Software Testing (continued)

Organization of this Lecture:

- Review of last lecture.
- Data flow testing
- Mutation testing
- Cause effect graphing
- Performance testing.
- Test summary report
- Summary

Review of last lecture

- White box testing:
  - requires knowledge about internals of the software.
  - design and code is required.
  - also called structural testing.

Review of last lecture

- We discussed a few white-box test strategies.
  - statement coverage
  - branch coverage
  - condition coverage
  - path coverage

Data Flow–based Testing

Data flow–based testing focuses on how data (variables) are defined, used, and killed (lifetime).

The idea is to design test cases that cover all important **definition–use (DU) paths** of variables.

- **Def**: Where a variable is **assigned a value**.
- **Use**: Where a variable is **read/used**.
- **Kill**: Where a variable is no longer needed (goes out of scope).
- **Variable s is defined at s = x + y.**
  s is used in condition if s > 10.
- **These test cases in green box ensure the variable s's definition and usage are both exercised.**

```
def compute_sum(x, y):
    s = x + y      # Definition of s
    if s > 10:     # Use of s
        return "Large"
    else:
        return "Small"
```

Test Cases for Data Flow Coverage

- **Case 1:**

compute_sum(3, 4) → s = 7, path covers **def** → **use** → goes to **"Small"**.

- **Case 2:**

compute_sum(8, 5) → s = 13, path covers def → use → goes to "Large".

Data Flow-Based Testing

- **Selects test paths of a program according to the locations of:**
  - **definitions and uses of different variables in a program.**
- For a statement numbered S,
  - **DEF(S)** = {X/statement S contains a definition of X}
  - **USES(S)**= {X/statement S contains a use of X}
  - Example: 1: a=b; DEF(1)={a}, USES(1)={b}.
  - Example: 2: a=a+b;

    DEF(2)={a}USES(2)={a,b}.

Data Flow-Based Testing

- A variable X is said to be live at statement S1, if
  - X is defined at a statement S:
  - there exists a path from S to S1 not containing any definition of X.

DU Chain Example

```
X(){
 a=5; /* Defines variable a */
 While(C1) {
    if (C2)
          b=a*a;   /*Uses variable a */
          a=a-1; /* Defines variable a */
    }
  print(a); } /*Uses variable a */
```

Definition-use chain (DU chain)

- **[X,S,S1],**
  - S and S1 are statement numbers,
  - X in DEF(S)
  - X in USES(S1), and
  - **the definition of X in the statement S is live at statement S1.**

Data Flow-Based Testing

- One simple data flow testing strategy:
  - every DU chain in a program be covered at least once.
- Data flow testing strategies:
  - **useful for selecting test paths of a program containing nested if and loop statements**

```python
def compute_discount(price, discount_flag):
    # DEF(X) at B1
    x = price

    if discount_flag > 0:
        # DEF(X) at B2
        x = x * 0.9   # 10% discount
    else:
        # DEF(X) at B3
        x = x * 0.95  # 5% discount

    if x > 100:
        # USE(X) at B4
        print("High value purchase:", x)
    else:
        # USE(X) at B5
        print("Normal purchase:", x)
    # USE(X) at B6
    return x
```

Step 1: Identify DEF and USE

- **DEF(X):**
  - **B1:** `x = price`
  - **B2:** `x = x * 0.9`
  - **B3:** `x = x * 0.95`
- **USE(X):**
  - **B4:** `if x > 100`
  - **B5:** `print(x)` **(else branch)**
  - **B6:** `return x`

So:

$$DEF(X) = \{\mathbf{B1}, \mathbf{B2}, \mathbf{B3}\},$$
$$USE(X) = \{\mathbf{B4}, \mathbf{B5}. \mathbf{B6}\}$$

Step 2: Total DU Chains

- **Each DEF can reach each USE → 3 × 3 = 9DU chains.**
- Example chains:
  - **(B1 → B4), (B1 → B5), (B1 → B6)**
  - **(B2 → B4), (B2 → B5), (B2 → B6)**
  - **(B3 → B4), (B3 → B5), (B3 → B6)**

```python
def compute_discount(price, discount_flag):
    # DEF(X) at B1
    x = price

    if discount_flag > 0:
        # DEF(X) at B2
        x = x * 0.9   # 10% discount
    else:
        # DEF(X) at B3
        x = x * 0.95  # 5% discount

    if x > 100:
        # USE(X) at B4
        print("High value purchase:", x)
    else:
        # USE(X) at B5
        print("Normal purchase:", x)
    # USE(X) at B6
    return x
```

Step 3: Paths Covering Multiple DU Chains

We don't need 9 separate tests — some paths cover multiple chains.

