| [Print](javascript:window.print()) | [Close](javascript:parent.window.close()) |

**Course Transcript**

White-Box Software Testing Techniques

**Structure-Based and Experience-Based Techniques**

| [1. Statement Testing and Coverage in Software Testing](http://xlibrary.skillport.com/courseware/Content/cca/sd_sftf_a05_it_enus/output/html/course_transcript.html#t2) |

| [2. Decision Testing and Coverage in Software Testing](http://xlibrary.skillport.com/courseware/Content/cca/sd_sftf_a05_it_enus/output/html/course_transcript.html#t5) |

| [3. Advanced Structure-Based Software Testing Techniques](http://xlibrary.skillport.com/courseware/Content/cca/sd_sftf_a05_it_enus/output/html/course_transcript.html#t9) |

| [4. Experience-Based Software Testing Techniques](http://xlibrary.skillport.com/courseware/Content/cca/sd_sftf_a05_it_enus/output/html/course_transcript.html#t12) |

| [5. Choosing Software Test Techniques](http://xlibrary.skillport.com/courseware/Content/cca/sd_sftf_a05_it_enus/output/html/course_transcript.html#t15) |

| [6. Structure- and Experience-Based Testing](http://xlibrary.skillport.com/courseware/Content/cca/sd_sftf_a05_it_enus/output/html/course_transcript.html#t18) |

Statement Testing and Coverage in Software Testing

Learning Objectives

*After completing this topic, you should be able to*

* *determine the effectiveness of a given test set in statement coverage white-box testing*
* *recognize the definition of coverage in software testing*

**1. Overview of white-box testing techniques**

White-box techniques, also known as structural or structure-based techniques, are based on an analysis of the structure of the component or system.

White-box techniques use information about how the software is constructed to derive the test cases, for example, using the structure of the code and the design specification to derive the test cases.

You can use structure-based testing techniques to measure *test coverage*. Test coverage is the extent to which a test suite exercises a specific test item.  
  
Measuring test coverage helps you to quantitatively assess the extent of testing performed using a set of tests. For example, in any situation where you can quantify some aspect of software such as code, and can then quantify how much of that code was tested by some particular test, you can then assign a measure to the test coverage. If the actual test coverage is less than that required, you would need to design additional structural tests to reach the required level of coverage.  
  
Test coverage therefore provides an objective measure and if applied to testing criteria based on priorities and risks, it may be applied as part of the completion or exit criteria defined in the test plan. Test coverage could then be used to determine when to stop testing in the final step of the test process.

Test coverage is measured as percentage of the coverage items. To measure the percentage of test coverage, you divide the number of coverage items exercised by the total number of coverage items and multiply the result by 100.

For example, you've written a program that has 20 statements. You've created a test set that covers 13 of 20 statements, which means the test coverage is 65%. This percentage value helps you determine whether you'd need additional test cases to cover the remaining statements. The coverage should ideally be 100%.

Depending on the number of system elements to be tested, you can measure coverage at various levels. They are

**component testing**

In *component testing*, each unit of code, such as a module, object, or class is tested separately. For example, if the code contains five functions, each of them is tested independently. Unit testing is typically done by the developers themselves, and any bugs are fixed at that stage.

**integration testing**

In *integration testing*, the integrated units of code are tested. You measure coverage of specific interactions between the units or modules in the code that have been tested. You can also measure the call coverage of modules, objects, or procedures. After the code is tested at the integration level, it is ready for acceptance testing.

**system or acceptance testing**

In *system* or *acceptance testing*, the code is tested as a whole. Coverage items include client requirements and user-interface elements, such as menu options. Coverage items could also include business transactions.

To effectively measure coverage, you can use testing tools. Using testing tools helps create test cases and improve productivity and test efficiency, by ensuring that most structural aspects are tested. To measure coverage, you

* determine the structural elements to be used
* count the structural elements
* instrument the code
* execute the tests for measuring coverage
* determine the percentage of elements exercised

When you instrument code, you add code next to each structural element, such as a For Each loop. When the test set is exercised, the instrumented code enables you to record whether this structural element has been exercised.  
  
The result of this instrumentation can be used to determine the percentage of elements that have been exercised. You can also identify tests that can be replaced with more meaningful tests.

Question

Which statements correctly define test coverage?

**Options:**

1. Test coverage is the extent to which a specific test item is exercised by a test suite
2. Measuring test coverage helps you quantitatively assess the extent and amount of testing
3. In test coverage, every line of code is exercised at least once
4. Test coverage is the extent to which executable lines of a specific test item have been exercised by a test suite

Answer

***Option 1:****Correct. Test coverage helps you determine the number of items exercised by the test set. Based on the test results, you can determine if additional test sets need to be designed.*

***Option 2:****Correct. Test coverage helps you quantitatively assess the extent of testing by enabling you to calculate the percent of test coverage.*

***Option 3:****Incorrect. Test coverage is a measurement of the amount of testing that has been performed by an executed set of tests.*

***Option 4:****Incorrect. Test coverage helps measure the extent to which a specific test item is exercised by a test suite.*

**Correct answer(s):**

1. Test coverage is the extent to which a specific test item is exercised by a test suite  
2. Measuring test coverage helps you quantitatively assess the extent and amount of testing

In structure-based testing, you need to generate test cases from code, so you need to be able to understand and analyze the code.

Pseudocode does not belong to any particular programming language. Rather, it is a simplified short-hand version of the source code that typically covers all the important control structures used in programming.

Note

*Sometimes programmers use pseudocode to decide on the design and structure of a program before actually writing the source code.*

There are two types of pseudocode. They are

Graphic

*The code sample is   
  
Line 1  
Line 2   
Line 3  
Line 4  
Line n  
. . .  
  
