Software Testing

Slides compiled by Sanghamitra De

Introduction

- Testing is the activity where the errors remaining from all the previous phases must be detected.
- Testing performs a very critical role for ensuring quality.

Error, Fault & Failure

- ***Error*** refers to the discrepancy between a computed, observed, or measured value and the true, specified, or theoretically correct value.
- ***Fault*** is a condition that causes a system to fail in performing its required function. A fault is the basic reason for software malfunction and is synonymous with the commonly used term *bug*.
- ***Failure*** is the inability of a system or component to perform a required function according to its specifications.
- A software failure occurs if the behavior of the software is different from the specified behavior.
- Failures may be caused due to functional or performance reasons.
- A failure is produced only when there is a fault in the system.
- Presence of a fault does not guarantee a failure.

Testing

- During the testing process, only failures are observed, by which the presence of faults is deduced.
- That is, testing only reveals the presence of faults.
- The actual faults are identified by separate activities, commonly referred to as "debugging."

Testing process

- The basic goal of the software development process is to produce software that has no errors or very few errors.
- In an effort to detect errors soon after they are introduced, each phase ends with a verification activity.
- Most of these verification activities in the early phases of software development are based on human evaluation and are thus unreliable.

Testing process

- Software typically undergoes changes even after it has been delivered.
- To validate that a change has not affected some old functionality of the system, regression testing is done.
- In regression testing, old test cases are executed with the expectation that the same old results will be produced.

Testing process

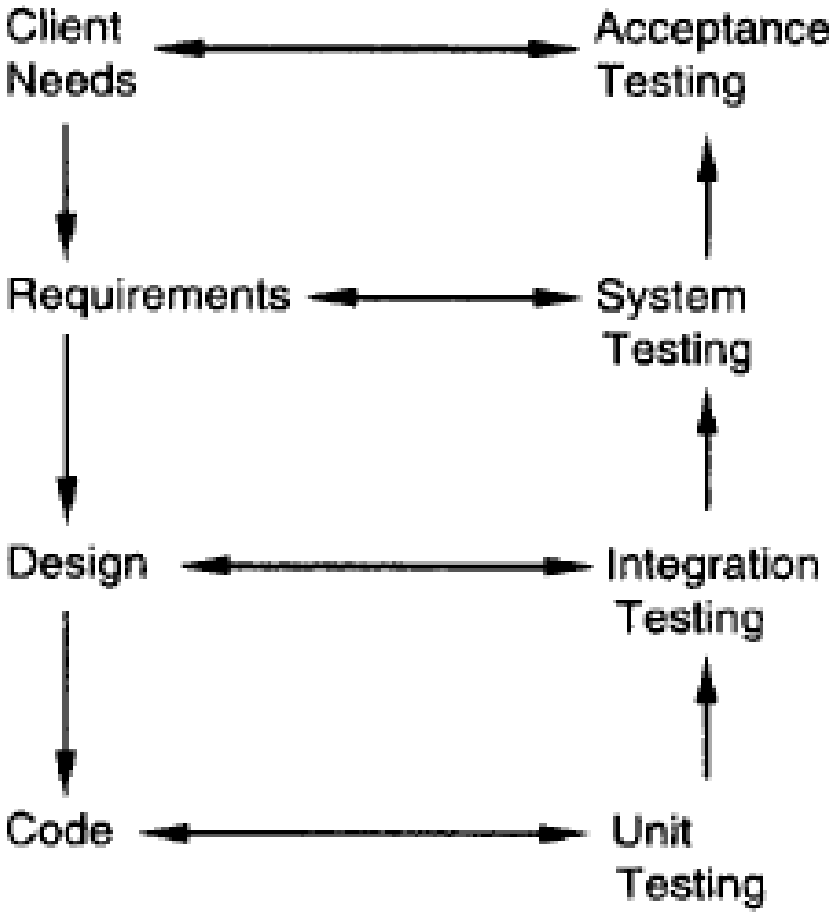
- Testing is the costliest activity in software development.
- It has to be carefully planned and the plan has to be properly executed.
- The testing process focuses on how testing should proceed for a particular project.

Levels of Testing

- Testing is usually relied upon to detect the faults remaining from earlier stages, in addition to the faults introduced during coding itself.
- Due to this, different levels of testing are used in the testing process; each level of testing aims to test different aspects of the system.

Levels of Testing

- Unit testing
- Integration testing
- System testing
- Acceptance testing



Relation of the faults introduced in different phases, and the different levels of testing.

Unit Testing

- Different modules are tested against the specifications produced during design for the modules.
- Unit testing is essentially for verification of the code produced during the coding phase, and hence the goal is to test the internal logic of the modules or smaller programs called ***units***.
- A unit may be a function, a small collection or functions, a class, or a small collection of classes.
- It is typically done by the programmer of the module.
- A module is considered for integration and use by others only after it has been unit tested satisfactorily.

Unit Testing

- Programmer, will execute the unit for a variety of test cases and study the actual behavior of the units being tested for these test cases.
- Based on the behavior, the tester decides whether the unit is working correctly or not.
- If the behavior is not as expected for some test case, then the programmer finds the defect in the program (an activity called debugging), and fixes it.
- After removing the defect, the programmer will generally execute the test case that caused the unit to fail again to ensure that the fixing has indeed made the unit behave correctly.

Unit Testing

- For a functional unit, unit testing will involve testing the function with different test data as input.
- Typically, the test data will include some data representing the normal case and special cases which might result in special or exceptional result.

Unit Testing

- A unit being tested is not a complete system but just a part, so it is not executable by itself.
- In its execution it may use other modules that have not been developed yet.
- This, requires use of drivers or stubs in unit testing.
- Drivers play the role of the "calling" module and are often responsible for getting the test data, executing the unit with the test data, and then reporting the result.
- Stubs are essentially "dummy" modules that are used in place of the actual module to facilitate unit testing.
- The need for stubs can be avoided, if coding and testing proceeds in a bottom-up manner.

Unit Testing

- In incremental coding unit testing needs to be performed every time the programmer adds some code.
- For this, automated scripts for unit testing are essential.
- For incremental testing it is desirable that the programmer develops this unit testing script and keeps enhancing it with additional test cases as the code evolves.
- If execution of the unit with the chosen test data is programmed then this program can be executed every time testing needs to be done.

Unit Testing

- In object-oriented programs, the unit to be tested is usually an object of a class.
- Testing of objects can be defined as the process of exercising the routines provided by an object with the goal of uncovering errors in the implementation of the routines or state of the object or both.
- State-based testing is a technique that can be used for unit testing an object.

