subsystems must be tested to make sure all components collaborate correctly. Thus, the goal of the integration test is to expose faults in the interfaces and in the interaction between integrated components.

Test basis

The test basis may be the software and system design or system architecture, or workflows through several interfaces and use cases. Why is integration testing necessary if each individual component has already been tested? The following example illustrates the problem.

Even if a complete component test had been executed earlier, such interface problems can still occur. Because of this, integration testing is necessary as a further test level. Its task is to find collaboration and interoperability problems and isolate their causes.

Integration testing in the large

As the example shows, interfaces to the system environment (i.e., external Systems) are also subject to integration and integration testing. When interfaces to external software systems are examined, we sometimes speak of ⊣system integration testing, higher-level integration testing, or integration testing in the large (integration of components is then integration test in the small, sometimes called ⊣component integration testing). System integration testing can be executed only after system testing. The development team has only one-half of such an external interface under its control. This constitutes a special risk. The other half of the interface is determined by an external system. It must be taken as it is, but it is subject to unexpected change. Passing a system integration test is no guarantee that the system will function flawlessly in the future.

Integration levels

Thus, there may be several integration levels for test objects of different sizes. Component integration tests will test the interfaces between internal components or between internal subsystems. System integration tests focus on testing interfaces between different systems and between hardware and software. For example, if business processes are implemented as a workflow through several interfacing systems and problems occur, it may be very expensive and challenging to find the defect in a special component or interface.

3.3.2 Test objects

Assembled components

Step-by-step, during integration, the different components are combined to form larger units. Ideally, there should be an integration test after each of these steps. Each subsystem may then be the basis for integrating further larger units. Such units (subsystems) may be test objects for the integration test later.

External systems or acquired components

In reality, a software system is seldom developed from scratch. Usually, an existing system is changed, extended, or linked to other systems (for example database systems, networks, new hardware). Furthermore, many system components are ⊣commercial off-the-shelf (COTS) software products .In component testing, such existing or standard components are probably not tested. In the integration test, however, these system components must be taken into account and their collaboration with other components must be examined. The most important test objects of integration testing are internal interfaces between components. Integration testing may also comprise configuration programs and configuration data. Finally, integration or system integration testing examines subsystems for correct database access and correct use of other infrastructure components.

3.3.3 The Test Environment

As with component testing, test drivers are needed in the integration test. They send test data to the test objects, and they receive and log the results. Because the test objects are assembled components that have no interfaces to the "outside" other than their constituting components, it is obvious and sensible to reuse the available test drivers for component testing.

Reuse of the test environment

If the component test was well organized, then some test drivers should be available. It could be one generic test driver for all components or at least test drivers that were designed with a common architecture and are compatible with each other. In this case, the testers can reuse these test drivers without much effort.

If a component test is poorly organized, there may be usable test drivers for only a few of the components. Their user interface may also be completely different, which will create trouble. During integration testing in a much later stage of the project, the tester will need to put a lot of effort into the creation, change, or repair of the test environment. This means that valuable time needed for test execution is lost.

Monitors are necessary

During integration testing, additional tools, called monitors, are required. Monitors are programs that read and log data traffic between components. Monitors for standard protocols (e.g., network protocols) are commercially available. Special monitors must be developed for the observation of project-specific component interfaces.

3.3.4 Test objectives

Wrong interface formats

The test objectives of the test level integration test are clear: to reveal inter- face problems as well as conflicts between integrated parts.

Problems can arise when attempting to integrate two single components. For example, their interface formats may not be compatible with each other because some files are missing or because the developers have split the system into completely different components than specified.

Typical faults in data exchange

The harder-to-find problems, however, are due to the execution of the connected program parts. These kinds of problems can only be found by dynamic testing. They are faults in the data exchange or in the communication between the components, as in the following examples:

■ A component transmits syntactically incorrect or no data. The receiving component cannot operate or crashes (functional fault in a component, incompatible interface formats, and protocol faults).
■ the communication works but the involved components interpret the received data differently (functional fault of a component, contradicting or misinterpreted specifications).
■ Data is transmitted correctly but at the wrong time, or it is late (timing problem), or the intervals between the transmissions are too short (throughput, load, or capacity problem).

Example: Integration problems in VSR
The following interface failures could occur during the VSR integration test. These can be attributed to the previously mentioned failure types:
■ In the GUI of the *DreamCar* subsystem, selected extra equipment items are not passed on to check_config(). Therefore, the price and the order data would be wrong.
■ In *DreamCar,* a certain code number (e.g., 442 for metallic blue) represents the color of the car. In the order management system running on the external mainframe, however, some code numbers are interpreted differently (there, for example, 442 may represent red). An order from the VSR, seen there as correct, would lead to delivery of the wrong product.

■ The mainframe computer confirms an order after checking whether delivery would be possible. In some cases, this examination takes so long that the VSR assumes a transmission failure and aborts the order. A customer who has carefully chosen her car would not be able to order it.

None of these failures can be found in the component test because the resulting failures occur only in the interaction between two software components. Nonfunctional tests may also be executed during integration testing, if attributes mentioned below are important or are considered at risk. These attributes may include reliability, performance, and capacity.

Can the component test be omitted?

Is it possible to do without the component test and execute all the test cases after integration is finished? Of course, this is possible, and in practice it is regretfully often done, but only at the risk of great disadvantages:

■ Most of the failures that will occur in a test designed like this are caused by functional faults within the individual components. An implicit component test is therefore carried out, but in an environment that is not suitable and that makes it harder to access the individual components
■ Because there is no suitable access to the individual component, some failures cannot be provoked and many faults, therefore, cannot be found.
■ If a failure occurs in the test, it can be difficult or impossible to locate its origin and to isolate its cause.

The cost of trying to save effort by cutting the component test is finding fewer of the existing faults and experiencing more difficulty in diagnosis. Combining a component test with a subsequent integration test is more effective and efficient.

3.3.5 Integration Strategies

In which order should the components be integrated in order to execute the necessary test work as efficiently—that is, as quickly and easily—as possible? Efficiency is the relation between the cost of testing (the cost of test personnel and tools, etc.) and the benefit of testing (number and severity of the problems found) in a certain test level. The test manager has to decide this and choose and implement an optimal integration strategy for the project.

Components are completed at different times

In practice, different software components are completed at different times, weeks or even months apart. No project manager or test manager can allow testers to sit around and do nothing while waiting until all the components are developed and they are ready to be integrated. An obvious ad hoc strategy to quickly solve this problem is to integrate the components in the order in which they are ready. This means that as soon as a component has passed the component test, it is checked to see if it fits with another already tested component or if it fits into a partially integrated subsystem. If so, both parts are integrated and the integration test between both of them is executed.

Example:
Integration Strategy in ***the VSR project***
In the VSR project, the central subsystem *ContractBase* turns out to be more complex than expected. Its completion is delayed for several weeks because the work on it costs much more than originally expected. To avoid losing even more time, the project manager decides to start the tests with the available components *DreamCar* and *NoRisk*. These do not have a common interface, but they exchange data through *ContractBase*. To calculate the price of the insurance, *NoRisk* needs to know which type of vehicle was chosen because this determines the price and other parameters of the insurance. As a temporary replacement for *ContractBase*, a stub is programmed. The stub receives simple car configuration data from *DreamCar*, then determines the vehicle type code from this data and passes it on about the customer. *NoRisk* calculates the insurance price from the data and shows it in a window so it can be checked. The price is also saved in a test log. The stub serves as a temporary replacement for the still missing subsystem *ContractBase*.

This example makes it clear that the earlier the integration test is started (in order to save time), the more effort it will take to program the stubs. The test manager has to choose an integration strategy in order to optimize both factors (time savings vs. cost for the testing environment).

Constraints for integration

Which strategy is optimal (the most timesaving and least costly strategy) depends on the individual circumstances in each project. The following items must be analyzed:

■ The **system architecture** determines how many and which components the entire system consists of and in which way they depend on each other.

■ The **project plan** determines at what time during the course of the project the parts of the system are developed and when they should be ready for testing. The test manager should be consulted when determining the order of implementation.

■ The **test plan** determines which aspects of the system shall be tested, how intensely, and on which test level this has to happen.

Discuss the integration strategy

The test manager, taking into account these general constraints, has to design an optimal integration strategy for the project. Because the integration strategy depends on delivery dates, the test manager should consult the project manager during project planning. The order of component implementation should be suitable for integration testing.

Generic strategies

When making plans, the test manager can follow these generic integration strategies:

■ Top-down integration

The test starts with the top-level component of the system that calls other components but is not called itself (except for a call from the operating system). Stubs replace all subordinate components. Successively, integration proceeds with lower-level components. The higher level that has already been tested serves as test driver.

• Advantage: Test drivers are not needed, or only simple ones are required, because the higher-level components that have already been tested serve as the main part of the test environment.

• Disadvantage: Stubs must replace lower-level components not yet integrated. This can be very costly.

■ Bottom-up integration

The test starts with the elementary system components that do not call further components, except for functions of the operating system. Larger subsystems are assembled from the tested components and then tested.

• Advantage: No stubs are needed.

• Disadvantage: Test drivers must simulate higher-level components.

■ Ad hoc integration

The components are being integrated in the (casual) order in which they are finished.

• Advantage: This saves time because every component is integrated as early as possible into its environment.

• Disadvantage: Stubs as well as test drivers are required.

■ Backbone integration

A skeleton or backbone is built and components are gradually integrated into it [Beizer 90].

• Advantage: Components can be integrated in any order.

• Disadvantage: A possibly labor-intensive skeleton or backbone is required.

Top-down and Bottom-up integration in their pure form can be applied only to program systems that are structured in a strictly hierarchical way; in reality, this rarely occurs. This is the reason a more or less individualized mix of the previously mentioned integration strategies11 might be chosen.

Avoid the big bang!

Any nonincremental integration—also called ⊣big bang integration—should be avoided. Big bang integration means waiting until all software elements are developed and then throwing everything together in one step. This typically happens due to the lack of an integration strategy. In the worst cases, even component testing is skipped. There are obvious disadvantages of this approach:

■ the time leading up to the big bang is lost time that could have been spent testing. As testing always suffers from lack of time, no time that could be used for testing should be wasted.
■ All the failures will occur at the same time. It will be difficult or impossible to get the system to run at all. It will be very difficult and time consuming to localize and correct defects.

➔ System Test

3.4.1 Explanation of Terms

After the integration test is completed, the third and next test level is the system test. System testing checks if the integrated product meets the specified requirements. Why is this still necessary after executing component and integration tests? The reasons for this are as follows:

Reasons for system test

■ In the lower test levels, the testing was done against technical specifications, i.e., from the technical perspective of the software producer. The system test, though, looks at the system from the perspective of the customer and the future user. The testers validate whether the requirements are completely and appropriately implemented.
■ Many functions and system characteristics result from the interaction of all system components; consequently, they are visible only when the entire system is present and can be observed and tested only there.

Example: VSR-System tests
The main purpose of the VSR-System is to make ordering a car as easy as possible.
While ordering a car, the user uses all the components of the VSR-System: the car is configured (*DreamCar*), financing and insurance are calculated (*Easy- Finance*, *NoRisk*), the order is transmitted to production (*JustInTime*), and the contracts are archived (*ContractBase*). The system fulfills its purpose only when all these system functions and all the components collaborate correctly. The system test determines whether this is the case.

The test basis includes all documents or information describing the test object on a system level. This may be system requirements, specifications, risk analyses if present, user manuals, etc.

3.4.2 Test Objects and Test Environment

After the completion of the integration test, the software system is complete. The system test tests the system as a whole in an environment as similar as possible to the intended ⊣production environment. Instead of test drivers and stubs, the hardware and software products that will be used later should be installed on the test platform (hardware, system software, device driver software, networks, external systems, etc.).

Figure 3-4 shows an example of the VSR-System test environment. The system test not only tests the system itself, it also checks system and user documentation, like system manuals, user manuals, training material, and so on. Testing configuration settings as well as optimizing the system configuration during load and performance testing (see section 3.7.2) must often be covered.

Data quality

It is getting more and more important to check the quality of data in systems that use a database or large amounts of data. This should be included in the system test. The data itself will then be new test objects. It must be assured that it is consistent, complete, and up-to-date. For example, if a system finds and displays bus connections, the station list and schedule data must be correct.

System test requires a separate test environment

One mistake is commonly made to save costs and effort: instead of the system being tested in a separate environment, the system test is executed in the customer's operational environment. This is detrimental for a couple of reasons:

■ During system testing, it is likely that failures will occur, resulting in damage to the customer's operational environment. This may lead to expensive system crashes and data loss in the production system.
■ The testers have only limited or no control over parameter settings and the configuration of the operational environment. The test conditions may change over time because the other systems in the customer's environment are running simultaneously with the test. The system tests that have been executed cannot be reproduced or can only be reproduced with difficulty.

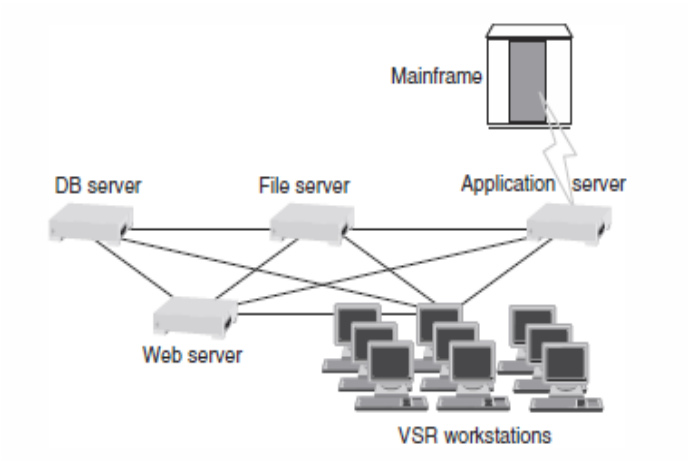


Figure 3–4 *Example of a system test environment*

System test effort is often underestimated

The effort of an adequate system test must not be underestimated, especially because of the complex test environment. States the experience that at the beginning of the system test, only half of the testing and quality control work has been done (especially when a client/server system is developed, as in the VSR-example).

3.4.3 Test Objectives

It is the goal of the system test to validate whether the complete system meets the specified functional and nonfunctional requirements and how well it does that. Failures from incorrect, incomplete, or inconsistent implementation of requirements should be detected. Even undocumented or forgotten requirements should be identified.

➔ Acceptance Test

All the test levels described thus far represent testing activities that are under the producer's responsibility. They are executed before the software is presented to the customer or the user.

Before installing and using the software in real life (especially for software developed individually for a customer), another last test level must be executed: the acceptance test. Here, the focus is on the customer's and user's perspective. The acceptance test may be the only test that the customers are actually involved in or that they can understand. The customer may even be responsible for this test!

→Acceptance tests may also be executed as a part of lower test levels or be distributed over several test levels:

■ A commercial-off-the-shelf product (COTS) can be checked for acceptance during its integration or installation.
■ Usability of a component can be acceptance tested during its component test.

■ Acceptance of new functionality can be checked on prototypes before system testing.

There are four typical forms of acceptance testing:
■ Contract acceptance testing
■ User acceptance testing
■ Operational acceptance testing
■ Field testing (alpha and beta testing)

How much acceptance testing?

How much acceptance testing should be done is dependent on the product risk. This may be very different. For customer-specific systems, the risk is high and a comprehensive acceptance test is necessary. At the other extreme, if a piece of standard software is introduced, it may be sufficient to install the package and test a few representative usage scenarios. If the system interfaces with other systems, collaboration of the systems through these interfaces must be tested.

Test basis

The test basis for acceptance testing can be any document describing the system from the user or customer viewpoint, such as, for example, user or system requirements, use cases, business processes, risk analyses, user process descriptions, forms, reports, and laws and regulations as well as descriptions of maintenance and system administration rules and processes.

3.5.1 Contract Acceptance Testing

If customer-specific software was developed, the customer will perform contract acceptance testing (in cooperation with the vendor). Based on the results, the customer considers whether the software system is free of (major) deficiencies and whether the service defined by the development contract has been accomplished and is acceptable. In case of internal software development, this can be a more or less formal contract between the user department and the IT department of the same enterprise.

Acceptance criteria

The test criteria are the acceptance criteria determined in the development contract. Therefore, these criteria must be stated as unambiguously as possible. Additionally, conformance to any governmental, legal, or safety regulations must be addressed here.

In practice, the software producer will have checked these criteria within his own system test. For the acceptance test, it is then enough to rerun the test cases that the contract requires as relevant for acceptance, demonstrating to the customer that the acceptance criteria of the contract have been met.

Because the supplier may have misunderstood the acceptance criteria, it is very important that the acceptance test cases are designed by or at least thoroughly reviewed by the customer.

Customer (site) acceptance test

In contrast to system testing, which takes place in the producer environment, acceptance testing is run in the customer's actual operational environment. Due to these different testing environments, a test case that worked correctly during the system test may now suddenly fail. The acceptance test also checks the delivery and installation procedures. The acceptance environment should be as similar as possible to the later operational environment. A test in the operational environment itself should be avoided to minimize the risk of damage to other software systems used in production.

The same techniques used for test case design in system testing can be used to develop acceptance test cases. For administrative IT systems, business transactions for typical business periods (like a billing period) should be considered.

3.5.2 Testing for User Acceptance

Another aspect concerning acceptance as the last phase of validation is the test for user acceptance. Such a test is especially recommended if the customer and the user are different.

Example: Different user groups
In the VSR example, the responsible customer is a car manufacturer. But the car manufacturer's shops will use the system. Employees and customers who want to purchase cars will be the system's end users. In addition, some clerks in the company's headquarter will work with the system, e.g., to update price lists in the system.

Get acceptance of every user group

Different user groups usually have completely different expectations of a new system. Users may reject a system because they find it "awkward" to use, which can have a negative impact on the introduction of the system. This may happen even if the system is completely OK from a functional point of view. Thus, it is necessary to organize a user acceptance test for each user group. The customer usually organizes these tests, selecting test cases based on business processes and typical usage scenarios.

Present prototypes to the users early

If major user acceptance problems are detected during acceptance testing, it is often too late to implement more than cosmetic countermeasures. To prevent such disasters, it is advisable to let a number of representatives from the group of future users examine prototypes of the system early.

3.5.3 Operational (Acceptance) Testing

Operational (acceptance) testing assures the acceptance of the system by the system administrators. It may include testing of backup/restore cycles (including restoration of copied data), disaster recovery, user management, and checks of security vulnerabilities.

3.5.4 Field Testing

If the software is supposed to run in many different operational environments, it is very expensive or even impossible for the software producer to create a test environment for each of them during system testing. In such cases, the software producer may choose to execute a ―field test after the system test. The objective of the field test is to identify influences from users' environments that are not entirely known or specified and to eliminate them if necessary. If the system is intended for the general market (a COTS system), this test is especially recommended.

Testing done by representative customers

For this purpose, the producer delivers stable prerelease versions of the software to preselected customers who adequately represent the market for this software or whose operational environments are appropriately similar to possible environments for the software.

These customers then either run test scenarios prescribed by the producer or run the product on a trial basis under realistic conditions. They give feedback to the producer about the problems they encountered along with general comments and impressions about the new product. The producer can then make the specific adjustments.

Alpha and beta testing

Such testing of preliminary versions by representative customers is also called ―alpha testing or ―beta testing. Alpha tests are carried out at the producer's location, while beta tests are carried out at the customer's site.

A field test should not replace an internal system test run by the producer (even if some producers do exactly this). Only when the system test has proven that the software is stable enough should the new product be given to potential customers for a field test.

Dogfood test

A new term in software testing is *dogfood test*. It refers to a kind of internal field testing where the product is distributed to and used by internal users in the company that developed

the software. The idea is that "if you make dogfood, try it yourself first." Large suppliers of software like Microsoft and Google advocate this approach before beta testing.

➔ Generic types of Testing-

✓ Functional

Functional testing includes all kind of tests that verify a system's input/output behavior. To design functional test cases, the black box testing methods discussed in section 5.1 are used, and the test bases are the functional requirements.

Functional requirements

Functional requirements →specify the behavior of the system; they describe what the system must be able to do. Implementation of these requirements is a precondition for the system to be applicable at all.
Characteristics of functionality, according to [ISO 9126], are suitability, accuracy, interoperability, and security.

Requirements definition

When a project is run using the V-model, the requirements are collected during the phase called "requirements definition" and documented in a requirements management system (see section 7.1.1). Text-based requirements specifications are still in use as well. Templates for this document are available in [IEEE 830].

The following text shows a part of the requirements paper concerning price calculation for the system VSR .

Example:
Requirements of the VSR-System
R 100: The user can choose a vehicle model from the current model list for configuration.
R 101: For a chosen model, the deliverable extra equipment items are indicated.
The user can choose the desired individual equipment from this list.
R 102: The total price of the chosen configuration is continuously calculated
from current price lists and displayed.

Requirements-based testing

Requirements-based testing uses the final requirements as the basis for testing. For each requirement, at least one test case is designed and documented in the test specification. The test specification is then reviewed. The testing of requirement 102 in the preceding example could look like the following example.

Example:
Requirements-based testing
T 102.1: A vehicle model is chosen; its base price according to the sales manual is displayed.
T 102.2: A special equipment item is selected; the price of this accessory is added.
T 102.3: A special equipment item is deselected; the price falls accordingly.
T 102.4: Three special equipment items are selected; the discount comes into effect as defined in the specification.

Usually, more than one test case is needed to test a functional requirement. Requirement 102 in the example contains several rules for different price calculations. These must be covered by a set of test cases (102.1–102.4 in the preceding example). Using black box test methods (e.g., →equivalence partitioning), these test cases can be further refined and extended if desired. The decisive fact is if the defined test cases (or a minimal subset of them) have run without failure, the appropriate functionality is considered validated.

Requirements-based functional testing as shown is mainly used in system testing and other higher levels of testing. If a software system's purpose is to automate or support a certain business process for the customer, business-process-based testing or use-case-based testing are other similar suitable testing methods.

Example:
Testing based on business process
From the dealer's point of view, VSR supports him in the sales process. The process can, for example, look like this:
■ The customer selects a type of vehicle he is interested in from the available models.
■ The customer gets the information about the type of extra equipment and prices and selects the desired car.
■ The dealer suggests alternative ways of financing the car.
■ The customer decides and signs the contract.

A business process analysis (which is usually elaborated as part of the requirements analysis) shows which business processes are relevant and how often and in which context they appear. It also shows which persons, enterprises, and external systems are involved. Test scenarios simulating typical business processes are constructed based on this analysis. The test scenarios are prioritized using the frequency and the relevance of the particular business processes.

Functional Vs. Non-Functional Testing

Parameters	**Functional**	**Non-functional testing**
Execution	It is performed before non-functional testing.	It is performed after the functional testing.
Focus area	It is based on customer's requirements.	It focuses on customer's expectation.
Requirement	It is easy to define functional requirements.	It is difficult to define the requirements for non-functional testing.
Usage	Helps to validate the behavior of the application.	Helps to validate the performance of the application.
Objective	Carried out to validate software actions.	It is done to validate the performance of the software.
Requirements	Functional testing is carried out using the functional specification.	This kind of testing is carried out by performance specifications
Manual testing	Functional testing is easy to execute by manual testing.	It's very hard to perform non-functional testing manually.
Functionality	It describes what the product does.	It describes how the product works.
Example Test Case	Check login functionality.	The dashboard should load in 2 seconds.
Testing Types	Examples of Functional Testing Types • Unit testing • Smoke testing • User Acceptance • Integration Testing • Regression testing • Localization • Globalization • Interoperability	Examples of Non-functional Testing Types • Performance Testing • Volume Testing • Scalability • Usability Testing • Load Testing • Stress Testing • Compliance Testing • Portability Testing • Disaster Recover Testing

Requirements-based testing focuses on single system functions (e.g., the transmission of a purchase order). Business-process-based testing, however, focuses on the whole process consisting of many steps (e.g., the sales conversation, consisting of configuring a car, agreeing on the purchase contract, and the transmission of the purchase order). This means a sequence of several tests.

Of course, for the users of the *VirtualShowRoom* system, it is not enough to see if they can choose and then buy a car. More important for ultimate acceptance is often how easily they can use the system. This depends on how easy it is to work with the system, if it reacts quickly enough, and if it returns easily understood information. Therefore, along with the functional criteria, the nonfunctional criteria must also be checked and tested.

- ✓ Non Functional

Nonfunctional requirements do not describe the functions; they describe the attributes of the functional behavior or the attributes of the system as a whole, i.e., "how well" or with what quality the (partial) system should work. Implementation of such requirements has a great influence on customer and user satisfaction and how much they enjoy using the product. Characteristics of these requirements are, according to [ISO 9126], reliability, usability, and efficiency. Indirectly, the ability of the system to be changed and to be installed in new environments also has an influence on customer satisfaction. The faster and the easier a system can be adapted to changed requirements, the more satisfied the customer and the user will be. These two characteristics are also important for the supplier, because they help to reduce maintenance costs.

According to [Myers 79], the following nonfunctional system characteristics should be considered in the tests (usually in system testing):

■ **Load test:** Measuring of the system behavior for increasing system loads (e.g., the number of users that work simultaneously, number of transactions)

■ **Performance test:** Measuring the processing speed and response time for particular use cases, usually dependent on increasing load

■ **Volume test:** Observation of the system behavior dependent on the amount of data (e.g., processing of very large files)

■ **Stress test:** Observation of the system behavior when the system is overloaded

■ **Testing of security** against unauthorized access to the system or data, denial of service attacks, etc.

■ **Stability** or **reliability test:** Performed during permanent operation (e.g., mean time between failures or failure rate with a given user profile)

■ **Robustness test:** Measuring the system's response to operating errors, bad programming, hardware failure, etc. as well as examination of exception handling and recovery

■ **testing of compatibility and data conversion:** Examination of compatibility with existing systems, import/export of data, etc.

■ **testing of different configurations of the system:** For example, different versions of the operating system, user interface language, hardware platform, etc. (→back-to-back testing)

■ **Usability test:** Examination of the ease of learning the system, ease and efficiency of operation, understandability of the system outputs, etc., always with respect to the needs of a specific group of users ([ISO 9241], [ISO 9126])

■ **Checking of the documentation:** For compliance with system behavior (e.g., user manual and GUI)

■ **Checking maintainability:** Assessing the understandability of the system documentation and whether it is up-to-date; checking if the system has a modular structure; etc.

A major problem in testing nonfunctional requirements is the often imprecise and incomplete expression of these requirements. Expressions like "the system should be easy to operate" and "the system should be fast" are not testable in this form.

Furthermore, many nonfunctional requirements are so fundamental that nobody really thinks about mentioning them in the requirements paper (presumed matters of fact).even such implicit characteristics must be validated because they may be relevant.

Example: Presumed requirements
The VSR-System is designed for use on a market-leading operating system. It is obvious that the recommended or usual user interface conventions are followed for the "look and feel" of the VSR GUI. The DreamCar GUI violates these conventions in several aspects. Even if no particular requirement is specified, such deviations from "matter of fact requirements" can and must be seen as faults or defects.

Excursion:
Testing nonfunctional requirements

In order to test nonfunctional characteristics, it makes sense to reuse existing functional tests. The nonfunctional tests are somehow "piggybacking" on the functional tests. Most nonfunctional tests are black box tests. An elegant general testing approach could look like this:

Scenarios that represent a cross section of the functionality of the entire system are selected from the functional tests. The nonfunctional property must be observable in the corresponding test scenario. When the test scenario is executed, the nonfunctional characteristic is measured. If the resulting value is inside a given limit, the test is considered "passed." The functional test practically serves as a vehicle for determining the nonfunctional system characteristics.

- ✓ Testing software structure

Structural techniques (→structure-based testing, white box testing) use information about the test object's internal code structure or architecture. Typically, the control flow in a component, the call hierarchy of procedures, or the menu structure is analyzed. Abstract models of the software may also be used. The objective is to design and run enough test cases to, if possible, completely cover all structural items. In order to do this, useful (and enough) test cases must be developed.

Structural techniques are most used in component and integration testing, but they can also be applied at higher levels of testing, typically as extra tests (for example, to cover menu structures).

- ✓ Regression Testing

When changes are implemented, parts of the existing software are changed or new modules are added. This happens when correcting faults and performing other maintenance activities. Tests must show that earlier faults are really repaired (→retesting). Additionally, there is the risk of unwanted side effects. Repeating other tests in order to find them is called regression testing.

Regression test

A regression test is a new test of a previously tested program following modification to ensure that faults have not been introduced or uncovered as a result of the changes made (uncovering masked defects).

Thus, regression testing may be performed at all test levels and applies to functional, nonfunctional, and →structural test. Test cases to be used in regression testing must be well documented and reusable. Therefore, they are strong candidates for →test automation.
The question is how extensive a regression test has to be. There are the following possibilities:

How much retest and regression test

1. Rerunning of all the tests that have detected failures whose reasons (the defects) have been fixed in the new software release (defect retest, confirmation testing)
2. Testing of all program parts that were changed or corrected (testing of altered functionality)
3. Testing of all program parts or elements that were newly integrated (testing of new functionality)

4. testing of the whole system (complete regression test)

A bare retest (1) as well as tests that execute only the area of modifications (2 and 3) are not enough because in software systems, simple local code changes can create side effects in any other, arbitrarily distant, system parts.

Changes can have unexpected side effects

If the test covers only altered or new code parts, it neglects the consequences these alterations can have on unaltered parts. The trouble with software is its complexity. With reasonable cost, it can only be roughly estimated where such unwanted consequences can occur. This is particularly difficult for changes in systems with insufficient documentation or missing requirements, which, unfortunately, is often the case in old systems.

Full regression test

In addition to retesting the corrected faults and testing changed functions, all existing test cases should be repeated. Only in this case would the test be as safe as the testing done with the original program version. Such a complete regression test would also be necessary if the system environment has been changed because this could have an effect on every part of the system.

In practice, a complete regression test is usually too time consuming and expensive. Therefore, we are looking for criteria that can help to choose which old test cases can be omitted without losing too much information. As always, in testing this means balancing risk and cost. The following test selection strategies are often used:

Selection of regression test cases

■ repeating only the high-priority tests according to the test plan
■ In the functional test, omitting certain variations (special cases)
■ Restricting the tests to certain configurations only (e.g., testing of the English product version only, testing of only one operating system version)
■ Restricting the test to certain subsystems or test levels

Excursion
Generally, the rules listed here refer to the system test. On the lower test levels, regression test criteria can also be based on design or architecture documents (e.g., class hierarchy) or white box information. Further information can be found in [Kung 95], [Rothermel 94], [Winter 98], and [Binder 99]. There, the authors not only describe special problems in regression testing object-oriented programs, they also describe the general principles of regression testing in detail.

3. Static Testing

Static investigations like reviews and tool-supported analysis of code and documents can be used very successfully for improving quality. This chapter presents the possibilities and techniques.

An often-underestimated examination method is the so-called static test, often named static analysis. Opposite to dynamic testing, the test object is not provided with test data and executed but rather analyzed. This can be done using one or more persons for an intensive investigation or through the use of tools. Such an investigation can be used for all documents relevant for software development and maintenance. Tool supported static analysis is only possible for documents with a formal structure.

The goal of examination is to find defects and deviations from the existing specifications, standards to comply with, or even the project plan. An additional benefit of the results of these examinations is optimizing the development process. The basic idea is defect prevention: defects and deviations should be recognized as early as possible before they have any effect in the further development process where they would result in expensive rework.

➔ Structured Group Examinations –

✓ Reviews

Review is a common generic term for all the different static analysis techniques people perform as well as the term for a specific document examination technique.

Another term, often used with the same meaning, is ⊣nspection. However, *inspection* is usually defined as a special, formal review using data collection and special rules [Fagan 76], [IEEE 1028], [Gilb 96]. All documents can be subjected to a review or an inspection, such as, for example, contracts, requirements definitions, design specifications, program code, test plans, and manuals. Often, reviews provide the only possibility to check the semantics of a document. Reviews rely on the colleagues of the author to provide mutual feedback. Because of this, they are also called peer reviews.

A means for quality assurance

Reviews are an efficient means to assure the quality of the examined documents. Ideally, they should be performed as soon as possible after a document is completed to find mistakes and inconsistencies early. The verifying examinations at the end of a phase in the general V-model normally use reviews (so-called phase exit reviews). Eliminating defects and inconsistencies leads to improved document quality and positively influences the whole development process because development is continued with documents that have fewer or even no defects.

Positive effects

In addition to defect reduction, reviews have further positive effects:

■ Cheaper defect elimination. If defects are found and eliminated early, productivity in development increases because fewer resources are needed for defect identification and elimination later. These resources can instead be used for development.
■ Shortened development time.
■ If defects are recognized and corrected early, costs and time needed for executing the dynamic tests (see chapter 5) decrease because there are fewer defects in the test object.
■ Because of the smaller number of defects, cost reduction can be expected during the whole product life. For example, a review may detect and clarify inconsistent and imprecise customer requests in the requirements. Foreseeable change requests after installation of the software system can thus be avoided.
■ During operation of the system, a reduced failure rate can be expected.

■ As the examinations are done using a team of people, reviews lead to mutual learning. People improve their working methods, and reviews will thus lead to enhanced quality of later products.
■ Because several persons are involved in a review, a clear and understandable description of the facts is required. Often the necessity to formulate a clear document lets the author find forgotten issues.
■ The whole team feels responsible for the quality of the examined object. The group will gain a common understanding of it.

Potential problem

The following problem can arise: In a badly moderated review session, the author may get into a psychologically difficult situation, feeling that he as a person and not the document is subject to critical scrutiny. Motivation to subject documents to a review will thus be destroyed. Concretely expressing the review objective, which is improving the document, may be helpful. One book extensively discusses how to solve problems with reviews.

Reviews costs and savings

The costs caused by reviews are estimated to be 10–15% of the development budget. The costs include the activities of the review process itself, analyzing the review results, and the effort put toward implementing them for process improvement. Savings are estimated to be about 14–25%. The extra effort for the reviews themselves is included in this calculation.

If reviews are systematically used and efficiently run, more than 70% of the defects in a document can be found and repaired before they are unknowingly inherited by the next work steps. Considering that the costs for defect removal substantially increase in later development steps, it is plausible that defect cost in development is reduced by 75% and more.

Important success factors

The following factors are decisive for success when using reviews (as suggested by [IEEE 1028]):
■ Every review has a clear goal, which is formulated beforehand.
■ The "right" people are chosen as review participants based on the review objective as well as on their subject knowledge and skills.

4.1.3 The General Process

The term *review* describes a whole group of static examinations. The process underlying all reviews is briefly described here in accordance with the IEEE Standard for Software Reviews.

A review requires six work steps: planning, kick-off, individual preparation, review meeting, rework, and follow-up.

1st. Planning

Reviews need planning

Early, during overall planning, management must decide which documents in the software development process shall be subject to which review technique. The estimated effort must be included in the project plans. Several analyses show optimal checking time for reviewing documents and code. During planning of the individual review, the review leader selects technically competent staff and assembles a review team. In cooperation with the author of the document to be reviewed, she makes sure that the document is in a reviewable state, i.e., it is complete enough and reasonably finished. In formal reviews, entry criteria (and the corresponding exit criteria) may be set. A review should continue only after any available entry criteria have been checked.

Different perspectives increase the effect

A review is, in most cases, more successful when the examined document is read from different viewpoints or when each person checks only particular aspects. The viewpoints or aspects to be used should be determined during review planning. A review might not involve the whole document. Parts of the document in which defects constitute a high risk could be

selected. A document may also be sampled only to make a conclusion about the general quality of the document.

If a kick-off meeting is necessary, the place and time must be agreed upon.

2nd. **Kick-Off**

The kick-off (or overview) serves to provide those involved in the review with all of the necessary information. This can happen through a written invitation or a meeting when the review team is organized. The purpose is sharing information about the document to be reviewed (*the review object*) and the significance and the objective of the planned review. If the people involved are not familiar with the domain or application area of the review object, then a short introduction to the material may be arranged, and a description of how it fits into the application or environment may be provided.

Higher-level documents are necessary

In addition to the review object, those involved must have access to other documents. These include the documents that help to decide if a particular statement is wrong or correct. The review is done against these documents (e.g., requirements specification, design, guidelines, or standards). Such documents are also called base documents or baselines. Furthermore, review criteria (for example, checklists) are very useful for supporting a structured process.

For more formal reviews, the entry criteria might be checked. If entry criteria are not met, the review should be canceled, saving the organization time that would otherwise be wasted reviewing material that may be "immature," i.e., not good enough.