Begin  
  
  Statement 1  
  Statement 2  
  Statement n  
  . . .   
End*

**nonexecutable**

In software development, because certain lines of code don't perform an action, they are considered *nonexecutable code* statements. Nonexecutable code includes statements that declare variables. These statements are usually written at the beginning of a program and do not return results.  
*The nonexecutable code is   
  
Line 1  
Line 2   
Line 3  
Line 4  
Line n*

**executable**

*Executable code* statements instruct the computer to take a specific action and return a result. For example, the code for calculating a value such as individual tax is executable code and is usually written between Begin and End constructs.  
*The executable code is  
  
Begin  
  
  Statement 1  
  Statement 2  
  Statement n  
  . . .  
End*

Depending on the function you want the code to perform, you can structure executable code in different ways. They are

Graphic

*The code is   
  
READ P  
READ Q*

**sequence**

When code statements are arranged in a *sequence*, they are executed in a linear fashion. This is a rather simple programming structure and doesn't involve decision points or iterations.

**selection**

You use the *selection* method when you want to insert decision points in the code. In this method, you insert a condition in the code. If the condition is true, a specified set of steps is performed. Otherwise, a different set of steps is executed.

**iteration**

If you want to repeat a step a specific number of times, you use the *iteration* method. In this method, you specify the number of times the action should be repeated before the program ends.

Consider a simple program that accepts numeric input values from the user and assigns them to variables A and B. You then want to store the difference between the values in variable C.  
  
Due to the simplicity of this program, you would write the statements in a sequential manner. When this program is executed, the statements are executed sequentially. The program first accepts the values of A and B. The difference between the values is then assigned to the variable C.

Note

*Lines of executable code in a program are called statements.*

Consider another code sample in which you want to evaluate the condition of the variable A. Because this code involves conditions, you use the selection method of programming that uses IF and ELSE statements.  
  
This program checks the value of A. If the value is less than 10, the condition is true and the program sets the value of B as the sum of A and B. If the condition is false, it sets the value of A as the difference between A and B.

Question

You want to write a program that requests an integer and determines whether or not it is greater than 0. If the integer is greater than zero, it prints a message indicating that it is greater than zero. If the integer is not greater than zero, it prints a message indicating that it is less than or equal to zero.  
  
Arrange the given lines of code in the correct order to complete the pseudocode.

**Options:**

1. READ A
2. IF A > 0
3. PRINT "The number is greater than 0."
4. ELSE PRINT "The number is less than or equal to 0."
5. ENDIF

Answer

**Correct answer(s):**

**READ A is ranked**

This line when executed, accepts or reads the value of A.

**IF A > 0 is ranked**

This line checks whether A is greater than 0.

**PRINT "The number is greater than 0." is ranked**

This line prints the message if A is greater than 0.

**ELSE PRINT "The number is less than or equal to 0." is ranked**

This line prints another message if A is not greater than 0.

**ENDIF is ranked**

This line ends the program.

**2. Statement testing and coverage**

You can write a statement on a single line, spread it across multiple lines, or have multiple statements on one line. You can also have statements within statements.

*Statement coverage* helps measure the percentage of executable statements tested by a test case suite. Effective and efficient test case suites should aim for 100% statement coverage.

You can calculate statement coverage by dividing the number of statements exercised by the total number of statements and then multiplying the result by 100.

Measuring statement coverage helps you assess whether testing is complete and thorough. If statement coverage is inadequate, you can use the results as a baseline and improve coverage by creating more test cases.  
  
A test case suite cannot be considered complete and exhaustive unless each line of code is exercised at least once.

Consider this code that accepts three integers and prints their median. You want to test this code for 100% statement coverage.

Graphic

*The existing code is  
BEGIN  
READ a  
READ b  
READ c  
IF a < b  
  IF b < c  
     median = b  
  ELSE  
     median = c  
  ENDIF  
ELSE  
  IF a < c  
     median = a  
   ELSE  
     median = c  
  ENDIF  
ENDIF  
PRINT (“a, b, c values are:”, a, b, c)  
PRINT (“The median is:", median)  
END*

Because this code contains three decision points, 100% statement coverage requires you to create four test sets with different values.  
  
Consider the first test set with values 1, 2, and 3. When this test set is exercised, 12 out of 20 lines of code are executed. This means the test coverage is 60%.

You execute the second test set with values of 1, 3, and 2. This test set exercises 13 of 20 statements. Both test sets cover a total of 14 out of the 20 statements, which is still less than 100%, so you design more test sets.

The third test set with values 2, 1, and 3 covers more statements. All three test sets together cover 18 of 20 statements which means 90% of the statements are covered.

You require one more test set with values 3, 1, and 2 to execute all statements in the code. Therefore, for this specific code sample, you need a total of four test sets for 100% statement coverage.

Consider another code sample. To test this code sample, you create two tests, which include

**Test set 1 that has the values A = 5 and B = 10**

When you use the values of Test set 1, the value of C is 30 and so, the line that prints "Large C" is not executed. For 100% coverage you need another test set.

**Test set 2 that has the values A = 20 and B = 0**

With Test set 2, C is 80. Similar to test set 1, the line that prints "Large C" is not executed. For 100% coverage you need a different test set.

Test sets 1 and 2 covered the same line of code and therefore only 5 of 6 lines are exercised. Statement coverage is 83%. To achieve 100% statement coverage, you need to create another test set to exercise the statement not covered by the first 2 test sets.

So you design a test set in which A is 20 and B is 50. The resulting value of C is 130 and all lines of code are exercised.

