Information System Design

IT60105

Lecture 25

Information System Testing

Lecture #25

- Software testing strategies
  - Unit testing
  - Integration testing
  - System testing
  - Regression testing

Preliminaries on Testing

- Testing objectives
  - The primary objective is to identify all defects existing in a software product
  - Testing requirements
    - Sequence of testing that is necessary to follow to adequately test a system
  - Test case design
    - A test case is the triplet [*I*, *S*, *O*], where *I* is the data input to the system, *S*, is the state of the system at which the data is input, and *O* is the expected output of the system
    - A good test case is one that has a high probability to uncover an error

Software Testing Strategy

- Testing objectives
  - Test oracle
    - Is a source of expected results for a test case
  - Test suit design
    - A test suit is the set of all test cases with which a given system is to be tested
    - A good test suit is a test suit with minimum number of test cases and successful to uncovered errors, if any
  - Test driver
    - Is a program which testing a system given a test suit as an input

Taxonomy of Testing Strategies

- Testing strategies
  - **Unit testing**
    - Start testing at the individual component (unit level)
  - **Integration testing**
    - The pre-tested individual components are slowly integrated and tested at each level of integration (integration level)
  - **System testing**
    - Final level testing to test the fully integrated system (system level)

Taxonomy of Testing Strategies

- **Unit testing**
  - Testing without execution
    - Code walkthrough
    - Code inspection
  - Testing with execution
    - Black-box testing
      - Exhaustive testing
      - Equivalence partition
      - Boundary-value analysis
      - Comparison testing
    - White-box testing

Taxonomy of Testing Strategies

- **Unit testing** (Contd..)
  - Testing with execution
    - Black-box testing
    - White-box testing
      - Control structure testing
        - » Statement coverage
        - » Condition coverage
        - » Branch coverage
        - » Data-flow testing
        - » Loop testing
      - Basis path testing
      - Mutation testing

Taxonomy of Testing Strategies

- **Unit testing**

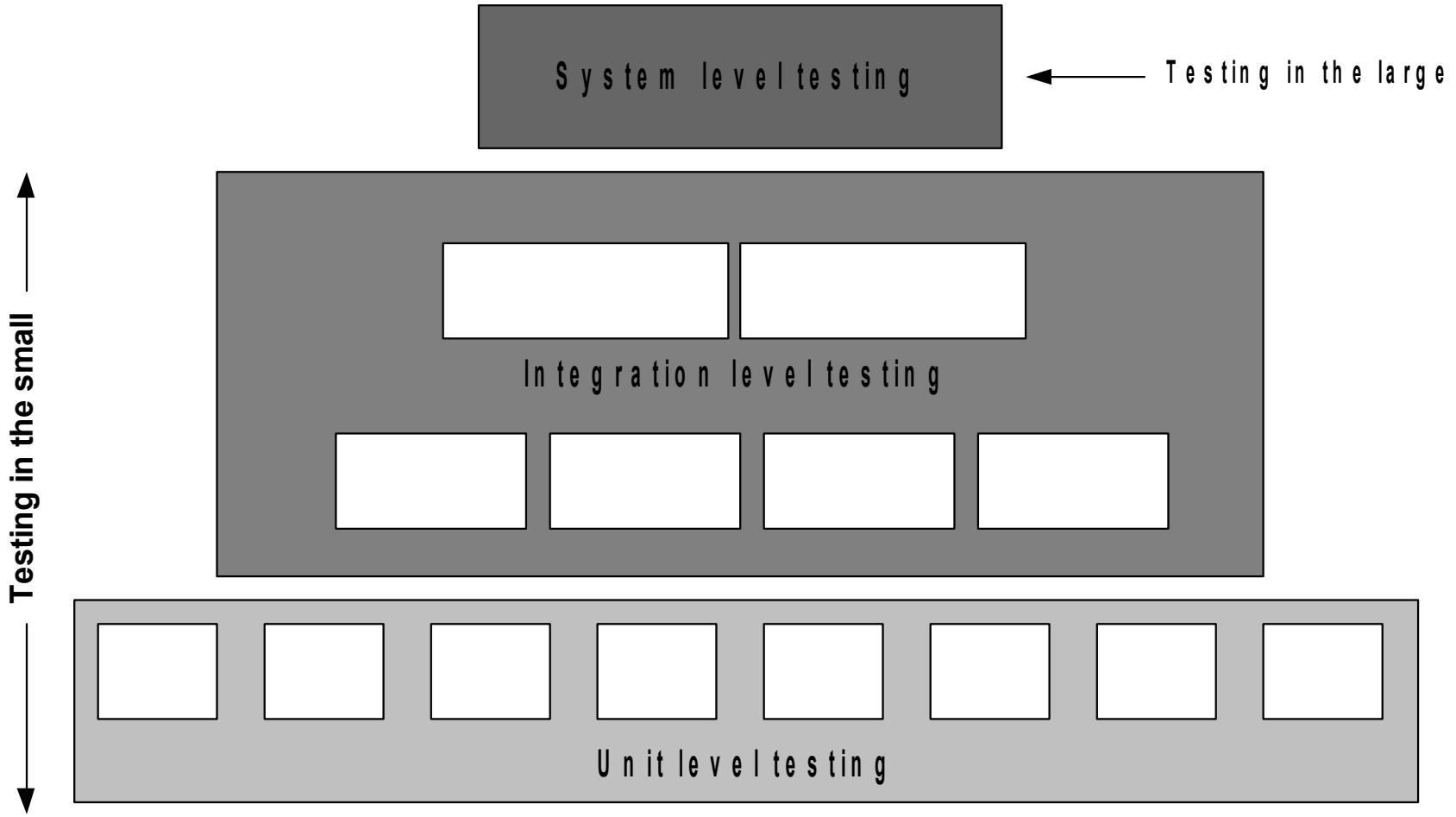
- **Integration testing**
  - Big-bang testing
  - Top-down testing
  - Bottom-up testing
  - Mixed integration testing
  - Smoke testing

