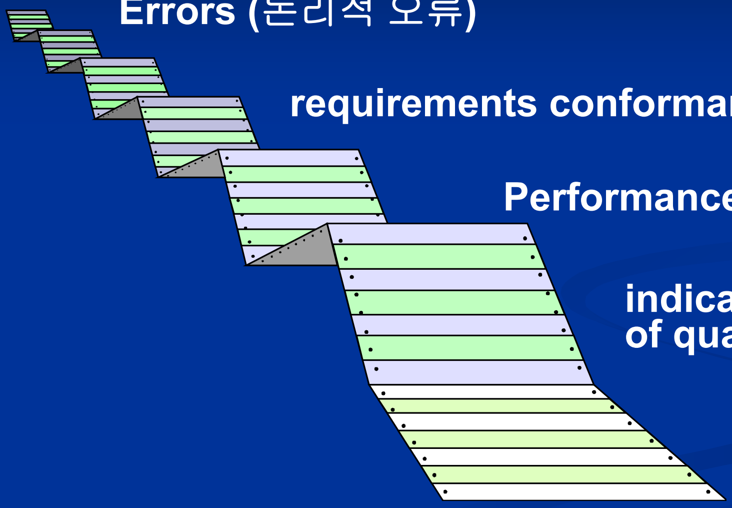
테스팅

한국항공대학교 소프트웨어학과 지승도교수
R.S. Pressman

SW 테스팅

Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

테스팅을 통해 얻을 수 있는 것



Errors (논리적 오류)

requirements conformance (기능 요구)

Performance (성능 요구)

indication
of quality (품질 수준)

누가 테스트해야 할까?

developer

- Understands the system
- but will test "gently"
- and is driven by "delivery"

"Constructive task"

independent tester

- Must learn about the system
- but will attempt to break it
- and is driven by "quality"

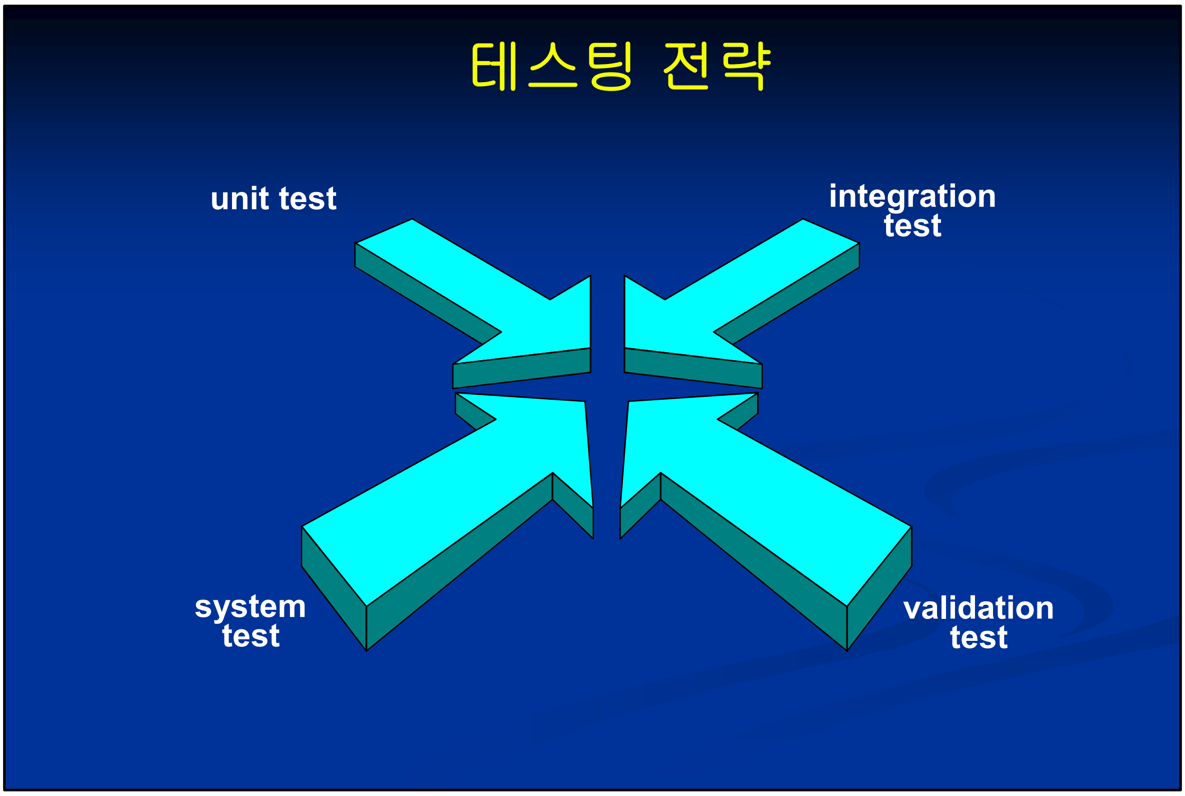
"Destructive task"