3rd. **Individual Preparation**

Intensive study of the review object

The members of the review team must prepare individually for the review meeting. A successful review meeting is only possible with adequate preparation.

The reviewers intensively study the review object and check it against the documents given as a basis for it as well as against their own experience. They note deficiencies (even any potential defects), questions, or comments.

4th. **Review Meeting**

A review leader or →moderator leads the review meeting. Moderator and participants should behave diplomatically (not be aggressive with each other) and contribute to the review in the best possible way.

The review leader must ensure that all experts will be able to express their opinion knowing that the product will be evaluated and not the author. Conflicts should be prevented. If this is not possible, a solution for the situation should be found.

Usually, the review meeting has a fixed time limit. The objective is to decide if the review object has met the requirements and complies with the standards and to find defects. The result is a recommendation to accept, repair, or rewrite the document. All the reviewers should agree upon the findings and the overall result.

Rules for review meetings

Here are some general rules for a review meeting:

1. The review meeting is limited to two hours. If necessary, another meeting is called, but it should not take place on the same day.
2. The moderator has the right to cancel or stop a meeting if one or more experts (reviewers) don't appear or if they are not sufficiently prepared.
3. The document (the review object) is subject to discussion, not the author:
   - The reviewers have to watch their expressions and their way of expressing themselves.
   - The author should not defend himself or the document. (That means, the author should not be attacked or forced into a defensive position. However, justification or explanation of the author's decisions is often seen as legitimate and helpful.)
4. The moderator should not also be a reviewer at the same time.

5. General style questions (outside the guidelines) shall not be discussed.
6. Solutions and discussing them isn't a task of the review team.
7. Every reviewer must have the opportunity to adequately present his or her issues.
8. The protocol must describe the consensus of the reviewers.
9. Issues must not be written as commands to the author (additional concrete suggestions for improvement or correction are sometimes considered useful and sensible for quality improvement).
10. The issues should be weighted2 as follows:
    • Critical defect (the review object is not suitable for its purpose, the defect must be corrected before the object is approved)
    • Major defect (the usability of the review object is affected, the defect must be corrected before the approval)
    • Minor defect (small deviation, for example, spelling error or bad expression, hardly affects the usage)
    • Good (flawless, this area should not be changed during rework).
11. The review team shall make a recommendation about the acceptance of the review object (see follow-up):
    • Accept (without changes)
    • Accept (with changes, no further review)
    • Do not accept (further review or other checking measures are necessary)
12. Finally, all the session participants should sign the protocol

Protocol and summary of results

The protocol contains a list of the issues/findings that were discussed during the meeting. An additional review summary report should collect all important data about the review itself, i.e., the review object, the people involved, their roles, a short summary of the most important issues, and the result of the review with the recommendation of the reviewers. In a more formal review, the fulfillment of formal exit criteria may be documented. If there was no physical meeting and, for example, electronic communication was used instead, there should definitely be a protocol.

5th. **Rework**

The manager decides whether to follow the recommendation or do something else. A different decision is, however, the sole responsibility of the manager. Usually, the author will eliminate the defects on the basis of the review results and rework the document. More formal reviews additionally require updating the defect status of every single found defect.

6th. **Follow-Up**

The proper correction of defects must be followed up, usually by the manager, moderator, or someone especially assigned this responsibility.

Second review

If the result of the first review was not acceptable, another review should be scheduled. The process described here can be rerun, but usually it is done in an abbreviated manner, checking only changed areas.

The review meetings and their results should then be thoroughly evaluated to improve the review process, to adapt the used guidelines and checklists to the specific conditions, and to keep them up-to-date. To achieve this, it is necessary to collect and evaluate measurement data.

Find and fix deficiencies in the software development process

Recurring, or frequently occurring, defect types point to deficiencies in the software development process or lack of technical knowledge of the people involved. Necessary improvements of the development process should be planned and implemented. Such defect types should be included in the checklists. Training must compensate for lack of technical knowledge.

For more formal reviews, the final activity is checking the exit criteria. If they are met, the review is finished. Otherwise, it must be determined whether rework can be done or if the whole review was unsuccessful.

Roles and Responsibilities

The description of the general approach included some information on roles and responsibilities. This section presents the people involved and their tasks.

Manager

The manager selects the objects to be reviewed, assigns the necessary resources, and selects the review team.

Representatives of the management level should not participate in review meetings because management might evaluate the qualifications of the author and not the document. This would inhibit a free discussion among the review participants. Another reason is that the manager often lacks the necessary detailed understanding of technical documents. In a review, the technical content is checked, and thus the manager would not be able to add valuable comments. Management reviews of project plans and the like are a different thing. In this case, knowledge of management principles is necessary.

Moderator

The moderator is responsible for executing the review. Planning, preparation, execution, rework, and follow-up should be done in such a way that the review objectives are achieved.

The moderator is responsible for collecting review data and issuing the review report.

This role is crucial for the success of the review. First and foremost, a moderator must be a good meeting leader, leading the meeting efficiently and in a diplomatic way. A moderator must be able to stop unnecessary discussions without offending the participants, to mediate when there are conflicting points of view, and be able to see "between the lines." A moderator must be neutral and must not state his own opinion about the review object.

Author

The author is the creator of the document that is the subject of a review. If several people have been involved in the creation, one person should be appointed to be responsible; this person assumes the role of the author. The author is responsible for the review object meeting its review entry criteria (i.e., that the document is reviewable) and for performing any rework required for meeting the review exit criteria.

It is important that the author does not interpret the issues raised on the document as personal criticism. The author must understand that a review is done only to help improve the quality of the product.

Reviewer

The reviewers, sometimes also called inspectors, are several (usually a maximum of five) technical experts that participate in the review meeting after necessary individual preparation.

They identify and describe problems in the review object. They should represent different viewpoints (for example, sponsor, requirements, design, code, safety, test). Only those viewpoints pertinent to the review of the product should be considered.

Some reviewers should be assigned specific review topics to ensure effective coverage. For example, one reviewer might focus on conformance with a specific standard, another on syntax. The manager should assign these roles when planning the review.

The reviewers should also label the good parts in the document. Insufficient or deficient parts of the review object must be labeled accordingly, and the deficiencies must be documented for the author in such a way that they can be corrected.

Recorder

The recorder (or scribe) shall document the issues (problems, action items, decisions, and recommendations) found by the review team. The recorder must be able to record in a short and precise way, correctly capturing the essence of the discussion. This may not be easy because contributions are often not clearly or well expressed. Pragmatic reasons may make it

meaningful to let the author be recorder. The author knows exactly how precisely and how detailed the contributions of the reviewers need to be recorded in order to have enough information for rework.

Possible difficulties

Reasons for less successful reviews

Reviews may fail to achieve their objectives due to several causes:

■ The required persons are not available or do not have the required qualifications or technical skills. This may be solved by training or by using qualified staff from consulting companies. This is especially true for the moderator, because he must have more psychological than technical skills.

■ Inaccurate estimates during resource planning by management may result in time pressure, which then causes unsatisfactory review results. Sometimes, a less costly review type can bring relief.

■ If reviews fail due to lack of preparation, this is mostly because the wrong reviewers were chosen. If a reviewer does not realize the importance of the review and its great effect on quality improvement, then figures must be shown that prove the productive benefit of reviews. Other reasons for review failure may be lack of time and lack of motivation.

■ A review can also fail because of missing or insufficient documentation. Prior to the review, it must be verified that all the needed documents exist and that they are sufficiently detailed. Only when this is the case should a review be performed.

■ The review process cannot be successful if there is lack of management support because the necessary resources will not be provided and the results will not be used for process improvement. Unfortunately, this is often the case.

Types of Reviews

Two main groups of reviews can be distinguished depending on the review object to be examined:

■ Reviews pertaining to products or intermediate products that have been created during the development process

■ Reviews that analyze the project itself or the development process

Excursion

Reviews in the second group are called →management reviews or project reviews. Their objective is to analyze the project itself or the development process. For example, such a review determines if plans and rules are followed, if the necessary work tasks are executed, or the effectiveness of process improvements or changes.

The project as a whole and determining its current state are the objects of such a review. The state of the project is evaluated with respect to technical, economic, time, and management aspects.

Management reviews are often performed when reaching a milestone in the project, when completing a main phase in the software development process, or as a "postmortem" analysis to learn from the finished project.

In the following sections, the first group of reviews is described in more detail. We can distinguish between the following review types: →walkthrough, inspection, →technical review, and →informal review. In the descriptions, the focus is on the main differences between the particular review type and the basic review process.

1st. **Walkthrough**

A walkthrough4 is a manual, informal review method with the purpose of finding defects, ambiguities, and problems in written documents. The author presents the document to the reviewers in a review meeting. Educating an audience regarding a software product is mentioned in [IEEE 1028] as a further purpose of walkthroughs. Further objectives of walkthroughs are to improve the product, to discuss alternative implementations, and to evaluate conformance to standards and specifications.

The main emphasis of a walkthrough is the review meeting (without a time limit). There is less focus on preparation compared to the other types of reviews; it can even be omitted sometimes.

Discussion of typical usage situations

In most cases, typical usage situations, also called scenarios, will be discussed. Test cases may even be "played through." The reviewers try to find possible errors and defects by spontaneously asking questions.

Suitable for small development teams

The technique is useful for small teams of up to five persons. It does not require a lot of resources because preparation and follow-up are minor or sometimes not even required. The walkthrough is useful for checking "noncritical" documents.

The author chairs the meeting and therefore has a great amount of influence. This can have a detrimental effect on the result if the author impedes an intensive discussion of the critical parts of the review object. The author is responsible for follow-up; there is no more checking. Before the meeting the reviewers prepare, the results are written in a protocol, and someone other than the author records the findings. In practice there is a wide variation from informal to formal walkthroughs.

Objectives

The main objectives of a walkthrough are mutual learning, development of an understanding of the review object, and error detection.

2nd. **Inspection**

Formal process

The inspection is the most formal review. It follows a formal, prescribed process. Every person involved, usually people who work directly with the author, has a defined role. Rules define the process. The reviewers use checklists containing criteria for checking the different aspects.

The goals are finding unclear items and possible defects, measuring review object quality, and improving the quality of the inspection process and the development process. The concrete objectives of each individual inspection are determined during planning. The inspectors (reviewers) prepare for only a specific number of aspects that will be examined. Before the inspection begins, the inspection object is formally checked with respect to entry criteria and reviewability. The inspectors prepare themselves using procedures or standards and checklists.

Traditionally, this method of reviewing has been called design inspection or code or software inspection. The name points to the documents that are subject to the inspection. However, inspections can be used for any document in which formal evaluation criteria exist.

Inspection meeting

A moderator leads the meeting. The inspection meeting follows this agenda:

■ The moderator first presents the participants and their roles as well as a short introduction to the topic of the inspection object.

■ The moderator asks the participants if they are adequately prepared. In addition, the moderator might ask how much time the reviewer used to prepare and how many and how severe were the issues found.

■ The group may review the checklists chosen for the inspection in order to make sure everyone is well prepared for the meeting.

■ Issues of a general nature concerning the whole inspection object are discussed first and written into the protocol.

■ A reviewer presents the contents of the inspection object quickly and logically. If it's considered useful, passages can also be read aloud.

■ The reviewers ask questions during this procedure, and the selected aspects of the inspection are thoroughly discussed. The author answers questions. The moderator makes sure that a list

of issues is written. If author and reviewer disagree about an issue, a decision is made at the end of the meeting.
■ The moderator must intervene if the discussion is getting out of control. The moderator also makes sure the meeting covers all aspects to be evaluated as well as the whole document. The moderator makes sure the recorder writes down all the issues and ambiguities that are detected.
■ At the end of the meeting, all recorded items are reviewed for completeness.
■ Discussions are conducted to resolve disagreements, for example, whether or not something can be classified a defect. If no resolution is reached, this is written in the protocol. There should be no discussion on how to solve the issues. Any discussion should be limited in time.
■ Finally, the reviewers judge the inspection object as a whole.
■ They decide if it must be reworked or not. In inspections, follow-up and reinspection are formally regulated.

Additional assessment of the development and inspection process

In an inspection, data are also collected for general quality assessment of the development process and the inspection process. Therefore, an inspection also serves to optimize the development process, in addition to assessing the inspected documents. The collected data are analyzed in order To find causes for weaknesses in the development process. After process improvement, comparing the collected data before the change to the current data checks the improvement effect.

Objective

The main objective of inspection is defect detection or, more precisely, the detection of defects causes and defects.

3rd. Technical Review

Does the review object fulfill its purpose?

In a technical review, the focus is compliance of the document with the specification, fitness for its intended purpose, and compliance to standards during preparation, the reviewers check the review object with respect to the specified review criteria.

Technical experts as reviewers

The reviewers must be technically qualified. Some of them should not be project participants in order to avoid "project blindness." Management does not participate. Basis for the review is only the "official" specification and the specified criteria for the review. The reviewers write down their comments and pass them to the moderator before the review meeting. The moderator (who ideally is properly trained) sorts these findings based on their presumed importance. During the review meeting, only selected remarks are discussed.

High preparation effort

Most of the effort is in preparation. The author does not normally attend the meeting. During the meeting, the recorder notes all the issues and prepares the final documentation of the results.
The review result must be approved unanimously by all involved reviewers and signed by everyone. Disagreement should be noted in the protocol. It is not the job of the review participants to decide on the consequences of the result; that is the responsibility of management. If the review is highly formalized, entry and exit criteria of the individual review steps may also be defined. In practice, very different versions of the technical review are found, from a very informal to a strictly defined, formal process.

Objective

Discussion is expressly requested during a technical review. Alternative approaches should be considered and decisions made. The specialists may solve the technical issues. The conformity of the review object with its specifications and applicable standards can be assessed. Technical reviews can, of course, reveal errors and defects.

4th. **Informal Review**

The informal review is a light version of a review. However, it more or less follows the general procedure for reviews in a simplified way. In most cases, the author initiates an informal review. Planning is restricted to choosing reviewers and asking them to deliver their remarks at a certain point in time. Often, there is no meeting or exchange of the findings. In such cases, the review is just a simple author-reader-cycle. The informal review is a kind of cross reading by one or more colleagues. The results need not be explicitly documented; a list of remarks or the revised document is in most cases enough. Pair programming, buddy testing, code swapping, and the like are types of informal review. The informal review is very common and highly accepted due to the minimal effort required. An informal review involves relatively little effort and low costs.

Objective

Discussion and exchange of information among colleagues are welcome "side effects" of the process.

Selection Criteria

Selecting the type of review

The type of review that should be used depends very much on how thorough the review needs to be and the effort that can be spent. It also depends on the project environment; we cannot give specific recommendations. The decision about what type of review is appropriate must be made on a case-by-case basis. Here are some questions and criteria that should help:

■ The form in which the results of the review should be presented can help select the review type. Is detailed documentation necessary, or is it good enough to present the results informally?

■ Will it be difficult or easy to find a date and time for the review? It can be difficult to bring together five or seven technical experts for one or more meetings.

■ Is it necessary to have technical knowledge from different disciplines?

■ What level (how deep) of technical knowledge is required for the review object? How much time will the reviewers need?

■ Is the preparation effort appropriate with respect to the benefit of the review (the expected result)?

■ How formally written is the review object? Is it possible to perform tool-supported analyses?

■ How much management support is available? Will management curtail reviews when the work is done under time pressure?

Testers as reviewers

It makes sense to use testers as reviewers. The reviewed documents are usually used as the test basis to design test cases. Testers know the documents early and they can design test cases early. By looking at documents from a testing point of view, testers may check new quality aspects, such as testability.

Success Factors

The following factors are crucial for review success and must be considered:

■ Reviews help improve the examined documents. Detecting issues, such as unclear points and deviations, is a wanted and required effect. The issues must be formulated in a neutral and objective way.

■ Human and psychological factors have a strong influence in a review. A review must be conducted in an atmosphere of trust. The participants must be sure that the outcome will not be used to evaluate them (for example, as a basis of their next job assessment). It's important that the author of the reviewed document has a positive experience.

■ Testers should be used as reviewers. They contribute to the review by finding (testing) issues. When they participate in reviews, testers learn about the product, which enables them to prepare tests earlier and in a better way.

■ The type and level of the examined document, and the state of knowledge of the participating people, should be considered when choosing the type of review to use.

■ Checklists and guidelines should be used to help in detecting issues during reviews.
■ Training is necessary, especially for more formal types of reviews, such as inspections.
■ Management can support a good review process by allocating enough resources (time and personnel) for document reviews in the software development process.
■ Continuous learning from executed reviews improves the review process and thus is important.

➔ Static Analysis –

Analysis without executing the program

The objective of static analysis is, as with reviews, to reveal defects or defect-prone parts in a document. However, in static analysis, tools do the analysis. For example, even spell checkers can be regarded as a form of →static analyzers because they find mistakes in documents and therefore contribute to quality improvement.

The term *static analysis* points to the fact that this form of checking does not involve an execution of the checked objects (of a program). An additional objective is to derive measurements, or metrics, in order to measure and prove the quality of the object.

Formal documents

The document to be analyzed must follow a certain formal structure in order to be checked by a tool. Static analysis makes sense only with the support of tools. Formal documents can note, for example, the technical requirements, the software architecture, or the software design. An example is the modeling of class diagrams in UML. Generated outputs in HTML or XML can also be subjected to tool-supported static analysis. Formal models developed during the design phases can also be analyzed and inconsistencies can be detected. Unfortunately, in practice, the program code is often the one and only formal document in software development that can be subjected to static analysis.

Developers typically use static analysis tools before or during component or integration testing to check if guidelines or programming conventions are adhered to. During integration testing, adherence to interface guidelines is analyzed.

Analysis tools often produce a long list of warnings and comments. In order to effectively and efficiently use the tools, the mass of generated information must be handled intelligently; for example, by configuring the tool. Otherwise, the tools might be avoided.

Static analysis and reviews

Static analysis and reviews are closely related. If a static analysis is performed before the review, a number of defects and deviations can be found and the number of the aspects to be checked in the review clearly decreases. Due to the fact that static analysis is tool supported, there is much less effort involved than in a review.

Not all defects can be found using static testing, though. Some defects become apparent only when the program is executed (that means at runtime) and cannot be recognized before. For example, if the value of the denominator in a division is stored in a variable, that variable can be assigned the value zero. This leads to a failure at runtime. In static analysis, this defect cannot easily be found, except for when the variable is assigned the value zero by a constant having zero as its value.

All possible paths through the operations may be analyzed, and the operation can be flagged as potentially dangerous. On the other hand, some inconsistencies and defect-prone areas in a program are difficult to find by dynamic testing. Detecting violation of programming standards or use of forbidden error-prone program constructs is possible only with static analysis (or reviews).

The compiler is an analysis tool

All compilers carry out a static analysis of the program text by checking that the correct syntax of the programming language is used. Most compilers provide additional information, which can be derived by static analysis. In addition to compilers, there are other tools that are so-called analyzers. These are used for performing special analyses or groups of analyses.
The following defects and dangerous constructions can be detected by static analysis:

- Syntax violations
- Deviations from conventions and standards
- →Control flow anomalies
- →Data flow anomalies

Finding security problems

Static analysis can be used to detect security problems. Many security holes occur because certain error-prone program constructs are used or necessary checks are not done. Examples are lack of buffer overflow protection and failing to check that input data may be out of bounds. Tools can find such deficiencies because they often search and analyze certain patterns.

- ✓ Control Flow Analysis

Control flow graph

In figure 4-1, a program structure is represented as a control flow graph. In this directed graph, the statements of the program are represented with nodes. Sequences of statements are also represented with a single node because inside the sequence there can be no change in the course of program execution. If the first statement of the sequence is executed, the others are also executed.

Changes in the course of program execution are made by decisions, such as, for example, in IF statements. If the calculated value of the condition is true, then the program continues in the part that begins with THEN. If the condition is false, then the ELSE part is executed. Loops lead to previous statements, resulting in repeated execution of a part of the graph.

Control flow anomalies

Due to the clarity of the control flow graph, the sequences through the program can easily be understood and possible anomalies can be detected. These anomalies could be jumps out of a loop body or a program structure that has several exits. They may not necessarily lead to failure, but they are not in accordance with the principles of structured programming. It is assumed that the graph is not generated manually but that it is generated by a tool that guarantees an exact mapping of the program text to the graph.

If parts of the graph or the whole graph are very complex and the relations, as well as the course of events, are not understandable, then the program text should be revised, because complex sequence structures often bear a great risk of being wrong.

For a control flow graph (G) of a program or a program part, the cyclomatic number can be computed like this:[11]

$$v(G) = e - n + 2$$

v(G) = cyclomatic number of the graph G
e = number of edges of the control flow graph
n = number of nodes of the control flow graph

A program part is represented by the graph shown in figure 4-1. It is a function that can be called. Thus, the cyclomatic number can be calculated like this:

Example for computing the cyclomatic number

$$v(G) = e - n + 2 = 17 - 13 + 2 = 6$$

e = number of edges in the graph = 17
n = number of nodes in the graph = 13

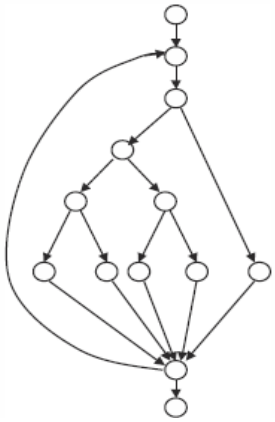


Figure 4–1
Control flow graph for the calculation of the cyclomatic number (identical to figure 2–2)

Excursion: Predecessor-successor table
In addition to graphs, a tool can generate predecessor-successor tables that show how every statement is related to the other statements. If a statement does not have a predecessor, then this statement is unreachable (so-called dead code). Thus a defect or at least an anomaly is detected. The only exceptions are the first and last statements of a program: They can legally have no predecessor or successor. For programs with several entrance and/or exit points, the same applies.

- ✓ Data Flow Analysis

Checking the use of data

→Data flow analysis is another means to reveal defects. Here, the usage of data on →paths through the program code is checked. It is not always possible to decide if an issue is a defect. Instead, we speak of →anomalies, or data flow anomalies. An anomaly is an inconsistency that can lead to failure but does not necessarily do so. An anomaly may be flagged as a risk.
An example of a data flow anomaly is code that reads (uses) variables without previous initialization or code that doesn't use the value of a variable at all. The analysis checks the usage of every single variable. The following three types of usage or states of variables are distinguished:
■ **Defined (d):** The variable is assigned a value.
■ **Referenced (r):** The value of the variable is read and/or used.
■ **Undefined (u):** The variable has no defined value.

Data flow anomalies

We can distinguish three types of data flow anomalies:
■ **ur-anomaly:** An undefined value (u) of a variable is read on a program path (r).
■ **du-anomaly:** The variable is assigned a value (d) that becomes invalid/ undefined (u) without having been used in the meantime.
■ **dd-anomaly:** The variable receives a value for the second time (d) and the first value had not been used (d).

Example of anomalies
We will use the following example (in C++) to explain the different anomalies.

The following function is supposed to exchange the integer values of the parameters Max and Min with the help of the variable Help, if the value of the variable Min is greater that the value of the variable Max:

```
void exchange (int& Min, int& Max) {
int Help;
if (Min > Max) {
Max = Help;
Max = Min;
Help = Min;
}}
```

After the usage of the single variables is analyzed, the following anomalies can be detected:

■ **ur-anomaly** of the variable Help: The domain of the variable is limited to the function. The first usage of the variable is on the right side of an assignment. At this time, the variable still has an undefined value, which is referenced there. There was no initialization of the variable when it was declared (this anomaly is also recognized by usual compilers, if a high warning level is activated).

■ **dd-anomaly** of the variable Max: The variable is used twice consecutively on the left side of an assignment and therefore is assigned a value twice. Either the first assignment can be omitted or the programmer forgot that the first value (before the second assignment) has been used.

■ **du-anomaly** of the variable Help: In the last assignment of the function, the variable Help is assigned another value that cannot be used anywhere because the variable is valid only inside the function.

Data flow anomalies are usually not that obvious

In this example, the anomalies are obvious. But it must be considered that between the particular statements that cause these anomalies there could be an arbitrary number of other statements. The anomalies would not be as obvious anymore and could easily be missed by a manual check such as, for example, a review. A tool for analyzing data flow can, however, detect the anomalies.

Not every anomaly leads directly to an incorrect behavior. For example, a du-anomaly does not always have direct effects; the program could still run properly. The question arises why this particular assignment is at this position in the program, just before the end of the block where the variable is valid. Usually, an exact examination of the program parts where trouble is indicated is worthwhile and further inconsistencies can be discovered.

- ✓ Tools for Static Testing

Static analysis can be executed on source code or on specifications before there are executable programs. Tools for static testing can therefore be helpful to find faults in early phases of the development cycle (i.e., the left branch of the generic V-model in figure 3-1). Faults can be detected and fixed soon after being introduced and thus dynamic testing will be less riddled with problems, which decreases costs and development time.

Tools for review support

Reviews are structured manual examinations using the principle that four eyes find more defects than two. Review support tools help to plan, execute, and evaluate reviews. They store information about planned and executed review meetings, meeting participants, and findings and their resolution and results. Even review aids like checklists can be provided online

and maintained. The collected data from many reviews may be evaluated and compared. This helps to better estimate review resources and to plan reviews but also to uncover typical weaknesses in the development process and specifically prevent them.

Tools for review support are especially useful when large, geographically distributed projects use several teams. Online reviews can be useful here and even may be the only possibility.

Static analysis

Static analyzers provide measures of miscellaneous characteristics of the program code, such as the cyclomatic number and other code metrics (see section 4.2). Such data are useful for identifying complex areas in the code, which tend to be defect prone and risky and should thus be reviewed. These tools can also check that safety- and security-related coding requirements have been followed. Finally, they can identify portability issues in the code.

Additionally, static analyzers can be used to find inconsistencies and defects in the source code. These are, for example, data flow and control flow anomalies, violation of programming standards, and broken or invalid links in website code.

Analyzers list all "suspicious" areas, whether there are really problems or not, causing the output lists to grow very long. Therefore, most tools are configurable; that is, it is possible to control the breadth and depth of analysis. When analyzing for the first time, the tools should be set to be less thorough. A more thorough analysis may be done later. In order to make such tools acceptable for developers, it is essential to configure them according to project-specific needs.

Model checker

Source code is not the only thing that may be analyzed for certain characteristics. Even a specification can be analyzed if it is written in a formal notation or if it is a formal model. The corresponding analysis tools are called *model checkers*. They "read" the model structure and check different static characteristics of these models. During checking, they may find problems such as missing states, state transitions, and other inconsistencies in a model. Such tools are very interesting for developers if they generate test cases.

4. Dynamic Analysis

These test design techniques are divided into three categories: black box testing, white box testing, and experience-based testing.

Execution of the test object on a computer

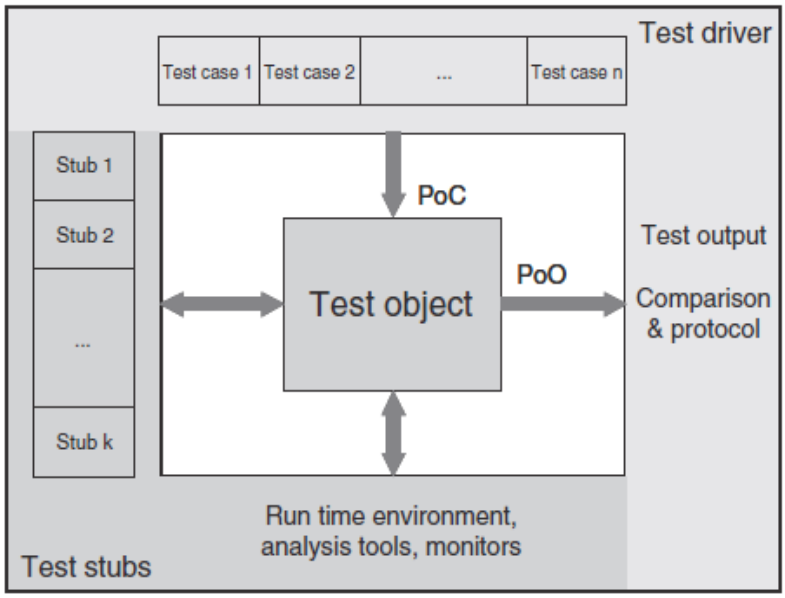
Usually, testing of software is seen as the execution of the test object on a computer. For further clarification, the phrase *dynamic* analysis is used. The test object (program) is fed with input data and executed. To do this, the program must be executable. In the lower test stages (component and integration testing), the test object cannot be run alone but must be embedded into a test harness or test bed to obtain an executable program.

A test bed is necessary

The test object will usually call different parts of the program through predefined interfaces. These parts of the program are represented by placeholders called stubs when they are not yet implemented and therefore aren't ready to be used or if they should be simulated for this particular test of the test object. Stubs simulate the input/output behavior of the part of the program that usually would be called by the test object. Furthermore, the test bed must supply the test object with input data. In most cases, it is necessary to simulate a part of the program that is supposed to call the test object. A test driver does this. Driver and stub combined establish the test bed. Together, they constitute an executable program with the test object itself.

The tester must often create the test bed, or the tester must expand or modify standard (generic) test beds, adjusting them to the interfaces of the test object. Test bed generators can be used as well. An executable test object makes it possible to execute the dynamic test.

Figure 5–1
Test bed



Systematic approach for determining the test cases

The objective of testing is to show that the implemented test object fulfills specified requirements as well as to find possible faults and failures. With as little cost as possible, as many requirements as possible should be checked and as many failures as possible should be found. This goal requires a systematic approach to test case design. Unstructured testing "from your gut feeling" does not guarantee that as many as possible, maybe even all, different situations supported by the test object are tested.

Step wise approach

The following steps are necessary to execute the tests:

■ Determine conditions and preconditions for the test and the goals to be achieved.

■ Specify the individual test cases.
■ Determine how to execute the tests (usually chaining together several test cases).

This work can be done very informally (i.e., undocumented) or in a formal way as described in this chapter. The degree of formality depends on several factors, such as the application area of the system (for example, safety critical software), the maturity of the development and test process, time constraints, and knowledge and skill level of the project participants, just to mention a few.

Conditions, preconditions, and goals

At the beginning of this activity, the test basis is analyzed to determine what must be tested (for example, that a particular transaction is correctly executed). he test objectives are identified, for example, demonstrating that requirements are met. The failure risk should especially be taken into account. The tester identifies the necessary preconditions and conditions for the test, such as what data should be in a database.

Traceability

The traceability between specifications and test cases allows an analysis of the impact of the effects of changed specifications on the test, that is, the necessity for creation of new test cases and removal or change of existing ones. Traceability also allows checking a set of test cases to see if it covers the requirements. Thus, coverage can be a criterion for test exit.

In practice, the number of test cases can soon reach hundreds or thousands. Only traceability makes it possible to identify the test cases that are affected by specification changes.

Test case specification

Part of the specification of the individual test cases is determining test input data for the test object. They are determined using the methods described in this chapter. However, the preconditions for executing the test case, as well as the expected results and expected post conditions, are necessary for determining if there is a failure.

Determining expected result and behavior

The expected results (output, change of internal states, etc.) should be determined and documented before the test cases are executed. Otherwise, an incorrect result can easily be interpreted as correct, thus causing a failure to be overlooked.

Test case execution

It does not make much sense to execute an individual test case. Test cases should be grouped in such a way that a whole sequence of test cases is executed (test sequence, test suite or test scenario). Such a test sequence is documented in the →test procedure specifications or test instructions. This document commonly groups the test cases by topic or by test objectives. Test priorities and technical and logical dependencies between the tests and regression test cases should be visible. Finally, the test execution schedule (assigning tests to testers and determining the time for execution) is described in a →test schedule document.

To be able to execute a test sequence, a test procedure or test script is required. A test script contains instructions for automatically executing the test sequence, usually in a programming language or a similar notation, the test script may contain the corresponding preconditions as well as instruction for comparing the actual and expected results. JUnit is an example of a framework that allows easy programming of test scripts in Java.

Black box and white box test design techniques

Several different approaches are available for designing tests. They can roughly be categorized into two groups: black box techniques and white box techniques. To be more precise, they are collectively called test case design techniques because they are used to design the respective test cases.

In black box testing, the test object is seen as a black box. Test cases are derived from the specification of the test object; information about the inner structure is not necessary or available. The behavior of the test object is watched from the outside (the →Point of

Parameter	Black Box testing	White Box testing
Definition	It is a testing approach which is used to test the software without the knowledge of the internal structure of program or application.	It is a testing approach in which internal structure is known to the tester.
Alias	It also known as data-driven, box testing, data-, and functional testing.	It is also called structural testing, clear box testing, code-based testing, or glass box testing.
Base of Testing	Testing is based on external expectations; internal behavior of the application is unknown.	Internal working is known, and the tester can test accordingly.
Usage	This type of testing is ideal for higher levels of testing like System Testing, Acceptance testing.	Testing is best suited for a lower level of testing like Unit Testing, Integration testing.
Programming knowledge	Programming knowledge is not needed to perform Black Box testing.	Programming knowledge is required to perform White Box testing.
Implementation knowledge	Implementation knowledge is not requiring doing Black Box testing.	Complete understanding needs to implement WhiteBox testing.
Automation	Test and programmer are dependent on each other, so it is tough to automate.	White Box testing is easy to automate.
Objective	The main objective of this testing is to check what functionality of the system under test.	The main objective of White Box testing is done to check the quality of the code.
Basis for test cases	Testing can start after preparing requirement specification document.	Testing can start after preparing for Detail design document.
Tested by	Performed by the end user, developer, and tester.	Usually done by tester and developers.
Granularity	Granularity is low.	Granularity is high.
Testing method	It is based on trial and error method.	Data domain and internal boundaries can be tested.
Time	It is less exhaustive and time-consuming.	Exhaustive and time-consuming method.
Algorithm test	Not the best method for algorithm testing.	Best suited for algorithm testing.
Code Access	Code access is not required for Black Box Testing.	White box testing requires code access. Thereby, the code could be stolen if testing is outsourced.
Benefit	Well suited and efficient for large code segments.	It allows removing the extra lines of code, which can bring in hidden defects.
Skill level	Low skilled testers can test the application with no knowledge of the implementation of programming language or operating system.	Need an expert tester with vast experience to perform white box testing.

Techniques	Equivalence partitioning is Black box testing technique is used for Black box testing. Equivalence partitioning divides input values into valid and invalid partitions and selecting corresponding values from each partition of the test data. Boundary value analysis checks boundaries for input values.	Statement Coverage, Branch coverage, and Path coverage are White Box testing technique. Statement Coverage validates whether every line of the code is executed at least once. Branch coverage validates whether each branch is executed at least once Path coverage method tests all the paths of the program.
Drawbacks	Update to automation test script is essential if you to modify application frequently.	Automated test cases can become useless if the code base is rapidly changing

Observation, or PoO, is outside the test object). The operating sequence of the test object can only be influenced by choosing appropriate input test data or by setting appropriate preconditions. The →Point of Control (PoC) is also located outside of the test object. Test cases are designed using the specification or the requirements of the test object. Often, formal or informal models of the software or component specification are used. Test cases can be systematically derived from such models.

In white box testing, the program text (code) is used for test design. During test execution, the internal flow in the test object is analyzed (the Point of Observation is inside the test object). Direct intervention in the execution flow of the test object is possible in special situations, such as, for example, to execute negative tests or when the component's interface is not capable of initiating the failure to be provoked (the Point of Control can be located inside the test object). Test cases are designed with the help of the program structure (program code or detailed specification) of the test object (see figure 5-2). The usual goal of white box techniques is to achieve a specified coverage; for example, 80% of all statements of the test object shall be executed at least once. Extra test cases may be systematically derived to increase the degree of coverage. White box testing is also called structural testing because it considers the structure (component hierarchy, control flow, data flow) of the test object. The black box testing techniques are also called functional, specification based, or behavioral testing techniques because the observation of the input/output behavior is the main focus. The functionality of the test object is the center of attention.

