Module 2

Testing Techniques

Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University
Pooja Malhotra

Static Testing

Static Testing

Static testing is a complimentary technique to dynamic testing technique to acquire higher quality software. Static testing techniques do not execute the software.

Static testing can be applied for most of the verification activities.

The objectives of static testing can be summarized as follows:

- To identify errors in any phase of SDLC as early as possible
- To verify that the components of software are in conformance with its requirements
- To provide information for project monitoring
- To improve the software quality and increase productivity

Static Testing

- Static testing techniques do not demonstrate that the software is operational or one function of software is working;
- They check the software product at each SDLC stage for conformance with the required specifications or standards. Requirements, design specifications, test plans, source code, user's manuals, maintenance procedures are some of the items that can be statically tested.
- Static testing has proved to be a cost-effective technique of error detection.
- Another advantage in static testing is that a bug is found at its exact location whereas a bug found in dynamic testing provides no indication to the exact source code location.

Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University
Pooja Malhotra

Static Testing

Types of Static Testing

- Software Inspections
- Walkthroughs
- Technical Reviews

Inspections

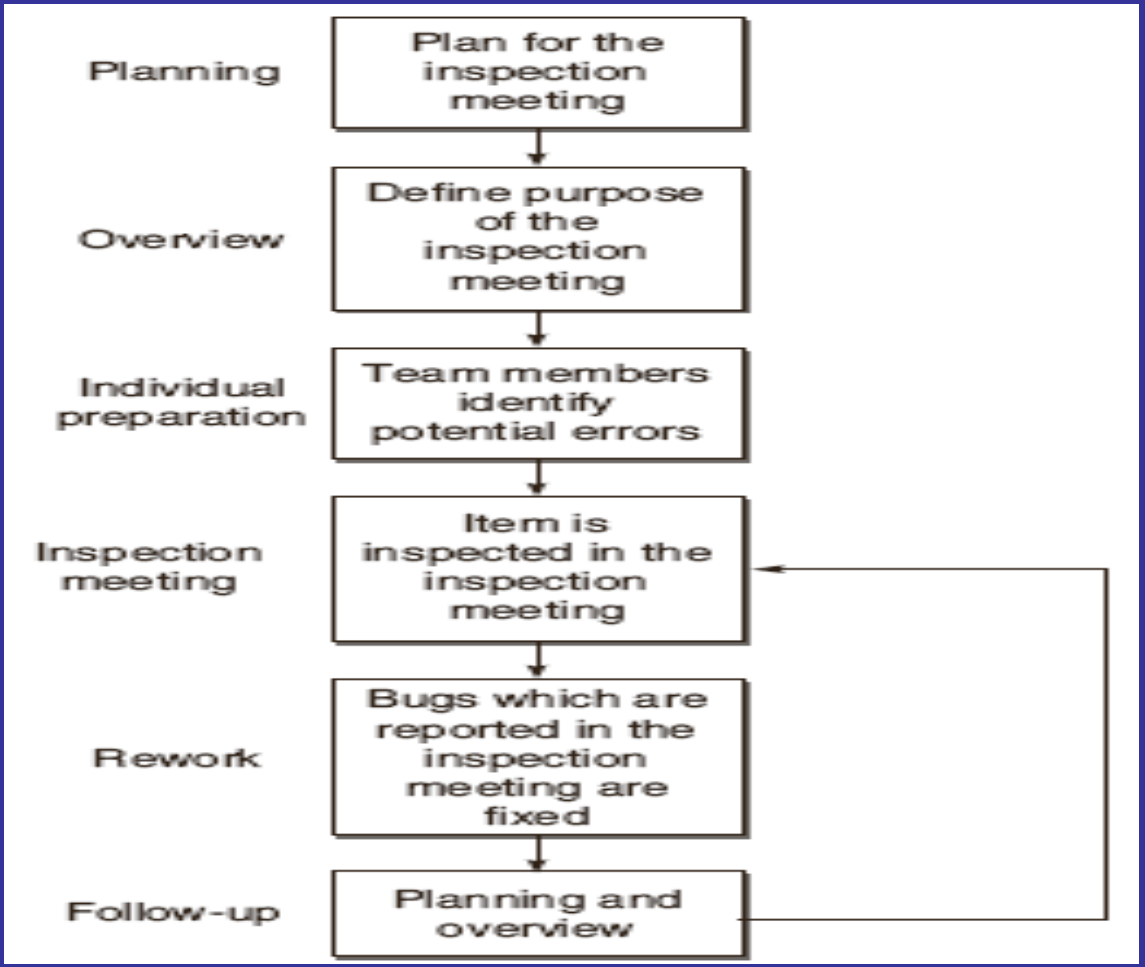
- Inspection process is an in-process manual examination of an item to detect bugs.
- Inspection process is carried out by a group of peers. The group of peers first inspects the product at individual level. After this, they discuss potential defects of the product observed in a formal meeting.
- It is a very formal process to verify a software product. The documents which can be inspected are SRS, SDD, code and test plan.

Inspections

Inspection process involves the interaction of the following elements:

- Inspection steps
- Roles for participants
- Item being inspected

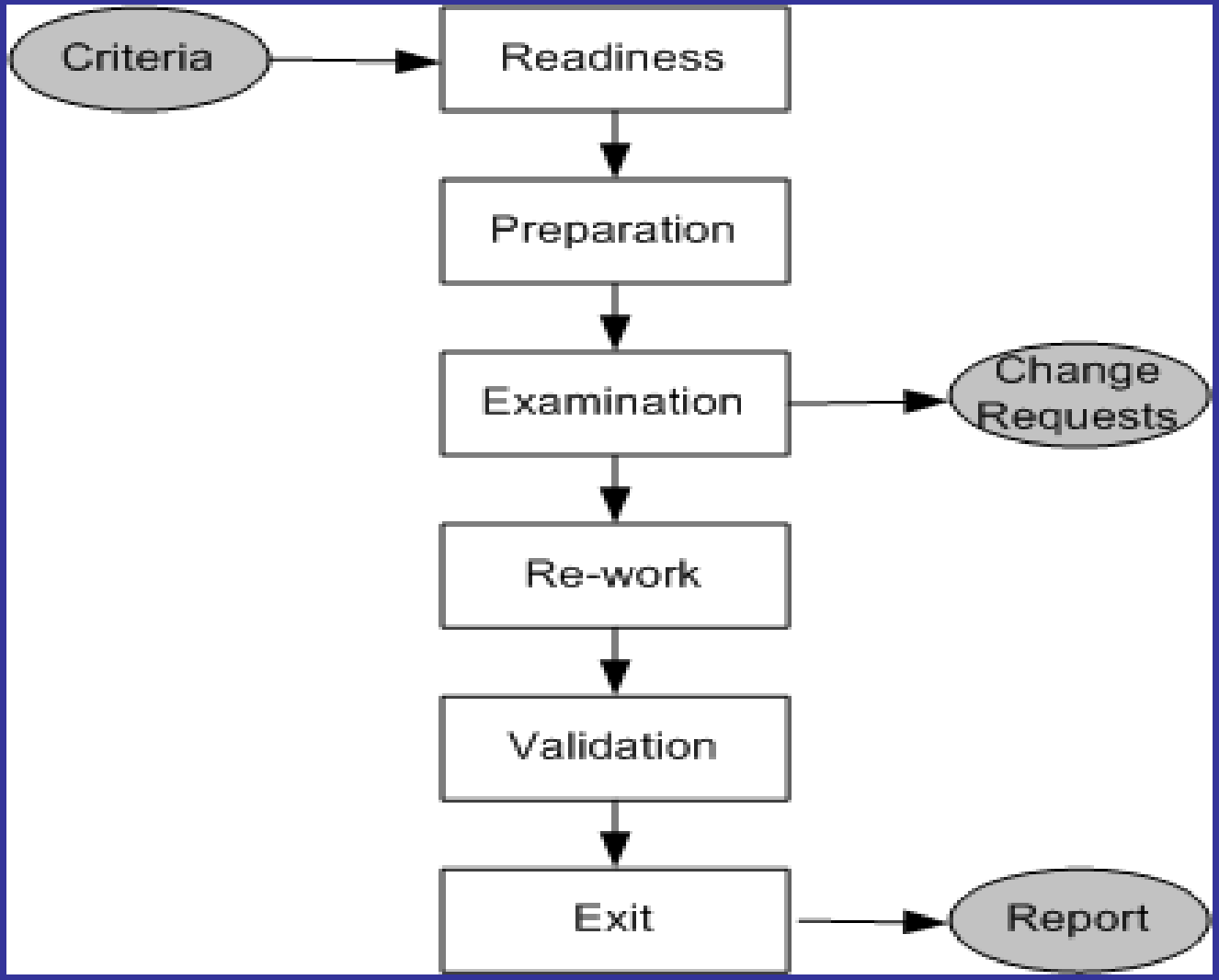
Inspection Process



Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University

Pooja Malhotra

Steps in the Inspection



Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University

Pooja Malhotra

Steps in the Inspection

1.Planning : During this phase, the following is executed:

- The product to be inspected is identified.
- A moderator is assigned.
- The objective of the inspection is stated. If the objective is defect detection, then the type of defect detection like design error, interface error, code error must be specified.

During planning, the moderator performs the following activities:

- Assures that the product is ready for inspection
- Selects the inspection team and assigns their roles
- Schedules the meeting venue and time
- Distributes the inspection material like the item to be inspected, checklists, etc.

Readiness Criteria

- Completeness ,Minimal functionality
- Readability, Complexity, Requirements and design documents

Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University
Pooja Malhotra

Steps in the Inspection

Inspection Team:

- Moderator
- Author
- Presenter
- Record keeper
- Reviewers
- Observer

2.Overview: In this stage, the inspection team is provided with the background information for inspection. The author presents the rationale for the product, its relationship to the rest of the products being developed, its function and intended use, and the approach used to develop it. This information is necessary for the inspection team to perform a successful inspection.

The opening meeting may also be called by the moderator. In this meeting, the objective of inspection is explained to the team members. The idea is that every member should be familiar with the overall purpose of the inspection.

Steps in the Inspection

3.Individual Preparation: After the overview, the reviewers individually prepare themselves for the inspection process by studying the documents provided to them in the overview session.

- List of questions
- Potential Change Request (CR)
- Suggested improvement opportunities

Completed preparation logs are submitted to the moderator prior to the inspection meeting.

Inspection Meeting/Examination:

- The author makes a presentation
- The presenter reads the code
- The record keeper documents the CR
- Moderator ensures the review is on track

Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University
Pooja Malhotra

Steps in the Inspection

At the end, the moderator concludes the meeting and produces a summary of the inspection meeting.

Change Request (CR) includes the following details:

- Give a brief description of the issue
- Assign a priority level (major or minor) to a **CR**
- Assign a person to follow it up
- Set a deadline for addressing a **CR**

Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University
Pooja Malhotra

Steps in the Inspection

4.Re-work: The summary list of the bugs that arise during the inspection meeting needs to be reworked by the author.

- Make the list of all the CRs
- Make a list of improvements
- Record the minutes meeting
- Author works on the CRs to fix the issue

5.Validation and Follow-up: It is the responsibility of the moderator to check that all the bugs found in the last meeting have been addressed and fixed.

- Moderator prepares a report and ascertains that all issues have been resolved. The document is then approved for release.
- If this is not the case, then the unresolved issues are mentioned in a report and another inspection meeting is called by the moderator.

Benefits of Inspection Process

- Bug Reduction
- Bug Prevention
- Productivity
- Real-time Feedback to Software Engineers
- Reduction in Development Resource
- Quality Improvement
- Project Management
- Checking Coupling and Cohesion
- Learning through Inspection
- Process Improvement

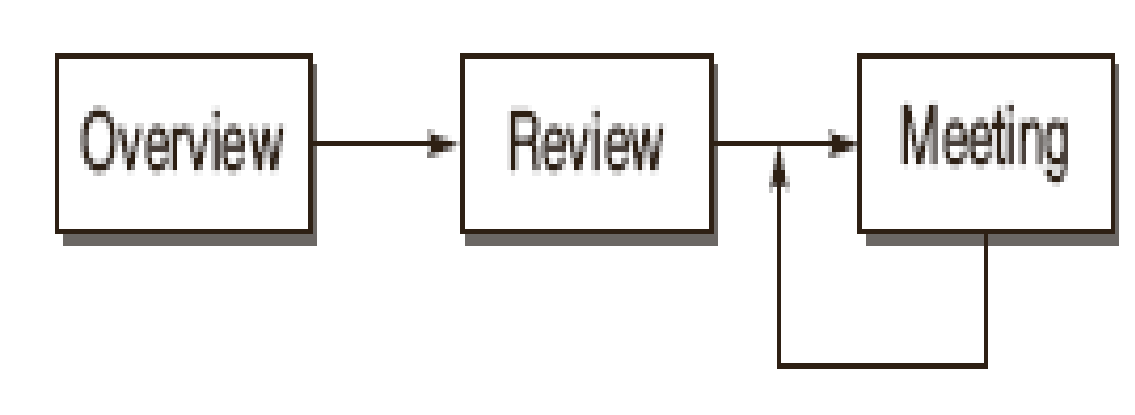
Variants of Inspection process

Active Design Reviews (ADRs)	Several reviews are conducted targeting a particular type of bugs and conducted by the reviewers who are experts in that area.
Formal Technical Asynchronous review method (FTArm)	Inspection process is carried out without really having a meeting of the members. This is a type of asynchronous inspection in which the inspectors never have to simultaneously meet.
Gilb Inspection	Defect detection is carried out by individual inspector at his level rather than in a group.
Humphrey's Inspection Process	Preparation phase emphasizes the finding and logging of bugs, unlike Fagan inspections. It also includes an analysis phase wherein individual logs are analysed and combined into a single list.
N-Fold inspections	Inspection process's effectiveness can be increased by replicating it by having multiple inspection teams.
Phased Inspection	Phased inspections are designed to verify the product in a particular domain by experts in that domain only.
Structured Walkthrough	Described by Yourdon. Less formal and rigorous than formal inspections. Roles are coordinator, scribe, presenter, reviewers, maintenance oracle, standards bearer, user representative. Process steps are Organization, Preparation, Walkthrough, and Rework. Lacks data collection requirements of formal inspections.