1. **Path: B1 → B2 → B4 → B6**
   Covers: (B1→B4), (B2→B4), (B2→B6).
2. **Path: B1 → B2 → B5 → B6**
   Covers: (B1→B5), (B2→B5).
3. **Path: B1 → B3 → B4 → B6**
   Covers: (B1→B4 again), (B3→B4), (B3→B6).
4. **Path: B1 → B3 → B5 → B6**
   Covers: (B1→B5 again), (B3→B5).
5. **Path: B1 → B6 (skipping conditions, say when price=0)**
   Covers: (B1→B6).

In the example, there are **9 Definition–Use (DU) chains** for variable x.
Naively, you might think **9 separate test cases** are needed to cover them but
With **5 well-chosen paths**, all **9 DU chains** are covered.

Mutation Testing

- The software is first tested:
  - using an initial testing method based on white-box strategies we already discussed.
- After the initial testing is complete,
  - mutation testing is taken up.
- The idea behind mutation testing:
  - **make a few arbitrary small changes to a program at a time.**

Mutation Testing

- Each time the program is changed,
  - it is called a mutated program
  - the change is called a mutant.
- **A mutated program:**
  - **tested against the full test suite of the program.**
- If there exists at least one test case in the test suite for which:
  - a mutant gives an incorrect result, then the **mutant is said to be dead.**

Mutation Testing

- If a **mutant remains alive:**
  - even after all test cases have been exhausted,
  - the test suite is enhanced to kill the mutant.
- The process of generation and killing of mutants:
  - can be automated by predefining a set of primitive changes that can be applied to the program.

1. Mutation Testing

```
def is_even(x):
    return x % 2 == 0
```

```
def is_even(x):
    return (x % 2) == 0
```

fault-based testing technique where small, artificial changes (mutations) are made in the program code to create **mutant versions**. The goal is to check whether the existing test cases can **detect these changes**.

- If the test case fails for the mutant → the mutant is **killed**.

- If the test case passes (i.e., does not detect the change) → the mutant **survives** (test suite is weak).

  Purpose: To measure the **effectiveness of test cases**.

Mutation Testing

- The primitive changes can be:
  - altering an arithmetic operator,
  - changing the value of a constant,
  - changing a data type, etc.
- A major disadvantage of mutation testing:
  - computationally very expensive,
  - a large number of possible mutants can be generated.

Online Voting System

Original Logic

```
def cast_vote(user_id, has_voted):
    if not has_voted:
        record_vote(user_id)
        return "Vote cast successfully"
    else:
        return "User has already voted"
```

Effect of the Mutant

- Normal users still follow the rule → **1 vote each**.
- But if a user has `user_id == 9999`, they can **vote multiple times** because the condition bypasses `has_voted`.
- If test cases never check for `user_id == 9999`, this mutant **passes all tests** (survives).

Mutant Logic (Flawed)

```
def cast_vote(user_id, has_voted):
    if not has_voted or user_id == 9999:#
Mutant
        record_vote(user_id)
        return "Vote cast successfully"
    else:
        return "User has already voted"
```

Aspect	Mutation Testing	Data Flow Testing
Focus	Effectiveness of test suite (fault injection)	Variable definitions & usages
Method	Modify code (mutants) and re-run tests	Analyze variable DU paths
Goal	Kill all mutants with strong test cases	Ensure all data flows are covered
Example	Change == to != and see if test catches	Check s = x+y is used in condition

Additional Blackbox Testing

- **Cause and Effect graph**
- **Testing would be a lot easier:**
  - **if we could automatically generate test cases from requirements.**
- IBM researchers in the 1970s–1980s worked on:
- **Cause–Effect Graphing**: A systematic method to derive test cases from requirements by mapping **causes (inputs/conditions)** to **effects (outputs/actions)**.

Cause Effect Graph

- **is a black box testing technique that graphically illustrates the relationship between a given outcome and all the factors that influence the outcome**.
- A **graph** is used to represent the situations of combinations of input conditions.
- The graph is then converted to a **decision table** to obtain the test cases.

Cause and Effect Graphs

- Examine the requirements:
  - restate them as logical relation between inputs and outputs.
  - **The result is a Boolean graph** representing the relationships
    - called a cause-effect graph.

Convert the graph to a decision table:

each column of the decision table corresponds to a test case for functional testing.

Steps to create cause-effect graph

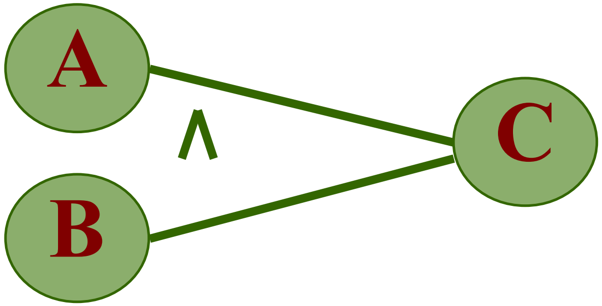
- Study the **functional requirements.**
- **Mark and number** all **causes and effects.**
- Numbered causes and effects:
  - become **nodes of the graph**.