Unit Testing

- A method is tested in all possible states that the object can assume, and after each invocation the resulting state is checked to see whether or not the method takes the object under test to the expected state.
- For state-based testing to be practical, the set of states in which a method is tested has to be limited.
- State modeling of classes can be used, or the tester can determine the important states of the object.
- After the different object states are decided, then a method is tested in all those states that form valid input for it.

Integration Testing

- In this many unit tested modules are combined into subsystems, which are then tested.
- The goal here is to see if the modules can be integrated properly.
- The emphasis is on testing interfaces between modules.
- This testing activity can be considered testing the design.

Integration Testing types

- Big Bang Approach
- All modules are tested in one go.
- Advantages:
  - ✓ All components are integrated at once and then the testing is processed
  - ✓ The method is suitable for a small system
- Disadvantage:
  - ✓ The testing teams have less time for execution in the testing phase as the integration testing is performed after the modules are designed

Integration Testing types

- Incremental Approach: This type of progressive approach is used to test two or more modules that are logically aligned and tested in a single batch.
- Sub-types: Bottom-up or Top-down
- Advantage:
  - ✓ In this type of approach, defects can be identified at the earliest
- Disadvantage:
  - ✓ This approach takes longer time as the drivers and stubs are developed and then used in the test

Integration Testing types

- Top Down Integration Testing: This process involves the testing of high level or the parent modules at first level, and then testing of the lower level or child modules, and then integrated together. Stubs, are used to simulate the data response of lower modules until they are completely tested and integrated.
- Advantages:
  - ✓ Since the integration test is performed by considering the real-time environment, the product that is tested maintains a good consistency
  - ✓ In this approach, a priority-based testing is performed for all critical modules at first
- Disadvantages:
  - ✓ In this approach, the need for having stubs is more
  - ✓ The basic functionality of the product is tested at the end of the cycle

Integration Testing types

- Bottom-Up Integration Testing: Here, first lower-level modules are tested before they are actually integrated with their parent modules. Drivers which simulate the data response of a connecting higher level or parent module is used instead of stub.
- Advantages:
  - ✓ In this method, testing and development are done simultaneously. This helps the product or the application to be efficient and meet the customer needs
  - ✓ Time is not wasted by waiting for all modules to be developed
- Disadvantages:
  - ✓ In this approach, the critical modules that control the flow of application are tested at the end, and this can lead to defects
  - ✓ Test drivers are required to be created for all modules at various levels (except for the top control)

Integration Testing types

- Hybrid/Sandwich Integration: This is a hybrid method that combines both bottom-up and top-down methods into single sandwich method.
- This type of testing ensures better and faster results.
- Advantage:
  - ✓ In this approach, there is a parallel test performed for the top and bottom layers
- Disadvantage:
  - ✓ Extensive test for the sub-systems is performed after integration

System Testing

- Here the entire software system is tested.
- The reference document for this process is the requirements document.
- The goal is to see if the software meets its requirements.
- This is essentially a validation exercise, and in many situations it is the only validation activity.

Acceptance Testing

- Acceptance testing is sometimes performed with realistic data of the client to demonstrate that the software is working satisfactorily.
- Testing here focuses on the external behavior of the system;
- Internal logic of the program is not emphasized.
- Mostly functional testing is performed at these levels.

Regression Testing

- When some changes are made to an existing system regression testing is performed.
- A modified software needs to be tested to make sure that the new features added do indeed work and that the modification has not had any undesired side effect of making some of the earlier services faulty.
- For regression testing, some test cases that have been executed on the old system are maintained, along with the output produced by the old system.
- These test cases are executed again on the modified system and its output compared with the earlier output to make sure that the system is working as before on these test cases.

Regression Testing

- . A regression testing script executes a suite of test cases.
- For each test case, it sets the system state for testing, executes the test case, determines the output or some aspect of system state after executing the test case, and checks the system state or output against expected values.
- These scripts are typically produced during *system testing*, as regression testing is generally done only for complete systems.

Regression Testing

- Regression testing of large systems can take a considerable amount of time, particularly because execution and checking of all the test cases cannot be automated.
- So, instead of executing the entire suite of test cases the system can be tested only with a subset of test cases.
- So, prioritization of test cases is required.
- For prioritization, generally more data about each test case is recorded, which is then used during a regression testing to prioritize.

Alpha Beta and Gamma Testing

- Terminology originated at IBM.
- The software release life cycle is the different stages that describe the stability of the software product/project.
- Pre-alpha: Pre alpha refers to all the activities performed on the software product prior to testing.
- These activities can include requirements analysis, software design, software developments and unit testing.

Alpha Beta and Gamma Testing

- QA can do following things in Pre-alpha stage:
  - ✓Requirement Analysis.
  - ✓Requirement Testing.
  - ✓Test Plan creation.
  - ✓Create the test scenarios and test cases.
  - ✓Test bed creation.
  - ✓Create the test execution plan.

Alpha Testing

- White box/Black box/Gray box software testing techniques are used to test the software product.
- Involves ***in-house testing*** of the product in presence of developers in laboratory setting.
- Alpha testing is done by developer himself, or separate testing team, or by client.
- All testing types are performed in alpha testing phase.
- Alpha testing phase ends with a feature freeze, indicating that no more features will be added to the software.

Alpha Testing

- Types of testing done by tester in Alpha phase:
  - ✓ Smoke testing.
  - ✓ Integration Testing.
  - ✓ System testing.
  - ✓ UI and Usability testing.
  - ✓ Functional Testing.
  - ✓ Security Testing.
  - ✓ Performance Testing.
  - ✓ Regression testing.
  - ✓ Sanity Testing.
  - ✓ Acceptance Testing.

Alpha Testing

- Purpose of the alpha testing is to validate the product in all perspective, which can be functional label, UI & usability label, security or performance label.

Beta Testing

- A beta test is the second phase of software testing in which ***a sampling of the intended audience tries the product out***.
- Beta testing can be considered "pre-release testing".
- Main objective behind the Beta testing is to get the feedback from the different groups of customers and check the compatibility of the product in different kind of networks, hardware's, impact of the different installed software on product, check the usability of the product.
- This is typically the first time that the software is available outside of the organization that developed it.
- The users of a beta version are called beta testers.

Beta Testing

- Developers release either a closed beta or an open beta.
- *Closed beta* versions are released to a select group of individuals for a user test and are invitation only.
- *Open betas* are from a larger group to the general public and anyone interested.
- The testers report any bugs that they find, and sometimes suggest additional features they think should be available in the final version.
- Open betas also serve the purpose of demonstrating a product to potential consumers.

Gamma Testing

- Gamma testing is done once the software is ready for release with specified requirements.
- This testing is done directly by skipping all the in-house test activities (no need to do all in-house quality check).
- The software is almost ready for final release.
- No feature development or enhancement of the software is undertaken.
- Gamma check is performed when the application is ready for release to the specified requirements; this check is performed directly without going through all the testing activities at home.