In this very simple example, the first two test sets covered most of the statements, but the third test set provided 100% statement coverage. Therefore, the third test set is considered more effective and efficient because this test set alone achieves 100% statement coverage.

Question

You have written code to calculate the median of three integers. Which test sets are required to achieve 100% statement coverage for the median program?

**Code**  
BEGIN  
READ a  
READ b  
READ c  
IF a < b  
   IF b < c  
      median = b  
   ELSE  
      median = c  
   ENDIF  
ELSE  
   IF a < c  
      median = a  
    ELSE  
      median = c  
   ENDIF  
ENDIF  
PRINT (“a, b, c values are:”, a, b, c)  
PRINT (“The median is:", median)  
END

**Options:**

1. Test set A with values 10, 20, 30; 10, 30, 20; 20, 10, 30; and 30, 10, 20
2. Test set B with values 10, 20, 30 and 10, 30, 20
3. Test set C with values 10, 20, 30; 20, 30, 10; and 20, 30, 30
4. Test set D with values 0, 10, 20; 10, 20, 20; and 20, 30, 30

Answer

***Option 1:****Correct. To achieve 100% statement coverage, you'd need four test sets with three integer values each.*

***Option 2:****Incorrect. With these two test sets, you'd exercise 14 of the 20 program statements and, as a result, the coverage is 70%.*

***Option 3:****Incorrect. With these three test sets, you'd exercise 14 of the 20 program statements and, as a result, the coverage is 70%.*

***Option 4:****Incorrect. With these three test sets, you'd exercise 14 of the 20 program statements and, as a result, the coverage is 70%.*

**Correct answer(s):**

1. Test set A with values 10, 20, 30; 10, 30, 20; 20, 10, 30; and 30, 10, 20

**3. Summary**

Using structure-based or white-box testing, you generate test cases based on the internal structure, architecture, and logic of a program.  
  
Each line of code in a program is called a statement. Statement testing is the testing of individual lines of source code in a program. Statement coverage measures the percentage of statements exercised by a test case suite during statement testing. You should focus on covering all statements in a program's source code for 100% coverage.   
  
Test coverage can be measured at component, integration, system, and acceptance-testing levels.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_sftf_a05_it_enus/output/html/course_transcript.html#top)

Decision Testing and Coverage in Software Testing

Learning Objective

*After completing this topic, you should be able to*

* *determine the effectiveness of a given test set in decision coverage white-box testing*

**1. Decision testing and coverage**

You can use control flow diagrams to visually depict program structures. Structures such as decision points may be easier to understand when visually depicted. You can also analyze code and arrive at test cases by understanding the control flow diagram.  
  
A control flow diagram uses two symbols to represent code structure – a rectangle and a diamond.

Sequential statements are represented by rectangles, and decision statements are depicted by diamonds.

Control flow diagrams can be used to depict structures such as

**Sequential structure**

Code that is structured sequentially is comprised of statements that are executed one after the other. You use rectangular boxes to depict a sequential structure in a control flow diagram. For example, you use rectangular boxes to depict a program that calculates the sum of 2 numbers. This type of operation is linear and does not involve decision points.

**Selection or decision**

A decision structure is used to represent decision points within a program. To visually depict code with a decision structure, you use diamond-shaped boxes. When the program reaches a decision point, it checks the condition and depending on the outcome, the program branches to the appropriate path in the program. For example, you can use decision structures to visually depict the testing of a condition such as whether the value of a variable is less than 0.

**Iteration or loop**

In an iteration structure in code, a statement is executed repeatedly until a specific condition is met. Once the condition is satisfied, the program ends or branches to another sequence, selection or loop. You can use the diamond-shaped box to depict an iteration structure.

For sequential statements, the flow graph is a straight line because there are no decision points that cause branching. For example, you would use a simple linear flow diagram to visually represent a program that calculates mean value.  
  
The input values for calculating mean are read, the mean is calculated using a formula, and the result is printed to the screen.

Graphic

*The flow chart depicts the program that reads integer values and then calculates their mean. It then prints the mean value.*

Decision points in a program are depicted using IF statements. You use IF statements to check whether a condition is TRUE or FALSE. A different set of statements are executed depending on whether the condition evaluates to TRUE or FALSE. In a program that contains decision points, you can test decision coverage.

*Decision coverage* measures the percentage of decision outcomes exercised by a test suite. You perform decision testing to be certain that a decision is adequately tested. You ensure that during testing, all appropriate statements are executed whether a condition evaluates to true or false.

Similar to statement testing, the goal of decision testing is to achieve 100% decision coverage. This involves testing both outcomes – true and false – of a decision. Furthermore, 100% decision coverage guarantees 100% statement coverage. However, 100% statement coverage doesn't necessarily mean 100% decision coverage. This will be discussed in detail shortly.

To measure decision coverage, you divide the number of decision outcomes exercised by the total number of decision outcomes and multiply the result by 100.

Suppose you've created a program that accepts three different integers, assign them to variables a, b, and c and then prints the maximum of the 3 variables.

If a is greater than b, the program branches off to the next decision point and checks whether a is greater than c. The program would then print the variable a or c depending on which was larger.

If a is not greater than b, the program branches off to a different decision point and checks whether b is greater than c. The program would then print the variable b or c depending on which was larger.

You want to now test the program for decision coverage. For 100% decision coverage, you need four test sets because there are three decisions and four paths through which the program can flow.

**Test set 1A**

The values in Test set 1A are 1, 2, and 3. When these values are substituted for a, b, and c, the result of the first decision point is false because a is not greater than b. The program then branches to the second decision point and checks whether b is greater than c. In this test set, b is not greater than c, so c is the largest integer.