전략적 접근

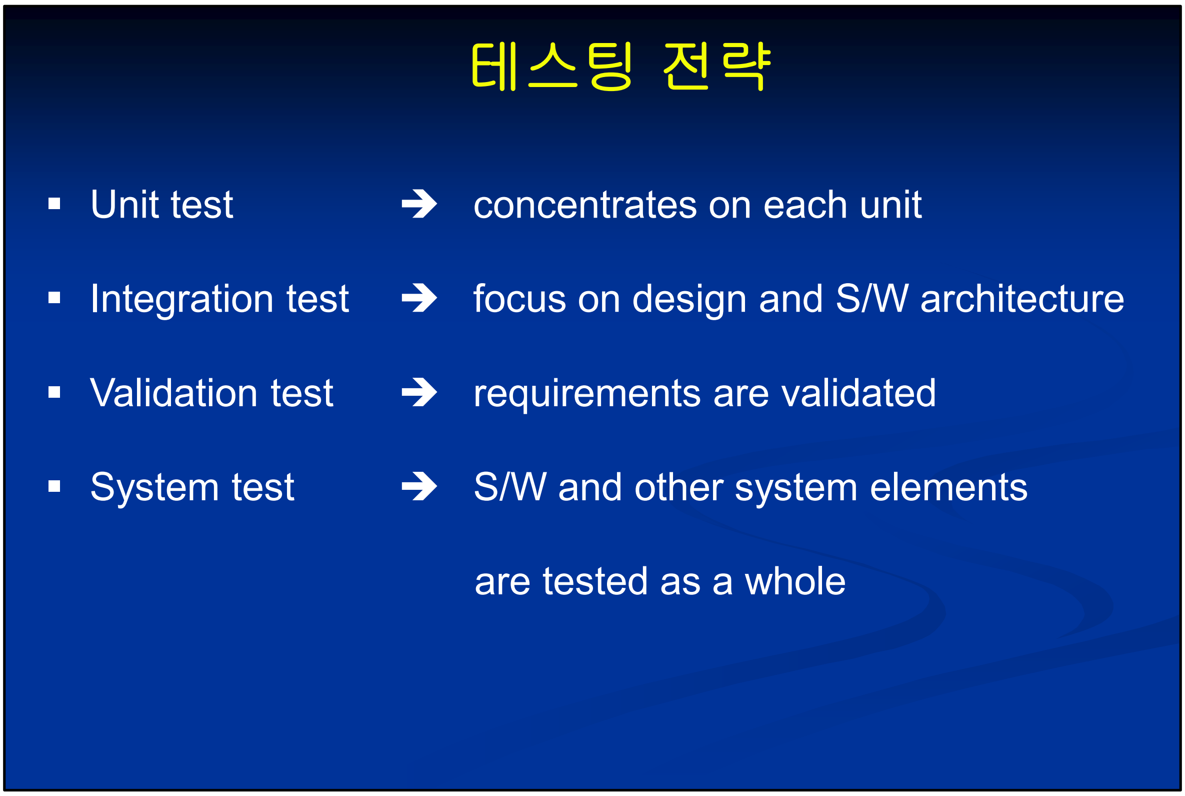
- ✓ Testing is a set of activities that can be planned in advance and conducted systematically.
- ✓ Characteristics;
  - Conduct effective technical reviews
  - Begins at the component level and works “outward”
  - Different testing techniques at different points in time
  - Conducted by developer as well as independent test group
  - Debugging must be accommodated

V&V: Verification & Validation

- ✓ Verification: refers to the set of tasks that ensure that software correctly implements a specific function.
- ✓ Validation: refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.
- ✓ V&V encompass SQA
  - Verification: “Are we building the product right? (논리적 검증)
  - Validation: “Are we building the right product? (유효성 검증)