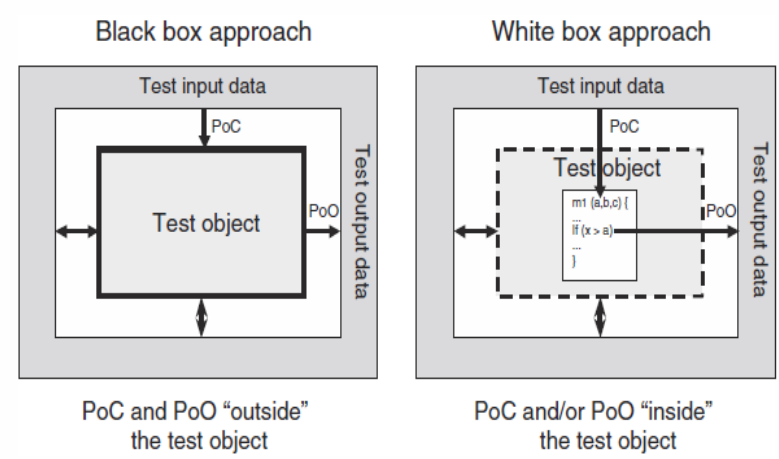


Figure 5–2
PoC and PoO at black box and white box techniques

White box testing can be applied at the lower levels of the testing ,i.e., component and integration test. A system test oriented on the program text is normally not very useful. Black box testing is predominantly used for higher levels of testing even though it is reasonable in component tests. Any test designed before the code is written (test-first programming, test driven development) is essentially applying a black box technique.

Most test methods can clearly be assigned to one of the two categories. Some have elements of both and are sometimes called *gray box techniques*.

Intuitive test case design

Intuitive and experience-based testing is usually black box testing. It is described in a special section because it is not a systematic technique. This test design technique uses the knowledge and skill of people (testers, developers, users, stakeholders) to design test cases. It also uses knowledge about typical or probable faults and their distribution in the test object.

➔ Black Box Testing-

In black box testing, the inner structure and design of the test object is unknown or not considered. The test cases are derived from the specification, or they are already available as part of the specification ("specification by example"). Black box techniques are also called specification based because they are based on specifications (of requirements). A test with all possible input data combinations would be a complete test, but this is unrealistic due to the enormous number of combinations. During test design, a reasonable subset of all possible test cases must be selected. There are several methods to do that, and they will be shown in the following sections.

✓ Equivalence Class Partitioning

Input domains are divided into equivalence classes

The domain of possible input data for each input data element is divided into equivalence classes (equivalence class partitioning). An equivalence class is a set of data values that the tester assumes are processed in the same way by the test object. Testing one representative of the equivalence class is considered sufficient because it is assumed that for any other input value of the same equivalence class, the test object will show the same reaction or behavior. Besides equivalence classes for correct input, those for incorrect input values must be tested as well.

Example for equivalence class partitioning

The example for the calculation of the dealer discount from section 2.2.2 is revisited here to clarify the facts. Remember, the program will prescribe the dealer discount. The following text is part of the description of the requirements: "For a sales price of less than $15,000, no discount shall be given. For a price up to $20,000, a 5% discount is given. Below $25,000, the discount is 7%, and from $25,000 onward, the discount is 8.5%."

Four different equivalence classes with correct input values (called *valid* equivalence classes, or vEC, in the table) can easily be derived for calculating the discount (see table 5-1).

Table 5–1 Valid equivalence classes and representatives

Parameter	Equivalence classes	Representative
Sales price	vEC1: 0 ≤ x < 15000	14500
	vEC2: 15000 ≤ x ≤ 20000	16500
	vEC3: 20000 < x < 25000	24750
	vEC4: x ≥ 25000	31800

In section 2.2.2, the input values 14,500, 16,500, 24,750, 31,800 (see table 2-2) were chosen.

In section 2.2.2, the input values 14,500, 16,500, 24,750, 31,800 (see table 2-2) were chosen.

Every value is a representative for one of the four equivalence classes. It is assumed that test execution with input values like, for example, 13400, 17000, 22300, and 28900 does not lead to further insights and therefore does not find further failures. With this assumption, it is not necessary to execute those extra test cases. Note that tests with boundary values of the equivalence classes (for example, 15000) are discussed in section 5.1.2.

Equivalence classes with invalid values

Besides the correct input values, incorrect or invalid values must be tested. Equivalence classes for incorrect input values must be derived as well, and test cases with representatives of these classes must be executed. In the example we used earlier, there are the following two invalid equivalence classes4 (iEC).

Systematic development of the test cases

The following describes how to systematically derive the test cases. For every input data element that should be tested (e.g., function/method parameter at component tests or input screen field at system tests), the domain of all possible input values is determined. This domain is the equivalence class containing all valid or allowed input values. Following the specification, the program must correctly process these values. The values outside of this domain are seen as equivalence classes with invalid input values. For these values as well, it must be tested how the test object behaves.

Parameter	Equivalence classes	Representative
Sales price	iEC1: x < 0 negative, i.e., wrong sales price	-4000
	iEC2: x > 1000000 unrealistically high sales price[a]	1500800

a. The value 1,000,000 is chosen arbitrarily. Discuss with the car manufacturer or dealer what is unrealistically high!

Table 5–2
Invalid equivalence classes and representatives

Further partitioning of the equivalence classes

The next step is refining the equivalence classes. If the test object's specification tells us that some elements of equivalence classes are processed differently, they should be assigned to a new (sub) equivalence class. The equivalence classes should be divided until each different requirement corresponds to an equivalence class. For every single equivalence class, a representative value should be chosen for a test case.

To complete the test cases, the tester must define the preconditions and the expected result for every test case.

Equivalence classes for output values

The same principle of dividing into equivalence classes can be used for the output data. However, identification of the individual test cases is more expensive because for every chosen output value, the corresponding input value combination causing this output must be determined. For the output values as well, the equivalence classes with incorrect values must not be left out.

Partitioning into equivalence classes and selecting the representatives should be done carefully. The probability of failure detection is highly dependent upon the quality of the partitioning as well as which test cases are executed. Usually, it is not trivial to produce the equivalence classes from the specification or from other documents.

Boundaries of the equivalence classes

The best test values are certainly those verifying the boundaries of the equivalence classes. There are often misunderstandings or inaccuracies in the requirements at these spots because our natural language is not precise enough to accurately define the limits of the equivalence classes. The colloquial phrase *... less than $15000 ...* within the requirements may mean that the value 15000 is inside (EC: x <= 15000) or outside of the equivalence class (EC: x < 15000). An additional test case with x = 15000 may detect a misinterpretation and therefore failure. Section 5.1.2 discusses the analysis of the boundary values for equivalence classes in detail.

- ✓ Boundary Value Analysis

A reasonable extension

Boundary value analysis delivers a very reasonable addition to the test cases that have been identified by equivalence class partitioning. Faults often appear at the boundaries of equivalence classes. This happens because boundaries are often not defined clearly or are misunderstood. A test with boundary values usually discovers failures. The technique can be applied only if the set of data in one equivalence class is ordered and has Identifiable boundaries.

Boundary value analysis checks the *borders* of the equivalence classes. On every border, the exact boundary value and both nearest adjacent values (inside and outside the equivalence class) are tested. The minimal possible increment in both directions should be used. For floating-point data, this can be the defined tolerance. Therefore, three test cases result from every boundary. If the upper boundary of one equivalence class equals the lower boundary of the adjacent equivalence class, then the respective test cases coincide as well.

In many cases there does not exist a "real" boundary value because the boundary value belongs to an equivalence class. In such cases, it can be sufficient to test the boundary with two values: one value that is just inside the equivalence class and another value that is just outside the equivalence class.

Example: Boundary values for discount

For computing the discount on the sales price (table 5-1), four valid equivalence classes were determined and corresponding values chosen for testing the classes.

Equivalence classes 3 and 4 are specified with vEC3: 20000 < x・ 25000 and vEC4: x・ 25000.

For testing the common boundary of the two equivalence classes (25000), the values 24999 and 25000 are chosen (to simplify the situation, it is assumed that only whole dollars are possible). The value 24999 lies in vEC3 and is the largest possible value in that equivalence class. The value 25000 is the least possible value in vEC4. The values 24998 and 25001 do not give any more information because they are further inside their corresponding equivalence classes. Thus, when are the values 24999 and 25000 sufficient and when should we additionally use the value 25001?

Two or three tests

It can help to look at the implementation. The program will probably contain the instruction if (x < 25000)….10 which test cases could find a wrong implementation of this condition? The test values 24999, 25000, and 25001 generate the truth-values true, false, and false for the IF statement and the corresponding program parts are executed. Test value 25001 does not seem to add any value because test value 25000 already generates the truth-value false (and thus the change to the neighbor equivalence class). Wrong implementation of the statement if (x<= 25000) leads to the truth-values true, true, and false. Even here, a test with the value 25001 does not lead to any new results and can thus be omitted, because the test with value 25000 will lead to a failure and thus find the fault. Only a totally wrong implementation stating, for example, if (x <> 25000) and the truth-values true, false, and true can be found with test case value 25001. The values 24999 and 25000 deliver the expected results, that is, the same ones as with the correct implementation.

Hint

Wrong implementation of the instruction in if (x > 25000) with false, false, and true and in if (x=>25000) with false, true, and true results in two or three differences between actual and expected result and can be found by test cases with the values 24999 and 25000.

To illustrate the facts, table 5-8 shows the different conditions and the truth values of the corresponding boundary values.

Table 5–8
Table with three boundary values to test the condition

Implemented condition	24999	25000	25001	Remark
X < 25000 (correct)	True	False	False	Expected result
X ≤ 25000	True	**True**	False	25000 finds the fault
X <> 25000	True	False	**True**	25001 find the fault
X > 25000	**False**	False	**True**	24999 and 25001 find the fault
X ≥ 25000	**False**	**True**	**True**	All three values find the fault
X == 25000	**False**	**True**	False	24999 and 25000 find the fault

It should be decided when a test with only two values is considered enough or when it is beneficial to test the boundary with three values. The wrong query in the example program, implemented as if (x <> 25000), can be found in a code review because it does not check the boundary of a value area if (x < 25000) but instead checks whether two values are unequal. However, this fault can easily be overlooked. Only with a boundary value test with three values can all possible wrong implementations of boundary conditions be found.

- ✓ State Transition Test

Consider history

In many systems, not only the current input but also the history of execution or events or inputs influences computation of the outputs and how the system will behave. History of system execution needs to be taken into account. To illustrate the dependence on history, →state diagrams are used. They are the basis for designing the test (→state transition testing).

The system or test object starts from an initial state and can then comes into different states. Events trigger state changes or transitions. An event may be a function invocation. State transitions can involve actions. Besides the initial state, the other special state is the end state. →Finite state machines, state diagrams, and state transition tables model this behavior.

Definition of a finite state machine

A finite state machine is formally defined as follows: An abstract machine for which the number of states and input symbols are both finite and fixed. A finite state machine consists of states (nodes), transitions (links), inputs (link weights), and outputs (link weights). There are a finite number of internal configurations, called states. The state of a system implicitly contains the information that has resulted from the earlier inputs and that is necessary to find the reaction of the system to new inputs.

Example: Stack

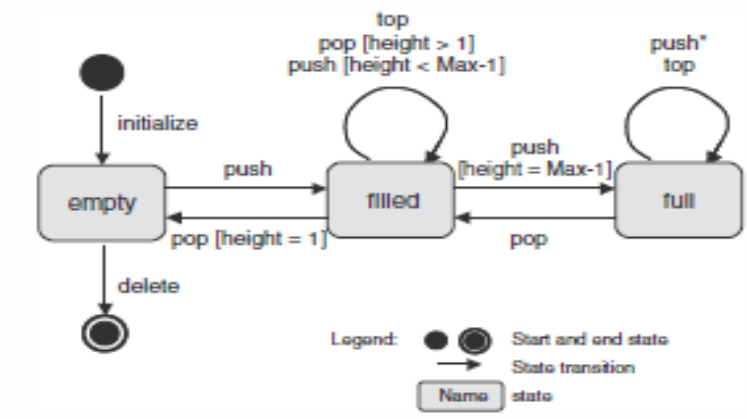


Figure 5–3
State diagram of a stack

Figure 5-3 shows the popular example of a stack. The stack—for example, a dish stack in a heating device—can be in three different states: empty, filled, and full. The stack is "empty" after initializing where the maximum height (Max) is defined (current height = 0). By adding an element to the stack (calling the function push), the state changes to "filled" and the current height is incremented. In this state further elements can be added (push, increment height) as well as withdrawn (call of the function pop, decrement height). The uppermost element can also be displayed (call of the function top, height unchanged). Displaying does not alter the stack itself and therefore does not remove any element. If the current height is one less than the maximum (height = Max – 1) and one element is added to the stack (push), then the state of the stack changes from "filled" to "full." No further element can be added. The condition (Max –1) is described as the *guard* for the transition between the initial and the resulting state. Appropriate guards are illustrated in figure 5-3. If one element is removed (pop) from a stack in the "full" state, the state is changed back from "full" to "filled." A transition from "filled" to "empty" happens only if the stack consists of just one element, which is removed (pop). The stack can only be deleted in the "empty" state. Depending upon the specification, you can define which functions (push, pop, top, etc.) can be called for which state of the stack. You must still clarify what happens when an element is added to a "full" stack (push*). The function must work differently from the case of a just–"filled" stack. Thus, the functions must behave differently depending on the state of the stack. The state of the test object is a decisive element and must be taken into account when testing.

A possible concrete test case

Here is a possible test case with pre- and post conditions for a stack that may store text strings:

Precondition: Stack is initialized, state is "empty"
Input: Push ("hello")
Expected reaction: The stack contains "hello"
Postcondition: State of the stack is "filled"

Further functions of the stack (showing the current level, showing the maximum level, enquiry if the stack is empty, etc.) are not considered in this example because they do not change the state of the stack.

The test object in state transition testing

In state transition testing, the test object can be a complete system with different system states as well as a class in an object-oriented system with different states. Whenever the input history leads to differing behavior, a state transition test must be applied.

Further test cases for the stack example

Different levels of test intensity can be defined for a state transition test. A minimum requirement is to get to all possible states. In the stack example, these states are empty, filled, and full. With an assumed maximum height of 4, all three states are reached after calling the following functions:
Test case 1: initialize [empty], push [filled], push, push, push [full].
Yet, even not all the functions of the stack have been called in this test.

Another requirement for the test is to invoke all functions. With the same stack as before, the following sequence of function calls is sufficient for compliance with this requirement:
Test case 2: initialize [empty], push [filled], top, pop [empty], delete.
However, in this sequence, not all the states have been reached.

Test criteria

A state transition test should execute all specified functions of a state at least once. Compliance between the specified and the actual behavior of the test object can thus be checked.

Design a transition tree

To identify the necessary test cases, the finite state machine is transformed into a so-called transition tree, which includes certain sequences of transitions ([Chow 78]). The cyclic

state transition diagram with potentially infinite sequences of states changes to a transition tree, which corresponds to a representative number of states without cycles. With this translation, all states must be reached and all transitions of the transition diagram must appear.

The transition tree is built from a transition diagram this way:
1. The initial or start state is the root of the tree.
2. For every possible transition from the initial state to a following state in the state transition diagram, the transition tree receives a branch from its root to a node, representing this next state.
3. The process for step 2 is repeated for every leaf in the tree (every newly added node) until one of the following two end conditions is fulfilled:

• The corresponding state is already included in the tree on the way from the root to the node. This end condition corresponds to one execution of a cycle in the transition diagram.

• The corresponding state is a final state and therefore has no further transitions to be considered.

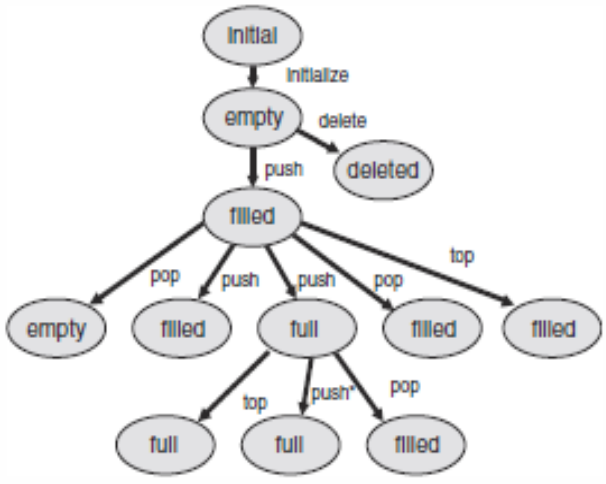
For the stack, the resulting transition tree is shown in figure 5-4.

Eight different paths can be recognized from the root to each of the end nodes (leaves). Each path represents a test case, that is, a sequence of function calls. Thereby, every state is reached at least once, and every possible function is called in each state according to the specification of the state transition diagram.

However, the transition tree doesn't show the appropriate guards, but they need to be taken care of when test cases are designed.

In test case 1 (shown previously), the maximum assumed stack height is four and the guard condition for the transition from the filled to the full state when push is called is max. height (4) – 1 = 3. Three push calls are therefore necessary to pass from filled to full in the transition tree. In addition, another first push call serves to change the state from empty to filled. If no guard conditions are set in a transition tree (as in figures 5-4 and 5-5), it looks like a single push call is sufficient to move from the filled to the full state.

Figure 5–4
Transition tree for the stack example



The transition tree shown in figure 5-4 includes all possible call *sequences* resulting from the state model shown in figure 5-3. In addition, the reaction of the state machine for wrong usage must be checked, which means that functions are called in states in which they are not supposed to be called. Here again the remark that push needs to work differently depends on the state. If

push is called in the "full" state, it cannot add an element to the stack but must leave it unchanged. A message may result, but this is not the same as a fault.

Incorrect use of functions

It is a violation of the specification if functions are called in states where they should not be used (e.g., to delete the stack while in the "full" state). A robustness test must be executed to check how the test object works when used incorrectly. It should be tested to see whether unexpected transitions appear. The test can be seen as an analogy to the test of unexpected input values.

The transition tree should be extended by adding a branch for every function from every node. This means that from every state, all the functions should be executed or at least an attempt should be made to execute them (see figure 5-5).

Producing an extended transition tree can help to find gaps in the specifications.

Here, for example, the pop and top calls that weren't present in the state diagram in figure 5-3 (i.e., that weren't specified) have been added to the state "empty." It definitely makes sense to define which reactions to expect when trying pop and top calls for an empty stack. A reasonable reaction would be, for example, an error message.

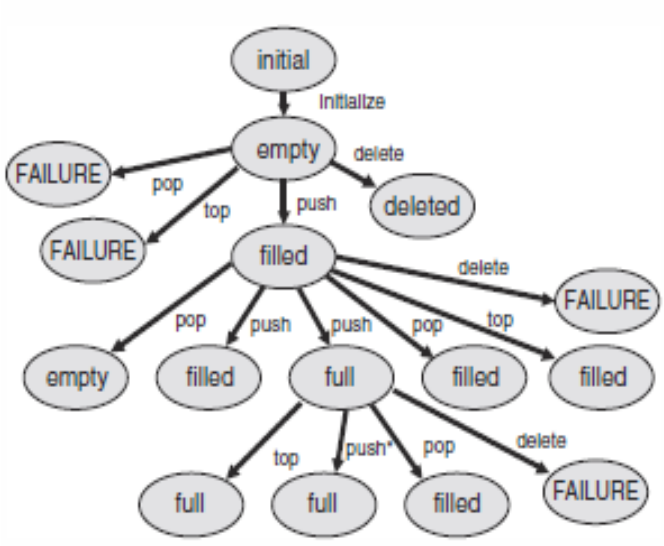


Figure 5–5
Transition tree for the test for robustness

State transition testing is also a good technique for system testing when testing the graphical user interface (GUI) of the test object: The GUI usually consists of a set of screens and dialog boxes; between those, the user can switch back and forth (via menu choices, an OK button, etc.). If screens and user controls are seen as states and input reactions as state transitions, then the GUI can be modeled with a state diagram. Appropriate test cases and test coverage can be identified by the state transition testing technique described earlier.

✓ Cause Effect Graphing

The previously introduced techniques look at the different input data independently. The input values are each considered separately for generating test cases. Dependencies among the different inputs and their effects on the outputs are not explicitly considered for test case design.

Cause-effect graphing describes a technique that uses the dependencies for identification of the test cases. It is known as →cause-effect graphing. The logical relationships between the causes and their effects in a component or a system are displayed in a so-called cause-effect

graph. The precondition is that it is possible to find the causes and effects from the specification. Every cause is described as a condition that consists of input values (or combinations thereof). The conditions are connected with logical operators (e.g., AND, OR, NOT). The condition, and thus its cause, can be true or false. Effects are treated similarly and described in the graph (see figure 5-7).

Example: Cause-effect graph for an ATM

In the following example, we'll use the act of withdrawing money at an automated teller machine (ATM) to illustrate how to prepare a cause-effect graph. In order to get money from the machine, the following conditions must be fulfilled:

- The bank card is valid.
- The PIN is entered correctly.
- The maximum number of PIN inputs is three.
- There is money in the machine and in the account.

The following actions are possible at the machine:

- Reject card.
- Ask for another PIN input.
- "Eat" the card.
- Ask for another amount.
- Pay the requested amount of money.

Figure 5-7 shows the cause-effect graph of the example.

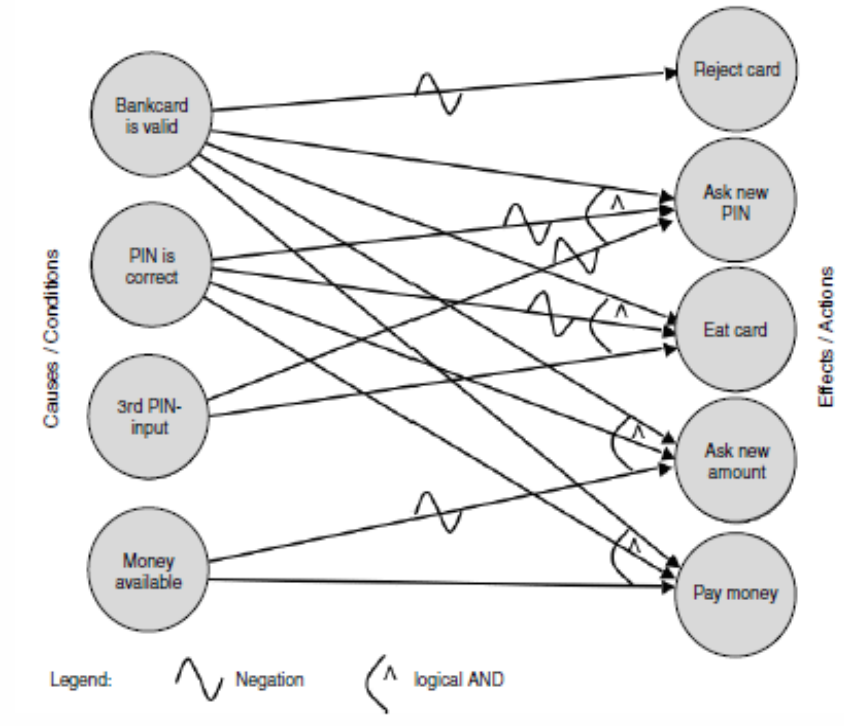


Figure 5–7
Cause-effect graph of the ATM

The graph makes clear which conditions must be combined in order to achieve the corresponding effects.

✓ Decision Table Technique

The graph must be transformed into a →decision table from which the test cases can be derived. The steps to transform a graph into a table are as follows:

1. Choose an effect.
2. Looking in the graph, find combinations of causes that have this effect and combinations that do not have this effect.
3. Add one column into the table for every one of these cause-effect combinations. Include the caused states of the remaining effects.
4. Check to see if decision table entries occur several times, and if they do, delete them.

Test with decision tables

The objective for a test based on decision tables is that it executes "interesting" combinations of inputs—interesting in the sense that potential failures can be detected. Besides the causes and effects, intermediate results with their truth-values may be included in the decision table.

A decision table has two parts. In the upper half, the inputs (causes) are listed; the lower half contains the effects. Every column defines the test situations, i.e., the combination of conditions and the expected effects or outputs.

In the easiest case, every combination of causes leads to one test case. However, conditions may influence or exclude each other in such a way that not all combinations make sense. The fulfillment of every cause and effect is noted in the table with a "yes" or "no." Each cause and effect should at least once have the values "yes" and "no" in the table.

Example: Decision table for an ATM

Because there are four conditions (from "bank card is valid" to "money available"), there are, theoretically, 16 (24) possible combinations. However, not all dependencies are taken into account here. For example, if the bank card is invalid, the other conditions are not interesting because the machine should reject the card.

An optimized decision table does not contain all possible combinations, but the impossible or unnecessary combinations are not entered. The dependencies between the inputs and the results (actions, outputs) lead to the following optimized decision table, showing the result (table 5-11).

Decision table		TC1	TC2	TC3	TC4	TC5
Conditions	Bank card valid?	N	Y	Y	Y	Y
	PIN correct?	-	N	N	Y	Y
	Third PIN attempt?	-	N	Y	-	-
	Money available?	-	-	-	N	Y
Actions	Reject card	Y	N	N	N	N
	Ask for new PIN	N	Y	N	N	N
	"Eat" card	N	N	Y	N	N
	Ask for new amount	N	N	N	Y	N
	Pay cash	N	N	N	N	Y

Table 5–11
Optimized decision table for the ATM

Every column of this table is to be interpreted as a test case. From the table, the necessary input conditions and expected actions can be found directly. Test case 5 shows the following condition: The money is delivered only if the card is valid, the PIN is correct after a maximum of three tries, and there is money available both in the machine and in the account.

This relatively small example shows how more conditions or dependencies can soon result in large and unwieldy graphs or tables. From a decision table, a decision tree may be derived. The decision tree is analogous to the transition tree in state transition testing in how it's used. Every

path from the root of the tree to a leaf corresponds to a test case. Every node on the way to a leaf contains a condition that determines the further path, depending on its truth-value.

Test Cases

Every column is a test case

In a decision table, the conditions and dependencies for the inputs, the corresponding predicted outputs, and the results for this combination of inputs can be read directly from every column to form a test case. The table defines logical test cases. They must be fed with concrete data values in order to be executed, and necessary preconditions and Postcondition must be defined.

Definition of the Test Exit Criteria

Simple criteria for test exit

As with the previous methods, criteria for test completion can be defined relatively easily. A minimum requirement is to execute every column in the decision table by at least one test case. This verifies all sensible combinations of conditions and their corresponding effects.

The Value of the Technique

The systematic and very formal approach in defining a decision table with all possible combinations may show combinations that are not included when other test case design techniques are used. However, errors can result from optimization of the decision table, such as, for example, when the input and condition combinations to be considered are (erroneously) left out.

As mentioned, the graph and the table may grow quickly and lose readability when the number of conditions and dependent actions increases. Without adequate support by tools, the technique is then very difficult.

Pairwise Combination Testing

This test design technique can be used when interactions between different parameters are unknown. This is the opposite of cause-effect graphing, which is designed to cover explicitly known dependencies. Pairwise combination testing has the objective of finding destructive interaction between presumably independent parameters (or parameters for which the specification does not include dependencies).

The technique starts from the equivalence class table. For every equivalence class, a representative value is chosen. Then, every representative for one class is combined with every representative for every other class (taking into account only pairs of combinations, not higher-level combinations). Pairwise combination tests will find any destructive interaction between supposedly independent parameters (provided the representative values chosen do this). Higher-level interactions will not necessarily be discovered. The technique is not easy to apply manually, but tools are available. The technique can be extended to cover higher levels of interaction.

- ✓ User Documentation Testing

User documentation covers all the manuals, user guides, installation guides, setup guides, read me file, software release notes, and online help that are provided along with the software to help the end user to understand the software system.

User documentation testing should have two objectives.

1. To check if what is stated in the document is available in the product.
2. To check if what is there in the product is explained correctly in the document.

When a product is upgraded, the corresponding product documentation should also get updated as necessary to reflect any changes that may affect a user. However, this does not necessarily happen all the time. One of the factors contributing to this may be lack of sufficient coordination between the documentation group and the testing/development groups. Over a period of time, product documentation diverges from the actual behavior of the product. User documentation testing focuses on ensuring what is in the document exactly matches the product behavior, by sitting in front of the system and verifying screen by screen, transaction by transaction and report by report. In addition, user documentation testing also checks for the language aspects of the document like spell check and grammar.
User documentation is done to ensure the documentation matches the product and vice-versa.
Testing these documents attain importance due to the fact that the users will have to refer to these manuals, installation, and setup guides when they start using the software at their locations. Most often the users are not aware of the software and need hand holding until they feel comfortable. Since these documents are the first interactions the users have with the product, they tend to create lasting impressions. A badly written installation document can put off a user and bias him or her against the product, even if the product offers rich functionality.
Some of the benefits that ensue from user documentation testing are:

1. User documentation testing aids in highlighting problems over looked during reviews.
2. High quality user documentation ensures consistency of documentation and product, thus minimizing possible defects reported by customers. It also reduces the time taken for each support call—sometimes the best way to handle a call is to alert the customer to the relevant section of the manual. Thus the overall support cost is minimized.
3. Results in less difficult support calls. When a customer faithfully follows the instructions given in a document but is unable to achieve the desired (or promised) results, it is frustrating and often this frustration shows up on the support staff. Ensuring that a product is tested to work as per the document and that it works correctly contributes to better customer satisfaction and better morale of support staff.
4. New programmers and testers who join a project group can use the documentation to learn the external functionality of the product.
5. Customers need less training and can proceed more quickly to advanced training and product usage if the documentation is of high quality and is consistent with the product. Thus high-quality user documentation can result in a reduction of overall training costs for user organizations.

Defects found in user documentation need to be tracked to closure like any regular software defect. In order to enable an author to close a documentation defect information about the defect/comment description, paragraph/page number reference, document version number reference, name of reviewer, name of author, reviewer's contact number, priority, and severity of the comment need to be passed to the author.
Since good user documentation aids in reducing customer support calls, it is a major contibutor to the bottom line of the organization. The effort and money spent on this effort would form a valuable investment in the long run for the organization.

- ✓ Domain Testing

White box testing required looking at the program code. Black box testing performed testing without looking at the program code but looking at the specifications. Domain testing can be considered as the next level of testing in which we do not look even at the specifications of a software product but are testing the product, purely based on domain knowledge and

expertise in the domain of application. This testing approach requires critical understanding of the day-to-day business activities for which the software is written. This type of testing requires business domain knowledge rather than the knowledge of what the software specification contains or how the software is written. Thus domain testing can be considered as an extension of black box testing. As we move from white box testing through black box testing to domain testing (as shown in Figure 4.5) we know less and less about the details of the software product and focus more on its external behavior.

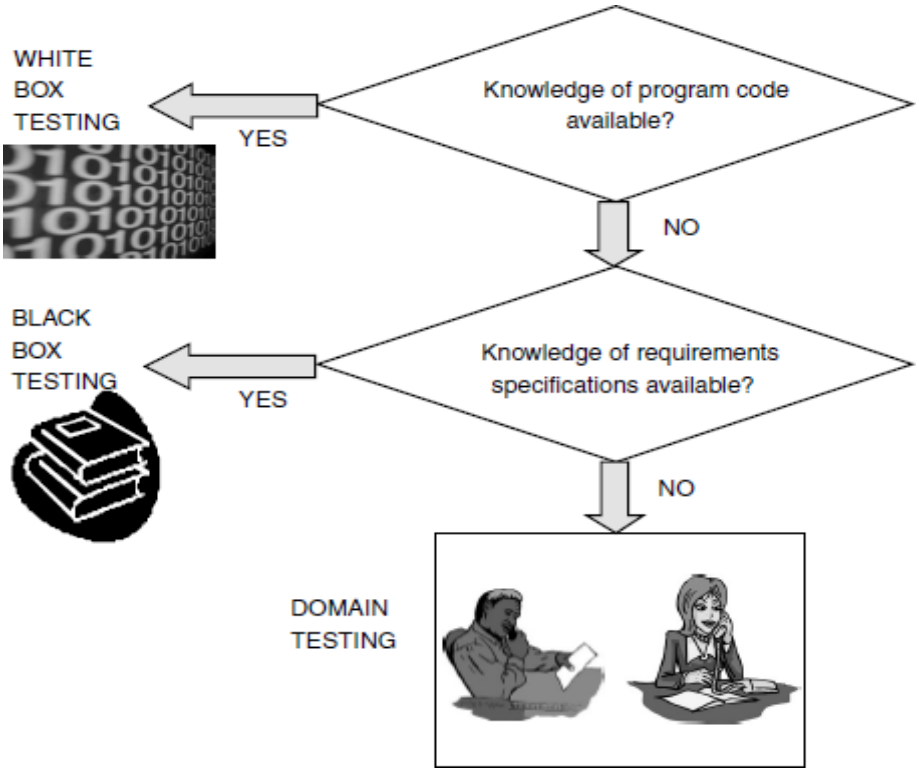


Figure 4.5 Context of white box, black box and domain testing.

The test engineers performing this type of testing are selected because they have in-depth knowledge of the business domain. Since the depth in business domain is a prerequisite for this type of testing, sometimes it is easier to hire testers from the domain area (such as banking, insurance, and so on) and train them in software, rather than take software professionals and train them in the business domain. This reduces the effort and time required for training the testers in domain testing and also increases the effectiveness of domain testing.

For example, consider banking software. Knowing the account opening process in a bank enables a tester to test that functionality better. In this case, the bank official who deals with account opening knows the attributes of the people opening the account, the common problems faced, and the solutions in practice. To take this example further, the bank official might have encountered cases where the person opening the account might not have come with the required supporting documents or might be unable to fill up the required forms correctly. In such cases, the bank official may have to engage in a different set of activities to open the account. Though most of these may be stated in the business requirements explicitly, there will be cases that the bank official would have observed while testing the software that are not captured in the requirement specifications explicitly. Hence, when he or she tests the software, the test cases are likely to be more thorough and realistic.

Domain testing is the ability to design and execute test cases that relate to the people who will buy and use the software. It helps in understanding the problems they are trying to solve and the ways in which they are using the software to solve them. It is also characterized by how well an individual test engineer understands the operation of the system and the business processes that system is supposed to support. If a tester does not understand the system or the business processes, it would be very difficult for him or her to use, let alone test, the application without the aid of test scripts and cases.
Domain testing exploits the tester's domain knowledge to test the suitability of the product to what the users do on a typical day.
Domain testing involves testing the product, not by going through the logic built into the product. The business flow determines the steps, not the software under test. This is also called "business vertical testing." Test cases are written based on what the users of the software do on a typical day.
Let us further understand this testing using an example of cash withdrawal functionality in an ATM, extending the earlier example on banking software. The user performs the following actions.

Step 1: Go to the ATM.
Step 2: Put ATM card inside.
Step 3: Enter correct PIN.
Step 4: Choose cash withdrawal.
Step 5: Enter amount.
Step 6: Take the cash.
Step 7: Exit and retrieve the card.

In the above example, a domain tester is not concerned about testing everything in the design; rather, he or she is interested in testing everything in the business flow. There are several steps in the design logic that are not necessarily tested by the above flow. For example, if you were doing other forms of black box testing, there may be tests for making sure the right denominations of currency are used. A typical black box testing approach for testing the denominations would be to ensure that the required denominations are available and that the requested amount can be dispensed with the existing denominations. However, when you are testing as an end user in domain testing, all you are concerned with is whether you got the right amount or not. (After all, no ATM gives you the flexibility of choosing the denominations.) When the test case is written for domain testing, you would find those intermediate steps missing. Just because those steps are missing does not mean they are not important. These "missing steps" (such as checking the denominations) are expected to be working before the start of domain testing.
Black box testing advocates testing those areas which matter more for a particular type or phase of testing. Techniques like equivalence partitioning and decision tables are useful in the earlier phases of black box testing and they catch certain types of defects or test conditions. Generally, domain testing is done after all components are integrated and after the product has been tested using other black box approaches (such as equivalence partitioning and boundary value analysis). Hence the focus of domain testing has to be more on the business domain to ensure that the software is written with the intelligence needed for the domain. To test the software for a particular "domain intelligence," the tester is expected to have the intelligence and knowledge of the practical aspects of business flow. This will reflect in better and more

effective test cases which examine realistic business scenarios, thus meeting the objective and purpose of domain testing.

➔ White Box

Wrong specification is not detected

The basis of all black box techniques is the requirements or specifications of the system or its components and how they collaborate. Black box testing will not be able to find problems where the implementation is based on incorrect requirements or a faulty design specification because there will be no deviation between the faulty specification or design and the observed results. The test object will execute as the requirements or specifications require, even when they are wrong. If the tester is critical toward the requirements or specifications and uses "common sense", she may find wrong requirements during test design.

Otherwise, to find inconsistencies and problems in the specifications, reviews must be used.

