Testing Concepts

Lesson 4: Test Techniques

Capgemini

Lesson Objectives

To understand the following topics:

- Categories of Test Techniques
  - Choosing Test Techniques
  - Categories of Test Techniques & their Characteristics
- Black-box Test Techniques
  - Equivalence Partitioning
  - Boundary Value Analysis
  - Decision Table Testing
  - State Transition Testing
  - Use Case Testing

Lesson Objectives

- White-box Test Techniques
  - Statement Testing and Coverage
  - Decision Testing and Coverage
  - The Value of Statement and Decision Testing
- Experience-based Test Techniques
  - Error Guessing
  - Exploratory Testing
  - Checklist-based Testing

4.1 Categories of Dynamic Test Techniques

- Dynamic Testing involves working with the software, giving input values and validating the output with the expected outcome
- Dynamic Testing is performed by executing the code
- It checks for functional behavior of software system , memory/CPU usage and overall performance of the system
- Dynamic Testing focuses on whether the software product works in conformance with the business requirements
- Dynamic testing is performed at all levels of testing and it can be either black or white box testing

Categories of Dynamic Test Techniques (Cont..)

White Box Test Techniques

- Code Coverage
  - Statement Coverage
  - Decision Coverage
  - Condition Coverage
  - Loop Testing
- Code complexity
  - Cyclomatic Complexity
- Memory Leakage

Black Box Test Techniques

- Equivalence Partitioning
- Boundary Value Analysis
- Use Case / UML
- Error Guessing
- Cause-Effect Graphing
- State Transition Testing

4.1.1 Choosing Test Techniques

The choice of which test techniques to use depends on a number of factors :

- Type of component or system
- Component or system complexity
- Regulatory standards
- Customer or contractual requirements
- Risk levels & Risk types
- Test objectives
- Available documentation
- Tester knowledge and skills
- Available tools
- Time and budget
- SDLC model
- Expected use of the software
- Previous experience with using the test techniques on the component or system to be tested
- The types of defects expected in the component or system

The use of test techniques in the test analysis, test design, and test implementation activities can range from very informal (little to no documentation) to very formal. The appropriate level of formality depends on the context of testing, including the maturity of test and development processes, time constraints, safety or regulatory requirements, the knowledge and skills of the people involved, and the software development lifecycle model being followed.

4.1.2 Categories of Test Techniques & their Characteristics

Black-box test techniques

- It is a.k.a. behavioral/behavior-based techniques are based on an analysis of the appropriate test basis (e.g., formal requirements documents, specifications, use cases, user stories, or business processes).
- These concentrate on the inputs and outputs of the test object without reference to its internal structure.
- These techniques are applicable to both functional and nonfunctional testing.

White-box test techniques

- It is a.k.a. structural/structure-based techniques are based on an analysis of the architecture, detailed design, internal structure, or the code of the test object.
- These concentrate on the structure and processing within the test object.

Experience-based test techniques

- These leverage the experience of developers, testers and users to design, implement, and execute tests.
- These techniques are often combined with black-box and white-box test techniques.

Characteristics of Black-box Test Techniques

- Test conditions, test cases, and test data are derived from a test basis that may include software requirements, specifications, use cases, and user stories
- Test cases may be used to detect gaps between the requirements and the implementation of the requirements, as well as deviations from the requirements
- Coverage is measured based on the items tested in the test basis and the technique applied to the test basis

Characteristics of White-box Test Techniques

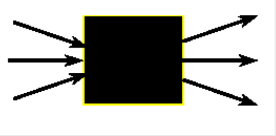
- Test conditions, test cases, and test data are derived from a test basis that may include code, software architecture, detailed design, or any other source of information regarding the structure of the software
- Coverage is measured based on the items tested within a selected structure (e.g., the code or interfaces)
- Specifications are often used as an additional source of information to determine the expected outcome of test cases

Characteristics of Experience-box Test Techniques

- Test conditions, test cases, and test data are derived from a test basis that may include knowledge and experience of testers, developers, users and other stakeholders.
- This knowledge and experience includes expected use of the software, its environment, likely defects, and the distribution of those defects

4.2 Black Box Test Techniques

- Black box is data-driven, or input/output-driven testing
- The Test Engineer is completely unconcerned about the internal behavior and structure of program
- Black box testing is also known as behavioral, functional, opaque-box and closed-box
- Black Box can be applied at different Test Levels – Unit, Subsystem and System.