Testing Methods

- Black Box Testing
- Glass Box/White Box Testing

Black Box Testing

- The tester only knows the inputs that can be given to the system and what output the system should give.
- Also called functional or behavioral testing.
- There are a number of **techniques or heuristics that can be used to select test cases** that have been found to be very successful in detecting errors.

Black Box Testing methods

- Equivalence Class Partitioning
- Boundary Value Analysis
- Cause-Effect Graphing
- Pair-wise Testing
- State-Based Testing

Equivalence Class Partitioning

- The approach is to divide the input domain into a set of equivalence classes, so that if the program works correctly for a value then it will work correctly for all the other values in that class.
- An equivalence class is formed of the inputs for which the behavior of the system is specified or expected to be similar.
- Each group of inputs for which the behavior is expected to be different from others is considered a separate equivalence class.

Equivalence Class Partitioning

- Define equivalence classes for invalid inputs also.
- One common approach for determining equivalence classes:
  - ✓ If there is reason to believe that the entire range of an input will not be treated in the same manner, then the range should be split into two or more equivalence classes, each consisting of values for which the behavior is expected to be similar.
  - ✓ Another approach for forming equivalence classes is to consider any special value for which the behavior could be different as an equivalence class. Also, for each valid equivalence class, one or more invalid equivalence classes should be identified.

Equivalence Class Partitioning

- It is often useful to consider equivalence classes in the output. For an output equivalence class, the goal is to have inputs such that the output for that test case lies in the output equivalence class.
- During testing, it is important to test for each of these, that is, give inputs such that each of these three outputs are generated.
- Determining test cases for output classes may be more difficult, but output classes have been found to reveal errors that are not revealed by just considering the input classes.

Equivalence Class Partitioning

- Once equivalence classes are selected for each of the inputs, then the issue is to select test cases suitably.
- One strategy is to select each test case covering as many valid equivalence classes as it can, and one separate test case for each invalid equivalence class.
- A somewhat better strategy which requires more test cases is to have a test case cover at most one valid equivalence class for each input, and have one separate test case for each invalid equivalence class.
- In the latter case, the number of test cases for valid equivalence classes is equal to the largest number of equivalence classes for any input, plus the total number of invalid equivalence classes.

Equivalence Class Partitioning

- Consider a program that takes two inputs—a string s of length up to N" and an integer n. The program is to determine the top n highest occurring characters in s.
- One set of valid and invalid equivalence classes for this is shown

Input	Valid Equivalence Classes	Invalid Equivalence Classes
s	EQ1: Contains numbers EQ2: Contains lower case letters EQ3: Contains upper case letters EQ4: Contains special characters EQ5: String length between 0-N	IEQ1: non-ASCII characters IEQ2: String length > N
n	EQ6: Integer in valid range	IEQ3: Integer out of range

Table 10.1: Valid and invalid equivalence classes.

Boundary Value Analysis

- It has been observed that programs that work correctly for a set of values in an equivalence class fail on some special values.
- These values often lie on the boundary of the equivalence class.
- Test cases that have values on the boundaries of equivalence classes are therefore likely to be "high-yield" test cases, and selecting such test cases is the aim of the boundary value analysis.
- In boundary value analysis, we choose an input for a test case from an equivalence class, such that the input lies at the edge of the equivalence classes.
- Boundary values for each equivalence class, including the equivalence classes of the output, should be covered.
- Boundary value test cases are also called *"extreme cases."*

Boundary Value Analysis

- A boundary value test case is a set of input data that lies on the edge or boundary of a class of input data or that generates output that lies at the boundary of a class of output data.
- If the range is $0.0 < x < 1.0$, then the test cases are 0.0, 1.0 (valid inputs), and -0.1, and 1.1 (for invalid inputs).
- We should also consider the outputs for boundary value analysis.
- If an equivalence class can be identified in the output, we should try to generate test cases that will produce the output that lies at the boundaries of the equivalence classes.
- Furthermore, we should try to form test cases that will produce an output that does not lie in the equivalence class.

Boundary Value Analysis

- If there are multiple inputs, then how should the set of test cases be formed covering the boundary values?
- Suppose each input variable has a defined range.
- Then there are 6 boundary values—the extreme ends of the range, just beyond the ends, and just before the ends.
- If an integer range is min to max, then the six values are min — 1, min, min + 1, max— 1, max, max + 1.

Boundary Value Analysis

- Suppose there are n such input variables.
- Two strategies for combining the boundary values for the different variables in test cases:
- 1st strategy: select the different boundary values for one variable, and keep the other variables at some nominal value. And we select one test case consisting of nominal values of all the variables. In this case, we will have 6n + 1 test cases.

Boundary Value Analysis

- 2^{nd} strategy is to try all possible combinations for the values for the different variables.
- As there are 7 values for each variable (6 boundary values and one nominal value), if there are n variables, there will be a total of 7^n test cases.
- For two variables X and Y, the 13 test cases will be as shown

