Software Engineering

Unit 4

Software Testing

Software Testing Fundamentals - Internal and External Views of Testing: White box testing, basis path testing - control structure testing - black box testing - unit testing - integration testing - regression testing - validation testing - system testing - security testing; Testing tools ; Debugging; Software Implementation: Coding Practices and Principles; Maintenance : Types.

* Software Testing Fundamentals

* Characteristics of Testable Software

(1) Operable - the better it works, the easier it is to test

(2) Observable - incorrect o/ps are easily identified, internal errors are automatically detected.

(3) Controllable - states & variables of the software can be controlled directly by the tester

(4) Decomposable - software is built from independent modules that can be tested independently

(5) Simple - program should be be functionally, structurally simple + simple code

(6) Stable - changes to software during testing are infrequent

(7) Understandable - architecture design is well understood, has documentation

* Test Characteristics

A good test:

→ has a high probability of finding an error

→ is not redundant

→ should be the 'best of breed' - tests that have the highest likelihood of uncovering a whole class of errors should be used

→ should be neither too simple nor too complex.

* Unit Testing

→ is of 2 types - white box and black box

A. White Box Testing

White box testing focuses on:

(i) internal working of the product

(ii) a close examination of procedural detail

(iii) test logical paths

(iv) test cases exercise specific sets of conditions & loops

→ Test all independent paths to check if they have been ③ exercised at least once

→ exercise all logical decisions

→ execute all loops at their boundaries

→ check internal data structures

* Basis Path Testing?

→ aims to derive a set of test cases that exercise each path at least once - helps ensure that every statement in the program is executed and that all logical conditions have been testing

Basis path testing is done with the following methods:/ terms:

A. Flow Graph Notation

→ A circle is a node - which is a statement

→ A node with a simple conditional expression is called a predicate node.

→ An edge is an arrow representing the flow of control in a specific direction

B. Independent Program Paths

→ defined as a path through the program from the start node till the end, moving along at least one edge that has not been traversed before.

C. Cyclomatic Complexity

→ provides a quantitative measure of the logical complexity of a program

→ defines the number of independent paths in the basic set

→ provides an upper bound for the number of test cases to be conducted.

→ It can be computed as:

(i) the number of regions

(ii) $V(G) = E - N + 2$ E = edges, N = nodes

(iii) $v(G) = P + 1$ P = predicate nodes

* Deriving the Basis Set and Test Cases

1. Draw flow graph from code
2. Determine cyclomatic complexity
3. Determine a set of independent paths
4. Prepare test cases that force their execution

Example Calculate cyclomatic complexity for the following code

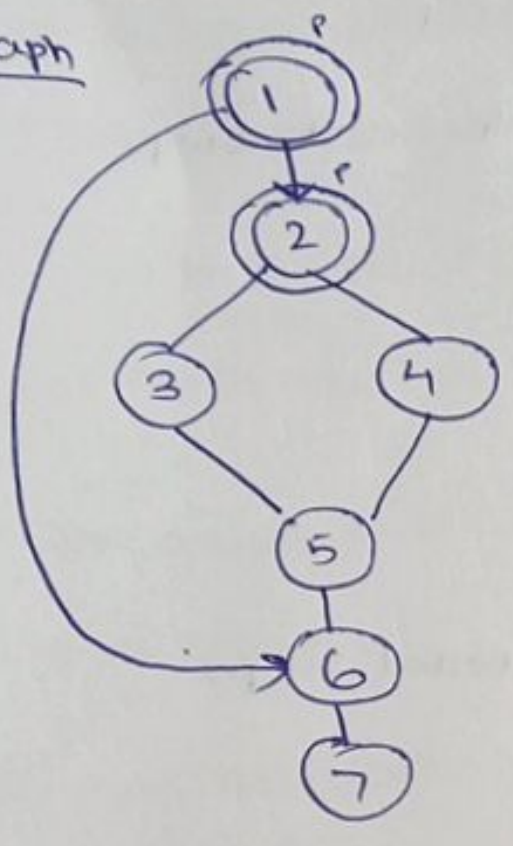
```
1. IF A=354
2. THEN IF B>C
3           THEN A=B
4           ELSE A=C
```

5 END IF

6 END IF

7 PRINT A

Ans Graph



Cyclomatic Complexity

= E - N + 2

= 8 - 7 + 2

= 3

(OR)

P + 1

= 2 + 1 = 3

* Loop Testing

→ A white box testing technique that focuses exclusively on the validity of loop constructs.