- **System testing**

Taxonomy of Testing Strategies

- **Unit testing**
- **Integration testing**
- **System testing**
  - **Acceptance testing**
    - **Alpha testing**
    - **Beta testing**
  - **Performance testing**
    - **Stress, Volume, Compatibility, Recovery, Security etc.**

Software Testing Strategy



Unit Testing Strategies

Unit Testing Strategies

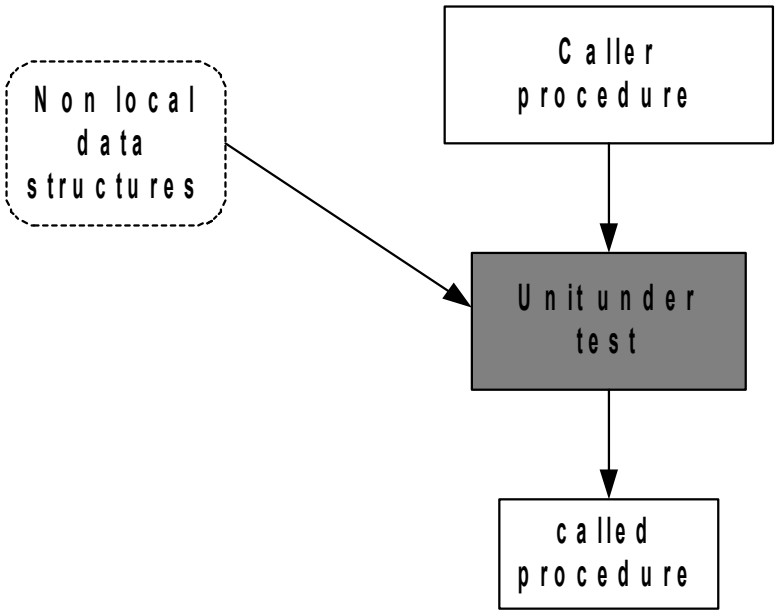
- Unit testing (or module testing) is the testing of different components in isolation
- Testing without execution
  - Code **walk-through**
    - Code is given to the testing team. Each team members selects some test cases and simulates execution of the code by hand
    - Discover any algorithmic or logical error is there in the module
    - Focuses on discovery of errors and not on how to fix the discovered errors
    - Informal meeting for debugging
  - Code **inspection**
    - To discover or fix any common types of errors caused due to oversight and improper programming
    - Also identifies coding standard

Unit Testing Strategies

- Testing with execution
  - **Black-box testing**
    - Test cases are designed from an examination of the input/output values only
    - Tests are based on requirements and functionality
    - No knowledge of design or code is required
  - **White-box testing**
    - Based on knowledge of the internal logic of an application's code
    - Tests are based on coverage of code statements, branches, paths, conditions etc.