Steps to create cause-effect graph

- Draw **causes** on the LHS
- Draw **effects** on the RHS
- Draw **logical relationship** between causes and effects
  - as **edges in the graph**.
- Extra nodes can be added
  - to simplify the graph

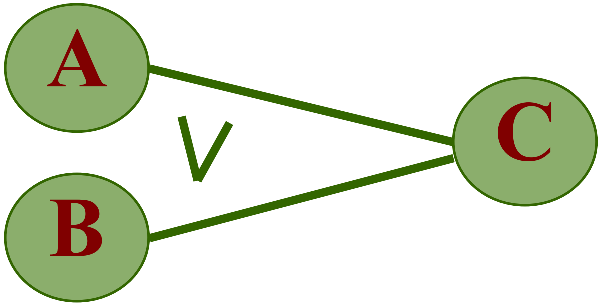
Drawing Cause-Effect Graphs

If A then B

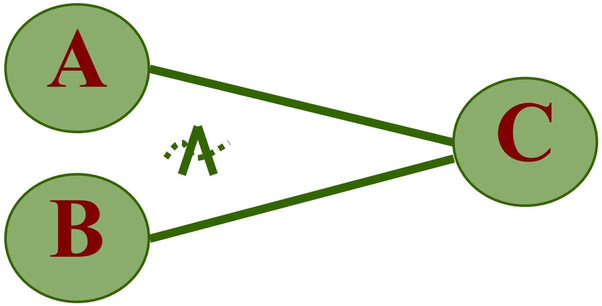


If (A and B)then C

Drawing Cause-Effect Graphs

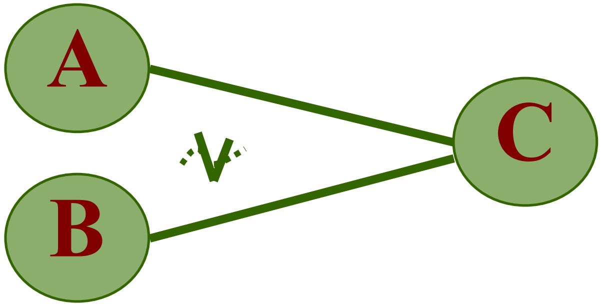


If (A or B)then C

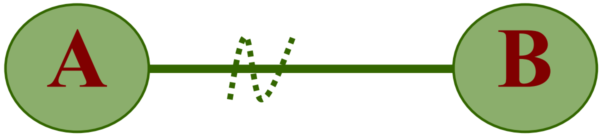


If (not(A and B))then C

Drawing Cause-Effect Graphs



If (not (A or B))then C



If (not A) then B

Example Case Study: Login System

Requirements:

1. User must enter a valid **username**.
2. User must enter a valid **password**.
3. If both are valid → **Login Success**.
4. If either is invalid → **Error Message**.
5. After 3 failed attempts → **Account Locked**.

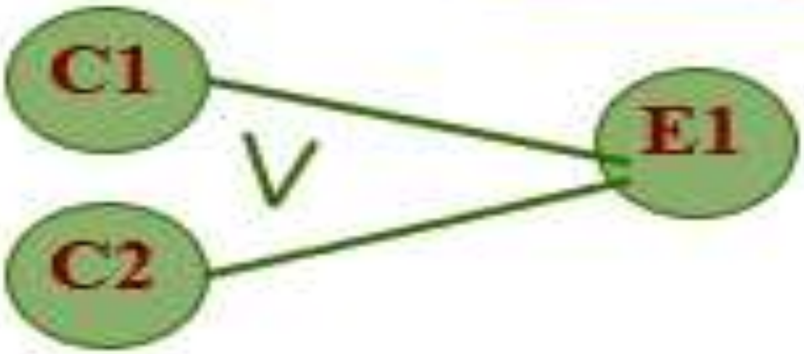
◆ Step 1: Identify Causes (Inputs)

- C1: Username is valid
- C2: Password is valid
- C3: Failed attempts ≥ 3

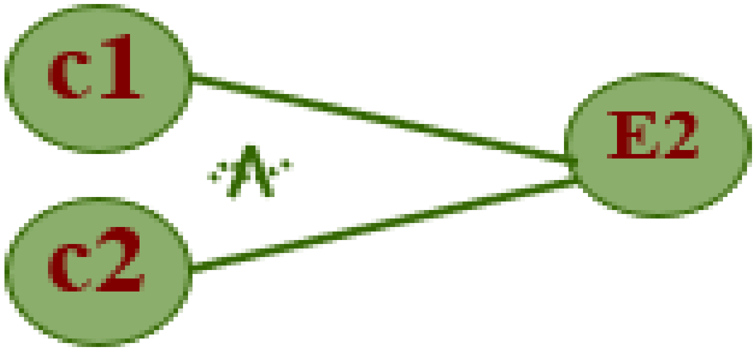
◆ Step 2: Identify Effects (Outputs)

