Software Testing

A Definition…

Testing is the process of executing a program with the intent of finding errors.

- Glen Myers

Which definition of SW Testing is most appropriate ?

a) Testing is the process of demonstrating that errors are not present.

b) Testing is the process of demonstrating that a program performs its intended functions.

c) Testing is the process of removing errors from a program and fixing them.

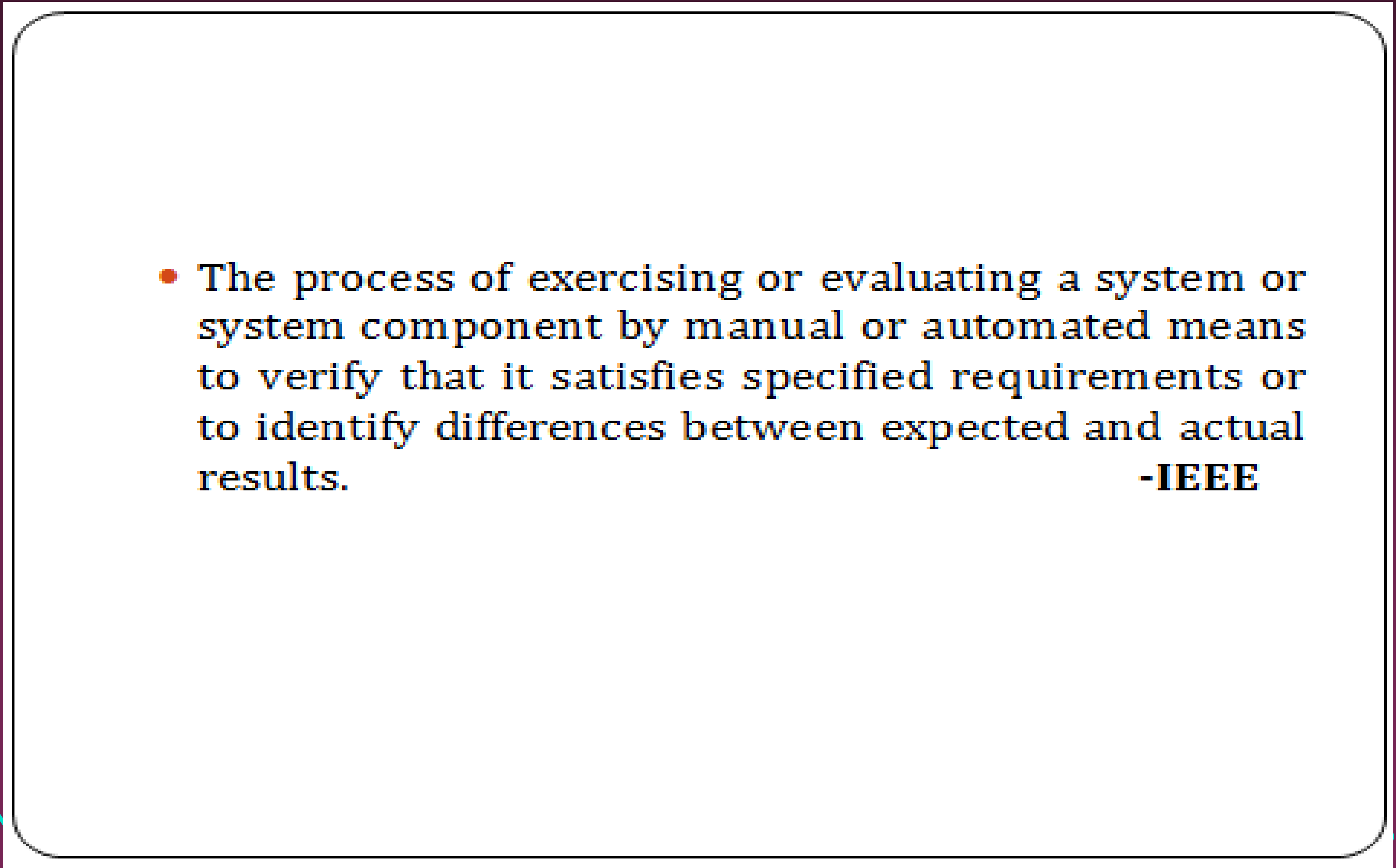
- *None of the above definitions set the right goal for effective SW Testing*

What is Software Testing ?

Testing is a verification (static) and validation (dynamic) activity that is performed by reviewing or executing program code.

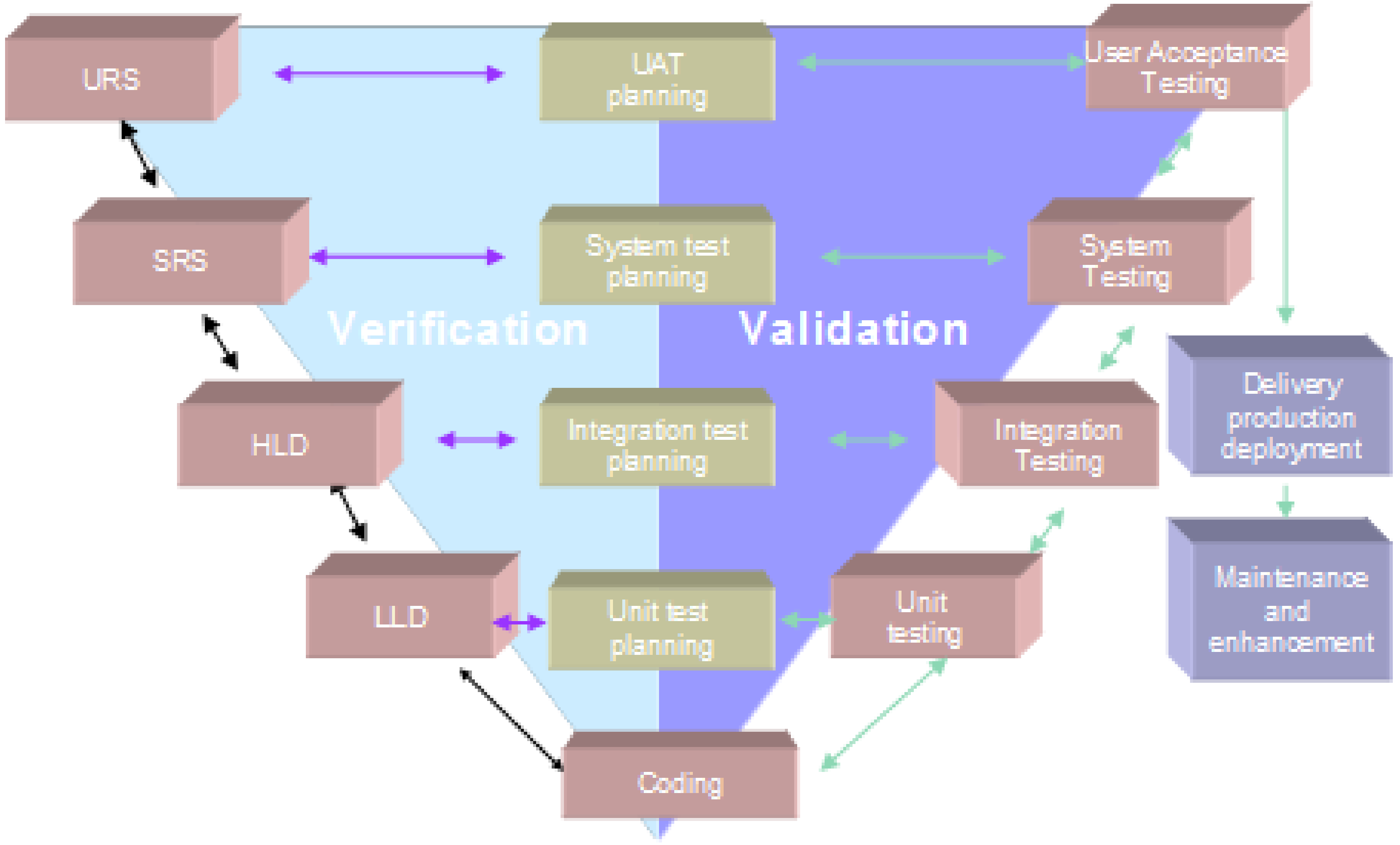
Testing is the practice of making objective judgments regarding the extent to which the system **meets, exceeds or fails to meet stated objectives/ requirements.**