**Test set 1B**

The values in Test set 1B are 1, 3, and 2. When these values are substituted for a, b, and c, the outcome of the first decision point – if a is greater than b – is still false. So the program branches to the next decision point where it checks whether b is greater than c. In this test set, b is greater than c and so b is the largest integer.

**Test set 1C**

The values in Test set 1C are 3, 2, and 1. When these values are substituted for a, b, and c, the outcome of the first decision point is true. The program then checks whether a is greater than c – which is the next decision point. In this case, a is greater than c and so a is the largest integer.

**Test set 1D**

The values in Test set 1D are 2, 1, and 3. When these values are substituted for a, b, and c, the outcome of the first decision point is true. The program then checks whether a is greater than c but in this case, it is not. So c is the largest integer.

For this program, four test sets are required for 100% decision coverage.   
  
However, there are programs for which you can create test sets that provide 100% statement coverage but don't provide 100% decision coverage. In such situations, you'll need to design more test sets to achieve 100% decision coverage.

Consider a program that reads two integers. The program sets the values of both integers and prints them.

For 100% statement coverage, one test set with values 1 and 2 would be sufficient because this test set would exercise all statements. However, this doesn't mean all decisions are exercised. This test set exercises only one outcome.

In this case, 100% decision coverage would mean testing both outcomes of the decision point. So you need to design a test set where the value of a is greater than b.

Question

You've created a program that accepts three input values and prints the largest value. You want 100% decision testing for the program so you've designed some test sets. Identify the test set that meets 100% decision coverage.

**Code**  
READ a  
READ b  
READ c  
  
IF a > b  
   IF a > c  
      max = a  
   ELSE  
      max = c  
   ENDIF  
ELSE  
   IF b > c  
      max = b  
   ELSE  
      max = c  
   ENDIF  
ENDIF  
PRINT (“The maximum is: ”, max)  
END

**Options:**

1. Test set 1 with the values 10, 30, 50; 10, 50, 30; 50, 30, 10; and 30, 10, 50
2. Test set 2 with the values 10, 30, 50; 10, 50, 30; and 30, 10, 50
3. Test set 3 with the values 10, 30, 50 and 10, 50, 30
4. Test set 4 with the values 10, 30, 50; 10, 50, 30; 50, 5, 10; and 30, 10, 20

Answer

***Option 1:****Correct. Using this test set ensures that all decisions outcomes are executed, thereby providing 100% decision coverage.*

***Option 2:****Incorrect. This set covers only five out of six decision outcomes. It would, therefore, achieve only 83% decision coverage.*

***Option 3:****Incorrect. With the specified set, only three of the six decision outcomes are covered, achieving only 50% decision coverage.*

***Option 4:****Incorrect. Although there are four tests, two of them test for the same outcome. So this test set covers only five of the six decision outcomes and achieves only 83% decision coverage.*

**Correct answer(s):**

1. Test set 1 with the values 10, 30, 50; 10, 50, 30; 50, 30, 10; and 30, 10, 50

**2. Summary**

You use control flow diagrams to visually depict program structure. These include sequence, selection or decision, and iteration structures. Rectangular boxes are used to depict sequence structures and diamond-shaped boxes are used to depict decision points or iteration structures in a program.  
  
For 100% decision coverage, all decisions in the program must be exercised. If the decision coverage is 100%, then statement coverage would also be 100%. To ensure that a decision has been adequately tested, you need to execute the associated statements both when the condition is true and when it is false.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_sftf_a05_it_enus/output/html/course_transcript.html#top)

Advanced Structure-Based Software Testing Techniques

Learning Objectives

*After completing this topic, you should be able to*

* *determine the effectiveness of a given test set in condition coverage white-box testing*
* *recognize how path coverage is used*

**1. Condition and path coverage**

Programs often contain numerous conditions, which can lead to numerous paths. To ensure 100% coverage, you need to test all possible outcomes of the conditions and paths. To do so, you use advanced techniques, such as condition and path testing.  
  
These techniques are more thorough than statement and decision coverage because 100% path coverage implies both 100% decision coverage and 100% statement coverage.

Because these tests are rigorous and exhaustive, they maximize the probability of detecting errors. These advanced techniques are implemented in safety-critical systems, such as medical software, where even a single instance of system failure is unacceptable.

Condition coverage checks and evaluates the outcomes of each individual condition. Condition coverage helps you measure the percentage of conditional outcomes exercised by a test suite. Typically, you perform condition coverage after decision coverage.

Note

*Decision coverage is closely related to branch coverage. These terms are interchangeably used in various sources.*

To achieve 100% condition coverage, each condition in the decision should be tested for both true and false outcomes.  
  
Suppose you've developed code that accepts four integers. You have two conditions in the code: a greater than b and c greater than d. You want to check for either of the conditions, so you use the OR operator. In this program, 100% Condition coverage would require you to test the outcomes of both the conditions.

Graphic

*A diamond box in a flow chart depicts decisions and checks for conditions. If the conditions are True, then a specific set of steps are performed; if the conditions are False, a different set of steps are performed.*

Consider the code in which you treat statements a>b as X and c>d as Y. To achieve 100% statement coverage for this code, you need only one test set, and for 100% decision coverage, you need two test sets. However, to achieve 100% condition coverage, you need three test cases to check the possible outcomes of each condition:

* X is true (if X is true Y is never tested)
* X is false and Y is true
* X is false and Y is false

Consider the flow chart that visually depicts a program that reads three integers and prints the largest integer. This program contains two decision points.