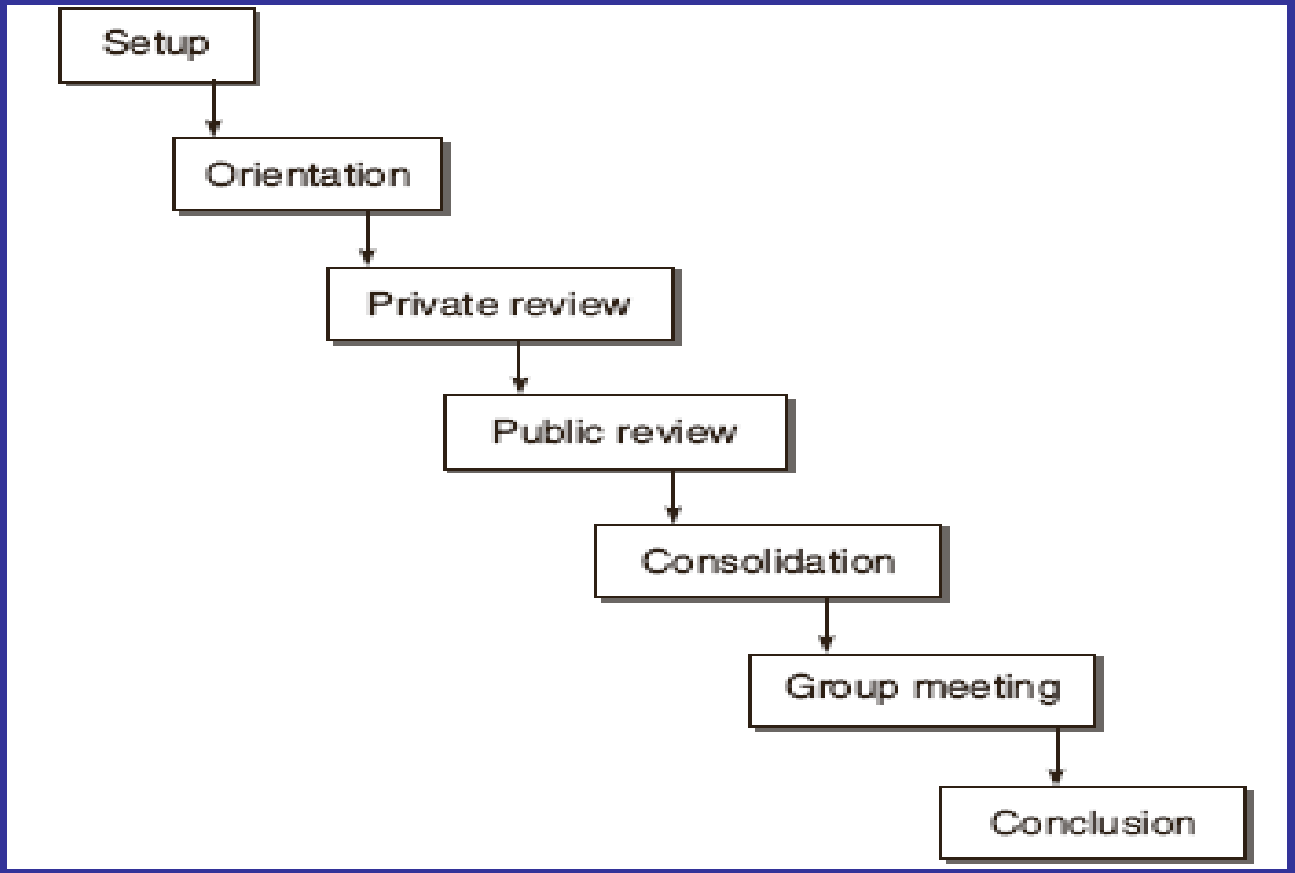
Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University

Pooja Malhotra

Active Design Reviews



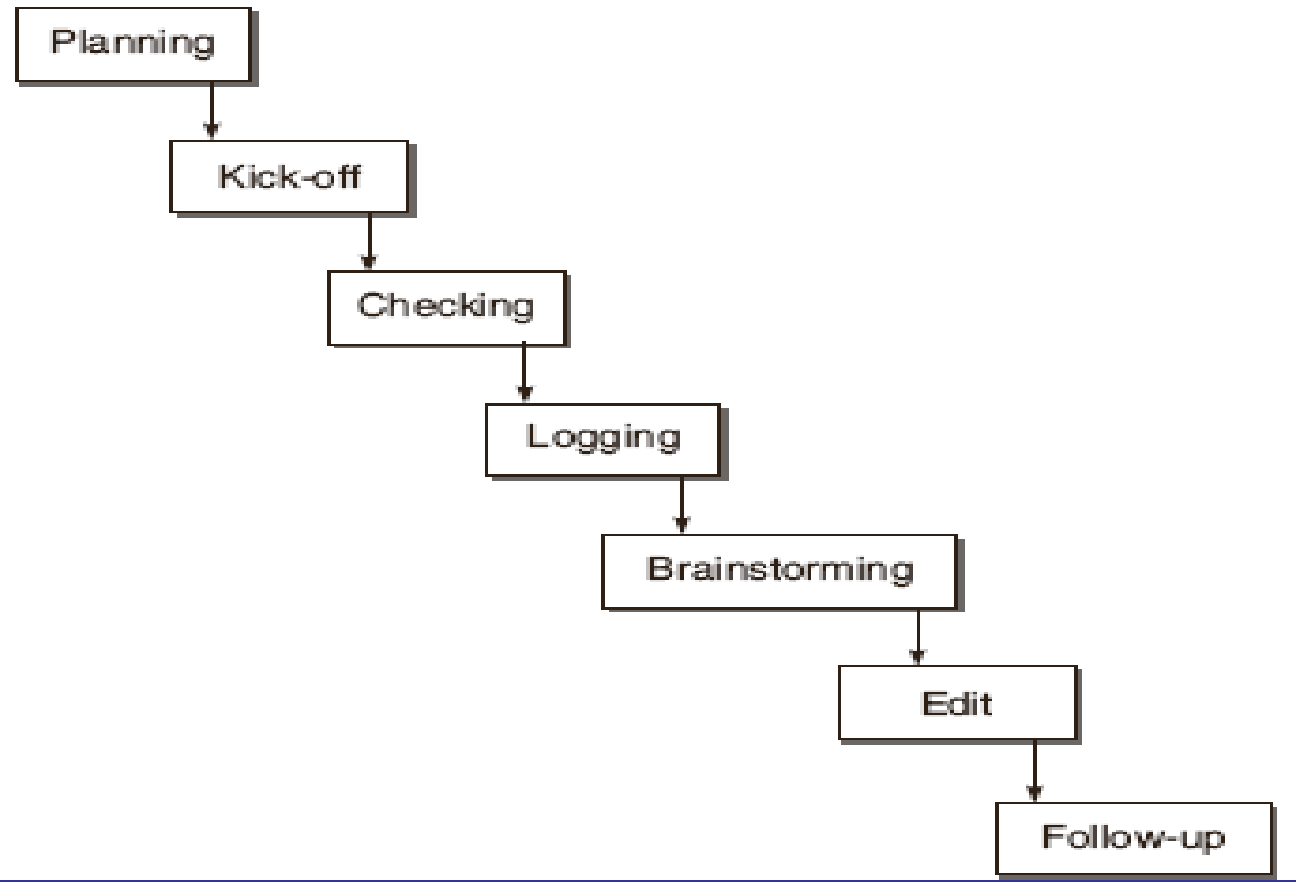
Formal Technical Asynchronous review method (FTArm)



Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University

Pooja Malhotra

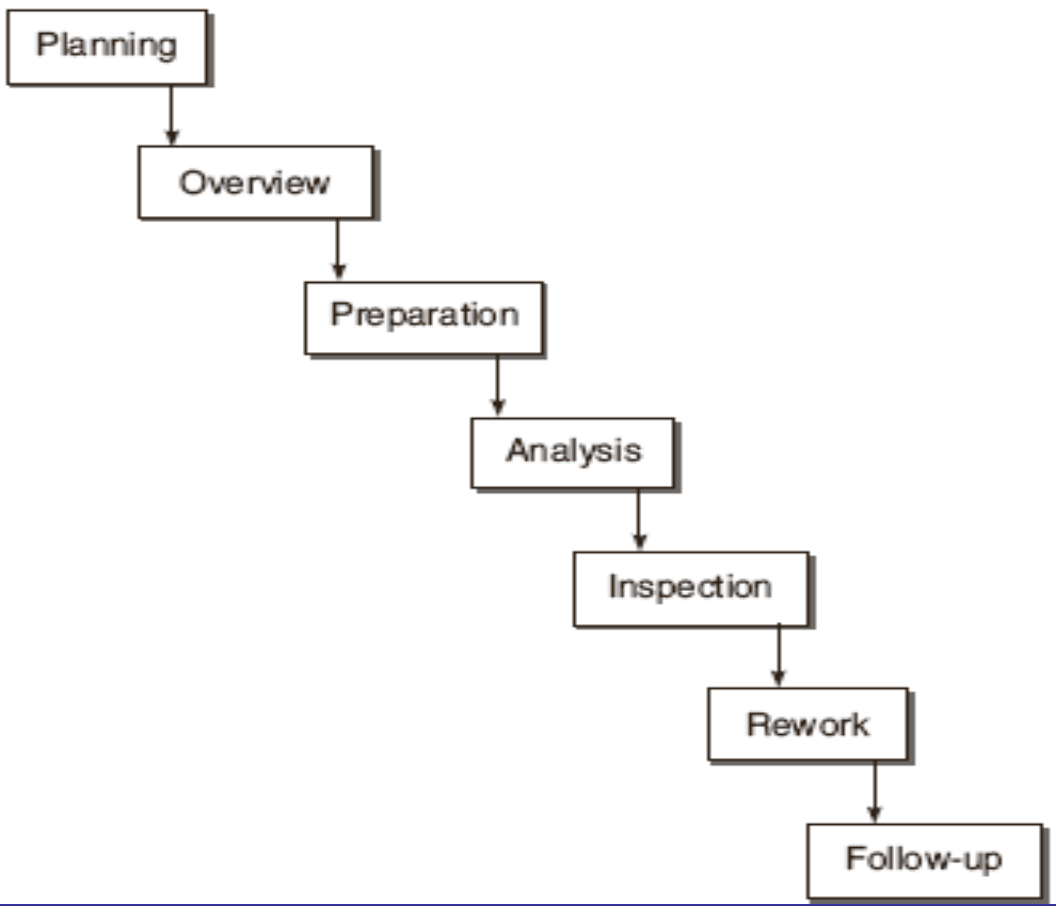
Gilb Inspection



Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University

Pooja Malhotra

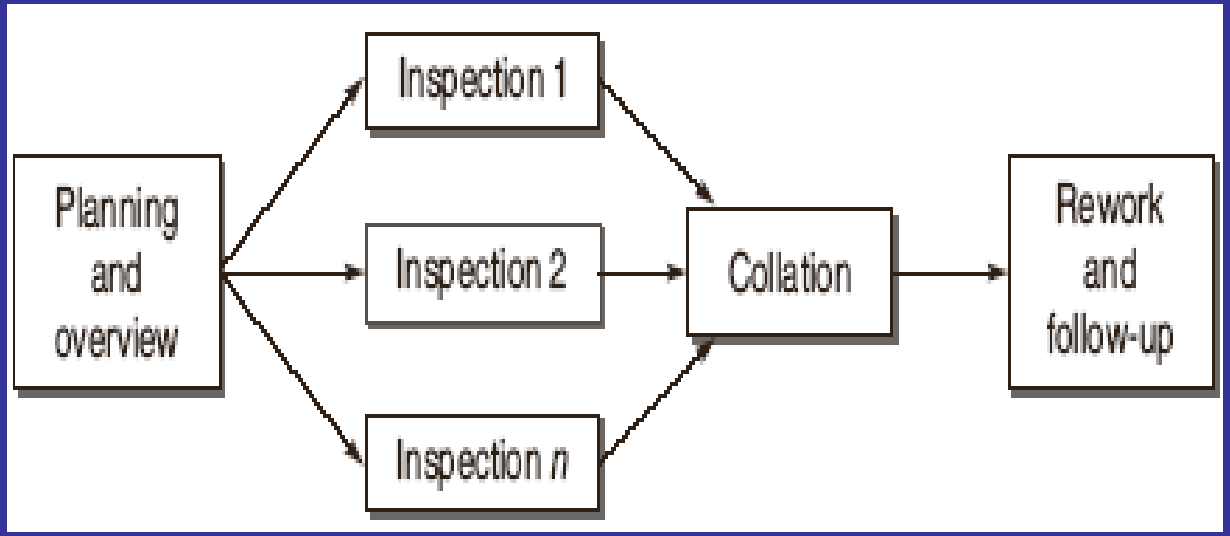
Humphrey's Inspection Process



Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University

Pooja Malhotra

N-Fold Inspection



Checklist

Structure:

- ❑ Does the code completely and correctly implement the design?
- ❑ Does the code conform to any applicable coding standards?
- ❑ Is the code well-structured, consistent in style, and consistently formatted?
- ❑ Are there any uncalled or unneeded procedures or any unreachable code?
- ❑ Are there any leftover stubs or test routines in the code?
- ❑ Can any code be replaced by calls to external reusable components or library functions?
- ❑ Are there any blocks of repeated code that could be condensed into a single procedure?
- ❑ Is storage use efficient?
- ❑ Are any modules excessively complex and should be restructured or split into multiple routines?