A Good Definition



- The process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results. **-IEEE**

Software Development Life Cycle



Static Testing	**Dynamic Testing**
Testing done without executing the program	Testing done by executing the program
This testing does verification process	Dynamic testing does validation process
Static testing gives assessment of code and documentation	Dynamic testing gives bugs/bottlenecks in the software system.
Static testing involves checklist and process to be followed	Dynamic testing involves test cases for execution
This testing can be performed before **compilation**	Dynamic testing is performed after compilation
Static testing covers the structural and statement coverage testing	Dynamic testing covers the executable file of the code
Cost of finding defects and fixing is **less**	Cost of finding and fixing defects is high
Return on investment will be **high** as this process involved at early stage	Return on investment will be low as this process involves after the development phase

Objectives of SW Testing

- The main objective of SW testing is to find errors.
- Indirectly, testing provides assurance that the SW meets its requirements.
- Testing helps in assessing the **quality and reliability** of software.

Testing v/s Debugging

- Debugging is not Testing
- Debugging always occurs as a consequence of testing
- Testing attempts to find the error (cause of an error) and Debugging is to correct it.

Psychology of Testing

- Testing is a destructive process -- show that a program does not work by finding errors in it.
- Start testing with the assumption that - the program contains errors.
- A successful test case is one that finds an error.
- It is **difficult** for a programmer to test his/her own program effectively with the proper frame of mind required for testing.

Basic Testing Strategies

- Black-box testing
- White-box testing

Black-Box Testing

- Tests that validate business requirements -- (what the system is supposed to do)
- Test cases are derived from the **requirements** specification of the software. No knowledge of internal program structure (**code)** is used.
- Also known as -- functional, data-driven, or Input/Output testing

White-Box Testing

- Tests that validate internal program logic (control flow, data structures, data flow, statements etc.)

- Test cases are derived by examination of the internal structure (code) of the program.

- Also known as -- structural or logic-driven or glass box testing

Black-box v/s White-Box Testing

- Black box testing can detect errors such as
  incorrect functions, missing functions

  It cannot detect design errors, coding errors, unreachable code, hidden functions

- White box testing can detect errors such as
  - logic errors, design errors

  It cannot detect whether the program is performing its expected functions, missing functionality.

- Both methods of testing are required.

Black-box v/s White-box Testing

Black-box Testing	White-box testing
Tests function Can find requirements specification errors Can find missing functions	Tests structure Can find design and coding errors Can’t find missing functions

Is Complete Testing Possible ?

Can Testing *prove* that a program is completely free of errors ?

--- No

- Complete testing in the sense of a *proof* is not theoretically possible, and certainly not practically possible.

Example

- Test a function that adds two 32-bit numbers and returns the result.
  - Assume we can execute 1000 test cases per sec
- How long will it take to thoroughly test this function?
- 2^64 combinations in all (to test this program THOROUGHLY)

Is Complete Testing Possible ?...

- *2^10 = 1024 (approx 1000)*

2^64 = 2^4 * (2^10)(2^10)(2^10)(2^10)(2^10)(2^10)

*= 16 * (1000)^6   (approx)*

= 16000,000,000,000,000,000.

*Instructions executed in one year = 1000*60*60*24*30*12*

Is Complete Testing Possible ?...

- ***585 million years (approx.)***

Is Complete Testing Possible ?

- Exhaustive Black-box testing is generally **not possible** because the input domain for a program may be infinite or incredibly large.

- Exhaustive White-box testing is generally **not possible** because a program usually has a very large number of paths.

Implications ...

- **Test-case design**
  - careful selection of a subset of all possible test cases
  - The objective should be to maximize the number of errors found by a small finite number of test cases.

Test cases and Test suites

- Test case is a triplet [I, S, O] where
  - I - is **input** data
  - S - is **state** of system at which data will be input
  - O - is the **expected output**
- Test suite is set of all test cases
- Test cases are not randomly selected. Instead, they need to be designed.

Black-Box Testing

- Program viewed as a Black-box, which accepts some inputs and produces some outputs
- Test cases are derived solely from the **specifications**, without knowledge of the internal structure of the program.

Functional Test-Case Design Techniques

- Equivalence class partitioning
- Boundary value analysis
- Cause-effect graphing
- Error guessing

Equivalence Class Partitioning

- Partition the program **input domain** into equivalence classes (classes of data which according to the specifications are treated **identically** by the program)

- The basis of this technique is that test of a **representative** value of each class is equivalent to a test of any other value of the same class.

- identify **valid** as well as **invalid** equivalence classes

- For each equivalence class, generate a test case to exercise an input representative of that class

Example

- Example: input condition $0 <= x <= max$

  valid equivalence class : $0 <= x <= max$

  invalid equivalence classes : $x < 0$, $x > max$

- 3 test cases

Guidelines for Identifying Equivalence Classes

Input Condition	Valid Eq Classes	Invalid Eq Classes
range of values (eg. 1 - 200)	one valid (value within range)	two invalid (one outside each end of range)
number N valid values	one valid	two invalid (none, more than N)
Set of input values each handled differently by the program (eg. A, B, C)	one valid eq class for each value	one (eg. any value not in valid input set)

Guidelines for Identifying Equivalence Classes

Input Condition	Valid Eq Classes	Invalid Eq Classes
must be condition (eg. **Id name** must begin with a **letter**)	one (eg. it is a letter)	one (eg. it is **not** a letter)

- If you know that elements in an equivalence class are not handled identically by the program, split the equivalence class into smaller equivalence classes.

Identifying Test Cases for Equivalence Classes

- Assign a unique number to each equivalence class
- Until all valid equivalence classes have been covered by test cases, write a new test case covering as many of the uncovered valid equivalence classes as possible.
- Each invalid equivalence class cover by a separate test case.
- Test Cases for--
- **Triangle** problem- equilateral, scalene, Isosceles.
- Railway **ticket** booking- Child(<12 yrs), Senior citizen or normal ticket.

Boundary Value Analysis

- Design test cases that exercise values that lie at the boundaries of an input equivalence class and for situations just beyond the ends.

- Also identify output equivalence classes, and write test cases to generate o/p at the boundaries of the output equivalence classes, and just beyond the ends.

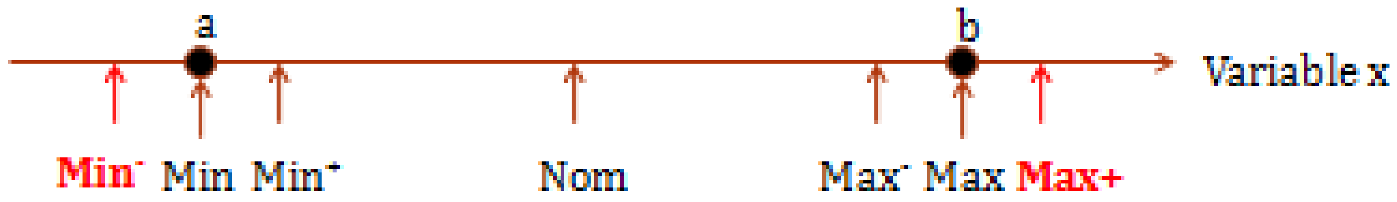
- Example: input condition 0 <= x <= max

  Test for values : 0, max   (valid inputs)

  : -1, max+1 (invalid inputs)

Boundary Value Analysis

- Consider a variable x with range [a, b]. Possible combinations using BVA are:
  1. x_{min-}
  2. x_{min}
  3. x_{min+}
  4. x_{nom}
  5. x_{max-}
  6. x_{max}
  7. x_{max+}



Error Guessing

- From perception and experience, enumerate a list of possible errors or error prone situations and then write test cases to expose those errors.

White Box Testing

White box testing is concerned with the degree to which test cases exercise or cover the logic (source code) of the program.

