Week 9: Software Testing

Dr. Sridhar Iyer, IIT Bombay
Dr. Prajish Prasad, FLAME University

Software Engineering

Testing - Motivation and Terminologies

Dr. Sridhar Iyer, IIT Bombay
Dr. Prajish Prasad, FLAME University

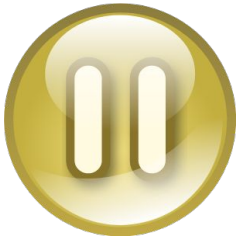
Importance of Testing

- Errors in software programs – even simple errors can cause entire systems to crash

- E.g. - Airport system failure, Spacecraft crash

Reflection Spot

How have you tested a program? What are the basic activities you did to check if your program is working correctly?



Please pause the video and written down your responses

Basic Activities in Testing

- Provide inputs
- Check output
- See if the output matches expectations
- Locate where the error is
- Identify the cause
- Fix the error
- Test again

Test Case

- Test case – triplet [I, S, R]
  - I – data input to the programming
  - S – State of the program at which the data is to be input
  - R – result expected to be produced by the program

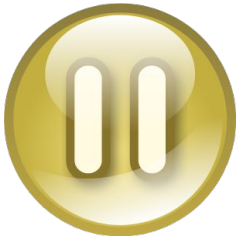
- Test suite – set of all test cases which have been designed to test a given program

Summary of Testing Activities

- Test suite design: Come up with appropriate test cases
- Run test cases and check results
- Locate error
- Error correction

Reflection Spot

Why do we need to “design” test cases?
Why do we need to choose test cases carefully?



Please pause the video and written down your responses

Why Design Test Cases

- Testing for a large collection of randomly selected test cases do not guarantee that all (most) errors will be uncovered

Why Design Test Cases

Calculate min between two values

if (x<y) min = x

else **min = x**

Designing Test cases

- Domain of all input values in a software system is sufficiently large

- Necessary to design a minimal test suite where each test case helps detect different types of errors

How to Design Test Cases

if (x<y) min = x

else min = x

Types of test cases -

- first value in the pair is minimum
- second value in the pair is minimum
- Both values are equal

How to Design Test Cases - Blackbox Testing

if (x<y) min = x

else min = x

Types of test cases -

- first value in the pair is minimum
- second value in the pair is minimum
- Both values are equal

Blackbox Testing

How to Design Test Cases - Whitebox Testing

- Analyse the structure of the program
- Analyse the code for every path
- Test coverage - checking if all paths of the program are covered by the test cases

Different Levels of Testing

- **Unit testing** – individual functions/units of a program are tested
- **Integration testing** – Units are incrementally integrated and tested after each step of integration
- **System testing** – the fully integrated system is tested
  - Alpha testing – test team within the organisation
  - Beta testing – select group of customers
- **Acceptance testing** – customer to determine whether to accept the delivery of the software

Unit Testing

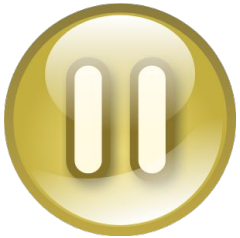
Dr. Sridhar Iyer, IIT Bombay
Dr. Prajish Prasad, FLAME University

Different Levels of Testing

- **Unit testing** – individual functions/units of a program are tested
- **Integration testing** – Units are incrementally integrated and tested after each step of integration
- **System testing** – the fully integrated system is tested
  - Alpha testing – test team within the organisation
  - Beta testing – select group of customers
- **Acceptance testing** – customer to determine whether to accept the delivery of the software

Reflection Spot

Why do unit testing? Why should we individually test each module and then integrate and test these modules again? Why not integrate and test once and for all?