Functionality that's not required is not detected

In addition, black box testing cannot reveal extra functionality that exceeds the specifications. (Such extra functionality is often the cause of security problems.) Sometimes additional functions are neither specified nor required by the customer. Test cases that execute those additional functions are performed by pure chance if at all. The coverage criteria, which serve as conditions for test exit, are exclusively identified on the basis of the specifications or requirements. They are not based on unmentioned or assumed functions.

Verification of the functionality

The center of attention for all black box techniques is the verification of the functionality of the test object. It is indisputable that the highest priority is that the software work correctly. Thus, black box techniques should always be applied.

Code-based testing techniques

The basis for white box techniques is the source code of the test object. Therefore, these techniques are often called structure-based testing techniques because they are based on the structure (of the program). They are also called →code-based testing techniques. The source code must be available, and in certain cases, it must be possible to manipulate it, that is, to add code.

All code should be executed

The foundation of white box techniques is to execute every part of the code of the test object at least once. Flow-oriented test cases are identified, analyzing the program logic, and then they are executed. However, the expected results should be determined using the requirements or specifications, not the code. This is done in order to decide if execution resulted in a failure.

A white box technique can focus on, for example, the statements of the test object. The primary goal of the technique is then to achieve a previously defined coverage of the statements during testing, such as, for example, to execute as many statements of the program as possible.

These are the white box test case design techniques:

■ →Statement testing

■ →Decision testing or →branch testing

■ testing of conditions

- →Condition testing
- →Multiple condition testing
- →Condition determination testing

■ →Path testing

✓ Statement Coverage

Control flow graph is necessary

This analysis focuses on each *statement* of the test object. The test cases shall execute a predefined minimum quota or even all statements of the test object. The first step is to translate the source code into a control flow graph. The graph makes it easier to specify in detail the control elements that must be covered. In the graph, the statements are represented as nodes (boxes) and the control flow between the statements is represented as edges (connections). If sequences of unconditional statements appear in the program fragment, they are illustrated as one single node because execution of the first statement of the sequence guarantees that all following statements will be executed. Conditional statements (IF, CASE) and loops (WHILE, FOR), represented as control flow graphs, have more than one edge going to the exit node.

After execution of the test cases, it must be determined which statements have been executed. When the previously defined coverage level has been achieved, the test is considered to be sufficient and will therefore be terminated. Normally, all instructions should be executed because it is impossible to verify the correctness of instructions that have not been executed.

Example

The following example will clarify how to do this. We chose a very simple program fragment for this example. It consists of only two decisions and one loop (figure 5-9).

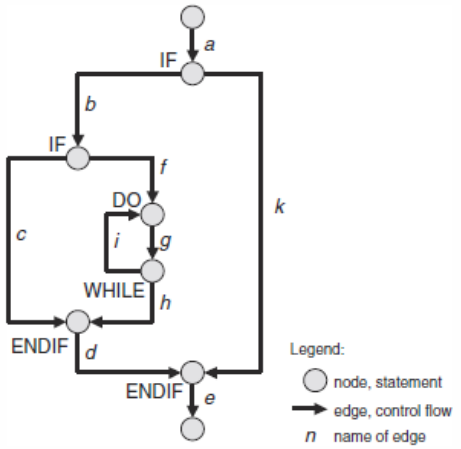


Figure 5–9
Control flow of a program fragment

Test cases

Coverage of the nodes of the control flow

In this example, all statements (all nodes) can be reached by a single test case. In this test case, the edges of the graph must be traversed in this order:
a, b, f, g, h, d, e

One test case is enough

After the edges are traversed in this way, all statements have been executed once. Other combinations of edges of the graph can also be used to achieve complete coverage. But the cost of testing should always be minimized, which means reaching the goal with the smallest possible number of test cases.

The expected results and the expected behavior of the test object should be identified in advance from the specification (not the code!). After execution, the expected and actual results, and the behavior of the test object, must be compared to detect any difference or failure.

Definition of the Test Exit Criteria

The exit criteria for the tests can be very clearly defined:

→Statement coverage =(number of executed statements / total number of statements) × 100%

C0-measure

Statement coverage is also known as C0-coverage (C-zero). It is a very weak criterion. However, sometimes 100% statement coverage is difficult to achieve, as when, for instance, exception conditions appear in the program that can be triggered only with great trouble or not at all during test execution.

The Value of the Technique

Unreachable codes can be detected

If complete coverage of all statements is required and some statements cannot be executed by any test case, this may be an indication of unreachable source code (dead statements).

Empty ELSE parts are not considered

If a condition statement (IF) has statements only after it is fulfilled (i.e., after the THEN clause) and there is no ELSE clause, then the control flow graph has a THEN edge, starting at the condition, with (at least) one node, but additionally a second outgoing ELSE edge without any intermediate nodes. The control flow of both of these edges is reunited at the terminating (ENDIF) node. For statement coverage, an empty ELSE edge (between IF and ENDIF) is irrelevant. Possible missing statements in this program part are not detected by a test using this criterion! Statement coverage is measured using test tools.

✓ Branch Coverage

A more advanced criterion for white box testing is →branch coverage of the control flow graph; for example, the edges (connections) in the graph are the center of attention. This time, the execution of decisions is considered instead of the execution of the statement. The result of the decision determines which statement is executed next. This should be used in testing.

→*branch or decision test*

If the basis for a test is the control flow graph with its nodes and edges, then it is called a branch test or branch coverage. A branch is the connection between two nodes of the graph. In the program text, there are IF or CASE statements, loops, and so on, also called *decisions*. This test is thus called *decision test* or *decision coverage.* There may be differences in the degree of coverage. The following example illustrates this: An IF statement with an empty ELSE-part is checked. Decision testing gives 50% coverage if the condition is evaluated to true. With one more test case where the condition is false, 100% decision coverage will be achieved. For the branch test, which is built from the control flow graph, slightly different values result. The THEN part consists of two branches and one node, the ELSE part only of one branch without any node (no statement there). Thus, the whole IF statement with the empty ELSE part consists of three branches. Executing the condition with true results in covering two of the three branches, that is, 66% coverage. (Decision testing gives 50% in this case). Executing the second test case with the condition being false, 100% branch coverage and 100% decision coverage are achieved. Branch testing is discussed further a bit later.

Empty ELSE-parts are considered

Thus, contrary to statement coverage, for branch coverage it is not interesting if, for instance, an IF statement has no ELSE-part. It must be executed anyway. Branch coverage requires the test of every decision outcome: both THEN and ELSE in the IF statement; all possibilities for the CASE statement and the fall-through case; for loops, both execution of the loop body, bypassing the loop body and returning to the start of the loop.

Test Cases

Additional test cases necessary

In the example (figure 5-9), additional test cases are necessary if all branches of the control flow graph must be executed during the test. For 100% statement coverage, a test case executing the following order of edges was sufficient:

a, b, f, g, h, d, e

The edges c, i, and k have not been executed in this test case. The edges c and k are empty branches of a condition, while the edge i is the return to the beginning of the loop. Three test cases are necessary:

a, b, c, d, e

a, b, f, g, i, g, h, d, e

a, k, e

Connection (edge) coverage of the control flow graph

All three test cases result in complete coverage of the edges of the control flow graph. With that, all possible branches of the control flow in the source code of the test object have been tested. Some edges have been executed more than once. This seems to be redundant, but it cannot always be avoided. In the example, the edges a and e are executed in every test case because there is no alternative to these edges.

For each test case, in addition to the preconditions and Postcondition, the expected result and expected behavior must be determined and then compared to the actual result and behavior. Furthermore, it is reasonable to record which branches have been executed in which test case in order to find wrong execution flows. This helps to find faults, especially missing code in empty branches.

Definition of the Test Exit Criteria

As with statement coverage, the degree of branch coverage is defined as follows:

Branch coverage = (number of executed branches / total number of branches) × 100%

C1-measure

Branch coverage is also called C1-coverage. The calculation counts only if a branch has been executed at all. The frequency of execution is not relevant. In our example, the edges a and e are each passed three times—once for each test case.

If we execute only the first three test cases in our example (not the fourth one), edge k will not be executed. This gives branch coverage of 9 executed branches out of 10 total:

(9 / 10) × 100% = 90%.

For comparison, 100% statement coverage has already been achieved after the first test case.

Depending on the criticality of the test object, and depending on the expected failure risk, the test exit criterion can be defined differently. For instance, 85% branch coverage can be sufficient for a component of one project, whereas for a different project, another component

must be tested with 100% coverage. The example shows that the test cost is higher for higher coverage requirements.

The Value of the Technique

More test cases necessary

Decision/branch coverage usually requires the execution of more test cases than statement coverage. How much more depends on the structure of the test object. In contrast to statement coverage, branch coverage makes it possible to detect missing statements in empty branches. Branch coverage of 100% guarantees 100% statement coverage, but not vice versa. Thus, branch coverage is a stronger criterion. Each of the branches is regarded separately and no particular combinations of single branches are required.

Inadequate for object oriented systems

For object-oriented systems, statement coverage as well as branch coverage are inadequate because the control flow of the functions in the classes is usually short and not very complex. Thus, the required coverage criteria can be achieved with little effort. The complexity in object-oriented systems lies mostly in the relationship between the classes, so additional adequate coverage criteria are necessary in this case. As tools often support the process of determining coverage, coverage data can be used to detect not called methods or program parts.

✓ Test of Conditions

Considering the complexity of combined conditions

Branch coverage exclusively considers the logical value of the result of a condition ("true" or "false"). Using this value, it is decided which branch in the control flow graph to choose and, accordingly, which statement is executed next in the program. If a decision is based on several (partial) conditions connected by logical operators, then the complexity of the condition should be considered in the test. The following sections describe different requirements and degrees of test intensity under consideration of combined conditions.

Condition Testing and Coverage

The goal of condition testing is to cause each →atomic (partial) condition in the test to adopt both a *true* and a *false* value.

Definition of an atomic partial condition

An atomic partial condition is a condition that has no logical operators such as AND, OR, and NOT but at the most includes relation symbols such as > and =. A condition in the source code of the test object can consist of multiple atomic partial conditions.

Example for combined conditions

An example for a combined condition is x > 3 OR y < 5. The condition consists of two conditions (x > 3; y < 5) connected by the logical operator OR. The goal of condition testing is that each partial condition (i.e., each individual part of a combined condition) is evaluated once, resulting in each of the logical values. The test data x = 6 and y = 8 result in the logical value *true* for the first condition (x > 3) and the logical value *false* for the second condition (y < 5). The logical value of the complete condition is *true* (*true* OR *false* = *true*). The second pair of test data with the values x = 2 and y = 3 results in *false* for the first condition and *true* for the second condition. The value of the complete condition results in *true* again (*false* OR *true* = *true*). Both parts of the combined condition have each resulted in both logical values. The result of the complete condition, however, is equal for both combinations.

A weak criterion

Condition coverage is therefore a weaker criterion than statement or branch coverage because it is not required that different logical values for the result of the complete condition are included in the test.

Multiple Condition Testing and Coverage

All combinations of the logical values

Multiple condition testing requires that all *true-false* combinations of the atomic partial conditions be exercised at least once. All variations should be built, if possible.

Continuation of the example

Four combinations of test cases are possible with the test data from the previous example for the two conditions (x > 3, y < 5):

x = 6 (T), y = 3 (T), x > 3 OR y < 5 (T)
x = 6 (T), y = 8 (F), x > 3 OR y < 5 (T)
x = 2 (F), y = 3 (T), x > 3 OR y < 5 (T)
x = 2 (F), y = 8 (F), x > 3 OR y < 5 (F)

Multiple condition testing subsumes statement and branch coverage

The evaluation of the complete condition results in both logical values. Thus, multiple condition testing meets the criteria of statement and branch coverage. It is a more comprehensive criterion that also takes into account the complexity of combined conditions. But this is a very expensive technique due to the growing number of atomic partial conditions that make the number of possible combinations grow exponentially (to 2n, with n being the number of atomic partial conditions).

Not all combinations are always possible

A problem results from the fact that test data cannot always generate all combinations.

Example for not feasible combinations of partial condition

An example should clarify this. For the combined condition of 3<=x AND x<5 not all combinations with the according values for the variable x can be produced because the parts of the combined condition depend on each other:

x = 4: 3<= x (T), x < 5 (T), 3<=x AND x < 5 (T)
x = 8: 3<= x (T), x < 5 (F), 3 <= x AND x < 5 (F)
x = 1: 3<= x (F), x < 5 (T), 3 <= x AND x < 5 (F)
x = ?: 3<= x (F), x < 5 (F), combination not possible because the value x shall be smaller than 3 and greater than or equal to 5 at the same time.

Condition Determination Testing / Minimal Multiple Condition Testing

Restriction of the combinations

Condition determination testing eliminates the problems discussed previously. Not all combinations must be included; however, include every possible combination of logical values where the modification of the logical value of an atomic partial condition can change the logical value of the whole combined condition. Stated in another way, for a test case, every atomic partial condition must have a meaningful impact on the result. Test cases in which the result does not depend on a change of an atomic partial condition need not be designed.

Continuation of the example

For clarification, we revisit the example with the two atomic partial conditions (x > 3, y < 5) and the OR-connection (x > 3 OR y < 5). Four combinations are possible (22):

1) x = 6 (T), y = 3 (T), x > 3 OR y < 5 (T)
2) x = 6 (T), y = 8 (F), x > 3 OR y < 5 (T)

3) x = 2 (F), y = 3 (T), x > 3 OR y < 5 (T)
4) x = 2 (F), y = 8 (F), x > 3 OR y < 5 (F)

Changing a partial condition without changing the result

For the first combination, the following applies: If the logical value is wrongly calculated for the first condition (i.e., an incorrect condition is implemented), then the fault can change the logical value of the first condition part from *true* (T) to *false* (F). But the result of the complete condition stays unchanged (T). The same applies for the second partial condition.

For the first combination, incorrect results of each partial condition are masked because they have no effect on the result of the complete condition and thus failures will not become visible at the outside. Consequently, the test with the first combination can be left out.

If the logical value of the first partial condition in the second test case is calculated wrongly as *false*, then the result value of the combined condition changes from *true* (T) to *false* (F). A failure then becomes visible because the value of the combined condition has also changed. The same applies for the second partial condition in the third test case. In the fourth test case, an incorrect implementation is detected as well because the logical value of the complete condition changes.

Small number of test cases

For every logical combination of the combined decision, it must be decided which test cases are sensitive to faults and for which combinations faults can be masked. Combinations where faults are masked need not be considered in the test. Here, the number of test cases is significantly smaller than in multiple condition testing.

Test Cases

For designing the test cases, it must be considered which input data leads to which result of the decisions or partial conditions and which parts of the program will be executed after the decision. The expected output and expected behavior of the test object should also be defined in advance in order to detect whether the program behaves correctly.

However, it may be very expensive to choose the input values in such a way that a certain part of the condition gets the logical value required by the test case.

Definition of the Test Exit Criteria

Analogous to the previous techniques, the proportion between the executed and all the required logical values of the (partial) condition (parts) can be calculated. For the techniques, which concentrate on the complexity of the decisions in the source code, it is reasonable to try to achieve a complete verification (100% coverage). If complexity of the decisions is not important in testing, branch coverage can be seen as sufficient.

The Value of the Technique

Complex conditions are often defect prone

If complex decisions are present in the source code, they must be tested intensively to detect possible failures. Combinations of logical expressions are especially defect prone. Thus, a comprehensive test is very important. However, condition determination testing is an expensive technique for test case design.

Excursion

A disadvantage of condition testing is that it checks Boolean expressions only inside a statement (for example, IF statement). In the following example of a program fragment, the following fact

remains undetected: the IF condition actually consists of multiple parts and condition determination testing needs to be applied.

```
...
Flag = (A || (B && C));
If (Flag)
...;
else ...;
...
```

This particular disadvantage can be circumvented if all Boolean expressions that occur are used as a basis for the creation of test cases.

The compiler terminates evaluation of expressions

Another problem occurs in connection with measuring the coverage of (partial) conditions. some compilers shortcut the evaluation of the Boolean expression as soon as the total result of the decision is known. For instance, if the value FALSE has been detected for one of two condition parts of an AND-combination, then the complete condition is FALSE regardless of the result of the second condition part. Some compilers even change the order of the evaluation, depending on the Boolean operators, to get the final result as quickly as possible and to be able to disregard any other partial conditions. Test cases that are supposed to achieve 100% coverage can be executed, but because of the shortened evaluation, this coverage cannot be verified.

✓ Path Coverage

Until now, test case determination focused on the statements or branches of the control flow as well as the complexity of decisions. If the test object includes loops or repetitions, the previous considerations are not sufficient for an adequate test. Path coverage requires the execution of all different paths through the test object.

Excursion:

All possible paths through a test object

Example for a path test

To clarify the use of the term *path*, consider the control flow graph in figure 5-9. The program fragment represented by the graph contains a loop. This DOWHILE loop is executed at least once. In the WHILE condition at the end of the loop, it is decided whether the loop must be repeated, that is, if a jump back to the start of the loop is necessary. When using branch coverage for test design, the loop has been considered in two test cases:

■ Loop without repetition:

a, b, f, g, h, d, e

■ Loop with single return (i) and a single repetition:

a, b, f, g, i, g, h, d, e

Usually a loop is repeated more than once. Further possible sequences of branches through the graph of the program are as follows:

a, b, f, g, i, g, i, g, h, d, e

a, b, f, g, i, g, i, g, i, g, h, d, e

a, b, f, g, i, g, i, g, i, g, i, g, h, d, e

etc.

This shows that there are indefinite numbers of paths in the control flow graph. Even with restrictions on the number of loop repetitions, the number of paths increases indefinitely.

Combination of program parts

A path describes the possible order of single program parts in a program fragment. Contrary to this, branches are viewed independently, each for itself. The paths consider dependencies between the branches, as for example with loops, at which one branch leads back to the beginning of another branch.

Example: Statement and branch coverage in VSR

In section 5.1.1 for the function calculate_price() of the VSR subsystem *DreamCar*, test cases have been derived from valid and invalid equivalence classes of the parameters. In the following code, test cases are evaluated by their Branch coverage of 100% should be achieved to ensure that during test execution all branches have been executed at least once.

For better understanding, the source code of the function from section 3.2.3 is repeated here:

```
double calculate_price (
double baseprice, double specialprice,
double extraprice, int extras, double discount)
{
double addon_discount;
double result;
if (extras >= 3) addon_discount = 10;
else if (extras >= 5) addon_discount = 15;
else addon_discount = 0;
if (discount > addon_discount)
addon_discount = discount;
result = baseprice /100.0*(100-discount)
+ specialprice
+ extraprice/100.0*(100-addon_discount);
return (result);
}
```

The control flow graph of the function calculate_price() is shown in figure 5-10.

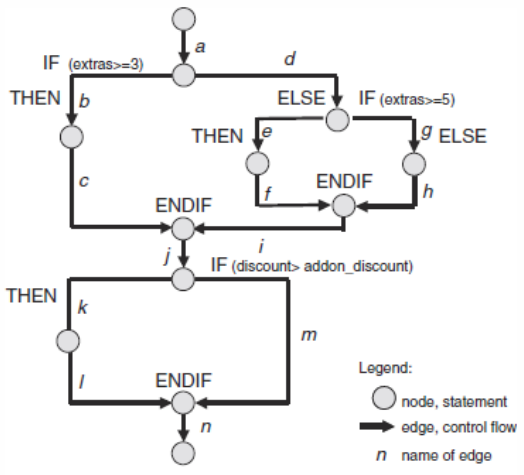


Figure 5–10
Control flow graph of the function `calculate_price()`

In section 3.2.3, the following two test cases have been chosen:

```
// testcase 01
price = calculate_price(10000.00,2000.00,1000.00,3,0);
test_ok = test_ok && (abs(price–12900.00) < 0.01);
```

// testcase 02
price = calculate_price(25500.00,3450.00,6000.00,6,0);
test_ok = test_ok && (abs(price-34050.00) < 0.01);
The test cases cause the execution of the following edges of the graph:
Test case 01: a, b, c, j, m, n
Test case 02: a, b, c, j, m, n

43% branch coverage achieved
The edges d, e, f, g, h, i, k, l have not been executed. The two test cases covered only 43% of the branches (6 out of 14). Test case 02 gives no improvement of the coverage and is not necessary for branch coverage. However, considering the specification, test case 02 should have led to execution of more statements because a different discount should have been calculated (with five or more pieces of extra equipment).

To increase test coverage, the following further test cases are specified:

// testcase 03
price = calculate_price(10000.00,2000.00,1000.00,0,10);
test_ok = test_ok && (abs(price–12000.00) < 0.01);

// testcase 04
price = calculate_price(25500.00,3450.00,6000.00,6,15);
test_ok = test_ok && (abs(price–30225.00) < 0.01);
These test cases cause the execution of the following edges of the graph:
Test case 03: a, d, g, h, i, j, k, l, n
Test case 04: a, b, c, j, k, l, n

86% path coverage achieved
These test cases lead to execution of further edges (d, g, h, i, k, and l) and thus increase branch coverage to 86%. Edges e and f have not yet been executed.

Evaluation of the conditions
Before trying to reach the missing edges by further test cases, the conditions of the IF statements are analyzed more closely, that is, the source code is analyzed in order to define further test cases. To get to the edges e and f, the result of the first condition (extras ・ 3) must be false in order to execute the ELSE-part. In this ELSE-part, the condition (extras ・ 5) must be true. Therefore, a value has to be found that meets the following condition:
¬(extras=>3) AND (extras =>5)
No such value exists and the missing edges can never be reached. The source code contains a defect.

Example: Relationship between the measures
This example should clarify the relationship between statement, branch, and path coverage as well. The test object consists of altogether three IF statements; two are nested and the third is placed separately from the others (figure 5-10). All statements (nodes) are executed by the following sequence of edges in the graph:
a, b, c, j, k, l, n
a, d, e, f, i, j, k, l, n
a, d, g, h, i, j, k, l, n

These sequences are sufficient to achieve 100% statement coverage. But not all branches (edges) have been covered yet. The edge m is still missing. An execution sequence might look like this:
a, b, c, j, m, n
This new sequence can replace the first execution sequence shown previously. With the resulting three test cases, which result in these three execution sequences, 100% branch coverage is achieved.

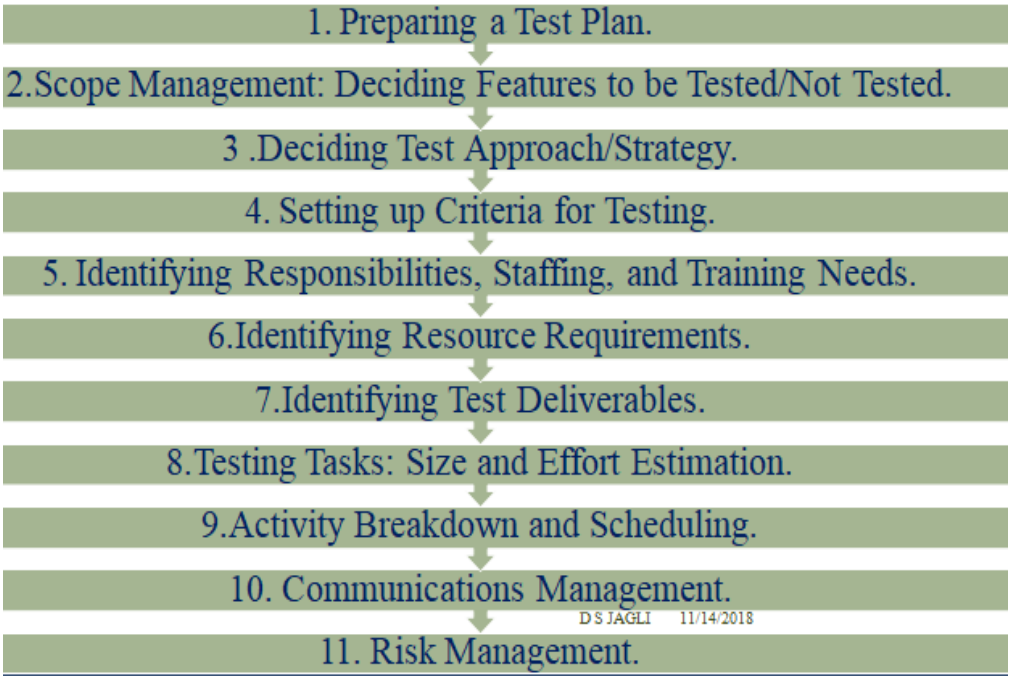
Further paths through the graph
But, even for this simple program fragment, there are still possibilities to traverse the graph differently and thus take care of all paths through the graph. Until now, the following paths have not been executed yet:
a, d, e, f, i, j, m, n
a, d, g, h, i, j, m, n
Altogether, six different paths through the source code result (the three possible paths through the graph before edge j multiplied by two for the two possible paths after edge j). There is the precondition that the conditions are independent from each other and the edges can be combined freely.

If there are loops in the source code, then every possible number of loop repetitions is counted as a possible path through the program fragment. It is obvious that 100% path coverage in testing is not feasible for a nontrivial program.

5. Test Management

➔ Test Planning

Testing is integrated into the endeavor of creating a given product or service; each phase and each type of testing has different characteristics and what is tested in each version could be different. Hence, testing satisfies this definition of a project fully. Given that testing can be considered as a project on its own, it has to be planned, executed, tracked, and periodically reported on. We will look at the test planning aspects in the next section. We will then look into the process that drives a testing project. Subsequently, we will look at the execution of tests and the various types of reporting that takes place during a testing project. We will conclude this chapter by sharing some of the best practices in test management and execution.



1. Preparing a Test Plan

Testing—like any project—should be driven by a plan. The test plan acts as the anchor for the execution, tracking, and reporting of the entire testing project and covers

1. What needs to be tested—the scope of testing, including clear identification of what will be tested and what will not be tested.
2. How the testing is going to be performed—breaking down the testing into small and manageable tasks and identifying the strategies to be used for carrying out the tasks.
3. What resources are needed for testing—computer as well as human resources.
4. The time lines by which the testing activities will be performed.
5. Risks that may be faced in all of the above, with appropriate mitigation and contingency plans.

Failing to plan is planning to fail.

2. Scope Management: Deciding Features to be Tested/Not Tested

As was explained in the earlier chapters, various testing teams do testing for various phases of testing. One single test plan can be prepared to cover all phases and all teams or there can be separate plans for each phase or for each type of testing. For example, there needs to be plans for unit testing integration testing, performance testing, acceptance testing, and so on. They can all be part of a single plan or could be covered by multiple plans. In situations where there are multiple test plans, there should be one test plan, which covers the activities common for all plans. This is called the *master test plan.*

Scope management pertains to specifying the scope of a project. For testing, scope management entails

1. Understanding what constitutes a release of a product;
2. Breaking down the release into features;
3. Prioritizing the features for testing;
4. Deciding which features will be tested and which will not be; and
5. Gathering details to prepare for estimation of resources for testing.

It is always good to start from the end-goal or product-release perspective and get a holistic picture of the entire product to decide the scope and priority of testing. Usually, during the planning stages of a release, the *features* that constitute the release are identified. For example, a particular release of an inventory control system may introduce new features to automatically integrate with supply chain management and to provide the user with various options of costing. The testing teams should get involved early in the planning cycle and understand the features. Knowing the features and understanding them from the usage perspective will enable the testing team to prioritize the features for testing.

The following factors drive the choice and prioritization of features to be tested.

Features those are new and critical for the release The new features of a release set the expectations of the customers and must perform properly. These new features result in new program code and thus have a higher susceptibility and exposure to defects. Furthermore, these are likely to be areas where both the development and testing teams will have to go through a learning curve. Hence, it makes sense to put these features on top of the priority list to be tested. This will ensure that these key features get enough planning and learning time for testing and do not go out with inadequate testing. In order to get this prioritization right, the product marketing team and some select customers participate in identification of the features to be tested.

Features whose failures can be catastrophic Regardless of whether a feature is new or not, any feature the failure of which can be catastrophic or produce adverse business impact has to be high on the list of features to be tested. For example, recovery mechanisms in a database will always have to be among the most important features to be tested.

Features that are expected to be complex to test Early participation by the testing team can help identify features that are difficult to test. This can help in starting the work on these features early and line up appropriate resources in time.

Features which are extensions of earlier features that have been defect prone As we have seen in Chapter 8, Regression Testing, certain areas of a code tend to be defect prone and such areas need very thorough testing so that old defects do not creep in again. Such features that are defect prone should be included ahead of more stable features for testing.

A product is not just a heterogeneous mixture of these features. These features work together in various combinations and depend on several environmental factors and execution conditions. The test plan should clearly identify these combinations that will be tested.

Given the limitations on resources and time, it is likely that it will not be possible to test all the combinations exhaustively. During planning time, a test manager should also consciously identify the features or combinations that will *not* be tested. This choice should balance the requirements of time and resources while not exposing the customers to any serious defects. Thus, the test plan should contain clear justifications of why certain combinations will not be tested and what are the risks that may be faced by doing so.

3. Deciding Test Approach/Strategy

Once we have this prioritized feature list, the next step is to drill down into some more details of what needs to be tested, to enable estimation of size, effort, and schedule. This includes identifying

1. What type of testing would you use for testing the functionality?
2. What are the configurations or scenarios for testing the features?
3. What integration testing would you do to ensure these features work together?
4. What localization validations would be needed?
5. What “non-functional” tests would you need to do?

We have discussed various types of tests in earlier chapters of this book. Each of these types has applicability and usefulness under certain conditions. The test approach/strategy part of the test plan identifies the right type of testing to effectively test a given feature or combination.
The test strategy or approach should result in identifying the right type of test for each of the features or combinations. There should also be objective criteria for measuring the success of a test. This is covered in the next sub-section.

4. Setting up Criteria for Testing

As we have discussed in earlier chapters (especially chapters on system and acceptance testing) there must be clear entry and exit criteria for different phases of testing. The test strategies for the various features and combinations determined how these features and combinations would be tested. Ideally, tests must be run as early as possible so that the last-minute pressure of running tests after development delays (see the section on Risk Management below) is minimized. However, it is futile to run certain tests too early. The entry criteria for a test specify threshold criteria for each phase or type of test. There may also be entry criteria for the entire testing activity to start. The completion/exit criteria specify when a test cycle or a testing activity can be deemed complete. Without objective exit criteria, it is possible for testing to continue beyond the point of diminishing returns.
A test cycle or a test activity will not be an isolated, continuous activity that can be carried out at one go. It may have to be suspended at various points of time because it is not possible to proceed further. When it is possible to proceed further, it will have to be resumed. Suspension criteria specify when a test cycle or a test activity can be suspended. Resumption criteria specify when the suspended tests can be resumed. Some of the typical suspension criteria include

1. Encountering more than a certain number of defects, causing frequent stoppage of testing activity;
2. Hitting show stoppers that prevent further progress of testing (for example, if a database does not start, further tests of query, data manipulation, and so on are is simply not possible to execute); and
3. Developers releasing a new version which they advise should be used in lieu of the product under test (because of some critical defect fixes).

When such conditions are addressed, the tests can resume.

5. Identifying Responsibilities, Staffing, and Training Needs

Scope management identifies *what* needs to be tested. The test strategy outlines *how* to do it. The next aspect of planning is the *who* part of it. Identifying responsibilities, staffing, and training needs addresses this aspect.
A testing project requires different people to play different roles. As discussed in the previous two chapters, there are the roles of test engineers, test leads, and test managers. There is also role definition on the dimensions of the modules being tested or the type of testing. These different roles should complement each other. The different role definitions should

1. Ensure there is clear accountability for a given task, so that each person knows what he or she has to do;
2. Clearly list the responsibilities for various functions to various people, so that everyone knows how his or her work fits into the entire project;
3. Complement each other, ensuring no one steps on an others' toes; and
4. Supplement each other, so that no task is left unassigned.

Role definitions should not only address technical roles, but also list the management and reporting responsibilities. This includes frequency, format, and recipients of status reports and other project-tracking mechanisms. In addition, responsibilities in terms of SLAs for responding to queries should also be addressed during the planning stage.
Staffing is done based on estimation of effort involved and the availability of time for release. In order to ensure that the right tasks get executed, the features and tasks are prioritized the basis of on effort, time, and importance.
People are assigned to tasks that achieve the best possible fit between the requirements of the job and skills and experience levels needed to perform that function. It may not always be

possible to find the perfect fit between the requirements and the skills available. In case there are gaps between the requirements and availability of skills, they should be addressed with appropriate training programs. It is important to plan for such training programs upfront as they are usually are de-prioritized under project pressures.

6. Identifying Resource Requirements

As a part of planning for a testing project, the project manager (or test manager) should provide estimates for the various hardware and software resources required. Some of the following factors need to be considered.

1. Machine configuration (RAM, processor, disk, and so on) needed to run the product under test
2. Overheads required by the test automation tool, if any
3. Supporting tools such as compilers, test data generators, configuration management tools, and so on
4. The different configurations of the supporting software (for example, OS) that must be present
5. Special requirements for running machine-intensive tests such as load tests and performance tests
6. Appropriate number of licenses of all the software

In addition to all of the above, there are also other implied environmental requirements that need to be satisfied. These include office space, support functions (like HR), and so on.

Underestimation of these resources can lead to considerable slowing down of the testing efforts and this can lead to delayed product release and to de-motivated testing teams. However, being overly conservative and "safe" in estimating these resources can prove to be unnecessarily expensive. Proper estimation of these resources requires co-operation and teamwork among different groups—product development team, testing team, system administration team, and senior management.

7. Identifying Test Deliverables

The test plan also identifies the deliverables that should come out of the test cycle/testing activity. The deliverables include the following, all reviewed and approved by the appropriate people.

1. The test plan itself (master test plan, and various other test plans for the project)
2. Test case design specifications
3. Test cases, including any automation that is specified in the plan
4. Test logs produced by running the tests
5. Test summary reports

As we will see in the next section, a defect repository gives the status of the defects reported in a product life cycle. Part of the deliverables of a test cycle is to ensure that the defect repository is kept current. This includes entering new defects in the repository and updating the status of defect fixes after verification. We will see the contents of some of these deliverables in the later parts of this chapter.

8. Testing Tasks: Size and Effort Estimation

The scope identified above gives a broad overview of what needs to be tested. This understanding is quantified in the estimation step. Estimation happens broadly in three phases.

1. Size estimation
2. Effort estimation
3. Schedule estimation

We will cover size estimation and effort estimation in this sub-section and address schedule estimation in the next sub-section.

Size estimate quantifies the actual amount of testing that needs to be done. Several factors contribute to the size estimate of a testing project.

Size of the product under test This obviously determines the amount of testing that needs to be done. The larger the product, in general, greater is the size of testing to be done. Some of the measures of the size of product under test are as follows.

1. Lines of code (LOC) is a somewhat controversial measure as it depends on the language, style of programming, compactness of programming, and so on. Furthermore, LOC represents size estimate only for the coding phase and not for the other phases such as requirements, design, and so on. Notwithstanding these limitations, LOC is still a popular measure for estimating size.
2. A function point (FP) is a popular method to estimate the size of an application. Function points provide a representation of application size, independent of programming language. The application features (also called functions) are classified as inputs, outputs, interfaces, external data files, and enquiries. These are increasingly complex and hence are assigned increasingly higher weights. The weighted average of functions (number of functions of each type multiplied by the weight for that function type) gives an initial estimate of size or complexity. In addition, the function point methodology of estimating size also provides for 14 environmental factors such as distributed processing, transaction rate, and so on.

This methodology of estimating size or complexity of an application is comprehensive in terms of taking into account realistic factors. The major challenge in this method is that it requires formal training and is not easy to use. Furthermore, this method is not directly suited to systems software type of projects.

3. A somewhat simpler representation of application size is the number of screens, reports, or transactions. Each of these can be further classified as "simple," "medium," or "complex." This classification can be based on intuitive factors such as number of fields in the screen, number of validations to be done, and so on.

Extent of automation required When automation is involved, the size of work to be done for testing increases. This is because, for automation, we should first perform the basic test case design (identifying input data and expected results by techniques like condition coverage, boundary value analysis, equivalence partitioning, and so on.) and then scripting them into the programming language of the test automation tool.

Number of platforms and inter-operability environments to be tested If a particular product is to be tested under several different platforms or under several different configurations, then the size of the testing task increases. In fact, as the number of platforms or touch points across different environments increases, the amount of testing increases almost exponentially.
All the above size estimates pertain to "regular" test case development. Estimation of size for regression testing involves considering the changes in the product and other similar factors.