White box Test case design techniques

Statement coverage
Decision coverage
Condition coverage
Decision-condition coverage
Multiple condition coverage

Basis Path Testing
Loop testing

Data flow testing

White Box Test-Case Design

Statement coverage

- write enough test cases to execute every statement **at least once** (in order to examine every statement of the code)

TER **(Test Effectiveness Ratio) or Coverage Analysis**

TER1 = statements exercised / total statements

Example

```
void function eval (int A, int B, int X )
{
if  ( A > 1)  and ( B = 0 )
    then        X = X / A;
if ( A = 2 ) or ( X > 1)
    then        X = X + 1;
}
```

Statement coverage test cases:

1) A = 2, B = 0, X = 3 (X can be assigned any value)

White Box Test-Case Design

- Decision coverage (branch coverage)
  - write test cases to exercise the true and false outcomes of every decision

  TER2 = branches exercised / total branches

- Condition coverage
  - write test cases such that each condition in a decision takes on all possible outcomes at least once

Condition coverage

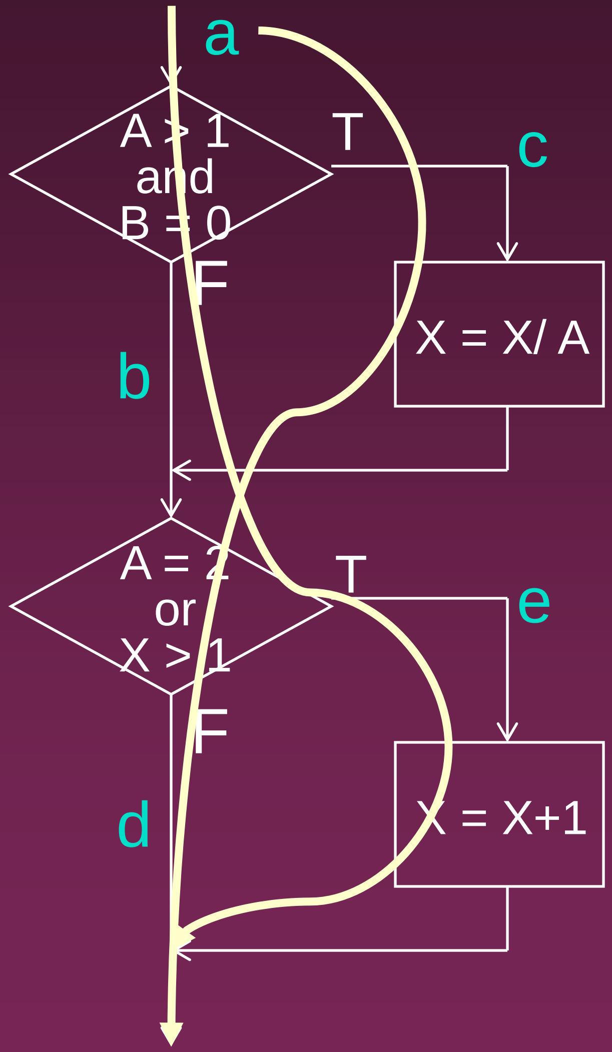
- In this structural testing, test cases are designed to make each component of a **composite conditional expression** to assume both true and false values.
- For example, in the conditional expression ((c1.and.c2).or.c3), the components c1, c2 and c3 are each made to assume both true and false values

Decision coverage Example

```
void function eval (int A, int B, int X )
{
if  ( A > 1)  and ( B = 0 ) then
        X = X / A;
if ( A = 2 ) or ( X > 1) then
        X = X + 1;
}
```

Decision coverage test cases:

1) A = 3, B = 0, X = 3   (acd)
2) A = 2, B = 1, X = 1   (abe)



Example

- Condition coverage test cases must cover conditions

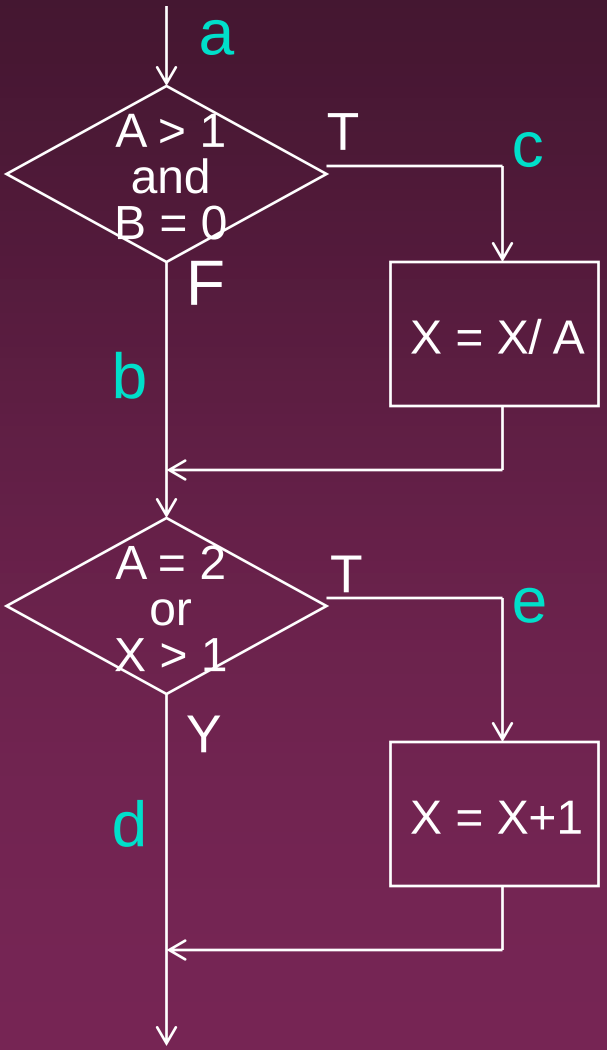
  A>1, A<=1, B=0, B !=0

  A=2, A !=2, X >1, X<=1

- Test cases:

1) A = 1, B = 0, X = 3 (abe)
2) A = 2, B = 1, X = 1 (abe)

- does not satisfy decision coverage



White Box Test-Case Design

- Decision Condition coverage
  - write test cases such that each **condition** in a decision takes on all possible outcomes at least once and each **decision** takes on all possible outcomes at least once

- Multiple Condition coverage
  - write test cases to exercise all *possible* **combinations** of True and False outcomes of conditions within a decision

Example

- Decision Condition coverage test cases must cover conditions

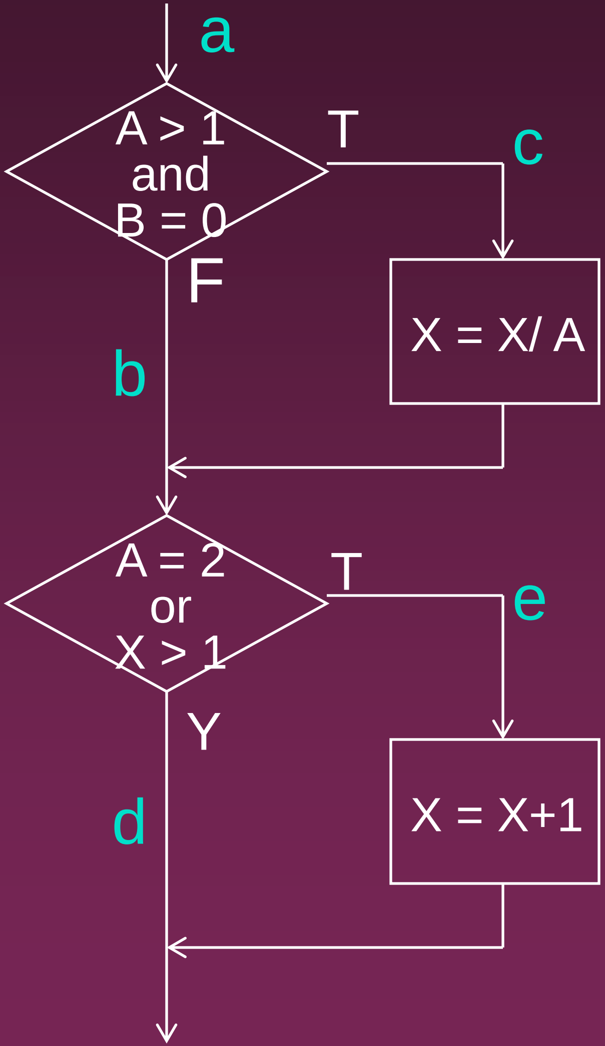
  A>1, A<=1,   B=0, B !=0

  A=2, A !=2,   X >1, X<=1

also (A > 1 and B = 0)   T, F
      (A = 2 or   X > 1)   T, F

- Test cases:

1) A = 2, B = 0, X = 4   (ace)
2) A = 1, B = 1, X = 1   (abd)



Basis Path Testing

1. Draw control flow graph of program from the program detailed design or code.

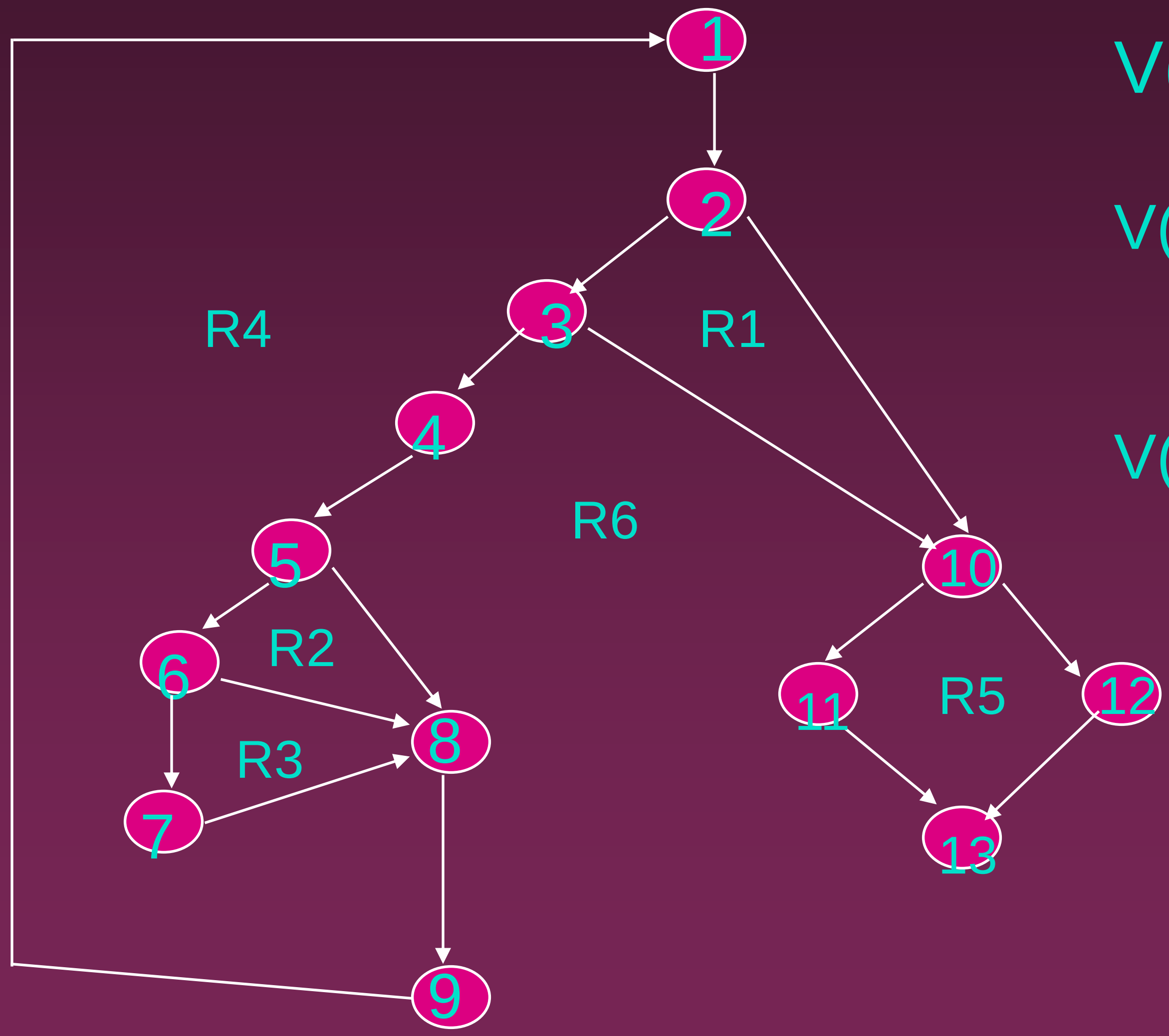
2. Compute the Cyclomatic complexity V(G) of the flow graph using any of the formulas:

   V(G) = #Edges - #Nodes + 2

   or V(G) = #regions in flow graph

   or V(G) = #predicates + 1

Example



V(G) = 6 regions

V(G) = #Edges - #Nodes
+ 2
= 17 - 13 + 2 = 6

V(G) = 5 predicate-nodes
+ 1 = 6

6 linearly
independent paths

- A linearly independent path is any path through the program that introduces at least one **new** edge that is not included in any other linearly independent paths. If a path has one new node compared to all other linearly independent paths, then the path is also linearly independent. This is because, any path having a new node automatically implies that it has a new edge. Thus, a path that is subpath of another path is not considered to be a linearly independent path.

Basis Path Testing (contd)

3. Determine a basis set of linearly independent paths.
   i—1-2-10-12-13;
   ii-- 1-2-10-11-13;   iii--1-2-3-10-12-13
   iv– 1-2-3-4-5-8-9-1…; v– 1-2-3-4-5-6-8-9-1…;
   vi—1-2-3-4-5-6-7-8-9-1…;
4. Prepare test cases that will force execution of each path in the Basis set.

- *The value of Cyclomatic complexity provides an upper bound on the number of tests that must be designed.*

Data Flow Testing

- Select test paths of a program based on the Definition-Use (DU) chain of variables in the program.

- Write test cases to cover every **DU chain** at least once.

- **Data-flow testing** uses the control flow graph and data flow graph to explore the unreasonable things that can happen to data (*i.e.,* anomalies).
- Consideration of data-flow anomalies leads to test path selection strategies that fill the gaps between complete path testing and branch or statement testing.

Data flow testing…

- (d) Defined, Created, Initialized
- • (k) Killed, Undefined, Released
- • (u) Used:
  - – (c) Used in a calculation
  - – (p) Used in a predicate

continue…

- **dd**: Probably harmless, but suspicious.
- • **dk**: Probably a bug.
- • **du**: Normal situation.
- • **kd**: Normal situation.
- • **kk**: Harmless, but probably a bug.
- • **ku**: Definitely a bug.
- • **ud**: Normal situation (reassignment).
- • **uk**: Normal situation.
- • **uu**: Normal situation.

Testing Principles

--- Glen Myers

- A good test case is one likely to show an error.

- Description of **expected output** or result is an essential part of test-case definition.

- A programmer should avoid attempting to test his/her own program.
    - testing is more effective and successful if performed by an **Independent** Test Team.

Testing Principles (contd)

- Avoid on-the-fly testing.
  Document all test cases.

- Test **valid** as well as invalid cases.