Input  Output

Black-box Tests are designed to answer the following questions:

How is functional validity tested ?

What classes of input will make good test cases?

Is the system particularly sensitive to certain input values?

What effect will specific combinations of data have on system operations?

Black box testing attempts to find errors in the following categories
incorrect or missing functions interface errors, errors in data structures or external database access behavior or performance errors, initialization errors.

Black Box Test Techniques

There are various techniques to perform Black box testing ;

- Equivalence Partitioning
- Boundary Value Analysis
- Decision Table Testing
- State transition testing
- Use Case Testing
- Error Guessing

4.2.1 Equivalence Partitioning

- Equivalence partitioning divides data into partitions called as equivalence classes in such a way that all the members of a given partition are expected to be processed in the same way.
- There are equivalence partitions for both valid and invalid values.
  - Valid values are values accepted by the component or system. An equivalence partition containing valid values is called a "valid equivalence partition."
  - Invalid values are values rejected by the component or system. An equivalence partition containing invalid values is called an "invalid equivalence partition."
- Partitions can be identified for any data element related to the test object, including inputs, outputs, internal values, time-related values (e.g., before or after an event) and for interface parameters (e.g., integrated components being tested during integration testing).
- Assumption: If one value in a group works, all will work. One from each partition is better than all from one.

- Any partition may be divided into sub partitions if required.
- Each value must belong to one and only one equivalence partition.
- When invalid equivalence partitions are used in test cases, they should be tested individually, i.e., not combined with other invalid equivalence partitions, to ensure that failures are not masked. Failures can be masked when several failures occur at the same time but only one is visible, causing the other failures to be undetected.
- To achieve 100% coverage with this technique, test cases must cover all identified partitions (including invalid partitions) by using a minimum of one value from each partition.
- Coverage is measured as the number of equivalence partitions tested by at least one value, divided by the total number of identified equivalence partitions, normally expressed as a percentage.
- Equivalence partitioning is applicable at all test levels.

Guidelines & Examples to identify Equivalence Classes

1. If an input condition specifies a continuous range of values, there is one valid class, and two invalid classes

   E.g. The valid range of a mortgage applicant's income is $1000 - $75,000
   Valid class: {1000 > = income < = 75,000}
   Invalid classes: {income < 1000}, {income > 75,000}

2. If an input condition specifies a set of values, there is reason to believe that each is handled differently in the program.

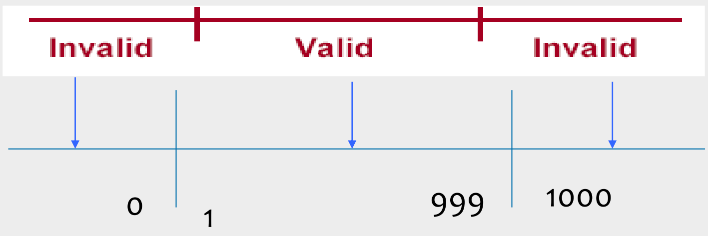
   E.g. Type of Vehicle must be Bus, Truck, Taxi). A valid equivalence class would be any one of the values and invalid class would be say Trailer or Van.

3. If a "must be" condition is required, there is one valid equivalence class and one invalid class

   E.g. The mortgage applicant must be a person
   Valid class: {person}
   Invalid classes:{corporation, ...anything else...}

Examples: Equivalence Partitioning

If an input condition specifies that a variable, say count, can take range of values(1 - 999). There is **one valid equivalence class (1 < count < 999) and two invalid equivalence classes (count < 1) & (count >999)**



If we have to test function int Max (int a , int b) the Equivalence Classes for the arguments of the functions will be

Arguments	Valid Values	Invalid Values
a	-32768 <= Value <= 32767	< - 32768 , >32767
b	-32768 <= Value <= 32767	< - 32768 , >32767

4.2.2 Boundary Value Analysis

"Bugs lurk in corners and congregate at boundaries ….." *Boris Beizer*

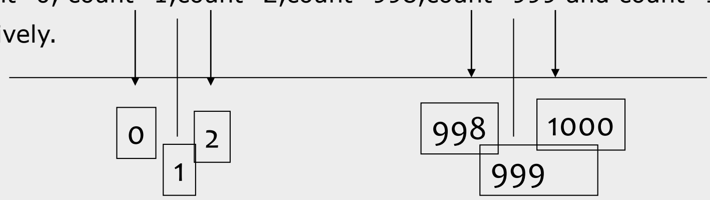
Boundary Conditions are those situations directly on, above, and beneath the edges of input equivalence classes and output equivalence classes.

