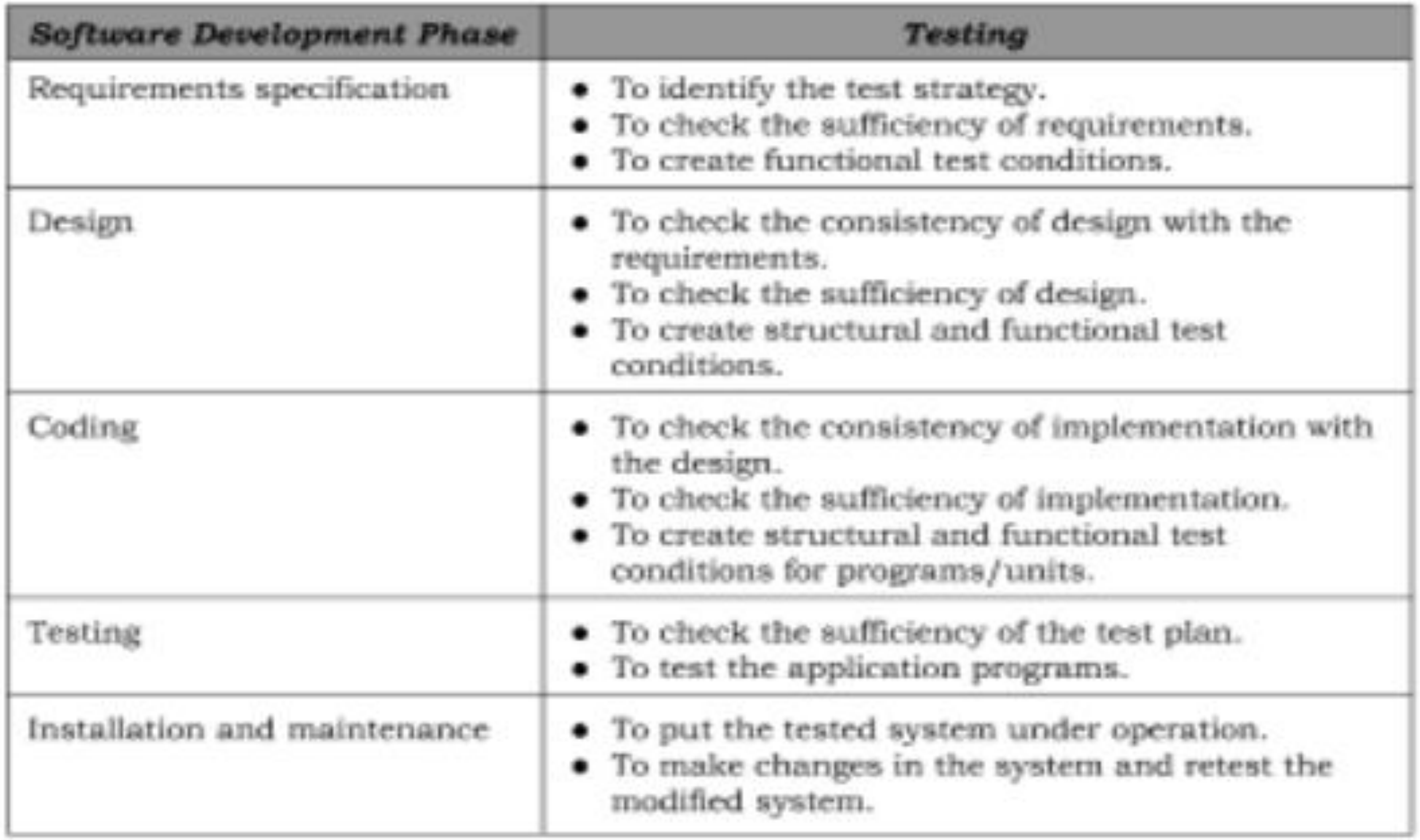
Software testing

Software testing

- Software testing determines the correctness, completeness and quality of software being developed.
- IEEE defines testing as the process of a system by manual or automated means to that it satisfies specified requirements or to identify differences between expected and actual results.
- Software testing is closely related to the terms Verification and Validation
- Verification refers to the process of ensuring that the software is developed according to its specifications. For verification, techniques like reviews, analysis, inspections and walkthroughs are commonly used.
- Validation refers to the process Of checking that the developed software meets the requirements specified by the user.
- Verification and validation can summarized thus as given
  - Verification: Is the software being developed in the right way?
  - Validation: Is the right software being developed?

Role of testing in various phases of SDLC



Software Development Phase	*Testing*
Requirements specification	• To identify the test strategy. • To check the sufficiency of requirements. • To create functional test conditions.
Design	• To check the consistency of design with the requirements. • To check the sufficiency of design. • To create structural and functional test conditions.
Coding	• To check the consistency of implementation with the design. • To check the sufficiency of implementation. • To create structural and functional test conditions for programs/units.
Testing	• To check the sufficiency of the test plan. • To test the application programs.
Installation and maintenance	• To put the tested system under operation. • To make changes in the system and retest the modified system.

Bug, Error , Fault and Failure

- Bug is defined as a logical mistake, which is caused by a software developer while writing the software code.
- Error is defined as the measure of deviation of the outputs given by the software from the outputs expected by the user.
- Fault is defined as the condition that leads to malfunctioning of the software.
- Defect that causes error in operation or negative impact is called failure.Failure is defined as that state Of software under which it is unable to perform functions according to user requirements

Errors in Software – Reasons

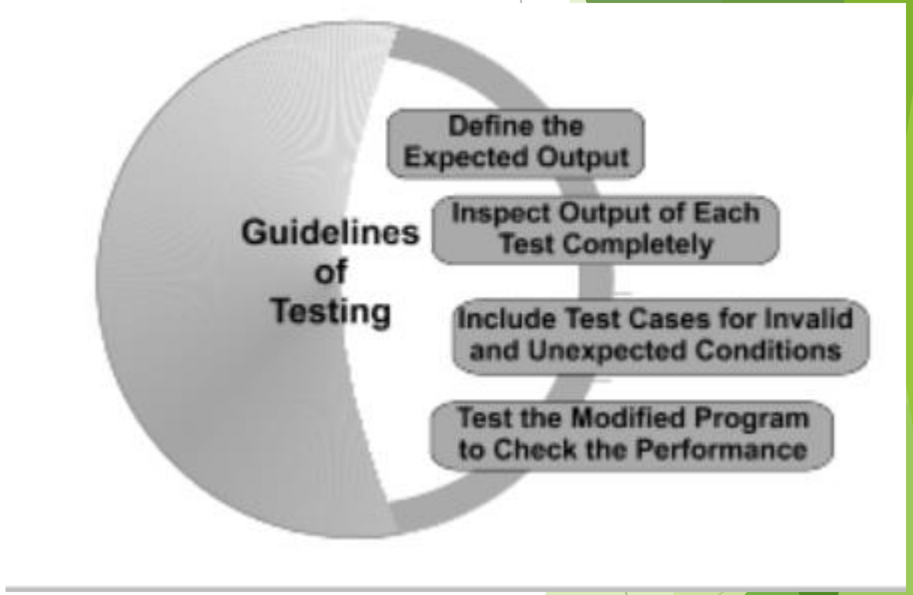
- Programming errors
- Unclear requirements
- Software complexity
- Changing requirements
- Time pressures
- Poorly documented code

Who perform testing?

