# UNIT II

# TEST CASE DESIGN STRATEGIES

Test case Design Strategies – Using Black Box Approach to Test Case Design – Boundary Value Analysis – Equivalence Class Partitioning – State based testing – Cause-effect graphing – Compatibility testing – user documentation testing – domain testing - Random Testing – Requirements based testing – Using White Box Approach to Test design – Test Adequacy Criteria – static testing vs. structural testing – code functional testing – Coverage and Control Flow Graphs – Covering Code Logic – Paths – code complexity testing – Additional White box testing approaches- Evaluating Test Adequacy Criteria

## 2.1 INTRODUCTION TO TEST CASE DESIGN STRATEGIES

- ❖ Generally software testing refers a set of activities which are well planned in advance and also conducted systematically.
- ❖ The testing strategy should be flexible to the people like customer, developer
- ❖ This testing strategy is developed by
  - ✓ Software developers
  - ✓ System engineers
  - ✓ Testing specialists

**Testing Maturity Model (TMM)**

- ❖ TMM is a learning tool which helps to know about testing
- ❖ It says both the technical and management aspects of testing
- ❖ It leads to natural evaluation of testing process
- ❖ There are 3 maturity goals at TMM level2. Among them two goals are managerial in nature. Another one is technical oriented.
- ❖ The technically oriented maturity goal gives important and basic technical issues related to the execution based testing.

## 2.2 SMART TESTER

- ❖ Software components have defects, even if we have very good defect prevention activities are implemented.
- ❖ Developers cannot prevent or eliminate all defects during development. Software must be tested before it is delivered to users.
- ❖ It is the responsibility of the testers to design tests. The test must reveal defects, and can be used to evaluate software performance, usability, and reliability.

- To achieve these goals, testers must select a finite number of test cases, from a very large **execution domain**.
- Testing is usually performed under budget and time constraints. Testers often are under pressures from management and marketing because testing is not well planned, and expectations are very high.
- The **smar*t tester*** must plan for testing, select the test cases, and monitor the process to ensure that the resources and time allocated for the job are utilized effectively. Testers need proper education and training and the ability to make use of management support.
- New testers may try to test a module or component using all possible inputs and exercise all possible software structures. Using this approach, will enable them to detect all defects. An experienced tester knows that is not an economically possible goal.
- The tester can select test inputs at random, with a hope that these tests will reveal critical defects. Some testing experts believe that randomly generated test inputs have a poor performance record.
- Goal of the smart tester is to understand the functionality, input/output domain, and the environment of use for the code being tested. A smart tester needs to use knowledge of the types of defects that are commonly injected during development or maintenance of this type of software.
- Using this information, the smart tester must then intelligently select a subset of test inputs as well as combinations of test inputs that he believes have the greatest possibility of revealing defects within the conditions and constraints placed on the testing process.
- This takes time and effort, and the tester must choose carefully to maximize use of resources. A smart tester who wants to maximize use of time and resources knows that she needs to develop effective test cases for execution-based testing.
- The ability to develop effective test cases is important to an organization evolving toward a higher-quality testing process.
- It has many positive consequences.
- For example, if test cases are effective there is
  - ✓ A greater probability of detecting defects,
  - ✓ A more efficient use of organizational resources,
  - ✓ A higher probability for test reuse,
  - ✓ closer adherence to testing
  - ✓ project schedules and budgets,
  - ✓ The possibility for delivery of a higher-quality software product.
- There are two basic strategies that can be used to design test cases. These are called the black box and white box test strategies.

## 2.3 TEST CASE DESIGN STRATERGIES

- A tester considers the software-under test to be an opaque box. There is no knowledge of its inner structure. The tester does not know how it works. The tester only has knowledge of what it does.
- The size of the software-under-test using this approach can be,
  - ✓ A simple module,
  - ✓ Member function,
  - ✓ Cluster of a subsystem or
  - ✓ A complete software system.

- The description of behaviour or functionality for the software-under-test can be obtained from a formal specification or an Input/Process/Output Diagram (IPO), or a well-defined set of pre and post conditions.
- Another source for information is a requirements specification document, example SRS It describes the functionality of the software-under-test and its inputs and expected outputs.
- The tester provides the specified inputs to the software-under-test, runs the test and then determines if the outputs produced are equivalent to those in the specification.
- The black box approach only considers software behaviour and functionality so it is often called functional, or specification-based testing. This approach is useful for revealing requirements and specification defects.

**Black Box Testing Strategy**

- A tester considers the software-under test to be an opaque box. There is no knowledge of its inner structure. The tester does not know how it works. The tester only has knowledge of what it does.
- The size of the software-under-test using this approach can be,
  - ✓ A simple module,
  - ✓ Member function,
  - ✓ Cluster of a subsystem or
  - ✓ A complete software system.

- The description of behaviour or functionality for the software-under-test can be obtained from a formal specification or an Input/Process/Output Diagram (IPO), or a well-defined set of pre and post conditions.
- Another source for information is a requirements specification document, example SRSItdescribes the functionality of the software-under-test and its inputs and expected outputs.
- The tester provides the specified inputs to the software-under-test, runs the test and then determines if the outputs produced are equivalent to those in the specification.
- The black box approach only considers software behaviour and functionality so it is often called functional, or specification-based testing. This approach is useful for revealing requirements and specification defects.

**White Box Testing Strategy**

- The white box approach focuses on the inner structure of the software to be tested. To design test cases using this strategy the tester must have knowledge of that structure. The code, used in the system must be available.
- Test cases are designed to exercise all statements or true/false branches that occur in a module or member function.
- Since designing, executing, and analyzing the results of white box testing is very time consuming, this strategy is usually applied to smaller-sized pieces of software such as a module or member function.
- White box testing methods are useful for revealing design and code-based control, logic and sequence defects, initialization defects, and data flow defects.

- The smart tester knows that to achieve the goal of providing users with low-defect, high-quality software, both of these strategies should be used to design test cases.

## 2.4 USING THE BLACK BOX APPROACH TO TEST CASE DESIGN

- Black box test strategy considers only inputs and outputs as a basis for designing test cases. Infinite time and resources are not available to completely test all possible inputs. This is expensive even if the target software is a simple software unit.
- The goal for the smart tester is to effectively use the resources available by developing a set of test cases that gives the maximum yield of defects for the time and effort spent.
- Usually combinations of testing methods are used to detect different types of defects. Some methods have greater practicality than others.
- The methods of block box testing are
  - ✓ Equivalence class partitioning
  - ✓ Boundary value analysis
  - ✓ State transition testing
  - ✓ Cause and effect graphing
  - ✓ Error guessing

## 2.5 BOUNDARY VALUE ANALYSIS: EQUIVALENCE CLASS PARTITIONING

**Boundary Value Analysis**

- Boundary Value Analysis (BVA) method is useful for creating tests and test cases that are effective in catching defects that happen at boundaries.
- Boundary Value Analysis is based on the concept that the probability of defect is more towards the boundaries. To illustrate concept of errors that happen at boundaries let us consider a billing system, that offers volume discounts to customers.

| **No. of units Bought** | **Price per unit** |
|---|---|
| First 10 units (i.e. 1 to 10 units) | Rs. 5 |
| Next 10 units (11 to 20 units) | Rs. 4.75 |
| Next 10 units (21 to 31 units) | Rs. 4.50 |
| More than 30 units | Rs 4.00 |

***Table 2.5: Example Boundary Value Analysis***

- Generally it has been found that most defects happen around the boundaries, for example, when buying 9, 10, 11, 19, 20, 21, 29, 30, 31 and similar number of items.
- Another situation where boundary value testing is useful in detecting defects, is when there are internal limits place on certain resources, variable, or data structures.

- Consider a database management system, which caches the recently used data blocks in a shared memory area. Usually such a cached area is limited by a parameter that the user specifies at the time of starting of the system.
- Assume that the database is brought up specifying that the most recent fifty database buffers have to be cached. When these buffers are full and a 51st block needs to be cached, the least recently used buffer needs to be released after storing it in secondary memory. Both the operations, inserting the new buffer as well as freeing up the first buffer, happen at the "Boundaries".
- There are four possible cases to be tested
    - ✓ **Case 1: All buffers free for use**
    - ✓ **Case 2: After inserting two buffers and still having free buffers**
    - ✓ **Case 3: After inserting the last available buffer, no free buffers**
    - ✓ **Case 4: No free buffers and new buffer coming in. First buffer needs freeing.**
- Boundary value Analysis can be applied to white box testing also. Internal data structures like arrays, stacks, and queues needs to be checked for boundary or limit conditions, when there are linked lists used as internal structures the behaviour of the list at the beginning and end have to be tested thoroughly.