Boundary value analysis is a test case design technique that complements Equivalence partitioning but can only be used when the partition is ordered, consisting of numeric or sequential data. The minimum and maximum values (or first and last values) of a partition are its boundary values.

Test cases at the boundary of each input Includes the values at the boundary, just below the boundary and just above the boundary.

Guidelines & Examples for Boundary Value Analysis

E.g. From previous example, we have valid equivalence class as (1 < count < 999). Now, according to boundary value analysis, we need to write test cases for count=0, count=1,count=2,count=998,count=999 and count=1000 respectively.



E.g. If we have to test the function int Max(int a , int b) the Boundary Values for the arguments of the functions will be :

Arguments	Valid Values	Invalid Values
a	-32768, -32767, 32767, 32766	-32769,32768
b	-32768, -32767, 32767, 32766	-32769,32768

General guidelines:

If an input or output condition specifies a range of values A and B, test cases should be designed with values A and B, just above and just below A and B respectively

If an input or output condition specifies a number of values, write test cases for the minimum and maximum number of values.

If the input or output of a procedure is an ordered set, focus attention on the first and last elements of the set.

Guidelines for BVA are similar in many respects to those provided for EP:

1. If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b as well as just above and just below.
2. If an input condition specifies a number of values test cases should be developed that exercise the minimum and maximum numbers. Values just above and below min and max are also tested.
3. Applying guidelines 1 and 2 output conditions. For example, assume that a temperature vs. pressure table is required as output from an engineering analysis program. Test cases should be designed to cerate an output report that produces the maximum (and Min) allowable number of table entries.
4. If internal program data structures have prescribed boundaries (e.g.- an array has defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

4.2.3 Decision Table Testing

- Decision Testing is useful for testing the implementation of system requirements that specify how different combinations of conditions result in different outcomes.
- When creating decision tables, the tester identifies conditions (often inputs) and the resulting actions (often outputs) of the system.
- These form the rows of the table, usually with the conditions at the top and the actions at the bottom.
  - Each column corresponds to a decision rule that defines a unique combination of conditions which results in the execution of the actions associated with that rule.
  - The values of the conditions and actions are usually shown as Boolean values (true or false) or can also be numbers or ranges of numbers.

Notations in Decision Tables

The common notation in decision tables is as follows:

For conditions:

- Y means the condition is true (may also be shown as T or 1)
- N means the condition is false (may also be shown as F or 0)
- — means the value of the condition doesn’t matter (may also be shown as N/A)

For actions:

- X means the action should occur (may also be shown as Y or T or 1)
- Blank means the action should not occur (may also be shown as – or N or F or 0)

Example of Decision Table

Printer troubleshooter

		Rules							
Conditions	Printer does not print	Y	Y	Y	Y	N	N	N	N
	A red light is flashing	Y	Y	N	N	Y	Y	N	N
	Printer is unrecognized	Y	N	Y	N	Y	N	Y	N
Actions	Check the power cable			X					
	Check the printer-computer cable	X		X					
	Ensure printer software is installed	X		X		X		X	
	Check/replace ink	X	X			X	X		
	Check for paper jam		X		X				

Advantages of Decision Table Testing

1. The strength of decision table testing is that it helps to identify all the important combinations of conditions, some of which might otherwise be overlooked.
2. It also helps in finding any gaps in the requirements.
3. It may be applied to all situations in which the behavior of the software depends on a combination of conditions, at any test level.

4.2.4 State Transition Testing

- A state transition diagram shows the possible software states, as well as how the software enters, exits, and transitions between states.
- A transition is initiated by an event (e.g., user input of a value into a field). The event results in a transition.
- If the same event can result in two or more different transitions from the same state, that event may be qualified by a guard condition.
- The state change may result in the software taking an action (e.g., outputting a calculation or error message).