- Software developers or Independent test group(ITG) consisting of several testers.
- Since Software developers are vested the job of coding, ITG is the best choice for testing
- ITG is usually a part of Software development team

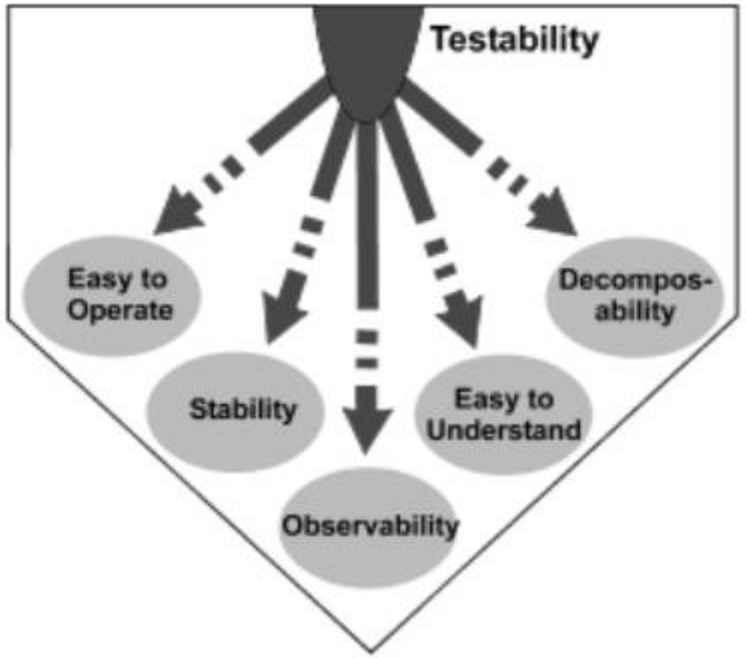
Guidelines for Software Testing

- Define the expected output
- Inspect output of each test completely
- Include test cases for invalid and unexpected conditions
- Test the modified program to check its expected performance


Guidelines of Testing
Define the Expected Output
Inspect Output of Each Test Completely
Include Test Cases for Invalid and Unexpected Conditions
Test the Modified Program to Check the Performance

Testability

- The ease with which a program is tested is known as Testability.
- Testability is required so as to detect errors with minimum efforts.



Characteristics of Testability

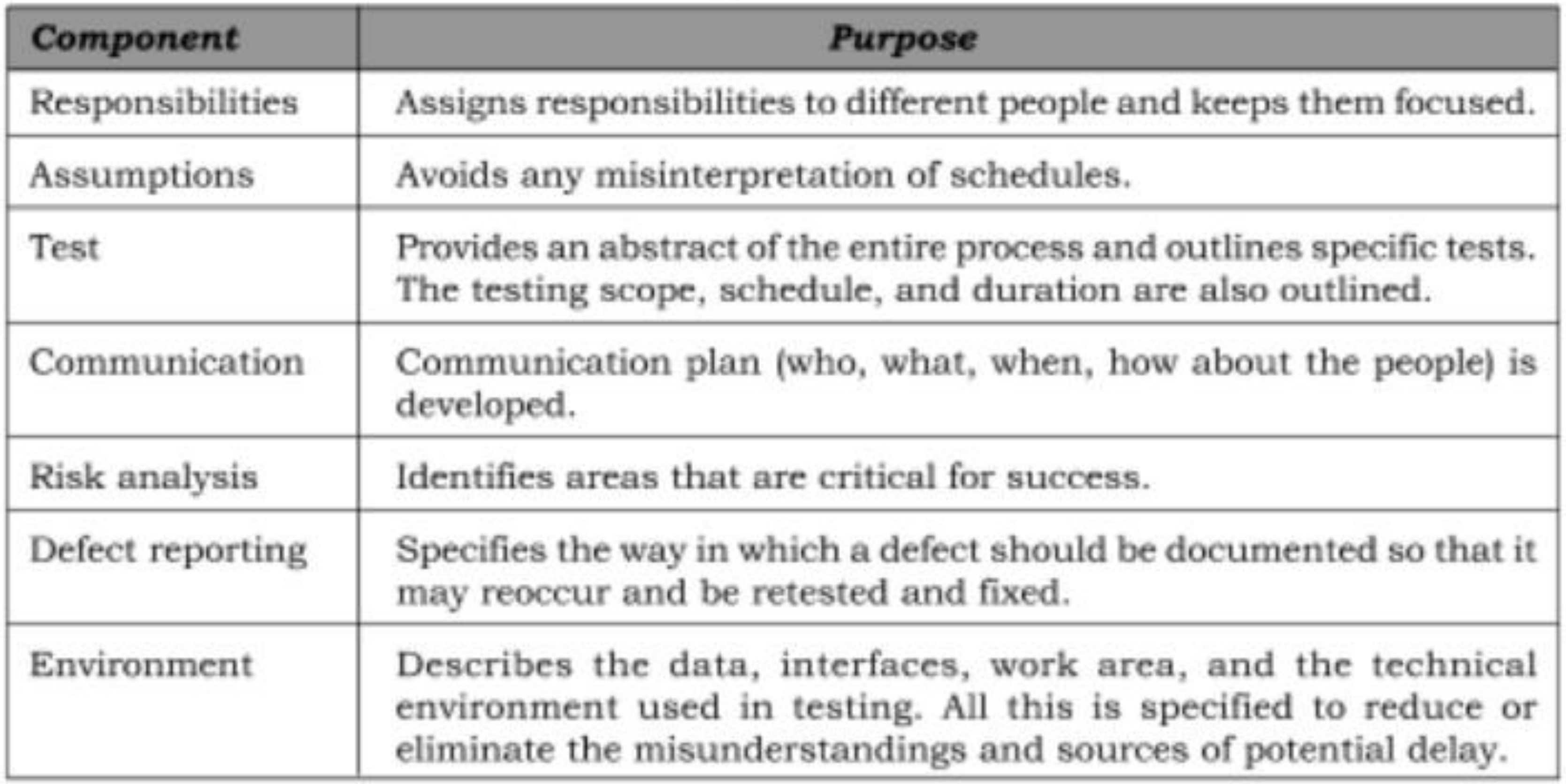
- Easy to operate
  - if the software is designed and implemented considering quality, then comparatively fewer errors will be detected during the execution of tests
- Stability
  - Software becomes stable when changes made to the software are controlled and when the existing tests can still be performed
- Observability
  - Testers can easily identify whether the output generated for certain input is accurate simply by observing it.
- Easy to understand
  - Software that is easy to understand can be tested in an efficient manner.
- Decomposability
  - By breaking software into independent modules, problems can be easily isolated and the modules can be easily tested

Test plan

- A test plan describes how testing would be accomplished.
- It is a document that specifies the purpose, scope, and method of software testing.
- It determines the testing tasks and the persons involved in executing those tasks, test items, and the features to be tested.
- It also describes the environment for testing and the test design and measurement techniques to be used

- A complete test plan helps the people who are not involved in test group to understand why product validation is needed and how it is to be performed.
- However, if the test plan is not complete, it might not be possible to check how the software operates when installed on different operating systems or when used with other software.
- TO avoid this problem, IEEE States Some Components that should be covered in a test plan

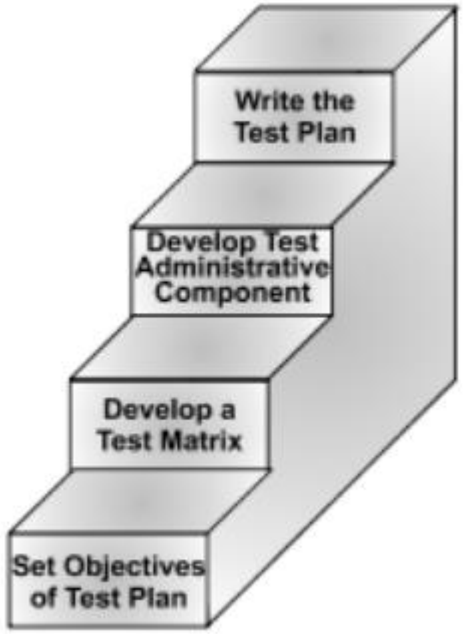
Components to be covered in a Test Plan



Component	*Purpose*
Responsibilities	Assigns responsibilities to different people and keeps them focused.
Assumptions	Avoids any misinterpretation of schedules.
Test	Provides an abstract of the entire process and outlines specific tests. The testing scope, schedule, and duration are also outlined.
Communication	Communication plan (who, what, when, how about the people) is developed.
Risk analysis	Identifies areas that are critical for success.
Defect reporting	Specifies the way in which a defect should be documented so that it may reoccur and be retested and fixed.
Environment	Describes the data, interfaces, work area, and the technical environment used in testing. All this is specified to reduce or eliminate the misunderstandings and sources of potential delay.