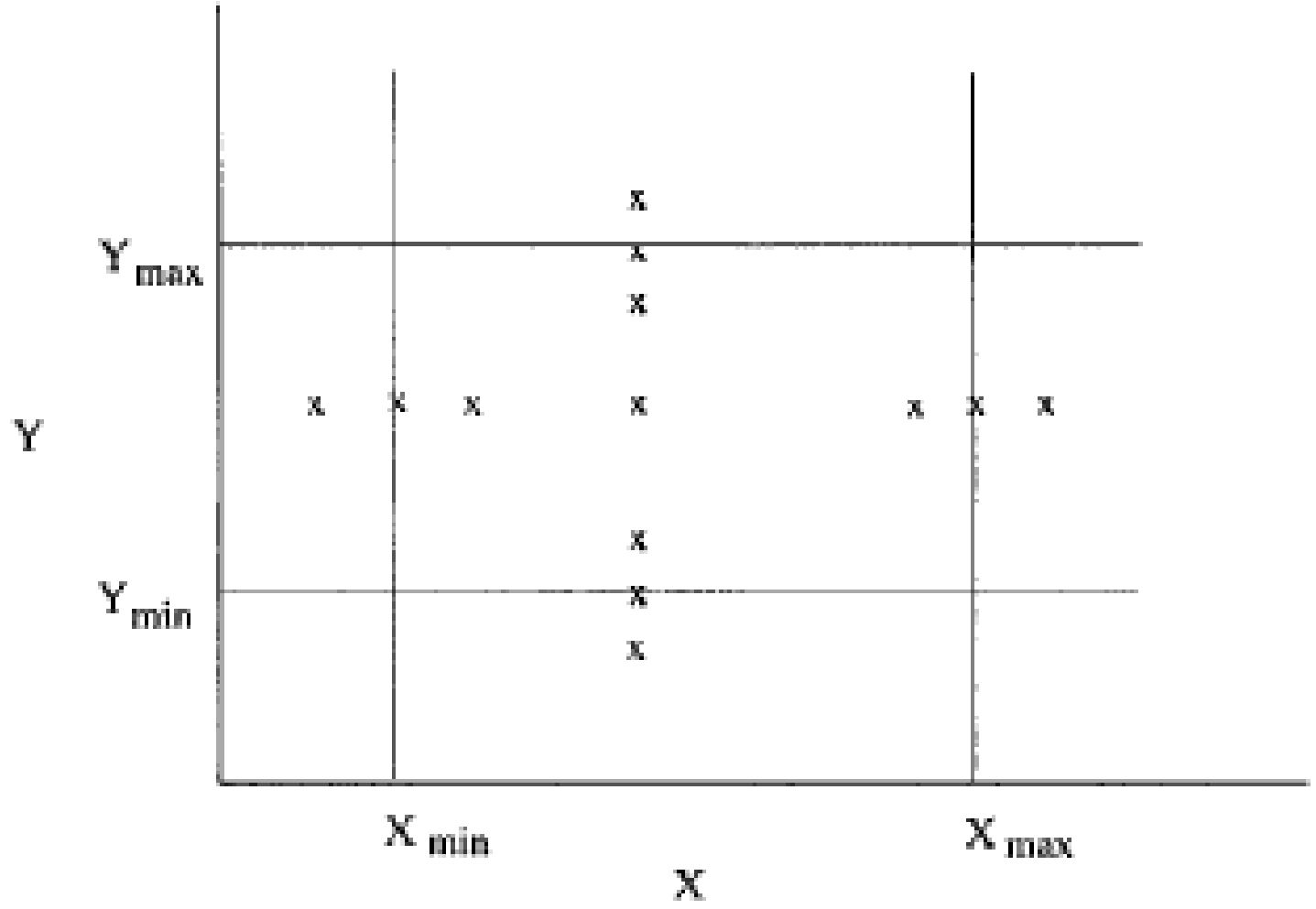


Figure 10.2: Test cases for BVA.

Cause-Effect Graphing

- Both equivalence class partitioning and boundary value concentrate on the conditions and classes of one input.
- They do not consider combinations of input circumstances that may form interesting situations that should be tested.
- One way to exercise combinations of different input conditions is to consider all valid combinations of the equivalence classes of input conditions.
- This approach will result in an unusually large number of test cases, many of which will not be useful for revealing any new errors.
- If there are n different input conditions, such that any combination of the input conditions is valid, we will have 2^n test cases.

Cause-Effect Graphing

- Cause-effect graphing is a technique that aids in selecting combinations of input conditions in a systematic way, such that the number of test cases does not become unmanageably large.
- It starts with identifying causes and effects of the system under testing.
- A ***cause*** is a distinct input condition, and an ***effect*** is a distinct output condition.
- Each condition forms a node in the cause-effect graph.
- The conditions should be stated such that they can be set to either true or false. For example, an input condition can be "file is empty," which can be set to true by having an empty input file, and false by a nonempty file.

Cause-Effect Graphing

- After identifying the causes and effects, for each effect we identify the causes that can produce that effect and how the conditions have to be combined to make the effect true.
- Conditions are combined using the Boolean operators "and," "or," and "not," which are represented in the graph by &, |, and ~.
- Then for each effect, all combinations of the causes that the effect depends on which will make the effect true are generated (the causes that the effect does not depend on are essentially "don't care").
- By doing this, we identify the combinations of conditions that make different effects true. A test case is then generated for each combination of conditions, which make some effect true.

Cause-Effect Graphing

- Suppose that for a bank database there are two commands allowed:

  - ✓ creditacct_numbertransaction_amount
  - ✓ debitacct_numbertransaction_amount

- The requirements are that if the command is credit and the acct_number is valid, then the account is credited.
- If the command is debit, the acct_number is valid, and the transaction_amount is valid (less than the balance), then the account is debited.
- If the command is not valid, the account number is not valid, or the debit amount is not valid, a suitable message is generated.

Cause-Effect Graphing

- Causes and effects from these requirements:
  - ✓ Causes:
    - C1. Command is credit
    - c2. Command is debit
    - c3. Account number is valid
    - c4. Transaction_amt is valid
  - ✓ Effects:
    - E1. Print "invalid command"
    - e2. Print "invalid account_number"
    - e3. Print "Debit amount not valid"
    - e4. Debit account
    - e5. Credit account