Example :

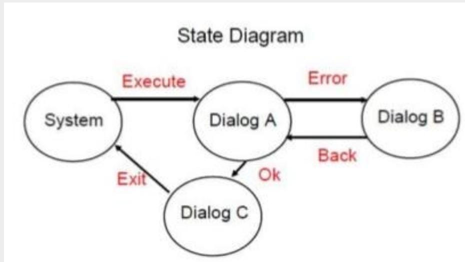
- The State Transition testing is used for Menu-based application.
- The program starts with an introductory menu. As an option is selected the program changes state and displays a new menu. Eventually it displays some information , data input screen.
- Each option in each menu should be tested to validate that each selection made takes us to the state we should reach next.

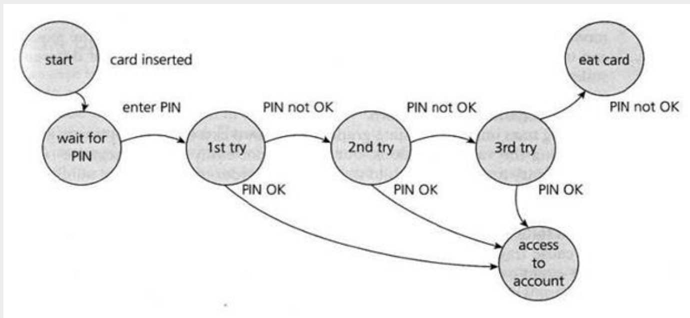
Example:
A registration form has two buttons, viz. OK and CANCEL. After filling up the entire application, OK button changes to SAVE caption to save the filled data.
So the single button is acting in two different ways depending on its state transition, which has to be tested.
Example :
State Transition testing is used within the embedded software industry. Washing machine has different modes like soak, wash, rinse & dry
Machine in these different states, exhibit different features :

Soak mode - clothes absorb soap water
Wash mode - clothes get washed with soap water
Rinse mode - It removes soup water from clothes
Dry mode - water gets removed from clothes

It is useful to create a state transition diagram to spot relationship between states and trace transition between states

Guidelines & Examples of State Transition Testing





Guidelines :

- A State transition table shows all valid transitions as well as invalid transitions between states; but, a State transition diagrams normally show only the valid transitions and exclude the invalid transitions.
- State Transition Coverage is commonly measured as the number of identified states or transitions tested, divided by the total number of identified states or transitions in the test object, normally expressed as a percentage.

4.2.5 Use Case Testing

- Tests can be derived from use cases, which are a specific way of designing interactions with software items, incorporating requirements for the software functions represented by the use cases.
- Use cases are associated with actors (human users, external hardware, or other components or systems) and subjects (the component or system to which the use case is applied).
- Each use case specifies some behavior that a subject can perform in collaboration with one or more actors.
- A use case can be described by interactions and activities, as well as preconditions, post conditions and natural language where appropriate.
- Interactions between the actors and the subject may result in changes to the state of the subject.

A use case can include possible variations of its basic behavior, including exceptional behavior and error handling (system response and recovery from programming, application and communication errors, e.g., resulting in an error message).
Tests are designed to exercise the defined behaviors (basic, exceptional or alternative, and error handling).
Coverage can be measured by the percentage of use case behaviors tested divided by the total number of use case behaviors, normally expressed as a percentage.

4.3 White Box Test Techniques

- White box is logic driven testing and permits Test Engineer to examine the internal structure of the program
- Examine paths in the implementation
- Make sure that each statement, decision branch, or path is tested with at least one test case.
- Desirable to use tools to analyze and track Coverage
- White box testing is also known as structural, glass-box and clear-box

There are various techniques to perform Black box testing ;

- Code Coverage
  - Statement Testing and Coverage
  - Decision Testing and Coverage
  - Condition Testing and Coverage
- Code complexity

Using white box testing methods, you can derive test cases that guarantee that all independent paths within a module have been excised at least once
exercise all logical decisions on their true and false sides execute all loops at their boundaries and within their operational bounds exercise internal data structures to ensure their validity.

4.3.1 Statement Coverage

Test cases must be such that all statements in the program is traversed at least once.

Example :

Consider the following snippet of code

```
void procedure(int a, int b, int x)
{
    If (a>1) && (b==0)
            { x=x/a;  }   //statement 1
    If (a==2) || (x>1)
            {  x=x+1; }  //statement 2
}
```

Coverage is measured as the number of statements executed by the tests divided by the total number of executable statements in the test object, normally expressed as a percentage.