Steps in development of a test plan

- Set objectives of test plan
- Develop a test matrix
- Develop test administrative compo
- Write the test plan


Write the Test Plan
Develop Test Administrative Component
Develop a Test Matrix
Set Objectives of Test Plan

Test Case

- A test case provides the description of inputs and their expected outputs to observe whether the software or a part of the software is working correctly.
- IEEE defines test case as 'a set of input values, execution preconditions, expected results and execution post conditions, developed for a particular objective or test condition such as to exercise a particular program path or to verify compliance with a specific requirement
- To avoid error during testing, the test cases must be prepared in such a way that they check the software with all possible inputs. This process is known as exhaustive testing

Test cases – Black box and White box

- Test cases are classified into blackbox tests and whitebox tests, depending on which aspect of the system model is tested.
- **Blackbox tests** focus on the input/output behavior of the component.
- Blackbox tests do not deal with the internal aspects of the component, nor with the behavior or the structure of the components.
- **Whitebox tests** focus on the internal structure of the component.
- A whitebox test makes sure that, independently from the particular input/output behavior, every state in the dynamic model of the object and every interaction among the objects is tested.

- As a result, whitebox testing goes beyond blackbox testing.
- In fact, most of the whitebox tests require input data that could not be derived from a description of the functional requirements alone.
- Unit testing combines both testing techniques: blackbox testing to test the functionality of the component, and whitebox testing to test structural and dynamic aspects of the component.

Test Case generation

- There are two ways of generating test cases. They are
  - Code base test case generation
  - Specification based test case generation

Code based test case generation

- This approach, also known as structure- based test case generation, is used to assess the entire software code to generate test cases.
- It considers only the actual software code to generate test cases and is not concerned with the user requirements.
- Test cases developed using this approach are generally used for performing unit testing.
- These test cases can easily test statements, branches, special values, and symbols present in the unit being tested.

Specification based test case generation

- This approach uses specifications, which indicate the functions that are produced by the software to generate test cases.
- In other words, it considers only the external view of the software to generate test cases.
- It is generally used for integration testing and system testing to ensure that the software is performing the required task.
- Since this approach considers only the external view of the software, it does not test the design decisions and may not cover all statements of a program.
- Moreover, as test cases are derived from specifications, the errors present in these specifications may remain uncovered.

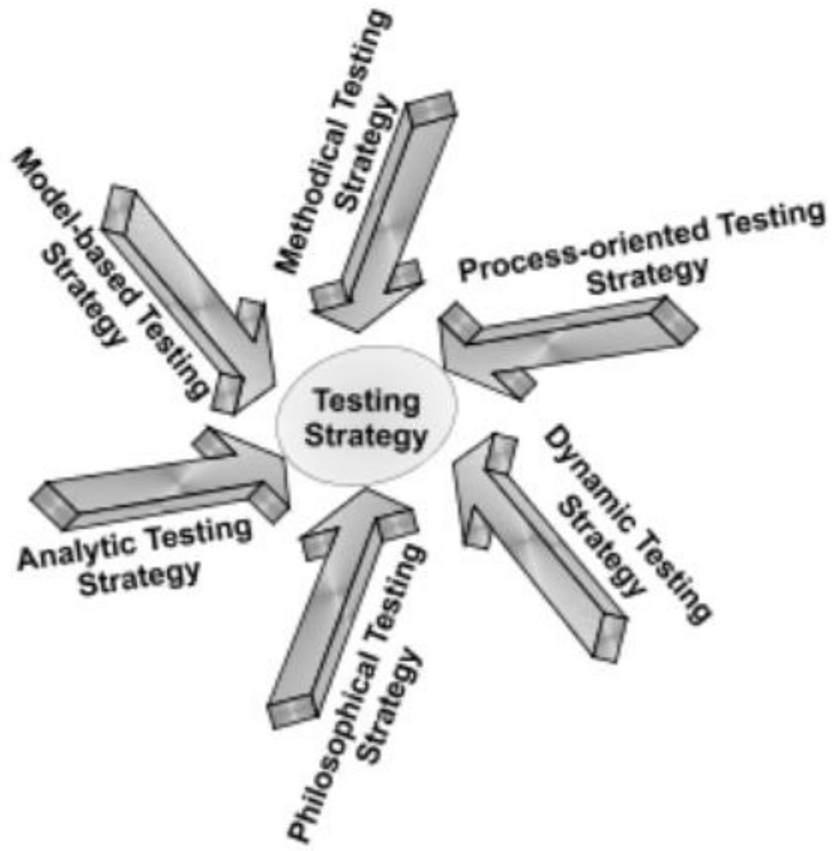
Software testing strategies

- A testing strategy is used to identify the levels of testing which are to be applied along with the methods, techniques, and tools to be used during testing.
- This strategy also decides test cases, test specifications, test case decisions, and puts them together for execution.

Characteristics of Testing strategies

- Testing proceeds in an outward manner. It starts from testing the individual units, progresses to integrating these units, and finally, moves to system testing.
- Testing techniques used during different phases of software development are different.
- Testing is conducted by the software developer and by an ITG.
- Testing and debugging should not be used synonymously. However, any testing strategy must accommodate debugging with itself.

Types of Testing strategies


Testing Strategy
Methodical Testing Strategy
Process-oriented Testing Strategy
Dynamic Testing Strategy
Philosophical Testing Strategy
Analytic Testing Strategy
Model-based Testing Strategy

Analytic testing strategy

- This uses formal and informal techniques to access and prioritize risks that arise during software testing.
- It takes a complete overview of requirements, design, and implementation of objects to determine the motive of testing.
- In addition, it gathers complete information about the software, targets to be achieved, and the data required for testing the software

Model-based testing strategy

- This strategy tests the functionality of the software according to the real world scenario (like software functioning in an organization).
- It recognizes the domain of data and selects suitable test cases according to the probability of errors in that domain.

Methodical testing strategy

- It tests the functions and status of software according to the checklist, which is based on user requirements.
- This strategy is also used to test the functionality, reliability, usability, and performance of the software.

Process-oriented testing strategy

- It tests the software according to already existing standards such as the IEEE standards.
- In addition, it checks the functionality Of the software by using automated testing tools.

Dynamic testing strategy

- This tests the software after having a collective decision of the testing team.
- Along with testing, this strategy provides information about the software such as test cases used for testing the errors present in it.