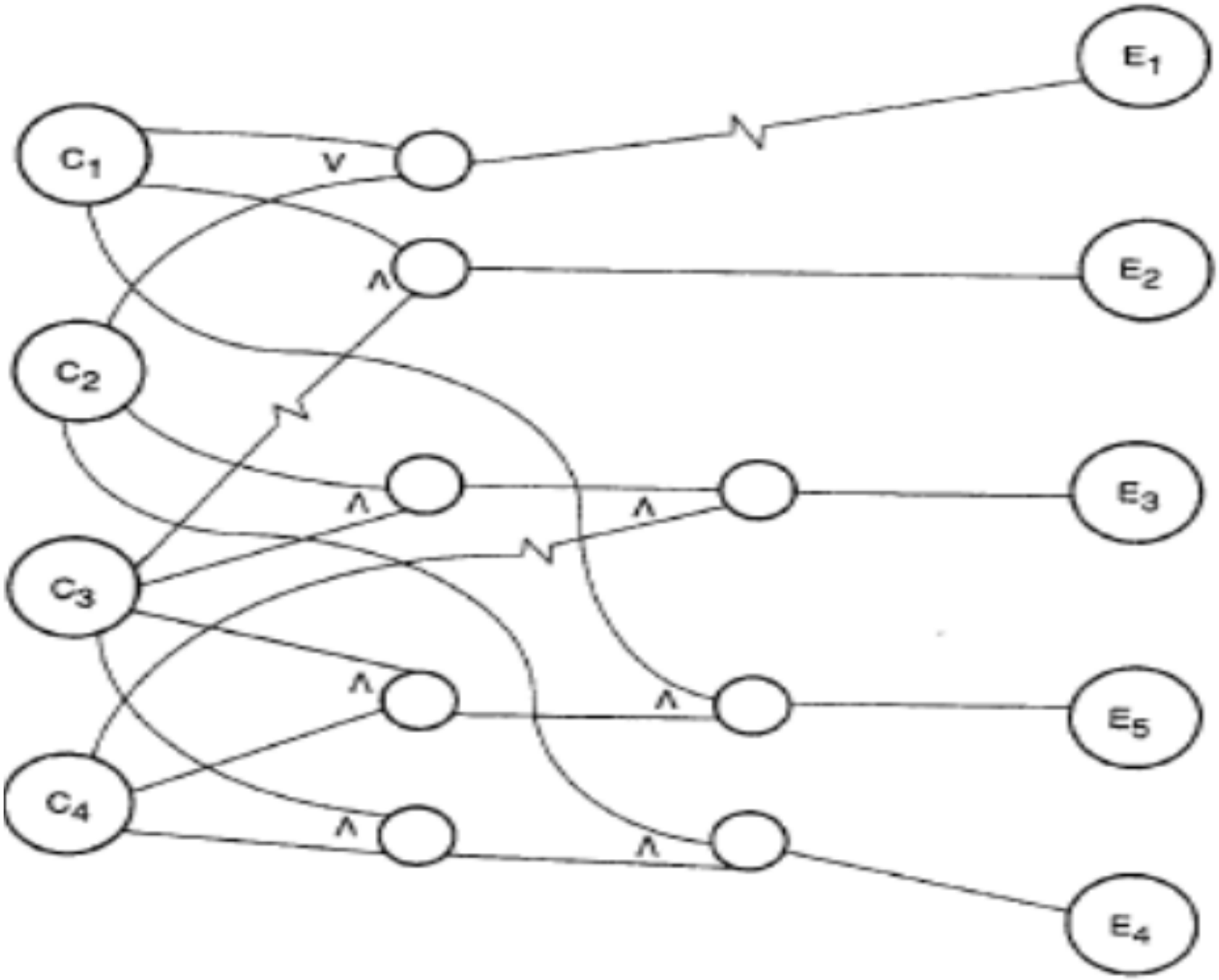


Figure 10.3: The cause-effect graph.

Cause-Effect Graphing

- From this graph, a list of test cases can be generated.
- The basic strategy is to set an effect to 1 and then set the causes that enable this condition.
- The condition of causes forms the test case.
- A cause may be set to false, true, or don't care (in the case when the effect does not depend at all on the cause).
- To do this for all the effects, it is convenient to use a decision table.

SNo.	1	2	3	4	5
c1	0	1	x	x	1
C2	0	x	1	1	x
c3	x	0	1	1	1
c4	x	x	0	1	1
e1	1				
e2		1			
e3			1		
e4				1	
e5					1

Decision table for the cause-effect graph.

Pair-wise Testing

- There are generally many parameters that determine the behavior of a software system.
- These parameters could be direct input to the software or implicit settings like those for devices.
- Many of the defects in software generally involve one condition, that is, some special value of one of the parameters. Such a defect is called *single-mode fault*.
- Simple examples of single mode fault are: a software not able to print for a particular type of printer, a software that cannot compute fare properly when the traveller is a minor, a telephone billing software that does not compute the bill properly for a particular country.

Pair-wise Testing

- Single-mode faults can be detected by testing for different values of different parameters. So, if there are n parameters for a system, and each one of them can take m different values then we can test for all the different values in m test cases.
- All faults are not single-mode and there are combinations of inputs that reveal the presence of faults.
- For example, a telephone billing software that does not compute correctly for night time calling (one parameter) to a particular country (another parameter).

Pair-wise Testing

- Multi-mode faults can be revealed during testing by trying different combinations of the parameter values—an approach called *combinatorial testing*.
- Full combinatorial testing is often not feasible.
- For a system with n parameters, each having m values, the number of different combinations is n^m.
- For a simple system with 5 parameters, each having 5 different values the total number of combinations is 3,125.
- Running 3125 test cases is infeasible

Pair-wise Testing

- For testing for double-mode faults, we need not test the system with all the combinations of parameter values, but need to test such that all combinations of values for each pair of parameters is exercised. This is called *pair-wise testing*.
- If there are n parameters, each with m values, then between each two parameter we have m*m pairs.

Pair-wise Testing

- The objective of pair-wise testing is to have a set of test cases that cover all the pairs.
- As there are n parameters, a test case is a combination of values of these parameters and will cover (n-1) + (n-2) + ... = n (n-1) /2 pairs.
- In the best case when each pair is covered exactly once by one test case, m^2 different test cases will be needed to cover all the pairs.

Pair-wise Testing

- A system has three parameters: A (operating system), B (memory size), and C (browser). Each of them can have three values which we will refer to as a1, a2, a3, b1, b2, b3, and c1, c2, c3.
  - ✓ Operating System: Windows, Solaris, Linux
  - ✓ Memory Size: 128M, 256M, 512M
  - ✓ Browser: IE, Netscape, Mozilla

- The total number of pair-wise combinations is 9*3 = 27.
- The number of test cases, however, to cover all the pairs is much less. A test case consisting of values of the three parameters covers three combinations (of A-B, B-C, and A-C).
- Hence, in the best case, we can cover all 27 combinations by 27/3=9 test cases.

A	B	C	Pairs
a1	b1	c1	(a1,b1) (a1,c1) (b1,c1)
a1	b2	c2	(a1,b2) (a1,c2) (b2,c2)
a1	b3	c3	(a1,b3) (a1,c3) (b3,c3)
a2	b1	c2	(a2,b1) (a2,c2) (b1,c2)
a2	b2	c3	(a2,b2) (a2,c3) (b2,c3)
a2	b3	c1	(a2,b3) (a2,c1) (b3,c1)
a3	b1	c3	(a3,b1) (a3,c3) (b1,c3)
a3	b2	c1	(a3,b2) (a3,c1) (b2,c1)
a3	b3	c2	(a3,b3) (a3,c2) (b3,c2)

Table 10.2: Test cases for pair-wise testing.

Pair-wise Testing

- Generating test cases to cover all the pairs is not a simple task.
- Efficient algorithms of generating the smallest number of test cases for pair-wise testing exist.
- Pair-wise testing is a practical way of testing large software systems that have many different parameters with distinct functioning expected for different values.