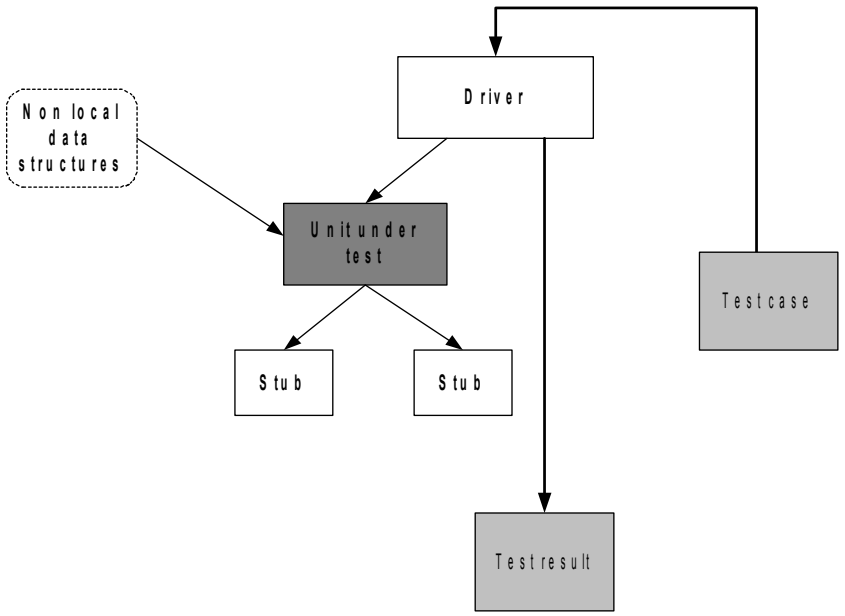
Requirement of Unit Testing

- Following are necessary to accomplish the unit testing
  - The procedures belonging to other modules that the unit under tests calls
  - Non local data structures that the module accesses
  - A procedure to call the functions of the module under test with appropriate parameters



Requirement of Unit Testing

- Caller module or called module may not be available during the test of the unit under test
- *Driver* – dummy for caller procedure
- *Stub* – dummy for called procedure
- Driver and stub are to simulate the behavior of actual caller and called procedures



Black-box Testing

- Black-box testing also called *behavioral testing*, focuses on the functional requirements of the unit under test
- Black-box testing attempts to find errors in the following categories
  - Incorrect or missing functions
  - Interface errors
  - Errors in data structures or external data access
  - Behavior or performance errors
  - Initialization or termination errors

Black-box Testing Strategies

- Following are few important black-box testing techniques
  - Exhaustive testing
  - Equivalence partitioning
  - Boundary value analysis
  - Comparison testing

Exhaustive Testing

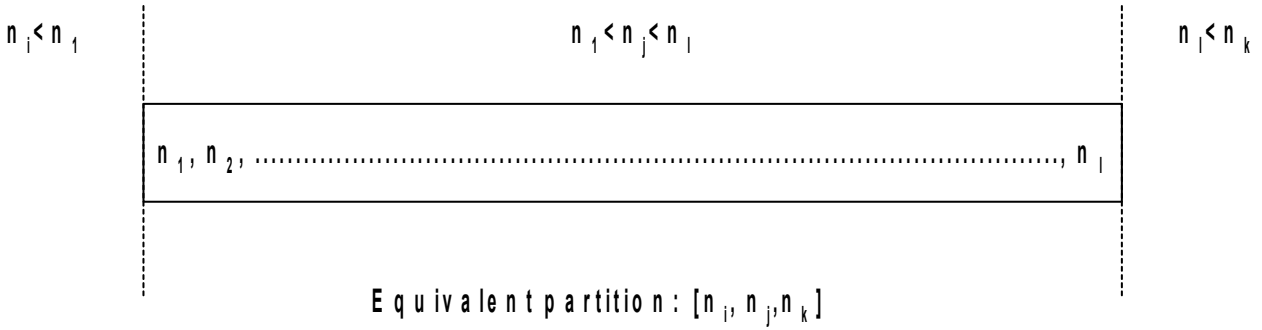
- For a set of all possible input, test case is derived as a permutation of all input
- Brain-less (or brute force) testing
- Suitable for units where number of input as well as their domain is less
- As the number of input values grows and number of discrete values for each data item increases, this testing strategy is infeasible

Equivalence Partitioning

- Equivalent partitioning method divides the input domain of a program into classes of data from which test cases can be derived
- The partitioning is done in such a way that the behavior of the program is similar to every input data belonging to the same equivalence classes

Example:

Insertion of an element into an array of sorted elements



Boundary Value Analysis

- Because of human psychological factor a number of error tends to occur at the boundaries of the input domain rather at the “center”
- BVA leads to a selection of test cases that exercise bounding values
- BVA is a test case design technique that complements equivalent partitioning
- Rather than selecting an element of an equivalent partition, BVA leads to the selection of test cases at the “edges” of the partition
- Rather than focusing solely on input conditions, BVA derives the test cases from the output domain as well

Boundary Value Analysis: Example

```
1.  BinarySearch(low, high)
2.  {
3.      mid = (low + high) / 2;
4.      if (A[mid] = key )
5.          return (mid);
6.      if (key < A[mid])
7.          high = mid – 1;
8.      else
9.                  low = mid + 1;
10.     BinarySearch (low, high);
11.   }
```

Output domain: Successful or Failure (lower / upper ends)

Comparison Testing

- Several versions of same module (preferably by different team) can be developed, even when only a single version will be used in the delivered system
- Test cases designed using other black-box techniques are provided as input to each version of the unit
- If the output from each version is the same, it is assumed that the module under test is correct
- Expensive; Preferable for critical applications only
- Not a foolproof technique – If all the versions are erroneous to the same input

White-box Testing

- Unit testing
  - White-box testing
    - Control structure testing
      - Statement coverage
      - Condition coverage
      - Branch coverage
      - Data flow testing
      - Loop testing
    - Basis path testing

Why White-box Testing

- We have done Black-box testing and the testing ensures that the program requirements have been met or not
  - Black-box testing discovers errors but not the sources of the errors
  - Black-box testing may not be exhaustive to uncover certain errors (such as, incorrect assumption, logical errors, typographical errors etc.)