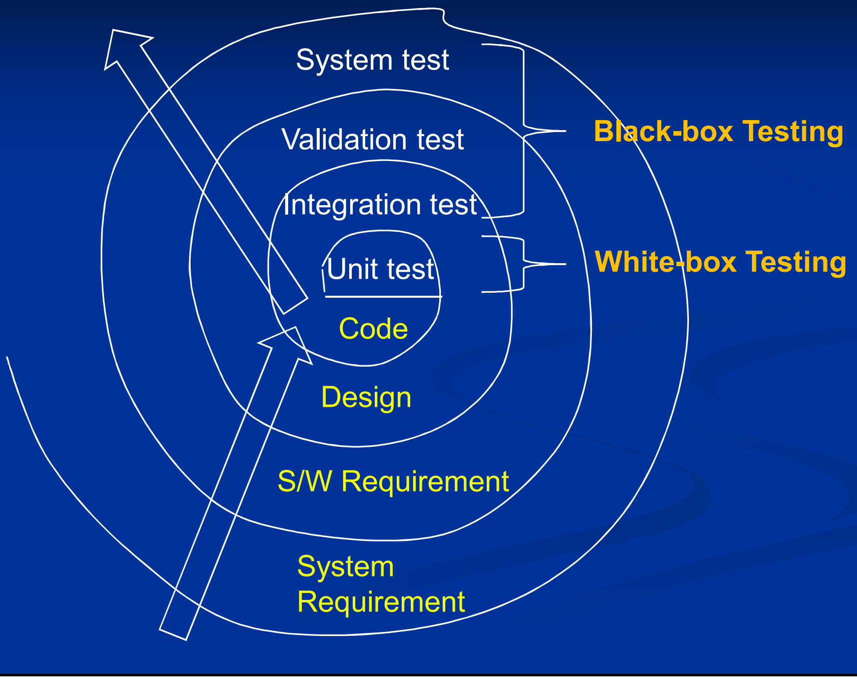
테스팅 전략



테스팅 전략

- Unit test → concentrates on each unit
- Integration test → focus on design and S/W architecture
- Validation test → requirements are validated
- System test → S/W and other system elements are tested as a whole

테스팅 전술 및 전략



테스팅 전략 (계속)

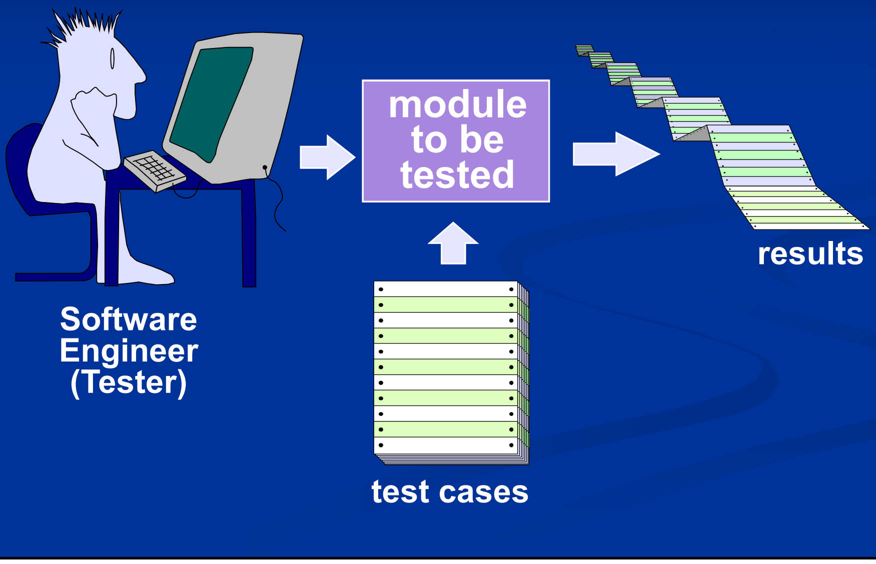
- We begin by ‘testing-in-the-small’ and move toward ‘testing-in-the-large’
- For conventional software
  - The module (component) is our initial focus
  - Integration of modules follows
- For OO software
  - our focus when “testing in the small” changes from an individual module (the conventional view) to an OO class that encompasses attributes and operations and implies communication and collaboration

테스팅 전략 요령

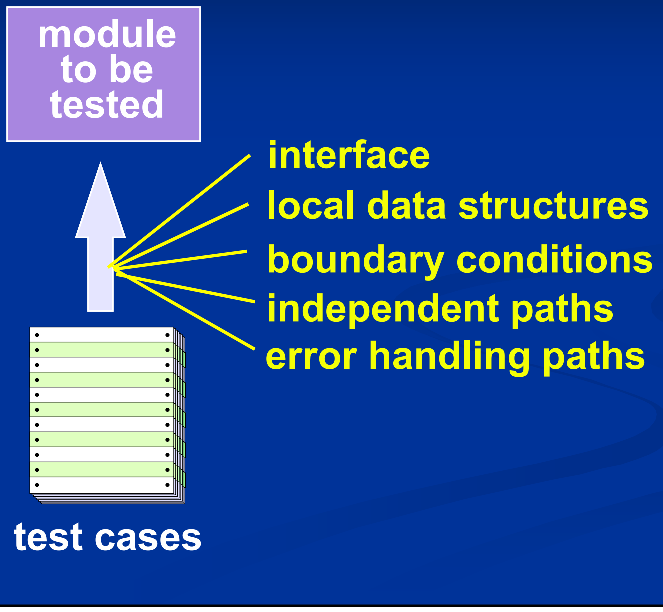
- Specify product requirements in a quantifiable manner long before testing commences.
- State testing objectives explicitly.
- Understand the users of the software and develop a profile for each user category.
- Develop a testing plan that emphasizes “rapid cycle testing.”
- Build “robust” software that is designed to test itself
- Use effective formal technical reviews as a filter prior to testing
- Conduct formal technical reviews to assess the test strategy and test cases themselves.
- Develop a continuous improvement approach for the testing process.