## 2.6 EQUIVALENCE PARTITIONING

- Equivalence partitioning is a software testing technique that evolves identifying a small set of representative input values that produce as many different output conditions as possible.
- The set of input values that generate one single expected output is called a partition. When behaviour of the software is same for a set of values, then the set is termed as an equivalence class or equivalence partition.
- One sample from the partitions enough for testing as the testing as the result of picking up some more values from the set will be the same and will not yield any addition defects. Since all the values produce equal and same output they are termed as equivalence portion.
- Testing by this technique involves,
    - ✓ Identifying all partitions for the complete set of inputs, output values for a product.
    - ✓ Picking up one member value from each partitions for testing to maximize complete coverage.

**Advantages**

- It gains good coverage with a small number of test cases.
- Redundancy of test is minimized by not repeating the same tests for multiple values in the same partition.

**Example: Life Insurance Premium Rates**

A life insurance company has base premium of Rs.50 for all ages. Based on the age group, an additional monthly premium has to be paid that is as listed in the table below,

| **Age Group** | **Additional Premium** |
|---|---|
| Under 35 | Rs.2 |
| 35 – 59 | Rs.3 |
| 60+ | Rs.6 |

***Table 2.6: Equivalence Classes for the life insurance premium example***

| **Sl.No** | **Equivalence Partitions** | **Type of Input** | **Test Data** | **Expected Results** |
|---|---|---|---|---|
| 1 | Age below 35 | Valid | 26,12 | Monthly Premium = Rs.52 |
| 2 | Age 35 – 39 | Valid | 37 | Monthly Premium = Rs.53 |
| 3 | Age above 60 | Valid | 65,90 | Monthly Premium = Rs.56 |
| 4 | Negative Age | Invalid | -23 | Warning Message |
| 5 | Age as 0 | Invalid | 0 | Warning Message |

***Table 2.7: Equivalence Classes for the life insurance premium example***

- The steps to prepare an equivalence partitions table are as follows,
  - ✓ Choose criteria for during the equivalence partitioning (Range, List of value, and so on).
  - ✓ Identify the valid equivalence classes based on the above criteria (Number of ranges allowed, values and so on)
  - ✓ Select a sample data from the partition
  - ✓ Write the expected result based on the requirements given.
  - ✓ Identify special values if any, and include them in the table.
  - ✓ Check to have expected results for all the cases prepared.
  - ✓ If the expected result is not clear for any particular test case, mark appropriately and escalate for corrective actions.

## 2.7 STATE BASED TESTING

- State or graph based testing is very useful in situations where:
  - ✓ **Language Processor** – Example: Compiler – The syntax of the language is automatically learnt as a state machine or a context free grammar represented by rail road diagram.
  - ✓ **Work flow modeling** – Depending on the current state and combinations of input variables, specific work flows are carried out, resulting in new output and new state.
  - ✓ **Dataflow modeling** – The system is modeled as a set of dataflow leading from the state to another.
- Consider an application that is required to validate a number according to the following simple rules:
  - ✓ A number can start with an optional sign
  - ✓ The optional sign can be followed by any number of digits
  - ✓ The digits can be optionally followed by a decimal point, represented by a period
  - ✓ If there is a decimal point, then there should be two digits after the decimal.

- ✓ Any number, whether or not it has a decimal point, should be terminated by a blank. The above rules can be represented in a state transition diagram

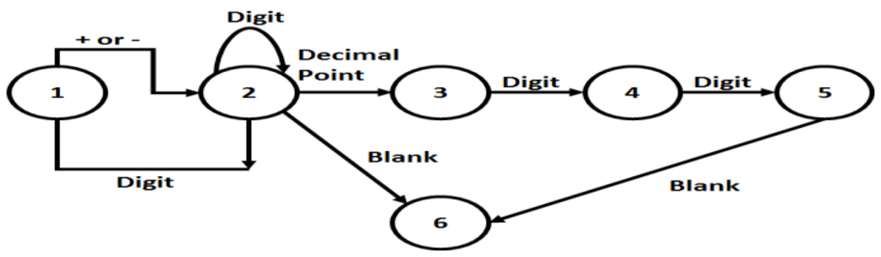


*Figure 2.1: An Example for State Transition*

| Current State | Input | Next State |
|---|---|---|
| 1 | Digit | 2 |
| 1 | + | 2 |
| 1 | - | 2 |
| 2 | Digit | 2 |
| 2 | Blank | 6 |
| 2 | Decimal Point | 3 |
| 3 | Digit | 4 |
| 4 | Digit | 5 |
| 5 | Blank | 6 |

*Table 2.8: State Transition Table*

- ❖ The state transition table can be used to derive test cases to test valid and invalid numbers. Valid test cases can be generated by:
  - ✓ Start from the start state (State 1)
  - ✓ Choose a path that leads to the next state
  - ✓ If invalid input is encountered, in a given state, generate an error, condition test case.
  - ✓ Repeat the process till you reach the final state. (State 6)
- ❖ A general outline for using state based testing method with respect to language processors is :

- ✓ Identify the grammar for the scenario. In the above example, we have represented using state machine diagram.
- ✓ Design test cases corresponding to each valid state input combination.
- ✓ Design test cases corresponding to the most common invalid combinations of state-input.

## 2.8 CAUSE-EFFECT GRAPHING

- ❖ Cause-and-effect graphing is a technique that can be used to combine conditions and derive an effective set of test cases that may disclose inconsistencies in a specification.
- ❖ The specification must be transformed into a graph that resembles a digital logic circuit. The tester should have knowledge of Boolean logic.
- ❖ The graph itself must be expressed in a graphical language.
- ❖ Developing the graph, especially for a complex module with many combinations of inputs, is difficult and time consuming.
- ❖ The graph must be converted to a decision table that the tester uses to develop test cases.
- ❖ The steps in developing test cases with a cause-and-effect graph are as follows:
    - ✓ The tester must decompose the specification of a complex software component into lower-level units.
    - ✓ For each specification, the tester needs to identify causes and their effects. A cause is a distinct input condition or an equivalence class of input conditions. An effect is an output condition or a system transformation.
    - ✓ From the cause-and-effect information, a Boolean cause-and-effect graph is created. Nodes in the graph are causes and effects.
    - ✓ Causes are placed on the left side of the graph and effects on the right.
    - ✓ Logical relationships are expressed using standard logical operators such as AND, OR, and NOT, and are associated with arcs.
    - ✓ The graph may be annotated with constraints that describe combinations of causes and/or effects that are not possible due to environmental or syntactic constraints.
    - ✓ The graph is then converted to a decision table.
    - ✓ The columns in the decision table are transformed into test cases.

- ❖ Consider a specification for a module that allows a user to perform a search for a character in an existing string.
- ❖ The specification states that the user must input the length of the string and the character to search for.
- ❖ If the string length is out-of-range an error message will appear.
- ❖ If the character appears in the string, its position will be reported.
- ❖ If the character is not in the string the message "not found" will be output.

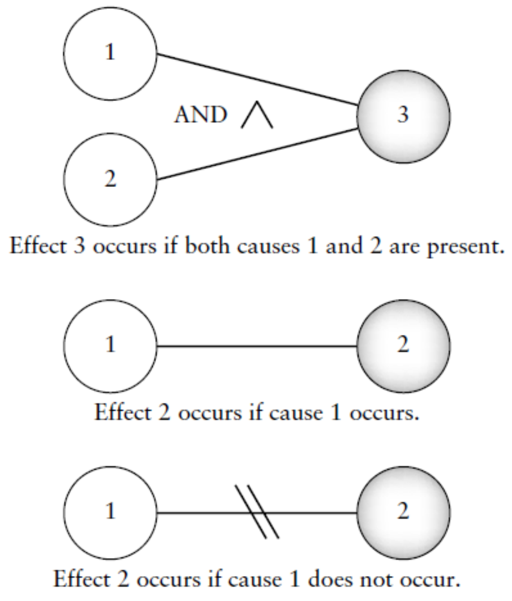

1
2
AND ∧
3
Effect 3 occurs if both causes 1 and 2 are present.
1
2
Effect 2 occurs if cause 1 occurs.
1
2
Effect 2 occurs if cause 1 does not occur.

***Figure 2.2: Samples of Cause-And-Effect Graph***

**The input conditions or causes are as follows:**

C1: Positive integer from 1 to 80

C2: Character to search for is in string

**The output conditions or effects are:**

E1: Integer out of range

E2: Position of character in string

E3: Character not found

**The rules or relationships can be described as follows:**

If C1 and C2, then E2.