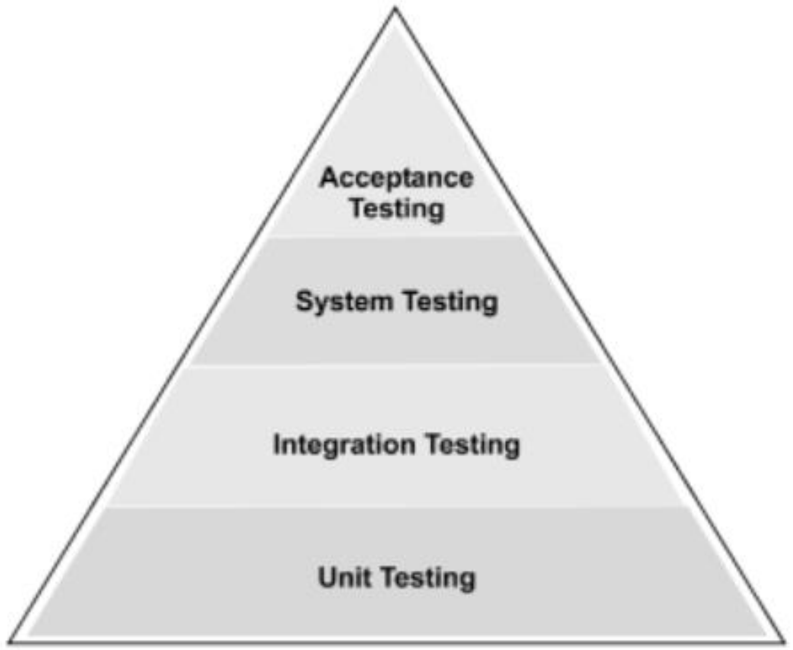
Philosophical testing strategy

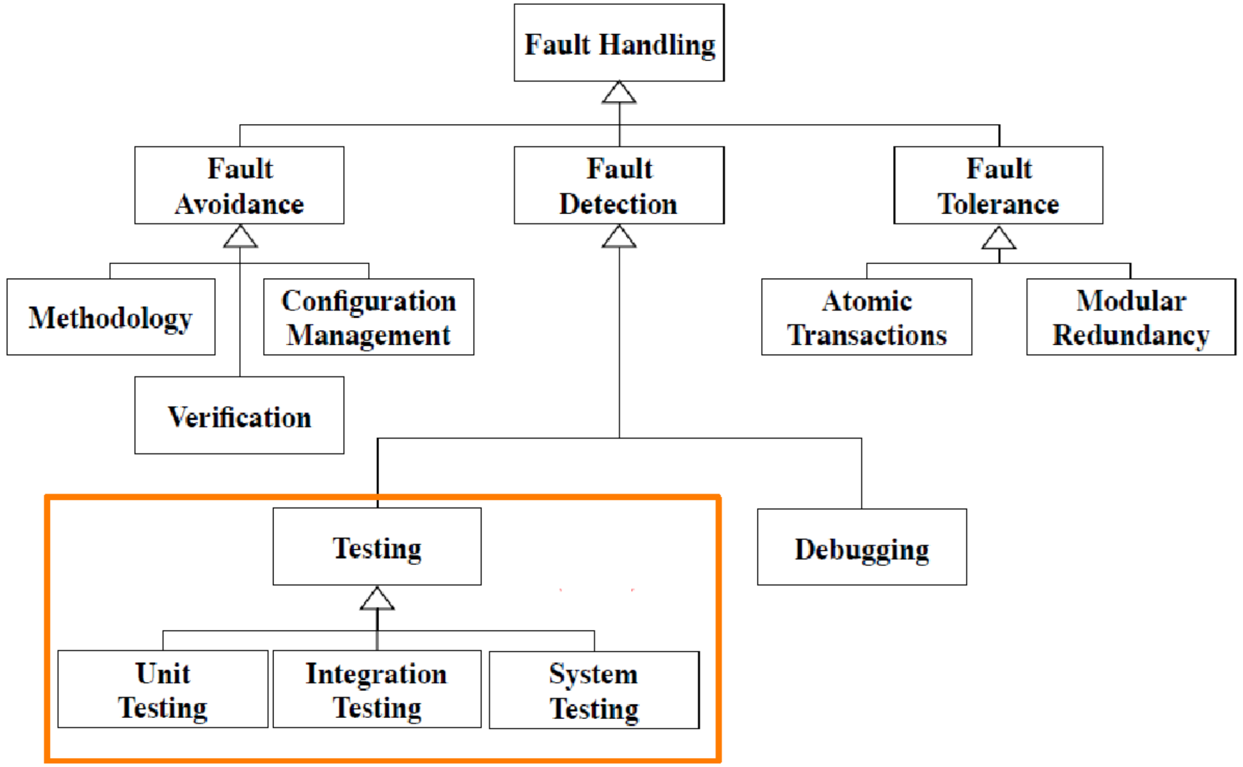
- It tests the software assuming that any component of the software can stop functioning anytime.
- It takes help from software developers, users and systems analysts to test the software.

Techniques for increasing the reliability of a software system

- Fault avoidance
  - Fault avoidance tries to prevent the insertion of faults into the system before it is released.
  - Fault avoidance includes development methodologies, configuration management, and verification.
- Fault detection
  - Fault detection techniques assist in finding faults in systems, but do not try to recover from the failures caused by them.
  - In general, fault detection techniques are applied during development, but in some cases they are also used after the release of the system
- Fault tolerance
  - Fault tolerance assume that a system can be released with faults and that system failures can be dealt with by recovering from them at runtime

Levels of Software testing




Fault Handling
Fault Avoidance
Fault Detection
Fault Tolerance
Methodology
Configuration Management
Verification
Atomic Transactions
Modular Redundancy
Testing
Debugging
Unit Testing
Integration Testing
System Testing

Unit testing

- **Unit testing** focuses on the building blocks of the software system, that is, objects and subsystems.
- There are three motivations behind focusing on these building blocks.
  - First, unit testing reduces the complexity of overall test activities, allowing us to focus on smaller units of the system.
  - Second, unit testing makes it easier to pinpoint and correct faults, given that few components are involved in the test.
  - Third, unit testing allows parallelism in the testing activities; that is, each component can be tested independently of the others.

Unit testing techniques

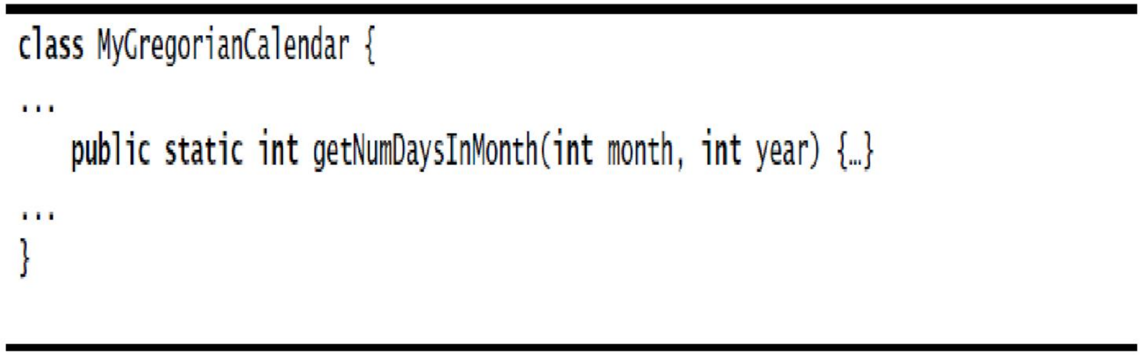
- equivalence testing
- boundary testing
- path testing
- state-based testing.