- E1: Login Successful
- E2: Error Message Shown
- E3: Account Locked

Step 3: Cause–Effect Graph (Textual Representation)



If (C1AND C2)then E1(Login Success)



If not (C1AND C2)then E2(Error Message)

If C3 true then E3 Account locked

Step 4: Decision Table

Case	C1 (Username)	C2 (Password)	C3 (Attempts ≥3)	E1 (Login)	E2 (Error)	E3 (Locked)
1	T	T	F	✓	–	–
2	T	F	F	–	✓	–
3	F	T	F	–	✓	–
4	F	F	F	–	✓	–
5	–	–	T	–	–	✓

We can determine the number of columns of the decision table by examining the lines flowing into the effect nodes of the graph.

- **Step 5: Derive Test Cases**
- Valid username + valid password → expect **Login Success**.
- Valid username + invalid password → expect **Error Message**.
- Invalid username + valid password → expect **Error Message**.
- Both invalid → expect **Error Message**.
- After 3 failed attempts → expect **Account Locked**.

Cause effect graph- Example- 2

- **A water level monitoring system**
  - used by an agency involved in flood control.
  - Input: level(a,b)
    - a is the height of water in dam in meters
    - b is the rainfall in the last 24 hours in cms
- Processing
  - The function calculates whether the level is safe, too high, or too low.
- Output
  - message on screen
    - level=safe
    - level=high
    - invalid syntax

```
def water_level(a, b):
    if a < 0 or b < 0:
        print("invalid syntax")
    elif a < 50 and b < 20:
        print("level = low")
    elif a < 100 and b < 50:
        print("level = safe")
    else:
        print("level = high")
```

Step 1: Identify Causes and Effects

Causes (inputs/conditions):

- **C1**: `a < 0`
- **C2**: `b < 0`
- **C3**: `a < 50`
- **C4**: `b < 20`
- **C5**: `a < 100`
- **C6**: `b < 50`

Effects (outputs):

- **E1**: `"invalid syntax"`
- **E2**: `"level = low"`
- **E3**: `"level = safe"`
- **E4**: `"level = high"`

```
def water_level(a, b):
    if a < 0 or b < 0:
        print("invalid syntax")
    elif a < 50 and b < 20:
        print("level = low")
    elif a < 100 and b < 50:
        print("level = safe")
    else:
        print("level = high")
```

Step 2: Logical Relationships

- **Invalid syntax** → if **C1 OR C2** → **E1**
- **Low level** → if (**C3 AND C4**) and not invalid → **E2**
- **Safe level** → if (**C5 AND C6**) and not low/invalid → **E3**
- **High level** → otherwise → **E4**

To simplify, we can add extra nodes:

- **N1 = (C1 OR C2)** → invalid
- **N2 = (C3 AND C4)** → low
- **N3 = (C5 AND C6)** → safe

Rule	C1 (a<0)	C2 (b<0)	C3 (a<50)	C4 (b<20)	C5 (a<100)	C6 (b<50)	Effect (Output)
1	T	–	–	–	–	–	E1 = Invalid
2	–	T	–	–	–	–	E1 = Invalid
3	F	F	T	T	–	–	E2 = Low
4	F	F	F	F	T	T	E3 = Safe
5	F	F	F	–	F/T	F/T	E4 = High

Step 2: Logical Relationships

- **Invalid syntax** → if **C1 OR C2** → **E1**
- **Low level** → if (**C3 AND C4**) and not invalid → **E2**
- **Safe level** → if (**C5 AND C6**) and not low/invalid → **E3**
- **High level** → otherwise → **E4**