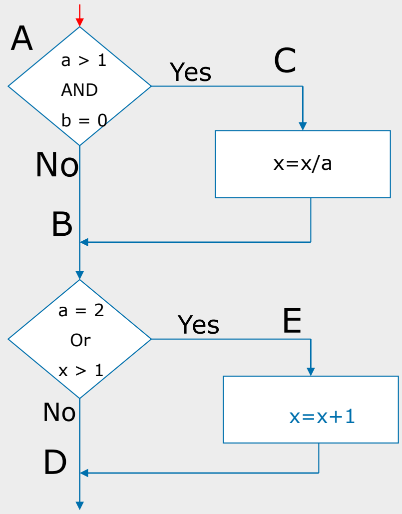
The goal is to execute every statement in the program at least once. Every statement must be executed.

Statement Coverage

Test Case 1: a=2,b=0, x=3.

Every statement will be executed once.

One test case is sufficient to execute all the statements in the code.



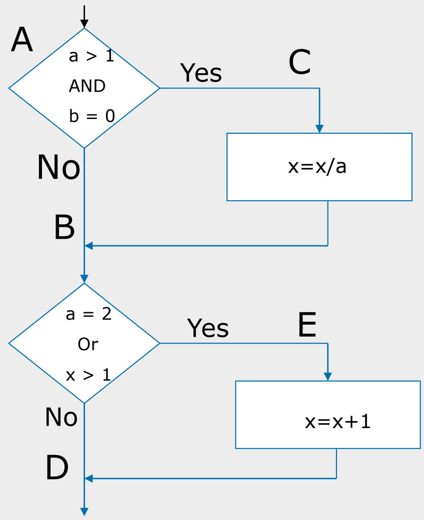
Every statement can be executed by writing a single test case. But, this case covers only ACE path.
This criteria is weak one. Since it is not considering other paths to traverse. So the path ABD, ACD, ABE would go undetected.
This test case does not help in detecting many of the bugs which may go unnoticed as the false outcomes of the conditions a>1 & b=0 , a=2 or x>1 are not tested.

4.3.2 Decision Coverage

Test Case 1: a=2, b=0, x>1
(Decision1 is True, Decision2 is True) (Path ACE)

Test Case 2: a<=1 , b!=0, x<=1
(Decision1 is False, Decision2 is False) (Path ABD).

Two test cases are sufficient to test all decisions – every decision should be tested at least once for both TRUE and FALSE sides.



Even if the above test cases satisfy decision coverage, it only covers the paths ACE and ABD. It still does not cover the path ACD and path ABE.
Though decision coverage is stronger than statement coverage, it is still weak. There is only 50 percent chance that we would explore the path.

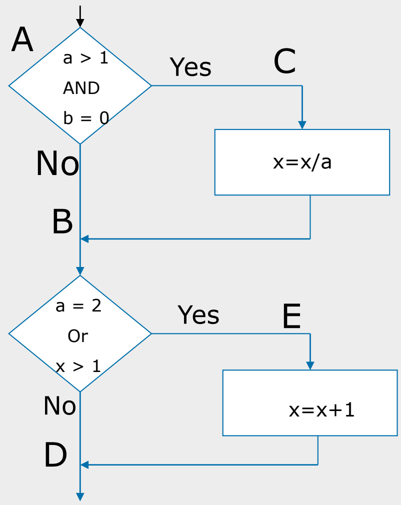
4.3.3 Value of Statement and Decision Testing

- When 100% statement coverage is achieved, it ensures that all executable statements in the code have been tested at least once, but it does not ensure that all decision logic has been tested. Statement testing provides less coverage than decision testing.
- When 100% decision coverage is achieved, it executes all decision outcomes, which includes testing the true outcome and also the false outcome, even when there is no explicit false statement (e.g., in the case of an IF statement without an else in the code).
- Statement coverage helps to find defects in code that was not exercised by other tests. Decision coverage helps to find defects in code where other tests have not taken both true and false outcomes.
- **Achieving 100% decision coverage guarantees 100% statement coverage.**

4.3.4 Condition Coverage

Test cases are written such that each condition in a decision takes on all possible outcomes at least once.