Unit Testing

➔ Focus verification of smallest unit of software



Unit Testing



module to be tested

interface

local data structures

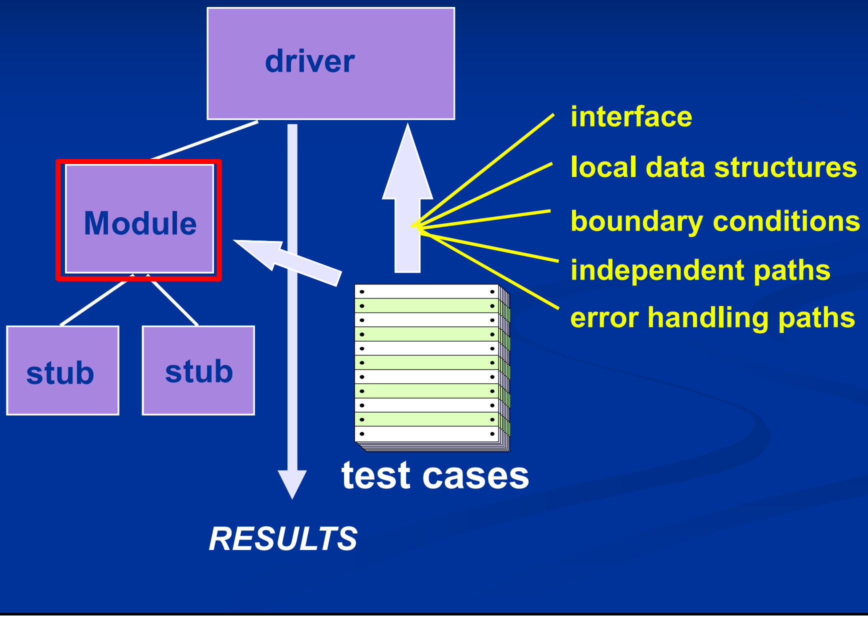
boundary conditions

independent paths

error handling paths

test cases

Unit Test Environment



driver

interface

local data structures

boundary conditions

independent paths

error handling paths

Module

stub

stub

test cases

RESULTS

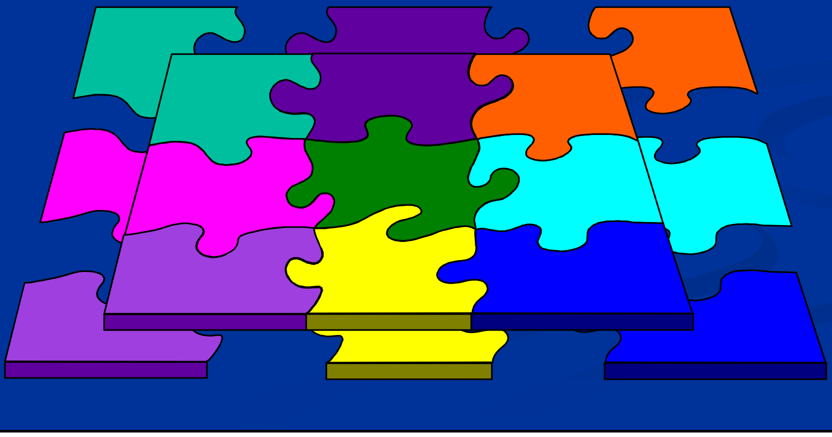
Driver vs. Stub

- Driver: main program that accepts test case data, passes such data to the component to be test, and print relevant results.
- Stub: It serves to replace modules that are subordinate invoked by the component to be test. It have to do minimal data manipulation, print verification of entry, and returns control to the module undergoing testing.

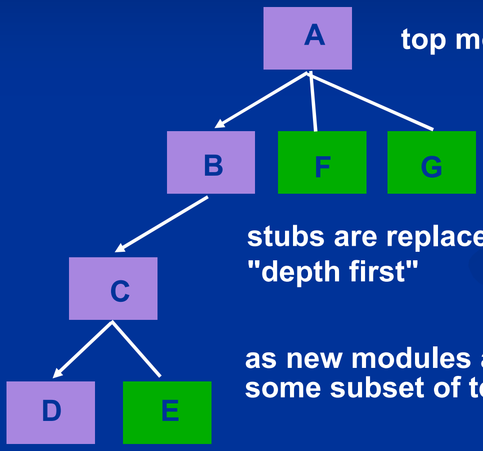
Integration Testing Strategies

Options:

- **a "big bang" strategy**
- **an incremental construction strategy**



Top-down Integration

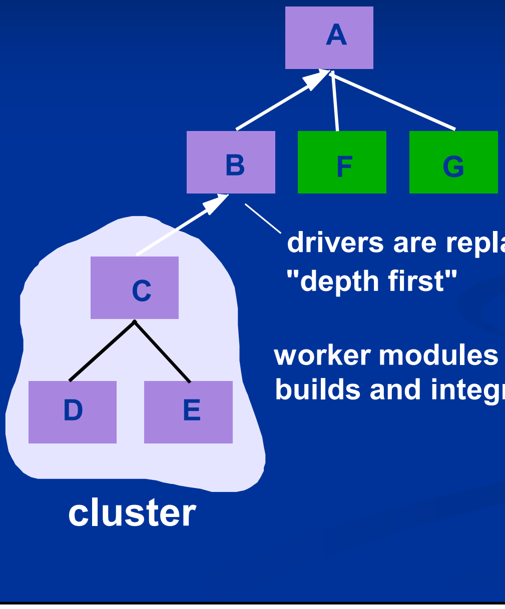


top module is tested with stubs

stubs are replaced one at a time
"depth first"

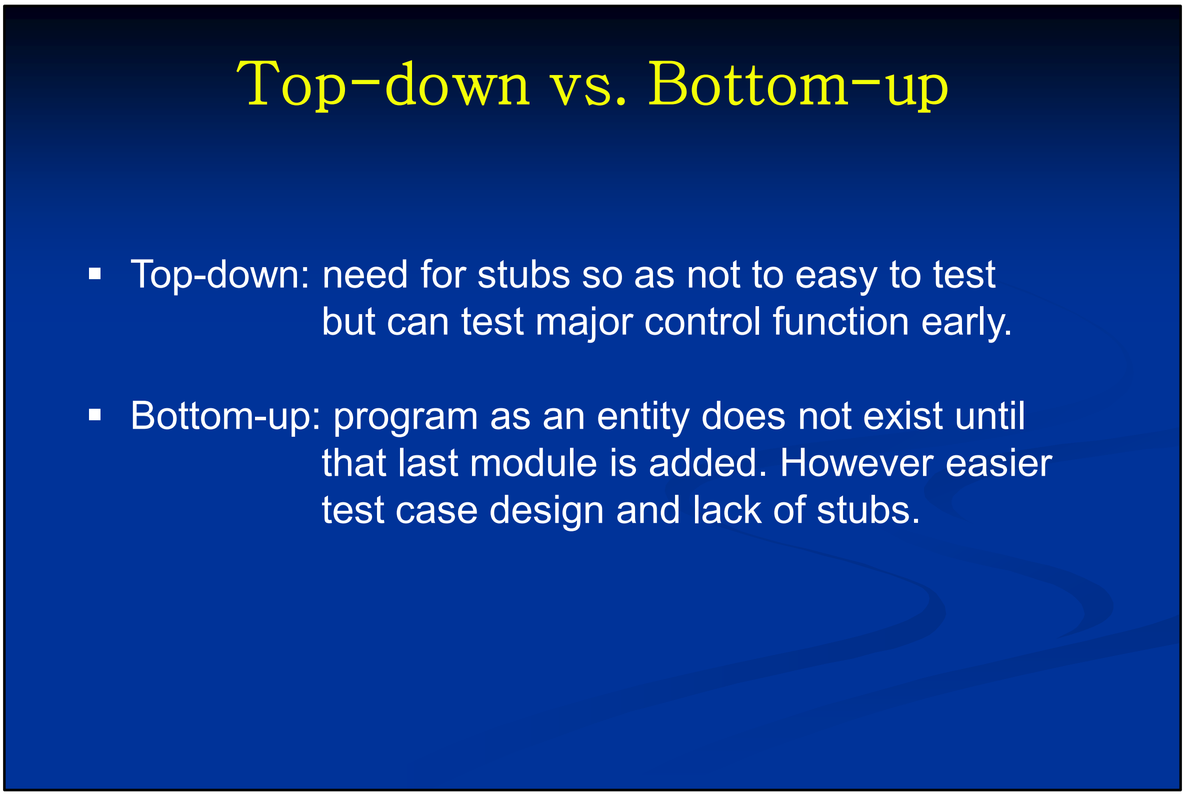
as new modules are integrated,
some subset of tests is re-run