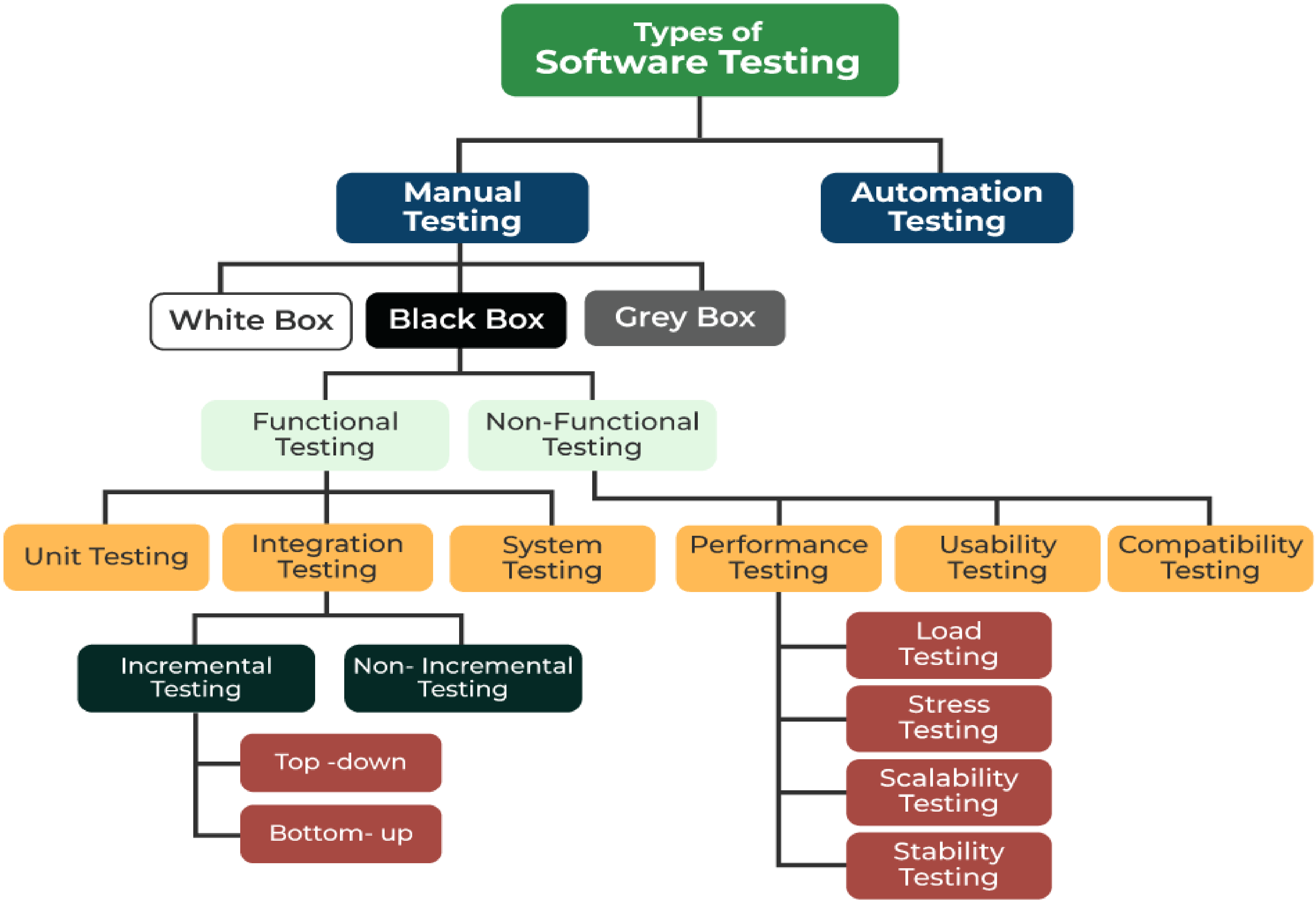
To simplify, we can add extra nodes:

- **N1 = (C1 OR C2)** → invalid
- **N2 = (C3 AND C4)** → low
- **N3 = (C5 AND C6)** → safe

Aspect	Advantages	Disadvantages
Test Case Design	**Systematic derivation of test cases** from requirements	Can **be time-consuming for large systems**
Requirement Coverage	Helps detect missing or inconsistent requirements early	May miss real-world scenarios or unexpected inputs
Complex Logic	Handles multiple input conditions and their interactions effectively	Requires expertise to create and interpret graphs correctly
Test Case Optimization	**Reduces number of test cases by** logically combining conditions	**Not ideal for continuous or highly** variable inputs
Visualization	Provides clear visual representation of cause-effect relationships	Maintenance overhead if requirements change frequently
Communication	Enhances understanding and discussion among stakeholders	-

Cause effect graph

- Not practical for systems which:
  - include timing aspects
  - feedback from processes is used for some other processes.


Types of
Software Testing
Manual
Testing
Automation
Testing
White Box
Black Box
Grey Box
Functional
Testing
Non-Functional
Testing
Unit Testing
Integration
Testing
System
Testing
Performance
Testing
Usability
Testing
Compatibility
Testing
Incremental
Testing
Non- Incremental
Testing
Top -down
Bottom- up
Load
Testing
Stress
Testing
Scalability
Testing
Stability
Testing

Testing- Different levels

- **Unit testing:**
  - test the functionalities of a single module or function.
- **Integration testing:**
  - test the interfaces among the modules.
- **System testing:**
  - test the fully integrated system against its functional and non-functional requirements.

Integration testing

- After different modules of a system have been coded and unit tested:
  - modules are integrated in steps according to an integration plan
  - partially integrated system is tested at each integration step.