If C1 and not C2, then E3.

If not C1, then E1.

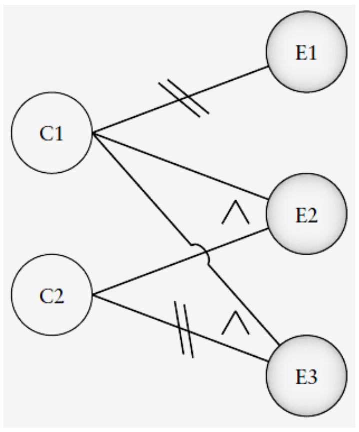


*Figure 2.3: Cause-And-Effect Graph for the character search example*

- ❖ The decision table reflects the rules and the graph and shows the effects for all possible combinations of causes.
- ❖ Given **n** causes this could lead to a decision table with **2n** entries, thus indicating a possible need for many test cases.
- ❖ A decision table will have a row for each cause and each effect.
- ❖ The entries are a reflection of the rules and the entities in the cause and effect graph.
- ❖ Enter "1" for a cause or effect that is present, "0" for a cause or effect that is not present, "-" for don`t care.

**Advantages**

- ❖ Development of the rules and the graph from the specification allows a thorough inspection of the specification.
- ❖ Any omissions, inaccuracies, or inconsistencies are likely to be detected.
- ❖ Exercising combinations of test data may not be considered using other black box testing techniques.

**Disadvantage**

- ❖ Problem is developing a graph and decision table when there are many causes and effects to consider.

## 2.9 COMPATIBILITY TESTING

- ❖ The test case results not only depend on the product for proper functioning, they depend equally on the infrastructure for delivering functionality.

- When infrastructure parameters are changed, the product is expected to still behave correctly and produce the desired or expected results.
- The infrastructure parameters could be of hardware, software, or other components.
- A black-box testing, not considering the effects of these parameters on the test case results, will be necessary be incomplete and ineffective, as it may not truly reflect the behaviour at a customer site. There is a need for compatibility testing.
- The parameter that generally affect the compatibility of the product are,
    - ✓ Processor and number of processors in the machine.
    - ✓ Architecture and characteristics of the machine.
    - ✓ Resource availability on the ,machine
    - ✓ Equipment, that the product is expected to work with.
    - ✓ Operating systems and Operating System Services
    - ✓ Middle tire infrastructure components such as web server, application server, network server
    - ✓ Back-end components such as database servers
    - ✓ Services that require special hardware cum software solutions
    - ✓ Any software used to generate product binaries
    - ✓ Various technological components used to generate components
- The above are just a few of the parameters. There are many more parameters that can affect the behaviour of the product features.
- In the above assumption of ten parameters and each parameter taking on four values, total number of combinations to be tested is $4^{10}$which is a large number and impossible to test exhaustively.
- In order to arrive at practical combinations of the parameters to be tested, a ***compatibility matrix*** is created.
- A compatibility Matrix has as its columns various parameters, the combinations of which have to be tested.
- Each row represents unique combinations of a specific set of values of the parameters.

***Table 2.9: Compatibility Matrix for Mail Application***

| Server | Application Server | Web Server | Client | Browser | MS-Office | Mail Server |
|---|---|---|---|---|---|---|
| Windows 2000 Advanced Server, with SP4 Microsoft SQL Server 2000 with SP3a | Windows 2000 advanced server with SP4 and .NET framework 1.1 | IIS 5.0 | Win 2K professional and Win 2K terminal server | IE 6.0 and IE 5.5 SP2 | Office 2K and Office XP | Exchange 5.5 and 2K |

| | | | | | | |
|---|---|---|---|---|---|---|
| Windows 2000 Advanced Server with SP4 Microsoft SQL server 2000 with SP3a | Windows 2000 advanced server with SP4 and .NET framework 1.1 | IIS 5.0 | Win 2K professional and Win 2K terminal server | Netscape 7.0.7.1, Safari and Mozilla | Office 2K and Office XP | Exchange 5.5 and 2K |

- ❖ Some of the common techniques that are used for performing compatibility testing, using a compatibility table are,
  1. Horizontal combination
  2. Intelligent sampling

**Horizontal Combination**

- ❖ Machines or environments are setup for each row and the set of product features are tested using each of these environments.

**Intelligent Sampling**

- ❖ In the horizontal combination method, each feature of the product has to be tested with each row in the compatibility matrix. This involves huge effort and time.
- ❖ To solve this problem, combinations of infrastructure parameter are combined with the set of features intelligently and tested.
- ❖ When these are problem due to any of the combinations then the test cases are executed, exploring the various permutations and combinations.
- ❖ The selection of intelligent sampling is based on information collected on the set of dependencies of the product with the parameters.
- ❖ If the product results are less dependent on a set of parameter, then they are removed from the list of intelligent samples.
- ❖ All other parameters are combined and tested. This method significantly reduces the number of permutation and combinations for the test cases.
- ❖ Compatibility testing not only includes parameter that is outside the product. But also includes some parameter that a part of the product.
- ❖ The compatibility testing for a product involving parts of itself can be further classified into two types
  - ✓ Backward compatibility testing
  - ✓ Forward compatibility testing

## Backward Compatibility Testing

- ❖ The testing that ensures the current version of the product continues to work with the older versions of the same product is called backward compatibility testing.
- ❖ The product parameters required for the backward compatibility matrix are tested.

**Forward Compatibility Testing**

- These are some provisions for the product to work with later version of the product and other infrastructure components, keeping future requirements in mind

## 2.10 USER DOCUMENTATION TESTING

- User documentation covers all of the following
  - ✓ Manuals,
  - ✓ User guides,
  - ✓ Installation guides,
  - ✓ Setup guides,
  - ✓ Read me file,
  - ✓ Software release notes,
  - ✓ Online help
- User documentation should have two objectives.
  - ✓ To check if what is stated in the document is available in the product
  - ✓ To check if what is there in the product is explained correctly in the document.
- When a product is upgraded the corresponding product documentation should also get updated as necessary to reflect any changes that may affect a user.
- User documentation testing focus on ensuring what is in the document exactly matches the product behaviour, by sitting in front of the system and verifying screen by screen, transaction by transaction and report by report.
- User documentation testing also checks for the language aspects of the documents like, spell check and the grammar.
- Testing these documents attains importance due to the fact that the users will have to refer to these manuals, installations, and the setup guides when they start using the software at their locations.
- Some of the benefits that the ensures from user documentation testing are,
  - ✓ User documentation testing aids in highlighting problems overlooked during review
  - ✓ High quality user documentation ensures consistency of documentation and product, thus minimizing possible defects reported by customers
  - ✓ Result in less difficult support calls.
  - ✓ New programmers and testers who join a project group can use the documentation to learn the external functionality of the product.
  - ✓ Customers need less training and can proceed more quickly to advanced training and product usage if the documentation is of high quality and is consistent with the product.

## 2.11 DOMAIN TESTING

- White box testing requires looking at the program code. Black box testing performs testing without looking at the program code but looking at the specifications.

- ❖ ***Domain testing*** can be considered as the next level of testing in which we do not look even at the specifications of a software product but are testing the product, purely based on domain knowledge and expertise in the domain of application.
- ❖ Domain testing is the ability to design and execute test cases that relate to the people who will buy and use the software. It helps in understanding the problem they are trying to solve and the ways in which they are using the software to solve them.
- ❖ Domain testing involves testing the product, not by going through the logic built into the product. The business flow determines the steps, not the software under test. This is also called ***"Business Vertical Testing"***. Test cases are written based on what the users of the software do on a typical day.

**Example**
Consider Cash withdrawal functionality in an ATM. The user performs the following actions.

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

- ❖ Domain testing requires domain knowledge rather than the knowledge of what the software specification contains or how the software is written. Thus domain testing can be considered as an extension of black box testing.
- ❖ The test engineers performing the domain testing are selected in such a way they have in depth knowledge of the business domain. This reduces the effort and time required for training the testers in domain testing and also increases the effectiveness of domain testing
- ❖ In the above example, a domain tester is not conserved about testing everything in the design, rather, he/she is interested in testing in the business flow. When you are testing as an end-user in a domain testing all you are concern with is whether you got the right amount or not.
- ❖ When the test case is written for domain testing, the intermediate steps would be missing. Just because those steps are missing does not mean they are not important. These "missing steps" (such as checking the denominations) are expected to be working before the start of domain testing.