To achieve 100% condition coverage, you need six test sets in all: four for the first decision point and two for the second one. For the first decision point, you need to check for True True, True False, False True, and False False output values.

And for the second decision point, you only need to check for True and False values.

In addition to condition coverage, you may have to use multiple condition coverage techniques. Multiple condition coverage is more rigorous than statement and decision testing and requires more test cases for 100% condition coverage.  
  
In multiple condition coverage, you not only test the main condition or expression but you also test the possible combination of conditions for the subexpressions of the main expression or condition.

Consider the pseudocode for a program that accepts three integers and prints whether or not the third integer is the highest. This pseudocode is explained using a flow chart.  
  
In this example, condition coverage is more rigorous than statement and decision coverage.

In the flow chart for the pseudocode, if the first part of the OR condition is true, the first OR condition is satisfied. The program doesn't need to check the second condition. This is called a short-circuited OR.

In this example, if the second condition after the OR statement is not tested, it indicates less than 100% coverage. Therefore, it should be specified that the OR condition is not short-circuited, and the entire expression must be evaluated to achieve 100% path coverage.

Question

You've created a program that prints the maximum of the three integers. You want to achieve 100% decision coverage for the program and have created some test sets. Which test set helps you achieve 100% condition coverage?

**Options:**

1. Test set 1.1 with values 3, 2, 1; 6, 4, 2; and 2, 3, 1
2. Test set 1.2 with values 3, 2, 1 and 6, 4, 2
3. Test set 1.3 with values 3, 2, 1; 2, 1, 3; 2, 3, 1, and 1, 2, 3
4. Test set 1.4 with values 1, 2, 3 and 2, 4, 6

Answer

***Option 1:****Incorrect. With these test sets, you can only achieve 50% condition coverage because only one of the conditions is exercised.*

***Option 2:****Incorrect. When you exercise this test set, you'll find that only one condition is tested. This means only 50% coverage.*

***Option 3:****Correct. You need four test cases to ensure 100% condition coverage because there are four possible combinations of outcomes for the first condition.*

***Option 4:****Incorrect. This test set covers only one condition, which means the test set achieves only 50% condition coverage.*

**Correct answer(s):**

3. Test set 1.3 with values 3, 2, 1; 2, 1, 3; 2, 3, 1, and 1, 2, 3

When measuring decision and condition coverage, the test cases would typically exercise each decision in the program. However, sometimes, when you exercise a certain *path* in a program, you might find critical errors. For example, the program might behave in an unexpected manner if that particular path is executed. In this case, the developer might need to review the program structure.

Path coverage testing is the most comprehensive type of testing that a test suite can provide. However, path coverage testing is difficult and is usually reserved for critical sections of code. A path in a program could be simple and linear or it could contain various decision points. Even if only a few decisions are involved, 100% path coverage could be very time consuming and complicated. Therefore, it can be impractical to test all paths in a program.

Path coverage can be measured at various levels, such as component and system testing. When unit-testing software, it helps to know the percentage of modules exercised by the test suite. However, when performing an acceptance-level test, it is highly recommended that you test all possible paths.

In addition to path coverage, you can use the linear code sequence and jump (LCSAJ) technique for measuring code coverage. This is a variation of the path coverage technique in which only the subpaths of the program are exercised. You can use this technique as long as the statements in the subpath execute in a linear manner at run time.

Consider the pseudocode for the ContainsNeg program. It reads three integers and determines whether the smallest integer is negative. This pseudocode is explained using the flow chart.  
  
This program demonstrates how quickly path coverage can grow in complexity. This example also helps explain that it is not always possible to test all the combinations in a path because the input data values may be dependent on each other.

To achieve optimal path coverage for the ContainsNeg program, you'd need eight test cases – a test case for each possible path of decision:

* Test case 1 with values 0, 0, and 0 to check the condition false false false
* Test case 2 with values 9000, 18000, and -36000 to check the condition false false true
* Test case 3 with values 4, -3, and 0 to check the condition false true false
* Test case 4 with values 4, -3, and -1 to check the condition false true true
* Test case 5 with values -12, 8, and 4 to check the condition true false false
* Test case 6 with values -10, 5, and -15 to check the condition true false true
* Test case 7 with values -4, -6, and 0 to check the condition true true false
* Test case 8 with values -3, -5, and -10 to check the condition true true true

Each of these test sets checks whether at least one of the integers is negative, neg = 1. But 100% path coverage would mean that you also check if the value of the variable neg is 0. For this, you would require another set of eight test cases, which means that you'd need a total of sixteen test sets for 100% path coverage. However, this may not be feasible and at the very least, would be very time consuming.

Sometimes, there may be no test cases that can satisfy all criteria for a program. Statements that cannot be exercised even once are unreachable. For example, you've written a program that performs calculations such as addition, subtraction, and multiplication.  
  
When you review the structure of the code, you find that the program doesn't call the multiplication function. In this case, the multiplication function can't be tested, and as a result, those statements are unreachable. You need to correct the code structure so that all functions are accessible and can be tested.

Unreachable statements can also exist in programs that check for uncommon error conditions. It could also occur when code becomes unmanageable because it keeps accumulating when converting legacy applications.  
  
This problem can occur even in well-designed and well-maintained code. If it happens, you need to perform detailed code coverage analysis and then test the code once again. The testing process should accommodate the time and effort spent on these additional activities.

Consider another piece of pseudocode called Countdown, which begins by setting a minimum integer, min to 10. The program then reads input values. If the input is less than 1, then the program terminates. If an integer less than 10, but greater than 0 is input, the program prints the integer and sets it as the new minimum.