Checklist

Arithmetic Operations:

- ❑ Does the code avoid comparing floating-point numbers for equality?
- ❑ Does the code systematically prevent rounding errors?
- ❑ Are divisors tested for zero or noise?

Checklist

Loops and Branches:

- ❑ Are all loops, branches, and logic constructs complete, correct, and properly nested?
- ❑ Are all cases covered in an IF- -ELSEIF or CASE block, including ELSE or DEFAULT clauses?
- ❑ Does every case statement have a default?
- ❑ Are loop termination conditions obvious and always achievable?
- ❑ Are indexes or subscripts properly initialized, just prior to the loop?
- ❑ Does the code in the loop avoid manipulating the index variable or using it upon exit from the loop?

Checklist

Documentation:

- ❑ Is the code clearly and adequately documented with an easy-to-maintain commenting style?
- ❑ Are all comments consistent with the code?

Variables:

- ❑ Are all variables properly defined with meaningful, consistent, and clear names?
- ❑ Do all assigned variables have proper type consistency or casting?
- ❑ Are there any redundant or unused variables?

Checklist

Input / Output errors:

- If the file or peripheral is not ready, is that error condition handled?
- Does the software handle the situation of the external device being disconnected?
- Have all error messages been checked for correctness, appropriateness, grammar, and spelling?
- Are all exceptions handled by some part of the code?

Scenario based Reading

Perspective based Reading

- software item should be inspected from the perspective of different stakeholders. Inspectors of an inspection team have to check software quality as well as the software quality factors of a software artifact from different perspectives.

Usage based Reading

- This method given is applied in design inspections. Design documentation is inspected based on use cases, which are documented in requirements specification.

Abstraction driven Reading

- This method is designed for code inspections. In this method, an inspector reads a sequence of statements in the code and abstracts the functions these statements compute.

Scenario based Reading

Task driven Reading

- This method is also for code inspections . In this method, the inspector has to create a data dictionary, a complete description of the logic and a cross-reference between the code and the specifications.

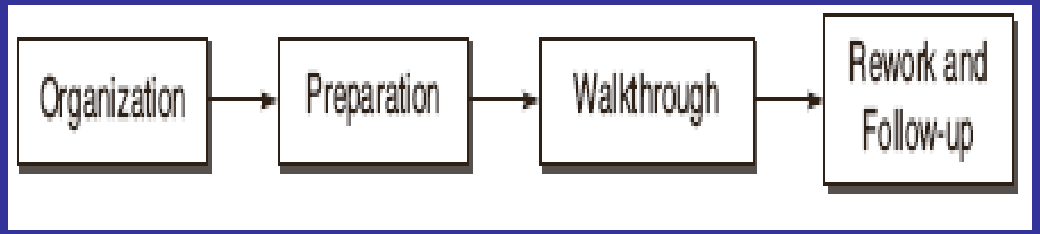
Function-point based Scenarios

- This is based on scenarios for defect detection in requirements documents [103]. The scenarios, designed around function-points are known as the Function Point Scenarios. A Function Point Scenario consists of questions and directs the focus of an inspector to a specific function-point item within the inspected requirements document.

Structured Walkthroughs

It is a less formal and less rigorous technique as compared to inspection.

- Author presents their developed artefact to an audience of peers.
- Peers question and comment on the artefact to identify as many defects as possible.
- It involves no prior preparation by the audience. Usually involves minimal documentation of either the process or any arising issues. Defect tracking in walkthroughs is inconsistent.
- A walk through is an evaluation process which is an informal meeting, which does not require preparation.
- The product is described by the author and queries for the comments of participants.
- The results are the information to the participants about the product instead of correcting it.



Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University
Pooja Malhotra

Technical Reviews

A technical review is intended to evaluate the software in the light of development standards, guidelines, and specifications and to provide the management with evidence that the development process is being carried out according to the stated objectives. A review is similar to an inspection or walkthrough, except that the review team also includes management. Therefore, it is considered a higher-level technique than inspection or walkthrough.

A technical review team is generally comprised of management-level representatives of the User and Project Management. Review agendas should focus less on technical issues and more on oversight than an inspection. The purpose is to evaluate the system relative to specifications and standards, recording defects and deficiencies. The moderator should gather and distribute the documentation to all team members for examination before the review. He should also prepare a set of indicators to measure the following points:

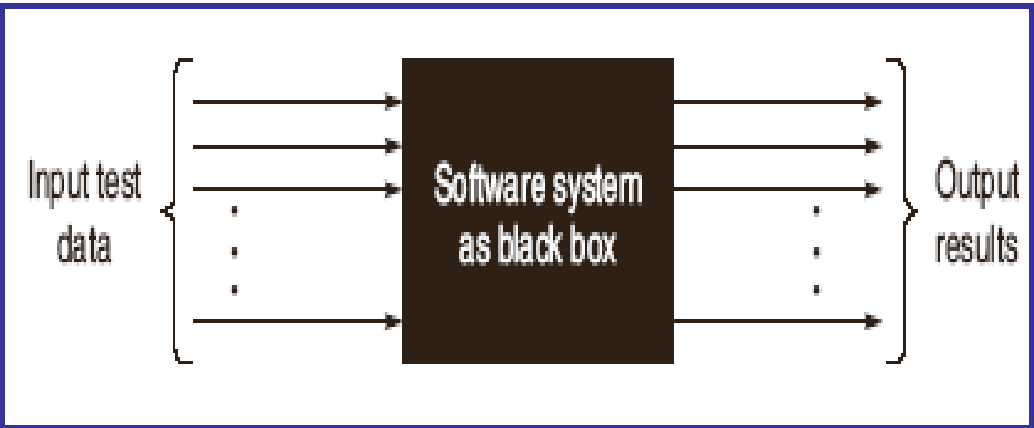
- Appropriateness of the problem definition and requirements
- Adequacy of all underlying assumptions
- Adherence to standards, Consistency , Completeness, Documentation

Dynamic Testing:

Black Box Testing

Black-box technique is one of the major techniques in dynamic testing for designing effective test cases. This technique considers only the functional requirements of the software or module. In other words, the structure or logic of the software is not considered. Therefore, this is also known as functional testing.

- The software system is considered as a black box, taking no notice of its internal structure, so it is also called as black-box testing technique.
- It is obvious that in black-box technique, test cases are designed based on functional specifications. Input test data is given to the system, which is a black box to the tester, and results are checked against expected outputs after executing the software,



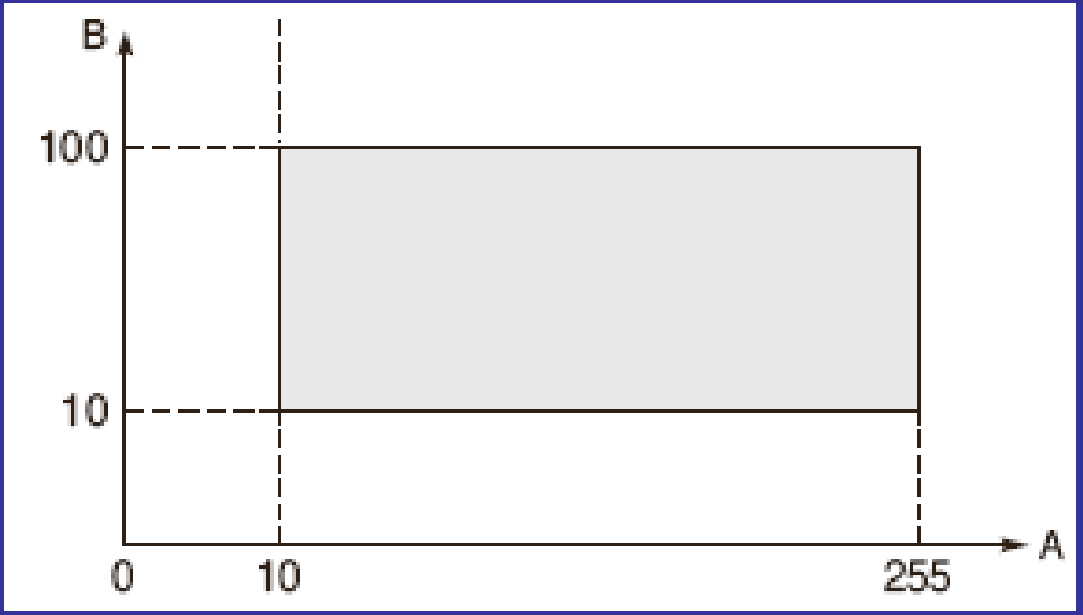
Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University
Pooja Malhotra

Black Box Testing

- To test the modules independently.
- To test the functional validity of the software
- Interface errors are detected.
- To test the system behavior and check its performance.
- To test the maximum load or stress on the system.
- Customer accepts the system within defined acceptable limits.

Boundary Value Analysis (BVA)

‘Boundary value analysis’ testing technique is used to identify errors at boundaries rather than finding those exist in centre of input domain.



Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University

Pooja Malhotra

Boundary Value Analysis (BVA)

- The BVA technique is an extension and refinement of the equivalence class partitioning technique

- In the BVA technique, the boundary conditions for each of the equivalence class are analyzed in order generate test cases

Guidelines for Boundary Value Analysis

- The equivalence class specifies a range
  - If an equivalence class specifies a range of values, then construct test cases by considering the boundary points of the range and points just beyond the boundaries of the range
- The equivalence class specifies a number of values
  - If an equivalence class specifies a number of values, then construct test cases for the minimum and the maximum value of the number
  - In addition, select a value smaller than the minimum and a value larger than the maximum value.
- The equivalence class specifies an ordered set
  - If the equivalence class specifies an ordered set, such as a linear list, table, or a sequential file, then focus attention on the first and last elements of the set.

BVA- “Single fault "assumption theory

“Single fault” assumption in reliability theory: failures are only rarely the result of the simultaneous occurrence of two (or more) faults.

The function f that computes the number of test cases for a given number of variables n can be shown as:

$f = 4n+ 1$

As there are four extreme values this accounts for the 4n.The addition of the constant one constitutes for the instance where all variables assume their nominal value.

BVA- “Single fault "assumption theory

The basic form of implementation is to maintain all but one of the variables at their nominal (normal or average) values and allowing the remaining variable to take on its extreme values. The values used to test the extremities are:

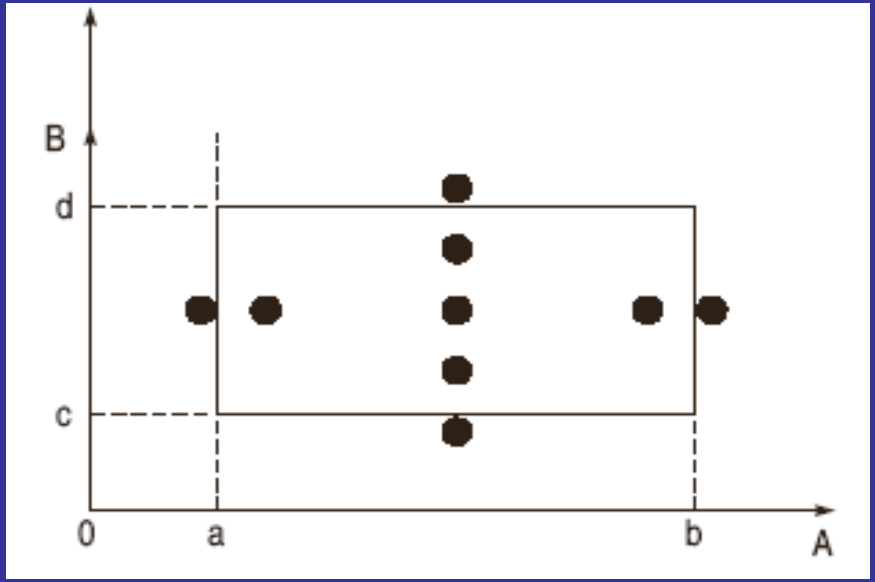
- Min ---------- - Minimal
- Min+ ---------- Just above Minimal
- Nom ---------- Average
- Max- ---------- -------- Just below Maximum
- Max ---------- ---------- Maximum

Boundary Value Checking

- Test cases are designed by holding one variable at its extreme value and other variables at their nominal values in the input domain. The variable at its extreme value can be selected at:
  - Minimum value (Min)
  - Value just above the minimum value (Min+)
  - Maximum value (Max)
  - Value just below the maximum value (Max-)

Boundary Value Checking

- Anom, Bmin
- Anom, Bmin+
- Anom, Bmax
- Anom, Bmax-
- Amin, Bnom
- Amin+, Bnom
- Amax, Bnom
- Amax-, Bnom
- Anom, Bnom



- 4n+1 test cases can be designed with boundary value checking method.

Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University
Pooja Malhotra

Robustness Testing Method

A value just greater than the Maximum value (Max+)

A value just less than Minimum value (Min-)

- When test cases are designed considering above points in addition to BVC, it is called Robustness testing.

- Amax+, Bnom
- Amin-, Bnom
- Anom, Bmax+
- Anom, Bmin-
-
- It can be generalized that for n input variables in a module, 6n+1 test cases are designed with Robustness testing.

Worst Case Testing Method

- When more than one variable are in extreme values, i.e. when more than one variable are on the boundary. It is called Worst case testing method.

- It can be generalized that for n input variables in a module, 5^n test cases are designed with worst case testing.

10. A_{min}, B_{min}	11. A_{min+}, B_{min}
12. A_{min}, B_{min+}	13. A_{min+}, B_{min+}
14. A_{max}, B_{min}	15. A_{max-}, B_{min}
16. A_{max}, B_{min+}	17. A_{max-}, B_{min+}
18. A_{min}, B_{max}	19. A_{min+}, B_{max}
20. A_{min}, B_{max-}	21. A_{min+}, B_{max-}
22. A_{max}, B_{max}	23. A_{max-}, B_{max}
24. A_{max}, B_{max-}	25. A_{max-}, B_{max-}

Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University

Pooja Malhotra

Example

- A program reads an integer number within the range [1,100] and determines whether the number is a prime number or not. Design all test cases for this program using BVC, Robust testing and worst-case testing methods.

- **1) Test cases using BVC**

Test Case ID	Integer Variable	Expected Output
1	1	Not a prime number
2	2	Prime number
3	100	Not a prime number
4	99	Not a prime number
5	53	Prime number

Min value = 1
Min^{+} value = 2
Max value = 100
Max^{-} value = 99
Nominal value = 50–55

Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University
Pooja Malhotra

Example

- **Test Cases Using Robust Testing**

Test Case ID	Integer Variable	Expected Output
1	0	Invalid input
2	1	Not a prime number
3	2	Prime number
4	100	Not a prime number
5	99	Not a prime number
6	101	Invalid input
7	53	Prime number

Min value = 1
Min$^-$ value = 0
Min$^+$ value = 2
Max value = 100
Max$^-$ value = 99
Max$^+$ value = 101
Nominal value = 50–55

Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University
Pooja Malhotra

BVA- The triangle problem

The triangle problem accepts three integers (a, b and c)as its input, each of which are taken to be sides of a triangle . The values of these inputs are used to determine the type of the triangle (Equilateral, Isosceles, Scalene or not a triangle).

For the inputs to be declared as being a triangle they must satisfy the six conditions:

C1. $1 \leq a \leq 200$.

C2. $1 \leq b \leq 200$.

C3. $1 \leq c \leq 200$.

C4. $a < b + c$.

C5. $b < a + c$.

C6. $c < a + b$.

Otherwise this is declared not to be a triangle.

The type of the triangle, provided the conditions are met, is determined as follows:

1. If all three sides are equal, the output is Equilateral.
2. If exactly one pair of sides is equal, the output is Isosceles.
3. If no pair of sides is equal, the output is Scalene.

Test Cases for the Triangle Problem

min = 1
min+ = 2
nom = 100
max- = 199
max = 200

Boundary Value Analysis Test Cases

Case	a	b	c	Expected Output
1	100	100	1	Isosceles
2	100	100	2	Isosceles
3	100	100	100	Equilateral
4	100	100	199	Isosceles
5	100	100	200	Not a Triangle
6	100	1	100	Isosceles
7	100	2	100	Isosceles
8	100	199	100	Isosceles
9	100	200	100	Not a Triangle
10	1	100	100	Isosceles
11	2	100	100	Isosceles
12	199	100	100	Isosceles
13	200	100	100	Not a Triangle

Equivalence Class Testing

- An input domain may be too large for all its elements to be used as test input
- The input domain is partitioned into a finite number of subdomains
- Each subdomain is known as an equivalence class, and it serves as a source of at least one test input
- A valid input to a system is an element of the input domain that is expected to return a non error value
- An invalid input is an input that is expected to return an error value.

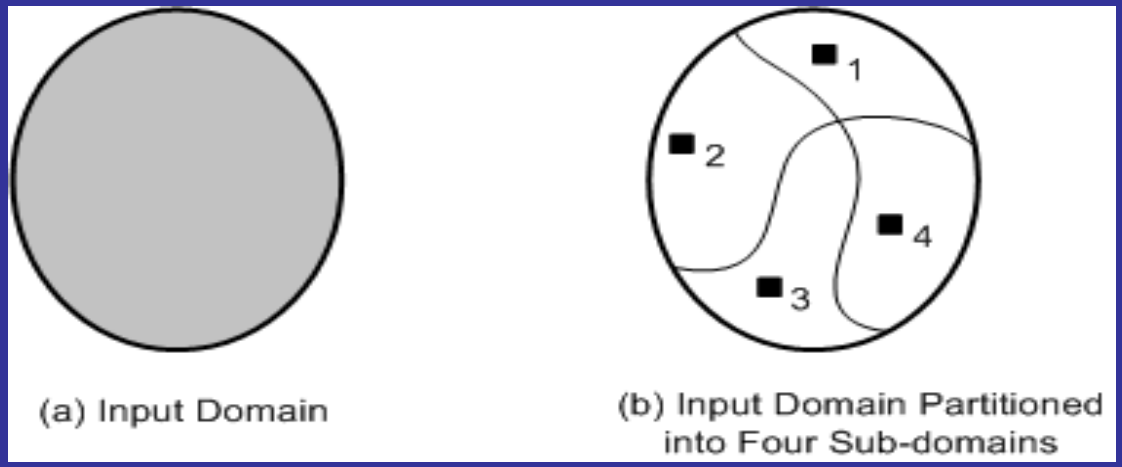
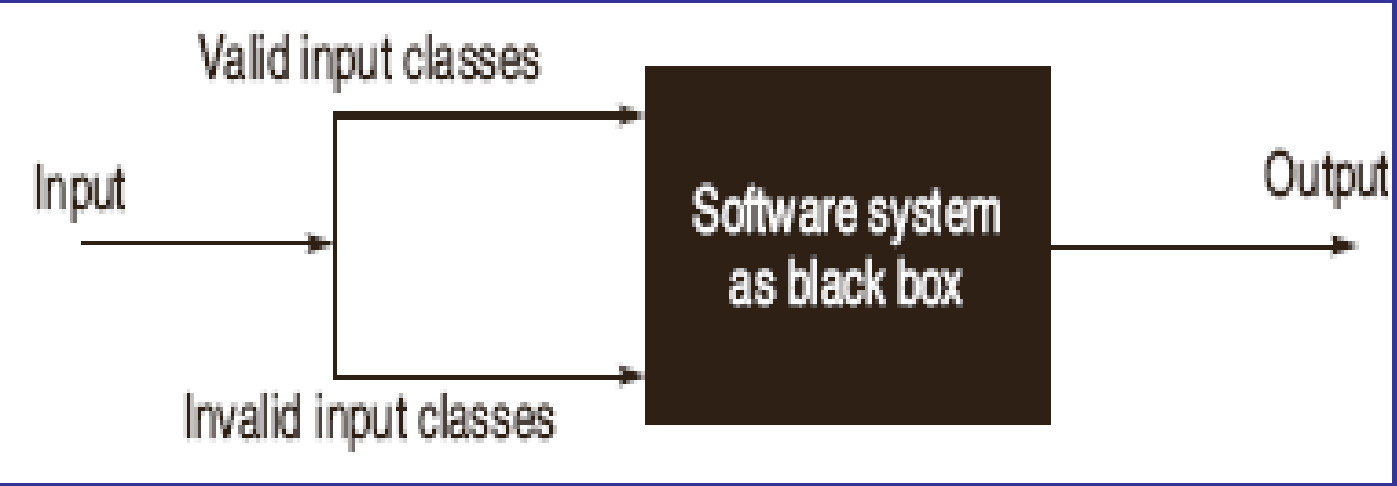


Figure (a) Too many test input; (b) One input is selected from each of the subdomain

Equivalence Class Testing

Equivalence partitioning is a method for deriving test cases wherein classes of input conditions called equivalence classes are identified such that each member of the class causes the same kind of processing and output to occur. Thus, instead of testing every input, only one test case from each partitioned class can be executed.



Guidelines for Equivalence Class Partitioning

- An input condition specifies a range [a, b]
  - one equivalence class for a < X < b, and
  - two other classes for X < a and X > b to test the system with invalid inputs
- An input condition specifies a set of values
  - one equivalence class for each element of the set $\{M_1\}$, $\{M_2\}$, ..., $\{M_N\}$, and
  - one equivalence class for elements outside the set $\{M_1, M_2, ..., M_N\}$
- Input condition specifies for each individual value
  - If the system handles each valid input differently then create one equivalence class for each valid input
- An input condition specifies the number of valid values (Say N)
  - Create one equivalence class for the correct number of inputs
  - two equivalence classes for invalid inputs – one for zero values and one for more than N values
- An input condition specifies a "must be" value
  - Create one equivalence class for a "must be" value, and
  - one equivalence class for something that is not a "must be" value

Identification of Test Cases

Test cases for each equivalence class can be identified by:

- Assign a unique number to each equivalence class
- For each equivalence class with valid input that has not been covered by test cases yet, write a new test case covering as many uncovered equivalence classes as possible
- For each equivalence class with invalid input that has not been covered by test cases, write a new test case that covers one and only one of the uncovered equivalence classes

Example

- A program reads three numbers A, B and C with range [1,50] and prints largest number. Design all test cases for this program using equivalence class testing technique.

I1 = {<A,B,C> : 1 ≤ A ≤ 50}
I2 = {<A,B,C> : 1 ≤ B ≤ 50}
I3 = {<A,B,C> : 1 ≤ C ≤ 50}
I4 = {<A,B,C> : A < 1}
I5 = {<A,B,C> : A > 50}
I6 = {<A,B,C> : B < 1}
I7 = {<A,B,C> : B > 50}
I8 = {<A,B,C> : C < 1}
I9 = {<A,B,C> : C > 50}

Test case ID	A	B	C	Expected result	Classes covered by the test case
1	13	25	36	C is greatest	I_1, I_2, I_3
2	0	13	45	Invalid input	I_4
3	51	34	17	Invalid input	I_5
4	29	0	18	Invalid input	I_6
5	36	53	32	Invalid input	I_7
6	27	42	0	Invalid input	I_8
7	33	21	51	Invalid input	I_9

Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University
Pooja Malhotra

Example

- I1 = {<A,B,C> : A > B, A > C}
- I2 = {<A,B,C> : B > A, B > C}
- I3 = {<A,B,C> : C > A, C > B}
- I4 = {<A,B,C> : A = B, A ≠ C}
- I5 = {<A,B,C> : B = C, A ≠ B}
- I6 = {<A,B,C> : A = C, C ≠ B }
- I7 = {<A,B,C> : A = B = C}

Test case ID	A	B	C	Expected Result	Classes Covered by the test case
1	25	13	13	A is greatest	I_1, I_5
2	25	40	25	B is greatest	I_2, I_6
3	24	24	37	C is greatest	I_3, I_4
4	25	25	25	All three are equal	I_7

Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University
Pooja Malhotra

Advantages of Equivalence Class Partitioning

- A small number of test cases are needed to adequately cover a large input domain
- One gets a better idea about the input domain being covered with the selected test cases
- The probability of uncovering defects with the selected test cases based on equivalence class partitioning is higher than that with a randomly chosen test suite of the same size
- The equivalence class partitioning approach is not restricted to input conditions alone – the technique may also be used for output domains

Decision Table Based Testing

A major limitation of the EC-based testing is that it only considers each input separately. The technique does not consider combining conditions.

Different combinations of equivalent classes can be tried by using a new technique based on the decision table to handle multiple inputs.

Formation of Decision Table

	ENTRY					
Condition Stub		Rule 1	Rule 2	Rule 3	Rule 4	...
	C1	True	True	False	I	
	C2	False	True	False	True	
	C3	True	True	True	I	
Action Stub	A1		X			
	A2	X			X	
	A3			X		

- Condition Stub
- Action Stub
- Condition Entry
- Action Entry

Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University
Pooja Malhotra

Formation of Decision Table

- It comprises a set of conditions (or, causes) and a set of effects (or, results) arranged in the form of a column on the left of the table
- In the second column, next to each condition, we have its possible values: Yes (Y), No (N), and Don't Care (Immaterial) state.
- To the right of the "Values" column, we have a set of rules. For each combination of the three conditions {C1,C2,C3}, there exists a rule from the set {R1,R2, ..}
- Each rule comprises a Yes (Y), No (N), or Don't Care ("-") response, and contains an associated list of effects(actions) {E1,E2,E3}
- For each relevant effect, an effect sequence number specifies the order in which the effect should be carried out, if the associated set of conditions are satisfied
- Each rule of a decision table represents a test case

Test case design using decision table

The steps for developing test cases using decision table technique:

- **Step 1:** The test designer needs to identify the conditions and the actions/effects for each specification unit.
  - A condition is a distinct input condition or an equivalence class of input conditions
  - An action/effect is an output condition. Determine the logical relationship between the conditions and the effects
- **Step 2:** List all the conditions and actions in the form of a decision table. Write down the values the condition can take.
- **Step 3:** Fill the columns with all possible combinations – each column corresponds to one combination of values.
- Step 4: Define rules by indicating what action occurs for a set of conditions.