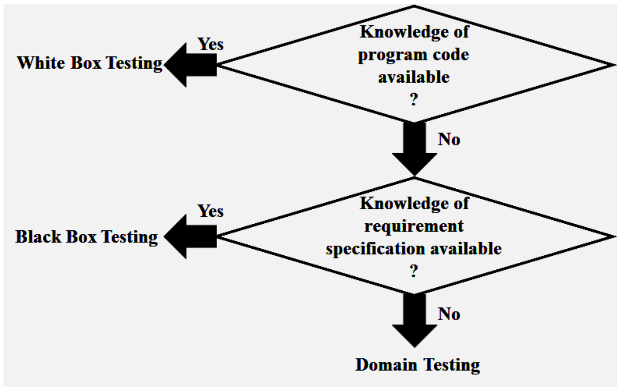

Knowledge of program code available ?
Yes
White Box Testing
No
Knowledge of requirement specification available ?
Yes
Black Box Testing
No
Domain Testing

***Figure 2.4:Context of White box,Black box and Domain Testing***

- Generally, domain testing is done after all components are integrated and after the product has been tested using other black box

## 2.12 RANDOM TESTING

- Each software module or system has an input domain from which test input data is selected. If a tester randomly selects inputs from the domain, this is called random testing.
- For example, if the valid input domain for a module is all positive integers between 1 and 100, the tester using this approach would randomly, or unsystematically, select values from within that domain; for example, the values 55, 24, 3 might be chosen.
- Some of the issues of random testing are as follows,
  - ✓ Are the three values adequate to show that the module meets its specification?
  - ✓ Should additional or fewer values be used to make the most effective use of resources?
  - ✓ Are there any input values, other than those selected, more likely to reveal defects?
  - ✓ Should any values outside the valid domain be used as test inputs?

**Advantages**

- Use of random test inputs may save some of the time and effort that other test input selection methods require.

**Disadvantages**

- Selecting test inputs randomly has less chance of producing an effective set of test data
- There are tools that generate random test data for stress tests. This type of testing can be very useful at the system level. Usually the tester specifies a range for the random value generator, or the test inputs are generated according to a statistical distribution associated with a pattern of usage.

## 2.13 REQUIREMENTS BASED TESTING

- Requirements testing deals with the validating the requirements given in the Software Requirement Specification (SRS) of the software system.
- Explicit requirement are stated and documented as part of the requirement specification. Implied or implicit requirements are those that are not documented but assumed to be incorporated in the system.
- The precondition for requirements testing is a detailed review of the requirements specification. Requirements review ensures that they are consistent, correct, complete and testable.
- This process ensures that some implied requirements are converted and documented as explicit requirements. This gives better clarity to requirements and making requirements based testing more effective.
- Some organizations follow variety of this method to bring more details into requirements.
- All explicit requirements and implicit requirements are collected and documented as "Test Requirement Specification (TRS)"

- Requirements based testing is conducted based on a TRS. It is based on the tester`s perspective also.
- A requirements specification for the lock and key example can be documents as given in the table.
- Requirements are tracked by Requirements Traceability Matrix (RTM). An RTM traces all the requirements from beginning through design, development and testing.
- This matrix evolves through the life cycle of the project.

***Table 2.1: Requirements Specification for lock and key system***

| **Sl. No** | **Requirements Identifier** | **Description** | **Priority (High, Med, Low)** |
|---|---|---|---|
| 1 | BR-01 | Inserting the key numbered 123-456 and turning it clockwise should lock. | H |
| 2 | BR-02 | Inserting the key numbered 123-456 and turning it anticlockwise should unlock | H |
| 3 | BR-03 | Only key numbers 123-456, can be used to lock and unlock | H |
| 4 | BR-04 | No other object should be used to lock | M |
| 5 | BR-05 | No other object should be used to unlock | M |
| 6 | BR-06 | The lock must not open even if hit by a big object | M |
| 7 | BR-07 | The lock and key must be made of metal and weight should be approximately 150 grams | L |
| 8 | BR-08 | Lock and unlock direction must be changeable, according to left and right opening doors | L |

***Table 1.2: Sample Requirements Traceability Matrix***

| **Requirements Identifier** | **Description** | **Priority (High, Med, Low)** | **Test Condition** | **Test Case ID** | **Phase of Testing** |
|---|---|---|---|---|---|
| BR-01 | Inserting the key numbered 123-456 and turning it clockwise should lock. | H | Use key 123-456 | Lock-001 | Unit, Component |
| BR-02 | Inserting the key numbered 123-456 and turning it anticlockwise should unlock | H | Use key 123-456 | Lock-002 | Unit, Component |
| BR-03 | Only key numbers 123-456, can be used to lock and unlock | H | Use key 123-456 to lock | Lock-003 | Component |

| | | | | | |
|---|---|---|---|---|---|
| | | | Use key 123-456 to Unlock | Lock-004 | |
| BR-04 | No other object should be used to lock | M | Use key 789-456 | Lock-005 | Integration |
| | | | Use hair pin | Lock-006 | |
| | | | Use Screw Driver | Lock-007 | |
| BR-05 | No other object should be used to unlock | M | Use key 789-456 | Lock-008 | Integration |
| | | | Use hair pin | Lock-009 | |
| | | | Use Screw Driver | Lock-010 | |
| BR-06 | The lock must not open even if hit by a big object | M | Use stone to break lock | Lock-011 | System |
| BR-07 | The lock and key must be made of metal and weight should be approximately 150 grams | L | Use weighing machine | Lock-012 | System |
| BR-08 | Lock and unlock direction must be changeable, according to left and right opening doors | L | | | Not Implemented |

- ❖ Tests for high priority requirements will performed before the low priority requirements. This ensures that the functionality with high risk is tested earlier. Defects reported by such testing can then be fixed as early as possible.
- ❖ The “Test Conditions” column lists the different ways of testing the requirements. These conditions can be grouped together to form a single test case or each test condition can be mapped to a test case.
- ❖ The “Test Case ID” column can be used to complete the mapping between test cases and their requirements. After the test case creation is completed, the RTM helps in identifying the relationship between the requirements and test case. The following combinations are possible.
  - ✓ One-to-one: For each requirement there is one test case
  - ✓ One-to-many: For each requirement there are many test cases
  - ✓ Many-to-one: For many requirements (a set of requirements) there is one test case.
  - ✓ Many-to-many: Many requirements can be tested by many test cases.
  - ✓ One-to-None: A requirement can have no test cases.
- ❖ A requirement is subjected to multiple phases of testing
  - ✓ Unit testing
  - ✓ Component testing
  - ✓ Integration testing
  - ✓ System testing

- ❖ An RTM plays a valuable role in requirements based testing; each of the requirements has to be tested. It prioritizes the requirements and helps the tester to test high-priority requirements first.
- ❖ Test conditions can be grouped before creation of test cases. Test conditions and test cases can be used as an estimate to estimate the resource, schedule and effort needed for testing.
- ❖ Some of the metrics that can be collected from the RTM matrix are,
  - ✓ Requirements addresses priority wise
  - ✓ Number of test cases requirements wise
  - ✓ Total Number of test cases prepared.
- ❖ After the test has completed, the test results can be used to collect the metric,
  - ✓ Total number of test cases passed
  - ✓ Total number of test cases failed
  - ✓ Total number of defects in the requirements
  - ✓ Number of requirements completed
  - ✓ Number of requirements pending.

## 2.14 POSITIVE AND NEGATIVE TESTING

- ❖ The purpose of positive testing is to prove that the product works as per specification and expectation. A product delivering an error when it is expected to give an error is also a part of positive testing. A set of positive test cases for the lock and key example are as follows,

| Requirements ID | Input 1 | Input 2 | Current State | Expected Output |
|---|---|---|---|---|
| BR-01 | Key 123-456 | Turn Clockwise | Unlocked | Locked |
| BR-01 | Key 123-456 | Turn Clockwise | Locked | No Change |
| BR-02 | Key 123-456 | Turn Anticlockwise | Unlock | No Change |
| BR-02 | Key 123-456 | Turn Anticlockwise | Locked | Unlocked |
| BR-04 | Hair Pin | Turn Clockwise | Locked | No Change |

***Table 2.3: Example of Positive Test Cases***

- ❖ Negative testing is done to show that the product does not fail when an unexpected input is given. The purpose of the negative testing is to try and break the system.
- ❖ Negative testing covers scenarios for which the product is not designed and coded. The tester needs to know the negative situations that may occur at the end-user level so that he can test the system.

| Sl. No | Input 1 | Input 2 | Current State | Expected Output |
|---|---|---|---|---|
| 1 | Some other lock`s key | Turn Clockwise | Lock | Lock |
| 2 | Some other lock`s key | Turn Anticlockwise | Unlock | Unlock |
| 3 | Thin piece of wire | Turn anticlockwise | Unlock | Unlock |

| | | | | |
|---|---|---|---|---|
| 4 | Hit with a stone | | Lock | Lock |

***Table 2.4: Negative Test Cases***

- ❖ The difference between the negative and positive test cases depends on the coverage. For positive testing if all documented requirements and test conditions are covered, then coverage can be said to be 100%. For negative testing there is no end at all. Negative testing requires a high level of creativity for the testers, in order to avoid failure at the customer's site.