Code

BEGIN  
  
min = 10  
input = min  
PRINT input  
WHILE input > 0 DO  
   IF input < min  
   THEN  
      PRINT input  
      min = input  
   ENDIF  
   READ input  
END DO  
  
END

This program uses the loop structure, and the loops create an infinite number of possible paths. This makes it difficult to test all the paths. In this example, you can achieve 100% statement coverage without 100% decision coverage.

If there are an infinite number of possible paths in a program, you need to follow specific rules when testing the program. These rules should be established at the time the application is created. If there are loop structures, like the DO WHILE loop in this example, you would want to test the looping section for the case where it processes once and exits. You would also want to test for the case when it processes multiple times and exits.

To ensure that each path within the loop is tested, you need an input value of less than min at least once, and you also need an input of not less than min at least once. An optimal testing set would contain paths in which the input value is:

* less than or equal to 0; for example, -1
* greater than 0 but less than min; for example, 5
* equal to 0; for example, 0
* greater than min; for example, 11
* equal to min; for example, 10

Question

You've created a program that sets 20 as the minimum value. This program accepts user input and if an input is less than the current minimum value, it sets the new value as minimum. If you enter a value less than 0, the program terminates. Which test set would be optimal for this program?

**Code**  
BEGIN  
  
min = 20  
input = min  
PRINT input  
WHILE input >= 0 DO  
   IF input < min  
   THEN  
      PRINT input  
      min = input  
   ENDIF  
   READ input  
END DO  
  
END

**Options:**

1. Test set 1 with values 10, 0, 21, -100
2. Test set 2 with values 0, 21, -100
3. Test set 3 with values 10, -100
4. Test set 4 with values 10, 0, 21

Answer

***Option 1:****Correct. Because loops contain an infinite number of possible paths, you need to test the program with a limited set of values to achieve optimal coverage.*

***Option 2:****Incorrect. For optimal coverage, you need one more value that is greater than 0 but less than the minimum value.*

***Option 3:****Incorrect. For optimal coverage, you need one more value to satisfy the condition where input is greater than 20, the minimum value.*

***Option 4:****Incorrect. To achieve optimal path coverage, you need a value that is less than 0.*

**Correct answer(s):**

1. Test set 1 with values 10, 0, 21, -100

**2. Summary**

You can use condition coverage techniques to check and evaluate the outcomes of each individual condition in a program. Condition coverage helps measure the percentage of conditional outcomes exercised by a test suite. To achieve 100% condition coverage, each condition in the decision should be tested for both true and false outcomes.  
  
In addition to condition coverage, you may also need to perform path coverage – the most comprehensive testing technique.  
  
Because it is a difficult and complex undertaking to perform path testing, it is usually reserved for critical sections of code.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_sftf_a05_it_enus/output/html/course_transcript.html#top)

Experience-Based Software Testing Techniques

Learning Objectives

*After completing this topic, you should be able to*

* *recognize how error guessing can be used in software testing*
* *recognize how exploratory testing can be used in software testing*

**1. Experience-based techniques**

In most of the structure-based testing techniques, such as statement, decision, condition, and path coverage testing, specifications for testing the software are already provided.  
  
However, in situations in which sufficient specifications aren't available for test cases, you have to rely on your experience and skill as a tester. This is called *experience-based* testing.

Experience-based testing supplements specification-based testing. In experience-based testing, you rely on past experience with similar technologies to help determine the areas likely to contain errors. You use your skill and intuition to locate defects that can't be found using any other testing technique.

Experience-based testing is called ad hoc or reactive testing because the specifics of these tests are defined while testing the software. In  
ad hoc testing, test details are either created spontaneously or as a reaction to the functionality of the software system.

Because this type of testing is ad hoc, you need a framework to ensure that adequate tests are exercised. To define the framework, you need to have a general understanding of the type of bugs, system weaknesses, and previous defects identified.

Because experience-based testing is open-ended and doesn't follow any specification, you need to keep two key factors in mind. These include

**managing time and effort**

Because reactive tests are open-ended, you have to run them in a "time-boxed" manner. For example, consider a situation where you've developed a pilot software program that has requirements that are not closed, and you need to test the product for possible enhancements. In this instance, you would run an experience-based test. The time allocated for testing the software would depend on the risks associated with the test conditions, the complexity of the area to be tested, and the time required to test a specific use case.

**tracking test coverage**

To track coverage, you would need test charters containing a high-level description of the test cases. For example, suppose you need to test the spell-check function of a word-processing application. In this case, you create the test charter called test\_spell\_check and specify the time box as 60 minutes. This means the tester is not expected to take more than 60 minutes to test the spell-check function and would exceed the time limit only if the product contains an unexpected number of bugs.

One type of experience-based testing technique is error guessing. In this technique, you guess the location of an error according to your testing experience with previous applications.

Although this type of testing has no specifications, you can create test cases according to the application. Consider an application that contains a Zip code field. When testing this application using the error guessing technique, you check for various cases including whether the user has entered a five-digit number.

Although error guessing is an ad hoc technique, you can systemize this test technique. For this, you need to understand how the software works and identify the possible ways for the software to fail.   
  
After you identify potential failures, you can organize and categorize them into types. You can include this information while designing test cases for the system.

You can also use error guessing to detect buffer overrun – an input string that is too long for the program to handle. This error can occur when developers do not insert checkpoints in the code to check the size of inputs.

Other defects you can detect using the error guessing technique are division by zero, no input values in a field, empty files, and entering wrong data in a field.  
  
Using this technique, you can also detect errors you think would never occur in the software. For example, the software manufacturer claims that you can use the software to modify 100 records simultaneously. So as an experienced tester, you should try entering 100, or approximately 100, records to test this feature of the software.