Equivalence testing

- This blackbox testing technique minimizes the number of test cases.
- The possible inputs are partitioned into equivalence classes, and a test case is selected for each class.
- The assumption of equivalence testing is that systems usually behave in similar ways for all members of a class.
- To test the behavior associated with an equivalence class, we only need to test one member of the class.
- Equivalence testing consists of two steps: identification of the equivalence classes and selection of the test inputs

Criteria used in determining the equivalence classes

- *Coverage*. Every possible input belongs to one of the equivalence classes
- *Disjointedness*. No input belongs to more than one equivalence class.
- *Representation*. If the execution demonstrates an erroneous state when a particular member of a equivalence class is used as input, then the same erroneous state can be detected by using any other member of the class as input

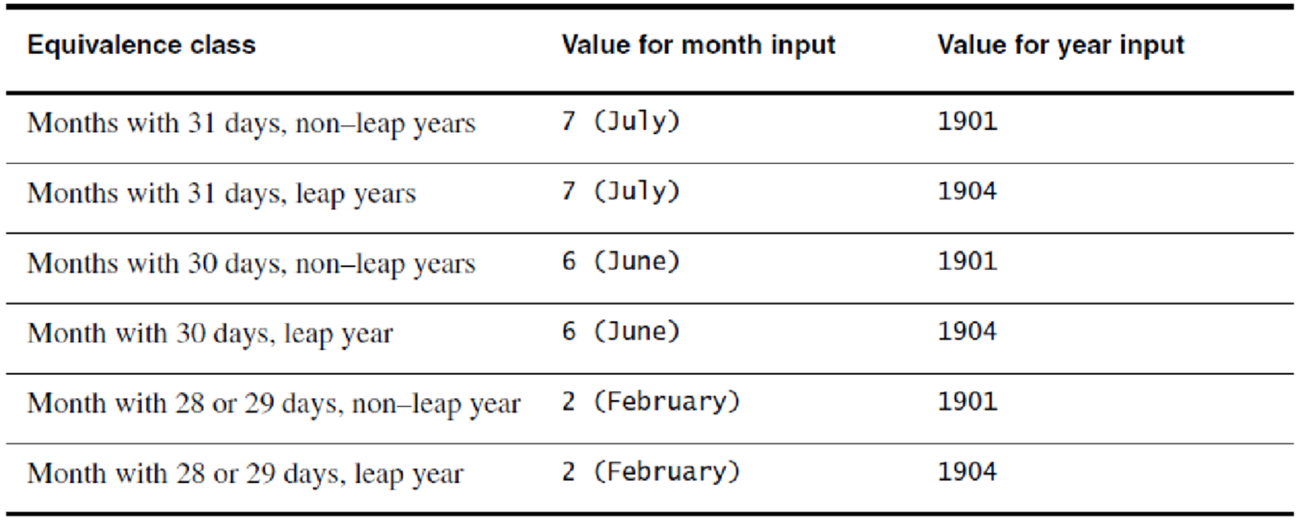


```
class MyGregorianCalendar {
...
    public static int getNumDaysInMonth(int month, int year) {…}
...
}
```

For example, consider a method that returns the number of days in a month, given the month and year . The month and year are specified as integers. By convention, 1 represents the month of January, 2 the month of February, and so on. The range of valid inputs for the year is 0 to maxInt.

- We find three equivalence classes for the month parameter:
  - months with 31 days (i.e., 1, 3, 5, 7, 8, 10, 12),
  - months with 30 days (i.e., 4, 6, 9, 11), and
  - February, which can have 28 or 29 days.
  - Nonpositive integers and integers larger than 12 are invalid values for the month parameter.
- Similarly, we find two equivalence classes for the year:
  - leap years and non–leap years.
  - By specification, negative integers are invalid values for the year.

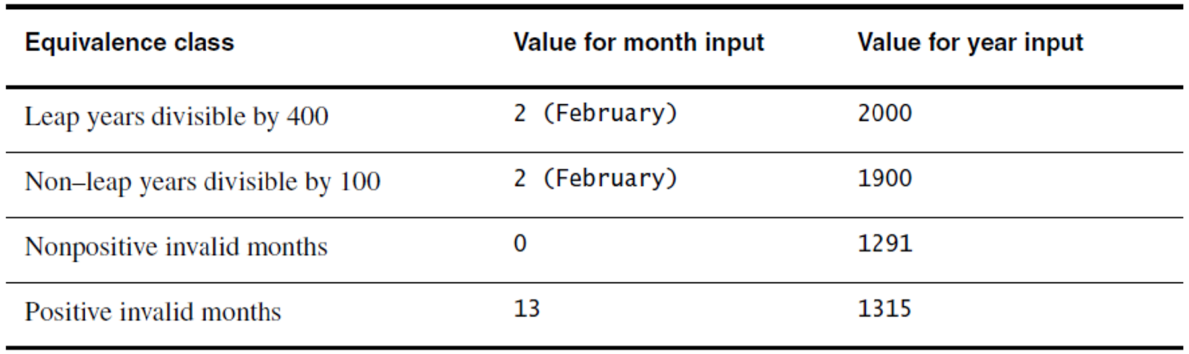
Equivalence classes and selected valid inputs for testing the getNumDaysInMonth() method



Equivalence class	Value for month input	Value for year input
Months with 31 days, non–leap years	`7 (July)`	1901
Months with 31 days, leap years	`7 (July)`	1904
Months with 30 days, non–leap years	`6 (June)`	1901
Month with 30 days, leap year	`6 (June)`	1904
Month with 28 or 29 days, non–leap year	`2 (February)`	1901
Month with 28 or 29 days, leap year	`2 (February)`	1904

Boundary testing

- This special case of equivalence testing focuses on the conditions at the boundary of the equivalence classes.
- Rather than selecting any element in the equivalence class, boundary testing requires that the elements be selected from the “edges” of the equivalence class.



Equivalence class	Value for month input	Value for year input
Leap years divisible by 400	2 (February)	2000
Non–leap years divisible by 100	2 (February)	1900
Nonpositive invalid months	0	1291
Positive invalid months	13	1315

Path testing

- This whitebox testing technique identifies faults in the implementation of the component.
- The assumption behind path testing is that, by exercising all possible paths through the code at least once, most faults will trigger failures.
- The identification of paths requires knowledge of the source code and data structures.
- The starting point for path testing is the flow graph.
- A flow graph consists of nodes representing executable blocks and edges representing flow of control

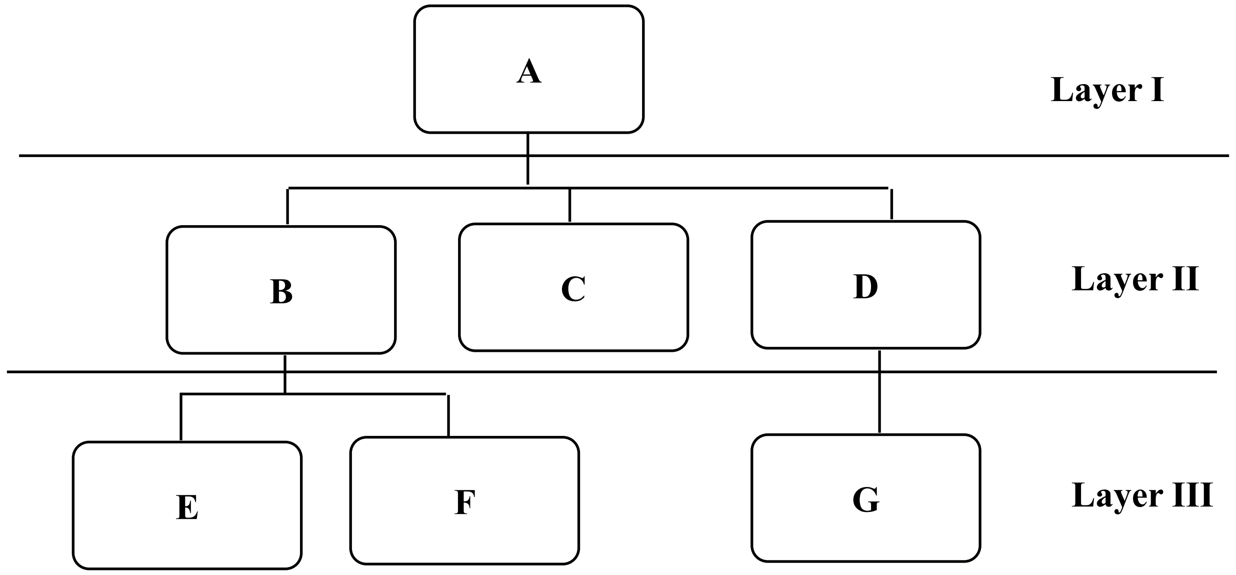
State-based testing

- This testing technique was recently developed for object-oriented systems
- State-based testing, compares the resulting state of the system with the expected state.
- In the context of a class, state-based testing consists of deriving test cases from the UML state machine diagram for the class.
- For each state, a representative set of stimuli is derived for each transition (similar to equivalence testing).
- The attributes of the class are then instrumented and tested after each stimuli has been applied to ensure that the class has reached the specified state.