Please pause the video and written down your responses

Why do Unit Testing

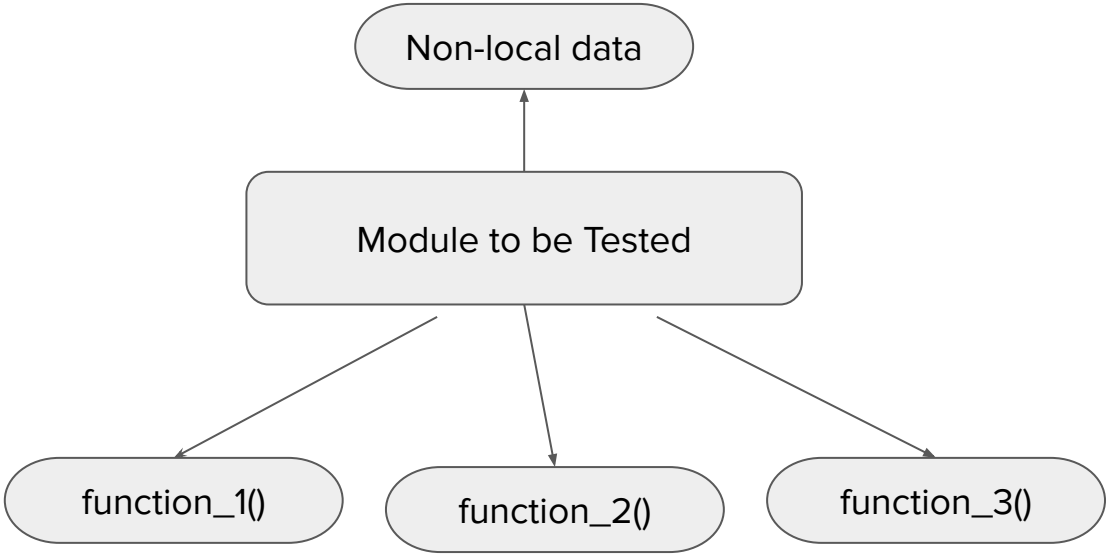
- While testing a module, other modules with which this module needs to interface may not be ready

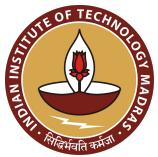
- Makes debugging easier
  - Failure/error is detected - when integrated set of modules are being tested - which module has the error?

Unit Testing

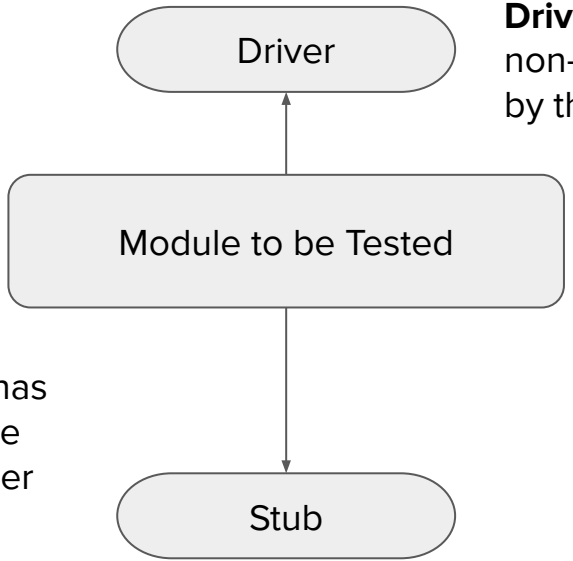
- When - During the coding of the module. Not in the testing phase
- Who - Person writing the code for the module
- What is involved:
  - Unit test cases have to be designed
  - Testing environment for the unit under test

Testing Environment



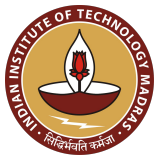


Testing Environment



Driver Module: Contains the non-local data structures accessed by the module under test.

Stub: Dummy procedure that has the same I/O parameters as the function called by the unit under test but has a highly simplified behaviour.