Integration Testing

- Develop the integration plan by examining the structure chart :
  - big bang approach
  - top-down approach
  - bottom-up approach
  - mixed approach

Big bang Integration Testing

- Put **all modules together at once** (like Patient, Billing, Pharmacy, etc.) and test the whole system in one go.
- Main problems with this approach:
  - if an error is found:
    - it is very difficult to localize the error
    - the error may potentially belong to any of the modules being integrated.
  - debugging errors found during big bang integration testing are very expensive to fix.

1. Big Bang Approach

•all the modules are simply put together and tested.

• simplest integration testing

• this technique is used only for very small systems.

• Easy to start, but **very hard to debug** because if something breaks, you don't know which module caused it.

•Example: You connect Registration, Appointment, Billing, Lab, etc., all at once → system crashes → **tough to find the faulty part.**

2. Top-Down Approach

Start testing from the **top-level module** (main control/entry point) and then go down step by step.

If lower-level modules are not ready, you use **stubs** (dummy pieces of code).

After the top-level 'skeleton' has been tested:

> immediate subordinate modules of the 'skeleton' are combined with it and tested.

Example: Start with the "Hospital Main System" → test Appointment Scheduling → then test Registration → then Lab.

3. Bottom-Up Approach

•Start testing from the **lowest-level modules** (like database, billing calculator, payment API) and move upwards.

•If top modules are not ready, you use **drivers** (dummy programs to call the lower modules).

•Example: First test Billing calculations, then Pharmacy stock, then combine them into the Hospital System.

Disadvantage in Large Systems

When the system has **many small subsystems**, bottom-up testing can become:

Too Fragmented:

You spend a **lot of time testing small**, isolated parts before seeing how the system works as a whole.

Delayed System-Level Testing:

The **full system behavior** (end-to-end flow) is tested very late.

Example: You only see how “Appointment → Doctor → Billing → Pharmacy” works after many small pieces are already tested separately.

Big Bang-like Effect:

In the extreme case (too many small modules), you end up testing and combining **a large number of components all at once**.

4. Mixed (Sandwich) Approach

- Combine **Top-Down + Bottom-Up** together.
- **Most common**
- **Middle-level modules are tested first, then connect upwards and downwards gradually.**
- Example: Test Patient Registration (middle) → then link it with Appointment (top) and Database (bottom).

Integration Testing

- In top-down approach:
  - testing waits till all top-level modules are coded and unit tested.
- In bottom-up approach:
  - testing can start only after bottom level modules are ready.

Phased versus Incremental Integration Testing

- Integration can be **incremental or phased.**
- In incremental integration testing,
  - only one new module is added to the partial system each time.

Phased versus Incremental Integration Testing

- In phased integration,
  - a group of related modules are added to the partially integrated system each time.
- In **phased testing**, modules are integrated in **big chunks/phases** and tested together.
- System is brought together in **stages**, but each stage may include several modules combined at once.
- More like *"batch testing"*.
- **Example (Hospital System)**:
  - Phase 1: Integrate and test *Patient Registration + Appointment Scheduling*.
  - Phase 2: Add *Doctor Module + EMR*.
  - Phase 3: Add *Billing + Pharmacy + Lab*.

Phased versus Incremental Integration Testing

- Phased integration requires less number of integration steps:
  - compared to the incremental integration approach.
- However, when failures are detected,
  - it is easier to debug if using incremental testing
    - since errors are very likely to be in the newly integrated module.

Incremental Testing

- In **incremental testing**, modules are added **one by one** (or very few at a time), with testing after each addition.
- Easier to isolate errors, since you test after every small integration step.
- **Example (Hospital System)**:
  - Step 1: Test *Patient Registration* alone.
  - Step 2: Add *Appointment Scheduling*, test the pair.
  - Step 3: Add *Doctor Module*, test again.
  - Step 4: Keep adding and testing until the full system is ready.

System Testing

- System tests are designed to validate a fully developed system:
  - to assure that it meets its requirements.
- There are essentially three main kinds of system testing:
  - **Alpha Testing**
  - **Beta Testing**
  - **Acceptance Testing**

Alpha testing

- System testing is carried out
  - by the **test team within the developing organization**.

Alpha testing is an internal testing process carried out by the development team and a small group of internal users before releasing the software to external users.

Where is it done: At the developer's site

Who performs it:

Internal employees, QA team, and sometimes selected users under supervision.

Purpose:

To **catch bugs early, before the product goes public.**

To validate core functionality and usability.

Process:

Performed in a controlled lab environment.Involves both white-box and black-box techniques.

Example:

A company builds a new **banking app**. **Before releasing it, developers and in-house testers use it internally, testing login, transactions, and UI flows. Bugs are reported and fixed before going to external users.**

Beta Testing

⌘Beta testing is the system testing:

- performed by **a select group of friendly customers.**
- Beta testing is **real-world testing** performed by **actual users** of the software **outside the development environment**, after alpha testing is completed.