Bottom-up Integration



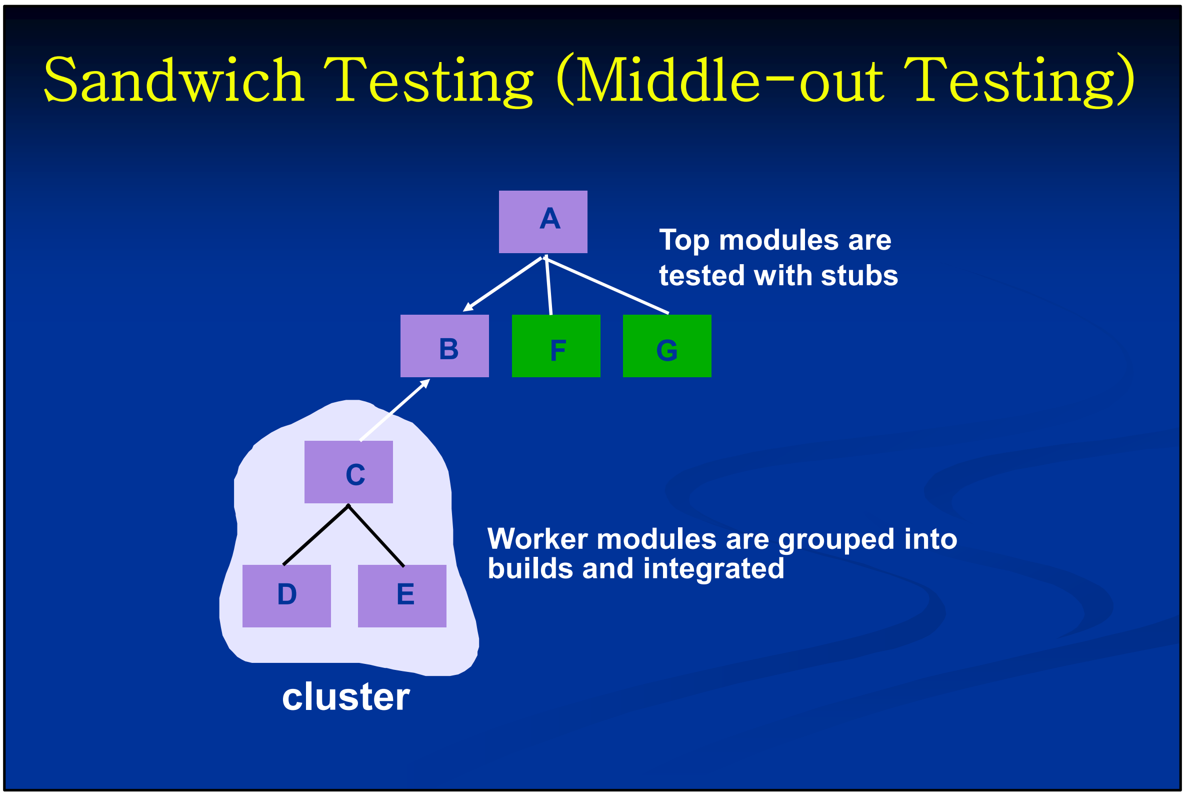
drivers are replaced one at a time
"depth first"

worker modules are grouped into
builds and integrated



Top-down vs. Bottom-up

- Top-down: need for stubs so as not to easy to test but can test major control function early.
- Bottom-up: program as an entity does not exist until that last module is added. However easier test case design and lack of stubs.



Sandwich Testing (Middle-out Testing)

Top modules are tested with stubs

Worker modules are grouped into builds and integrated

cluster

OOT Strategy

- class testing is equivalent of unit testing
  - operations within the class are tested
  - the state behavior of the class is examined
- integration applied three different strategies
  - thread-based testing—integrates the set of classes required to respond to one input or event
  - use-based testing—integrates the set of classes required to respond to one use case
  - cluster testing—integrates the set of classes required to demonstrate one collaboration

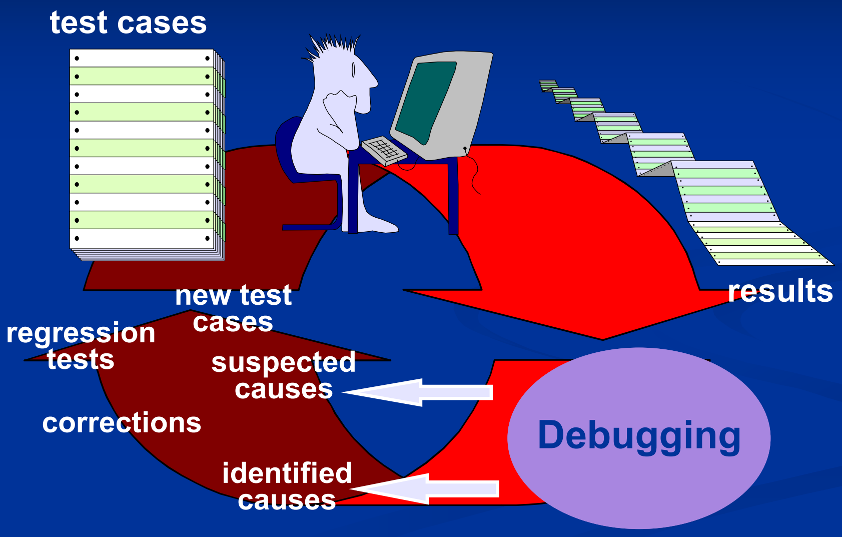
High Order Testing

- Validation testing
  - → Focus is on software requirements
- System testing
  - → Focus is on system integration
- Alpha/Beta testing
  - → Focus is on customer usage
- Recovery testing
  - → forces the software to fail in a variety of ways and verifies that recovery is properly performed
- Security testing
  - → verifies that protection mechanisms built into a system will, in fact, protect it from improper penetration
- Stress testing
  - → executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- Performance Testing
  - → test the run-time performance of software within the context of an integrated system