→ There are 4 different classes of loops:

(i) simple loops

(ii) nested loops

(iii) concatenated loops

(iv) unstructured loops

→ Testing occurs by varying the loop boundary conditions.

A. Testing Simple Loops

1. Skip loop
2. Make one pass
3. Make 2 passes
4. Make m passes, $m < n$
5. Make $n-1$, n, $n+1$ passes (n = max passes)

B. Testing Nested Loops

1. Start at the inner most loop, set all other values to minimum
2. Conduct simple loop tests for inner loops
3. Work outward to conduct a test for the next loop.
4. Add out of range/ excluded values

C. Testing Concatenated Loops

→ If simple loops, use method A

→ Otherwise, use method B, for nested loops.

D. Testing Unstructured Loops

→ Redesign the code to reflect more structured programming

→ Use Methods A, B or C.

* Black Box Testing?

→ used in later stages of testing after white box testing has been performed.

→ focuses more on functional requirements

→ Black box testing looks for:

(i) incorrect or missing functions
(ii) interface errors
(iii) errors in data structures or external data base access
(iv) behavior or performance errors
(v) initialization and termination errors

Black Box Testing Strategies

A. Equivalence Partitioning

→ divides the input domain of a program into classes of data from which classes are derived.

→ An ideal test case single-handedly uncovers a complete class of errors, reducing the total no. of test cases

→ Test case design is based on the evaluation of equivalence classes for an input condition

→ From each equivalence class, there are invalid & valid states.

Guidelines for Defining Equivalence Classes

(i) if there is a range - make 1 valid and two invalid equivalence classes

eg. i/p range = 1-10 → equ. classes {1..10} {x<1}, {x>10}

(ii) if there's a value - set one valid & 2 invalid conditions

eg. i/p value = 250 → equ. classes {250}, {x<250}, {x>250}

(iii) if its a set of values - specify one valid & one invalid set

eg If i/p set = {1,2,3} → equ. classes = {1,2,3}, {any other set}

(iv) if its a boolean value - specify one valid & one invalid condn

eg. i/p {true} → equ classes {true}, {false}

B. Boundary Value Analysis

→ A greater number of errors occur at the boundaries of the input domain rather than at the center.

→ BVA selects test cases at the edges of class

Guidelines for BVA

1. If the specified range is between a and b, then test cases should be designed with values a & b as well as values just above and just below a & b.

2. Also exercise the min & max criteria, by using values outside the range.

C. Graph-based Testing

→ create a graph of important objects and their relationships

→ Each node is an object, which are connected by either by directed / bidirectional / parallel link

directed ⇒ one-way relation

bidirection ⇒ relation goes both ways

parallel ⇒ a number of different relationships between objects

→ Derive test cases by traversing the graph and by covering each of the relationships. These test cases are meant to find errors in any of the relationships.

10. Orthogonal Array Testing?

→ used where the input domain is relatively small but too large to accommodate exhaustive testing?.

→ used to find region fault - a type of error associated w/ software components

→ For eg. if there are 4 variables P_1, P_2, P_3, P_4 - each of which can take 3 values : then there would be $3^4 = 81$ test cases in all. However, with OAT, it can be reduced to 9 cases, where each case of parameter values appear together at least once.

* Software Testing?

→ Software testing integrates software test cases into a well-planned series of steps for the successful development of the software.

→ It provides a road map of the steps to be taken, when, how much effort, time, planning & resources.

→ The strategy incorporates (i) test planning
(ii) test case design
(iii) test execution
(iv) test result collection & evaluation

* Characteristics of Software Testing?