Test Case1 : a=2, b=0, x=3 (Condition1 is True,Condn2 is True)
(Path ACE)
Test Case2 : a=3, b=0, x=0
(Condn1 is True,Condn2 is False,Condn3 is False)
(Path ACD)
Test Case3 : a=1, b=0, x=3
(Condition1 is False,Condition2 is True)
(Path ABE)
Test Case4: a=1, b=1, x=1
(Condition1 is False,Condition2 is False)
(Path ABD)

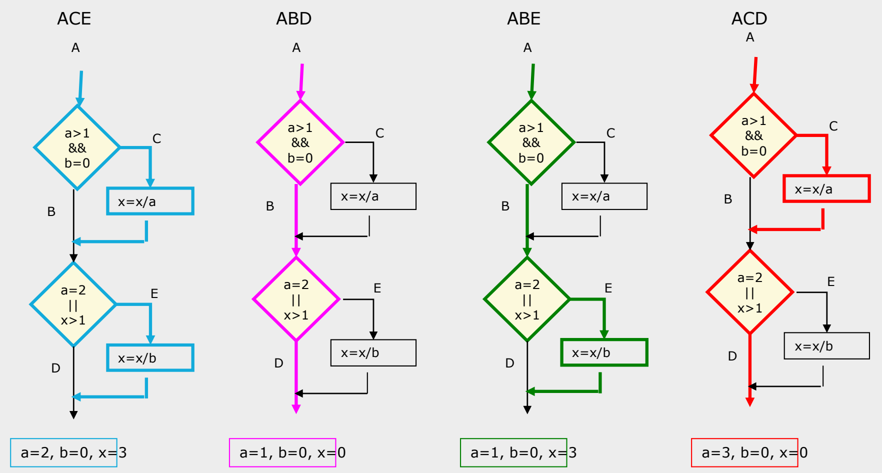


Condition testing is a test case design method that exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or a relational expression. Relational operator is one of the following <, <=, =, not =, > , =>.

A compound condition is composed of two or more simple conditions, Boolean operators, and parentheses.

Condition coverage focuses on testing each condition in a program. The purpose of the condition testing is to detect not only errors in the conditions of a program but also other errors in the program.

Condition Coverage



What does "coverage" mean?

- NOT all possible combinations of data values or paths can be tested
- Coverage is a way of defining how many of the paths were actually exercised by the tests
- Coverage goals can vary by risk, trust, and level of test

In the above diagrams, each condition in decision takes all possible outcomes at least once.

Cyclomatic Complexity

- Cyclomatic Complexity (Code Complexity) is a software metric that provides a quantitative measure of logical complexity of a program
- When Used in the context of the basis path testing method, value for cyclomatic complexity defines number of independent paths in basis set of a program
- Also provides an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once
- Cyclomatic complexity is often referred to simply as program complexity, or as McCabe's complexity

Introduced by Thomas McCabe in 1976:

Count the regions of the flow graph (including the exterior)
Or compute by e-n+2 This is called Cyclomatic Complexity
The number of paths to test , all decision options are tested

How many paths (McCabe's technique for units)?

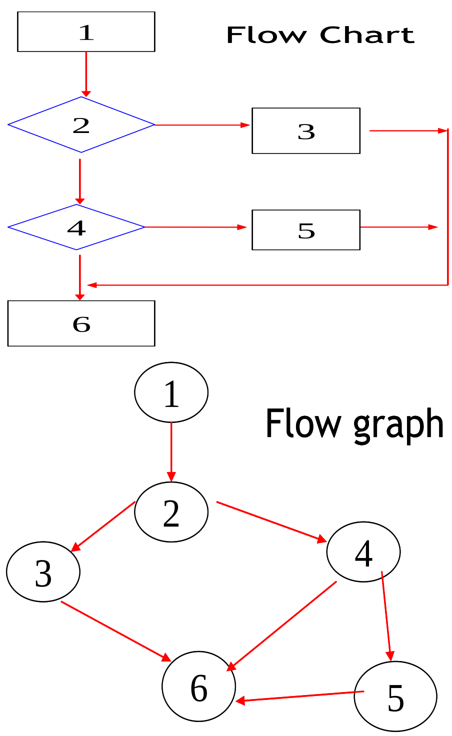
Cyclomatic complexity defines the number of independent paths. This provides minimum number of tests to be conducted to ensure all the statements have been executed at least once.
An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined.

Cyclomatic complexity is a useful metric for predicting those modules that are likely to be error prone. Use it for test planning as well as test case design.
Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program.
When used in context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us the upper bound of the number of tests that must be conducted to ensure that all statements have been executed at least once.