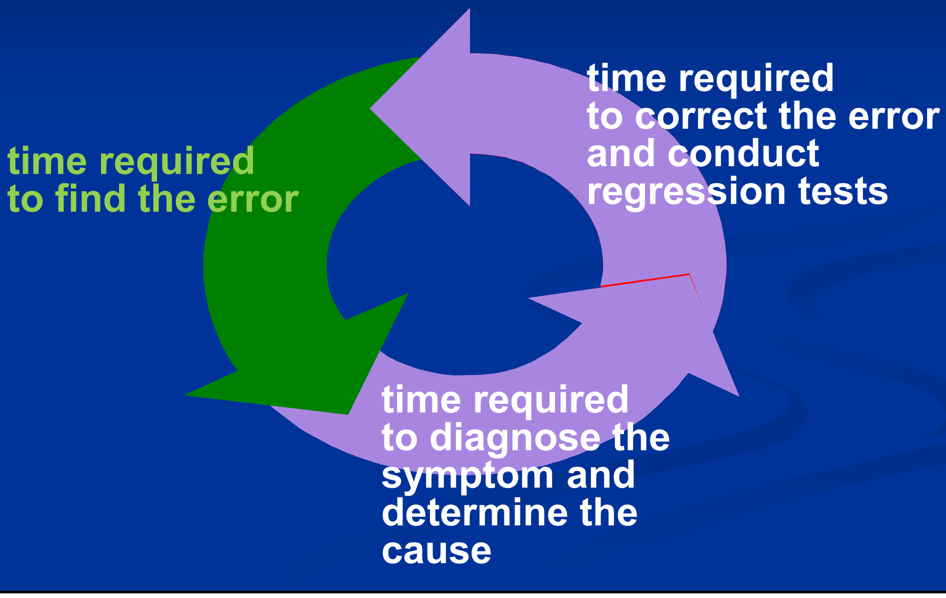
디버깅:
A Diagnostic Process

- Occurs as consequence of successful testing
- Poorly understood mental process that connect a symptom to a cause

디버깅 절차



Debugging Effort

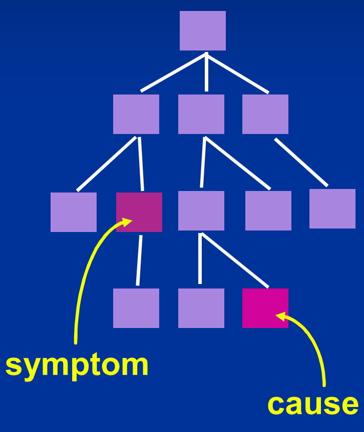


time required to find the error

time required to correct the error and conduct regression tests

time required to diagnose the symptom and determine the cause

Symptoms & Causes



symptom

cause

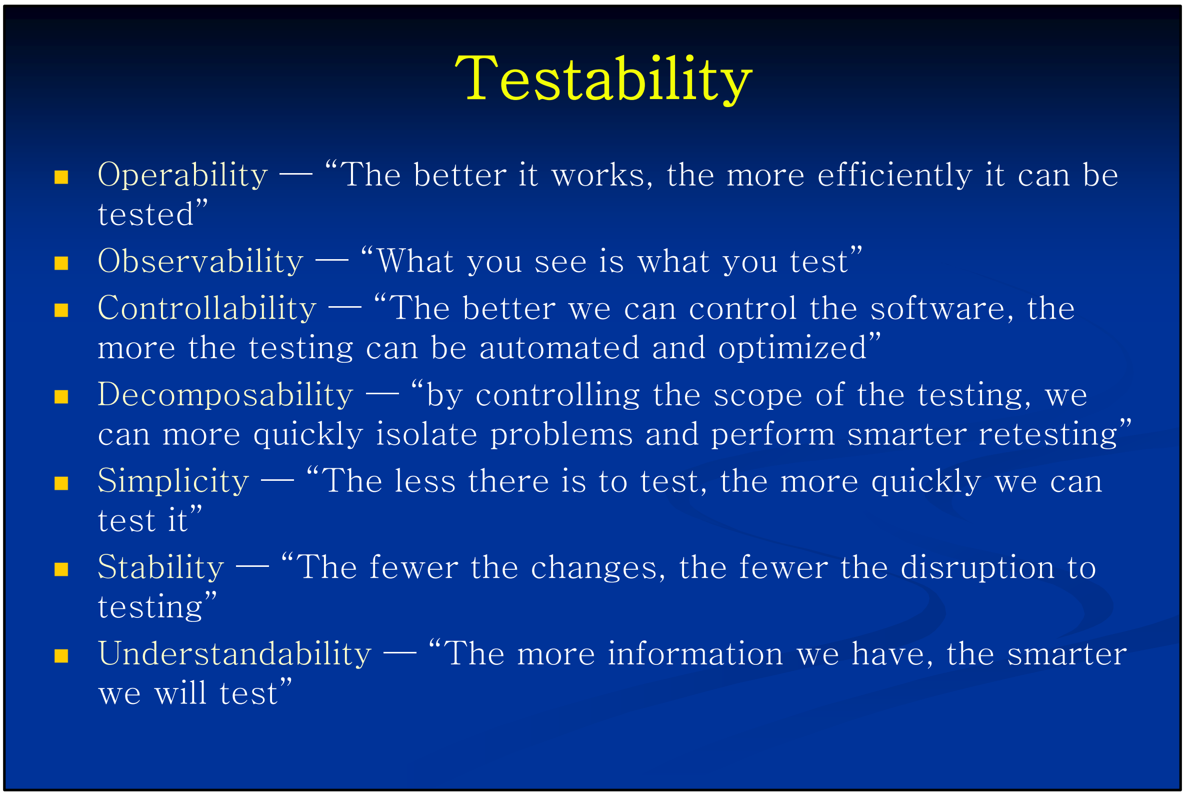
- symptom and cause may be geographically separated
- symptom may disappear when another problem is fixed
- cause may be due to a combination of non-errors
- cause may be due to a system or compiler error
- cause may be due to assumptions that everyone believes
- symptom may be intermittent