Test case design using decision table

- Interpret condition stubs as the inputs for the test case.
- Interpret action stubs as the expected output for the test case.
- Rule, which is the combination of input conditions becomes the test case itself.
- The columns in the decision table are transformed into test cases.
- If there are K rules over n binary conditions, there are at least K test cases and at the most 2^n test cases.

Decision Table Based Testing

Example

- A program calculates the total salary of an employee with the conditions that if the working hours are less than or equal to 48, then give normal salary. The hours over 48 on normal working days are calculated at the rate of 1.25 of the salary. However, on holidays or Sundays, the hours are calculated at the rate of 2.00 times of the salary. Design the test cases using decision table testing.

Decision Table Based Testing

The decision table for the program is shown below:

ENTRY				
		Rule 1	Rule 2	Rule3
Condition Stub	C1: Working hours > 48	I	F	T
	C2: Holidays or Sundays	T	F	F
Action Stub	A1: Normal salary		X	
	A2: 1.25 of salary			X
	A3: 2.00 of salary	X		

The test cases derived from the decision table are given below:

Test Case ID	Working Hour	Day	Expected Result
1	48	Monday	Normal Salary
2	50	Tuesday	1.25 of salary
3	52	Sunday	2.00 of salary

Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University
Pooja Malhotra

Dynamic Testing: White Box Testing Techniques

White-box testing (also known as clear box testing, glass box testing, transparent box testing, and structural testing) is a method of testing software that tests internal structures or workings of an application. White-box testing can be applied at the unit, integration and system levels of the software testing process.

- White box testing needs the full understanding of the logic/structure of the program.
- Test case designing using white box testing techniques
  - Control Flow testing method
    - Basis Path testing method
    - Loop testing
  - Data Flow testing method
  - Mutation testing method
- Control flow refers to flow of control from one instruction to another
- Data flow refers to propagation of values from one variable or constant to another variable

Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University

Basis Path Testing

Basis path testing is the technique of selecting the paths that provide a basis set of execution paths through the program.

- Path Testing is based on control structure of the program for which flow graph is prepared.
- requires complete knowledge of the program's structure.
- closer to the developer and used by him to test his module.
- The effectiveness of path testing is reduced with the increase in size of software under test.
- Choose enough paths in a program such that maximum logic coverage is achieved.

Control Flow Graph

The control flow graph is a graphical representation of control structure of a program. Flow graphs can be prepared as a directed graph. A directed graph (V, E) consists of a set of vertices V and a set of edges E that are ordered pairs of elements of V. Based on the concepts of directed graph, following notations are used for a flow graph:

Node: It represents one or more procedural statements. The nodes are denoted by a circle. These are numbered or labelled.

Edges or links: They represent the flow of control in a program. This is denoted by an arrow on the edge. An edge must terminate at a node.

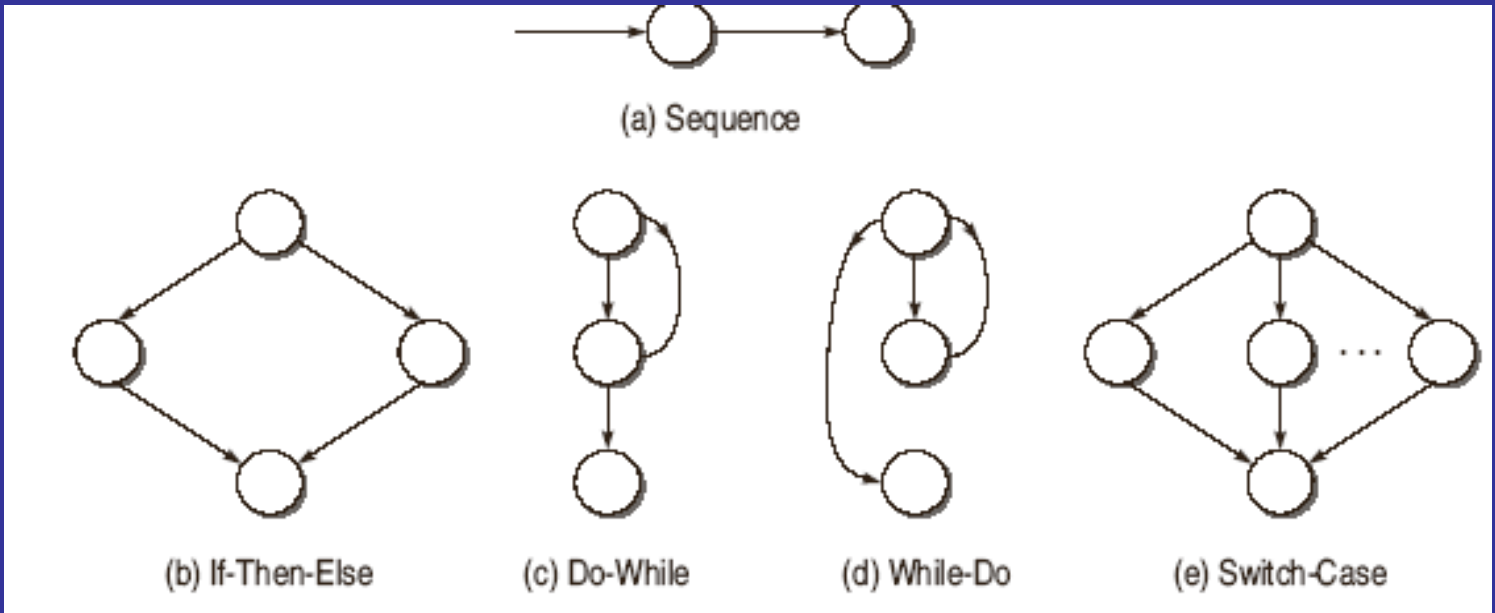
Decision node: A node with more than one arrow leaving it is called a decision node.

Junction node: A node with more than one arrow entering it is called a junction.

Regions: Areas bounded by edges and nodes are called regions. When counting the regions, the area outside the graph is also considered a region.

Control Flow Graph

Flow Graph Notations for Different Programming Constructs



Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University

Path Testing Terminology

Path: A path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same, junction, decision, or exit.

Segment: Paths consist of segments. The smallest segment is a link, that is, a single process that lies between two nodes (e.g., junction-process-junction, junction process-decision, decision-process-junction, decision-process-decision).

Path Segment: A path segment is a succession of consecutive links that belongs to some path.

Length of a Path: The length of a path is measured by the number of links in it and not by the number of instructions or statements executed along the path. An alternative way to measure the length of a path is by the number of nodes traversed.

Independent Path: An independent path is any path through the graph that introduces at least one new set of processing statements or new conditions. An independent path must move along at least one edge that has not been traversed before the path is defined.

Path Testing Terminology

Cyclomatic complexity is a software metric used to indicate the complexity of a program. It is a quantitative measure of the number of linearly independent paths through a program's source code. It was developed by Thomas J. McCabe, Sr. in 1976.

- The testing strategy, called basis path testing by McCabe who first proposed it, is to test each linearly independent path through the program; in this case, the number of test cases will equal the cyclomatic complexity of the program.
- **Cyclomatic Complexity (logical complexity of program)**

Cyclomatic complexity number can be derived through any of the following three formulae

1. V(G) = e – n + 2p where e is number of edges, n is the number of nodes in the graph, and p is number of components in the whole graph.
2. V(G) = d + p where d is the number of decision nodes in the graph.
3. V(G) = number of regions in the graph.

Path Testing Terminology

- **Calculating the number of decision nodes for Switch-Case/Multiple If-Else :**

When a decision node has exactly two arrows leaving it, then we count it as a single decision node. However, switch-case and multiple if-else statements have more than two arrows leaving a decision node, and in these cases, the formula to calculate the number of nodes is

d = k – 1, where k is the number of arrows leaving the node.

- **Calculating the cyclomatic complexity number of the program having many connected components:**

Let us say that a program P has three components: X, Y, and Z. Then we prepare the flow graph for P and for components, X, Y, and Z. The complexity number of the whole program is

V(G) = V(P) + V(X) + V(Y) + V(Z)

We can also calculate the cyclomatic complexity number of the full program with the first formula by counting the number of nodes and edges in all the components of the program collectively and then applying the formula

V(G) = e – n + 2P

Guidelines for Basis Path Testing

We can use the cyclomatic complexity number in basis path testing. Cyclomatic number, which defines the number of independent paths, can be utilized as an upper bound for the number of tests that must be conducted to ensure that all the statements have been executed at least once. Thus, independent paths are prepared according to the upper limit of the cyclomatic number. The set of independent paths becomes the basis set for the flow graph of the program. Then test cases can be designed according to this basis set.

The following steps should be followed for designing test cases using path testing:

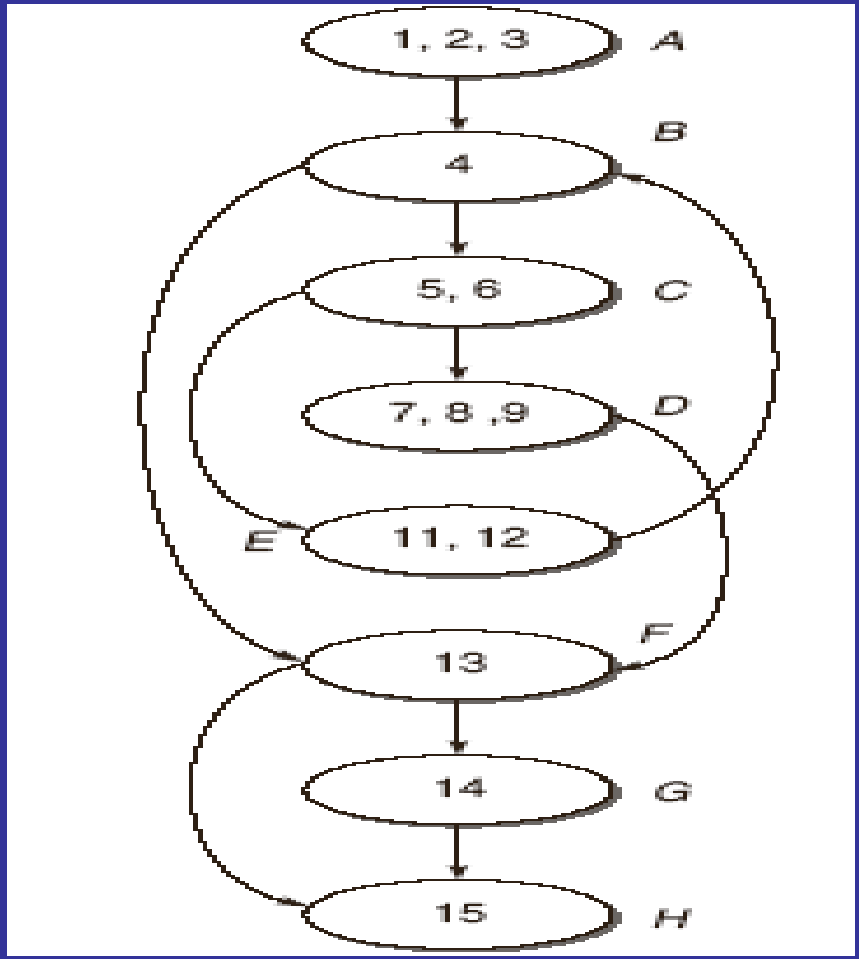
- Draw the fl ow graph using the code provided for which we have to write test cases.
- Determine the cyclomatic complexity of the flow graph.
- Cyclomatic complexity provides the number of independent paths.
- Determine a basis set of independent paths through the program control structure.
- The basis set is in fact the base for designing the test cases. Based on every independent path, choose the data such that this path is executed.

Example: Consider the following program segment:

```
main()
{
      int number, index;
1.    printf("Enter a number");
2.    scanf("%d, &number);
3.    index = 2;
4.    while(index <= number - 1)
5.    {
6.          if (number % index == 0)
7.          {
8.                printf("Not a prime number");
9.                break;
10.         }
11.         index++;
12.         }
13.          if(index == number)
14.                printf("Prime number");
15.   } //end main
```

1. Draw the DD graph for the program.
2. Calculate the cyclomatic complexity of the program using all the methods.
3. List all independent paths.
4. Design test cases from independent paths.