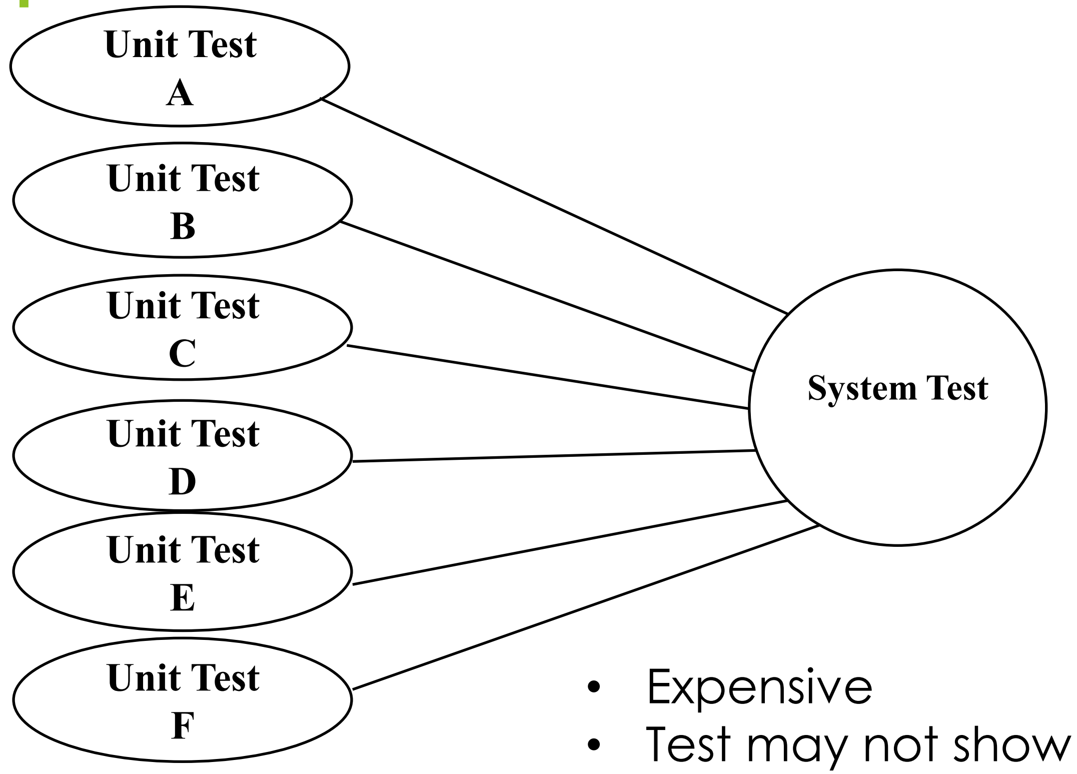
Integration Testing Strategy

- The entire system is viewed as a collection of subsystems (sets of classes) determined during the system and object design.
- The order in which the subsystems are selected for testing and integration determines the testing strategy
  - Big bang integration (Nonincremental)
  - Bottom up integration
  - Top down integration
  - Sandwich testing
  - Variations of the above
- For the selection use the system decomposition from the System Design

Example: Three Layer Call Hierarchy

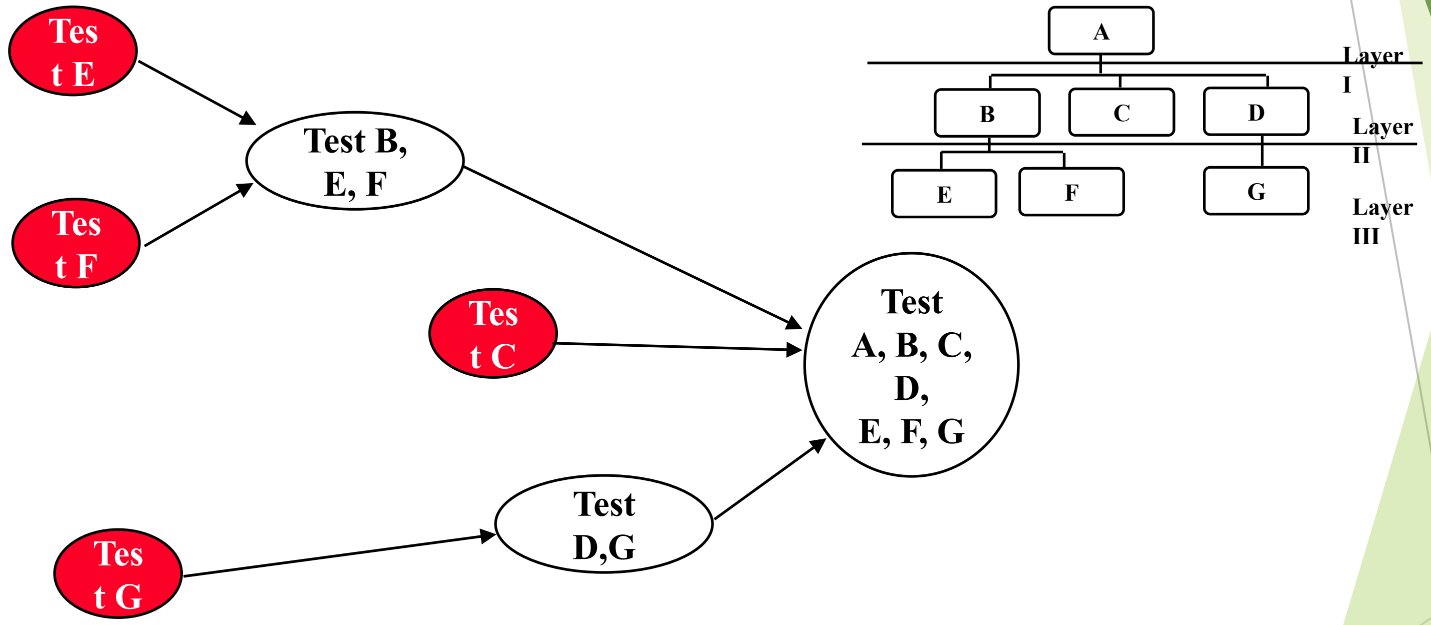


Integration Testing: Big-Bang Approach



- Expensive
- Test may not show failures in interface
- Cant show specific components of failure