Python Unit Testing Tools

- unittest
- pytest

Blackbox and Whitebox Testing

Dr. Sridhar Iyer, IIT Bombay
Dr. Prajish Prasad, FLAME University

Introduction

- Unit Testing - Individual functions/units of a program are tested
- Blackbox Testing - Program ➔ Black box
- Whitebox Testing - Analyze the structure of the program

Blackbox Testing

- Designing Test cases -
  - Examining input/output values only
  - No knowledge of design or code required

Equivalence Classes

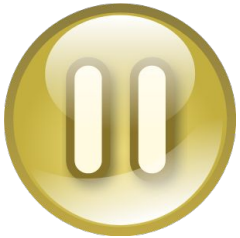
- Domain of input values partitioned into a set of equivalence classes

- Program behaves similarly for every input data in a particular equivalence class

- Example - Calculate the minimum of 2 values - x and y
- Equivalence classes - (x>y), (x<y), (x=y)

Reflection Spot

What are the equivalence classes for the following function - isPrime(num) which returns true if num is Prime, else returns False



Please pause the video and written down your responses

Equivalence Classes: Example

- What are the equivalence classes for the following function - isPrime(num) which returns true if num is Prime, else returns False

- Prime Numbers
- Non-Prime Numbers

Boundary Value Analysis

- Examples:
  - `for i in range(n) or for i in range(n-1) ?`
- Examine the values at boundaries of the equivalence classes
- function - isPrime(num) which returns true if num is Prime, else returns False
  Boundary values -- Check for 0, 1

Blackbox Testing - Summary

- From the requirements/specifications, identify the input and output structures and values
- Identify equivalence classes
- Design test cases - identify one representative test case from each equivalence class
- Design boundary value test cases

Whitebox Testing

- Analyze the structure of the program using some heuristics
- One such heuristic - coverage

Coverage-based Testing

- Branch coverage
- Multiple condition coverage
- Path coverage

Branch Coverage

- Every branch in the program needs to be taken at least once

```
if (x<y) min = x
else min = y
```

Branch coverage ➜
Test case 1 - x<y
Test case 2 -x>y

Reflection Spot

What are the test cases that will guarantee branch coverage?

```
1   def isPrime(num):
2       if num == 0 or num == 1:
3           return False
4       for n in range(2, num):
5           if num%n == 0:
6               return False
7       return True
```

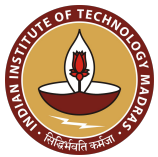
Please pause the video and written down your responses

Answer: Reflection Spot

What are the test cases that will guarantee branch coverage?

```
1   def isPrime(num):
2       if num == 0 or num == 1:
3           return False
4       for n in range(2, num):
5           if num%n == 0:
6               return False
7       return True
```

Test cases - {0, 2, 3, 4}



Coverage-based Testing

- Branch coverage
- **Multiple condition coverage**
- Path coverage

Multiple Condition Coverage

- Composite conditional expressions ➔ Each component condition takes true and false values

- num == 0 or num == 1
  {0, 1} achieves multiple condition coverage

Multiple Condition Coverage

- If a number is outside 2 boundary values

```
if(num<50   || num>150)
        doSomething();
    else
        doSomethingElse();|
```

- {49,100} → Branch Coverage

- {49,100,200} → Multiple Condition Coverage

Coverage-based Testing

- Branch coverage
- Multiple condition coverage
- **Path coverage**

Path Coverage

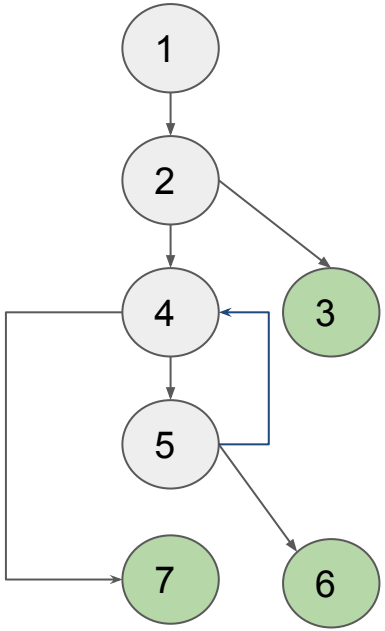
- A test-suite achieves path coverage - if it access all linearly independent path at least once.