Question

You've created an authentication system that enables users to access the applications hosted on your organization's intranet. To log on to the application, you need to provide your employee code and password. How will you identify errors in the application by using error guessing?

**Options:**

1. By checking if the employee code field accepts special characters
2. By checking if the password field accepts zero input
3. By asking a trainee software engineer to test the application
4. By testing the application according to the specifications provided by the client

Answer

***Option 1:****Correct. You'd use error guessing to determine whether the employee code field accepts special characters. If it does, this would be a defect.*

***Option 2:****Correct. Using error guessing , you can check if you can log on without providing the password to determine the stability of the application.*

***Option 3:****Incorrect. Error guessing requires previous experience of testing similar applications or technology.*

***Option 4:****Incorrect. In error guessing, no specifications are provided; you're required to create test cases depending on the type of application.*

**Correct answer(s):**

1. By checking if the employee code field accepts special characters  
2. By checking if the password field accepts zero input

**2. Exploratory techniques**

Exploratory testing technique is another kind of experience-based testing technique. It is a combination of ad hoc and structure-based testing.

Exploratory testing is the process by which you design test cases, execute tests, and log test results based on a test charter within a time box. A test charter contains test objectives and the time box denotes a specific span of time within which testing should be completed.

You typically perform exploratory tests if specifications are missing or if they're insufficient or if time is limited. You perform exploratory tests in addition to specification-based tests to ensure that most of the critical defects are detected.

In the exploratory test technique, the testers control the design of the tests. Based on their experience, the testers decide the aspects of the product to be tested, the coverage of tests, and the time frame within which the testing should be completed. The information gained from the tests is used to create new tests.

In exploratory testing, you learn about the software by exploring it and figuring out how it works. When you explore the system, you make decisions about what to test within the stipulated time.

To perform exploratory tests, you follow a sequence of steps:

* understanding how the system works
* making informed guesses regarding the location of the errors
* designing a test case to check for bugs
* executing the tests

When time is limited, skilled testers design and execute test cases in parallel, based on a test charter. Therefore, while the tests are performed, you need to simultaneously document the test cases and test conditions because no separate time is allotted for documenting.

As a skilled tester, you should generate a test log that contains the main aspects of the tested product, defects, if any, and the strategies implemented to discover those defects. You can then use these findings to create test cases for other similar projects.

Although you can perform exploratory tests on any type of project, they're usually performed when the requirements of a product or a project change frequently. An example of such a product is one that is updated every two months. With each new release, product features are enhanced. Exploratory testing can help you test the new features and check the compatibility between current and previous versions of the software.

In this case, it would be difficult to maintain scripted test cases because you need to constantly update them. And because exploratory tests don't require you to maintain test cases, you save time and effort spent on documenting them.

You can also use exploratory tests when you want to provide feedback on a new product or feature. You use this technique if you want to find the most serious defect in the shortest possible time.

Exploratory tests performed by skilled and experienced testers can be more beneficial and effective than structured tests performed by less-experienced testers.

Question

You need to test the three new features added to a word-processing application that need to be released as soon as possible. You have not tested a word processing application before. There isn't sufficient time for an extensive testing routine. What would you do in this situation?

**Options:**

1. Perform exploratory tests by understanding how the new features in the application work
2. Create some test-case designs based on application specifications and then perform exploratory tests
3. Design and execute test cases simultaneously after studying and analyzing how the new features work
4. Guess a possible area of error, based on your past experience with similar applications

Answer

***Option 1:****Correct. To perform exploratory tests, you need to first understand how the system works and then take an educated guess about where an error could exist.*

***Option 2:****Incorrect. The scenario suggests that there isn’t sufficient time to build test cases based on application specifications. Exploratory tests are normally employed when application specifications are unavailable or insufficient, or will require too much time to be referenced in creating test cases.*

***Option 3:****Correct. After you've understood the system, you can design and execute the test cases.*

***Option 4:****Incorrect. If the application you are currently testing is similar to a previous application, you use error guessing.*

**Correct answer(s):**

1. Perform exploratory tests by understanding how the new features in the application work  
3. Design and execute test cases simultaneously after studying and analyzing how the new features work

**3. Summary**

In experience-based testing, you primarily rely on the tester's skill and previous experience with the software and similar technologies. It is a good idea to supplement structure-based testing techniques with experience-based testing. This is because it helps you detect defects that you might otherwise miss. Experience-based testing is also referred to as ad hoc and reactive testing.  
  
To perform experience-based testing, you need to understand how the software system works. Then you need to make an educated guess about the possible weak points. You focus on those points and design and execute test cases.  
  
The two types of experience-based testing are error guessing and exploratory testing. In error guessing, you need to understand how the software system works, identify the various ways in which the software could fail, and then create test cases based on your analysis. In exploratory testing, you design test cases, execute them, and log test results based on a test charter within a time box.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_sftf_a05_it_enus/output/html/course_transcript.html#top)

Choosing Software Test Techniques

Learning Objective

*After completing this topic, you should be able to*

* *recognize which set of software testing techniques are most appropriate in a given scenario*

**1. Selecting a test technique**

You can use various software testing techniques, such as structure-based, specification-based, and experience-based, depending on the requirements of your project. None of these techniques are right, wrong, better, or worse because each technique is suitable for a specific situation and covers a certain area. However, sometimes you might need to use multiple techniques because a single technique would be inadequate for detecting errors.

Sometimes, test specifications are well documented, but the code doesn't cover all the specifications. Such discrepancies can be detected only through specification-based testing.