State-Based Testing

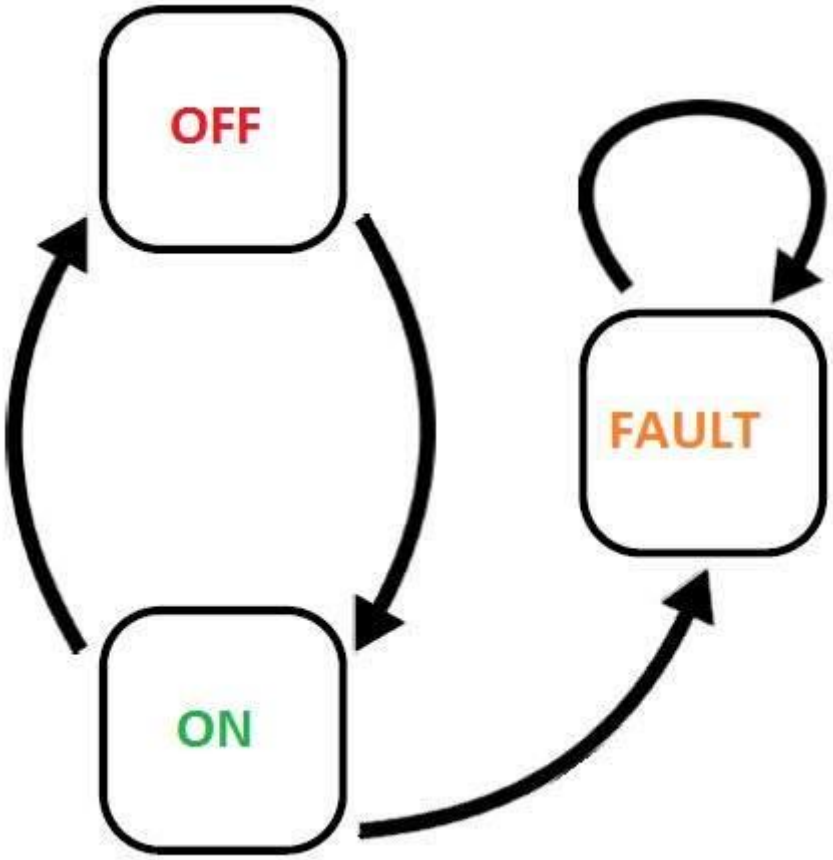
- Many systems behavior is state-based in that for identical inputs they behave differently at different times and may produce different outputs.
- The reason for different behavior is the state of the system, that is, the behavior and outputs of the system depend not only on the inputs provided, but also on the state of the system.
- The *state of the system depends on the past inputs* the system has received.
- In hardware the sequential systems fall in this category.
- In software, many large systems fall in this category as past state is captured in databases or files and used to control the behavior of the system.

State-Based Testing

- Any software that saves state can be modelled as a *state machine*.
- If the set of states of a system is manageable, a state model of the system can be built.
- A state model for a system has four components:
  - ✓ States- Represent the impact of the past inputs to the system.
  - ✓ Transitions- Represent how the state of the system changes from one state to another in response to some events.
  - ✓ Events- Inputs to the system.
  - ✓ Actions- The outputs for the events.

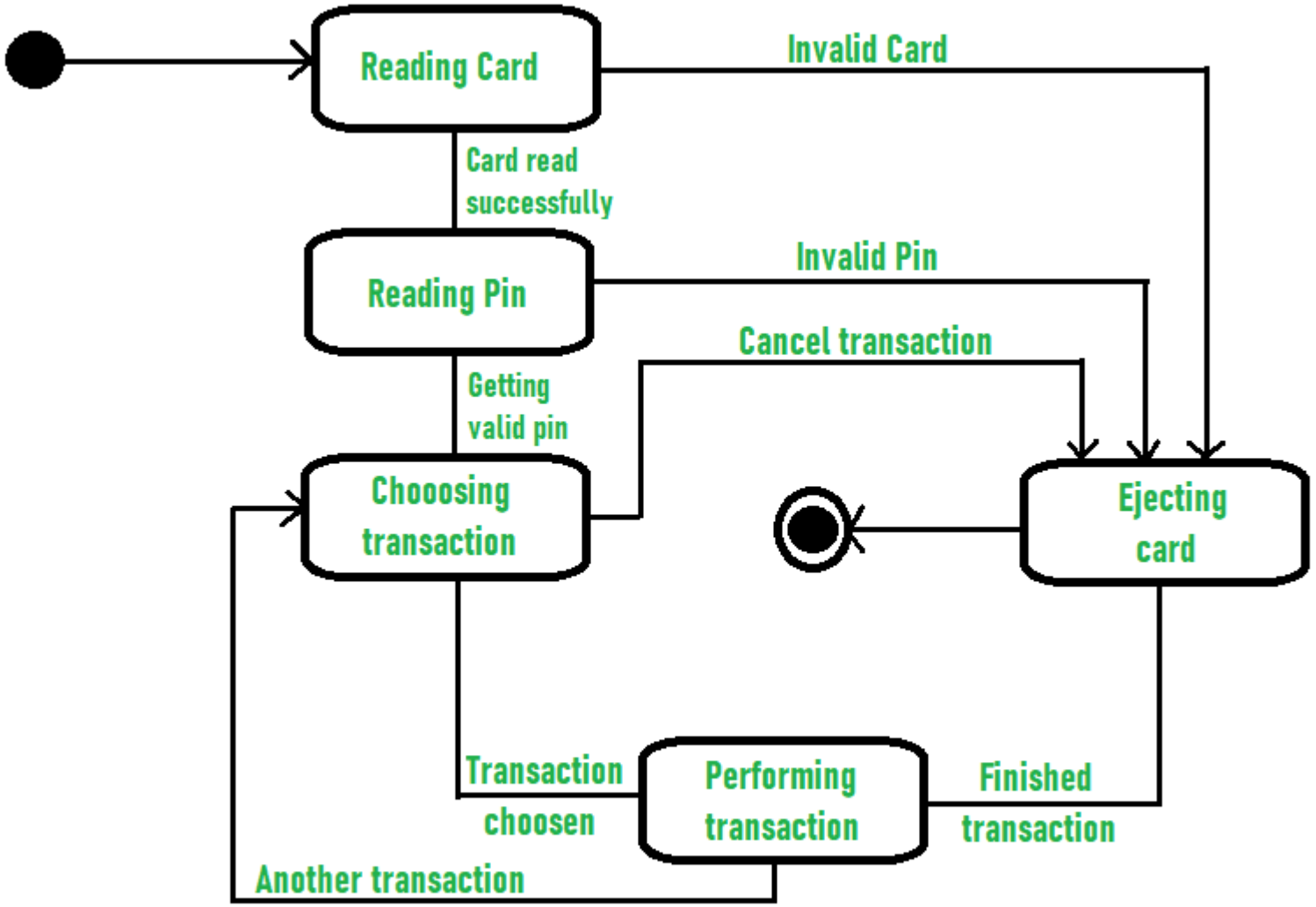
State-Based Testing

- State model shows what state transitions occur and what actions are performed in a system in response to events.
- When a state model is built from the requirements of a system, we can only include the states, transitions, and actions that are stated in the requirements.
- More information from the design specifications, gives a richer state model.
- Once the state model is built, we can use it to select test cases.



The tests are derived from the above state and transition and below are the possible scenarios that need to be tested.

Tests	Test 1	Test 2	Test 3
Start State	Off	On	On
Input	Switch ON	Switch Off	Switch off
Output	Light ON	Light Off	Fault
Finish State	ON	OFF	On



State Transition Diagram for ATM System

White Box/Glassbox Testing

- White-box testing is concerned with testing the implementation of the program.
- The intent of this testing is not to exercise all the different input or output conditions but to exercise the different programming structures and data structures used in the program.
- Whitebox testing is also called **structural testing**.