## 2.15 USING WHITE BOX TESTING APPROACH TO TEST DESIGN

- ❖ The white box approach focuses on the inner structure of the software to be tested. To design test cases using this strategy the tester must have knowledge of that structure. The code, used in the system must be available.
- ❖ Test cases are designed to exercise all statements or true/false branches that occur in a module or member function.
- ❖ Since designing, executing, and analysing the results of white box testing is very time consuming, this strategy is usually applied to smaller-sized pieces of software such as a module or member function.
- ❖ White box testing methods are useful for revealing design and code-based control, logic and sequence defects, initialization defects, and data flow defects.
- ❖ The smart tester knows that to achieve the goal of providing users with low-defect, high-quality software, both of these strategies should be used to design test cases.

## 2.16 TEST ADEQUACY CRITERIA

- ❖ The goal for white box testing is to ensure that the internal components of a program are working properly. A common focus is on structural elements such as statements and branches.
- ❖ The tester develops test cases that exercise these structural elements to determine if defects exist in the program structure.
- ❖ Testers need a framework for deciding which structural elements to select as the focus of testing, for choosing the appropriate test data, and for deciding when the testing efforts are adequate enough to terminate the process with confidence that the software is working properly.
- ❖ Such a framework exists in the form of test adequacy criteria. a test data adequacy criterion is a stopping rule.
- ❖ The application scope of adequacy criteria also includes:
  - ✓ Helping testers to select properties of a program to focus on during test;
  - ✓ Helping testers to select a test data set for a program based on the selected properties;
  - ✓ Supporting testers with the development of quantitative objectives for testing;
  - ✓ Indicating to testers whether or not testing can be stopped for that program.

- If a test data adequacy criterion focuses on the structural properties of a program it is said to be a program-based adequacy criterion.
- A test data set is statement, or branch, adequate if a test set T for program P causes all the statements, or branches, to be executed respectively.
- The concept of test data adequacy criteria, and the requirement that certain features or properties of the code are to be exercised by test cases, leads to an approach called “coverage analysis,” which in practice is used to set testing goals and to develop and evaluate test data.
- Under some circumstances, the planned degree of coverage may be less than 100% possibly due to the following:
    - ✓ **The nature of the unit**
        - Some statements/branches may not be reachable.
        - The unit may be simple, and not mission, or safety, critical, and so complete coverage is thought to be unnecessary.
    - ✓ **The lack of resources**
        - The time set aside for testing is not adequate to achieve 100% coverage.
        - There are not enough trained testers to achieve complete coverage for all of the units.
        - There is a lack of tools to support complete coverage. Other project-related issues such as timing, scheduling, and marketing constraints
- The concept of coverage is not only associated with white box testing. Coverage can also be applied to testing with usage profiles.

## 2.17 STATIC TESTING Vs. STRUCTURALTESTING

### Static Testing

- Static testing is a type of testing which requires only the source code of the product, not the binaries or executable.
- Static testing does not involve executing the programs on computers but involves select people going through the code to find out whether:
    - ✓ The code works according to the functional requirement;
    - ✓ The code has been written in accordance with the design developed earlier in the project life cycle;
    - ✓ The code for any functionality has been missed out;
    - ✓ The code handles errors properly.

### Static Testing by Humans

- These methods rely on the principle of humans reading the program code to detect errors rather than computers executing the code to find errors. This process has several advantages.

**Advantages**

- Sometimes humans can find errors that computers cannot By making multiple humans read and evaluate the program, we can get multiple perspectives and therefore have more problems identified up front than a computer could.
- A human evaluation of the code can compare it against the specifications or design and thus ensure that it does what is intended to do.
- A human evaluation can detect many problems at one go and can even try to identify the root causes of the problems.
- By making humans test the code before execution, computer resources can be saved.
- Static testing minimizes the delay in identification of the problems.

- There are multiple methods to achieve static testing by humans. They are (in the increasing order of formalism) as follows:
  - ✓ Desk checking of the code
  - ✓ Code walkthrough
  - ✓ Code review
  - ✓ Code inspection

**Desk checking**

- Normally done manually by the author of the code, desk checking is a method to verify the portions of the code for correctness.
- Such verification is done by comparing the code with the design or specifications to make sure that the code does what it is supposed to do and effectively.
- This method of catching and correcting errors is characterized by:
  - ✓ No structured method or formalism to ensure completeness and
  - ✓ No maintaining of a log or checklist

**Advantages**

- The programmer who knows the code and the programming language very well is well equipped to read and understand his or her own code.
- It is done by one individual; there are fewer scheduling and logistics overheads.
- The defects are detected and corrected with minimum time delay.

**Structural Testing**

- Structural testing, also known as glass box testing or white box testing is an approach where the tests are derived from the knowledge of the software's structure or internal implementation.
- The other names of structural testing include clear box testing, open box testing, logic driven testing or path driven testing.

**Structural Testing Techniques:**

- **Statement Coverage -** This technique is aimed at exercising all programming statements with minimal tests.

- **Branch Coverage -** This technique is running a series of tests to ensure that all branches are tested at least once.
- **Path Coverage -** This technique corresponds to testing all possible paths which means that each statement and branch are covered.

**Advantages of Structural Testing:**

- Forces test developer to reason carefully about implementation
- Reveals errors in "hidden" code
- Spots the Dead Code or other issues with respect to best programming practices.

**Disadvantages of Structural Box Testing:**

- Expensive as one has to spend both time and money to perform white box testing.
- Every possibility that few lines of code is missed accidentally. In depth knowledge about the programming language is necessary to perform white box testing.

## 2.18 CODE FUNCTIONAL TESTING

- Code Functional Testing is a testing technique that is used to test the features/functionality of the system or Software, should cover all the scenarios including failure paths and boundary cases.
- Tests for correct functionality at all levels including modules, classes, system, or interfaces. It tests against the system requirements specification.
- The other major Functional Testing techniques include:
  - ✓ Unit Testing
  - ✓ Integration Testing
  - ✓ Smoke Testing
  - ✓ User Acceptance Testing
  - ✓ Localization Testing
  - ✓ Interface Testing
  - ✓ Usability Testing
  - ✓ System Testing
  - ✓ Regression Testing
  - ✓ Globalization Testing
- In functional testing basically the testing of the functions of component or system is done. It refers to activities that verify a specific action or function of the code.
- Functional test tends to answer the questions like “can the user do this” or “does this particular feature work”.
- This is typically described in a requirements specification or in a functional specification.

## 2.19 COVERAGE AND CONTROL FLOW GRAPHS

- The application of coverage analysis is associated with the use of control and data flow models to represent program structural elements and data. The logic elements most commonly considered for coverage are based on the flow of control in a unit of code.
- For example:
  - ✓ Program statements;
  - ✓ Decisions/branches;
  - ✓ Conditions;
  - ✓ Combinations of decisions and conditions;
  - ✓ Paths.
- All structured programs can be built from three basic primes-sequential (e.g., assignment statements), decision (e.g., if/then/else statements), and iterative (e.g., while, for loops).