White-box Testing

- Also called *glass-box* testing entirely based on the program structure
- To derive the tests cases, so that
  - Exercise all statements in the module
  - Exercise all logical decisions on their true and false sides
  - Execute all loops at their boundaries and within their operational bounds
  - All independent paths within a module have been exercised at least once
  - Exercise internal data structures to ensure their validity

White-box Testing Strategies

- White-box testing strategies
  - Control structure testing
    - Statement coverage
    - Branch coverage
    - Condition coverage
    - Loop testing
  - Basis path testing
  - Data flow testing

Statement Coverage

- The statement coverage strategy aims to design test cases so that every statement in a program is executed at least once
- Statement coverage debugs the failure due to some illegal memory access (e.g. pointers), wrong result computation etc.
- This technique derives test case so that all statements in a program is executed at least once

Statement Coverage: Example 1

Euclid's GCD Algorithm

```
int GCD(int x, int y)
{
1 while (x != y) {
2    if (x>y) then
3         x=x-y;
4    else y=y-x;
5 }
6     return x;
}
```

Cover statement 1: [x = 5, y = 4]

Cover statement 3: [x = 5, y = 4]

Cover statement 4: [x = 4, y = 5]

So the test case: { [5,4], [4,5] }

Statement Coverage: Example 2

```
Read(x,y)
{
If(x>y)
   Print(x)
Else
    Print (y)
}
```

Cover statement : [x = 5, y = 2]

Cover statement : [x = 2, y = 5] [x = 5, y = 5]

The test case: { ?…? }

Branch Coverage Testing

```
int GCD(int x, int y)
{
1 while (x != y) {

2    if (x>y) then
3        x=x-y;
4    else y=y-x;
5 }
6     return x;
}
```

Branch 1: [x = 5, y = 3]
[x = 3, y = 5]
[x = 3, y = 3]

Branch 2: [x = 5, y = 3] (true)
[x = 3, y = 5] (false)

Condition Coverage Testing

1. if C then { S1 } else { S2 }
2. While C then { S }
3. do { S } while C
4. switch C
   case { S1 }
   case { S2 }
   ….
   ….

C = condition with boolean operators
and/or relational operator

Example 1: C = (x and y) or z

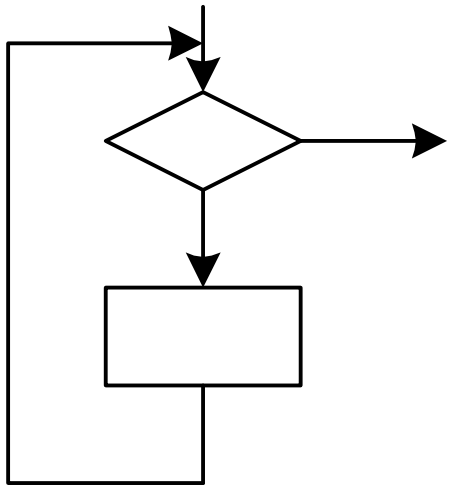
True			False		
x	y	z	x	y	z
1	1	1	0	0	0
0	0	1	0	1	0
1	0	1			
0	1	1			
1	1	0			

Test case = 2^3

Example 2: C = (x and y) and z

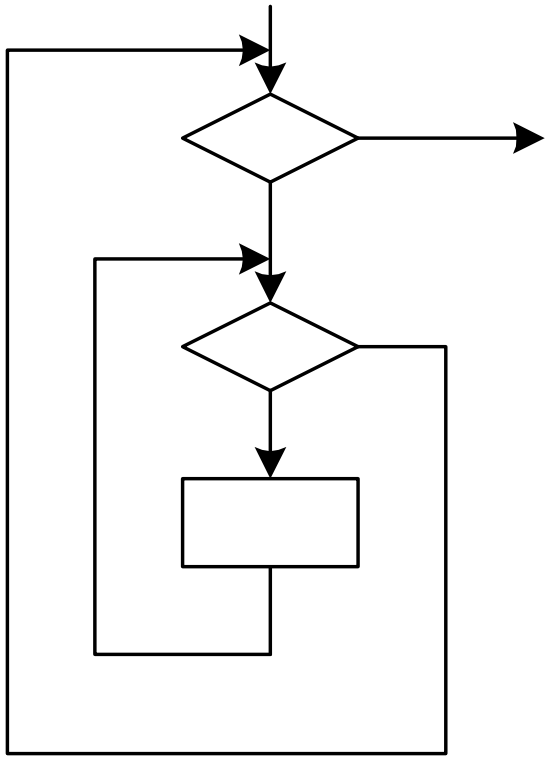
True: 1 1 1 False: 0 1 0

Loop Testing



- Test for
  - Skip the lop entirely
  - Only one pass
  - Two passes
  - m passes, m < n
  - n-1, n, n+1 passes