In order to have a better handle on the size estimate, the work to be done is broken down into smaller and more manageable parts called work breakdown structure (WBS) units. For a testing project, WBS units are typically test cases for a given module, test cases for a given platform, and so on. This decomposition breaks down the problem domain or the product into simpler parts and is likely to reduce the uncertainty and unknown factors.
Size estimate is expressed in terms of any of the following.

1. Number of test cases
2. Number of test scenarios
3. Number of configurations to be tested

Size estimate provides an estimate of the actual ground to be covered for testing. This acts as a primary input for estimating effort. Estimating effort is important because often effort has a more direct influence on cost than size. The other factors that drive the effort estimate are as follows.

Productivity data Productivity refers to the speed at which the various activities of testing can be carried out. This is based on historical data available in the organization. Productivity data can be further classified into the number of test cases that can be developed per day (or some unit time), the number of test cases that can be run per day, the number of pages of pages of documentation that can be tested per day, and so on. Having these fine-grained productivity data enables better planning and increases the confidence level and accuracy of the estimates.

Reuse opportunities If the test architecture has been designed keeping reuse in mind, then the effort required to cover a given size of testing can come down. For example, if the tests are designed in such a way that some of the earlier tests can be reused, then the effort of test development decreases.

Robustness of processes Reuse is a specific example of process maturity of an organization. Existence of well-defined processes will go a long way in reducing the effort involved in any activity. For example, in an organization with higher levels of process maturity, there are likely to be

1. Well-documented standards for writing test specifications, test scripts, and so on;
2. Proven processes for performing functions such as reviews and audits;
3. Consistent ways of training people; and
4. Objective ways of measuring the effectiveness of compliance to processes.

All these reduce the need to reinvent the wheel and thus enable reduction in the effort involved. Effort estimate is derived from size estimate by taking the individual WBS units and classifying them as "reusable," "modifications," and "new development." For example, if parts of a test case can be reused from existing test cases, then the effort involved in developing these would be close to zero. If, on the other hand, a given test case is to be developed fully from scratch, it is reasonable to assume that the effort would be the size of the test case divided by productivity. Effort estimate is given in person days, person months, or person years. The effort estimate is then translated to a schedule estimate. We will address scheduling in the next sub-section.

9. Activity Breakdown and Scheduling

Activity breakdown and schedule estimation entail translating the effort required into specific time frames. The following steps make up this translation.

1. Identifying external and internal dependencies among the activities
2. Sequencing the activities, based on the expected duration as well as on the dependencies
3. Identifying the time required for each of the WBS activities, taking into account the above two factors
4. Monitoring the progress in terms of time and effort
5. Rebalancing schedules and resources as necessary

During the effort estimation phase, we have identified the effort required for each of the WBS unit, factoring in the effect of reuse. This effort was expressed in terms of person months. If the effort for a particular WBS unit is estimated as, say, 40 person months, it is not possible to trade the "persons" for "months," that is, we cannot indefinitely increase the number of people working on it, expecting the duration to come down proportionally. As stated in [BROO-74], adding more people to an already delayed project is a sure way of delaying the project even further! This is because, when new people are added to a project, it increases the communication overheads and it takes some time for the new members to gel with the rest of the team. Furthermore, these WBS units cannot be executed in any random order because there will be dependencies among the activities. These dependencies can be external dependencies or internal dependencies. External dependencies of an activity are beyond the control and purview of the manager/person performing the activity. Some of the common external dependencies are

1. Availability of the product from developers;
2. Hiring;
3. Training;
4. Acquisition of hardware/software required for training; and
5. Availability of translated message files for testing.

Internal dependencies are fully within the control of the manager/person performing that activity. For example, some of the internal dependencies could be.

1. Completing the test specification
2. Coding/scripting the tests
3. Executing the tests

The testing activities will also face parallelism constraints that will further restrict the activities that can be done at a time. For example, certain tests cannot be run together because of conflicting conditions (for example, requiring different versions of a component for testing) or a high-end machine may have to be multiplexed across multiple tests.

Based on the dependencies and the parallelism possible, the test activities are scheduled in a sequence that helps accomplish the activities in the minimum possible time, while taking care of all the dependencies. This schedule is expressed in the form of a Gantt chart as shown in Figure 15.1. The colored figure is available on Illustrations.

	Task Name	Duration	Start	Finish
1	**Test schedule for release**	**33 days**	**Tue 2/22/00**	**Thu 4/6/00**
2	Test plan	2 days	Tue 2/22/00	Wed 2/23/00
3	Test strategy	2 days	Tue 2/22/00	Wed 2/23/00
4	Developing test cases	3 days	Thu 2/24/00	Mon 2/28/00
5	Review of test cases	1 day	Tue 2/29/00	Tue 2/29/00
6	Performance benchmark plan	2 days	Wed 3/1/00	Thu 3/2/00
7	Unit testing	4 days	Fri 3/3/00	Wed 3/8/00
8	Integration testing	3 days	Thu 3/9/00	Mon 3/13/00
9	Buddy testing	2 days	Tue 3/14/00	Wed 3/15/00
10	System testing	4 days	Thu 3/16/00	Tue 3/21/00
11	Regression testing	3 days	Wed 3/22/00	Fri 3/24/00
12	Performance test 1	3 days	Mon 3/27/00	Wed 3/29/00
13	Performance test 2	2 days	Thu 3/30/00	Fri 3/31/00
14	Documentation and reports	2 days	Mon 4/3/00	Tue 4/4/00
15	User acceptance test	1 day	Wed 4/5/00	Wed 4/5/00
16	**Release**	1 day	Thu 4/6/00	Thu 4/6/00

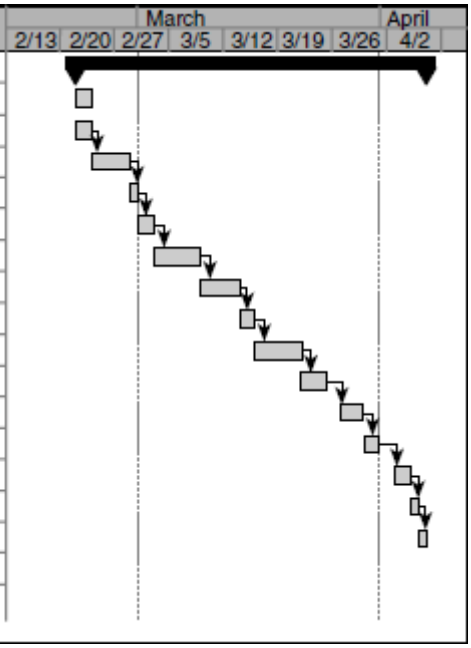


Figure 15.1 Gantt chart.

10. Communications Management

Communications management consists of evolving and following procedures for communication that ensure that everyone is kept in sync with the right level of detail. Since this is intimately connected with the test execution and progress of the testing project, we will take this up in more detail in Section 15.3when we take up the various types of reports in a test cycle.

11. Risk Management

Just like every project, testing projects also face risks. Risks are events that could potentially affect a project's outcome. These events are normally beyond the control of the project manager. As shown in Figure 15.2, risk management entails

1. Identifying the possible risks;
2. Quantifying the risks;
3. Planning how to mitigate the risks; and
4. Responding to risks when they become a reality.

As some risks are identified and resolved, other risks may surface. Hence as risks can happen any time, risk management is essentially a cycle, which goes through the above four steps repeatedly.

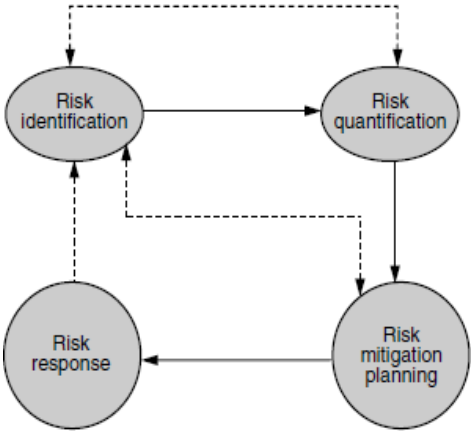


Figure 15.2 Aspects of risk management.

Risk identification consists of identifying the possible risks that may hit a project. Although there could potentially be many risks that can hit a project, the risk identification step should focus on those risks that are more likely to happen. The following are some of the common ways to identify risks in testing.

1. **Use of checklists** Over time, an organization may find new gleanings on testing that can be captured in the form of a checklist. For example, if during installation testing, it is found that a particular step of the installation has repeatedly given problems, then the checklist can have an explicit line item to check that particular problem. When checklists are used for risk identification, there is also a great risk of the checklist itself being out of date, thereby pointing to red herrings instead of risks!
2. **Use of organizational history and metrics** When an organization collects and analyzes the various metrics (see Chapter 17), the information can provide valuable insights into what possible risks can hit a project. For example, the past effort variance in testing can give pointers to how much contingency planning is required.
3. **Informal networking across the industry** The informal networking across the industry can help in identifying risks that other organizations have encountered.

Risk quantification deals with expressing the risk in numerical terms. There are two components to the quantification of risk. One is the *probability* of the risk happening and the other is the *impact* of the risk, if the risk happens. For example, the occurrence of a low-priority defect may have a high probability, but a low impact. However, a show stopper may have (hopefully!) a low probability, but a very high impact (for both the customer and the vendor organization). To quantify both these into one number, *Risk exposure* is used. This is defined as the product of risk probability and risk impact. To make comparisons easy, risk impact is expressed in monetary terms (for example, in dollars).

Risk mitigation planning deals with identifying alternative strategies to combat a risk event, should that risk materialize. For example, a couple of mitigation strategies for the risk of attrition are to spread the knowledge to multiple people and to introduce organization-wide processes and standards. To be better prepared to handle the effects of a risk, it is advisable to have multiple mitigation strategies.

When the above three steps are carried out systematically and in a timely manner, the organization would be in a better position to respond to the risks, should the risks become a reality. When sufficient care is not given to these initial steps, a project may find itself under immense pressure to react to a risk. In such cases, the choices made may not be the most optimal or prudent, as the choices are made under pressure.

The following are some of the common risks encountered in testing projects and their characteristics.

Unclear requirements The success of testing depends a lot on knowing what the correct expected behavior of the product under test is. When the requirements to be satisfied by a

product are not clearly documented, there is ambiguity in how to interpret the results of a test. This could result in wrong defects being reported or in the real defects being missed out. This will, in turn, result in unnecessary and wasted cycles of communication between the development and testing teams and consequent loss of time. One way to minimize the impact of this risk is to ensure upfront participation of the testing team during the requirements phase itself.

Schedule dependence The schedule of the testing team depends significantly on the schedules of the development team. Thus, it becomes difficult for the testing team to line up resources properly at the right time. The impact of this risk is especially severe in cases where a testing team is shared across multiple-product groups or in a testing services organization. A possible mitigation strategy against this risk is to identify a backup project for a testing resource. Such a backup project may be one of that could use an additional resource to speed up execution but would not be unduly affected if the resource were not available. An example of such a backup project is chipping in for speeding up test automation.

Insufficient time for testing Throughout the book, we have stressed the different types of testing and the different phases of testing. Though some of these types of testing—such as white box testing—can happen early in the cycle, most of the tests tend to happen closer to the product release. For example, system testing and performance testing can happen only after the entire product is ready and close to the release date. Usually these tests are resource intensive for the testing team and, in addition, the defects that these tests uncover are challenging for the developers to fix. As discussed in performance testing chapter, fixing some of these defects could lead to changes in architecture and design. Carrying out such changes into the cycle may be expensive or even impossible. Once the developers fix the defects, the testing team would have even lesser time to complete the testing and is under even greater pressure. The use of the V model to at least shift the test design part of the various test types to the earlier phases of the project can help in anticipating the risks of tests failing at each level in a better manner. This in turn could lead to a reduction in the last-minute crunch. The metric *days needed for release* when captured and calculated properly, can help in planning the time required for testing better.

"Show stopper" defects When the testing team reports defects, the development team has to fix them. Certain defects which are show stoppers may prevent the testing team to proceed further with testing, until development fixes such show stopper defects. Encountering this type of defects will have a double impact on the testing team: Firstly, they will not be able to continue with the testing and hence end up with idle time. Secondly, when the defects do get fixed and the testing team restarts testing, they would have lost valuable time and will be under tremendous pressure with the deadline being nearer. This risk of show stopper defects can pose a big challenge to scheduling and resource utilization of the testing teams. The mitigation strategies for this risk are similar to those seen on account of dependence on development schedules.

Availability of skilled and motivated people for testing People Issues in Testing, hiring and motivating people in testing is a major challenge. Hiring, retaining, and constant skill upgrade of testers in an organization is vital. This is especially important for testing functions because of the tendency of people to look for development positions.

Inability to get a test automation tool Manual testing is error prone and labor intensive. Test automation, as discussed in Chapter 16, alleviates some of these problems. However, test automation tools are expensive. An organization may face the risk of not being able to afford a test automation tool. This risk can in turn lead to less effective and efficient testing as well as more attrition. One of the ways in which organizations may try to reduce this risk is to develop in-house tools. However, this approach could lead to an even greater risk of having a poorly written or inadequately documented in-house tool.

These risks are not only potentially dangerous individually, but even more dangerous when they occur in tandem. Unfortunately, often, these risks do happen in tandem! A testing group plans its schedules based on development schedules, development schedules slip, testing team resources get into an idle time, pressure builds, schedules slip, and the vicious cycle starts all

over again. It is important that these risks be caught early or before they create serious impact on the testing teams. Hence, we need to identify the symptoms for each of these risks. These symptoms and their impacts need to be tracked closely throughout the project.
Table 15.1 gives typical risks, their symptoms, impacts and mitigation/contingency plans.

Table 15.1 Typical risks, symptoms, impact, and mitigation plans.

Risks	Symptoms	Impacts	Mitigation/contingency plans
Development delay	• Code drops from development going through constant slippage	• Less time becomes available for testing • Pushes product release date forward	• Constant involvement of the testing team in development plans • Periodic and timely communication • Staggering the testing activities by following the modified V model (Chapter 2)
Show stopper defects	• The test cycle gets suspended/resumed often	• Wasted/idle testing resources • Pressure on the testing teams when testing resumes after the defects are fixed • Possible pushing of the schedule forward	• Having clear exit criteria for development before a product can be accepted for testing • Making the testing team perform other functions during the wait time
Unclear requirements	• Defects getting uncovered by customers after the product has passed all the internal tests	• Rework that may extend all the way back to requirements gathering • Customer dissatisfaction because of product not meeting requirements	• Early user/product marketing involvement in development of prototypes to better elicit requirements • Clear definitions of acceptance criteria • Stringent approval cycles for the requirements
Insufficient time for testing	• Constantly overworked test engineers • Time spent on testing is a small fraction of the overall product life cycle	• Defects seeping out to customers • Attrition in the testing team	• Distributing the testing activities throughout the life cycle of the product • Automating testing activities • Getting upfront consensus for time schedules from all stakeholders
Over cautiousness in testing	• Insignificant defects getting reported • Testing team becoming a bottleneck for release	• Resources on testing not producing a good "bang for the buck"	• Setting objective exit criteria for testing
Lack of skilled people for testing	• Trend of constant request for transfers out of testing function into other functions (for example, development) • Higher attrition in testing teams vis-à-vis other teams	• Poor quality of testing, resulting in more defects escaping the testing net to the customers • Reduced credibility for the testing team	• Periodic training and skill upgradation • Showing career paths and demonstrating role models for testing • Job rotation among development, testing, and support teams
Lack of automation tools	• Long hours spent on manual testing	• Wasted efforts in manual testing • Dissatisfaction among test engineers	• Demonstration of success stories of using automation

Testing should not be the only measure for quality assurance (QA). It should be used in combination with other quality assurance measures. Therefore, an overall plan for quality assurance is needed that should be documented in the quality assurance plan.

6.2.1 Quality Assurance Plan

Guidelines for structuring the quality assurance plan can be found in IEEE standard 730-2002 [IEEE 730-2002]. The following subjects shall be considered (additional sections may be added as required. Some of the material may also appear in other documents).

Contents of a Software Quality Assurance Plan as defined in IEEE 730-2002:

1. Purpose
2. Reference documents
3. Management
4. Documentation
5. Standards, practices, conventions, and metrics
6. Software reviews
7. Test
8. Problem reporting and corrective action
9. Tools, techniques, and methodologies
10. Media control
11. Supplier control
12. Records collection, maintenance, and retention
13. Training
14. Risk management
15. Glossary
16. SQA Plan Change Procedure and History

a. IEEE Standard 730 in its new form from 2013 [IEEE 730-2013] has a new title, Standard for Software Quality Assurance Processes, and does not contain a standard layout for a software quality assurance plan anymore.

During quality assurance planning, the role the tests play as special, analytical measures of quality control is roughly defined. The details are then determined during test planning and documented in the test plan.

6.2.2 Test Plan

A task as extensive as testing requires careful planning. This planning and test preparation starts as early as possible in the software project. The test policy of the organization and the objectives, risks, and constraints of the project as well as the criticality of the product influence the test plan.

The test manager might participate in the following planning activities:

Test planning activities

■ Defining the overall approach to and strategy for testing.
■ deciding about the test environment and test automation
■ defining the test levels and their interaction, and integrating the testing activities with other project activities
■Deciding how to evaluate the test results
■ Selecting metrics for monitoring and controlling test work, as well as defining test exit criteria
■ Determining how much test documentation shall be prepared and determining templates
■ Writing the test plan and deciding on what, who, when, and how much testing
■ Estimating test effort and test costs; (re)estimating and (re)planning the testing tasks during later testing work

The results are documented in the test plan.
IEEE Standard 829-1998 [IEEE 829] provides a template.

Test Plan according to IEEE 829-1998
1. Test plan identifier
2. Introduction
3. Test items
4. Features to be tested
5. Features not to be tested
6. Approach
7. Item pass/fail criteria (test exit criteria)
8. Suspension criteria and resumption requirements
9. Test deliverables
10. Testing tasks
11. Environmental needs
12. Responsibilities
13. Staffing and training needs
14. Schedule
15. Risk and contingencies
16. Approvals

This structure2 works well in practice. The sections listed will be found in real test plans in many projects in the same, or slightly modified, form. The new edition of IEEE 829-2008 [IEEE 829-2008] differentiates between "Master Test Plan" and "Level Test Plan." The overall test plan ("Master Test Plan") is required for every project. The different level test plans are optional, depending on the criticality of the product developed. An existing test plan according to IEEE 829-1998 can be changed into the structure of the master test plan in IEEE 829-2008 using mapping or a cross-reference listing. The new standard also has a different approach: There is an explicit requirement for tailoring the test documentation depending on product risks and organizational needs. The standard encourages putting some information from the plans into tools or, if necessary, other plans.

When preparing for an exam using the Foundation syllabus version 2015, IEEE Standard 829-2008, not 1998, should be studied! Test planning is a continuous activity for the test manager throughout all phases of the development project. The test plan and related plans must Be updated regularly, based on feedback from test activities and reacting to changing project risks.

6.2.3 Prioritizing Tests

Even with good planning and control, it is possible that the time and budget for the total test, or for a certain test level, are not sufficient for executing all planned test cases. In this case, it is necessary to select test cases in a suitable way. Even with a reduced number of executable test cases, it must be assured that as many as possible critical faults are found. This means test cases must be prioritized.

Prioritization rule

Test cases should be prioritized so that if any test ends prematurely, the best possible test result at that point of time is achieved.

The most important test cases first

Prioritization also ensures that the most important test cases are executed first. This way important problem can be found early.

The criteria for prioritization, and thus for determining the order of execution of the test cases, are outlined next. Which criteria are used depends on the project, the application area, and the customer requirements.

The following criteria for prioritization of test cases may be used:

Criteria for prioritization

■ The **usage frequency** of a function or the **probability** of failure in software use. If certain functions of the system are used often and they contain a fault, then the probability of this fault

leading to a failure is high. Thus, test cases for this function should have a higher priority than test cases for a less-often-used function.

■**Failure risk**. Risk is the combination (mathematical product) of severity and failure probability. The severity is the expected damage. Such risks may be, for example, that the business of the customer using the software is impaired, thus leading to financial losses for the customer. Tests that may find failures with a high risk get higher priority than tests that may find failures with low risks.

■ The **visibility** of a failure for the end user is a further criterion for prioritization of test cases. This is especially important in interactive systems. For example, a user of a city information service will feel unsafe if there are problems in the user interface and will lose confidence in the remaining information output.

■ Test cases can be chosen depending on the **priority of the requirements**. The different functions delivered by a system have different importance for the customer. The customer may be able to accept the loss of some of the functionality if it behaves wrongly. For other parts, this may not be possible.

■ Besides the functional requirements, the **quality characteristics** may have differing importance for the customer. Correct implementation of the important quality characteristics must be tested. Test cases for verifying conformance to required quality characteristics get a high priority.

■ Prioritization can also be done from the perspective of development or system architecture. Components that lead to severe consequences when they fail (for example, a crash of the system) should be tested especially intensively.

■ **Complexity** of the individual components and system parts can be used to prioritize test cases. Complex program parts should be tested more intensively because developers probably introduced more faults. However, it may happen that program parts seen as easy contain many faults because development was not done with the necessary care. Therefore, prioritization in this area should be based on experience data from earlier projects run within the organization.

■ Failures having a high **project risk** should be found early. These are failures that require considerable correction work that in turn requires special resources and leads to considerable delays of the project.

In the test plan, the test manager defines adequate priority criteria and priority classes for the project. Every test case in the test plan should get a priority class using these criteria. This helps in deciding which test cases can be left out if resource problems occur.

Where there are many defects, there are probably more

Where many faults were found before, more are present. This phenomenon occurs often in projects. To react appropriately, it must be possible to change test case priority. In the next test cycle, additional test cases should be executed for such defect-prone test objects.

Without prioritizing test cases, it is not possible to adequately allocate limited test resources. Concentration of resources on high-priority test cases is a MUST.

6.2.4 Test Entry and Exit Criteria

Defining clear test entry and exit criteria is an important part of test planning. They define when testing can be started and stopped (totally or within a test level).

Test start criteria

Here are typical criteria, or checkpoints, that need to be fulfilled before executing the planned tests:

■ The test environment is ready.
■ The test tools are ready for use in the test environment.
■ Test objects are installed in the test environment.
■ the necessary test data is available.

These criteria are preconditions for starting test execution. They prevent the test team from wasting time trying to run tests that are not ready.

Exit criteria

Exit criteria are used to make sure test work is not stopped by chance or prematurely. They prevent tests from ending too early, for example, because of time pressure or because of resource shortages. But they also prevent testing from being too extensive. Here are some typical exit criteria and corresponding metrics or indicators:

■ Achieved test coverage: Tests run, covered requirements, code coverage, etc.
■ Product quality: Defect density, defect severity, failure rate, and reliability of the test object
■ Residual risk: Tests not executed, defects not repaired, incomplete coverage of requirements or code, etc.
■ Economic constraints: Allowed cost, project risks, release deadlines, and market chances

The test manager defines the project-specific test exit criteria in the test plan. During test execution, these criteria are then regularly measured and evaluated and serve as the basis for decisions by test and project management.

➔ Test Management

In the previous section, we considered testing as a project in its own right and addressed some of the typical project management issues in testing. In this section, we will look at some of the aspects that should be taken care of in planning such a project. These planning aspects are proactive measures that can have an across-the-board influence on all testing projects.

15.3.1 Choice of Standards

Standards comprise an important part of planning in any organization. Standards are of two types—external standards and internal standards. External standards are standards that a product should comply with, are externally visible, and are usually stipulated by external consortia. From a testing perspective, these standards include standard tests supplied by external consortia and acceptance tests supplied by customers. Compliance to external standards is usually mandated by external parties.

Internal standards are standards formulated by a testing organization to bring in consistency and predictability. They standardize the processes and methods of working within the organization. Some of the internal standards include

1. Naming and storage conventions for test artifacts;
2. Document standards;
3. Test coding standards; and
4. Test reporting standards.

Naming and storage conventions for test artifacts Every test artifact (test specification, test case, test results, and so on) have to be named appropriately and meaningfully. Such naming conventions should enable

1. Easy identification of the product functionality that a set of tests are intended for; and
2. Reverse mapping to identify the functionality corresponding to a given set of tests.

As an example of using naming conventions, consider a product P, with modules **M01, M02,** and **M03.** The test suites can be named as **PM01nnnn.<file type>.** Here **nnnn** can be a running sequence number or any other string. For a given test, different files may be required. For example, a given test may use a test script (which provides details of the specific actions to be performed), a recorded keystroke capture file, an expected results file. In addition, it may require other supporting files (for example, an SQL script for a database). All these related files can have the same file name (for example, **PMO1nnnn)** and different file types (for example, **.sh, .SQL, .KEY, .OUT**). By such a naming convention, one can find

- All files relating to a specific test (for example, by searching for all files with file name **PM01nnnn),** and
- All tests relating to a given module (for example, those files starting with name **PM01** will correspond to tests for module **M01)** those

With this, when the functionality corresponding to module **M01** changes, it becomes easy to locate tests that may have to be modified or deleted.

This two-way mapping between tests and product functionality through appropriate naming conventions will enable identification of appropriate tests to be modified and run when product functionality changes.

In addition to file-naming conventions, the standards may also stipulate the conventions for directory structures for tests. Such directory structures can group logically related tests together (along with the related product functionality). These directory structures are mapped into a configuration management repository (discussed later in the chapter).

Documentation standards Most of the discussion on documentation and coding standards pertain to automated testing. In the case of manual testing, documentation standards correspond to specifying the user and system responses at the right level of detail that is consistent with the skill level of the tester.

While naming and directory standards specify how a test entity is represented externally, documentation standards specify how to capture information about the tests within the test scripts themselves. Internal documentation of test scripts are similar to internal documentation of program code and should include the following.

1. Appropriate header level comments at the beginning of a file that outlines the functions to be served by the test.
2. Sufficient in-line comments, spread throughout the file, explaining the functions served by the various parts of a test script. This is especially needed for those parts of a test script that are difficult to understand or have multiple levels of loops and iterations.
3. Up-to-date change history information, recording all the changes made to the test file.

Without such detailed documentation, a person maintaining the test scripts is forced to rely only on the actual test code or script to guess what the test is supposed to do or what changes happened to the test scripts. This may not give a true picture. Furthermore, it may place an undue dependence on the person who originally wrote the tests.

Test coding standards Test coding standards go one level deeper into the tests and enforce standards on how the tests themselves are written. The standards may

1. Enforce the right type of initialization and clean-up that the test should do to make the results independent of other tests;
2. Stipulate ways of naming variables within the scripts to make sure that a reader understands consistently the purpose of a variable. (for example, instead of using generic names such as i, j, and so on, the names can be meaningful such as network_init_flag);
3. Encourage reusability of test artifacts (for example, all tests should call an initialization module init_env first, rather than use their own initialization routines); and
4. Provide standard interfaces to external entities like operating system, hardware, and so on. For example, if it is required for tests to spawn multiple OS processes, rather than have each of the tests directly spawn the processes, the coding standards may dictate that they should all call a standard function, say, create_os_process. By isolating the

external interfaces separately, the tests can be reasonably insulated from changes to these lower-level layers.

Test reporting standards Since testing is tightly interlinked with product quality, all the stakeholders must get a consistent and timely view of the progress of tests. Test reporting standards address this issue. They provide guidelines on the level of detail that should be present in the test reports, their standard formats and contents, recipients of the report, and so on. We will revisit this in more detail later in this chapter.

Internal standards provide a competitive edge to a testing organization and act as a first-level insurance against employee turnover and attrition. Internal standards help bring new test engineers up to speed rapidly. When such consistent processes and standards are followed across an organization, it brings about predictability and increases the confidence level one can have on the quality of the final product. In addition, any anomalies can be brought to light in a timely manner.

15.3.2 Test Infrastructure Management

Testing requires a robust infrastructure to be planned upfront. This infrastructure is made up of three essential elements.

1. A test case database (TCDB)
2. A defect repository
3. Configuration
4. management repository and tool

A test case database captures all the relevant information about the test cases in an organization. Some of the entities and the attributes in each of the entities in such a TCDB are given in Table 15.2.

Entity	Purpose	Attributes
Test case	Records all the "static" information about the tests	Test case ID Test case name (file name) Test case owner Associated files for the test case
Test case - Product cross-reference	Provides a mapping between the tests and the corresponding product features; enables identification of tests for a given feature	Test case ID Module ID
Test case run history	Gives the history of when a test was run and what was the result; provides inputs on selection of tests for regression runs (see Chapter 8)	Test case ID Run date Time taken Run status (success/failure)
Test case—Defect cross-reference	Gives details of test cases introduced to test certain specific defects detected in the product; provides inputs on the selection of tests for regression runs	Test case ID Defect reference # points to a record in the defect repository)

Table 15.2 Content of a test case database.

A defect repository captures all the relevant details of defects reported for a product. The information that a defect repository includes is given in Table 15.3.

Entity	Purpose	Attributes
Defect details	Records all the "static" information about the tests	Defect ID Defect priority/severity Defect description Affected product(s) Any relevant version information Environmental information (for example, OS version) Customers who encountered the problem (could be reported by the internal testing team also) Date and time of defect occurrence
Defect test details	Provides details of test cases for a given defect. Cross-references the TCDB	Defect ID Test case ID
Fix details	Provides details of fixes for a given defect; cross-references the configuration management repository	Defect ID Fix details (file changed, fix release information)
Communication	Captures all the details of the communication that transpired for this defect among the various stakeholders. These could include communication between the testing team and development team, customer communication, and so on. Provides insights into effectiveness of communication	Test case ID Defect reference # Details of communication

Table 15.3 Information in a defect repository.

The defect repository is an important vehicle of communication that influences the work flow within a software organization. It also provides the base data in arriving at several of the metrics we will discuss in Chapter 17, Metrics and Measurements. In particular, most of the metrics classified as testing defect metrics and development defect metrics are derived out of the data in defect repository.

Yet another infrastructure that is required for a software product organization (and in particular for a testing team) is a software configuration management (SCM) repository. An SCM repository also known as (CM repository) keeps track of change control and version control of all the files/entities that make up a software product. A particular case of the files/entities is test files. Change control ensures that

1. Changes to test files are made in a controlled fashion and only with proper approvals.
2. Changes made by one test engineer are not accidentally lost or overwritten by other changes.
3. Each change produces a distinct version of the file that is recreatable at any point of time.
4. At any point of time, everyone gets access to only the most recent version of the test files (except in exceptional cases).

Version control ensures that the test scripts associated with a given release of a product are *baseline d*along with the product files. Baselining is akin to taking a snapshot of the set of related files of a version, assigning a unique identifier to this set. In future, when anyone wants to recreate the environment for the given release, this label would enable him or her to do so.
TCDB, defect repository, and SCM repository should complement each other and work together in an integrated fashion as shown in Figure 15.3. For example, the defect repository links the defects, fixes, and tests. The files for all these will be in the SCM. The meta data about the modified test files will be in the TCDB. Thus, starting with a given defect, one can trace all the test cases that test the defect (from the TCDB) and then find the corresponding test case files and source files from the SCM repository.

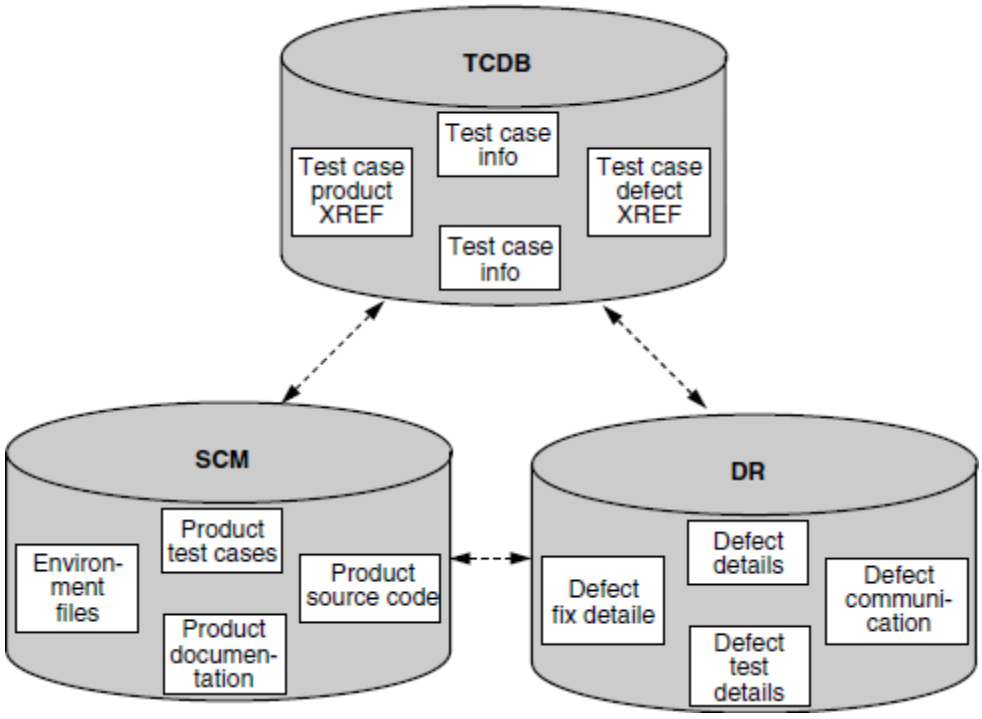


Figure 15.3 Relationships SCM, AR, and TCDB.

Similarly, in order to decide which tests to run for a given regression run,

1. The defects recently fixed can be obtained from the defect repository and tests for these can be obtained from the TCDB and included in the regression tests.
2. The list of files changed since the last regression run can be obtained from the SCM repository and the corresponding test files traced from the TCDB.
3. The set of tests not run recently can be obtained from the TCDB and these can become potential candidates to be run at certain frequencies

15.3.3 Test People Management

Developer: These testing folks… they are always nitpicking!
Tester: Why don't these developers do anything right?!
Sales person: When will I get a product out that I can sell?!

People management is an integral part of any project management. Often, it is a difficult chasm for engineers-turned-managers to cross. As an individual contributor, a person relies only on his or her own skills to accomplish an assigned activity; the person is not necessarily trained on how to document what needs to be done so that it can be accomplished by someone else. Furthermore, people management also requires the ability to hire, motivate, and retain the right people. These skills are seldom formally taught (unlike technical skills). Project managers often learn these skills in a "sink or swim" mode, being thrown head-on into the task.

Most of the above gaps in people management apply to all types of projects. Testing projects present several additional challenges. We believe that the success of a testing organization (or an individual in a testing career) depends vitally on judicious people management skills. Since the people and team-building issues are significant enough to be considered in their own right, we have covered these in detail in Chapter 13, on People Issues in Testing, and in Chapter 14, on Organization Structures for Testing Teams. These chapters address issues relevant to building and managing a good global testing team that is effectively integrated into product development and release.

We would like to stress that these team-building exercises should be ongoing and sustained, rather than be done in one burst. The effects of these exercises tend to wear out under the pressure of deadlines of delivery and quality. Hence, they need to be periodically recharged. The important point is that the common goals and the spirit of teamwork have to be internalized by all the stakeholders. Once this internalization is achieved, then they are unlikely to be swayed by operational hurdles that crop up during project execution. Such an internalization and upfront team building has to be part of the planning process for the team to succeed.

15.3.4 Integrating with Product Release

Ultimately, the success of a product depends on the effectiveness of integration of the development and testing activities. These job functions have to work in tight unison between themselves and with other groups such as product support, product management, and so on. The schedules of testing have to be linked directly to product release. Thus, project planning for the entire product should be done in a holistic way, encompassing the project plan for testing and development. The following are some of the points to be decided for this planning.

1. Sync points between development and testing as to when different types of testing can commence. For example, when integration testing could start, when system testing could start and so on. These are governed by objective entry criteria for each phase of testing (to be satisfied by development).
2. Service level agreements between development and testing as to how long it would take for the testing team to complete the testing. This will ensure that testing focuses on finding relevant and important defects only.
3. Consistent definitions of the various priorities and severities of the defects. This will bring in a shared vision between development and testing teams, on the nature of the defects to focus on.
4. Communication mechanisms to the documentation group to ensure that the documentation is kept in sync with the product in terms of known defects, workarounds, and so on.

The purpose of the testing team is to identify the defects in the product and the risks that could be faced by releasing the product with the existing defects. Ultimately, the decision to

release or not is a management decision, dictated by market forces and weighing the business impact for the organization and the customers.