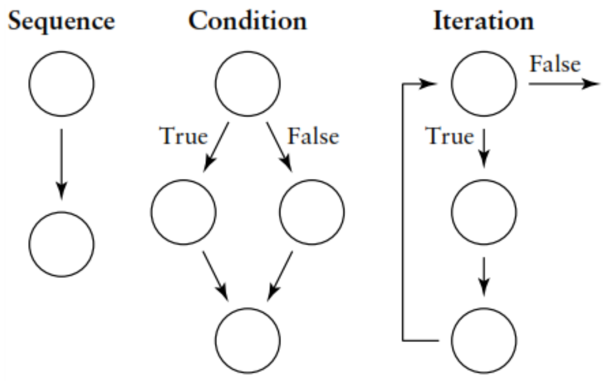


***Figure 2.5: Program Primes***

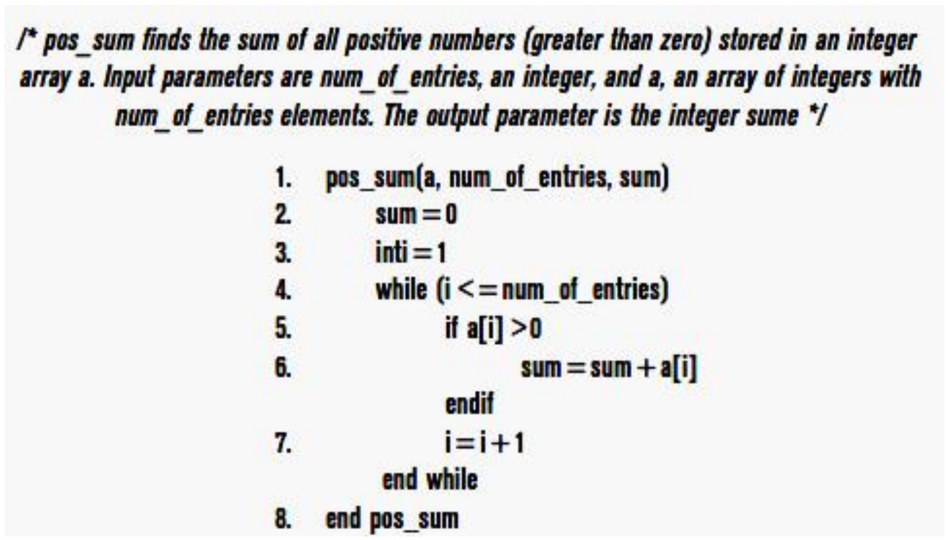
- There are commercial tools that will generate control flow graphs from code and in some cases from pseudo code.
- The tester can use tool support for developing control flow graphs especially for complex pieces of code.
- A control flow representation for the software under test facilitates the design of white box–based test cases as it clearly shows the logic elements needed to design the test cases using the coverage criterion of choice.

## 2.20 COVERING CODE LOGIC

- Logic-based white box–based test design and use of test data adequacy/ coverage concepts provide two major advantages for the tester:

  (i) Quantitative coverage goals can be proposed
  (ii) Commercial tool support is readily available to facilitate the tester's work

- ❖ If the goal is to satisfy the statement adequacy/ coverage criterion, then the tester should develop a set of test cases so that when the module is executed, all (100%) of the statements in the module are executed at least once.
- ❖ In addition to statements, the other logic structures are also associated with corresponding adequacy/coverage criteria.
- ❖ Complete decision coverage is considered to be a stronger coverage goal than statement coverage since its satisfaction results in satisfying statement coverage as well (covering all the edges in a flow graph will ensure coverage of the nodes).



*/* pos_sum finds the sum of all positive numbers (greater than zero) stored in an integer array a. Input parameters are num_of_entries, an integer, and a, an array of integers with num_of_entries elements. The output parameter is the integer sume */*

```
1.  pos_sum(a, num_of_entries, sum)
2.      sum = 0
3.      inti = 1
4.      while (i <= num_of_entries)
5.          if a[i] > 0
6.              sum = sum + a[i]
            endif
7.          i = i + 1
        end while
8.  end pos_sum
```

*Figure 2.6: Code sample with branch and loop.*

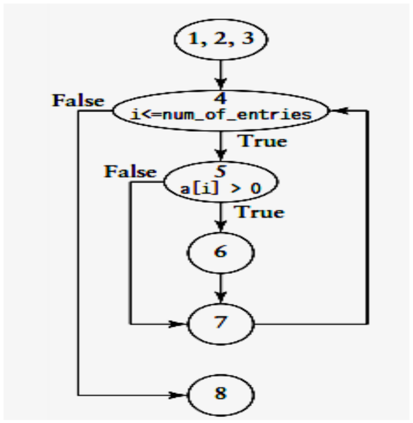
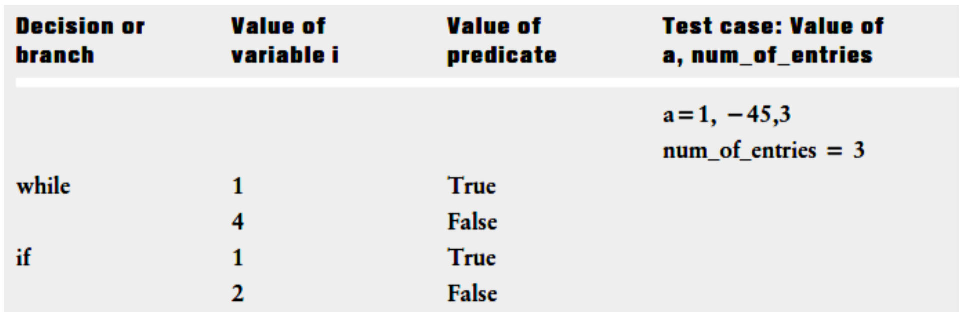


*Figure 2.7: A Control flow graph for the given code above*

- ❖ Decision (branch) coverage for the code example in the Figure requires test cases to be developed for the two decision statements, that is, the four true/false edges in the control flow graph of Figure given below.
- ❖ Input values must ensure execution the true/false possibilities for the decisions in line 4 (while loop) and line 5 (if statement).



| Decision or branch | Value of variable i | Value of predicate | Test case: Value of a, num_of_entries |
|---|---|---|---|
| | | | a = 1, −45,3 |
| | | | num_of_entries = 3 |
| while | 1 | True | |
| | 4 | False | |
| if | 1 | True | |
| | 2 | False | |

***Table 2.10: A test case for the code that satisfies the decision coverage criterion.***

- ❖ If (*x* _ MIN and *y* _ MAX and (not INT Z)) ***has 3 conditions :***

  ***(i)*** *x* _ MIN,
  ***(ii)*** *y* _ MAX,
  ***(iii)*** Not INT Z.

- ❖ Decision coverage only requires that we exercise at leastonce all the possible outcomes for the branch or loop predicates *as a whole*, not for each individual condition contained in a compound predicate.

- ❖ A simple example is showing the test cases for a decision statement with a compound predicate.

```
if(age <65 and married == true)
do X
do Y ........
else
do Z
```

## 2.21 PATHS

- ❖ The control flow graph is an aid to white box test design, tools are available to generate control flow graphs. These tools calculate a value for a software attribute called McCabe's Cyclomatic Complexity V (G) from a flow graph.
- ❖ The Cyclomatic complexity attribute is very useful to a tester. The complexity value is usually calculated from the control flow graph (G) by the formula

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

- ❖ The value E is the number of edges in the control flow graph and N is the number of nodes. This formula can be applied to flow graphs where there are no disconnected components. The Cyclomatic complexity value of a module is useful to the tester in several ways.
- ❖ One of its uses is to provide an approximation of the number of test cases needed for branch coverage in a module of structured code. A path is a sequence of control flow nodes usually beginning from the entry node of a graph through to the exit node.
- ❖ An independent path is a special kind of path in the flow graph. Deriving a set of independent paths using a flow graph can support a tester in identifying the control flow features in the code and in setting coverage goals.

- A tester identifies a set of independent paths for the software unit by starting out with one simple path in the flow graph and iteratively adding new paths to the set by adding new edges at each iteration until there are no more new edges to add.
- The number of independent paths in a basis set is equal to the Cyclomatic complexity of the graph. The basis set is a special set of paths and does not represent all the paths in a module; it serves as a tool to aid the tester in achieving decision coverage.

## 2.22 CODE COMPLEXITY TESTING

- Cyclomatic complexity is a metric that quantifies the complexity of a program. A program is represented in the form of flow graph. A flow graph consists of nodes and edges.
- The steps to convert the standard flow chart into a flow graph to compute Cyclomatic complexity are:
    - ✓ Identify the predicates (or Decision point) in the program
    - ✓ Ensure that the predicates are simple. If there are look constructs, break the loop termination checks into simple predicates
    - ✓ Combine all sequential statements into a single node.
    - ✓ When a set of sequential statements are followed by a simple predicate combine all the sequential statements and the predicate check into one node and have two edges emanating from this one node. Such nodes with tw edges emanating from them are called “Predicate Nodes”.
    - ✓ Make sure that all the edges terminate at some node. Add a node to represent all the set of sequential statements at the end of the program.

## 2.23 ADDITIONAL WHITE BOX TESTING APPROACHES

**Unit Testing**

Unit Testing is one of the basic steps, which is performed in the early stages. Most of the testers prefer performing to check if a specific unit of code is functional or not. Unit Testing is one of the common steps performed for every activity because it helps in removing basic and simple errors.