- Access these paths - Control flow graph of a program

Path Coverage Example

Paths -

- {1,2,3}
- {1,2,4,7}
- {1,2,4,5,6}
- {1,2,4,5,4,7}

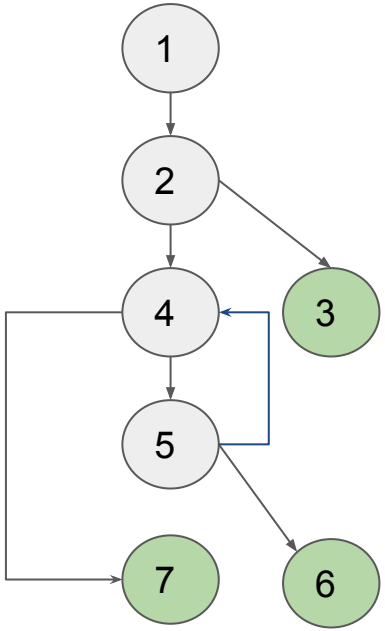



```
1   def isPrime(num):
2       if num == 0 or num == 1:
3           return False
4       for n in range(2, num):
5           if num%n == 0:
6               return False
7       return True
```

Reflection Spot: Test Cases for Each Path?

Paths -

- {1,2,3}
- {1,2,4,7}
- {1,2,4,5,6}
- {1,2,4,5,4,7}

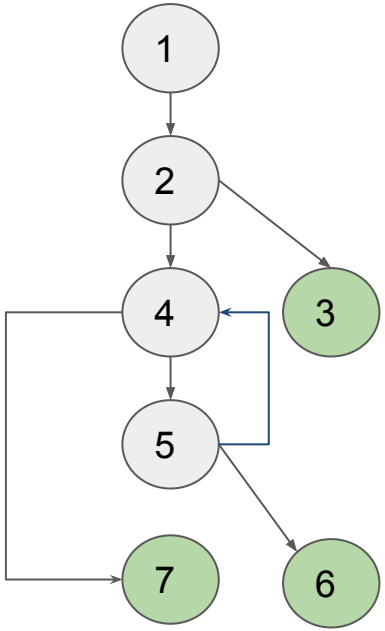



```
1   def isPrime(num):
2       if num == 0 or num == 1:
3           return False
4       for n in range(2, num):
5           if num%n == 0:
6               return False
7       return True
```

Reflection Spot: Test Cases for Each Path

Paths -

- {1,2,3} - **0/1**
- {1,2,4,7} - **2**
- {1,2,4,5,6} - **4**
- {1,2,4,5,4,7} - **3**

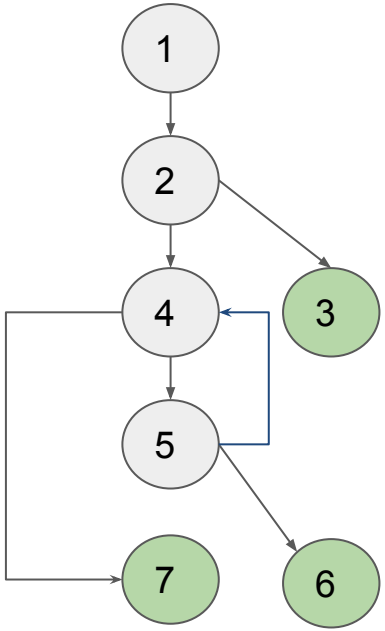



```python
def isPrime(num):
    if num == 0 or num == 1:
        return False
    for n in range(2, num):
        if num%n == 0:
            return False
    return True
```