➔ Test Process

Testing is a process rather than a single activity. This process starts from test planning then designing **test cases**, preparing for execution and evaluating status till the test closure. So, we can divide the activities within the fundamental test process into the following basic steps:

1) Planning and Control
2) Analysis and Design
3) Implementation and Execution
4) Evaluating exit criteria and Reporting
5) Test Closure activities

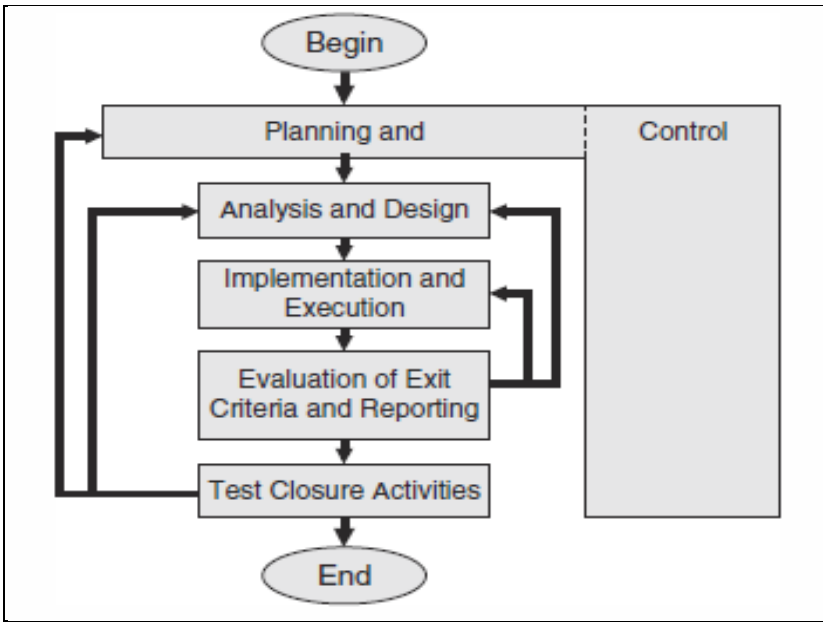
1) Planning and Control:

Test planning has following major tasks:

i. To determine the scope and **risks** and identify the objectives of testing.
ii. To determine the test approach.
iii. To implement the test policy and/or the **test strategy**. (Test strategy is an outline that describes the testing portion of the **software development cycle**. It is created to inform PM, testers and developers about some key issues of the testing process. This includes the testing objectives, method of testing, total time and resources required for the project and the testing environments.).
iv. To determine the required test resources like people, test environments, PCs, etc.
v. To schedule test analysis and design tasks, test implementation, execution and evaluation.
vi. To determine the **Exit criteria** we need to set criteria such as **Coverage criteria.** (Coverage criteria are the percentage of statements in the software that must be executed during testing. This will help us track whether we are completing test activities correctly. They will show us which tasks and checks we must complete for a particular level of testing before we can say that testing is finished.)

Test control has the following major tasks:

i. To measure and analyze the results of reviews and testing.
ii. To monitor and document progress, **test coverage** and exit criteria.
iii. To provide information on testing.
iv. To initiate corrective actions.
v. To make decisions.



2) Analysis and Design:

Test analysis and Test Design has the following major tasks:

i. To review the **test basis.** (The test basis is the information we need in order to start the test analysis and create our own test cases. Basically it's a documentation on which test cases are based, such as requirements, design specifications, product risk analysis, architecture and interfaces. We can use the test basis documents to understand what the system should do once built.)

ii. To identify test conditions.

iii. To design the tests.

iv. To evaluate testability of the requirements and system.

v. To design the test environment set-up and identify and required infrastructure and tools.

3) Implementation and Execution:

During test implementation and execution, we take the test conditions into **test cases** and procedures and other **testware** such as scripts for automation, the test environment and any other test infrastructure. (Test cases are a set of conditions under which a tester will determine whether an application is working correctly or not.)

(Testware is a term for all utilities that serve in combination for testing a software like scripts, the test environment and any other test infrastructure for later reuse.)

Test implementation has the following major task:

i. To develop and prioritize our test cases by using techniques and create **test data** for those tests. (In order to test a software application you need to enter some data for testing most of the features. Any such specifically identified data which is used in tests is known as test data.)

We also write some instructions for carrying out the tests which is known as **test procedures.**

We may also need to automate some tests using **test harness** and automated tests scripts. (A test harness is a collection of software and test data for testing a program unit by running it under different conditions and monitoring its behavior and outputs.)

ii. To create test suites from the test cases for efficient test execution.

(Test suite is a collection of test cases that are used to test a software program to show that it has some specified set of behaviors. A test suite often contains detailed instructions and information for each collection of test cases on the system configuration to be used during testing. Test suites are used to group similar test cases together.)

iii. To implement and verify the environment.

Test execution has the following major task:

i. To execute test suites and individual test cases following the test procedures.

ii. To re-execute the tests that previously failed in order to confirm a fix. This is known as **confirmation testing or re-testing.**

iii. To log the outcome of the test execution and record the identities and versions of the software under tests. The **test log** is used for the audit trial. (A test log is nothing but, what are the test cases that we executed, in what order we executed, who executed that test cases and what is the status of the test case (pass/fail). These descriptions are documented and called as test log.).

iv. To Compare actual results with expected results.

v. Where there are differences between actual and expected results, it report discrepancies as Incidents.

4) Evaluating Exit criteria and Reporting:

Based on the risk assessment of the project we will set the criteria for each test level against which we will measure the "enough testing". These criteria vary from project to project and are known as **exit criteria**.

Exit criteria come into picture, when:

— Maximum test cases are executed with certain pass percentage.

— Bug rate falls below certain level.

— When achieved the deadlines.

Evaluating exit criteria has the following major tasks:
i. To check the test logs against the exit criteria specified in test planning.
ii. To assess if more test are needed or if the exit criteria specified should be changed.
iii. To write a test summary report for stakeholders.

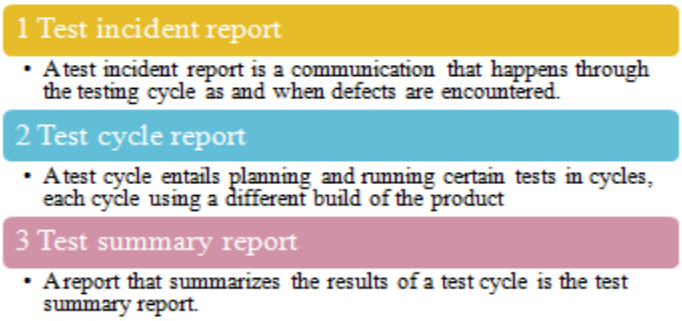
5) Test Closure activities:
Test closure activities are done when software is delivered. The testing can be closed for the other reasons also like:

- When all the information has been gathered which are needed for the testing.
- When a project is cancelled.
- When some target is achieved.
- When a maintenance release or update is done.

Test closure activities have the following major tasks:
i. To check which planned deliverables are actually delivered and to ensure that all incident reports have been resolved.
ii. To finalize and archive testware such as scripts, test environments, etc. for later reuse.
iii. To handover the testware to the maintenance organization. They will give support to the software.
iv To evaluate how the testing went and learn lessons for future releases and projects.

➔ Test Reporting

Testing requires constant communication between the test team and other teams (like the development team). Test reporting is a means of achieving this communication. There are two types of reports or communication that are required: test incident reports and test summary reports (also called test completion reports).



15.5.1.1 Test incident report

A test incident report is a communication that happens through the testing cycle as and when defects are encountered. Earlier, we described the defect repository. A test incident report is nothing but an entry made in the defect repository. Each defect has a unique ID and this is used to identify the incident. The high impact test incidents (defects) are highlighted in the test summary report.

15.5.1.2 Test cycle report

As discussed, test projects take place in units of test cycles. A test cycle entails planning and running certain tests in cycles, each cycle using a different build of the product. As the product progresses through the various cycles, it is to be expected to stabilize. A test cycle report, at the end of each cycle, gives

1. A summary of the activities carried out during that cycle;
2. Defects that were uncovered during that cycle, based on their severity and impact;
3. Progress from the previous cycle to the current cycle in terms of defects fixed;
4. Outstanding defects that are yet to be fixed in this cycle; and

5. Any variations observed in effort or schedule (that can be used for future planning).

15.5.1.3 Test summary report

The final step in a test cycle is to recommend the suitability of a product for release. A report that summarizes the results of a test cycle is the test summary report.

There are two types of test summary reports.

1. Phase-wise test summary, which is produced at the end of every phase
2. Final test summary reports (which has all the details of all testing done by all phases and teams, also called as "release test report")

A summary report should present

1. A summary of the activities carried out during the test cycle or phase
2. Variance of the activities carried out from the activities planned. This includes
   a. the tests that were planned to be run but could not be run (with reasons);
   b. modifications to tests from what was in the original test specifications (in this case, the TCDB should be updated);
   c. additional tests that were run (that were not in the original test plan);
   d. differences in effort and time taken between what was planned and what was executed; and
   e. any other deviations from plan.
3. Summary of results should include
   a. tests that failed, with any root cause descriptions; and
   b. severity of impact of the defects uncovered by the tests.

4 .Comprehensive assessment and recommendation for release should include

   a. "Fit for release" assessment; and
   b. Recommendation of release.

➔ Incident Management –

To ensure reliable and fast elimination of failures detected by the various test levels, a well-functioning procedure for communicating and managing those incident reports is needed. Incident management starts during test execution or upon test cycle completion by evaluating the test log.

✓ Test Log

Test log analysis

After each test run, or at the latest upon completion of a test cycle, the test logs are evaluated. Real results are compared to the expected results. If the test was automated, the tool will normally do this comparison immediately. Each significant, unexpected event that occurred during testing could be an indication of a test objects malfunctioning. Corresponding passages in the test log are analyzed. The testers ascertain whether a deviation from the predicted outcome really has occurred or whether an incorrectly designed test case, incorrect test automation, or incorrect test execution caused the deviation (testers, too, can make mistakes).

Documenting incidents

If the test object caused the problem, a defect or incident report is created. This is done for every unexpected behavior or observed deviation from the expected results found in the test log. An observation may be a duplicate of an observation recorded earlier. In this case, it should be checked to see whether the second observation yields additional information, which may make it possible to more easily search for the cause of the problem. Otherwise, to prevent duplication of an incident record, the same incident should not be recorded a second time.

Cause analysis is a developer task

However, the testers do not have to investigate the cause of a recorded incident. This (*debugging)* is the developers' responsibility.

✓ Incident Reporting

In general, a central database is established for each project, in which all incidents and failures discovered during testing (and possibly during operation) are registered and managed. All personnel involved in development as well as customers and users can report incidents. These reports can refer to problems in the tested (parts of) programs as well as to faults in specifications, user manuals, or other documents.

Incident reporting is also referred to as problem, anomaly, defect, or failure reporting. Not every incident or problem is due to a developer mistake. *Incident reporting* sounds less like an "accusation." Incident reporting is not a one-way street because every developer can comment on reports— for example, by requesting comments or clarification from a tester or by rejecting an unjustified report. Should a developer correct a test object, the corrections will also be documented in the incident database. This enables the responsible tester to understand this correction's implications in order to retest it in the following test cycle.

At any point in time, the incident database enables the test manager and the project manager to get an up-to-date and complete picture of the number and state of problems and about the progress of corrections. For this purpose, the database should offer appropriate possibilities for reporting and analysis.

Table 6–1 *Incident report template*

	Attribute	Meaning
Identification	Id / Number	Unique identifier/number for each report
	Test object	Identifier or name of the test object
	Version	Identification of the exact version of the test object
	Platform	Identification of the HW/SW platform or the test environment where the problem occurs
	Reporting person	Identification of the reporting tester (possibly with test level)
	Responsible developer	Name of the developer or the team responsible for the test object
	Reporting date	Date and possibly time when the problem was observed
Classification	Status	The current state (and complete history) of processing for the report (section 6.6.4)
	Severity	Classification of the severity of the problem (section 6.6.3)
	Priority	Classification of the priority of correction (section 6.6.3)
	Requirement	Pointer to the (customer-) requirements which are not fulfilled due to the problem
	Problem source	The project phase, where the defect was introduced (analysis, design, programming); useful for planning process improvement measures
Problem description	Test case	Description of the test case (name, number) or the steps necessary to reproduce the problem
	Problem description	Description of the problem or failure that occurred; expected vs. actual observed results or behavior
	Comments	List of comments on the report from developers and other staff involved
	Defect correction	Description of the changes made to correct the defect
	References	Reference to other related reports

Standardized reporting format

To allow for smooth communication and to enable statistical analysis of the incident reports, every report must follow a project-wide unique report template. The test manager should define this template and reporting structure in, for example, the test plan.

In addition to the description of the problem, the incident report typically contains information identifying the tested software, test environment, name of the tester, and defect class and prioritization as well as other information that's important for reproducing and localizing the fault. Table 6-1 shows an example of an incident report template.

A similar, slightly less complex structure can be found in [IEEE 829]. Or a report can include many additional attributes and more detail, as shown in [IEEE 1044].

If the incident database is used in acceptance testing or product support, additional customer data must be collected. The test manager has to develop a template or scheme suitable for the particular project.

Document all information relevant to reproduction and correction

In doing so, it is important to collect all information necessary for reproducing and localizing a potential fault as well as information enabling analysis of product quality and correction progress. Irrespective of the scheme agreed upon, the following rule must be observed: Each report must be written in such a way that the responsible developer can identify the problem with minimal effort and find its cause as fast as possible. Reproducing problems, localizing the cause of problems, and repairing faults are usually unplanned extra work for developers. Thus, the tester has the task of "selling" the incident report to the developers. In this situation, it is very tempting for developers to ignore or postpone analysis and repair of problems, which are unclearly described or difficult to understand.

✓ Classification

Defect Classification

An important criterion for managing a reported problem is its severity, that is, how far product use is impaired. The degree of severity will certainly be different for 100 open defect reports concerning system crashes in the database than it would be with layout errors in windows. Severity can be classified using the classes given in table 6-2.

Table 6–2*Failure severity*

Class	Description
1 – FATAL	System crash, possibly with loss of data. The test object cannot be released in this form.
2 – VERY SERIOUS	Essential malfunctioning; requirements not adhered to or incorrectly implemented; substantial impairment to many stakeholders. The test object can only be used with severe restrictions (difficult or expensive workaround).
3 – SERIOUS	Functional deviation or restriction ("normal" failure); requirement incorrectly or only partially implemented; substantial impairment to some stakeholders. The test object can be used with restrictions.
4 – MODERATE	Minor deviation; modest impairment to few stakeholders. System can be used without restrictions.
5 – MILD	Mild impairment to few stakeholders; system can be used without restrictions. For example, spelling errors or wrong screen layout.

The severity of a problem should be assigned from the point of view of the user or future user of the test object. The classifications in table 6-2, however, do not indicate how quickly a particular problem should be corrected. Priority associated with handling the problem (→failure priority) is a different matter and should not be confused with severity! When determining the priority of

corrections, additional requirements defined by product or project management (for example, correction complexity), as well as requirements about further test execution (blocked tests), must be taken into account. Therefore, the question of how quickly a fault should be corrected is answered by an additional attribute, *fault priority* (or rather, *correction priority*). Table 6-3 presents a possible classification.

***Table 6–3** Fault priority*

Priority	Description
1 – IMMEDIATE	The user's business or working process is blocked or the running tests cannot be continued. The problem requires immediate, or if necessary, provisional repair (→"patch").
2 – NEXT RELEASE	The correction will be implemented in the next regular product release or with the delivery of the next (internal) test object version.
3 – ON OCCASION	The correction will take place when the affected system parts are due for a revision anyway.
4 – OPEN	Correction planning has not taken place yet.

Incident analysis for controlling the test process

Analyzing the severity and priority of reported incidents allows the test manager to make statements about product robustness or deliverability. Apart from test status determination and clarification of questions relating to how many faults were found, how many of them are corrected, and how many are still to be corrected, trend analyses are important. This means making predictions based on the analysis of the trend of incoming incident reports over time. In this context, the most important question is whether the volume of product problems still increases or whether the situation seems to improve.

Incident analysis for improving the test process

Data from incident reports can also be used to improve the test process; for example, a comparison of data from several test objects can demonstrate which test objects show an especially small number of faults. This could mean a lack of tests or that the program has been implemented especially carefully.

✓ Status

Test management not only has a responsibility to make sure incidents are collected and documented properly but is additionally responsible (in cooperation with project management) for enabling and supporting rapid fault correction and delivery of improved versions of the test object.

This necessitates continuous monitoring of the defect analysis and correction process. For this purpose the incident status is used. Every incident report (see table 6-1) passes a series of predefined states, covering all steps from original reporting to successful defect resolution. Table 6-4 shows an example for an incident status scheme. Figure 6-2 demonstrates this procedure.

Only the tester may set the state to "Closed"

A crucial fact that is often ignored is that only the tester may set the state to "Closed" and not the developer! And this should happen only after the repeated test (retest) has proven that the Problem described in the problem report does not occur anymore. Should new failures occur as side effects after bugs are fixed, these failures should be reported in new incident reports.

Figure 6–2 *Incident status model*

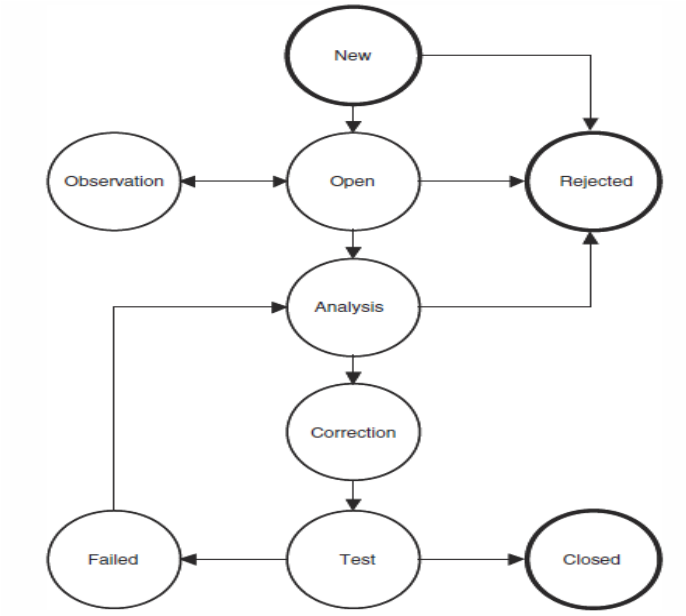


Table 6–4*Incident status scheme*

Status (set by)	**Description**
New (Tester)	A new report was written. The person reporting has included a sensible description and classification.
Open (Test manager)	The test manager regularly checks the new reports on comprehensibility and complete description of all necessary attributes. If necessary, attributes will be adjusted to ensure a project-wide uniform assessment. Duplicates or obviously useless reports are adjusted or rejected. The report is assigned to a responsible developer and its status is set to "Open."
Rejected (Test manager)	Duplicated or clearly wrong or unjustified incidents are rejected (no fault in the test object, request for change not taken into account).
Analysis (Developer)	As soon as the responsible developer starts processing this report, the status is set to "Analysis." The result of the analysis (cause, possible remedies, estimated correction effort, etc.) will be documented in comments.
Observation (Developer)	The incident described can neither be reconstructed nor be eliminated. The report remains outstanding until further information/insights are available.
Correction (Project manager)	Based on the analysis, the project manager decides if correction should take place and therefore sets the status to "Correction." The responsible developer performs the corrections and documents the kind of corrections done using comments.
Test (Developer)	As soon as the responsible developer has corrected the problem from his point of view, the report is set to "Test" status. The new software version containing this correction is identified.
Closed (Tester)	Reports carrying the status "Test" are verified in the next test cycle. For this purpose, at least the test cases, which discovered the problem, are repeated. Should the test confirm that the repair was successful, the tester finishes the report-history by setting the final status "Closed."
Failed (Tester)	Should the repeated test show that the attempt to repair was unsuccessful or insufficient, the status is set to "Failed" and a repeated analysis becomes necessary.

The scheme described previously can be applied to many projects. However, the model must be tailored to cover existing or necessary decision processes in the project. In the basic model, all decisions lie with one single person. In larger-scale projects, groups make the decisions. The decision processes grow more complex because representatives of many stakeholders must be heard.

Change control board

In many cases, changes to be done by the developers are not really fault corrections, but real (functional) enhancements. Because the distinction between "incident report" and "enhancement request" and the rating as "justified" or "not justified" is often a matter of opinion, an institution accepting or rejecting incident reports and →change requests is needed.

This institution, called the *change control board*, usually consists of representatives from the following stakeholders: product management, project management, test management, and the customer.

6. Test Automation

It is absolutely necessary for any testing organization to move forward to become more efficient, in particular in the direction of test automation. The reasons for automating test cases are given in Table 12.13. It is important to think about automation as a strategic business activity. A strategic activity requires senior management support; otherwise it will most likely fail due to lack of funding. It should be aligned with the business mission and goals and a desire to speed up delivery of the system to the market without compromising quality. However, automation is a long-term investment; it is an on-going process. It cannot be achieved overnight; Expectation need to be managed to ensure that it is realistically achievable within a certain time period.

TABLE 12.13 Benefits of Automated Testing

1. Test engineer productivity
2. Coverage of regression testing
3. Reusability of test cases
4. Consistency in testing
5. Test interval reduction
6. Reduced software maintenance cost
7. Increased test effectiveness

The organization must assess and address a number of considerations before test automation can proceed. The following prerequisites need to be considered for an assessment of whether or not the organization is ready for test automation:

· The system is stable and its functionalities are well defined.
· The test cases to be automated are unambiguous.
· The test tools and infrastructure are in place.
· The test automation professionals have prior successful experience with automation.
· Adequate budget has been allocated for the procurement of software tools.

The system must be stable enough for automation to be meaningful. If the system is constantly changing or frequently crashing, the maintenance cost of the automated test suite will be rather high to keep the test cases up to date with the system. Test automation will not succeed unless detailed test procedures are in place. It is very difficult to automate a test case which is not well defined to be manually executed. If the tests are executed in an ad hoc manner without developing the test objectives, detailed test procedure, and pass−fail criteria, then they are not ready for automation. If the test cases are designed as discussed in Chapter 11, then automation is likely to be more successful.

The test engineers should have significant programming experience. It is not possible to automate tests without using programming languages, such as Tcl (Tool command language), C, Perl, Python, Java, and Expect. It takes months to learn a programming language. The development of an automation process will fail if the testers do not have the necessary programming skills or are reluctant to develop it.

Adding temporary contractors to the test team in order to automate test cases may not work. The contractors may assist in developing test libraries but will not be able to maintain an automated test suite on an on-going basis. Adequate budget should be available to purchase and maintain new software and hardware tools to be used in test automation. The organization should keep aside funds to train the staff with new software and hardware tools. Skilled professionals with good automation background may need to be added to the test team in order to carry out the test automation project. Therefore, additional head count should be budgeted by the senior executive of the organization.

➔ Design and Architecture for Automation

We have seen several types of testing and how test cases can be developed for those testing types. When these test cases are run and checked, the coverage and quality of testing

(and the quality of the product) will definitely improve. However, this throws up a challenge that additional time is required to run those test cases. One way to overcome that challenge is to automate running of most of the test cases that are repetitive in nature.
Developing software to test the software is called test automation. Test automation can help address several problems.
Automation saves time as software can execute test cases faster than human do This can help in running the tests overnight or unattended. The time thus saved can be used effectively for test engineers to
1. Develop additional test cases to achieve better coverage;
2. Perform some esoteric or specialized tests like ad hoc testing; or
3. Perform some extra manual testing.
The time saved in automation can also be utilized to develop additional test cases, thereby improving the coverage of testing. Moreover, the automated tests can be run overnight, saving the elapsed time for testing, thereby enabling the product to be released frequently.

Test automation can free the test engineers from mundane tasks and make them focus on more creative tasks We read about ad hoc testing in Chapter 10. This testing requires intuition and creativity to test the product for those perspectives that may have been missed out by planned test cases. If there are too many planned test cases that need to be run manually and adequate automation does not exist, then the test team may spend most of its time in test execution. This creates a situation where there is no scope for intuition and creativity in the test team. This also creates fatigue and boredom in the test team. Automating the more mundane tasks gives some time to the test engineers for creativity and challenging tasks. As we saw in Chapter 13, People Issues in Testing, motivating test engineers is a significant challenge and automation can go a long way in helping this cause.

Automated tests can be more reliable When an engineer executes a particular test case many times manually; there is a chance for human error or a bias because of which some of the defects may get missed out. As with all machine-oriented activities, automation can be expected to produce more reliable results every time, and eliminates the factors of boredom and fatigue.

Automation helps in immediate testing Automation reduces the time gap between development and testing as scripts can be executed as soon as the product build is ready. Automation can be designed in such a way that the tests can be kicked off automatically, after a successful build is over. Automated testing need not wait for the availability of test engineers.

Automation can protect an organization against attrition of test engineers Automation can also be used as a knowledge transfer tool to train test engineers on the product as it has a repository of different tests for the product. With manual testing, any specialized knowledge or “undocumented ways” of running tests gets lost with the test engineer's leaving. On the other hand, automating tests makes the test execution less person dependent.

Test automation opens up opportunities for better utilization of global resources Manual testing requires the presence of test engineers, but automated tests can be run round the clock, twenty-four hours a day and seven days a week. This will also enable teams in different parts of the world, in different time zones, to monitor and control the tests, thus providing round the-clock coverage.

Certain types of testing cannot be executed without automation Test cases for certain types testing such as reliability testing, stress testing, load and performance testing, cannot be executed without automation. For example, if we want to study the behavior of a system with thousands of users logged in, there is no way one can perform these tests without using automated tools.
Automation makes the software to test the software and enables the human effort to be spent on creative testing.

Automation means end-to-end, not test execution alone Automation does not end with developing programs for the test cases. In fact, that is where it starts. Automation should consider all activities such as picking up the right product build, choosing the right configuration, performing installation, running the tests, generating the right test data,

analyzing the results, and filing the defects in the defect repository. When talking about automation, this large picture should always be kept in mind.

Automation should have scripts that produce test data to maximize coverage of permutations and combinations of inputs and expected output for result comparison. They are called *test data generators.* It is not always easy to predict the output for all input conditions. Even if all input conditions are known and the expected results are met, the error produced by the software and the functioning of the product after such an error may not be predictable. The automation script should be able to map the error patterns dynamically to conclude the result. The error pattern mapping is done not only to conclude the result of a test, but also to point out the root cause. The definition of automation should encompass this aspect.

While it is mentioned in the above paragraphs, that automation should try to cover the entire spectrum of activities for testing, it is important for such automation to relinquish the control back to test engineers in situations where a further set of actions to be taken are not known or cannot be determined automatically (for example, if test results cannot be ascertained by test scripts for pass/fail). If automated scripts determine the results wrongly, such conclusions can delay the product release. If automated scripts are found to have even one or two such problems, irrespective of their quality, they will lose credibility and may not get used during later cycles of the release and the team will depend on manual testing. This may lead the organization to believe that the automation is the cause of all problems. This in turn will result in automation being completely ignored by the organization. Hence automation not only should try to cover the entire operation of activities but also should allow human intervention where required.

Once tests are automated keeping all the requirements in mind, it becomes easier to transfer or rotate the ownership for running the tests. As the objective of testing is to catch defects early, the automated tests can be given to developers so that they can execute them as part of unit testing. The automated tests can also be given to a CM engineer (build engineer) and the tests can be executed soon after the build is ready. As we discussed in Chapter 8, on regression tests, many companies have a practice of *Daily build and smoke test,* to ensure that the build is ready for further testing and that existing functionality is not broken due to changes.

Terms Used In Automation

We have been using the term "test case" freely in this book. As formally defined in Chapter 15, a *test case* is a set of sequential steps to execute a test operating on a set of predefined inputs to produce certain expected outputs. There are two types of test cases—automated and manual. As the names suggest, a manual test case is executed manually while an automated test case is executed using automation. Test cases in this chapter refer to automated test cases, unless otherwise specified. A test case should always have an expected result associated when executed.

A test case (manual or automated) can be represented in many forms. It can be documented as a set of simple steps, or it could be an assertion statement or a set of assertions. An example of an assertion is "Opening a file, which is already opened should fail." An assertion statement includes the expected result in the definition itself, as in the above example, and makes it easy for the automation engineer to write the code for the steps and to conclude the correctness of result of the test case.

As we have seen earlier, testing involves several phases and several types of testing. Some test cases are repeated several times during a product release because the product is built several times. Not only are the test cases repetitive in testing, some operations that are described as steps in the test cases too are repetitive. Some of the basic operations such as "log in to the system" are generally performed in a large number of test cases for a product. Even though these test cases (and the operations within them) are repeated, every time the intent or area of focus may keep changing. This presents an opportunity for the automation code to be reused for different purposes and scenarios.

Table 16.1 describes some test cases for the log in example, on how the log in can be tested for different types of testing.

Table 16.1 same test cases being used for different types of testing

. **S.No.**	**Test cases for testing**	**Belongs to what type of testing**
1	Check whether log in works	Functionality
2	Repeat log in operation *in a loop for 48 hours*	Reliability
3	Perform log in *from 10000 clients*	Load/stress testing
4	*Measure time taken* for log in operations in different conditions	Performance
5	Run log in operation from a *machine running Japanese language*	Internationalization

Table 16.1 can be further extended for other types of testing. From the above example, it is obvious that certain operations of a product (such as log in) get repeated when we try to test the product for different types of testing. If this is kept in mind, the code written for automating the log in operation can be reused in many places, thereby saving effort and time.

If we closely go through the above table, one can observe that there are two important dimensions: "*What operations have to be tested*," and "*how the operations have to be tested.*" The how portion of the test case is called *scenarios* (shown in *Italics* in the above table). "What an operation has to do" is a product-specific feature and "how they are to be run" is a *framework*-specific requirement. The framework-/test tool-specific requirements are not just for test cases or products. They are generic requirements for all products that are being tested in an organization.

The automation belief is based on the fact that product operations (such as log in) are repetitive in nature and by automating the basic operations and leaving the different scenarios (how to test) to the framework/test tool, great progress can be made. This ensures code re-use for automation and draws a clear boundary between "what a test suite has to do" and "what a framework or a test tool should complement." When scenarios are combined by basic operations of the product, they become automated test cases.

When a set of test cases is combined and associated with a set of scenarios, they are called "test suite." A Test suite is nothing but a set of test cases that are automated and scenarios that are associated with the test cases.

➔ Test Automation-

✓ Design and Architecture for Automation

Design and architecture is an important aspect of automation. As in product development, the design has to represent all requirements in modules and in the interactions between modules. As we have seen in Chapter 5, Integration Testing, both internal interfaces and external interfaces have to be captured by design and architecture. In Figure 16.2, the thin arrows represent the internal interfaces and the direction of flow and thick arrows show the external interfaces. All the modules, their purpose, and interactions between them are described in the subsequent sections. The colored figure is available on Illustrations.

Architecture for test automation involves two major heads: a test infrastructure that covers a test case database and a defect database or defect repository. These are shown as external modules in Figure 16.2. Using this infrastructure, the test framework provides a backbone that ties the selection and execution of test cases.

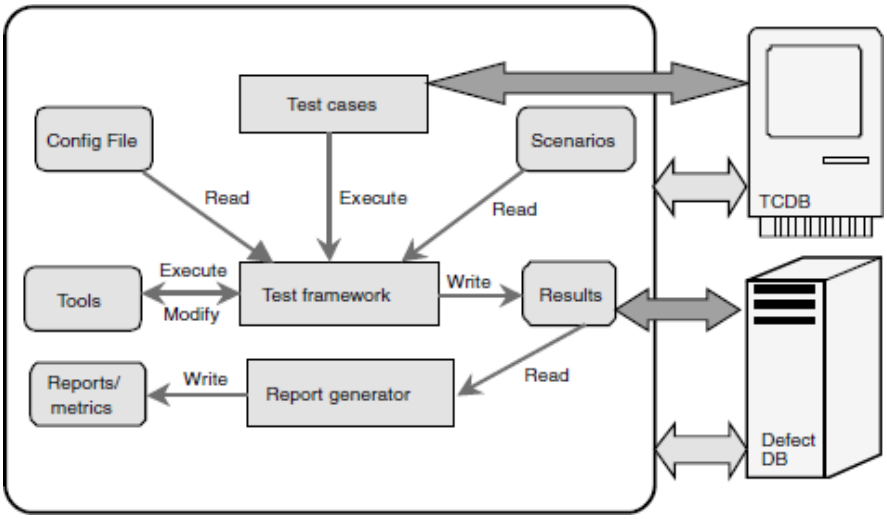
16.5.1 External Modules

There are two modules that are external modules to automation—TCDB and defect DB. We have described details of these in Chapter 15. To recall, all the test cases, the steps to

execute them, and the history of their execution (such as when a particular test case was run and whether it passed/failed) are stored in the TCDB. The test cases in TCDB can be manual or automated. The interface shown by thick arrows represents the interaction between TCDB and the automation framework only for automated test cases. Please note that manual test cases do not need any interaction between the framework and TCDB.

Defect DB or *defect database* or *defect repository* contains details of all the defects that are found in various products that are tested in a particular organization. It contains defects and all the related information (when the defect was found, to whom it is assigned, what is the current status, the type of defect, its impact, and so on). Test engineers submit the defects for manual test cases. For automated test cases, the framework can automatically submit the defects to the defect DB during execution.

These external modules can be accessed by any module in automation framework, not just one or two modules. In Figure 16.2, the "*green*" thick arrows show specific interactions and "*blue*" thick arrows show multiple interactions.



16.5.2 Scenario and Configuration File Modules

As we have seen in earlier sections, *scenarios* are nothing but information on "how to execute a particular test case."

A *configuration file* contains a set of variables that are used in automation. The variables could be for the test framework or for other modules in automation such as tools and metrics or for the test suite or for a set of test cases or for a particular test case. A configuration file is important for running the test cases for various execution conditions and for running the tests for various input and output conditions and states. The values of variables in this configuration file can be changed dynamically to achieve different execution, input, output, and state conditions.

16.5.3 Test Cases and Test Framework Modules

A *test case* in Figure 16.2 means the automated test cases that are taken from TCDB and executed by the framework. Test case is an object for execution for other modules in the architecture and does not represent any interaction by itself.

A *test framework* is a module that combines "what to execute" and "how they have to be executed." It picks up the specific test cases that are automated from TCDB and picks up the scenarios and executes them. The variables and their defined values are picked up by the test framework and the test cases are executed for those values.

The test framework is considered the core of automation design. It subjects the test cases to different scenarios. For example, if there is a scenario that requests a particular test case be executed for 48 hours in a loop, then the test framework executes those test cases in the loop and times out when the duration is met. The framework monitors the results of every iteration and the results are stored. The test framework contains the main logic for interacting,

initiating, and controlling all modules. The various requirements for the test framework are covered in the next section.

A test framework can be developed by the organization internally or can be bought from the vendor. Test framework and test tool are the two terms that are used interchangeably in this chapter. To differentiate between the usage of these two terms (wherever needed), in this chapter "framework" is used to mean an internal tool developed by the organization and "test tool" is used to mean a tool obtained from a tool vendor.

16.5.4 Tools and Results Modules

When a test framework performs its operations, there are a set of tools that may be required. For example, when test cases are stored as source code files in TCDB, they need to be extracted and compiled by build tools. In order to run the compiled code, certain runtime tools and utilities may be required. For example, IP Packet Simulators or User Login Simulators or Machine Simulators may be needed. In this case, the test framework invokes all these different tools and utilities.

When a test framework executes a set of test cases with a set of scenarios for the different values provided by the configuration file, the results for each of the test case along with scenarios and variable values have to be stored for future analysis and action. The results that come out of the tests run by the test framework should not overwrite the results from the previous test runs. The history of all the previous tests run should be recorded and kept as archives. The archive of results help in executing test cases based on previous test results. For example, a test engineer can request the test framework to "execute all test cases that are failed in previous test run." The audit of all tests that are run and the related information are stored in the module of automation. This can also help in selecting test cases for regression runs, as explained in Chapter 8, Regression Testing.

16.5.5 Report Generator and Reports/Metrics Modules

Once the results of a test run are available, the next step is to prepare the test reports and metrics. Preparing reports is a complex and time-consuming effort and hence it should be part of the automation design. There should be customized reports such as an executive report, which gives very high level status; technical reports, which give a moderate level of detail of the tests run; and detailed or debug reports which are generated for developers to debug the failed test cases and the product. The periodicity of the reports is different, such as daily, weekly, monthly, and milestone reports. Having reports of different levels of detail and different periodicities can address the needs of multiple constituents and thus provide significant returns.