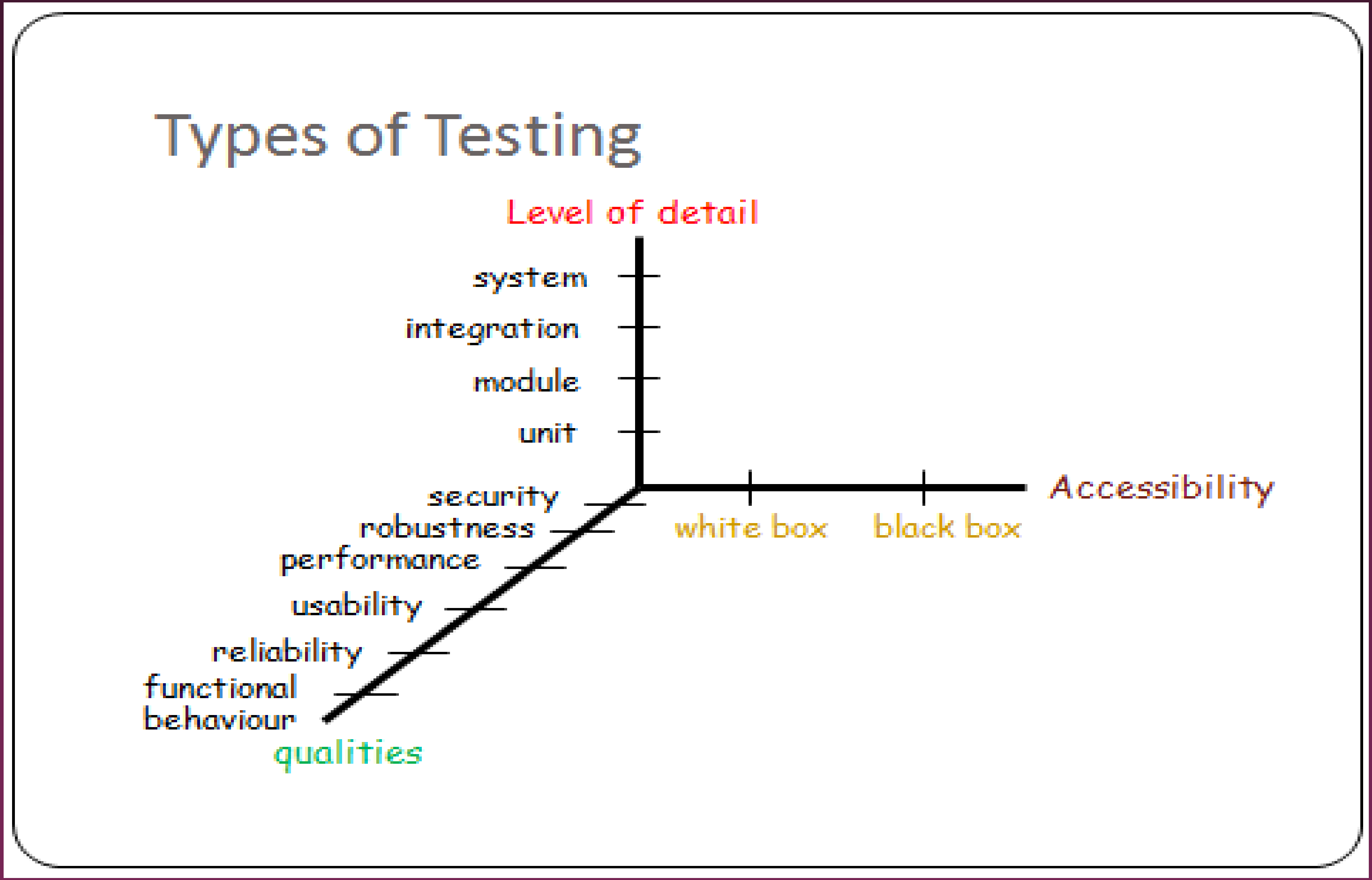
- Thoroughly inspect all test results.

- More detected errors implies even more errors may be present.

Testing Principles (contd)

- Decide in advance when to stop testing
- Do not plan testing effort under the implied assumption that no errors will be found.
- Testing is an extremely creative and intellectually challenging task.

Software Testing - II

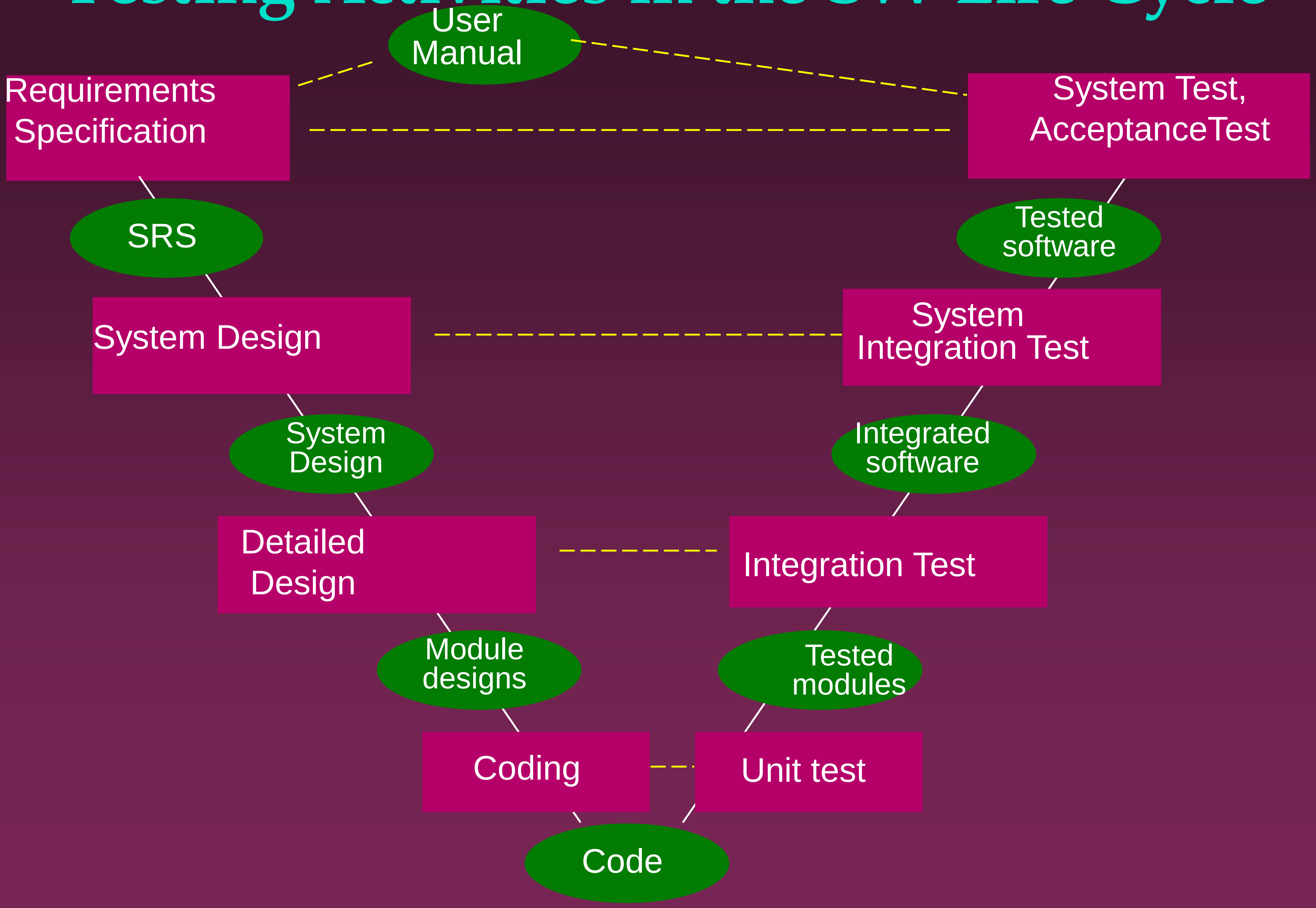


Types of Testing

SOFTWARE TESTING LIFECYCLE - PHASES

- Requirements study for testing purpose
- Test Case Design and Development
- Test Execution
- Test Log and Closure
- Test Process Analysis

Testing Activities in the SW Life Cycle



Levels of Testing

Type of Testing	Performed By
Low-level testing	
– Unit (module) testing	Programmer
– integration testing	Development team
High-level testing	
– Function testing	Independent Test Group
– System testing	Independent Test Group
– Acceptance testing	Customer