Reflection Spot: Test Cases for Each Path

Paths -

- {1,2,3} - **0/1**
- {1,2,4,7} - **2**
- {1,2,4,5,6} - **4**
- {1,2,4,5,4,7} - **3**




```
1   def isPrime(num):
2       if num == 0 or num == 1:
3           return False
4       for n in range(2, num):
5           if num%n == 0:
6               return False
7       return True
```

Cyclomatic Complexity -

No of decision and loop statements + 1

Summary

- Blackbox testing
  - Equivalence classes
  - Boundary conditions

- Whitebox testing
  - Branch coverage
  - Condition coverage
  - Path coverage

Software Engineering

Integration and System Testing

Dr. Sridhar Iyer, IIT Bombay
Dr. Prajish Prasad, FLAME University

Introduction

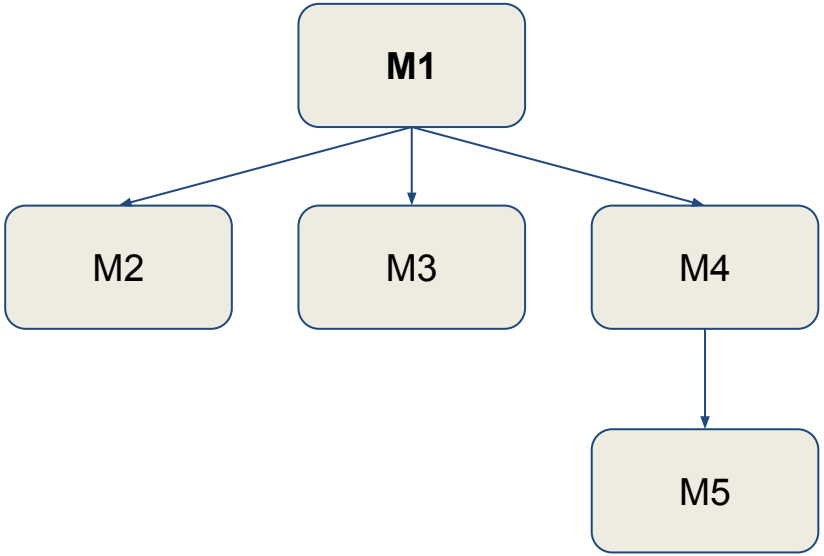
- Unit testing – individual functions/units of a program are tested
- Integration testing – Units are incrementally integrated and tested after each step of integration
- System testing – the fully integrated system is tested