Whitebox Testing methods

- Control Flow-Based Criteria
- Data Flow-Based Testing
- Mutation Testing

Control Flow-Based Criteria

- Most common structure-based criteria are based on the control flow of the program.
- The control flow graph of a program is considered and coverage of various aspects of the graph are specified as criteria.

Control Flow-Based Criteria

- Let the control flow graph (or simply flow graph) of a program P be G.
- A node in this graph represents a block of statements that is always executed together, i.e., whenever the first statement is executed, all other statements are also executed.
- An edge (i, j) (from node i to node j) represents a possible transfer of control after executing the last statement of the block represented by node i to the first statement of the block represented by node j.
- A node corresponding to a block whose first statement is the start statement of P is called the start node of G, and a node corresponding to a block whose last statement is an exit statement is called an exit node.
- A path is a finite sequence of nodes $(n_1,n_2, ...,n_k)$,$k > 1$, such that there is an edge (n_i,n_{i+i}) for all nodes U_i in the sequence (except the last node n_k).
- A complete path is a path whose first node is the start node and the last node is an exit node.

Control Flow-Based Criteria

- The simplest coverage criteria is *statement coverage*, which requires that each statement of the program be executed at least once during testing.
- It requires that the paths executed during testing include all the nodes in the graph. This is also called the *all-nodes criterion*.
- This coverage criterion is not very strong, and can leave errors undetected.

Wrong program!!

```
int abs (x)
int x;
    {
        if (x >= 0) X = 0 - x;
            return (x)
    }
```

Control Flow-Based Criteria

- More general coverage criterion is *branch coverage*, which requires that each edge in the control flow graph be traversed at least once during testing.
- In other words, branch coverage requires that each decision in the program be evaluated to true and false values at least once during testing.
- Testing based on branch coverage is often called *branch testing*.
- The 100% branch coverage criterion is also called the *all-edges criterion*.

Control Flow-Based Criteria

- Branch coverage implies statement coverage, as each statement is a part of some branch. In other words, $C_{branch} => C_{stmt}$
- The trouble with branch coverage comes if a decision has many conditions in it (consisting of a Boolean expression with Boolean operators and/or).
- In such situations, a decision can evaluate to true and false without actually exercising all the conditions.

Wrong program!!

```
int check(x)
int x;
{
if ((x >= 0) && (x <= 200))
check = True;
else check = False;
}
```

Explanation

- *The module is incorrect, as it is checking for x < 200 instead of 100 (perhaps a typing error made by the programmer). Suppose the module is tested with the following set of test cases: {x = 5, x = -5}. The branch coverage criterion will be satisfied for this module by this set. However, error will not be revealed, and the behavior of the module is consistent with its specifications for all test cases in this set. Coverage criterion is satisfied, but the error is not detected. This occurs because the decision is evaluating to true and false because of the condition (x >= 0). The condition (x <= 200) never evaluates to false during this test, hence the error in this condition is not revealed.*

Control Flow-Based Criteria

- This is called the *path coverage criterion* or the *all-paths criterion*, and the testing based on this criterion is often called path testing.
- The difficulty with this criterion is that programs that contain loops can have an infinite number of possible paths.
- Furthermore, not all paths in a graph may be "feasible" in the sense that there may not be any inputs for which the path can be executed. It should be clear that $C_{path} => C_{branch}$.

Control Flow-Based Criteria

- As the path coverage criterion leads to a potentially infinite number of paths, some efforts have been made to suggest criteria between the branch coverage and path coverage.
- One method to limit the number of paths is to consider two paths the same if they differ only in their subpaths that are caused due to the loops.
- Even with this restriction, the number of paths can be extremely large.

Control Flow-Based Criteria

- Another approach is based on the cyclomatic complexity.
- The test criterion is that if the cyclomatic complexity of a module is V, then at least V distinct paths must be executed during testing.
- We have seen that cyclomatic complexity V of a module is the number of independent paths in the flow graph of a module.
- As these are independent paths, all other paths can be represented as a combination of these basic paths.
- These basic paths are finite, whereas the total number of paths in a module having loops may be infinite.

Control Flow-Based Criteria

- None of these criteria is sufficient to detect all kind of errors in programs.
- For example, if a program is missing some control flow paths that are needed to check for a special value (like pointer equals nil and divisor equals zero), then even executing all the paths will not detect the error.
- Similarly, if the set of paths is such that they satisfy the all-path criterion but exercise only one part of a compound condition, then the set will not reveal any error in the part of the condition that is not exercised.
- Hence, even the path coverage criterion, is not strong enough to guarantee detection of all the errors.

Data Flow-Based Testing

- In data flow-based testing approach, besides the control flow, information about where the variables are defined and where the definitions are used is also used to specify the test cases.
- Basic idea behind data flow-based testing is to make sure that during testing, definitions of variables and their subsequent use is tested.

Data Flow-Based Testing

- For data flow-based criteria, a definition-use graph {***def/use*** graph, for short) for the program is first constructed from the control flow graph of the program.
- A statement in a node in the flow graph representing a block of code has variable occurrences in it.
- A variable occurrence can be one of the following three types:
  - ***def*** represents the definition of a variable. The variable on the left-hand side of an assignment statement is the one getting defined.
  - ***c-use*** represents computational use of a variable. In an assignment statement, all variables on the right-hand side have a c-use occurrence. In a read and a write statement, all variable occurrences are of this type.
  - ***p-use*** represents predicate use. These are all the occurrences of the variables in a predicate (i.e., variables whose values are used for computing the value of the predicate), which is used for transfer of control.

Data Flow-Based Testing

- Data flow testing comprises of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of variables or data objects.
- Dataflow Testing focuses on the points at which variables receive values and the points at which these values are used.
- There are four ways data can be used: ***defined***, ***used in a predicate***, ***used in a calculation***, and ***killed***.

Data Flow-Based Testing

- Data Flow Testing helps us to pinpoint any of the following issues:
  - ✓A variable that is declared but never used within the program.
  - ✓A variable that is used but never declared.
  - ✓A variable that is defined multiple times before it is used.
  - ✓Deallocating a variable before it is used

Mutation Testing

- Mutation testing takes the program and creates many mutants of it by making simple changes to the program.
- The goal of testing is to make sure that during the course of testing, each mutant produces an output different from the output of the original program.
- It requires the set of test cases to be such that they can distinguish between the original program and its mutants.
- In mutation testing, faults of some pre-decided types are introduced in the program being tested.
- Testing then tries to identify those faults in the mutants.
- The idea is that if all these "faults" can be identified, then the original program should not have these faults; otherwise they would have been identified in that program by the set of test cases.