In some cases, you may need to use a combination of testing techniques because one alone may not be sufficient to detect all types of bugs in the software. For example, to test complex software, you must use specification-based test techniques. Then to complement this, you would use experience-based testing. This approach helps discover bugs that couldn't be detected using specification-based techniques alone.

Graphic

*The three types of test techniques are specification-based, structure-based, and experience-based.*

Suppose you're testing software that doesn't contain sufficient documentation for creating specification-based test cases. You have a limited amount of time to run a structured test. In this scenario, you should rely on the skills and experience of the tester to exercise the important areas of the software.

Assume that you've created a complex web application. All the specifications are well documented and there is separate time allocated for testing the software before it is deployed.  
  
You might want to perform exhaustive and thorough testing. So you use structure-based techniques, such as statement and decision testing. These techniques help you test the structure and logic of the application code.

Graphic

*The first column contains four conditions listed on four rows. In the second column, Rule1, there is Y in the conditions 1 and 2 rows and there is N in the conditions 3 row. In the third column, Rule2, there is N in the conditions 1 row and Y in the conditions 2 and 3 rows. In the fourth column, there is N in the conditions 2 and 3 rows.*

Determining the technique used to test an application or software depends on various internal factors.

Graphic

*The tree diagram depicts the three types of test techniques – Specification-based, Structure-based, and Experience-based.*

**The models used while developing the system**

Your choice of testing technique would depend on the software model used during specification, design, development, and implementation of the system. For example, if the specification requires you to test various cases with different possible outcomes, you use the decision table testing technique.

**Knowledge and experience of the tester**

The knowledge, skill, and experience of the tester is another factor that influences the choice of test techniques. If you want to test a system based on equivalence partitioning and boundary values and if the tester is an expert in boundary-value analysis, you might decide to perform boundary-value analysis testing instead of equivalence testing.

**Potential defects**

Based on past experience, if you know that certain defects are likely to be present in the system, it's easier to select the test technique. You can find the areas with likely defects if you have experience with similar applications or technology. For example, if you have tested the previous version of an application, you can use that experience to discover likely errors in the current version.

**Test objective**

The test technique you select also depends on the test objective. Suppose you've created a financial application and you want to test whether it functions as expected. So, you employ the use-cases approach. If you want to perform complete and rigorous testing, you use structure-based techniques.

**Documentation**

The extent and style of documentation plays an important role in determining the test technique. If the specifications are missing or aren't well documented, you use an experience-based test approach. However, if the specifications are detailed and contain diagrams that explain state transition, you use the state transition testing technique.

**Life cycle model**

Your choice of test technique depends on the testing life cycle model you use. If you're using the iterative life cycle model where you have to test various values in a loop structure, exploratory testing should be used. But if you're using the sequential model, you can test the application using structure-based techniques.

In addition to the internal factors, various external factors affect the choice of test techniques. They are

**risk**

The risk involved in testing safety-critical systems, such as radiation therapy machines, is very high. So you need to devise structure-based testing techniques to thoroughly test the application. There could also be an associated commercial risk involving quality or time-to-market issues, so you use exploratory testing to test the application.

**contractual requirements**

Your company has signed a contractual agreement with the client to test their software applications. The contract specifies that you need to use boundary value analysis and equivalence partitioning techniques. Even though other techniques, such as decision and path testing would equally be suitable, you have to use the technique stipulated in the contract.

**type of system**

The type of test technique you use also depends on the type of systems, such as an embedded or a financial system. Suppose you're testing a loan application and want to test the various conditions for approving loan amounts. You also want to test the outcomes of all the conditions. In such a situation, you use the decision-table testing technique.

**regulatory requirements**

Some industries have regulatory guidelines that form a basis for choosing test techniques. For example, the medical industry might require the use of equivalence partitioning and boundary value analysis for testing life-critical medical software systems. So you use these techniques in addition to structure-based techniques, such as decision coverage and path testing.

**time and cost**

If you have sufficient time and if the required budget is sanctioned, you can use a combination of different test techniques. But if time and budget are limited, you might use the one technique, such as experience-based testing, that best meets the requirements of the application.

Question

You're testing an e-learning software application. The documented specifications for this application seem insufficient and there is limited time for gathering the missing specifications.  
  
Identify the software testing techniques most appropriate for this situation.

**Options:**

1. Error guessing
2. Condition coverage
3. Exploratory
4. Structure-based

Answer

***Option 1:****Correct. If time is limited and the required test specifications are missing, you use the expertise of an experienced tester to guess the areas that may contain errors.*

***Option 2:****Incorrect. Condition coverage helps you measure the percentage of conditional outcomes exercised by a test suite.*

***Option 3:****Correct. You use the exploratory technique when there are commercial risks involving quality or time-to-market issues.*

***Option 4:****Incorrect. You use structure-based test techniques to review the structure of code for bugs.*

**Correct answer(s):**

1. Error guessing  
3. Exploratory

**2. Summary**

Depending on the application, you can use any test technique to help detect the maximum number of defects.To do this, you can either use individual techniques, such as statement coverage, experience-based, and exploratory testing or a combination of techniques.  
  
The technique you choose depends on various internal and external factors. Internal factors include the knowledge of the tester and the test objective, and external factors include risks, time, and budget.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_sftf_a05_it_enus/output/html/course_transcript.html#top)

Structure- and Experience-Based Testing

Learning Objectives

*After completing this topic, you should be able to*

* *evaluate the efficiency of white-box software testing techniques*
* *evaluate the efficiency of experience-based software testing techniques in comparison with white-box techniques*

**1. Exercise Overview**

**2. Evaluating white-box testing techniques**

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_sftf_a05_it_enus/output/html/course_transcript.html#top)

© 2016 Skillsoft Ireland Limited