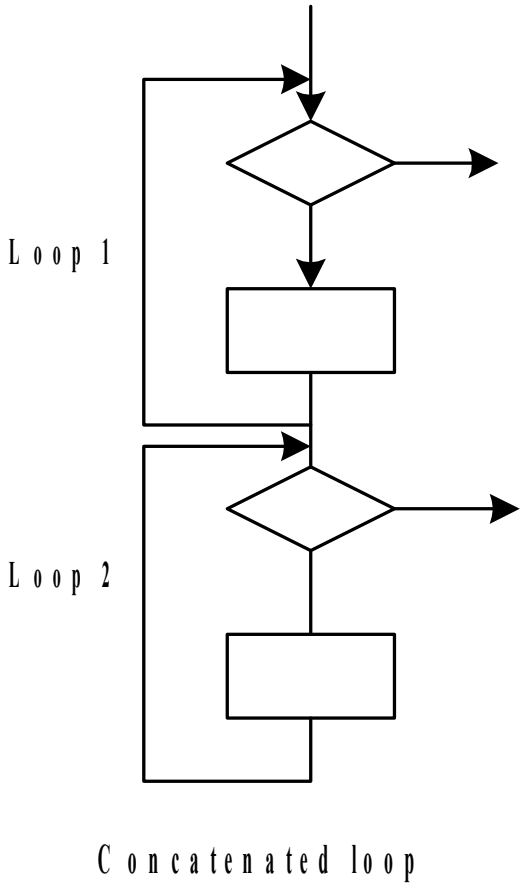
Loop Testing



Nested loop

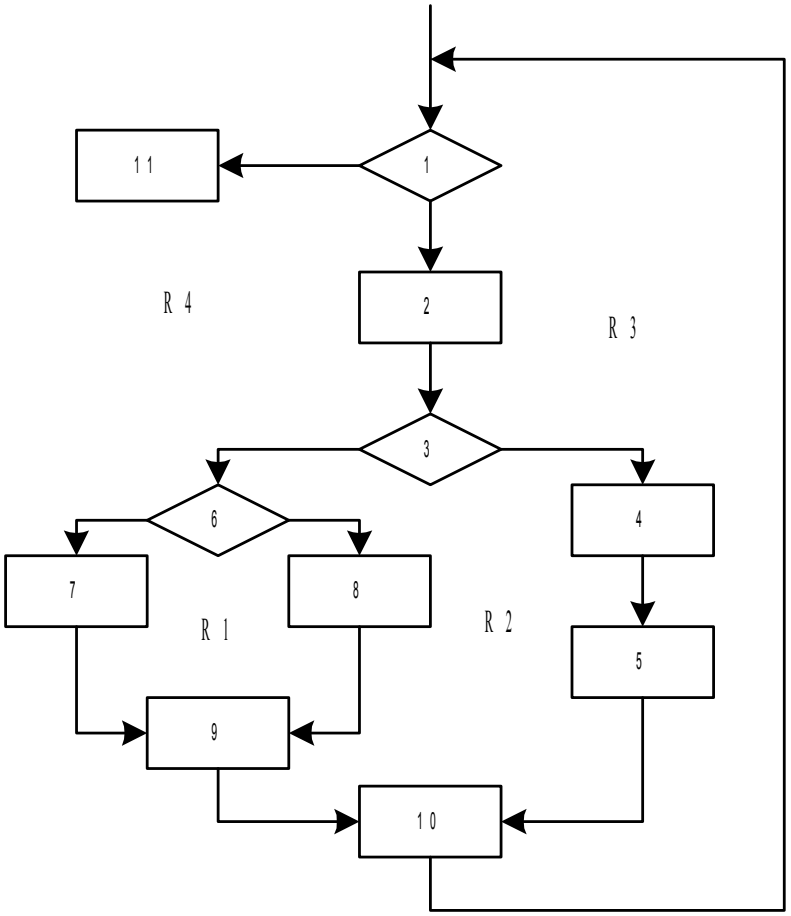
- Test for
  - Innermost loop → simple loop
  - Outermost loop → simple loop

Loop Testing



- Test for
  - Loop 1 → simple loop
  - Loop 2 → simple loop

Basis Path Testing



Cyclomatic complexity metric

(McCabe's Number)