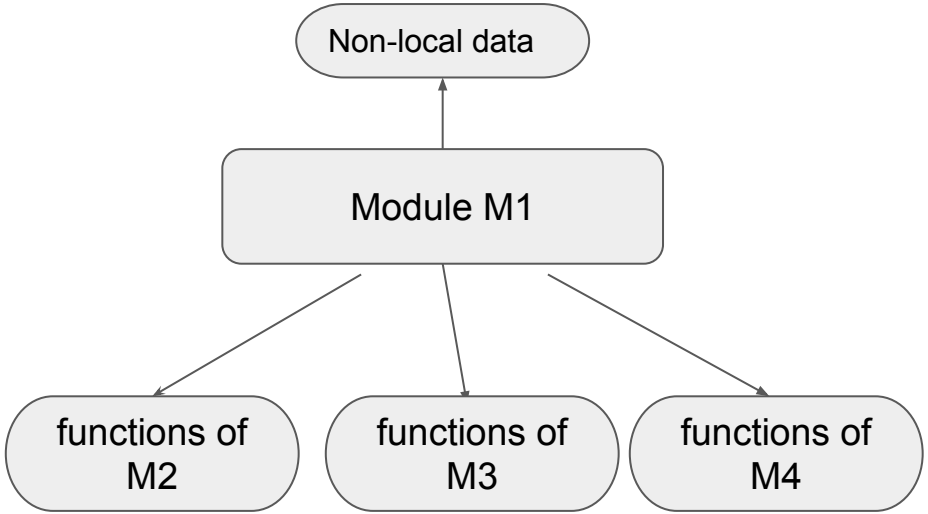
Integration Testing

- When? When at least a few or all modules have undergone unit testing

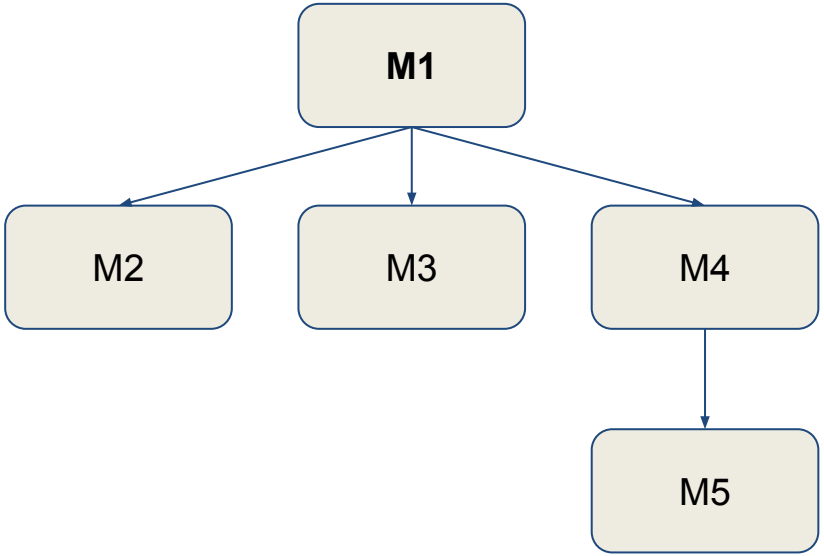
- Objective of integration testing - detect errors at the module interfaces