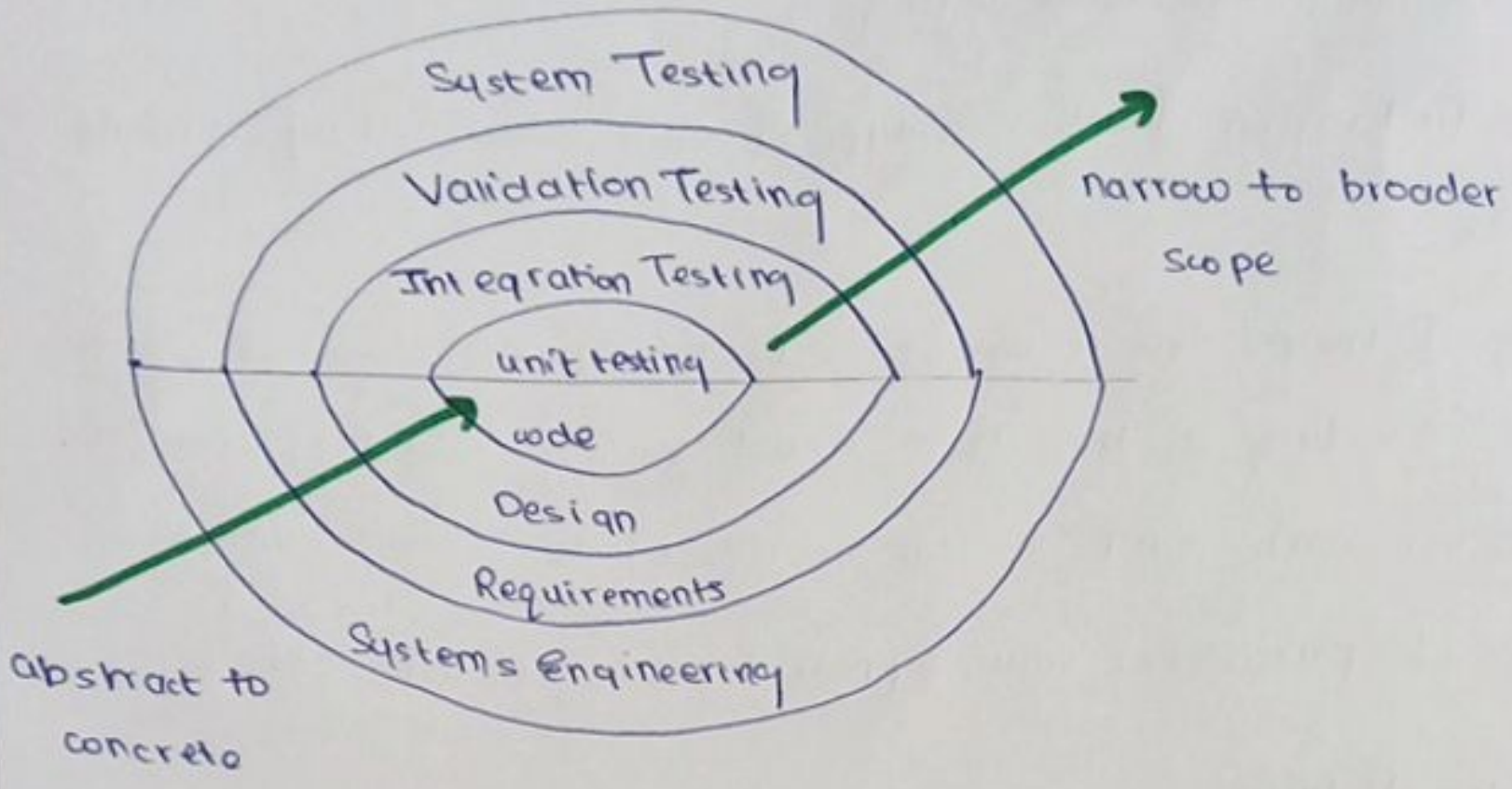
→ conduct effective formal technical reviews

→ testing is conducted by the developer of the software

→ testing is a part of verification and validation

→ Testing should aim at breaking the software.

* Testing Conventional Software



A. Unit Testing

→ testing the functions of the module

→ concentrate on internal processing logic & data structures

→ It is easy when a module has high cohesion (reduces test cases)

→ If resources are limited, concentrate on critical modules & those with high cyclomatic complexity.