R = E – N + 2

E = # edge

N = # node

R = Number of independent paths

= Number of predicate nodes + 1

Example: Number of basis paths = 4

Path 1: 1-11

Path 2: 1-2-3-4-5-10-1-11

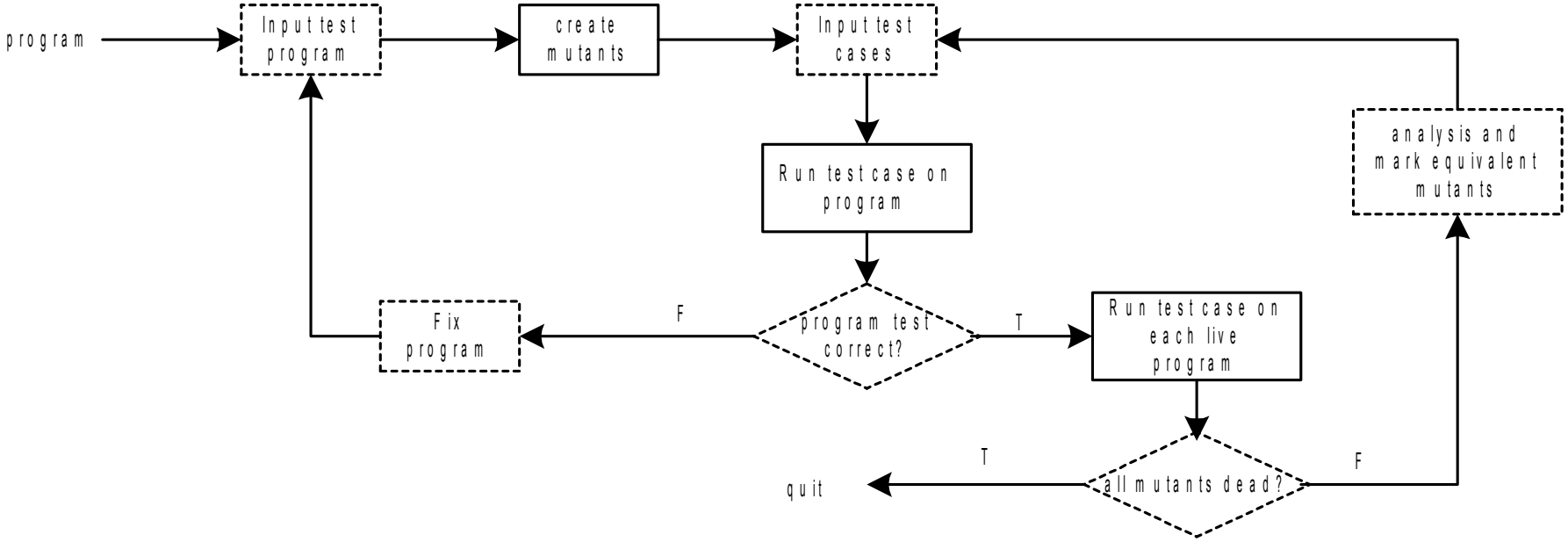
Path 3: 1-2-3-6-8-8-10-1-11

Path 4: 1-2-3-6-7-9-10-1-11

Mutation Testing

- It is a fault-based testing
- Faults are deliberately injected into a program, in order to determine whether or not a set of test inputs can distinguish between the original program and the programs with injected faults
- It is based on adequacy criteria: whether a test set is adequate to cover all faults or not
- Mutation: is a simple change (error) into the code being tested; each version is called a mutant
- Program neighborhood: The original program plus the mutant programs are collectively known as the program neighborhood
- Mutant dead: if the execution of the mutated code against the test set distinguishes the behavior or output from the original program
- Mutation adequacy score: To supply test set until all mutants are dead

Mutation Testing



Integration Testing

Why Integration Testing?

- Integration testing
  - Interfacing
    - Data can be lost across an interface
    - One module can have an inadvertent, adverse effect on another
    - Sub functions, when combined, may not produce the desired function
    - Individually acceptable imprecision may be magnified to unacceptable levels
    - Global data structures can causes problems

    and so on .....

Integration Plan and Testing

- Integration plan
  - A systematic integration plan should be adopted prior to testing
  - The integration plan specifies the steps and order in which modules are to be combined to realize the full system
  - An important factor to guide the integration plan is the module dependency graph as obtained in the structured design of the system
  - After each integration step, the partially integrated system is to be tested

Integration Testing Strategies

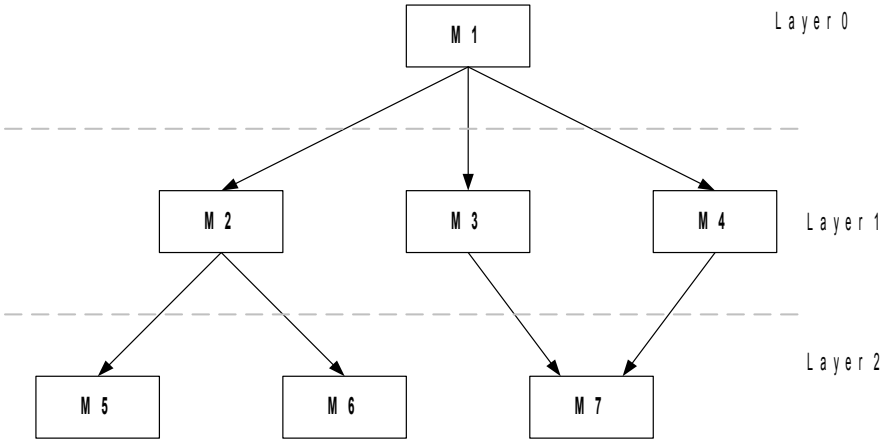
- Following are the well known practices
  - Big-bang integration testing
  - Top down integration testing
  - Bottom-up integration testing
  - Mixed integration testing
  - Smoke testing