- **Where done**: At the **customer's location** (real environment).
- **Who performs it**: A limited set of **end-users** who volunteer or are selected.
- **Purpose**:
  - To get feedback on performance, usability, and reliability in **real-world conditions**.
  - To discover issues that developers may have missed.
- **Process**:
  - Distributed to external users (sometimes called **beta release**).
  - Feedback is collected, bugs are fixed, and improvements are made.
- **Example**:

WhatsApp releases a **beta version** of a new feature (like disappearing messages) to selected users worldwide. Based on their feedback, the feature is refined before global release.

Acceptance Testing

- Acceptance testing is the system testing performed by the customer
  - to determine whether he should accept the delivery of the system.
- It is the **final level of software testing**.
- The goal: **Check if the system meets the business requirements and is ready for use by the end users.**
- Done **after system testing** and before delivering the product to the customer.

System Testing

- During system testing, in addition to functional tests:
  - performance tests are performed.

Performance Testing

- Addresses non-functional requirements.
  - May sometimes involve testing hardware and software together.
  - There are several categories of performance testing.

Stress testing/ endurance testing

- Evaluates system performance
  - when stressed for short periods of time.
- Stress tests are black box tests:
  - designed to impose a range of abnormal and even illegal input conditions
  - so as to stress the capabilities of the software.

Stress Testing

- If the requirements is to handle a specified number of users, or devices:
  - stress testing **evaluates system performance when all users or devices are busy simultaneously**.
- If an operating system is supposed to support 15 multiprogrammed jobs,
  - the system is stressed by attempting to run 15 or more jobs simultaneously.
- A real-time system might be tested
  - to determine the effect of simultaneous arrival of several high-priority interrupts.

Stress Testing

- Stress testing usually involves an element of **time or size**,
  - such as the number of records transferred per unit time,
  - the maximum number of users active at any time, input data size, etc.
- Therefore stress testing may not be applicable to many types of systems.

Volume Testing

- Addresses handling large amounts of data in the system:
  - whether data structures (e.g. queues, stacks, arrays, etc.) are large enough to handle all possible situations
  - Fields, records, and files are stressed to check if their size can accommodate all possible data volumes.

Configuration Testing

- Analyze system behavior:
  - in various **hardware and software configurations** specified in the requirements
  - sometimes systems are built in various configurations for different users
  - for instance, a minimal system may serve a single user,
    - other configurations for additional users.

Compatibility Testing

- These tests are needed when the system interfaces with other systems:
  - check whether the interface functions as required.
- If a system is to communicate with a large database system to retrieve information:
  - a compatibility test examines speed and accuracy of retrieval.

Recovery Testing

- These tests check response to:
  - presence of faults or to the loss of data, power, devices, or services
  - subject system to loss of resources
    - check if the system recovers properly.

Maintenance Testing

- Diagnostic tools and procedures:
  - help find source of problems.
  - It may be required to supply
    - memory maps
    - diagnostic programs
    - traces of transactions,
    - circuit diagrams, etc.
- Verify that:
  - all required artifacts for maintenance exist
  - they function properly

Documentation tests

- Check that required documents exist and are consistent:
  - user guides,
  - maintenance guides,
  - technical documents
- Sometimes requirements specify:
  - format and audience of specific documents
  - documents are evaluated for compliance

Usability tests

- All aspects of user interfaces are tested:
  - Display screens
  - messages
  - report formats
  - navigation and selection problems

Environmental test

- These tests check the system's ability to perform at the installation site.
- Requirements might include tolerance for
  - heat
  - humidity
  - chemical presence
  - portability
  - electrical or magnetic fields
  - disruption of power, etc.

Test Summary Report

- Generated towards the end of testing phase.
- Covers each subsystem:
  - a summary of tests which have been applied to the subsystem.

Test Summary Report

- Specifies:
  - how many tests have been applied to a subsystem,
  - how many tests have been successful,
  - how many have been unsuccessful, and the degree to which they have been unsuccessful,
    - e.g. whether a test was an outright failure
    - or whether some expected results of the test were actually observed.