Unit Testing

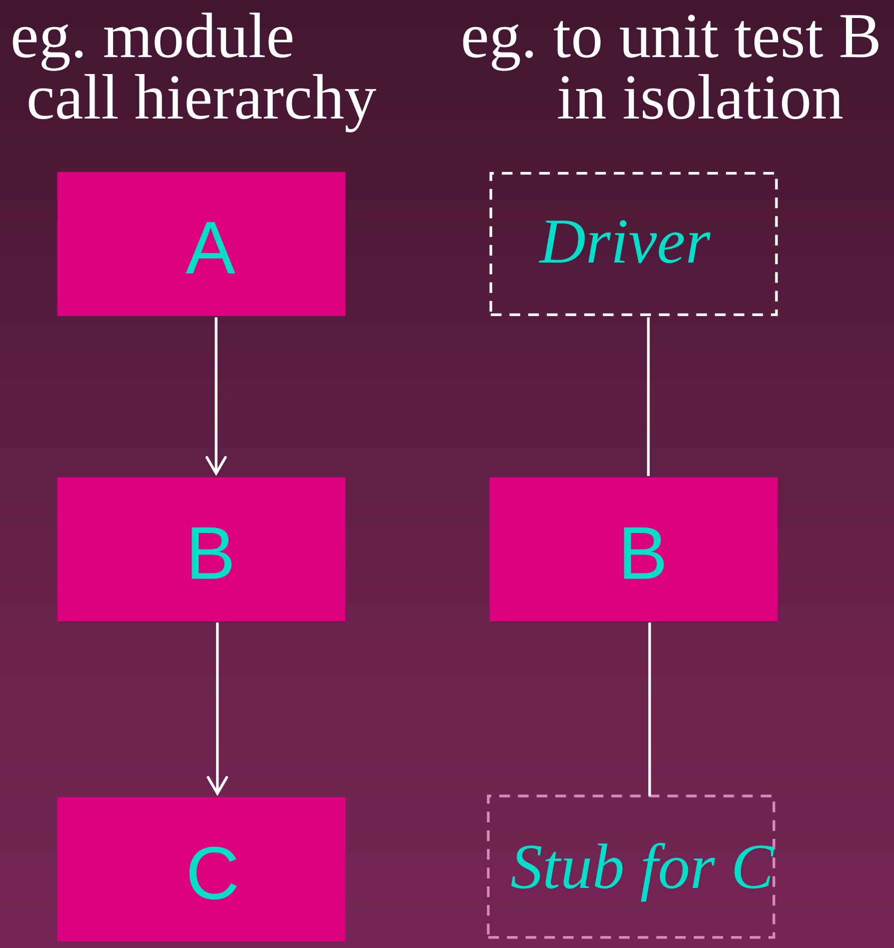
- done on individual modules
- test module w.r.t module specification
- largely white-box oriented
- mostly done by programmer
- Unit testing of several modules can be done in parallel
- requires *stubs* and *drivers*

Unit testing

Objectives	• To test the function of a program or unit of code such as a program or module • To test internal logic • To verify internal design • To test path & conditions coverage • To test exception conditions & error handling
When	• After modules are coded
Input	• Internal Application Design • Master Test Plan • Unit Test Plan
Output	• Unit Test Report

What are Stubs, Drivers ?

- Stub
  - dummy module which simulates the function of a module called by a given module under test
- Driver
  - a module which transmits test cases in the form of input arguments to the given module under test and either prints or interprets the results produced by it



Integration Testing

- tests a group of modules, or a subsystem
- test subsystem structure w.r.t design, subsystem functions
- focuses on module interfaces
- largely structure-dependent
- done by one/group of developers

Integration testing

Objectives	• To technically verify proper interfacing between modules, and within sub-systems
When	• After modules are unit tested
Input	• Internal & External Application Design • Master Test Plan • Integration Test Plan
Output	• Integration Test report

Integration Test Approaches

- Non-incremental (one step/ Big-Bang integration)
  - unit test each module independently
  - combine all the modules to form the system in one step, and test the combination
- Incremental
  - instead of testing each module in isolation, the next module to be tested is first combined with the set of modules that have already been tested
  - testing approaches:- Top-down, Bottom-up

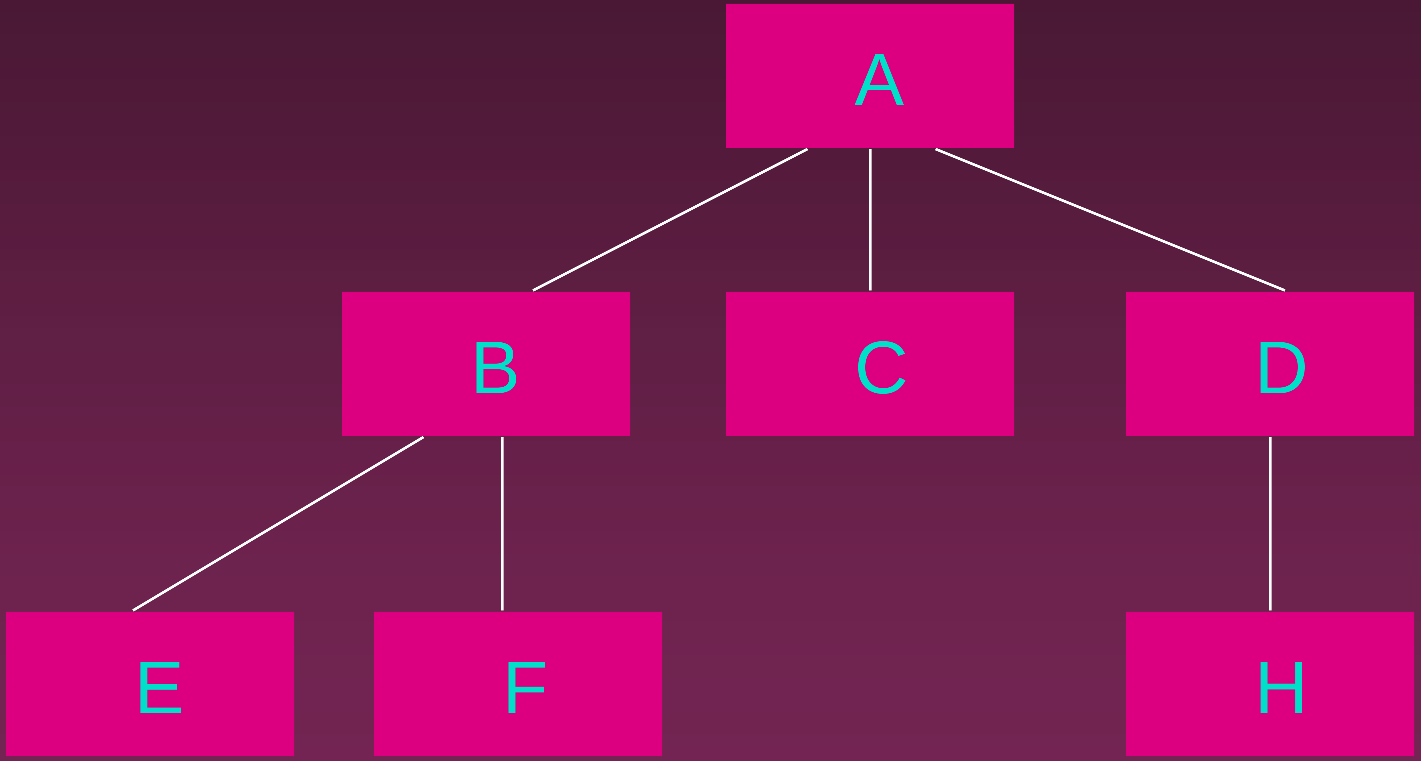
Comparison

Non-Incremental	Incremental
requires more stubs,drivers	requires less stubs, drivers
module interfacing errors detected late	module interfacing errors detected early
debugging errors is difficult	debugging errors is easier
	results in more thorough testing of modules