Big-bang Integration Testing

- The most simple integration testing (non incremental) approach
- All modules making up a system are integrated at one go
- The entire program is tested as a whole
  - The chaos usually results!
  - Errors debugging are very expensive to fix
- Suitable, only for very small system (or a part of a lager subsystem)

Top-down Integration Testing

- It is an incremental approach to build and test a system
- Starts with main module and add the modules
  - Depth-first integration
  - Breadth-first integration



Top-down Integration Testing

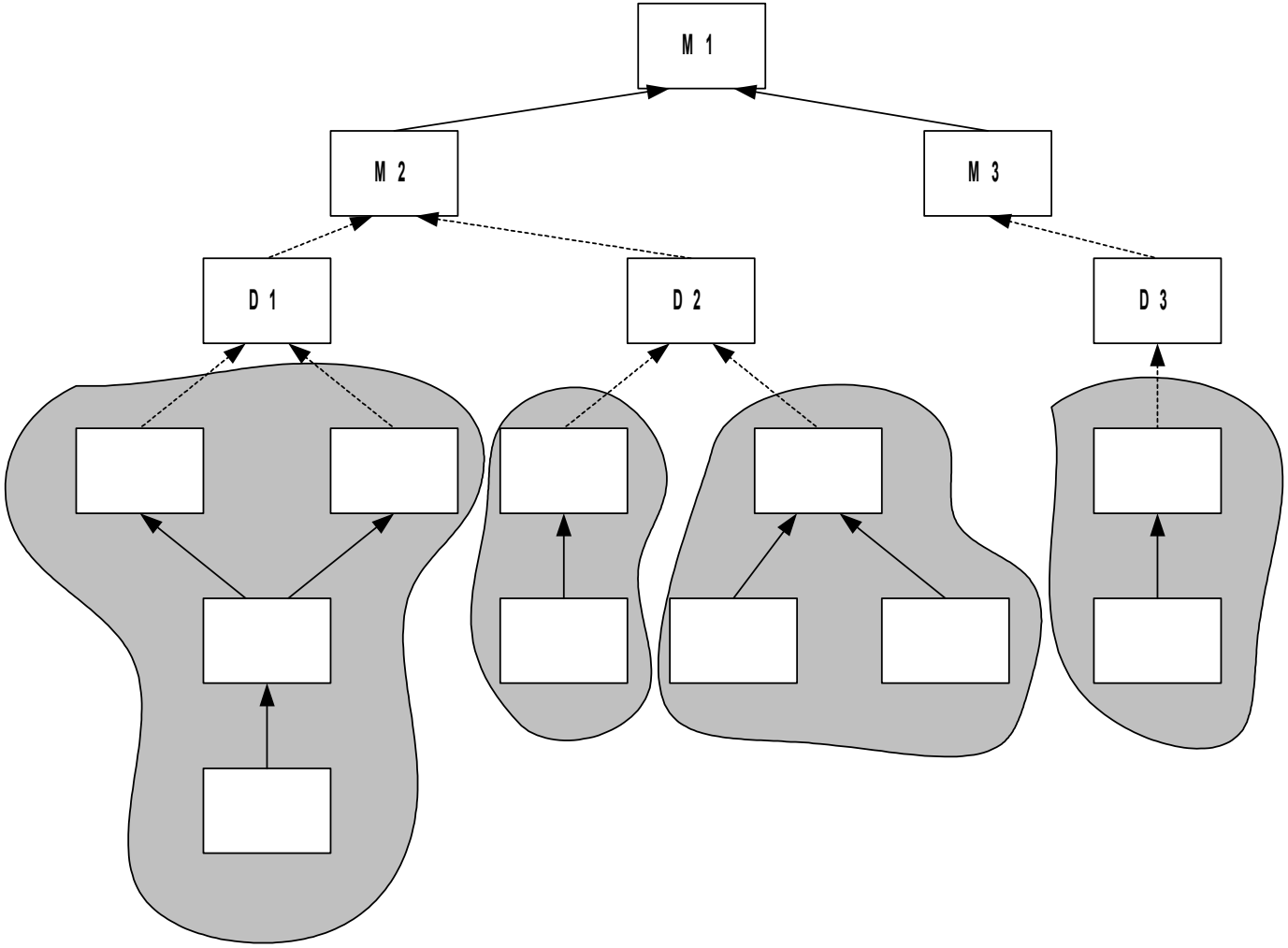
- This testing strategy requires the use of stubs
  - Depending on integration approach (depth or breadth first), stubs are replaced one at a time with actual components
  - Retesting with actual modules usually recommended
- Logistical problem can arise
  - Low-level stub replacement at top-level does not ensure the data flow in upward direction as the integration continued