Bottom-up Integration

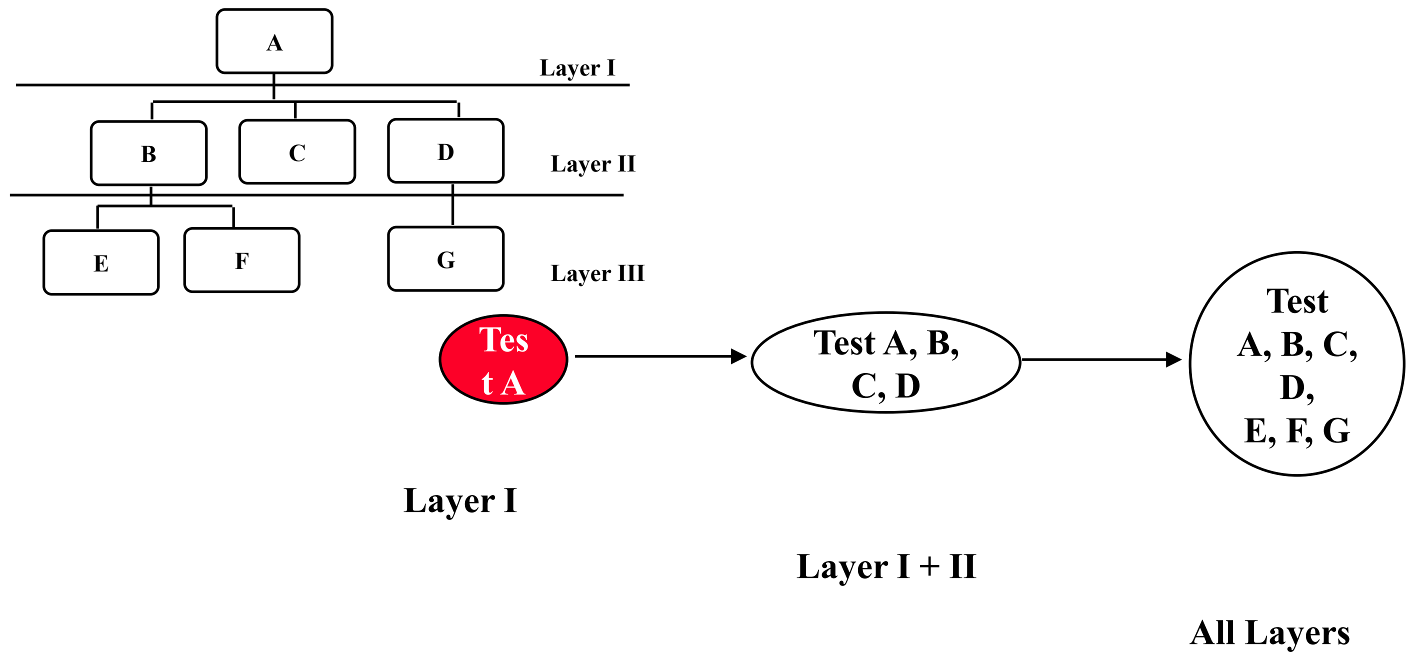


- Double test, triple test, quadruple test

Pros and Cons of bottom up integration testing

- Bad for functionally decomposed systems:
  - Tests the most important subsystem (UI) last
- Useful for integrating the following systems
  - Object-oriented systems
  - real-time systems
  - systems with strict performance requirements

Top-down Integration Testing



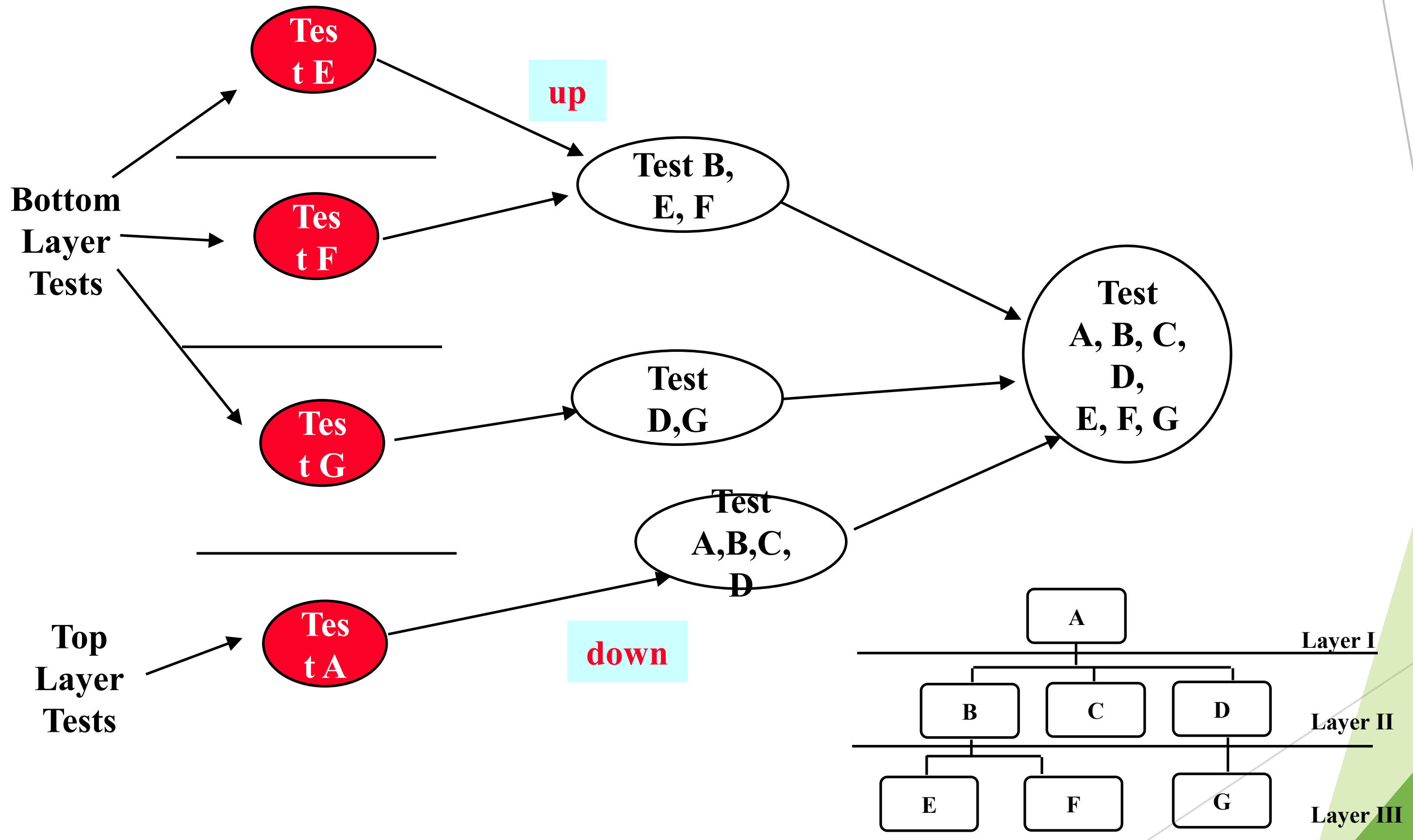
Pros and Cons of top-down integration testing

- Test cases can be defined in terms of the functionality of the system (functional requirements)
- *Writing stubs can be difficult*: Stubs must allow all possible conditions to be tested.
- Possibly a very large number of stubs may be required, especially if the lowest level of the system contains many methods.
- One solution to avoid too many stubs: *Modified top-down testing strategy*
  - Test each layer of the system decomposition individually before merging the layers
  - Disadvantage of modified top-down testing: Both, stubs and drivers are needed

Sandwich Testing Strategy

- Combines top-down strategy with bottom-up strategy
- *The system is viewed as having three layers*
  - A target layer in the middle
  - A layer above the target
  - A layer below the target
  - Testing converges at the target layer