The module that takes the necessary inputs and prepares a formatted report is called a *report generator.* Once the results are available, the report generator can generate *metrics.*

All the reports and metrics that are generated are stored in the reports/metrics module of automation for future use and analysis.

✓ Generic Requirements for test Tool/Framework

In the previous section, we described a generic framework for test automation. We will now present certain detailed criteria that such a framework and its usage should satisfy.

While illustrating the requirements, we have used examples in a hypothetical meta language to drive home the concept. The reader should verify the availability, syntax, and semantics of his or her chosen automation tool.

Requirement 1: No hard coding in the test suite

One of the important requirements for a test suite is to keep all variables separately in a file. By following this practice, the source code for the test suite need not be modified every time it is required to run the tests for different values of the variables. This enables a person who does not know the program to change the values and run the test suite. As we saw earlier, the variables for the test suite are called *configuration variables.* The file in which all variable names and their associated values are kept is called *configuration file.* It is quite possible that there

could be several variables in the configuration file. Some of them could be for the test tool and some of them for the test suite. The variables belonging to the test tool and the test suite need to be separated so that the user of the test suite need not worry about test tool variables. Moreover, inadvertently changing test tool variables, without knowing their purpose, may impact the results of the tests. Providing inline comment for each of the variables will make the test suite more usable and may avoid improper usage of variables. An example of such a well-documented configuration file is provided below.

 # Test framework Configuration Parameters
 # WARNING: DO NOT MODIFY THIS SET WITHOUT CONSULTING YOUR SYADMIN

TOOL_PATH=/tools	Path for test tool
COMMONLIB_PATH=/tools/crm/lib	Common Library Functions
SUITE_PATH=/tools/crm	Test suite path

 # Parameters common to all the test cases in the test suite

VERBOSE_LEVEL=3	Messaging Level to screen
MAX_ERRORS=200	Maximum allowable errors before the
	Test suite exits
USER_PASSWD=hello123	System administrator password

 # Test Case 1 parameters

TC1_USR_CREATE=0	Whether users to be created, 1 = yes, 0 = no
TC1_USR_PASSWD=hello1	User Password
TC1_USR_PREFI×=user	User Prefix
TC1_MAX_USRS=200 #	Maximum users

In a product scenario involving several releases and several test cycles and defect fixes, the test cases go through large amount of changes and additionally there are situations for the new test cases to be added to the test suite. Test case modification and new test case insertion should not result in the existing test cases failing. If such modifications and new test cases result in the quality of the test suite being impacted, it defeats the purpose of automation, and maintenance requirements of the test suite become high. Similarly, test tools are not only used for one product having one test suite. They are used for various products that may have multiple test suites. In this case, it is important for the test suites be added to the framework without affecting other test suites. To summarize

- Adding a test case should not affect other test cases
- Adding a test case should not result in retesting the complete test suite
- Adding a new test suite to the framework should not affect existing test suites

Requirement 2: Test case/suite expandability

As we have seen in the “log in” example, the functionality of the product when subjected to different scenarios becomes test cases for different types of testing. This encourages the reuse of code in automation. By following the objectives of framework and test suite to take care of the “how” and “what” portions of automation respectively, reuse of test cases can be increased. The reuse of code is not only applicable to various types of testing; it is also applicable for modules within automation. All those functions that are needed by more than one test case can be separated and included in libraries. When writing code for automation, adequate care has to be taken to make them modular by providing functions, libraries and including files. To summarize

1. The test suite should only do what a test is expected to do. The test framework needs to take care of “how,” and
2. The test programs need to be modular to encourage reuse of code.

Requirement 3: Reuse of code for different types of testing, test cases

For each test case there could be some prerequisite to be met before they are run. The test cases may expect some objects to be created or certain portions of the product to be configured in a particular way. If this portion is not met by automation, then it introduces some manual intervention before running the test cases. When test cases expect a particular setup to run the tests, it will be very difficult to remember each one of them and do the setup accordingly in the manual method. Hence, each test program should have a "setup" program that will create the necessary setup before executing the test cases. The test framework should have the intelligence to find out what test cases are executed and call the appropriate setup program.

Requirement 4: Automatic setup and cleanup

A setup for one test case may work negatively for another test case. Hence, it is important not only to create the setup but also "undo" the setup soon after the test execution for the test case. Hence, a "cleanup" program becomes important and the test framework should have facilities to invoke this program after test execution for a test case is over.

We discussed test case expandability in requirement 2. The test cases need to be independent not only in the design phase, but also in the execution phase. To execute a particular test case, it should not expect any other test case to have been executed before nor should it implicitly assume that certain other test case will be run after it. Each test case should be executed alone; there should be no dependency between test cases such as test case-2 to be executed after test case-1 and so on. This requirement enables the test engineer to select and execute any test case at random without worrying about other dependencies.

Requirement 5: Independent test cases

Contrary to what was discussed in the previous requirement, sometimes there may be a need for test cases to depend on others. Making test cases independent enables any one case to be selected at random and executed. Making a test case dependent on an other makes it necessary for a particular test case to be executed before or after a dependent test case is selected for execution. A test tool or a framework should provide both features. The framework should help to specify the dynamic dependencies between test cases.

Requirement 6: Test case dependency

Insulating test cases from the environment is an important requirement for the framework or test tool. At the time of test case execution, there could be some events or interrupts or signals in the system that may affect the execution. Consider the example of automatic pop-up screens on web browsers. When such pop-up screens happen during execution, they affect test case execution as the test suite may be expecting some other screen based on an earlier step in the test case.

Requirement 7: Insulating test cases during execution

Hence, to avoid test cases failing due to some unforeseen events, the framework should provide an option for users to block some of the events. There has to be an option in the framework to specify what events can affect the test suite and what should not.

Coding standards and proper directory structures for a test suite may help the new engineers in understanding the test suite fast and help in maintaining the test suite. Incorporating the coding standards improves portability of the code. The test tool should have an option to specify (and sometimes to force) coding standards such as POSIX, XPG3, and so on. The test framework should provide an option or force the directory structure to enable multiple programmers to develop test suites/test cases in parallel, without duplicating the parts of the test case and by reusing the portion of the code.

Requirement 8: Coding standards and directory structure

A framework may have multiple test suites; a test suite may have multiple test programs; and a test program may have multiple test cases. The test tool or a framework should

have a facility for the test engineer to select a particular test case or a set of test cases and execute them. The selection of test cases need not be in any order and any combination should be allowed. Allowing test engineers to select test cases reduces the time and limits the focus to only those tests that are to be run and analyzed. These selections are normally done as part of the scenario file. The selection of test cases can be done dynamically just before running the test cases, by editing the scenario file.

Requirement 9: Selective execution of test cases

Example:
test-program-name 2, 4, 1, 7-10
In the above scenario line, the test cases 2, 4, 1, 7, 8, 9, 10 are selected for execution in the same order mentioned. The hyphen (-) is used to mention the test cases in the chosen range—(7-10) have all to be executed. If the test case numbers are not mentioned in the above example, then the test tool should have the facility to execute all the test cases.

Requirement 10: Random execution of test cases

While it is a requirement for a test engineer to select test cases from the available test cases as discussed in requirement 8 above, the same test engineer may sometimes need to select a test case randomly from a list of test cases. Giving a set of test cases and expecting the test tool to select the test case is called random execution of test cases. A test engineer selects a set of test cases from a test suite; selecting a random test case from the given list is done by the test tool. Given below are two examples to demonstrate random execution.

Example 1:	*Example 2:*
random test-program-name 2, 1, 5	random test-programl (2, 1, 5) test-program2 test-program3

In the first example, the test engineer wants the test tool to select one out of test cases 2, 1, 5 and executed. In the second example, the test engineer wants one out of test programs 1, 2, 3 to be randomly executed and if program 1 is selected, then one out of test cases 2, 1, 5 to be randomly executed. In this example if test programs 2 or 3 are selected, then all test cases in those programs are executed.

Requirement 11: Parallel execution of test cases

There are certain defects which can be unearthed if some of the test cases are run at the same time. In a multi-tasking and multi processing operating systems it is possible to make several instances of the tests and make them run in parallel. Parallel execution simulates the behavior of several machines running the same test and hence is very useful for performance and load testing.

Example 1:	*Example 2:*
instances,5 Test-program1(3)	Time_loop, 5 hours test-program1(2, 1, 5) test-program2 test-program3

In the first example above, 5 instances of test case 3 from test programl, are created; in the second example, 5 instances of 3 test programs are created. Within each of the five instances that are created, the test programs 1, 2, 3 are executed in sequence.

Requirement 12: Looping the test cases

As discussed earlier, reliability testing requires the test cases to be executed in a loop. There are two types of loops that are available. One is the *iteration* loop which gives the number of iterations of a particular test case to be executed. The other is the *timed* loop, which keeps

executing the test cases in a loop till the specified time duration is reached. These tests bring out reliability issues in the product.

Example 1:	*Example 2:*
Repeat_loop, 50 Test-program1(3)	Time_loop, 5 hours test-program1(2, 1, 5) test-program2 test-program3

In the first example, test case 3 from test programl is repeated 50 times and in the second example, test cases 2, 1, 5 from test program1 and all test cases from test programs 2 and 3 are executed in order, in a loop for five hours.

Requirement 13: Grouping of test scenarios

We have seen many requirements and scenarios for test execution. Now let us discuss how we can combine those individual scenarios into a group so that they can run for a long time with a good mix of test cases. The group scenarios allow the selected test cases to be executed in order, random, in a loop all at the same time. The grouping of scenarios allows several tests to be executed in a predetermined combination of scenarios.
The following is an example of a group scenario.

Example:
group_scenario1
parallel, 2 AND repeat, 10 @ scenl
scen1
test_program1 (2, 1, 5) 21

test_program2
test_program3

In the above example, the group scenario was created to execute two instances of the individual scenario "scen1" in a loop for 10 times. The individual scenario is defined to execute test program1 (test cases 2, 1 and 5), test program2 and test program3. Hence, in the combined scenario, all test programs are executed by two instances simultaneously in an iteration loop for 10 times.

Requirement 14: Test case execution based on previous results

As we have seen in Chapter 8, regression test methodology requires that test cases be selected based on previous result and history. Hence, automation may not be of much help if the previous results of test execution are not considered for the choice of tests. Not only for regression testing, it is a requirement for various other types of testing also. One of the effective practices is to select the test cases that are not executed and test cases that failed in the past and focus more on them. Some of the common scenarios that require test cases to be executed based on the earlier results are
1. Rerun all test cases which were executed previously;
2. Resume the test cases from where they were stopped the previous time;
3. Rerun only failed/not run test cases; and
4. Execute all test cases that were executed on "Jan 26, 2005."
With automation, this task becomes very easy if the test tool or the framework can help make such choices.

Requirement 15: Remote execution of test cases

Most product testing requires more than one machine to be used. Hence there is a facility needed to start the testing on multiple machines at the same time from a central place. The central machine that allocates tests to multiple machines and co-ordinates the execution and result is called *test console or test monitor.* In the absence of a test console, not only does executing the results from multiple machines become difficult, collecting the results from all

those machines also becomes difficult. In the absence of a test console, the results of tests need to be collected manually and consolidated. As it is against the objective of automation to introduce a manual step, this requirement is important for the framework to have. To summarize

 It should be possible to execute/stop the test suite on any machine/set of machines from the test console.

 The test results and logs can be collected from the test console.

 The progress of testing can be found from the test console.

Figure 16.3 illustrates the role played be a test console and the multiple test machine. The colored figure is available on Illustrations.

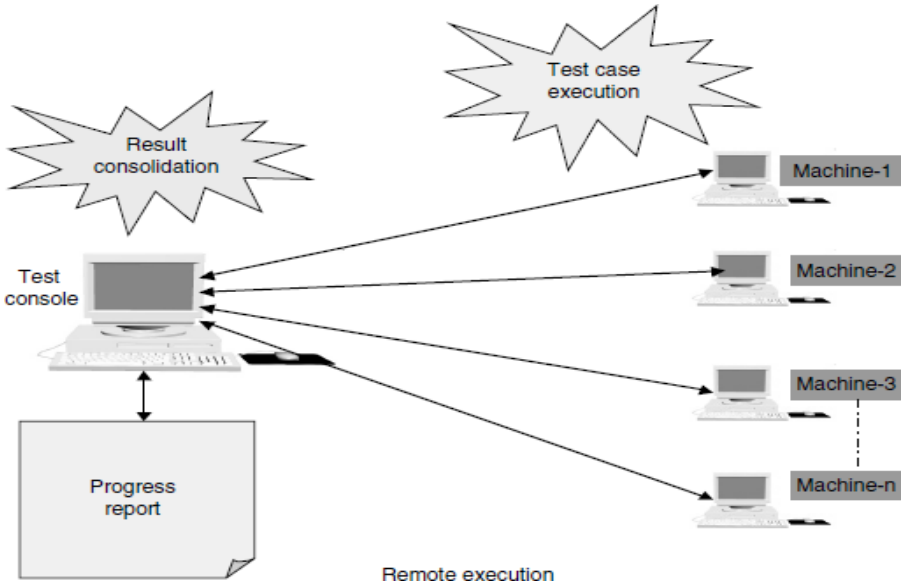


Figure 16.3 Role of test console and multiple execution machine.

In requirement 13, we have seen that test cases are repeated on the basis of previous results. To confirm the results or to reproduce the problem, it is not enough repeat the test cases. The test cases have to be repeated the same way as before, with the same scenarios, same configuration variables and values, and so on. This requires that all the related information for the test cases have to be archived. Hence, this requirement becomes very important for repeating test cases and for analysis. Archival of test data must include

1. What configuration variables were used;

2. What scenario was used; and

3. What programs were executed and from what path.

Requirement 16: Automatic archival of test data

Every test suite needs to have a reporting scheme from where meaningful reports can be extracted. As we have seen in the design and architecture of framework, the report generator should have the capability to look at the results file and generate various reports. Hence, though the report generator is designed to develop dynamic reports, it is very difficult to say what information is needed and what not. Therefore, it is necessary to store all information related to test cases in the results file.

Requirement 17: Reporting scheme

It is not only configuration variables that affect test cases and their results. The tunable parameters of the product and operating system also need to be archived to ensure repeatability or for analyzing defects.

Audit logs are very important to analyze the behavior of a test suite and a product. They store detailed information for each of the operations such as when the operation was invoked, the values for variables, and when the operation was completed, For performance tests, information such as when a framework was invoked, when a scenario was started, when a particular test case started and their corresponding completion time are important to calculate the performance of the product. Hence; a reporting scheme should include

1. When the framework, scenario, test suite, test program, and each test case were started/completed;
2. Result of each test case;
3. Log messages;
4. Category of events and log of events; and
5. Audit reports.

While coding for automation, there are some test cases which are easier coded using the scripting language provided by the tool, some in C and some in C++ and so on. Hence, a framework or test tool should provide a choice of languages and scripts that are popular in the software development area. Irrespective of the languages/scripts are used for automation, the framework should function the same way, meeting all requirements. Many test tools force the test scripts to be written in a particular language or force some proprietary scripts to be used. This needs to be avoided as it affects the momentum of automation because a new language has to be learned. To summarize

 A framework should be independent of programming languages and scripts.
 A framework should provide choice of programming languages, scripts, and their combinations.
 A framework or test suite should not force a language/script.
 A framework or test suite should work with different test programs written using different languages and scripts.
 A framework should have exported interfaces to support all popular, standard languages, and scripts.
 The internal scripts and options used by the framework should allow the developers of a test suite to migrate to better framework.

Requirement 18: Independent of languages

With the advent of platform-independent languages and technologies, there are many products in the market that are supported in multiple OS and language platforms. Products being cross-platform and test framework not working on some of those platforms are not good for automation. Hence, it is important for the test tools and framework to be cross-platform and be able to run on the same diversity of platforms and environments under which the product under test runs.

Requirement 19: Portability to different platforms

Having said that the test tools need to be cross-platform, it is important for the test suite developed for the product also be cross-platform or portable to other platforms with minimum amount of effort.

With a checklist for cross-platform capabilities of the test tools, it is important to look at platform architectures also. For example, 64-bit operating systems and products that work in 64-bit architecture are available in the market. The product being 64-bit and test suite functioning as a 32-bit application may not be a big issue, since all platforms support compatibility with legacy applications running 32 bits. However, this prevents some of the defects that are in the product from being unearthed. In a pure 64-bit environment which does not provide backward compatibility (for example, Digital Tru64), these 32-bit test suite cannot be run at all. To summarize

 The framework and its interfaces should be supported on various platforms.
 Portability to different platforms is a basic requirement for test tool/test suite.
 The language/script used in the test suite should be selected carefully so that it run on different platforms.
 The language/script written for the test suite should not contain platform-specific calls.

✓ Criteria for selecting test tools

A test automation tool is a software application that assists in the automation of test cases that would otherwise be run manually. Some tools are commercially available in the market, but for testing complex, imbedded, real-time systems, very few commercial test tools are available in the market. Therefore, most organizations build their own test automation frameworks using programming languages such as C and Tcl. It is essential to combine both hardware and software for real-time testing tools. This is due to the fact that special kinds of interface cards are required to be connected to the SUT. The computing power of personal computers with network interface cards may not be good enough to send traffic to the SUT.

Test professionals generally build their own test tools in high-technology fields, such as telecommunication equipment and application based on IP. Commercial third-party test tools are usually not available during the system testing phase. For example, there were no commercially available test tools during the testing of the 1xEv-DO system described in Chapter 8. The second author of this book developed in-house software tools to simulate access terminals using their own products. However, we advocate that testers should build their own test automation tools only if they have no alternative. Building and maintaining one's own test automation tool from scratch are time-consuming tasks and an expensive undertaking.

The test tool evaluation criteria are formulated for the selection of the right kind of software tool. There may be no tool that fulfills all the criteria. Therefore, we should be a bit flexible during the evaluation of off-the-shelf automation tools available in the market. The broad criteria for evaluating test automation tools have been classified into the following eight categories as shown in Figure 12.3.

1. *Test Development Criteria:* An automation test tool should provide a high-level, preferably nonproprietary, easy-to-use test scripting language such as Tcl. It should have the ability to interface and drive modules that can be easily written in, for example, C, Tcl, Perl, or Visual Basic. The tool must provide facility to directly access, read, modify, and control the internals of the automated test scripts. The input test data should be stored separately from the test script but easily cross-referenced to the corresponding test scripts, if necessary. The tool should have built-in templates of test scripts, test cases, tutorials, and demo application examples to show how to develop automated test cases. Finally, no changes should be made to the SUT in order to use the tool. The vendors recommended environment should match the real test laboratory execution environment.
2. *Test Maintenance Criteria:* The tool should possess a rich set of features, such as version control capability on test cases, test data, and migration of test cases across different platforms. The tool must provide powerful, easy-to-use facilities to browse, navigate, modify, and reuse the test suites. The tool should have the ability to select a subset of test cases to form a group for a particular test run based on one or more distinguishing characteristics. A tool needs to have features to allow modification and replication of test cases, easy addition of new test cases, and import from another. The tool should have the capability to add multiple tags to a test case and modify those tags so that the test case can be easily selected in a subgroup of test cases sharing a common characteristic.

3. *Test Execution Criteria:* An automation tool should allow test cases to be executed individually, as a group, or in a predefined sequence. The user should have the ability to check the interim results during the execution of a group of tests and exercise other options for the remainder of the tests based on the interim results. The user should have the option to pause and resume the execution of a test suite. The tool should have the facility to execute the test

suite over the Internet. The tool should allow simultaneous execution of several test suites that can be distributed across multiple machines for parallel execution. This substantially reduces the time needed for testing if multiple test machines are available. The test tool should have a capability for monitoring, measuring, and diagnosing performance characteristics of the SUT. Finally, the tool should have the capability to be integrated with other software tools which are either in use or expected to be used.

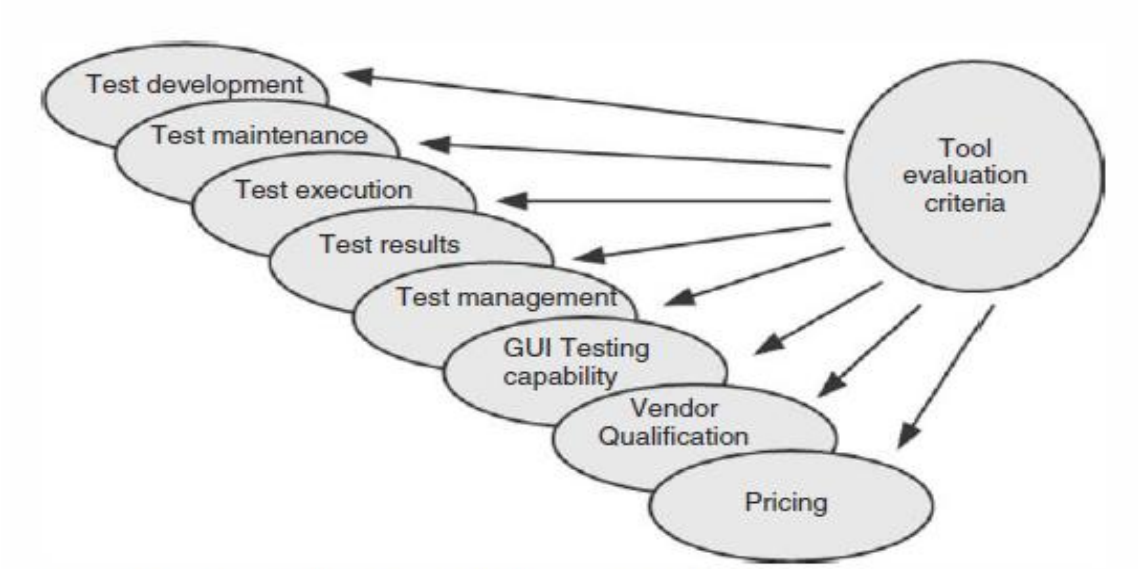


Figure 12.3 Broad criteria of test automation tool evaluation.

4. *Test Results Criteria:* The test tool must provide a flexible, comprehensive logging process during execution of the test suite, which may include detailed records of each test case, test results, time and date, and pertinent diagnostic data. A tool should have the capability to cross-reference the test results back to the right versions of test cases. The test result log can be archived in an industry standard data format and the tool should have an effective way to access and browse the archived test results. The tool should provide query capability to extract test results, analyze the test status and trend, and produce graphical reports of the test results. Finally, the tool should have the capability to collect and analyze response time and throughput as an aid to performance testing.

5. *Test Management Criteria:* A tool should have the ability to provide a test structure, or hierarchy that allows test cases to be stored and retrieved in a manner that the test organization wants to organize. The tool should have the capability to allocate tests or groups of tests to specific test engineers and compare the work status with the plan through graphic display. A tool needs to have authorization features. For example, a test script developer may be authorized to create and update the test scripts, while the test executer can only access them in the run mode. The tool should have the capability to send out emails with the test results after completion of test suite execution.

6. *GUI Testing Capability Criteria:* An automated GUI test tool should include a record/playback feature which allows the test engineers to create, modify, and run automated tests across many environments. These tools should have a capability to recognize and deal with all types of GUI objects, such as list boxes, radio buttons, icons, joysticks, hot keys, and bit-map images with changes in color shades and presentation fonts. The recording activity of the tool capturing the keystrokes entered by the test engineer can be represented as scripts in a high-level programming language and saved for future replay. The tools must allow test engineers to modify test scripts to create reusable test procedures to be played back on a new software image for comparison. The performance of a GUI test tool needs to be evaluated. One may

consider the question: How fast can the tool record and playback a complex test scenario or a group of test scenarios?

7. *Vendor Qualification Criteria:* Many questions need to be asked about the vendors financial stability, age of the vendor company, and its capability to support the tool. The vendor must be willing to fix problems that arise with the tool. A future roadmap must exist for the product. Finally, the maturity and market share of the product must be evaluated.

8. *Pricing Criteria:* Pricing is an important aspect of the product evaluation criteria. One can ask a number of questions: Is the price competitive? Is it within the estimated price range for an initial tool purchase? For a large number of licenses, a pricing discount can be negotiated with the vendor. Finally, the license must explicitly cap the maintenance cost of the test tool from year to year.

Tool vendors may guarantee the functionality of the test tool; however, experience shows that often test automation tools do not work as expected within the particular test environment. Therefore, it is recommended to evaluate the test tool by using it before making the decision to purchase it. The test team leader needs to contact the tool vendor to request a demonstration. After a demonstration of the tool, if the test team believes that the tool holds potential, then the test team leader may ask for a temporary license of the tool for evaluation. At this point enough resources are allocated to evaluate the test tool. The evaluator should have a clear understanding of the tool requirements and should make a test evaluation plan based on the criteria outlined previously. The goal here is to ensure that the test tool performs as advertised by the vendor and that the tool is the best product for the requirement. Following the hands-on evaluation process, an evaluation report is prepared. The report documents the hands-on experience with the tool. This report should contain background information, tool summary, technical findings, and a conclusion. This document is designed to address the management concerns because eventually it has to be approved by executive management.

12.12 TEST SELECTION GUIDELINES FOR AUTOMATION

Test cases should be automated only if there is a clear economic benefit over manual execution. Some test cases are easy to automate while others are more cumbersome.

The general guideline shown in Figure 12.4 may be used in evaluating the suitability of test cases to be automated as follows:

Less Volatile: A test case is stable and is unlikely to change over time. The test case should have been executed manually before. It is expected that the test steps and the pass−fail criteria are not likely to change any more.

Repeatability: Test cases that are going to be executed several times should be automated. However, one-time test cases should not be considered for automation. Poorly designed test cases which tend to be difficult to reuse are not economical for automation.

High Risk: High-risk test cases are those that are routinely rerun after every new software build. The objectives of these test cases are so important that one cannot afford to not reexecute them. In some cases the propensity of the test cases to break is very high. These test cases are likely to be fruitful in the long run and are the right candidates for automation.

Easy to Automate: Test cases that are easy to automate using automation tools should be automated. Some features of the system are easier to test than other features, based on the characteristics of a particular tool. Custom objects with graphic and sound features are likely to be more expensive to automate.

Manually Difficult: Test cases that are very hard to execute manually should be automated. Manual test executions are a big problem, for example, causing eye strain from having to look at too many screens for too long in a GUI test. It is strenuous to look at transient results in real-time applications. These nasty, unpleasant test cases are good candidates for automation.

Boring and Time Consuming: Test cases that are repetitive in nature and need to be executed for longer periods of time should be automated. The testers time should be utilized in the development of more creative and effective test cases.

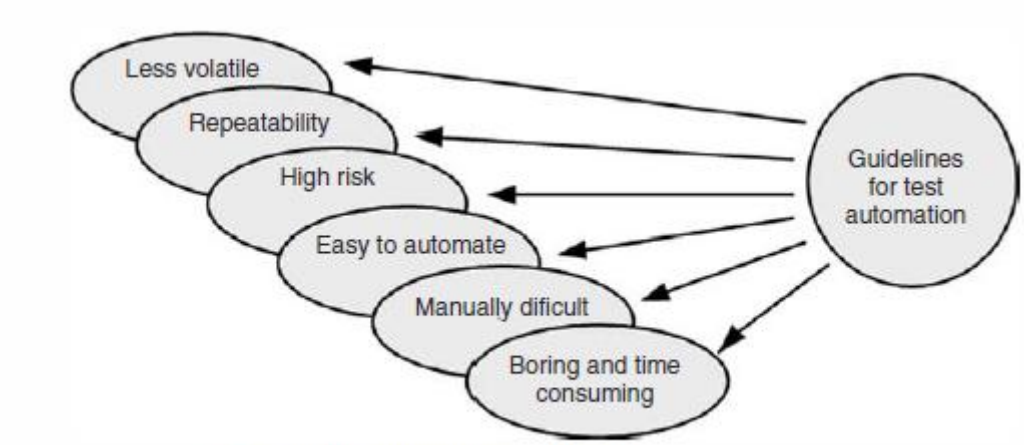


Figure 12.4 Test selection guideline for automation.

12.13 CHARACTERISTICS OF AUTOMATED TEST CASES

The largest component of test case automation is programming. Unless test cases are designed and coded properly, their execution and maintenance may not be effective. The design characteristics of effective test cases were discussed in Chapter 11. A formal model of a standard test case schema was also provided in Chapter 11. In this section, we include some key points which are pertinent to the coding of test cases. The characteristics of good automated test cases are given in Figure 12.5 and explained in the following.

1. *Simple:* The test case should have a single objective. Multiobjective test cases are difficult to understand and design. There should not be more than 10−15 test steps per test case, excluding the setup and cleanup steps. Multipurpose test cases are likely to break or give misleading results. If the execution of a complex test leads to a system failure, it is difficult to isolate the cause of the failure.

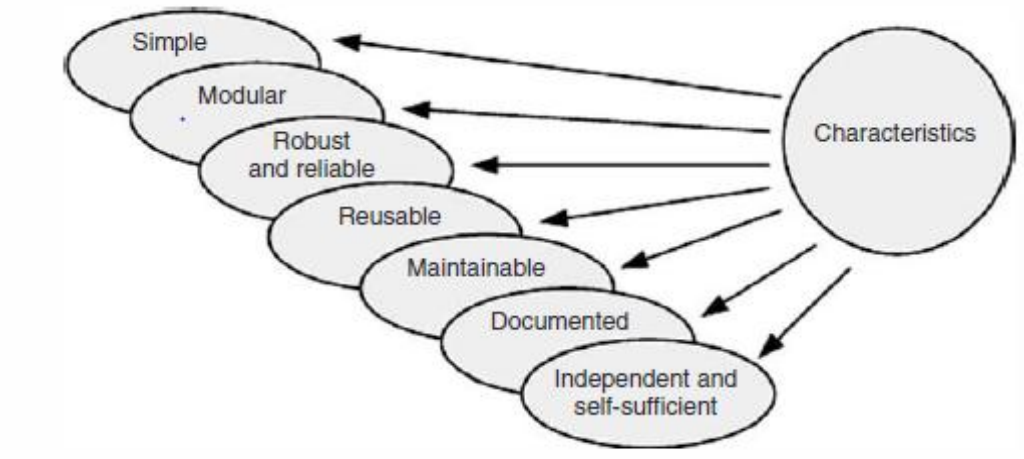


Figure 12.5 Characteristics of automated test cases.

2. *Modular:* Each test case should have a setup and cleanup phase before and after the execution test steps, respectively. The setup phase ensures that the initial conditions are met before the start of the test steps. Similarly, the cleanup phase puts the system back in the initial state, that is, the state prior to setup. Each test step should be small and precise. One input stimulus should be provided to the system at a time and the response verified (if applicable) with an interim verdict. The test steps are building blocks from reusable libraries that are put together to form multistep test cases.

3. *Robust and Reliable:* A test case verdict (pass−fail) should be assigned in such a way that it should be unambiguous and understandable. Robust test cases can ignore trivial failures such as one-pixel mismatch in a graphical display. Care should be taken so that false test results are minimized. The test cases must have built-in mechanisms to detect and recover from errors. For example, a test caseneed not wait indefinitely if the SUT has crashed. Rather, it can wait for a while and terminate an indefinite wait by using a timer mechanism.
4. *Reusable:* The test steps are built to be configurable, that is, variables should not be hard coded. They can take values from a single configurable file. Attention should be given while coding test steps to ensure that a single global variable is used, instead of multiple, decentralized, hard-coded variables. Test steps are made as independent of test environments as possible. The automated test cases are categorized into different groups so that subsets of test steps and test cases can be extracted to be reused for other platforms and/or configurations. Finally, in GUI automation hard-coded screen locations must be avoided.
5. *Maintainable:* Any changes to the SUT will have an impact on the automated test cases and may require necessary changes to be done to the affected test cases. Therefore, it is required to conduct an assessment of the test cases that need to be modified before an approval of the project to change the system. The test suite should be organized and categorized in such a way that the affected test cases are easily identified. If a particular test case is data driven, it is recommended that the input test data be stored separately from the test case and accessed by the test procedure as needed. The test cases must comply with coding standard formats. Finally, all the test cases should be controlled with a version control system.
6. *Documented:* The test cases and the test steps must be well documented. Each test case gets a unique identifier, and the test purpose is clear and understandable. Creator name, date of creation, and the last time it was modified must be documented. There should be traceability to the features and requirements being checked by the test case. The situation under which the test case cannot be used is clearly described. The environment requirements are clearly stated with the source of input test data (if applicable). Finally, the result, that is, pass or fail, evaluation criteria are clearly described.
7. *Independent and Self-Sufficient:* Each test case is designed as a cohesive entity, and test cases should be largely independent of each other. Each test case consists of test steps which are naturally linked together. The predecessor and successor of a test step within a test case should be clearly understood. It is useful to keep the following three independence rules while automating test cases:
· *Data value independent:* The possible corruption of data associated with one test case should have no impact on other test cases.
· *Failure independent:* The failure of one test case should not cause a ripple of failures among a large number of subsequent test cases.
· *Final state independent:* The state in which the environment is left by a test case should have no impact on test cases to be executed later.

We must take into consideration the characteristics outlined in this section during the development of test scripts. In addition, we must obey the syntax of the test case defined in the next section while implementing a test case.

12.14 STRUCTURE OF AN AUTOMATED TEST CASE

An automated test case mimics the actions of a human tester in terms of creating initial conditions to execute the test, entering the input data to drive the test, capturing the output, evaluating the result, and finally restoring the system back to its original state. The six major steps in an automated test case are shown in Figure 12.6. Error handling routines are incorporated in each step to increase the maintainability and stability of test cases.

Setup: The setup includes steps to check the hardware, network environment, software configuration, and that the SUT is running. In addition, all the parameters of the SUT that are specific to the test case are configured. Other variables pertinent to the test case are initialized.

Drive the Test: The test is driven by providing input data to the SUT. It can be a single step or multiple steps. The input data should be generated in such a way that the SUT can read, understand, and respond.

Capture the Response: The response from the SUT is captured and saved. Manipulation of the output data from the system may be required to extract the information that is relevant to the objective of the test case.

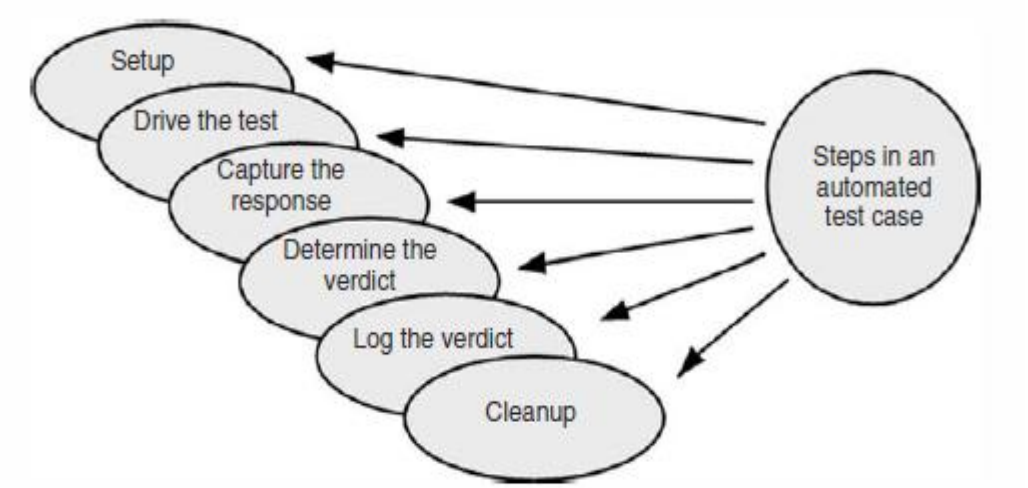


Figure 12.6 Six major steps in automated test case.

Determine the Verdict: The actual outcome is compared with the expected outcome. Predetermined decision rules are applied to evaluate any discrepancies between the actual outcomes against the expected outcome and decide whether the test result is a pass or a fail. If a fail verdict is assigned to the test case, additional diagnostic information is needed. One must be careful in designing the rules for assigning a passed/failed verdict to a test case. A failed test procedure does not necessarily indicate a problem with the SUT−the problem could be a *false positive*. Similarly, a passed test procedure does not necessarily indicate that there is no problem with the SUT−the problem could be due to a *false negative*. The problems of false negative and false positive can occur due to several reasons, such as setup errors, test procedure errors, test script logic errors, or user errors [18].