**Static Analysis**

As the term says, the step involves testing some of the static elements in the code. The step is conducted to figure out any of the possible defects or errors in the application code.

The static analysis is an important step because it helps in filtering simple errors in the initial stage of the process.

**Dynamic Analysis**

Dynamic Analysis is the further step of static analysis in general path testing. Most of the people prefer performing both static and dynamic at the same time.

The dynamic analysis helps in analyzing and executing the source code depending on the requirements. The final stage of the step helps in analyzing the output without affecting the process.

**Statement Coverage**

Statement coverage is one of the pivotal steps involved in the testing process. It offers a whole lot of advantages in terms of execution from time to time.

The process takes place to check whether all the functionalities are working or not. Most of the testers use the step because it is designed to execute all the functions atleast once. As the process starts, we will be able to figure out the possible errors in the web application.

**Branch Testing Coverage**

The modern-day software and web applications are not coded in a continuous mode because of various reasons. It is necessary to branch out at some point in time because it helps in segregating effectively.

Branch coverage testing gives a wide room for testers to find quick results. It helps in verifying all the possible branches in terms of lines of code. The step offers better access to find and rectify any kind of abnormal behavior in the application easily.

**Security Testing**

It is a known fact that security is one of the primary protocol, which needs to be in place all the time. Most of the companies prefer having a regular security testing activity because of obvious reasons. It is essential to have a process in place to protect the application or software automatically.

Security testing is more like a process because it comes with a lot of internal steps to complete. It verifies and rectifies any kind of unauthorized access to the system. The process helps in avoiding any kind of breach because of hacking or cracking practices.

Security testing requires a set of techniques, which deal with a sophisticated testing environment.

**Mutation Testing**

The last step in the process and requires a lot of time to complete effectively. Mutation testing is generally conducted to re-check any kind of bugs in the system.

The step is carried out to ensure using the right strategy because of various reasons. It gives enough information about the strategy or a code to enhance the system from time to time.

## 2.24 EVALUATING TEST ADEQUACY CRITERIA

- Testers have to decide on which criterion to apply to a given item under test based on the nature of the item and the constraints of the test environment such as time, costs, and resources. The tester can use the test adequacy criterion hierarchy to select an appropriate criterion.
- Satisfying an adequacy criterion at the higher levels of the hierarchy implies a greater thoroughness in testing. The criterion at the top includes the entire criterion below it. Statement adequacy is the weakest of the test adequacy criteria.
- In many organizations statement coverage is not even included as a minimal testing goal. Each adequacy criterion has both strengths and weaknesses. Each is effective in revealing certain types of defects. A "Stronger" criterion usually requires more tester time, resources and higher testing costs. Criterion should be selected based on the testing conditions, and the nature of the software.
- Testers can use a set of axioms to
  - ✓ Recognize both strong and weak adequacy criteria
  - ✓ Focus attention on the properties that an effective test data adequacy criterion should exhibit
  - ✓ Select an appropriate criterion for the item under test
  - ✓ Stimulate thought for the development of new criteria

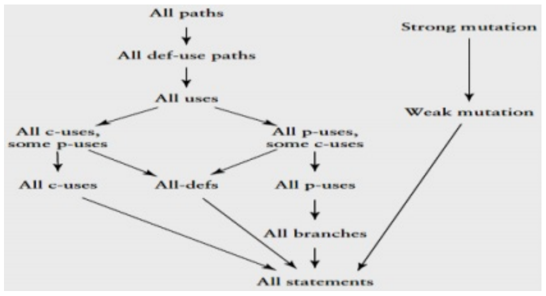


***Figure 2.8: Test Adequacy Criterion Hierarchy***

The **axioms/properties** are the following:

- **Applicability Property**

  "For every program there exists an adequate test set." for all programs we should be able to design an adequate test set that properly tests it.
  The test set may be very large so the tester will want to select representable points of the specification domain to test it.
- **Non-exhaustive Applicability Property**

  "For a program P and a test set T, P is adequately tested by the test set T, and T is not an exhaustive test set." A tester does not need an exhaustive test set in order to adequately test a program.
- **Monotonicity Property**

  "If a test set T is adequate for program P, and if T is equal to, or a subset of T', then T' is adequate for program P."
- **Inadequate Empty Set**

  "An empty test set is not an adequate test for any program." If a program is not tested at all, a tester cannot claim it has been adequately tested!

- **Anti-extensionality Property**

  "There are programs P and Q such that P is equivalent to Q, and T is adequate for P, but T is not adequate for Q."

  Just because two programs are semantically equivalent (they may perform the same function) does not mean we should test them using the same methods.

- **General Multiple Change Property**

  "There are programs P and Q that have the same shape, and there is a test set T such that T is adequate for P, but is not adequate for Q." The concept of shape to express a syntactic equivalence is used in this property.

  Two programs are the same shape if one can be transformed into the other by applying the set of rules shown below any number of times:

  - Replace relational operator r1 in a predicate with relational operator r2;
  - Replace constant c1 in a predicate of an assignment statement with constant c2;
  - Replace arithmetic operator a1 in an assignment statement with arithmetic operator a2.

- **Anti-decomposition Property**

  "There is a program P and a component Q such that T is adequate for P, T' is the set of vectors of values that variables can assume on entrance to Q for some t in T, and T' is not adequate for Q."

  Implications for these properties are

  - A routine that has been adequately tested in one environment may not have been adequately tested to work in another environment.
  - Although we may think of P, as being more complex than Q it may not be.Q may be more semantically complex.

- **Anti-composition Property**

  "There are programs P and Q, and test set T, such that T is adequate for P, and the set of vectors of values that variables can assume on entrance to Q for inputs in T is adequate for Q, but T is not adequate for P; Q (the composition of P and Q)."

  Adequately testing each individual program component in isolation does not necessarily mean that we have adequately tested the entire program.

- **Renaming Property**

  "If P is a renaming of Q, then T is adequate for P only if T is adequate for Q. A program P is a renaming of Q if P is identical to Q expect for the fact that all instances of an identifier, let us say a in Q have been replaced in P by an identifier, let us say b, where "b" does not occur in Q, or if there is a set of such renamed identifiers."

  An inessential change in a program such as changing the names of the variables should not change the nature of the test data that are needed to adequately test the program.

- **Complexity Property**

  "For every n, there is a program P such that P is adequately tested by a size n test set, but not by any size n - 1 test set." This means that for every program, there are other programs that require more testing.

- **Statement Coverage Property**

  "If the test set T is adequate for P, then T causes every executable statement of P to be executed."

Ensuring that their test set executed all statements in a program is a minimum coverage goal for a tester.
Using these new criteria, testers will be able to have greater confidence that the code under test has been adequately tested. Until then testers will need to continue to use exiting criteria such as branch- and statement-based criteria.

## QUESTION BANK

## PART-A

### TEST CASE DESIGN STRATERGIES

1. **Write the two basic testing strategies used to design test cases. [Nov 2012] (or) Identify the test case design strategies. [Nov 2009][May 2017]**
    - Black box testing
    - White box testing
2. **What are the knowledge sources for Black box testing? [May 2017][ Nov 2017]**
    - Requirements
    - Document specification
    - Domain knowledge
    - Defect analysis data

3. **What are the knowledge sources for White box testing?[May 2017] [ Nov 2017]**
    - High level design
    - Detailed design
    - Control flow graphs
    - Cyclomatic complexity

4. **List the methods of Black box testing.**
    - Equivalence class partitioning
    - Boundary value analysis
    - State transition testing
    - Cause and effect graphing
    - Error guessing

5. **List the methods of White box testing.**
    - Statement testing
    - Branch testing
    - Path testing
    - Data flow testing
    - Mutation testing
    - Loop testing

## USING BLACK BOX APPROACH TO TEST CASE DESIGN

**6. Mention the size of the software under test using black box approach.[May 2016]**

The size of the software-under-test using black box approach can be,

- A simple module,
- Member function,
- Cluster of a subsystem or
- A complete software system.

## RANDOM TESTING

**7. What is random testing?**

Each software system or module has an input domain from which test input data is selected. If a tester randomly selects input from the domain, this is called Random testing.

**8. What are the issues in random testing? (Or) Write down the limitations of random testing.**

Some of the issues of random testing are as follows:

- Are the three values adequate to show that the module meets its specification?
- Should additional or fewer values be used to make the most effective use of resources?
- Are there any input values, other than those selected, more likely to reveal defects?
- Should any values outside the valid domain be used as test inputs.