Calculating Cyclomatic Complexity

- The cyclomatic complexity of a software module is calculated from a flow graph of the module , when used in context of the basis path testing method
- Cyclomatic Complexity V(G) is calculated one of the three ways:
  - V(G) = E - N + 2 , where E is the number of edges and N = the number of nodes of the graph
  - V(G) = P+1, where P is the number of predicate nodes
  - V(G) = R , where number of region in the graph

Here, a flow chart is used to depict program control structure. Flow chart is mapped into corresponding flow graph. Each circle is called as flow graph node, represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node.

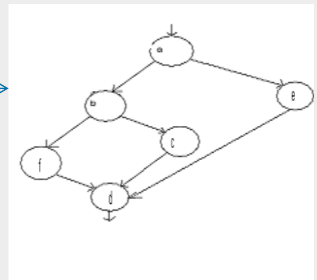


A flow chart depicts program control structure.
Flow chart is mapped into flow graph.
A sequence of process boxes and a decision diamond are map into a single node.

Each circle is called as graph node.
Arrows are called as edges.
The arrows on the flow graph is called edges, represents flow of control. An edge must be terminated at a node.

Calculating Cyclomatic Complexity : Example

In the given figure a and b are predicate nodes



1. Cyclomatic Complexity, V(G) for a flow Graph G is V(G) = E – N + 2
   E = Number of Edges in the graph (7 in the above figure)
   N = number of flow graph Nodes (6)
   R = number of Regions (3)
   Hence V(G) = 7-6+2 = 3
2. V(G) can also be calculated as V(G) = P+1, where P is the number of predicate nodes. Here V(G) = 2+1 = 3
3. Also V(G) can be calculated as V(G) = R hence V(G) = 3

The value of V(G) provides the number of linearly independent paths through the program.
Here, in the above graph, value of V(G) is 3. That is, minimum 3 test cases must be designed to guarantee coverage of all program statements.
In the above graph, the cyclomatic complexity is 3. That means, there are 3 independent paths in this program. Independent path is any path through the program that has set of statements or a condition.
Path 1: a-e-d
Path 2: a-b-f-d
Path 3: a-b-c-d
In order to have complete coverage of this code, there are minimum 3 test cases are required which traversed thorough the above paths.

4.4 Experience-based Test Techniques

- When applying experience-based test techniques, the test cases are derived from the tester’s skill and intuition, and their experience with similar applications and technologies.
- These techniques can be helpful in identifying tests that were not easily identified by other more systematic techniques.
- Depending on the tester’s approach and experience, these techniques may achieve widely varying degrees of coverage and effectiveness.
- Coverage can be difficult to assess and may not be measurable with these techniques.

Commonly used experience-based techniques are

- Error Guessing
- Exploratory Testing
- Checklist-based Testing

4.4.1 Error Guessing

It is an ad hoc approach

Error guessing is a technique used to anticipate the occurrence of mistakes, defects, and failures, based on the tester's knowledge, including:

- How the application has worked in the past
- What types of mistakes the developers tend to make
- Failures that have occurred in other applications

A methodical approach to the error guessing technique is to create a list of possible mistakes, defects, and failures, and design tests that will expose those failures and the defects that caused them.

Example :

Suppose we have to test the login screen of an application. An experienced test engineer may immediately see if the password typed in the password field can be copied to a text field which may cause a breach in the security of the application.

Here mention about Myers Probability study that the probability of errors remaining in the program is proportional to the number of errors that have been found so far, which provides a rich source for productive error guessing.

Example : Error guessing testing for sorting subroutine situations

The input list empty
The input list contains only one entry
All entries in the list have the same value
Already sorted input list

Other Examples :

Empty or null lists/strings
Zero occurrences
Blanks or null character in strings
Negative numbers

4.4.2 Exploratory Testing

- In exploratory testing, informal (not pre-defined) tests are designed, executed, logged, and evaluated dynamically during test execution.
- The test results are used to learn more about the component or system, and to create tests for the areas that may need more testing.
- Exploratory testing is sometimes conducted using session-based testing to structure the activity.
- Exploratory testing is most useful when there are few or inadequate specifications or significant time pressure on testing.
- Exploratory testing is also useful to complement other more formal testing techniques. Exploratory testing is strongly associated with reactive test strategies (see section 5.2.2). Exploratory testing can incorporate the use of other black-box, white-box, and experience-based techniques.