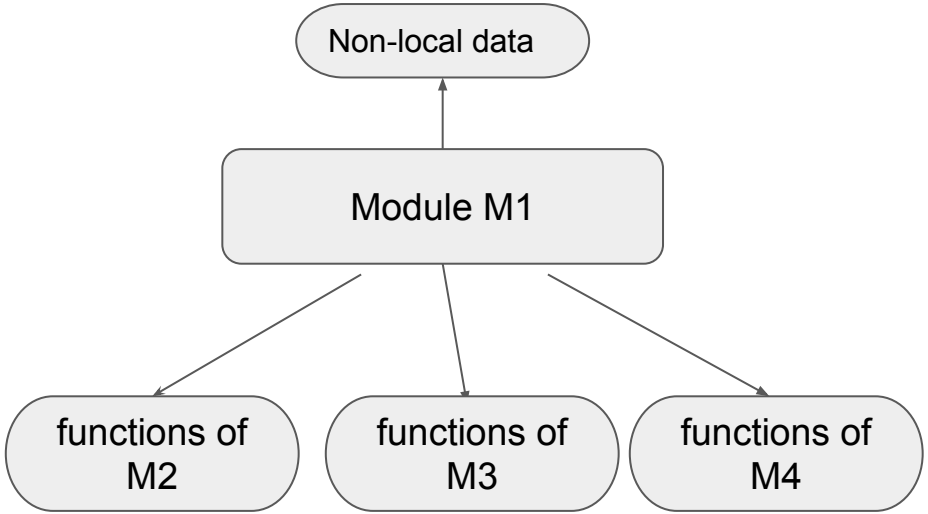
Example: Integration Testing





Approaches for Integration Testing

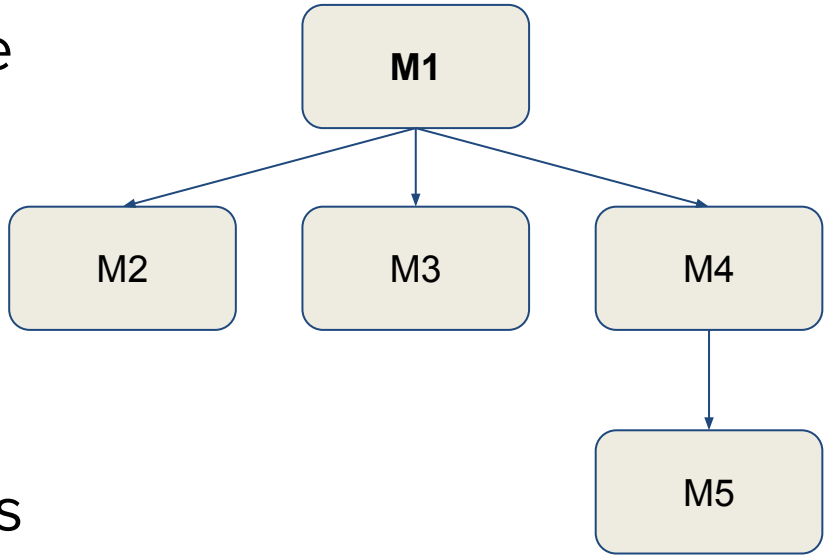




What are ways in which integration testing can be done?

Big Bang Approach

- All modules integrated in a single step
- Problem:
  Difficult to localize errors
- Meaningful only for small systems

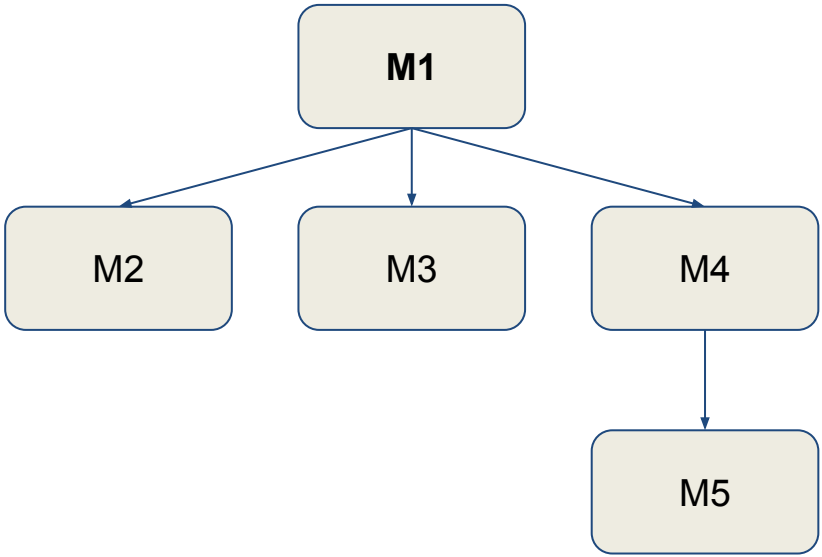


Bottom Up Approach

- Modules of each sub system are integrated

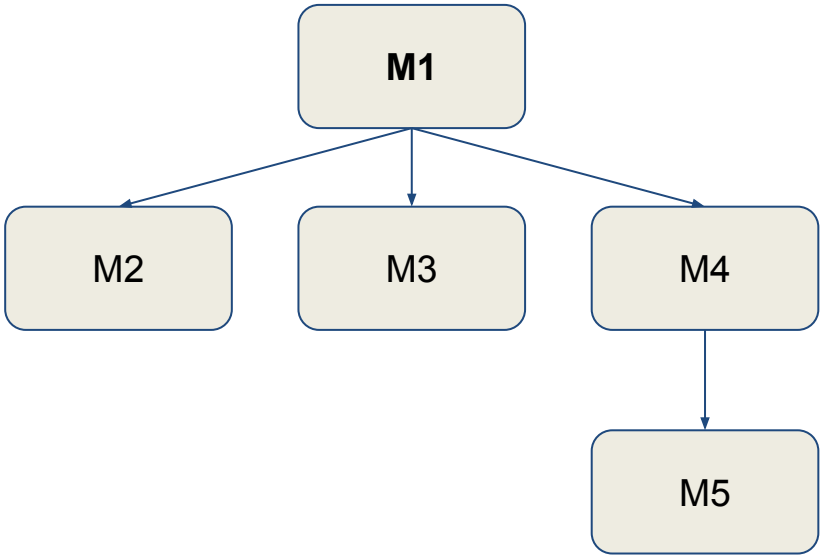
- E.g.
  - M1, M2, M3 - one sub-system
    M4, M5 - another sub-system