Mutation Testing

- Clearly this technique will be successful only if the changes introduced in the main program capture the most likely faults in some form.
- This is assumed to hold due to:
  - ✓competent programmer hypothesis
  - ✓coupling effect

Mutation Testing

- For a program under test P, mutation testing prepares a set of mutants by applying mutation operators on the text of P.
- The set of mutation operators depends on the language in which P is written.
- In general, a mutation operator makes a small unit change in the program to produce a mutant.

Mutation Testing

- Examples of mutation operators are: replace an arithmetic operator with some other arithmetic operator, change an array reference (say, from A to B), replace a constant with another constant of the same type (e.g., change a constant to 1), change the label for a goto statement, and replace a variable by some special value (e.g., an integer or a real variable with 0).

Mutation Testing

- First a set of test cases T is prepared by the tester, and P is tested by the set of test cases in T.
- If P fails, then T reveals some errors, and they are corrected.
- If P does not fail during testing by T, then it could mean that either the program P is correct or that P is not correct but T is not sensitive enough to detect the faults in P.
- To rule out the latter possibility (and therefore to claim that the confidence in P is high), the sensitivity of T is evaluated through mutation testing and more test cases are added to T until the set is considered sensitive enough for "most" faults.

Mutation Testing

- If P does not fail on T, the following steps are performed:
  1) Generate mutants for P. Suppose there are N mutants.
  2) By executing each mutant and P on each test case in T, find how many mutants can be distinguished by T. Let D be the number of mutants that are distinguished; such mutants are called dead.
  3) For each mutant that cannot be distinguished by T (called a live mutant), find out which of them are equivalent to P. That is, determine the mutants that will always produce the same output as P. Let E be the number of equivalent mutants.
  4) The mutation score is computed as D/ (N — E).
  5) Add more test cases to T and continue testing until the mutation score is 1.
- If no errors are detected and the mutation score reaches 1, then the testing is considered adequate by the mutation testing criterion.

Gray Box Testing

- Gray-box testing is a combination of white-box testing and black-box testing.
- The aim of this testing is to search for the defects if any due to improper structure or improper usage of applications.
- A black-box tester is unaware of the internal structure of the application to be tested, while a white-box tester has access to the internal structure of the application.
- A gray-box tester partially knows the internal structure, which includes access to the documentation of internal data structures as well as the algorithms used.

Test Metrics

- Reliability estimation can be used to decide whether enough testing has been done.
- Unreliability of software is primarily due to bugs or design faults in the software.
- Reliability is a probabilistic measure that assumes that the occurrence of failure of software is a random phenomenon.
- Let X be the random variable that represents the life of a system.
- Reliability of a system is the probability that the system has not failed by time t.

$$R(t)=P(X>t)$$

Test Metrics

- The reliability of a system can also be specified as the mean time to failure (MTTF).
- MTTF represents the expected lifetime of the system. From the reliability function, it can be obtained:

$$MTTF = \int_0^\infty R(x)dx.$$

Test Metrics

- One can obtain the MTTF from the reliability function but the reverse is not always true.
- Reliability function can, however, be obtained from the MTTF if the failure process is assumed to be Poisson, that is, the life time has an exponential distribution.
- With exponential distribution, if the failure rate of the system is known as λ, the MTTF is equal to $1/\lambda$

Test Metrics

- Reliability can also be defined in terms of the number of failures experienced by the system by time t.
- Clearly, this number will also be random as failures are random.
- With this random variable, we define the *failure intensity* $\lambda(t)$ of the system as the number of expected failures per unit time at time t.

Musa's reliability model

- This model focuses on failure intensity while modelling reliability.
- It assumes that the failure intensity decreases with time, that is, as (execution) time increases, the failure intensity decreases.
- This is because during testing, if a failure is observed, the fault that caused that failure is detected and the fault is removed.
- Consequently, the failure intensity decreases.

Testing Documents

- Test Plan
- Test Case Specifications

Test Plan

- A test plan is a general document for the entire project that defines the scope, approach to be taken, and the schedule of testing as well as identifies the test items for the entire testing process and the personnel responsible for the different activities of testing.
- The inputs for forming the test plan are:
  - ✓ project plan
  - ✓ requirements document
  - ✓ system design document.

Test Plan

- A test plan should contain the following:
  - ✓Test unit specification
  - ✓Features to be tested
  - ✓Approach for testing
  - ✓Test deliverables
  - ✓Schedule and task allocation

Test Plan

- A **test unit** is a set of one or more modules, together with associated data, that are from a single computer program and that are the object of testing.
- **Features to be tested** include all software features and combinations of features that should be tested. A software feature is a software characteristic specified or implied by the requirements or design documents. These may include *functionality, performance, design constraints, and attributes*.

Test Plan

- The **approach for testing** specifies the overall approach to be followed in the current project. The techniques that will be used to judge the testing effort should also be specified.
- **Testing deliverables** could be a list of test cases that were used, detailed results of testing including the list of defects found, test summary report, and data about the code coverage. In general, a test case specification report, test summary report, and a list of defects should always be specified as deliverables.

Test Plan

- Schedule should be consistent with the overall project schedule.
- For detailed planning and execution, the different tasks in the test plan should be enumerated and allocated to test resources who are responsible for performing them.

Test Case Specifications

- Test case specification gives, for each unit to be tested, all test cases, inputs to be used in the test cases, conditions being tested by the test case, and outputs expected for those test cases.
- Sometimes test case specifications document is also used to record the result of testing.
- In a round of testing, the outcome of all the test cases is recorded (i.e., pass or fail).
- Careful selection of test cases that satisfy the criterion and approach specified is essential for proper testing.
- Test case specifications contain not only the test cases, but also the rationale of selecting each test case (such as what condition it is testing) and the expected output for the test case.

Test Case Specifications

- Test cases are specified before they are used for testing because:
  - ✓ Effectiveness of testing depends very heavily on the exact nature of the test cases.
  - ✓ Constructing "good" test cases that will reveal errors in programs is still a very creative activity that depends a great deal on the ingenuity of the tester.
  - ✓ Evaluation of quality of test cases is done through "test case review.“
  - ✓ Process of sitting down and specifying all the test cases that will be used for testing helps the tester in selecting a good set of test cases.
  - ✓ Specifications can be used as "scripts" during regression testing, particularly if regression testing is to be performed manually.
  - ✓ Specification document eventually also becomes a record of the testing results.

Test Case Specifications

- Evaluation of quality of test cases is done through "test case review.“
- The test case specification document is reviewed, using a formal review process, to make sure that the test cases are consistent with the policy specified in the plan, satisfy the chosen criterion, and in general cover the various aspects of the unit to be tested.

Other testing types

- See notes

End