Top-down Integration

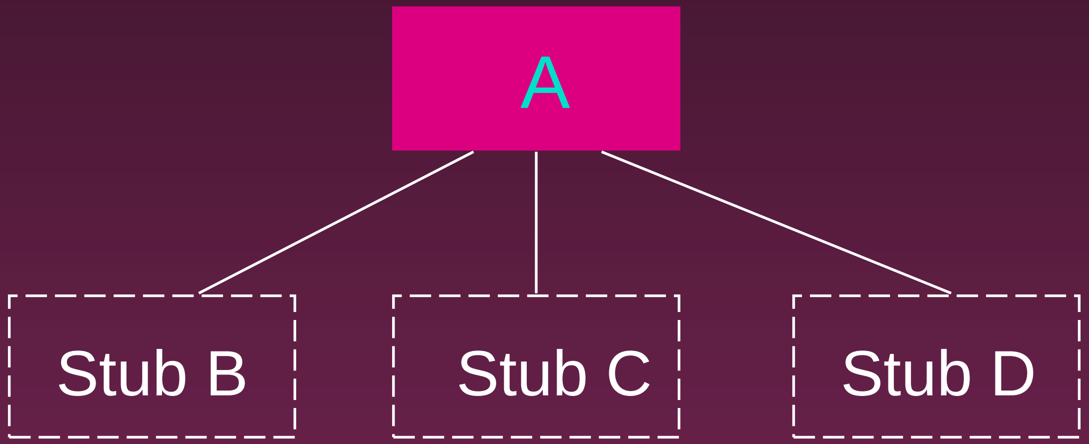
- Begin with the top module in the module call hierarchy
- Stub modules are produced
  - Stubs are often complicated
- The next module to be tested is any module with atleast one previously tested superordinate (calling) module
- After a module has been tested, one of its stubs is replaced by an actual module (the next one to be tested) and its required stubs

Example: Module Hierarchy



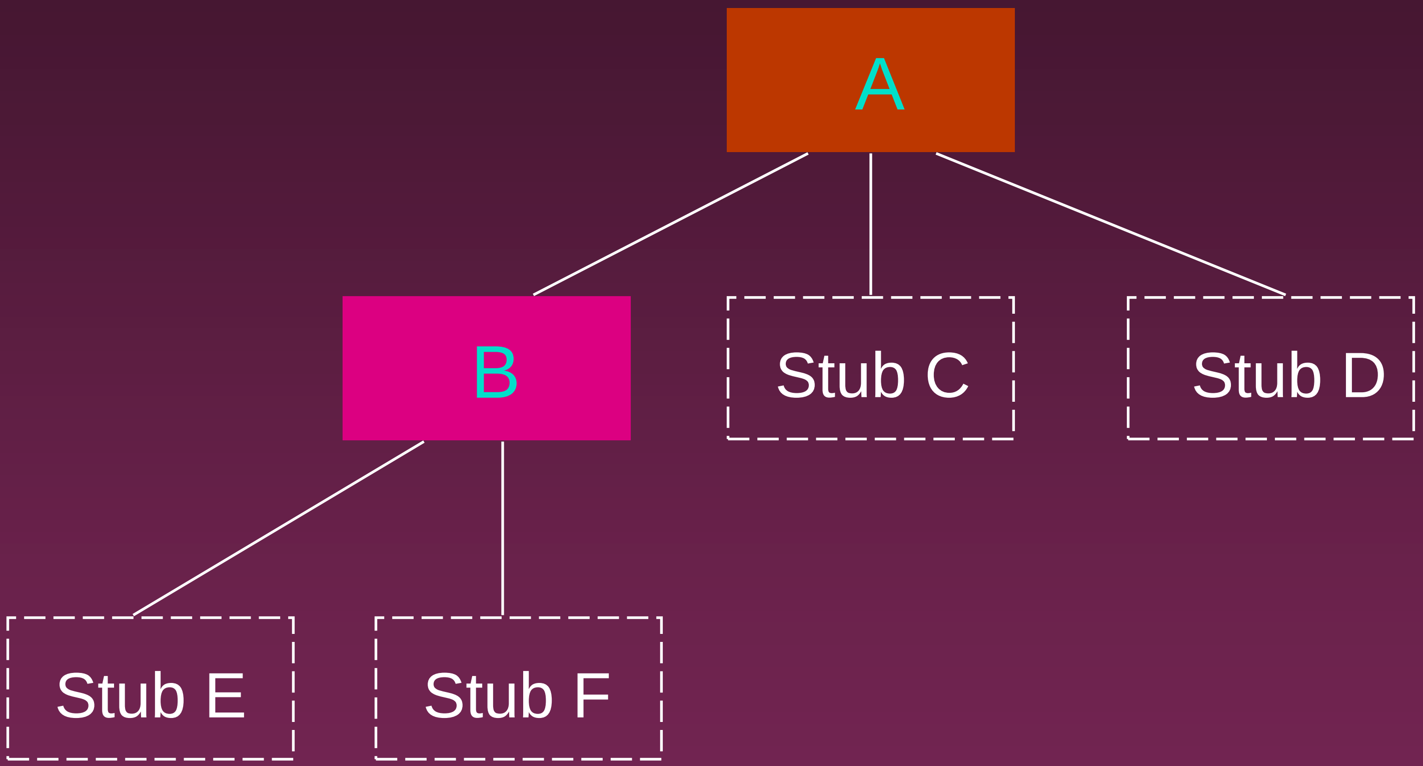
Top-down Integration Testing

Example:



Top-down Integration Testing

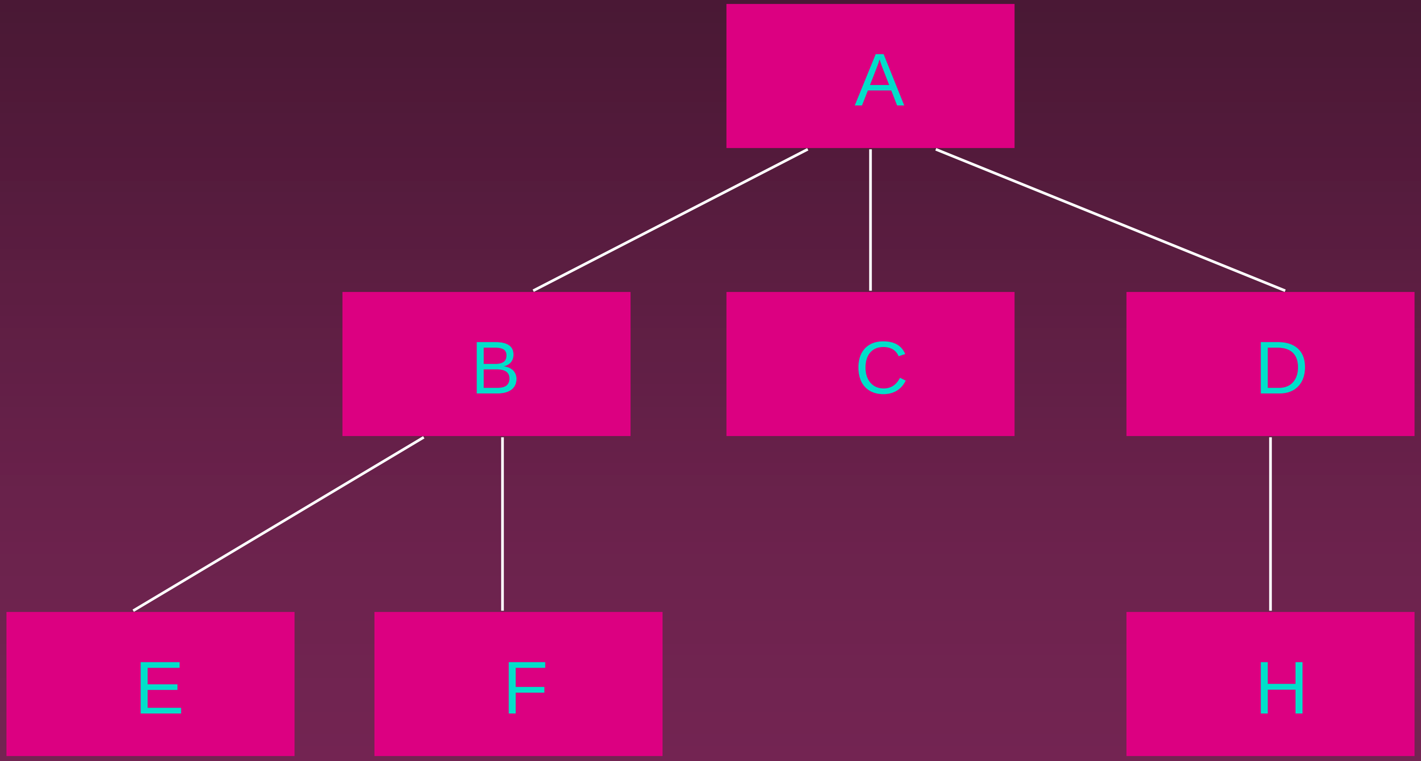
Example:



Bottom-Up Integration

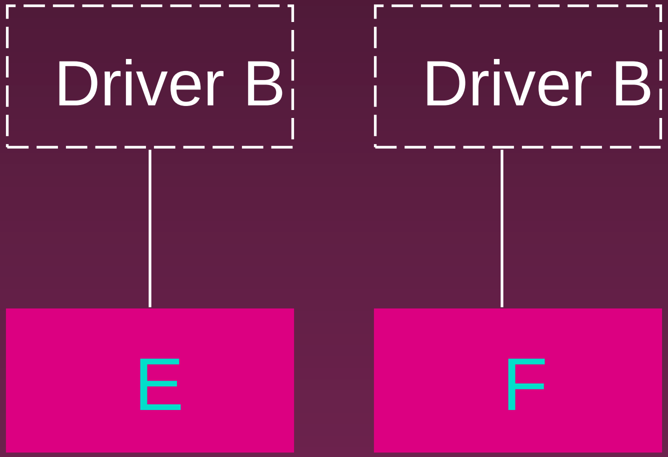
- Begin with the terminal modules (those that do not call other modules) of the modules call hierarchy
- A driver module is produced for every module
- The next module to be tested is any module whose subordinate modules (the modules it calls) have all been tested
- After a module has been tested, its driver is replaced by an actual module (the next one to be tested) and its driver

Example: Module Hierarchy



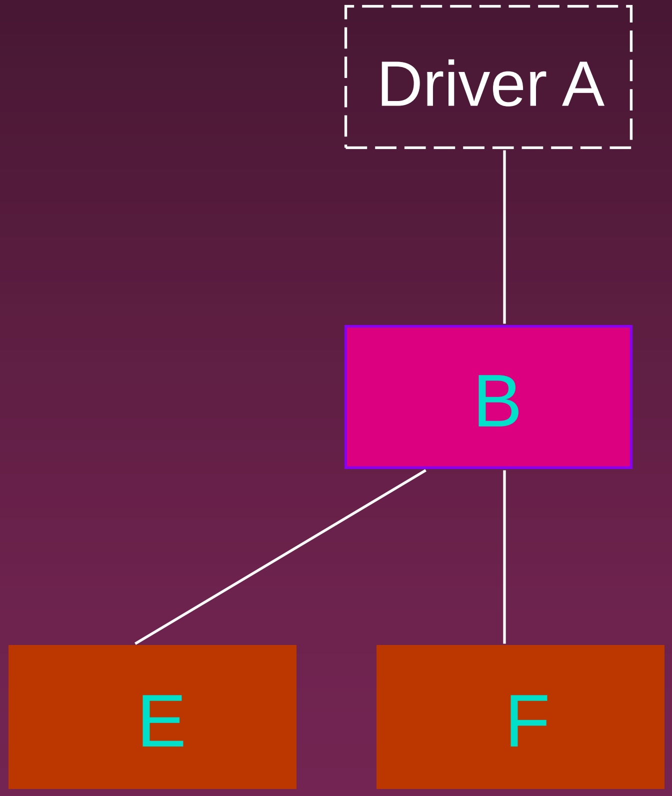
Bottom-Up Integration Testing

Example:



Bottom-Up Integration Testing

Example:



Comparison

Top-down Integration

- Advantage
  - a skeletal version of the program can exist early
- Disadvantage
  - required stubs could be expensive

Bottom-up Integration

- Disadvantage
  - the program as a whole does not exist until the last module is added

No clear winner

- *Effective alternative* -- use Hybrid of bottom-up and top-down
  - prioritize the integration of modules based on risk
  - highest risk functions are integration tested earlier than modules with low risk functions

Function Testing

- Test the complete system with regard to its functional requirements

- Test cases derived from system’s functional specification
  - all black-box methods for test-case design are applicable

System Testing

- Different from Function testing (Non-functional testing)
- Process of attempting to demonstrate that the program or system does not meet its original requirements and objectives as stated in the requirements specification
- Test cases derived from
  - requirements specification
  - system objectives, user documentation

Types of System Tests

- Volume testing
  - to determine whether the program can handle the required volumes of data, requests, etc.
- Load/Stress testing
  - to identify peak load conditions at which the program will fail to handle required processing loads within required time spans (multiprogramming environment) e.g. a web site can handle maxim 1000 user requests at a time
- Usability (human factors) testing
  - to identify discrepancies between the **user interfaces** of a product and the human engineering requirements of its potential users.
- Security Testing
  - to show that the program's security requirements can be threatend

Acceptance Testing

- performed by the Customer or End user
- compare the software to its initial requirements and needs of its end users

Alpha and Beta Testing

Tests performed on a SW Product before its released to a wide user community.

- Alpha testing
   - conducted at the developer’s site by a User
   - tests conducted in a **controlled environment**
- Beta testing
   - conducted at one or more User sites by the end user of the SW
   - it is a **“live”** use of the SW in an environment over which the developer has no control

Mutation testing

- To determining if a set of test data or test cases is useful, by **deliberately introducing** various bugs in the program.
- Re-testing with the original test data/cases to determine if the bugs are detected.
- This is basically to test the test cases

Steps of Mutation process

- **Step 1**: Faults are introduced into the source code of the program by creating many versions called mutants. Each mutant should contain a single fault, and the goal is to cause the mutant version to fail which demonstrates the effectiveness of the test cases.
- **Step 2**: Test cases are applied to the original program and also to the mutant program. A test case should be adequate, and it should able to detect faults in a program.

Cont..

- **Step 3**: Compare the results of original and mutant program.
- **Step 4**: If the original program and mutant programs generate the different output, then that the mutant is killed by the test case. Hence the test case is good enough to detect the change between the original and the mutant program.
- **Step 5**: If the original program and mutant program generate same output, Mutant is kept alive. In such cases , more effective test cases need to be created that kill all mutants.

Regression Testing

- Re-run of previous tests to ensure that SW already tested has not regressed to an earlier error level after **making changes** to the SW.
- This can be done through Impact analysis and Program slicing

- During **maintenance phase,** changes are done in original program P, after changes in P a new version P’ is created.
- How to test P’ for new changes introduced plus impact of these changes on **unmodified part in limited time and resources?**
- For this there should be test set T’ for testing of P’.
  T’ must include -
- Test cases which will test the previous functionality and impact of changes.
- And test cases which will test new changes done.

When to Stop Testing ?

- Stop when the scheduled time for testing expires

- Stop when all the test cases execute without detecting errors

-- both criteria are not good

Better Test Completion Criteria

Base completion on use of specific test-case design methods.

- Example: Test cases derived from
  1) satisfying multi-condition coverage and
  2) boundary-value analysis and
  3) cause-effect graphing and

  all resultant test cases are eventually unsuccessful

When to stop (close) testing ??

- **Most common factors helpful in deciding when to stop the testing are:**
- 1) Stop the Testing when deadlines, like **release deadlines** or **testing deadlines** (allocated time period) have reached,
- 2) Stop the Testing when all the **test cases** have been **completed** with some prescribed **pass percentage.**
- 3) Stop the Testing when the testing **budget** comes to **its end** (for all kind of **resources**).

  4) Stop the Testing when the **code coverage, functional requirements and performance requirements** come to a desired level.
- 5) Stop the Testing when bug rate **drops** below a prescribed level.
- 6) Stop the Testing when the **period** of beta testing / alpha testing gets over.