## REQUIREMENTS BASED TESTING

**9. What is RTM?**

Requirements are tracked by Requirements Traceability Matrix (RTM). An RTM traces all the requirements from beginning through design, development and testing. This matrix evolves through the life cycle of the project.

**10. What are the relationships identified between requirements and test cases?**

The following combinations are possible between requirements and test cases:

- One-to-one: For each requirement there is one test case
- One-to-many: For each requirement there are many test cases
- Many-to-one: For many requirements (a set of requirements) there is one test case.
- Many-to-many: Many requirements can be tested by many test cases.
- One-to-None: A requirement can have no test cases.

## BOUNDARY VALUE ANALYSIS

**11. What is boundary value analysis?**

Boundary Value Analysis (BVA) method is useful for creating tests and test cases that are effective in catching defects that happen at boundaries. Boundary Value Analysis is based on the concept that the probability of defect is more towards the boundaries.

## EQUIVALENCE CLASS PARTITIONING

**12. Define Equivalence class partitioning.**

If a tester is viewing the software-under-test as a black box with well defined inputs and outputs, a good approach to selecting test inputs is to use a method called Equivalence class partitioning.

**13. List the advantages of Equivalence class partitioning.**

- It eliminates the need for exhaustive testing, which is not feasible.
- It guides a tester in selecting a subset of test inputs with a high probability of detecting a defect.
- It allows a tester to cover a larger domain of inputs/outputs with a smaller subset selected from an Equivalence class.

## STATE-BASED TESTING

**14. Define State.**

A state is an internal configuration of a system or component. It is defined in terms of values assumed at a particular time for the variables that characterize the system or component.

**15. Define Finite-state machine.**

It is an abstract machine that can be represented by a state graph having a finite number of states and a finite number of transitions between states.

## CAUSE-EFFECT GRAPHING

**16. Define Cause effect graphing.**

It is a technique that can be used to combine conditions and derive an effective set of test cases that may disclose inconsistencies in a specification.

**17. What are the advantages of Cause effect graphing?**

- Development of the rules and the graph from the specification allows a thorough inspection of the specification. Any omissions, inaccuracies, or inconsistencies are likely to be detected.
- Exercising combinations of test data that may not be considered using other black box testing techniques.

## COMPATIBILITY TESTING

**18. What is Compatibility Matrix?**

A compatibility Matrix has as its columns various parameters, the combinations of which have to be tested. Each row represents unique combinations of a specific set of values of the parameters.

**19. Mention some of the parameters that affect compatibility testing.**

- Processor and number of processors in the machine.
- Architecture and characteristics of the machine.
- Resource availability on the machine
- Equipment, that the product is expected to work with.
- Operating systems and Operating System Services

**20. What are the types of compatibility testing?**

- Backward compatibility testing
- Forward compatibility testing

**21. What is backward compatibility testing?**

The testing that ensures the current version of the product continues to work with the older versions of the same product is called backward compatibility testing. The product parameters required for the backward compatibility matrix are tested.

**22. What is Forward compatibility testing?**

These are some provisions for the product to work with later version of the product and other infrastructure components, keeping future requirements in mind.

## USER DOCUMENTATION TESTING

**23. What are the objectives of user documentation testing?**

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

**24. What are the advantages of user documentation testing?**

- User documentation testing aids in highlighting problems overlooked during review
- High quality user documentation ensures consistency of documentation and product, thus minimizing possible defects reported by customers

## DOMAIN TESTING

**25. What is domain testing?**

Domain testing is the ability to design and execute test cases that relate to the people who will buy and use the software. It helps in understanding the problem they are trying to solve and the ways in which they are using the software to solve them. Domain testing involves testing the product, not by going through the logic built into the product.

## USING WHITE BOX APPROACH TO TEST DESIGN

**26. How Test Case can be designed Using White-Box Approach?**

The tester has knowledge of the internal logic structure of the software under test. The tester's goal is to determine if all the logical and data elements in the software unit are functioning properly. This is called the white box, or glass box, approach to test case design.

## TEST ADEQUACY CRITERIA

**27. What is Test data set?**

A test data set is statement, or branch, adequate if a test set T for program P causes all the statements, or branches, to be executed respectively.

**28. What is the application scope of adequacy criteria?**

The application scope of adequacy criteria also includes:

- Helping testers to select properties of a program to focus on during test;
- Helping testers to select a test data set for a program based on the selected properties;
- Supporting testers with the development of quantitative objectives for testing;
- Indicating to testers whether or not testing can be stopped for that program.

## STATIC TESTING Vs STRUCTURAL TESTING

**29. Define static testing.**

Static testing is a type of testing which requires only the source code of the product, not the binaries or executable.

**30. What are the methods used to achieve static testing?**

- Desk Checking
- Code walkthrough
- Code review
- Code inspection

**31. What is meant by desk checking? [May 2012]**

Desk checking is a method to verify the portions of the code for correctness. Such verification is done by comparing the code with the design or specifications to make sure that the code does what it is supposed to do and effectively.

**32. What is structural testing?**

Structural testing, also known as glass box testing or white box testing is an approach where the tests are derived from the knowledge of the software's structure or internal implementation.

## CODE FUNCTIONAL TESTING

**33. State the need for code functional testing in test case design [Nov 2012]**

- This testing is done against business requirements of an application
- Involves the complete integration system to evaluate the system's compliance with its specified requirements.

## COVERAGE AND CONTROL FLOW GRAPHS

**34. What is the use of coverage and control flow control?**

The logic elements most commonly considered for coverage are based on the flow of control in a unit of code. For example,

- Program statements;
- Decisions/branches;
- Conditions;
- Combinations of decisions and conditions;
- Paths.

## COVERING CODE LOGIC

**35. What are the advantages of covering code logic?**

- Quantitative coverage goals can be proposed, and
- Commercial tool support is readily available to facilitate the tester's work

## PATHS

**36. Define Path.**

A path is a sequence of control flow nodes usually beginning from the entry node of a graph through to the exit node.

**37. What is control graph? [Nov 2009]**

A control flow graph (CFG) is a representation, using graph notation, of all paths that might be traversed through a program during its execution.

## CODE COMPLEXITY TESTING

**38. Define code complexity testing. How it can happen? [May 2012]**

Cyclomatic complexity is a metric that quantifies the complexity of a program. A program is represented in the form of flow graph. A flow graph consists of nodes and edges.

## EVALUATING TEST ADEQUACY CRITERIA

**39. Mention some axioms/properties for evaluating test adequacy criteria.**

- Applicability Property
- Non-exhaustive Applicability Property
- Monotonicity Property
- Inadequate Empty Set
- Anti-extensionality Property
- General Multiple Change Property

**40. What are the errors covered by block box testing?**

- Interface errors and Performance errors
- Internal Data Structure errors
- Initialization and termination errors
- Accessing the database errors.

**41. What are the errors uncovered by block box testing? [May 2016]**

- Logic and sequence defects
- Initialization defects
- Data flow defects

**42. What are the basic primes for all structured program. [May 2016] [ Nov 2017]**

- Sequential ( e.g., Assignment statements)
- Condition (e.g., if/then/else statements)
- Iteration (e.g., while, for loops)

## PART B

**1.** Describe in detail about various test case design strategies. [ 16M]

**2.** Explain the concepts of equivalence class partitioning and boundary value analysis. [Nov 2012] [ Nov 2017] [16M]

**3.** Write short notes on random testing. [4M]

**4.** Explain in detail the requirement based testing with example. [16M][May 2017][4M]

**5.** Write short notes on state based testing. [8M] [May 2017][4 M]

6. Explain in detail cause-effect graphing. [16M]
7. Explain in detail about Compatibility Testing. [16M] (or) Explain the test factors that must be followed to design a customized test strategy. [Nov 2009] [8M]
8. Write short notes on user documentation testing. [May 2017][4M]
9. Write Short notes on Positive and Negative Testing.[ May 2017][4M]
10. Explain domain testing with example. [8M]
11. Write short notes on test adequacy criteria. [4M]
12. Write short notes on static testing and structural testing. [8M]
13. Write short notes on code functional testing. [4M]
14. Write short notes on Coverage and control flow graphs. [4M]
15. Discuss in detail about code coverage testing. [May 2012] [8M]
16. Explain how to evaluate test adequacy criteria in white box test approach. [16M]
17. Explain briefly about path and Cyclomatic complexity. [Nov 2009] [8M](or)Explain with neat flow chart code complexity testing. [May 2012] [8M](or)Describe the role of Oaths in white box testing and explain any two white box design approaches. [Nov 2009] [16M][May 2017] [ Nov 2017] [8 M]