Unit Testing Targets : (i) Module interface

(ii) local data structures

(iii) Boundary ~~structures~~ conditions

(iv) Basis paths

(v) Error handling paths

X. Drivers and Stubs in Unit Testing

X.

Driver - a simple main program that accepts test case data & passes the data to the component being tested, and returns the results

Stubs - serves to replace modules that are subordinate to & called by the components to be tested.

— does minimal data manipulation, provides verification of entry and returns control back to the module.

Note: Both drivers & stubs represent overhead - both must be written but are not part of the installed software product.

B. Integration Testing

→ a systematic technique for constructing the software architecture

→ uncover errors associated w/ interfaces

→ There are 2 approaches:

- Non-incremental integration testing
- Incremental integration testing

(i) Non-Incremental Integration Testing

→ called the 'BigBang' Approach

→ all components are combined in advance

→ The entire program is tested as a whole

→ chaos results

(ii) Incremental Integration Testing

→ of 3 kinds
- Top-down integration
- Bottom-up integration
- Sandwich Integration

→ program is written and tested in small increments

→ errors are easier to isolate and correct, interfaces can be fully tested

I - Top-Down Integration

→ modules are integrated by moving downward through the control hierarchy, beginning with the main module.

→ Subordinate modules are added & verified using BFS or DFS

Adv - can verify major control or decision points early

DisAdv - stubs need to be made for modules not built yet.

II - Bottom-Up Integration

→ Integration and testing starts with the most atomic modules in the control hierarchy.

Adv - verifies low level data processing right at the start
- need for stubs is eliminated

Disadv - driver modules need to be built

- more testing may be needed when the upper level modules are available

III - Sandwich Integration

→ consists of a combination of top down & bottom up integration

→ proceeds using functional groups of modules - with each group completed before the next.

→ reaps benefits of both kinds of integration

→ Requires a disciplined approach so that integration doesn't tend towards the big bang scenario.

c. Regression Testing

→ Each addition or change to software may cause problems with functions that previously worked flawlessly.

→ Regression testing re-executes a small subset of tests that have already been conducted.

→ It helps ensure that changes have not propagated unintended side effects or additional errors

→ Regression tests are of 3 types:

(i) a representative sample of tests that will exercise all software functions

(ii) additional tests checking out the changes affecting other functions

(iii) testing the functions that have been changed.

D. **Smoke Testing**

→ Taken from the world of hardware - power is applied and a technician checks for smoke / sparks or other dramatic signs of failure

→ designed for time-critical projects

→ The following activities are carried out in smoke testing:

(i) compile software and link to a build

(ii) expose errors - especially show-stopping ones

(iii) integrate builds with other builds and smoke test daily

Benefits of Smoke Testing

(i) Integration risk is minimized.

(ii) The quality of the end-product is improved.

(iii) Error diagnosis & correction are simplified

(iv) Progress is easier to assess.

E. **Validation Testing**

→ Validation testing follows integration testing.

→ It is designed to ensure that:

(i) all functional requirements are satisfied
(ii) all behavioral characteristics are achieved
(iii) all performance requirements are attained

(iv) documentation is correct

→ Any deviations from specifications are identified and adeficiency list is created.

Alpha and Beta Testing?

* Alpha Testing?

→ conducted at the developer's site by end users

→ developers watch usage

→ Testing is a controlled environment

* Bela Testing

→ conducted at end-user sites

→ developer is not present

→ a live application of the software in an environment that cannot be controlled by the developer.

→ The end user records all problems that are encountered and report these to developers at regular intervals

→ After β-testing, software modifications are made & the software is prepared for release to the entire customer base.

* System Testing

System Testing is done in the following forms:

A. Recovery Testing

→ tests for recovery from system faults

→ forces the software to fail in various ways & verifies that recovery is properly performed.

→ Tests reinitialization, checkpointing mechanisms, data recovery & restart.

B. Security Testing

→ verifies that protection mechanisms built into a system will in fact protect it from improper access.

C. Stress Testing

→ executes a system in a manner that demands resources in an abnormal quantity, frequency or volume.