Log the Verdict: A detailed record of the results is written in a log file. If the test case failed, additional diagnostic information is needed, such as environment information at the time of failure, which may be useful in reproducing the problem later.

Cleanup: A cleanup action includes steps to restore the SUT to its original state so that the next test case can be executed. The setup and cleanup steps within the test case need to be efficient in order to reduce the overhead of test execution.

12.15 TEST AUTOMATION INFRASTRUCTURE

A test automation infrastructure, or framework, consists of test tools, equipment, test scripts, procedures, and people needed to make test automation efficient and effective. The creation and maintenance of a test automation framework is key to the success of any test automation project within an organization. The implementation of an automation framework generally requires an automation test group, as discussed in Chapter 16. The six components of a test automation framework are shown in Figure 12.7. The idea behind an automation infrastructure is to ensure the following:

· Different test tools and equipment are coordinated to work together.

· The library of the existing test case scripts can be reused for different test projects, thus minimizing the duplication of development effort.

· Nobody creates test scripts in their own ways.

· Consistency is maintained across test scripts.
· The test suite automation process is coordinated such that it is available just in time for regression testing.
· People understand their responsibilities in automated testing.

System to Be Tested: This is the first component of an automation infrastructure. The subsystems of the system to be tested must be stable; otherwise test automation will not be cost effective. As an example, the 1xEV-DO system described in Chapter 8 consists of three subsystems, BTS, BSC, and EMS. All three subsystems must be stable and work together as a whole before the start of an automation test project.

Test Platform: The test platform and facilities, that is, the network setup on which the system will be tested, must be in place to carry out the test automation project. For example, a procedure to download the image of the SUT, configuration management utilities, servers, clients, routers, switches, and hubs are necessary to set up the automation environment to execute the test scripts.

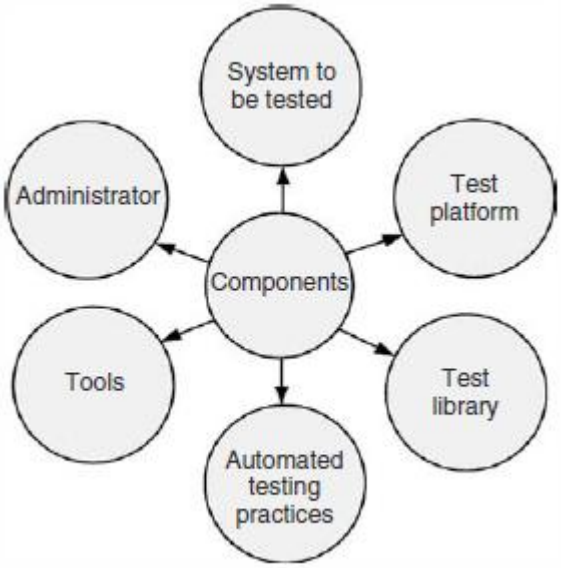


Figure 12.7 Components of automation infrastructure.

Test Case Library: It is useful to compile libraries of reusable test steps of basic utilities to be used as the building blocks of automated test scripts. Each utility typically performs a distinct task to assist the automation of test cases. Examples of such utilities are *ssh* (secure shell) from client to server, *exit* from client to server, response capture, information extraction, rules for verdicts, verdict logging, error logging, cleanup, and setup.

Automated Testing Practices: The procedures describing how to automate test cases using test tools and test case libraries must be documented. A template of an automated test case is useful in order to have consistency across all the automated test cases developed by different engineers. A list of all the utilities and guidelines for using them will enable us to have better efficiency in test automation. In addition, the maintenance procedure for the library must be documented.

Tools: Different types of tools are required for the development of test scripts. Examples of such tools are test automation tool, traffic generation tool, traffic monitoring tool, and support tool. The support tools include test factory, requirement analysis, defect tracking, and configuration management tools. Integration of test automation and support tools, such as defect tracking, is crucial for the automatic reporting of defects for failed test cases. Similarly, the test factory tool can generate automated test execution trends and result patterns.

Administrator: The automation framework administrator (i) manages test case libraries, test platforms, and test tools; (ii) maintains the inventory of templates; (iii) provides tutorials; and (iv)helps test engineers in writing test scripts using the test case libraries. In addition, the administrator provides tutorial assistance to the users of test tools and maintains a liaison with the tool vendors and the users.

- ✓ Testing of Object Oriented Systems

7. Software Quality

➔ Five Views of software quality

In the early days of computers, software developers mainly focused on product functionalities, and most of the end users were highly qualified professionals, such as mathematicians, scientists, and engineers. Development of personal computers and advances in computer networks, the World Wide Web, and graphical user interface made computer software highly accessible to all kinds of users. These days there is widespread computerization of many processes that used to be done by hand. For example, until the late 1990s taxpayers used to file returns on paper, but these days there are numerous web-based tax filing systems. There has been increasing customer expectations in terms of better quality in software products, and developers are under tremendous pressure to deliver high-quality products at a lower cost. Even though competing products deliver the same functionalities, it is the lower cost products with better quality attributes that survive in the competitive market. Therefore, all stakeholders–users, customers, developers, testers, and managers–in a product must have a broad understanding of the overall concept of software quality.

A number of factors influence the making and buying of software products. These factors are user's needs and expectations, the manufacturer's considerations, the inherent characteristics of a product, and the perceived value of a product. To be able to capture the quality concept, it is important to study quality from a broader perspective. This is because the concept of quality predates software development. In a much cited paper published in the *Sloan Management Review* [1], Garvin has analyzed how quality is perceived in different manners in different domains, namely, philosophy, economics, marketing, and management:

Transcendental View: In the transcendental view quality is something that can be recognized through experience but is not defined in some tractable form. Quality is viewed to be something ideal, which is too complex to lend itself to be precisely defined. However, a good-quality object stands out, and it is easily recognized. Because of the philosophical nature of the transcendental view, no effort is made to express it using concrete measures.

User View: The user view concerns the extent to which a product meets user needs and expectations. Quality is not just viewed in terms of what a product can deliver, but it is also influenced by the service provisions in the sales contract. In this view, a user is concerned with whether or not a product is fit for use. This view is highly personalized in nature. The idea of *operational profile*, discussed in Chapter 15, plays an important role in this view. Because of the personalized nature of the product view, a product is considered to be of good quality if it satisfies the needs of a large number of customers. It is useful to identify what product attributes users consider to be important. The reader may note that the user view can encompass many subjective elements apart from the expected functionalities central to user satisfaction. Examples of subjective elements are *usability*, *reliability* , *testability*, and *efficiency*.

Manufacturing View: The manufacturing view has its genesis in the manufacturing sectors, such as the automobile and electronics sectors. In this view, quality is seen as conforming to requirements. Any deviation from the stated requirements is seen as reducing the quality of the product. The concept of *process* plays a key role in the manufacturing view. Products are to be manufactured "right the first time" so that development cost and maintenance cost are reduced. However, there is no guarantee that conforming to process standards will lead to good products. Some criticize this

view with an argument that conformance to a process can only lead to *uniformity* in the products, and, therefore, it is possible to manufacture bad-quality products in a consistent manner. However, product quality can be incrementally enhanced by continuously improving the process. Development of the *capability maturity model* (CMM) [2] and ISO 9001 [3] are based on the manufacturing view.

Product View: The central hypothesis in the product view is this: *If a product is manufactured with good internal properties, then it will have good external qualities*. The product view is attractive because it gives rise to an opportunity to explore causal relationships between *internal properties* and *external qualities* of a product. In this view, the current quality level of a product indicates the presence or absence of measurable product properties. The product view of quality can be assessed in an objective manner. An example of the product view of software quality is that high degree of modularity, which is an internal property, makes a software testable and maintainable.

Value-Based View: The value-based view represents a merger of two independent concepts: *excellence* and *worth*. Quality is a measure of excellence, and value is a measure of worth. The central idea in the value-based view is how much a customer is willing to pay for a certain level of quality. The reality is that quality is meaningless if a product does not make economic sense. Essentially, the value-based view represents a trade-off between cost and quality.

Measuring Quality The five viewpoints help us in understanding different aspects of the quality concept. On the other hand, measurement allows us to have a quantitative view of the quality concept. In the following, we explain the reasons for developing a quantitative view of a software system [4]:

· Measurement allows us to establish baselines for qualities. Developers must know the minimum level of quality they must deliver for a product to be acceptable.

· Organizations make continuous improvements in their process models–and an improvement has a cost associated with it. Organizations need to know how much improvement in quality is achieved at a certain cost incurred due to process improvement. This causal relationship is useful in making management decisions concerning process improvement. Sometimes it may be worth investing more in process improvement, whereas some other time the return may not be significant.

· The present level of quality of a product needs to be evaluated so the need for improvements can be investigated.

Measurement of User's View The user's view encompasses a number of quality factors, such as functionality, reliability, and usability. It is easy to measure how much of the functionalities a software product delivers by designing at least one test case for each functionality. A product may require multiple test cases for the same functionality if the functionality is to be performed in different execution environments. Then, the ratio of the number of passed test cases to the total number of test cases designed to verify the functionalities is a measure of the functionalities delivered by the product. Among the qualities that reflect the users view, the concept of *reliability* has drawn the most attention of researchers.

In the ISO 9126 quality model, *usability* has been broken down into three subcharacteristics, namely, *learnability*, *understandability*, and *operability*. Learnability can be specified as the average elapsed time for a typical user to gain a certain level of competence in using the product. Similarly, understandability can be quantified as the

average time needed by a typical user to gain a certain level of understanding of the product. One can quantify operability in a similar manner. The basic idea of breaking down usability into learnability, understandability, and operability can be seen in light of Gilbs technique [5]: *The quality concept is broken down into component parts until each can be stated in terms of directly measurable attributes*. Gilbs technique is a general one to be applicable to a wide variety of user-level qualities.

Measurement of Manufacturer's View Manufacturers are interested in obtaining measures of the following two different quantities:

· **Defect Count:** How many defects have been detected?
· **Rework Cost:** How much does it cost to fix the known defects?

Defect count represents the number of all the defects that have been detected so far. If a product is in operation, this count includes the defects detected during development and operation. A defect count reflects the quality of work produced. Merely counting the defects is of not much use unless something can be done to improve the development process to reduce the defect count in subsequent projects. One can analyze the defects as follows:

· For each defect identify the development phase in which it was introduced and the phase in which it was discovered. Let us assume that a large fraction of the defects are introduced in the requirements gathering phase, and those are discovered during system testing. Then, we can conclude that requirement analysis was not adequately performed. We can also conclude that work done subsequently, such as design verification and unit testing, were not of high standard. If a large number of defects are found during system operation, one can say that system testing was not rigorously performed.
· Categorize the defects based on modules. Assuming that a module is a cohesive entity performing a well-defined task, by identifying the modules containing most of the defects we can identify where things are going wrong. This information can be used in managing resources. For example, if a large number of defects are found in a communication module in a distributed application, more resource could be allocated to train developers in the details of the communication system.
· To compare defects across modules and products in a meaningful way, normalize the defect count by product size. By normalizing defect count by product size in terms of the number of LOC, we can obtain a measure, called *defect density*. Intuitively, defect density is expressed as the *number of defects found per thousand lines of code*.
· Separate the defects found during operation from the ones found during development. The ratio of the number of defects found during operation to the total number of defects is a measure of the effectiveness of the entire gamut of test activities. If the ratio is close to zero, we can say that testing was highly effective. On the other hand, if the ratio is farther from the ideal value of zero, say, 0.2, it is apparent that all the testing activities detected only 80% of the defects.

After defects are detected, the developers make an effort to fix them. Ultimately, it costs some money to fix defects–this is apart from the "reputation" cost to an organization from defects discovered during operation. The rework cost includes all the additional cost associated with defect-related activities, such as fixing documents. Rework is an additional cost that is incurred due to work being done in a less than perfect manner the first time it was done. It is obvious that organizations strive to

reduce the total cost of software development, including the rework cost. The rework cost can be split into two parts as follows:

· **Development Rework Cost:** This is the rework cost incurred *before* a product is released to the customers.
· **Operation Rework Cost:** This is the rework cost incurred *when* a product is in operation.

On the one hand, the development rework cost is a measure of development efficiency. In other words, if the development rework cost is zero, then the development efficiency is very high. On the other hand, the operation rework cost is a measure of the delivered quality of the product in operation. If the development rework cost is zero, then the delivered quality of the product in operation is very high. This is because the customers have not encountered any defect and, consequently, the development team is not spending any resource on defect fixing.

➔ ISO 9126 Quality Characteristics

There has been international collaboration among experts to define a general framework for software quality. An expert group, under the aegis of the ISO, standardized a software quality document, namely, ISO 9126, which defines six broad, independent categories of quality characteristics as follows:

Functionality: A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.

Reliability: A set of attributes that bear on the capability of software to maintain its performance level under stated conditions for a stated period of time.

Usability: A set of attributes that bear on the effort needed for use and on the individual assessment of such use by a stated or implied set of users.

Efficiency: A set of attributes that bear on the relationship between the software's performance and the amount of resource used under stated conditions.

Maintainability: A set of attributes that bear on the effort needed to make specified modifications (which may include corrections, improvements, or adaptations of software to environmental changes and changes in the requirements and functional specifications).

Portability: A set of attributes that bear on the ability of software to be transferred from one environment to another (this includes the organizational, hardware or, software environment).

The ISO 9126 standard includes an example quality model, as shown in Figure 17.2, that further decomposes the quality characteristics into more concrete subcharacteristics. For example, the maintainability characteristic has been decomposed into four subcharacteristics, namely, *analyzability*, *changeability*, *stability*, and *testability.* The decomposition shown in Figure 17.2 is just a sample model−and not a universal one. The 20 subcharacteristics of Figure 17.2 are defined as follows:

Suitability: The capability of the software to provide an adequate set of functions for specified tasks and user objectives.

Accuracy: The capability of the software to provide the right or agreed-upon results or effects.

Interoperability: The capability of the software to interact with one or more specified systems.

Security: The capability of the software to prevent unintended access and resist deliberate attacks intended to gain unauthorized access to confidential information or to make unauthorized modifications to information or to the program so as to provide the attacker with some advantage or so as to deny service to legitimate users.

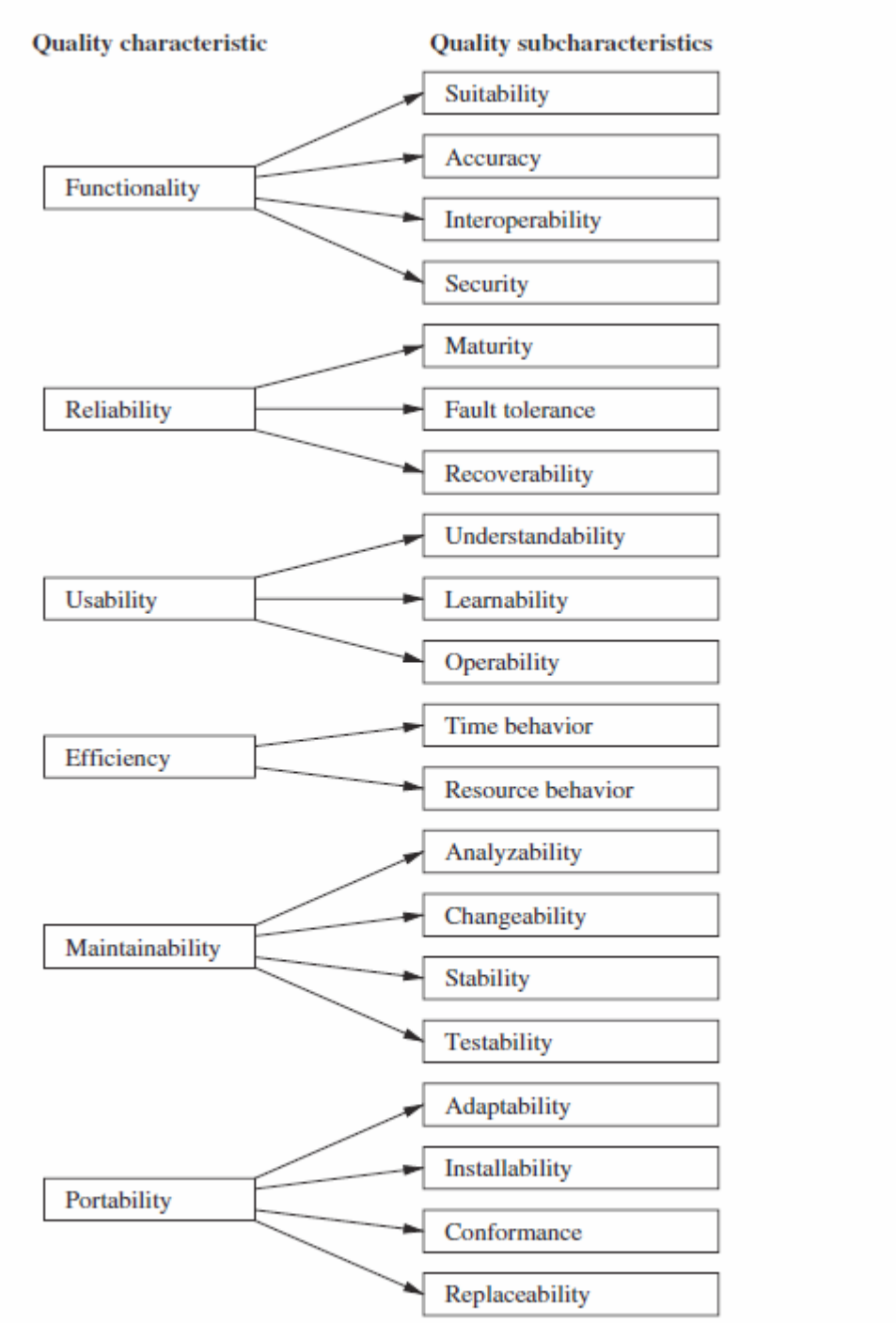


Figure 17.2 ISO 9126 sample quality model refines standard's features into subcharacteristics. (From ref. 4. © 1996 IEEE.)

Maturity: The capability of the software to avoid failure as a result of faults in the software.

Fault Tolerance: The capability of the software to maintain a specified level of performance in case of software faults or of infringement of its specified interface.

Recoverability: The capability of the software to reestablish its level of performance and recover the data directly affected in the case of a failure.

Understandability: The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use.

Learnability: The capability of the software product to enable the user to learn its applications.

Operability: The capability of the software product to enable the user to operate and control it.

Attractiveness: The capability of the software product to be liked by the user.

Time Behavior: The capability of the software to provide appropriate response and processing times and throughput rates when performing its function under stated conditions.

Resource Utilization: The capability of the software to use appropriate resources in an appropriate time when the software performs its function under stated condition.

Analyzability: The capability of the software product to be diagnosed for deficiencies or causes of failures in the software or for the parts to be modified to be identified.

Changeability: The capability of the software product to enable a specified modification to be implemented.

Stability: The capability of the software to minimize unexpected effects from modifications of the software.

Testability: The capability of the software product to enable modified software to be validated.
Adaptability: The capability of the software to be modified for different specified environments without applying actions or means other than those provided for this purpose for the software considered.

Installability: The capability of the software to be installed in a specified environment.
Coexistence: The capability of the software to coexist with other independent software in a common environment sharing common resources.

Replaceability: The capability of the software to be used in place of other specified software in the environment of that software.
Organizations must define their own quality characteristics and subcharacteristics after a fuller understanding of their needs. In other words, organizations must identify the level of the different quality characteristics they need to satisfy within their context of software development. Reaching an ideally best quality level from the present one is a gradual process. Therefore, it is important to understand the need for moving on to the next achievable step toward the highest level–the ideally best level.
At this point it is useful to compare McCall's quality model with the ISO 9126 model. Since the two models focus on the same abstract entity, namely, *software quality*, it is natural that there are many similarities between the two models. What is called *quality factor* in McCall's model is called *quality characteristic* in the ISO 9126 model. The following high-level quality factors/characteristics are found in both models: reliability, usability, efficiency, maintainability, and portability. However, there are several differences between the two models as explained in the following:

· The ISO 9126 model emphasizes characteristics *visible* to the users, whereas the McCall model considers *internal* qualities as well. For example, reusability is an internal characteristic of a product. Product developers strive to produce reusable components, whereas its impact is not perceived by customers.

· In McCall' s model, one quality criterion can impact several quality factors, whereas in the ISO 9126 model, one subcharacteristic impacts exactly one quality characteristic.

· A high-level quality factor, such as testability, in the McCall model is a low-level sub characteristic of maintainability in the ISO 9126 model.

Following are a few concerns with the quality models [4]:

· There is no consensus about what high-level quality factors are most important at the top level. McCall et al. suggest 11 high-level quality factors, whereas the ISO 9126 standard defines only 6 quality characteristics. Some of the quality factors in the McCall model are more important to developer's .For example, reusability and interoperability are important to developers. However, the ISO 9126 model just considers the product.

· There is no consensus regarding what is a top-level quality factor/ characteristic and what is more concrete quality criterion/subcharacteristics. These days many applications run on computer and communications networks. However, interoperability is not an independent, top-level quality characteristic in the ISO 9126 model. It is not clear why interoperability is a part of functionality. The absence of a rationale makes it difficult to follow a prescribed quality model.

➔ ISO 9000:2000 & Latest Software Quality Standards

There are ongoing efforts at the international level for standardizing different aspects of computer communications and software development. Standardization has been particularly successful in the field of computer networking and wireless communications. For example, the collaborative work of the Internet Engineering Task Force (IETF) has been the key to the proliferation of the Internet. Similarly, standardization efforts from the IEEE have led to the successful development of the local area network (LAN) standard, namely the IEEE 802.3 standard, and the wireless local area network (WLAN) standards, namely IEEE 802.11a/b/g.

In spite of the positive consequence of standardization in the field of communications, standardization in software development is met with mixed reactions. On the one hand, the main argument against standardization is that it curtails individual drive to be innovative. On the other hand, standards reduce the activity of reinventing the same, or similar, processes for development and quality assurance. Repeatability of processes is a key benefit emanating from standardization−and repeatability reduces the cost of software development and produces a base quality level of software products.

The ISO has developed a series of standards, collectively known as the ISO 9000. The ISO was founded in 1946, and it is based in Geneva, Switzerland. It develops and promotes international standards in the field of quality assurance and quality management. The ISO 9000 standards are generally applicable to all tangible products manufactured with human endeavor, say, from spices to software−Even some brands of spice and rice used in everyday cooking are claimed to be ISO 9000 certified. The ISO 9000 standards are reviewed and updated from time to time, once every 5−8 years. The latest ISO 9000 standards released in the year 2000 are referred to as ISO 9000:2000. There are three components of the ISO

9000:2000 standard as follows:
ISO 9000 : Fundamentals and vocabulary [7]
ISO 9001 : Requirements [8]
ISO 9004 : Guidelines for performance improvements [9]

At this point we remind the reader that ISO 9002 and ISO 9003 were parts of ISO 9000:1994, but these are no longer parts of ISO 9000:2000. ISO 9002 dealt with the quality system model

for quality assurance in production and installation, whereas ISO 9003 dealt with the quality system model for quality assurance in final inspection and testing.

17.4.1 ISO 9000:2000 Fundamentals

The ISO 9000:2000 standard is based on the following eight principles:

· **Principle 1. Customer Focus:** Success of an organization is highly dependent on satisfying the customers. An organization must understand its customers and their needs on a continued basis. Understanding the customers helps in understanding and meeting their requirements. It is not enough to just meet customer requirements. Rather, organizations must make an effort to exceed customer expectations. By understanding the customers, one can have a better understanding of their real needs and their unstated expectations. People in different departments of an organization, such as marketing, software development, testing, and customer support, must capture the same view of the customers and their requirements. An example of customer focus is to understand how they are going to use a system. By accurately understating how customers are going to use a system, one can produce a better *user profile*.

· **Principle 2. Leadership:** Leaders set the direction their organization should take, and they must effectively communicate this to all the people involved in the process. All the people in an organization must have a coherent view of the organizational direction. Without a good understanding of the organizational direction, employees will find it difficult to know where they are heading. Leaders must set challenging but realistic goals and objectives. Employee contribution should be recognized by the leaders. Leaders create a positive environment and provide support for the employees to collectively realize the organizational goal. They reevaluate their goals on a continual basis and communicate the findings to the staff.

· **Principle 3. Involvement of People:** In general, organizations rely on people. People are informed of the organizational direction, and they are involved at all levels of decision making. People are given an opportunity to develop their strength and use their abilities. People are encouraged to be creative in performing their tasks.

· **Principle 4. Process Approach:** There are several advantages to performing major tasks by using the concept of *process*. A process is a sequence of activities that transform inputs to outputs. Organizations can prepare a plan in the form of allocating resources and scheduling the activities by making the process defined, repeatable, and measurable. Consequently, the organization becomes efficient and effective. Continuous improvement in processes leads to improvement in efficiency and effectiveness.

· **Principle 5. System Approach to Management:** A system is an interacting set of processes. A whole organization can be viewed as a *system* of interacting processes. In the context of software development, we can identify a number of processes. For example, gathering customer requirements for a project is a distinct process involving specialized skills. Similarly, designing a functional specification by taking the requirements as input is another distinct process. There are simultaneous and sequential processes being executed in an organization. At any time, people are involved in one or more processes. A process is affected by the outcome of some other processes, and, in turn, it affects some other processes in the organization. It is important to understand the overall goal of the organization and the individual sub goals associated with each process. For an organization as a whole to succeed in terms of effectiveness and efficiency, the interactions among processes must be identified and analyzed.

· **Principle 6. Continual Improvement:** Continual improvement means that the processes involved in developing, say, software products are reviewed on a periodic basis to identify where and how further improvements in the processes can be effected. Since no process can be a perfect one to begin with, continual improvement plays an important role in the success of organizations. Since there are independent changes in many areas, such as customer views and technologies, it is natural to review the processes and seek improvements. Continual process improvements result in lower cost of production and maintenance. Moreover, continual

improvements lead to less differences between the expected behavior and actual behavior of products. Organizations need to develop their own policies regarding when to start a process review and identify the goals of the review.

· **Principle 7. Factual Approach to Decision Making:** Decisions may be made based on facts, experience, and intuition. Facts can be gathered by using a sound measurement process. Identification and quantification of parameters are central to measurement. Once elements are quantified, it becomes easier to establish methods to measure those elements. There is a need for methods to validate the measured data and make the data available to those who need it. The measured data should be accurate and reliable. A quantitative measurement program helps organizations know how much improvement has been achieved due to a process improvement.

· **Principle 8. Mutually Beneficial Supplier Relationships:** Organizations rarely make all the components they use in their products. It is a common practice for organizations to procure components and subsystems from third parties. An organization must carefully choose the suppliers and make them aware of the organization's needs and expectations. The performance of the products procured from outside should be evaluated, and the need to improve their products and processes should be communicated to the suppliers. A mutually beneficial, cooperative relationship should be maintained with the suppliers.

17.4.2 ISO 9001:2000 Requirements

In this section, we will briefly describe five major parts of the ISO 9001:2000. For further details, we refer the reader to reference 8. The five major parts of the ISO 9001:2000, found in parts 4−8, are presented next.

Part 4: Systemic Requirements The concept of a *quality management system* (QMS) is the core of part 4 of the ISO 2001:2000 document. A quality management system is defined in terms of quality policy and quality objectives. In the software development context, an example of a quality policy is to review all work products by at least two skilled persons. Another quality policy is to execute all the test cases for at least two test cycles during system testing. Similarly, an example of a quality objective is to fix all defects causing a system to crash before release. Mechanisms are required to be defined in the form of processes to execute the quality policies and achieve the quality objectives. Moreover, mechanisms are required to be defined to improve the quality management system. Activities to realize quality policies and achieve quality objectives are defined in the form of *interacting quality processes*. For example, requirement review can be treated as a distinct process. Similarly, system-level testing is another process in the quality system. Interaction between the said processes occur because of the need to make all requirements testable and the need to verify that all requirements have indeed been adequately tested. Similarly, measurement and analysis are important processes in modern-day software development. Improvements in an existing QMS is achieved by defining a measurement and analysis process and identifying areas for improvements. Documentation is an important part of a QMS. There is no QMS without proper documentation. A QMS must be properly documented by publishing a quality manual. The quality manual describes the quality policies and quality objectives. Procedures for executing the QMS are also documented. As a QMS evolves by incorporating improved policies and objectives, the documents must accordingly be controlled. A QMS document must facilitate effective and efficient planning, execution, and management of organizational processes. Records generated as a result of executing organizational processes are documented and published to show evidence that various ISO 9001:2000 requirements have been met. All process details and organizational process interactions are documented. Clear documentation is key to understanding how one process is influenced by another. The documentation part can be summarized as follows:

· Document the organizational policies and goals. Publish a vision of the organization.
· Document all quality processes and their interrelationship.
· Implement a mechanism to approve documents before they are distributed.

· Review and approve updated documents.
· Monitor documents coming from suppliers.
· Document the records showing that requirements have been met.
· Document a procedure to control the records.

Part 5: Management Requirements The concept of quality cannot be dealt with in bits and pieces by individual developers and test engineers. Rather, upper management must accept the fact that quality is an all-pervasive concept. Upper management must make an effort to see that the entire organization is aware of the quality policies and quality goals. This is achieved by defining and publishing a QMS and putting in place a mechanism for its continual improvement. The QMS of the organization must be supported by upper management with the right kind and quantity of resources. The following are some important activities for upper management to perform in this regard:

· Generate awareness for quality to meet a variety of requirements, such as customer, regulatory, and statutory.
· Develop a QMS by identifying organizational policies and goals concerning quality, developing mechanisms to realize those policies and goals, and allocating resources for their implementations.
· Develop a mechanism for continual improvement of the QMS.
· Focus on customers by identifying and meeting their requirements in order to satisfy them.
· Develop a quality policy to meet the customers 'needs, serve the organization itself, and make it evolvable with changes in the marketplace and new developments in technologies.
· Deal with the quality concept in a planned manner by ensuring that quality objectives are set at the organizational level, quality objectives support quality policy, and quality objectives are measurable.
· Clearly define individual responsibilities and authorities concerning the implementation of quality policies.
· Appoint a manager with the responsibility and authority to oversee the implementation of the organizational QMS. Such a position gives clear visibility of the organizational QMS to the outside world, namely, to the customers.
· Communicate the effectiveness of the QMS to the staff so that the staff is in a better position to conceive improvements in the existing QMS model.
· Periodically review the QMS to ensure that it is an effective one and it adequately meets the organizational policy and objectives to satisfy the customers. Based on the review results and changes in the marketplace and technologies, actions need to be taken to improve the model by setting better policies and higher goals.

Part 6: Resource Requirements Resources are key to achieving organizational policies and objectives. Statements of policies and objectives must be backed up with allocation of the right kind and quantity of resources. There are different kinds of resources, namely, staff, equipment, tool, financial, and building, to name the major ones. Typically, different resources are controlled by different divisions of an organization. In general, resources are allocated to projects on a need basis. Since every activity in an organization needs some kind of resources, the resource management processes interact with other kinds of processes. The important activities concerning resource management are as follows:

· Identify and provide resources required to support the organizational quality policy in order to realize the quality objectives. Here the key factor is to identify resources to be able to meet−and even exceed−customer expectations.
· Allocate quality personnel resources to projects. Here, the quality of personnel is defined in terms of education, training, experience, and skills.
· Put in place a mechanism to enhance the quality level of personnel. This can be achieved by defining an acceptable, lower level of competence. For personnel to be able to move up to the

minimum acceptable level of competence, it is important to identify and support an effective training program. The effectiveness of the training program must be evaluated on a continual basis.

· Provide and maintain the means, such as office space, computing needs, equipment needs, and support services, for successful realization of the organizational QMS.

· Manage a work environment, including physical, social, psychological, and environmental factors, that is conducive to producing efficiency and effectiveness in "people" resources.

Part 7: Realization Requirements This part deals with processes that transform customer requirements into products. The reader may note that not much has changed from ISO 9001:1994 to ISO 9001:2000 in the realization part. The key elements of the realization part are as follows:

· Develop a plan to realize a product from its requirements. The important elements of such a plan are identification of the processes needed to develop a product, sequencing the processes, and controlling the processes. Product quality objectives and methods to control quality during development are identified during planning.

· To realize a product for a customer, much interaction with the customer is necessary to understand and capture the requirements. Capturing requirements for a product involves identifying different categories of requirements, such as requirements generated by the customers, requirements necessitated by the product's use, requirements imposed by external agencies, and requirements deemed to be useful to the organization itself.

· Review the customers' requirements before committing to the project. Requirements that are not likely to be met should be rejected in this phase. Moreover, develop a process for communicating with the customers. It is important to involve the customers in all phases of product development.

· Once requirements are reviewed and accepted, product *design* and *development* take place:

> Product design and development start with planning: Identify the stages of design and development, assign various responsibilities and authorities, manage interactions between different groups, and update the plan as changes occur.
>
> Specify and review the *inputs* for product design and development. Create and approve the *outputs* of product design and development. Use the outputs to control product quality.
>
> Periodically review the outputs of design and development to ensure that progress is being made.
>
> Perform design and development verifications on their outputs.
>
> Perform design and development validations.
>
> Manage the changes effected to design and development: Identify the changes, record the changes, review the changes, verify the changes, validate the changes, and approve the changes.

· Follow a defined purchasing process by evaluating potential suppliers based on a number of factors, such as ability to meet requirements and price, and verify that a purchased product meets its requirements.

· Put in place a mechanism and infrastructure for controlling production. This includes procedures for validating production processes, procedures for identifying and tracking both concrete and abstract items, procedures for protecting properties supplied by outside parties, and procedures for preserving organizational components and products.

· Identify the monitoring and measuring needs and select appropriate devices to perform those tasks. It is important to calibrate and maintain those devices. Finally, use those devices to gather useful data to know that the products meet the requirements.

Part 8: Remedial Requirements This part is concerned with measurement, analysis of measured data, and continual improvement. Measurement of performance indicators of

processes allows one to determine how well a process is performing. If it is observed that a process is performing below the desired level, then corrective action can be taken to improve the performance of the process. Consider the following example. We find out the sources of defects during system-level testing and count, for example, those introduced in the design phase. If too many defects are found to be introduced in the design phase, actions are required to be taken to reduce the defect count. For instance, an alternative design review technique can be introduced to catch the defects in the design phase. In the absence of measurement it is difficult to make an objective decision concerning process improvement. Thus, measurement is an important activity in an engineering discipline. Part 8 of the ISO 9001:2000 addresses a wide range of performance measurement needs as explained in the following:

· The success of an organization is largely determined by the satisfaction of its customers. Thus, the standard requires organizations to develop methods and procedures for measuring and tracking the customer's satisfaction level on an ongoing basis. For example, the number of calls to the help line of an organization can be considered as a measure of customer satisfaction−too many calls is a measure of less customer satisfaction.

· An organization needs to plan and perform internal audits on a regular basis to track the status of the organizational QMS. An example of an internal audit is to find out whether or not personnel with adequate education, experience, and skill have been assigned to a project. An internal audit needs to be conducted by independent auditors using a documented procedure. Corrective measures are expected to be taken to address any deficiency discovered by the auditors.

· The standard requires that both processes, including QMS processes, and products be monitored using a set of key performance indicators. An example of measuring product characteristics is to verify whether or not a product meets its requirements. Similarly, an example of measuring process characteristics is to determine the level of modularity of a software system.

· As a result of measuring product characteristics, it may be discovered that a product does not meet its requirements. Organizations need to ensure that such products are not released to the customers. The causes of the differences between an expected product and the real one need to be identified.

· The standard requires that the data collected in the measurement processes are analyzed for making objective decisions. Data analysis is performed to determine the effectiveness of the QMS, impact of changes made to the QMS, level of customer satisfaction, conformance of products to their requirements, and performance of products and suppliers.

· We expect that products have defects, since manufacturing processes may not be perfect. However, once it is known that there are defects in products caused by deficiencies in the processes used; efforts must be made to improve the processes. Process improvement includes both corrective actions and preventive actions to improve the quality of products.

➔ SQA Planning:
- ✓ SQA plan
- ✓ Organizational Level Initiatives.

8. Software Measurement & Metrics

➔ Measurement during Software Life Cycle Context
➔ Defect Metrics
➔ Metrics for software Maintenance & Requirements
➔ Measurement Principles
➔ Case study for Identifying Appropriate Measures & Metrics for Projects