Sandwich Testing Strategy



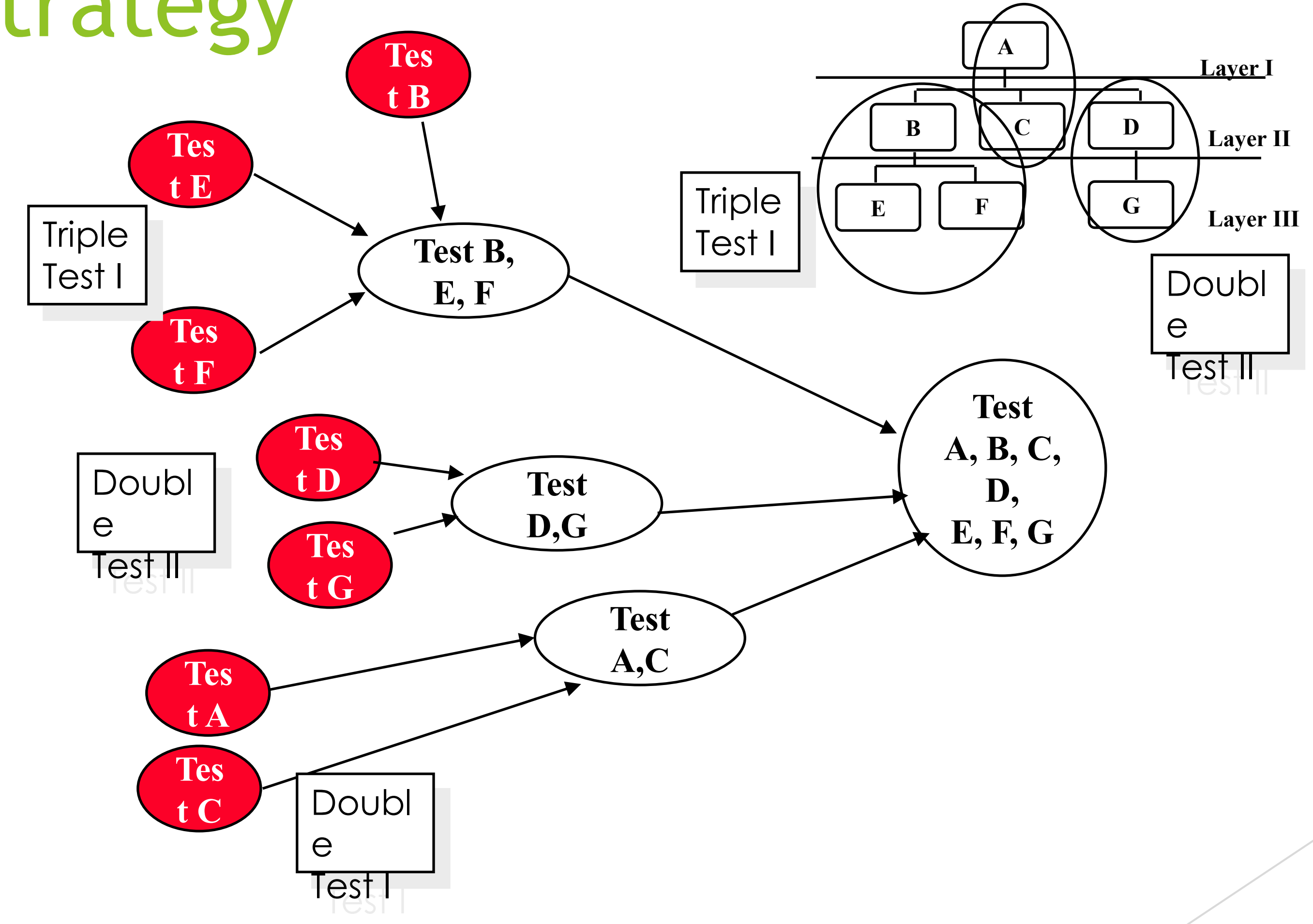
Pros and Cons of Sandwich Testing

- Top and Bottom Layer Tests can be done in ***parallel***
- Does not test the individual subsystems thoroughly before integration

Modified Sandwich Testing Strategy

- Test in parallel:
  - Middle layer with *drivers and stubs*
  - Top layer with stubs
  - Bottom layer with drivers
- Test in parallel:
  - Top layer accessing middle layer (top layer replaces drivers)
  - Bottom accessed by middle layer (bottom layer replaces stubs)

Modified Sandwich Testing Strategy



Steps in Integration-Testing

1. Based on the integration strategy, *select a component* to be tested. Unit test all the ***classes*** in the component.

2. Put selected ***component*** together; do any *preliminary fix-up* necessary to make the integration test operational (drivers, stubs)

3. Do *functional testing*: Define test cases that exercise all ***uses cases*** with the selected component

4. Do *structural testing:* Define test cases that exercise the selected ***component***

5. Execute *performance tests*

6. *Keep records* of the test cases and testing activities.

7. Repeat steps 1 to 7 until the full system is tested.

The primary *goal of integration testing is to identify errors* in the (current) component configuration.

Which Integration Strategy should you use?

- Factors to consider
  - Amount of test harness (stubs &drivers)
  - Location of critical parts in the system
  - Availability of hardware
  - Availability of components
  - Scheduling concerns
- Bottom up approach
  - good for object oriented design methodologies
  - Test driver interfaces must match component interfaces
  - ...
  - ...Top-level components are usually important and cannot be neglected up to the end of testing
  - Detection of design errors postponed until end of testing
- Top down approach
  - Test cases can be defined in terms of functions examined
  - Need to maintain correctness of test stubs
  - Writing stubs can be difficult

System Testing

- Functional Testing
- Structure Testing
- Performance Testing
- Acceptance Testing
- Installation Testing

Impact of requirements on system testing:

- The more explicit the *requirements*, the easier they are to test.
- Quality of *use cases* determines the ease of functional testing
- Quality of subsystem decomposition determines the ease of structure testing
- Quality of *nonfunctional requirements and constraints* determines the ease of performance tests:

Structure Testing

- *Essentially the same as white box testing.*
- Goal: Cover all paths in the system design
  - Exercise all input and output parameters of each component.
  - Exercise all components and all calls (each component is called at least once and every component is called by all possible callers.)
  - Use conditional and iteration testing as in unit testing.

Functional Testing

.

Essentially the same as black box testing

- Goal: Test functionality of system
- **Test cases are designed from the requirements analysis document (better: user manual) and centered around requirements and key functions (*use cases*)**

- The system is treated as black box.
- Unit test cases can be reused, but in end user oriented new test cases have to be developed as well.

Performance Testing

- Stress Testing
  - Stress limits of system (maximum # of users, peak demands, extended operation)
- Volume testing
  - Test what happens if large amounts of data are handled
- Configuration testing
  - Test the various software and hardware configurations
- Compatibility test
  - Test backward compatibility with existing systems
- Security testing
  - Try to violate security requirements
- Timing testing
  - Evaluate response times and time to perform a function
- Environmental test
  - Test tolerances for heat, humidity, motion, portability
- Quality testing
  - Test reliability, maintain- ability & availability of the system
- Recovery testing
  - Tests system’s response to presence of errors or loss of data.
- Human factors testing
  - Tests user interface with user

Acceptance Testing

- Goal: Demonstrate system is ready for operational use
    - Choice of tests is made by *client*/sponsor
    - Many tests can be taken from integration testing
    - Acceptance test is performed by the client, not by the developer.
- Majority of all bugs in software is typically found by the client after the system is in use, not by the developers or testers. Therefore two kinds of additional tests:

- *Alpha* *test:*
    - Sponsor uses the software at the *developer's site*.
    - Software used in a controlled setting, with the developer always ready to fix bugs.
- *Beta* *test:*
    - Conducted at *sponsor's site* (developer is not present)
    - Software gets a realistic workout in target environ-ment
    - Potential customer might get discouraged

Testing has its own Life Cycle

Establish the test objectives

Design the test cases

Write the test cases

Test the test cases

Execute the tests

Evaluate the test results

Change the system

Do regression testing