Example



Example

Cyclomatic Complexity

V(G) = e – n + 2 * P

= 10 – 8 +2

= 4

V(G) = Number of predicate nodes + 1

= 3 (Nodes B,C and F) + 1

= 4

V(G) = No. of Regions

- = 4 (R1, R2, R3, R4)

Example

Independent Paths

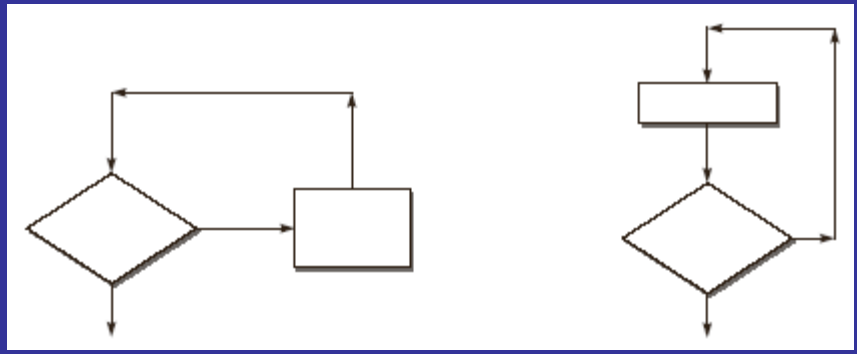
- A-B-F-H
- A-B-F-G-H
- A-B-C-E-B-F-G-H
- A-B-C-D-F-H

Test case ID	Input num	Expected result	Independent paths covered by test case
1	1	No output is displayed	A-B-F-H
2	2	Prime number	A-B-F-G-H
3	4	Not a prime number	A-B-C-D-F-H
4	3	Prime number	A-B-C-E-B-F-G-H

Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University

Loop Testing

Simple Loops

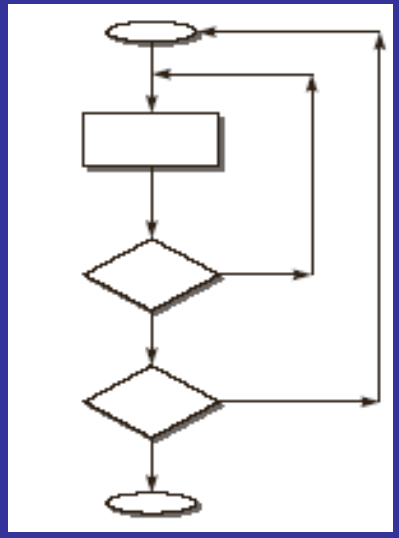


- Check whether you can bypass the loop or not. If the test case for bypassing the loop is executed and, still you enter inside the loop, it means there is a bug.
- Check whether the loop control variable is negative.
- Write one test case that executes the statements inside the loop.
- Write test cases for a typical number of iterations through the loop.
- Write test cases for checking the boundary values of maximum and minimum number of iterations defined (say min and max) in the loop. It means we should test for the min, min+1, min-1, max-1, max and max+1 number of iterations through the loop.

Loop Testing

Nested Loops: Nested loops When two or more loops are embedded, it is called a nested loop.

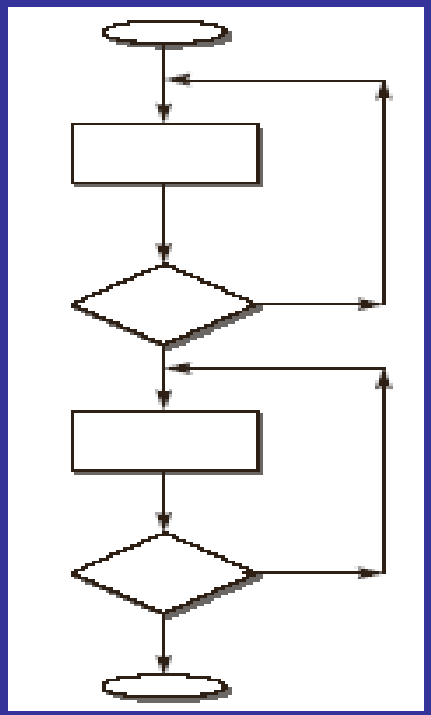
The the strategy is to start with the innermost loops while holding outer loops to their minimum values. Continue this outward in this manner until all loops have been covered



Loop Testing

Concatenated Loops:

Loops are concatenated if it is possible to reach one after exiting the other while still on a path from entry to exit.



Data Flow Testing

A program unit accepts inputs, performs computations, assigns new values to variables, and returns results.

- One can visualize of "flow" of data values from one statement to another.
- A data value produced in one statement is expected to be used later.

Example:

Obtain a file pointer ……. use it later.

If the later use is never verified, we do not know if the earlier assignment is write.

Two motivations of data flow testing:

1. The memory location for a variable is accessed in a "desirable" way.
2. Verify the correctness of data values "defined" (i.e. generated) – observe that all the "uses" of the value produce the desired results.

A programmer can perform a number of tests on data values. These tests are collectively known as data flow testing.

Data Flow Testing

Data flow testing is a white-box testing technique that can be used to detect improper use of data values due to coding errors. Errors may be unintentionally introduced in a program by programmers.

For instance, a programmer might use a variable without defining it. Moreover, he may define a variable, but not initialize it and then use that variable in a predicate.

For example,

```
int a;
 if(a == 67) { }
```

In this way, data flow testing gives a chance to look out for inappropriate data definition, its use in predicates, computations, and termination. It identifies potential bugs by examining the patterns in which that piece of data is used.

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.

Data Flow Testing

Detect improper use of data values due to coding errors.

Closely examines the state of the data in the control flow graph resulting in a richer test suite than the one obtained from control flow graph based path testing strategies like branch coverage, all statement coverage, etc.

Data Flow Testing

States of data object

Defined (d):

A data object is called defined when it is initialized, i.e. when it is on the left side of an assignment statement

A=9; file opened etc.

Killed / Undefined / Released (k):

When the data has been reinitialized or the scope of a loop control variable finishes, i.e. exiting the loop or memory is released dynamically or a file has been closed.

Reinitialized data, exiting from loop, closed file ,release of memory etc.

Usage (u):

When the data object is on the right side of assignment or used as a control variable in a loop, or in an expression used to evaluate the control flow of a case statement, or as a pointer to an object, etc.

Computational use (c-use) or predicate use (p-use).

Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University

Data Flow Testing

Data-Flow Anomalies: Data-flow anomalies represent the patterns of data usage which may lead to an incorrect execution of the code.

- **Two-character Data-Flow Anomalies**

Anomaly	Explanation	Effect of Anomaly
du	Define-use	Allowed. Normal case.
dk	**Define-kill**	**Potential bug. Data is killed without use after definition.**
ud	Use-define	Data is used and then redefined. Allowed. Usually not a bug because the language permits reassignment at almost any time.
uk	Use-kill	Allowed. Normal situation.
ku	**Kill-use**	**Serious bug because the data is used after being killed.**
kd	Kill-define	Data is killed and then redefined. Allowed.
dd	**Define-define**	**Redefining a variable without using it. Harmless bug, but not allowed.**
uu	Use-use	Allowed. Normal case.
kk	**Kill-kill**	**Harmless bug, but not allowed.**

Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University

Data Flow Testing

One -character Data-Flow Anomalies

Anomaly	Explanation	Effect of Anomaly
~d	First definition	Normal situation. Allowed.
~u	**First Use**	**Data is used without defining it. Potential bug.**
~k	**First Kill**	**Data is killed before defining it. Potential bug.**
D~	**Define last**	**Potential bug.**
U~	Use last	Normal case. Allowed.
K~	Kill last	Normal case. Allowed.

It can be observed that not all data-flow anomalies are harmful, but most of them are suspicious and indicate that an error can occur.

Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University

Data Flow Testing

Terminology used in Data Flow Testing

- **Definition Node:** Defining a variable means assigning value to a variable for the very first time in a program.

  Input statements, Assignment statements, Loop control statements, Procedure calls, etc.
- **Usage Node:** It means the variable has been used in some statement of the program.

  Output statements, Assignment statements (Right), Conditional statements, Loop control statements, etc.
- **Loop Free Path Segment:** Path segment for which every node is visited once at most.
- **Simple Path Segment: Segment:** Path segment in which at most one node is visited twice.
- **Definition-Use Path (du-path)**

  A du-path with respect to a variable v is a path between definition node and usage node of that variable. Usage node can be p-usage or c- usage node.
- **Definition-Clear path(dc-path)**

  A dc-path with respect to a variable v is a path between definition node and usage node such that no other node in the path is a defining node of variable v.

Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University

Data Flow Testing

- The du-paths which are not dc-paths are important from testing viewpoint, as these are potential problematic spots for testing persons.
- Those du-paths which are definition-clear are easy to test in comparison to du-paths which are not dc-paths

Data Flow Testing

Static Data Flow Testing : With static analysis, the source code is analyzed without executing it.

Dynamic Data-Flow Testing : Dynamic data flow testing is performed with the intention to uncover possible bug in data usage during the execution of the code.

Strategies to create test cases:

- **All-du Paths (ADUP)**
- **All-uses (AU)**
- **All-p-uses / Some-c-uses (APU + C)**
- **All-c-uses / Some-p-uses (ACU + P)**
- **All-Predicate-Uses(APU)**
- **All-Computational-Uses(ACU)**
- **All-Definition (AD)**

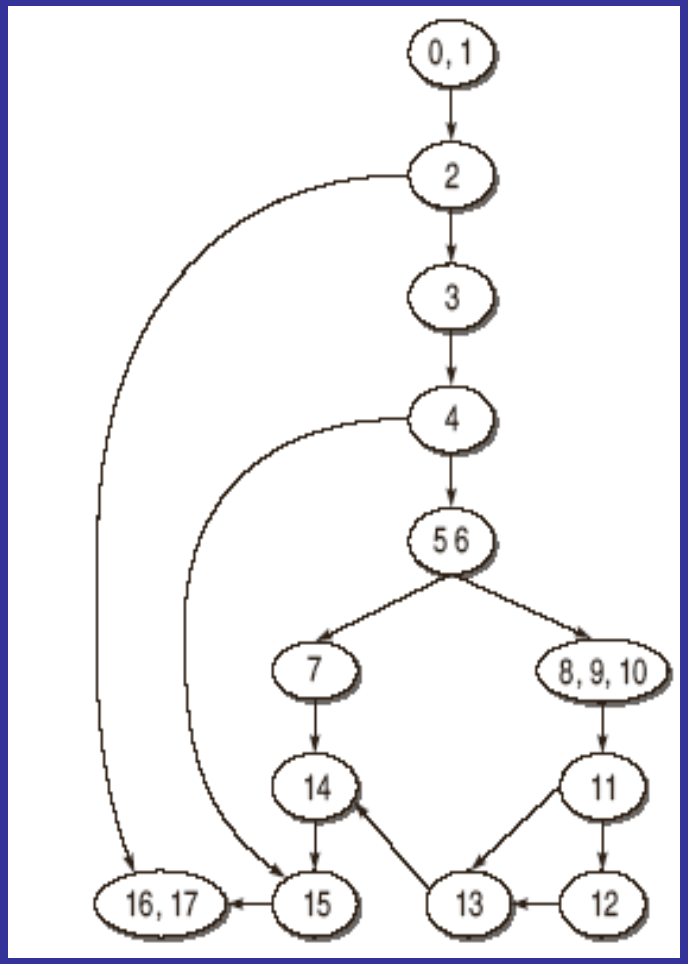
Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University

Data Flow Testing

```
     main()
     {
     int work;
 0.  double payment =0;
 1.  scanf("%d", work);
 2.  if (work > 0) {
 3.      payment = 40;
 4.  if (work > 20)
 5.  {
 6.      if(work <= 30)
 7.          payment = payment + (work - 25) * 0.5;
 8.      else
 9.      {
10.          payment = payment + 50 + (work -30) * 0.1;
11.              if (payment >= 3000)
12.                  payment = payment * 0.9;
13.      }
14.  }
15.  }
16.  printf("Final payment", payment);
```


Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University

Data Flow Testing



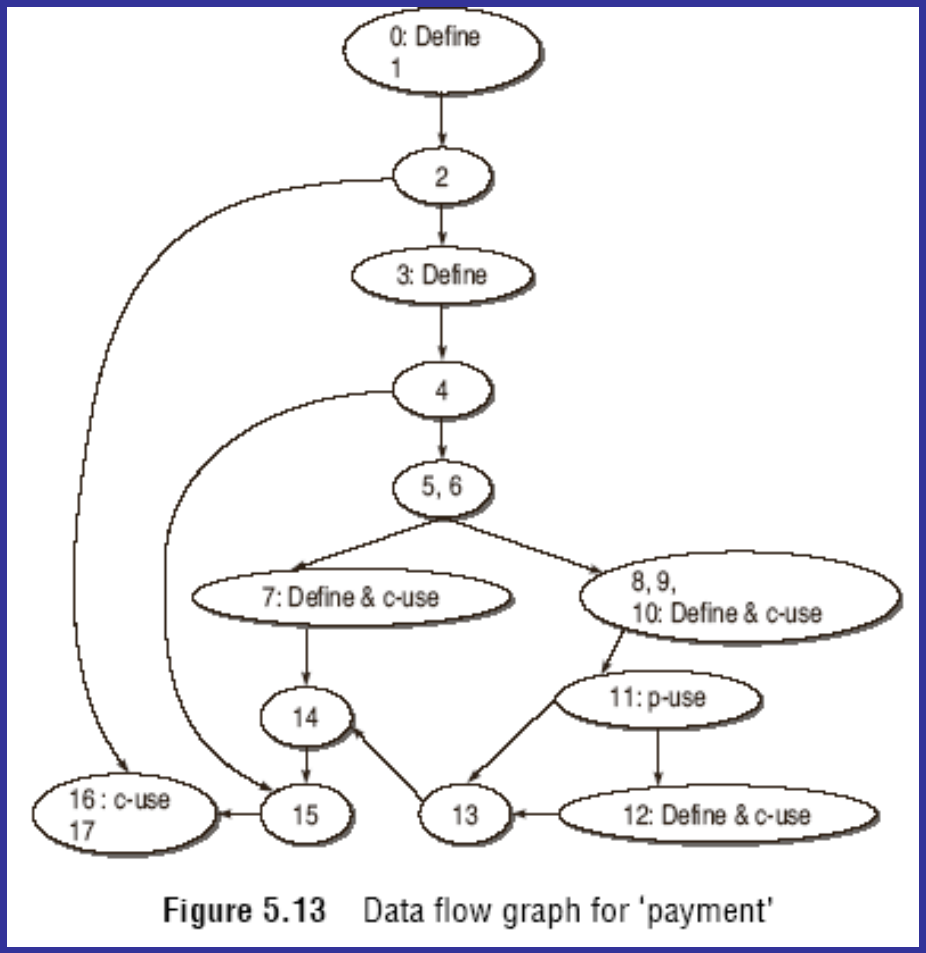


Figure 5.13 Data flow graph for 'payment'

Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University

Data Flow Testing

Find :

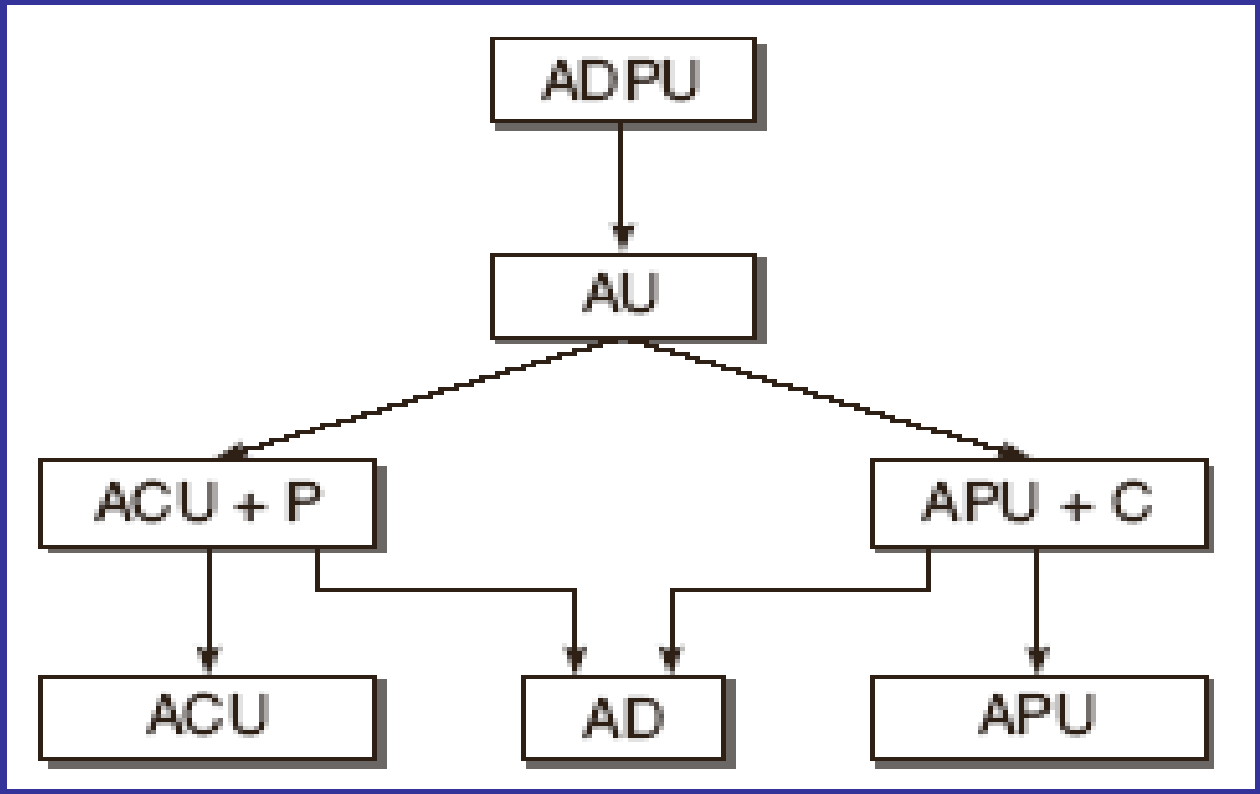
du Paths

dc Paths

for variable payment.

Variable	Defined at	Used at
Payment	0,3,7,10,12	7,10,11,12,16

Data Flow Testing



Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University

Mutation Testing

Mutation testing is a technique that focuses on measuring the adequacy of test data (or test cases).

The original intention behind mutation testing was to expose and locate weaknesses in test cases. Thus, mutation testing is a way to measure the quality of test cases.

Mutation Testing

- Mutation testing is the process of mutating some segment of code(putting some error in the code) and then testing this mutated code with some test data. If the test data is able to detect the mutations in the code, then the test data is quite good.

- Mutation testing helps a user create test data by interacting with the user to iteratively strengthen the quality of test data. During mutation testing, faults are introduced into a program by creating many versions of the program, each of which contains one fault. Test data are used to execute these faulty programs with the goal of causing each faulty program to fail.

- Faulty programs are called *mutants* of the original program and a mutant is said to be *killed* when a test case causes it to fail. When this happens, the mutant is considered *dead*

Mutation Testing

- Modify a program by introducing a single small change to the code
- A modified program is called *mutant*
- A mutant is said to be *killed* when the execution of test case cause it to fail. The mutant is considered to be *dead*
- A mutant is an *equivalent* to the given program if it always produce the same output as the original program
- A mutant is called *killable* or *stubborn*, if the existing set of test cases is insufficient to kill it.

Mutation Score

A mutation *score* for a set of test cases is the percentage of non-equivalent mutants *killed* by the test suite

100*D/(N-E) where

D -> Dead

N-> Total No of Mutants

E-> No of equivalent mutants

The test suite is said to be *mutation-adequate* if its mutation score is 100%

Mutation Testing

Primary Mutants:

- Let us take one example of C program shown below

```
…
If (a>b)
    x = x + y;
else
    x = y;
printf("%d",x);
….
```

We can consider the following mutants for above example:

- M1: x = x – y;
- M2: x = x / y;
- M3: x = x+1;
- M4: printf("%d",y);

Mutation Testing

Test Data	x	y	Initial Program Result	Mutant Result
TD1	2	2	4	0 (M1)
TD2(x and y # 0)	4	3	7	1.4 (M2)
TD3 (y #1)	3	2	5	4 (M3)
TD4(y #0)	5	2	7	2 (M4)

Secondary Mutants:

Multiple levels of mutation are applied on the initial program.

Example Program:

If(a<b)

c=a;

Mutant for this code may be :

If(a==b)

c=a+1;

Mutation Testing Process

- Step 1: Begin with a program *P and a set of test cases T known to be correct.*
- Step 2: Run each test case in *T against the program P.*
  - *If it fails (o/p incorrect) P must be modified and restarted. Else, go to step 3*
- Step 3: Create a set of mutants {*Pi }, each differing from P by a simple, syntactically* correct modification of *P.*

Mutation Testing Process

- Step 4: Execute each test case in *T against each mutant Pi .*
- *If the o/p is differ* → the mutant *Pi is considered incorrect and is said to be killed by the test case*
- If *Pi* produces exactly the same results:
  - *P and Pi are equivalent*
  - *Pi is killable (*new test cases must be created to kill it)

Mutation Testing Process

- Step 5: Calculate the mutation score for the set of test cases *T.*
- Mutation score = 100×*D/(N −E),*

- Step 6: If the estimated mutation adequacy of *T in step 5 is not sufficiently high,* then design a new test case that distinguishes *Pi from P, add the new* test case to *T, and go to step 2.*

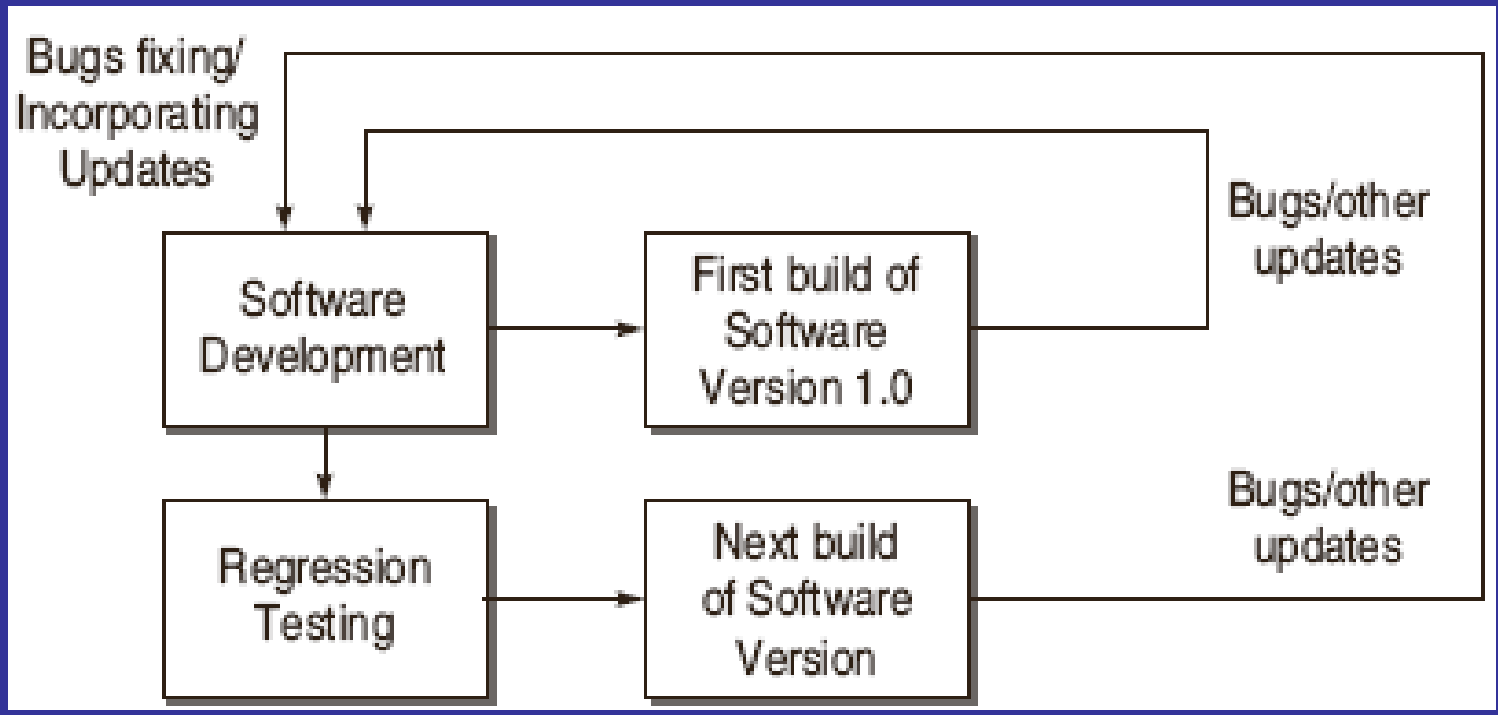
Regression Testing

Progressive Vs Regressive Testing

- Regression testing is defined as a type of software testing to confirm that a recent program or code change has not adversely affected existing features.
- Regression testing is nothing but full or partial selection of already executed test cases which are re-executed to ensure existing functionalities work fine.
- This testing is done to make sure that new code changes should not have side effects on the existing functionalities. It ensures that old code still works once the new code changes are done.
- Regression Testing is necessary to maintain software whenever there is update in it.
- Regression testing is not another testing activity. Rather, it is the re-execution of some or all of the already developed test cases.
- Regression testing increases quality of software.

Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University

Regression Testing produces Quality Software



Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University

Objectives of Regression Testing

- Regression Tests to check that the bug has been addressed
- Regression Tests to find other related bugs
- Regression tests to check the effect on other parts of the program

Need / When to do regression testing?

- **Software maintenance**
  - **Corrective maintenance**
  - **Adaptive maintenance**
  - **Perfective maintenance**
  - **Preventive maintenance**

Adaptive – modifying the system to cope with changes in the software environment (DBMS, OS)
Perfective – implementing new or changed user requirements which concern functional enhancements to the software
Corrective – diagnosing and fixing errors, possibly ones found by users
Preventive – increasing software maintainability or reliability to prevent problems in the future

- **Rapid iterative development**
- **First step of integration**
- **Compatibility assessment and benchmarking**

Regression Testing Types

Bug-Fix Regression

Side-Effect Regression / Stability Regression

Regression Testing Types

Bug-Fix regression:

This testing is performed after a bug has been reported and fixed. Its goal is to repeat the test cases that expose the problem in the first place.

Side-Effect regression/Stability regression:

It involves retesting a substantial part of the product. The goal is to prove that the change has no detrimental effect on something that was earlier in order. It tests the overall integrity of the program, not the success of software fixes.

Usability testing

- The testing that validates the ease of use, speed, and aesthetics of the product from the user's point of view is called usability testing.
- some of the characteristics of “usability testing” or “usability validation” are as follows:
  - Usability testing tests the product from the users’ point of view. It encompasses a range of techniques for identifying how users actually interact with and use the product.
  - Usability testing is for checking the product to see if it is easy to use for the various categories of users.
  - Usability testing is a process to identify discrepancies between the user interface of the product and the human user requirements, in terms of the pleasantness and aesthetics aspects.

Usability testing

From the above definition it is easy to conclude that Something that is easy for one user may not be easy for another user due to different types of users a product can have . Something what is considered fast (interms of say, response time) by one user may be slow for another user as the machines used by them and the expectations of speed can be different. Something that is considered beautiful by someone may look ugly to another. A view expressed by one user of the product may not be the view of another.