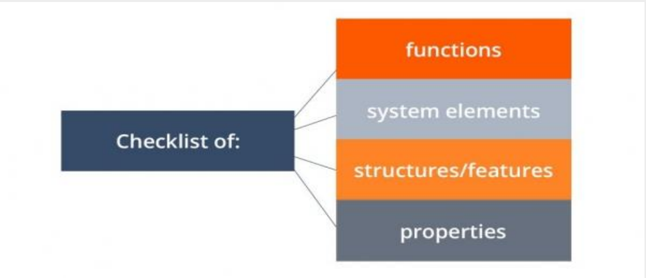
In session-based testing, exploratory testing is conducted within a defined time-box, and the tester uses a test charter containing test objectives to guide the testing. The tester may use test session sheets to document the steps followed and the discoveries made.

4.4.3 Checklist-based Testing

- In checklist-based testing, testers design, implement, and execute tests to cover test conditions found in a checklist.
- As part of analysis, testers create a new checklist or expand an existing checklist, but testers may also use an existing checklist without modification.
- Such checklists can be built based on experience, knowledge about what is important for the user, or an understanding of why and how software fails.
- Checklists can be created to support various test types, including functional and non-functional testing.
- In the absence of detailed test cases, checklist-based testing can provide guidelines and a degree of consistency.
- As these are high-level lists, some variability in the actual testing is likely to occur, resulting in potentially greater coverage but less repeatability.

Check-list based Technique

Commonly used Testing Checklists



Functional (black-box) checklists contain checks for dominant functions of the complete system or for the definite functions of the lower levels.
System elements checklists (white-box) examine sub-systems and modules of higher levels as well as special data items at secondary levels.
Structure/feature checklists path through various aspects such as the list of customers and producers for definite sources or units sharing some average data, etc.
Properties checklists test fixed values such as definite specification units, code systems, etc.

Advantages of using checklists in testing

- Flexibility.
- Easy to create.
- Analyzing the results.
- Team integration.
- Deadlines control.
- Test case reusability can help in cutting down costs incurred in missing out on important testing aspects.
- innovative testing strategy can be added in the testing checklist.

Flexibility. This kind of checking can be used in all testing types.
Easy to create. It is not difficult to create, use and maintain a checklist.
Analyzing the results. Checklists are easy to follow and examine.
Team integration. The checklist can be ready-made guidance and help new testing personnel to integrate into work.
Deadlines control. This type of testing helps to control test accomplishment and not to miss some critical bugs before the deadline.

Difficulties of using checklists in testing

- Different interpretation
- Difficulty in Test results reproducibility
- “Holes” in coverage.
- Item overlap.
- Reporting problems.

Different interpretation. QA engineers with various backgrounds can accomplish identical tasks using different approaches.
“Holes” in coverage. It is difficult to capture all functional or structural components, especially those of higher levels.
Item overlap. Trying to cover a big scope of material there may be a duplication of the same information. This can lead to excessive testings.
Reporting problems. Complex system components, functions, and their interaction are difficult or even impossible to illustrate, applying checklists.

Summary

In this lesson, you have learnt:

- The test case techniques discussed so far need to be combined to form overall strategy
- Each technique contributes a set of useful test cases, but none of them by itself contributes a thorough set of test cases

Review Question

Question 1: For calculating cyclomatic complexity, flow graph is mapped into corresponding flow chart

- Option: True / False

Question 2: How many minimum test cases required to test a simple loop?

Question 3: Incorrect form of logic coverage is :

- Statement coverage
- Pole coverage
- Condition coverage
- Path coverage

Question 4: One test condition will have ____________ test cases.

Review Question

Question 5: For Agile development model conventional testing approach is followed.

- Option: True / False

Question 6: A test case is a set of _________, ______________, and ___________ developed for a particular objective.

Question 7: An input field takes the year of birth between 1900 and 2004. State the boundary values for testing this field.

- 0, 1900,2004,2005
- 1900, 2004
- 1899, 1900, 2004, 2005
- 1899, 1900, 1901, 2003, 2004, 2005

Review Question: Match the Following

1. Code coverage
2. Interface errors
3. Code complexity

A. Flow graph
B. Loop testing
C. Black box testing
D. Flow chart
E. Condition testing
F. White box testing