Bottom-up Integration Testing

- Begins construction and testing at lowest level

1. Low-level modules are combined into clusters (also called builds) to build the higher level modules
2. A driver is required to simulate the behavior of a cluster
3. The cluster is tested
4. Drivers are removed and clusters are combined moving upward in the system structure

Bottom-up Integration Testing



Bottom-up Integration Testing

- **Advantages**
  - The bottom-up integration conforms with the basic intuition of the system building
  - Several disjoint subsystem can be tested simultaneously
  - No stubs are required; only the test drivers are required
- **Disadvantage**
  - Complexity increases as the number of subsystem increases
  - Extreme case corresponds to the big-bang approach

Mixed Integration Testing

- Also called **sandwiched integration testing** approach
- It combines the top-down and bottom-up integration testing approaches
- Using top-down approach, testing can start only after the top-level module have been coded and unit tested
- Using bottom-up approach, start the bottom-up testing as soon as bottom-level modules are ready
- Then move up-ward as well as down-ward to perform tests with the currently available modules
- **Advantages**
  - The mixed approach overcomes the shortcoming of the top-down and bottom-up approaches
  - This is one of the most commonly adopted testing approach

Smoke Testing

- Smoke testing - typically a testing effort to determine if a new software version is performing well enough to accept it for a major testing effort

Example:

If the new software is crashing systems every 5 minutes, bogging down systems to a crawl, or corrupting databases, the software may not be in a 'sane' enough condition to warrant further testing in its current state

- Smoke testing is also alternatively termed as **Sanity Testing**

Smoke Testing

- Important
  - Smoke test should exercise the entire system from end to end and on a regularly basis
  - It does not have to be exhaustive, but it should be capable of exposing major errors
- Advantages
  - Minimize the integration risk
  - The quality of the end-product is improved, as it is likely to uncover both functional errors, architectural and component-level design defects
  - Error diagnosis and corrections are simplified, as the test is associated with incremental (new build then smoke test and then rebuild etc.)

System Testing

System Testing

- System tests are designed to validate a fully developed system to assure that it meets all the requirements as specified in the SRS document
- System testing can be considered as the black-box testing
- There are two main objectives of the System testing
  - **Acceptance testing**
    - To check whether the system satisfies the functional requirements as documented in the SRS
    - To judge the acceptability of the system by the user or customer
  - **Performance testing**
    - To check whether the system satisfies the non-functional requirements as documented in the SRS

System Testing

- Acceptance testing
  - Conducted by the end-user rather than software engineers
  - **Alpha testing**
    - The test is carried out by a customer but at the developer's site
    - Developer looking over the shoulder of the user and recording errors and usage problems
  - **Beta testing**
    - The test is conducted at one or more customer sites by the end-user of the software
    - Unlike the Alpha testing, the developer is generally not present
    - Customers record all problems and report these to the developer at regular interval
    - As a result of test report, developer makes modifications and then release next beta version

System Testing

- Performance testing
  - The performance testing is carried out on a system depend on the different nonfunctional requirement of the system documented in the SRS document
  - There are several types of performance testing
    - Stress testing
    - Volume testing
    - Compatibility testing
    - Recovery testing
    - Security testing

System Testing

- **Stress testing**
  - To test the behavior of the system against a range of abnormal and invalid input condition
  - Input data volume, input data rate, utilization of memory etc. are tested beyond the designed capacity

  Example
    - A system in a concurrent environment with 60 users with 20 transaction per second can be tested beyond this threshold

System Testing

- **Volume testing**
  - The test is to ensure the validity of the data structures (arrays, stacks, queues etc.) under some extraordinary situation

  Example
  - A compiler might be tested to check whether the symbol table overflows when a very large recursive program is compiled

System Testing

- **Compatibility testing**
  - To test whether the system is compatible with other types of system
  - Basically to check whether the interface functions are able to communicate satisfactorily or not

  Example
    - A Browser is compatible with Unix, Windows etc.

System Testing

- **Security testing**
  - To test that security mechanism built into a system will, in fact, protect it from unauthorized access or not
  - The tester try to challenge the security measures in the system

  Example
    - Attempt to acquire password, write some routine to breakdown the defenses, may overwhelm the system to deny the services to others etc.

Regression Testing

- It is the practice of running an old test suite after each change to the system or after each bug fix to ensure that no new bug has been introduced as a result of the change made or bug fixed

- It does belong to all categories of the testing

Problems to Ponder

- What are 5 common problems in the software development process?
- What are 5 common solutions to software development problems?
- What is software 'quality'?
- What is 'good code'?
- What is 'good design'?
- Will automated testing tools make testing easier?
- What makes a good Software Test engineer?
- What makes a good Software QA engineer?
- What makes a good QA or Test manager?