Usability testing

Throughout the industry, usability testing is gaining momentum as sensitivity towards usability in products is increasing and it is very difficult to sell a product that does not meet the usability requirements of the users. There are several standards (for example, accessibility guidelines), organizations, tools (for example, Microsoft Magnifier), and processes that try to remove the subjectivity and improve the objectivity of usability testing.

Usability testing

Usability testing is not only for product binaries or executables. It also applies to documentation and other deliverables that are shipped along with a product. The release media should also be verified for usability. Let us take an example of a typical AUTORUN script that automatically brings up product setup when the release media is inserted in the machine. Sometimes this script is written for a particular operating system version and may not get auto executed on a different OS version. Even though the user can bring up the setup by clicking on the setup executable manually, this extra click (and the fact that the product is not automatically installed) may be considered as an irritant by the person performing the installation.

Who performs Usability testing

Generally, the people best suited to perform usability testing are typical representatives of the actual user segments who would be using the product, so that the typical user patterns can be captured, and People who are new to the product, so that they can start without any bias and be able to identify usability problems. A person who has used the product several times may not be able to see the usability problems in the product as he or she would have “got used” to the product's (potentially inappropriate) usability. Hence, a part of the team performing usability testing is selected from representatives outside the testing team. Inviting customer-facing teams (for example, customer support, product marketing) who know what the customers want and their expectations, will increase the effectiveness of usability testing.

Deliverables /Usability testing

A right approach for usability is to test every artifact that impacts users—such as product binaries, documentation, messages, media—covering usage patterns through both graphical and command user interfaces, as applicable.

Usability testing

Usability should not be confused with graphical user interface (GUI). Usability is also applicable to non-GUI interface such as command line interfaces (CLI). A large number of Unix/Linux users find CLIs more usable than GUIS. SQL command is another example of a CLI, and is found more usable by database users. Hence, usability should also consider CLI and other interfaces that are used by the users

WHEN TO DO USABILITY TESTING?

The most appropriate way of ensuring usability is by performing the usability testing in two phases. **First is design validation and the second is usability testing** done as a part of component and integration testing phases of a test cycle. When planning for testing, the usability requirements should be planned in parallel, upfront in the development cycle, similar to any other type of testing. Generally, however, usability is an ignored subject (or at least given less priority) and is not planned and executed from the beginning of the project. When there are two defects—one on functionality and other on usability—the functionality defect is usually given precedence. This approach is not correct as usability defects may demotivate users from using the software (even if it performs the desired function) and it may mean a huge financial loss to the product organization if users reject the product. Also, postponing usability testing in a testing cycle can prove to be very expensive as a large number of usability defects may end up as needing changes in design and needing fixes in more than one screen, affecting different code paths. All these situations can be avoided if usability testing is planned upfront.

WHEN TO DO USABILITY TESTING?

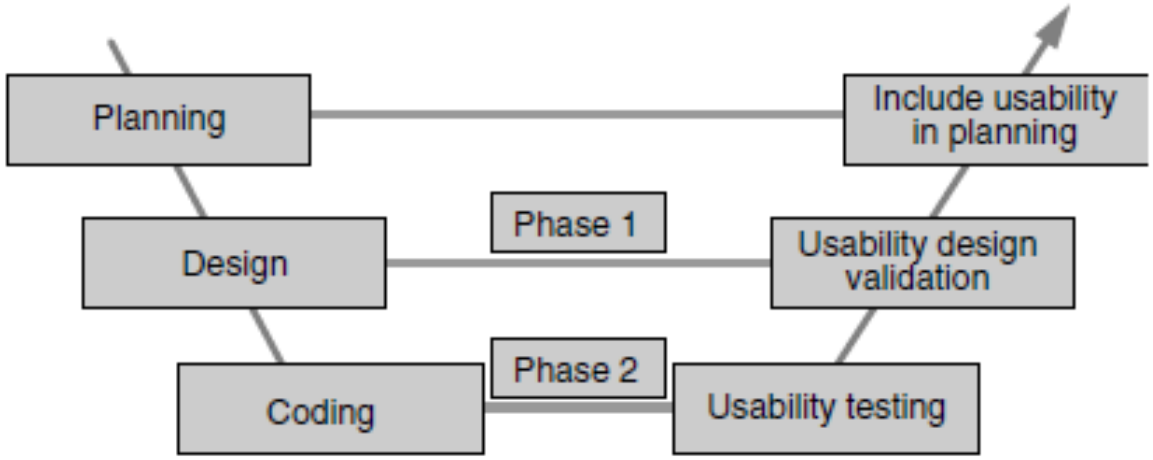


Figure 12.2 Phases and activities of usability testing.

Reference: Software Testing Principles and Practices,Naresh Chauhan , Oxford University

WHEN TO DO USABILITY TESTING?

Usability design is verified through several means. Some of them are as follows:

- Style sheets : Style sheets are grouping of user interface design elements. Use of style sheets ensures consistency of design elements across several screens and testing the style sheet ensures that the basic usability design is tested. Style sheets also include frames, where each frame is considered as a separate screen by the user. Style sheets are reviewed to check whether they force font size, color scheme, and so on, which may affect usability.
- Screen prototypes : Screen prototype is another way to test usability design. The screens are designed as they will be shipped to the customers, but are not integrated with other modules of the product. Therefore, this user interface is tested independently without integrating with the functionality modules. This prototype will have other user interface functions simulated such as screen navigation, message display, and so on. The prototype gives an idea of how exactly the screens will look and function when the product is released. The test team and some real-life users test this prototype and their ideas for improvements are incorporated in the user interface. Once this prototype is completely tested, it is integrated with other modules of the product.

WHEN TO DO USABILITY TESTING?

- Paper designs : Paper design explores the earliest opportunity to validate the usability design, much before the actual design and coding is done for the product. The design of the screen, layout, and menus are drawn up on paper and sent to users for feedback. The users visualize and relate the paper design with the operations and their sequence to get a feel for usage and provide feedback. Usage of style sheets requires further coding, prototypes need binaries and resources to verify, but paper designs do not require any other resources. Paper designs can be sent through email or as a printout and feedback can be collected.
- Layout design : Style sheets ensure that a set of user interface elements are grouped and used repeatedly together. Layout helps in arranging different elements on the screen dynamically. It ensures arrangement of elements, spacing, size of fonts, pictures, justification, and so on, on the screen. This is another aspect that needs to be tested as part of usability design.

WHEN TO DO USABILITY TESTING?

- If an existing product is redesigned or enhanced, usability issues can be avoided by using the existing layout, as the user who is already familiar with the product will find it more usable. Making major usability changes to an existing product (for example, reordering the sequence of buttons on a screen) can end up confusing users and lead to user errors.
- **In the second phase**, tests are run to test the product for usability. Prior to performing the tests, some of the actual users are selected (who are new to the product and features) and they are asked to use the product. Feedback is obtained from them and the issues are resolved. Sometimes it could be difficult to get the real users of the product for usability testing. In such a case, the representatives of users can be selected from teams outside the product development and testing teams—for instance, from support, marketing, and sales teams. When to do usability also depends on the type of the product that is being developed.

QUALITY FACTORS FOR USABILITY

Some quality factors are very important when performing usability testing. As was explained earlier, usability is subjective and not all requirements for usability can be documented clearly. However focusing on some of the quality factors given below help in improving objectivity in usability testing are as follows.

Comprehensibility : The product should have simple and logical structure of features and documentation. They should be grouped on the basis of user scenarios and usage. The most frequent operations that are performed early in a scenario should be presented first, using the user interfaces. When features and components are grouped in a product, they should be based on user terminologies, not technology or implementation.

Consistency: A product needs to be consistent with any applicable standards, platform look-and-feel, base infrastructure, and earlier versions of the same product. Also, if there are multiple products from the same company, it would be worthwhile to have some consistency in the look-and-feel of these multiple products. Following same standards for usability helps in meeting the consistency aspect of the usability.

WHEN TO DO USABILITY TESTING?

Navigation : This helps in determining how easy it is to select the different operations of the product. An option that is buried very deep requires the user to travel to multiple screens or menu options to perform the operation. The number of mouse clicks, or menu navigations that is required to perform an operation should be minimized to improve usability. When users get stuck or get lost, there should be an easy option to abort or go back to the previous screen or to the main menu so that the user can try a different route.

Responsiveness : How fast the product responds to the user request is another important aspect of usability. This should not be confused with performance testing. Screen navigations and visual displays should be almost immediate after the user selects an option or else it could give an impression to the user that there is no progress and cause him or her to keep trying the operation again. Whenever the product is processing some information, the visual display should indicate the progress and also the amount of time left so that the users can wait patiently till the operation is completed. Adequate dialogs and popups to guide the users also improve usability.

USABILITY TESTING : AESTHETICS TESTING

AESTHETICS TESTING : Another important aspect in usability is making the product “beautiful.” Performing aesthetics testing helps in improving usability further. This testing is important as many of the aesthetics related problems in the product from many organizations are ignored on the ground that they are not functional defects. All the aesthetic problems in the product are generally mapped to a defect classification called “Cosmetic,” which is of low priority. Having a separate cycle of testing focusing on aesthetics helps in setting up expectations and also in focusing on improving the look and feel of the user interfaces. Aesthetics is not in the external look alone. It is in all the aspects such as messages, screens, colors, and images. A pleasant look for menus, pleasing colors, nice icons, and so on can improve aesthetics.

Accessibility testing

Accessibility Testing is a subset of usability testing, and it is performed to ensure that the application being tested is usable by people with disabilities like hearing, color blindness, old age and other disadvantaged groups.

- People with disabilities use assistive technology which helps them in operating a software product.

Accessibility testing involves testing these alternative methods of using the product and testing the product along with accessibility tools. Accessibility is a subset of usability and should be included as part of usability test planning.

Verifying the product usability for physically challenged users is called accessibility testing.

Accessibility testing

Accessibility testing may be challenging for testers because they are unfamiliar with disabilities. It is better to work with disabled people who have specific needs to understand their challenges.

Accessibility to the product can be provided by two means :

Making use of accessibility features provided by the underlying infrastructure (for example, operating system), called basic accessibility, and

Providing accessibility in the product through standards and guidelines, called product accessibility.

Accessibility testing

Basic Accessibility: Basic accessibility is provided by the hardware and operating system. All the input and output devices of the computer and their accessibility options are categorized under basic accessibility.

Examples:

Keyboard accessibility

Screen accessibility

- **Speech Recognition Software -** It will convert the spoken word to text , which serves as input to the computer.
- **Screen reader software** - Used to read out the text that is displayed on the screen
- **Screen Magnification Software**- Used to enlarge the monitor and make reading easy for vision-impaired users.
- **Special keyboard** made for the users for easy typing who have motion control difficulties.

Accessibility testing

Product Accessibility :A good understanding of the basic accessibility features is needed while providing accessibility to the product. A product should do everything possible to ensure that the basic accessibility features are utilized by it. For example, providing detailed text equivalent for multimedia files ensures the captions feature is utilized by the product.

Accessibility testing

Sample requirement #1: Text equivalents have to be provided for audio, video, and picture images.

Sample requirement #2: Documents and fields should be organized so that they can be read without requiring a particular resolution of the screen, and templates (known as style sheets).

Sample requirement #3: User interfaces should be designed so that all information conveyed with color is also available without color.

Sample requirement #4: Reduce flicker rate, speed of moving text; avoid flashes and blinking text.

Sample requirement #5: Reduce physical movement requirements for the users when designing the interface and allow adequate time for user responses.

Table 12.2 Sample list of usability and accessibility tools.

Name of the tool	Purpose
JAWS	For testing accessibility of the product with some assistive technologies.
HTML validator	To validate the HTML source file for usability and accessibility standards.
Style sheet validator	To validate the style sheets (templates) for usability standards set by W3C.
Magnifier	Accessibility tool for vision challenged (to enable them to enlarge the items displayed on screen
Narrator	Narrator is a tool that reads the information displayed on the screen and creates audio descriptions for vision-challenged users.
Soft keyboard	Soft keyboard enables the use of pointing devices to use the keyboard by displaying the keyboard template on the screen.

Accessibility testing

Following are the point's needs to be checked for application to be used by all users. This checklist is used for signing off accessibility testing.

- Whether an application provides keyboard equivalents for all mouse operations and windows?
- Whether instructions are provided as a part of user documentation or manual? Is it easy to understand and operate the application using the documentation?
- Whether tabs are ordered logically to ensure smooth navigation?
- Whether shortcut keys are provided for menus?
- Whether application supports all operating systems?
- Whether color of the application is flexible for all users?
- Whether images or icons are used appropriately, so it's easily understood by the end users?

Accessibility testing

- Whether an application has audio alerts?
- Whether user can adjust or disable flashing, rotating or moving displays?
- Check to ensure that color-coding is never used as the only means of conveying information or indicating an action
- Whether highlighting is viewable with inverted colors? Testing of color in the application by changing the contrast ratio
- Whether audio and video related content are properly heard by the disability people ? Test all multimedia pages with no speakers in websites
- Whether training is provided for users with disabilities that will enable them to become familiar with the software or application?

References:

1.Software Testing Principles and Practices, Naresh Chauhan, Second edition, Oxford Higher Education
2. Software Testing: Principles and Practice by Srinivasan Desikan, Gopalaswamy Ramesh