D. Performance Testing

→ tests run-time performance

→ coupled with stress-testing and requires both hardware & software instrumentation.

→ can uncover situations that lead to degradation & possible system failure.

* Debugging

→ Debugging is a consequence of successful testing

→ The debugging process begins with the execution of a test case.

→ Debugging helps asses the difference between expected and actual behavior.

→ It helps match symptom with cause, helping lead to error correction.

→ Debugging is often very difficult because :

(i) The symptom and cause may be geographically remote.

(ii) The symptom may temporarily disappear when another error is corrected.

(iii) The symptom may be actually caused by non-errors

(iv) The symptom may be caused by human errors.

(v) due to timing problems, may be intermittent

(vi) may be difficult to accurately reproduce

* Debugging Stategies

→ Debugging to find bugs is a combination of systematic evaluation, intuition and luck.

→ There are 3 main strategies :

(i) Brute Force
(ii) Backtracking
(iii) Cause elimination

A. Backtracking

B. Brute Force

→ most commonly used and least effective method

→ used when all else fails

→ involves the use of memory dumps, run-time traces and output statements

→ a waste of time.

B. Backtracking

→ can be used successfully in small programs

→ start at the location where a symptom has been uncovered

→ Trace backward manually until the location of the cause is found

→ In large programs, the no. of potential backward paths may become unmanageably large.

C. Cause Elimination

→ Involves the use of induction or deduction and introduces the idea of binary partitioning.

(i) induction - prove a specific starting value is true, and try to generalize it

(ii) deduction - show that a specific conclusion follows from a set of general premises

→ Data related to the error is ~~isolated~~ organized to isolate potential causes.

→ A cause hypothesis is devised, and the data is used to prove or disprove the hypothesis.

* Testing Tools

→ Software testing tools help in: (i) higher test coverage
(ii) save time and resources
(iii) provide support for multiple platforms
(iv) bug free releases
(v) easy finding & fixing defects
(vi) faster releases

* How to select a testing tool

→ Understand project requirements

→ Consider existing tool as benchmark - understand pros and cons

→ Consider criteria like its ease of use, OS compatibility, budget, support for languages.

→ Make a matrix comparing the shortlisted tools.

* Types of Software Testing Tools

A. Static Testing Tools - test software without executing it
- involves analyzing code for syntax, checking documentation

B. Dynamic Testing Tools - Tools interact with software while executing
- provides info about programs & diff. event

C. Open Source Tools - free to use tools with code on the internet.

D. Vendor Tools - developed by companies that come with licenses to use, cost money

E. In House Tools - built by companies for their own use - rather than purchasing other tools.

Based on Functionality

1. Agile Testing tools - tracks defects, makes reports
   - eg. Jira by Atlassian

2.
   - Soap UI
   - Selenium WebDriver - supports multiple programming langs

2. Automation Testing Tools - (i) HP UFT - test mobile platforms on web browsing

   (ii) Selenium - automate regression tasks for web browsers

   (iii) Watir - automates web browsers

3. Mobile Testing Tools - eg. eggPlant
   Appium

4. Load Testing Tools - eg. Tsung
   wapt

5. Test Management Tools - eg. Zephyr
   Qmetry

Coding Practices and Principles

① DRY - Don't Repeat Yourself

→ don't replicate

→ use abstraction to summarize things in a single area.

② KISS - Keep It Short and Simple

→ be mindful about the code you are writing - keep it as precise as possible

→ if something can be written in a single line, write it in a single line

③ Refactor

→ Examine code again and again and looks for way to optimize it.

→ Make more effective while keeping results the same

④ Document Your Code

→ write code w/ comments

→ make specific features more easier to understand

⑤ Creation Over Legacy

→ When dealing with complex behaviors, create new objects or components rather than using legacy elements

⑥ Clean Code at All Costs

→ Keep it precise yet explicit

→ Don't package a lot of logic on one line - make it easily readable.