Regression Testing

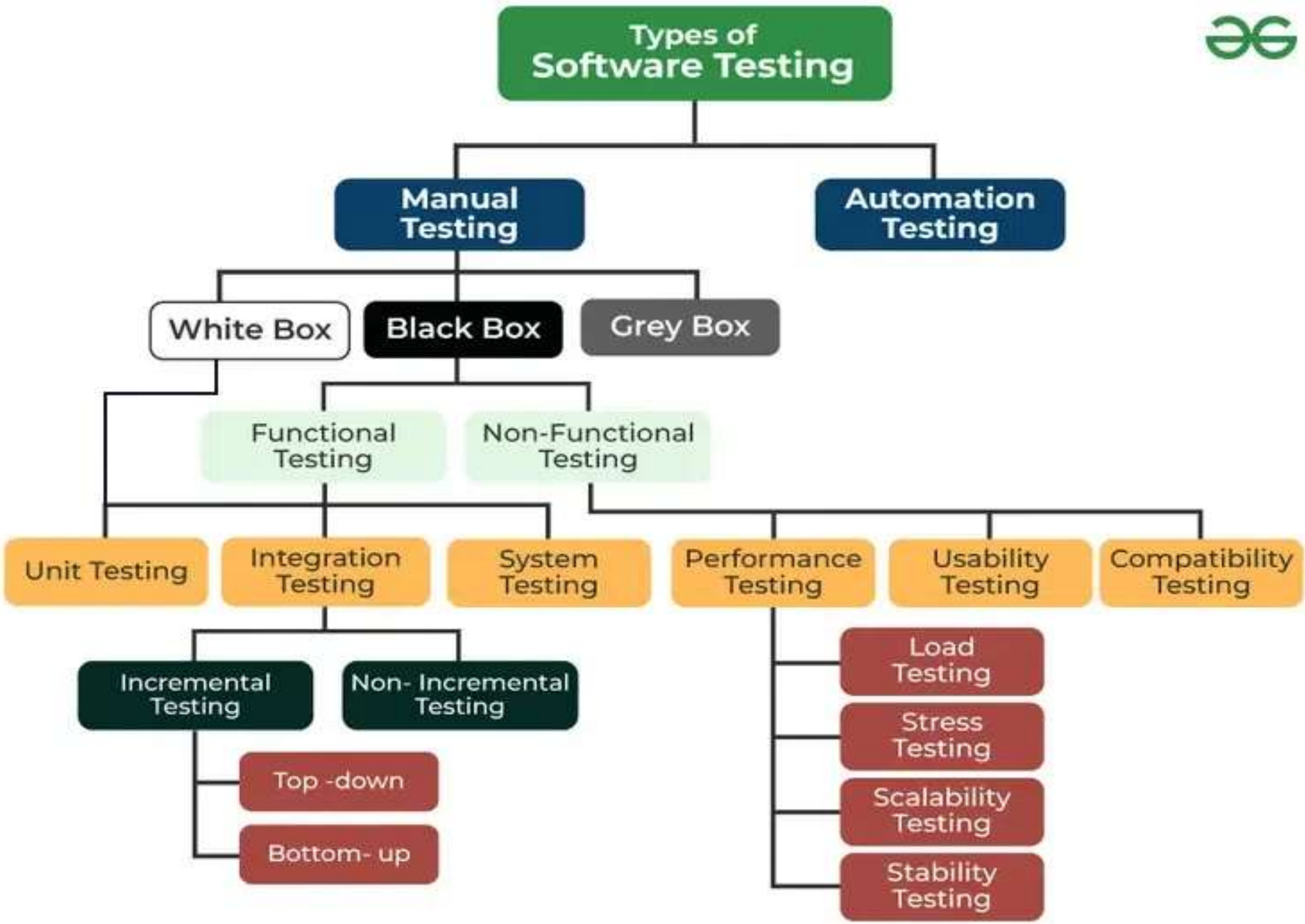
- Does not belong to either unit test, integration test, or system test.
  - Instead, it is a separate dimension to these three forms of testing.

Regression testing

- Regression testing is the running of test suite:
  - after each change to the system or after each bug fix
  - ensures that no new bug has been introduced due to the change or the bug fix.

Regression testing

- Regression tests assure:
  - the new system’s performance is at least as good as the old system
  - always used during phased system development.


Types of
Software Testing
Manual
Testing
Automation
Testing
White Box
Black Box
Grey Box
Functional
Testing
Non-Functional
Testing
Unit Testing
Integration
Testing
System
Testing
Performance
Testing
Usability
Testing
Compatibility
Testing
Incremental
Testing
Non- Incremental
Testing
Top -down
Bottom- up
Load
Testing
Stress
Testing
Scalability
Testing
Stability
Testing

Summary

- We discussed two additional white box testing methodologies:
  - data flow testing
  - mutation testing

Summary

- Data flow testing:
  - derive test cases based on definition and use of data
- Mutation testing:
  - make arbitrary small changes
  - see if the existing test suite detect these
  - if not, augment test suite

Summary

- Cause-effect graphing:
  - can be used to automatically derive test cases from the SRS document.
  - Decision table derived from cause-effect graph
  - each column of the decision table forms a test case

Summary

- Integration testing:
  - Develop integration plan by examining the structure chart:
    - big bang approach
    - top-down approach
    - bottom-up approach
    - mixed approach

Summary: System testing

- Functional test
- Performance test
  - stress
  - volume
  - configuration
  - compatibility
  - maintenance