- Not necessary to create stubs
- Drivers are required



Top Down Approach

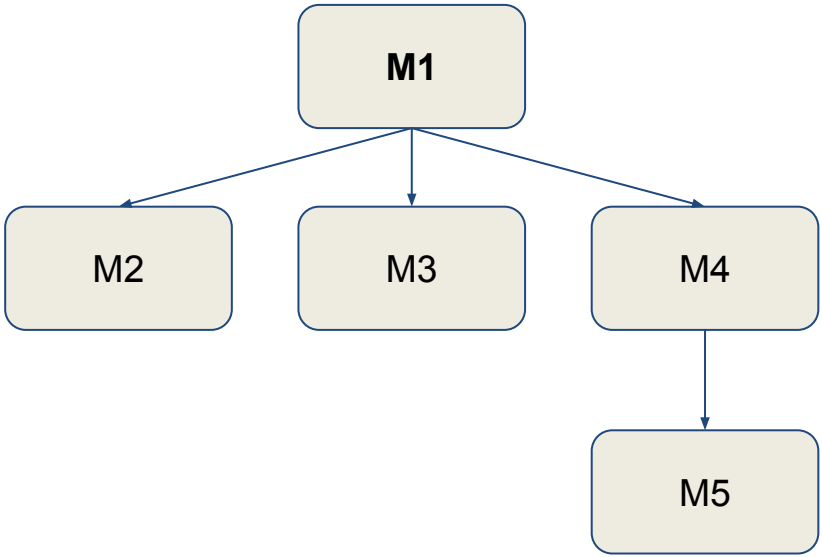
- Starts with the root module + 1-2 sub-ordinate modules of the root module
- After top-level modules are tested - modules at the next level are tested.



- Does not require drivers
- Stubs are required

Mixed Approach

- Use both top-down and bottom-up testing
- Integration Testing happens as and when modules become available after unit-testing



System Testing

- Unit testing – individual functions/units of a program are tested
- Integration testing – Units are incrementally integrated and tested after each step of integration
- **System testing** – the fully integrated system is tested

System Testing

- Alpha testing – test team within the organisation
- Beta testing – select group of customers
- Acceptance testing – customer to determine whether to accept the delivery of the software

Other Types of System Testing

- **Smoke Testing -**
  - Carried out before initial system testing
  - Checking whether basic functionalities are working
  - Few basic test cases designed
  - E.g. - Seller Portal - Add Products, Add Catalogue, Inventory etc.

Other Types of System Testing

- **Performance Testing -**
  - Check whether the system meets non-functional requirements
  - E.g. - **Stress Testing**
    - Input data volume, input data rate, processing time, utilisation of memory, etc., are tested beyond the designed capacity
    - E.g. - Simulate several transactions to DB beyond the expected capacity
  - Checking for performance, reliability, robustness etc.

Test-Driven Development (TDD)

Dr. Sridhar Iyer, IIT Bombay
Dr. Prajish Prasad, FLAME University

Test-Driven Development (TDD)

- Used especially in Agile Processes
- Write tests first for the functionality that we want to implement
- Drive design and development of that functionality from tests

Test-Driven Development (TDD)

- Express the functionality/feature/requirement in the form of a test
- Create a test Run the test - See it **FAIL**

- Create the Minimum code to meet the needs of the test
- Run it and See it **PASS**

- **REFACTOR** code - quality, make it modular, more elegant etc.

Test-Driven Development (TDD)

- RED: Write a Test and See it Fail
- GREEN: Write Code and See it Pass
- REFACTOR: Make the code better