7. You Are Not Going to Need It

→ never code just for the sake of performance, imagining that it will help someday

8. Open / Closed

→ make code open for expansion, yet closed for adjustment

9. Single Responsibility

→ each module / class should have only a specific functionality

→ keep modules clean and minimal

10. Separation of Concerns

→ MVC plan - Model View Controller

→ isolate a program into 3 unique regions: information, rationales & showcase

11. Encapsulate the changes

→ confine or summarize code

→ easier to test encapsulated code

12. Delegation Principles

→ Assign work to ~~principles~~ each person

13. Interface Segregation Principle (ISP)

→ ~~client~~ A customer shouldn't use an interface if it doesn't require it,

→ when an interface has a lot of features, but the customer uses just one.

14. **Liskov Substitution Principle (LSP)**

→ subtypes should be a replacement for a supertype

15. **Programming for Interface Rather than Implementation**

→ program for the interface so that is compatible with any other future similar interface.

16. **Favor Composition instead of Inheritance**

→ change behavior of a class during run-time

→ makes it flexible to change it w/ implementation

* Software Maintenance

→ Software maintenance is the act of keeping / the expenditure required to keep an asset in condition, so that it performs efficiently.

→ can also be modification after delivery to correct fault, improve performance or adapt to environments.

* Types of Software Maintenance

A. **Corrective Maintenance**

→ Reactive modification of a software product performed after delivery to correct discovered problems

system: - response to equipment malfunctions

characteristics: - inefficient maintenance dept
- unpredictable equipment operation

Example - light bulb replacement

results - steady degradation of equipment

maintenance dept. responsibility - respond to emergencies
- get production back on line

B. **Adaptive Maintenance**

→ modification after delivery to keep a product usable

system - equipment design w/ minimal maintenance requirements

characteristics - close relationship with suppliers

examples - roadways, websites

results - continually improving equipment

responsibility - minimize & eliminate maintenance requirements

c. **Preventive Maintenance**

→ correct latent faults before they become effective faults

system - periodic checks & adjustments

characteristics - more predictable, efficient

example - changing oils & filters

result - maintain level of equipment

maintenance dept responsibility - checking, replacing & overhauling
perform checks

D. Perfective Maintenance

→ modification of a software product after delivery to improve performance or maintainability

system - periodic measurement

characteristics - planned & scheduled repairs

example - software updates

results - maintain equipment performance with min. disruption to production

maintenance dept. responsibilities - log repairs
predict repair cycles

* Software Maintenance Process

There are 6 software maintenance processes as follows:.

① Implementation Process

→ includes preparation activities like the creation of the maintenance plan, preparation for handling problems found during development

→ also has follow up on product config. management

② Problem & Modification Analysis

→ The maintenance programmer must analyze each request, confirm it by reproducing the situation and check its validity

investigate it and propose a solution

→ The solution must be documented, and all the required authorizations must be obtained to apply the modifications

③ **Implementation of Modification**

④ **Acceptance of Modification**

→ Confirm the modified work with the individual who submitted the request in order to make sure that the modification provided a solution

⑤ **Migration Process**

→ an exceptional case, not a part of daily maintence task

→ when the software must be ported to another platform without any change in functionality, a maintenance project team is assigned to the task

⑥ **Retirement of Software**

→ an event that does not occur on a daily basis - a piece of software is completely retired

* **Need for Software Maintenance**

→ repair software faults

→ adapt software to a diff. environment

→ add or modify the system's functionality

* Optimal Maintenance

→ a discipline concerned with maintaining a system that maximizes profit or minimizes cost

→ parameters considered are:

(i) cost of failure

(ii) cost per unit time of downtime

(iii) cost per unit time of preventive and corrective maintenance

(iv) cost of a repairable replacement

* Maintenance Costs

→ can range from 2 x to 100 x the OG cost.

→ can asise from technical & non-technical factors

→ A deployed system is difficult & expensive to change

→ Maintenance costs increase over ~~tear~~ time as the system evolves

* Maintenance Cost factors

→ Team stability

→ Contractual responsibility

→ Staff skills

→ Program age and structure

* Strategies to reduce maintenance costs

→ review basic operations

→ conduct physical analysis

→ correct even slight defects

→ ensure basic equipment conditions are maintained

* Difficulties in Software maintenance

→ someone else's program

→ developer not available

→ proper documentation doesn't exist

→ not designed for change

→ maintenance activity not highly regarded