Debugging Techniques

- □ brute force testing
- □ backtracking
- □ cause elimination

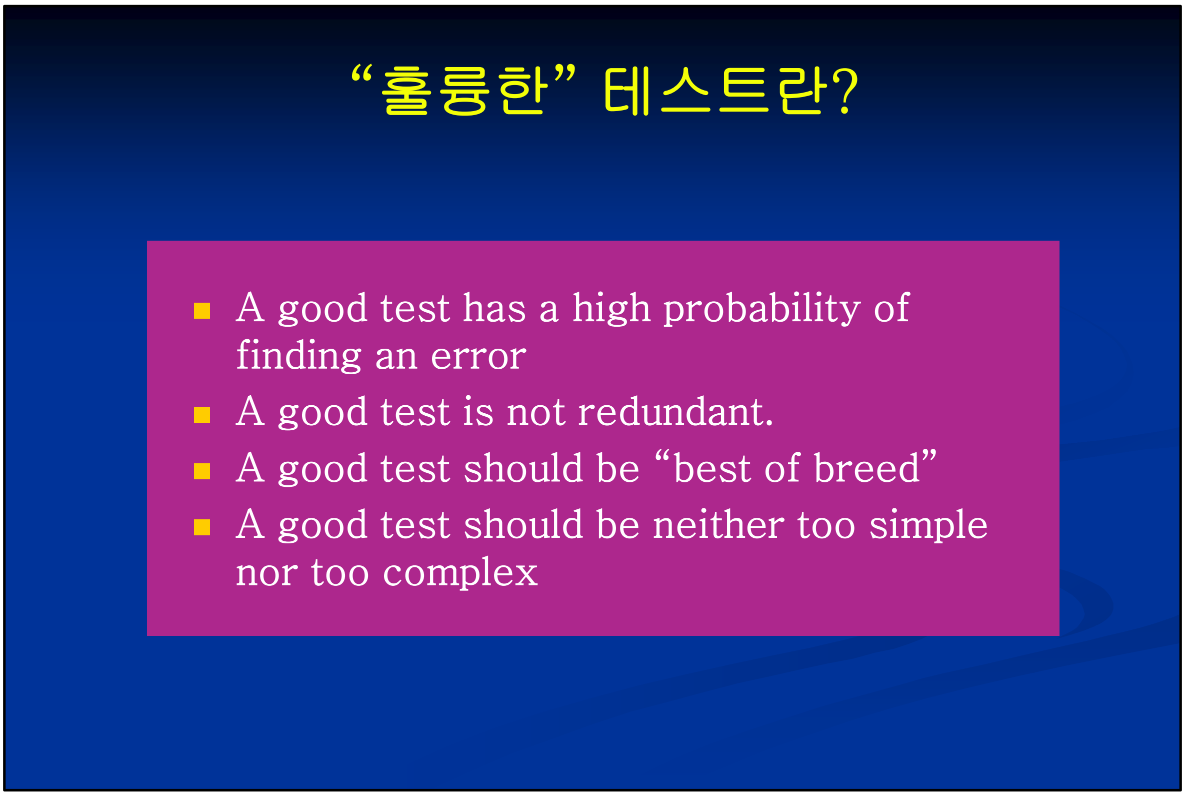
Correcting the errors

1. Reproduced in another part?
2. Next bug?
3. Prevent in first place?



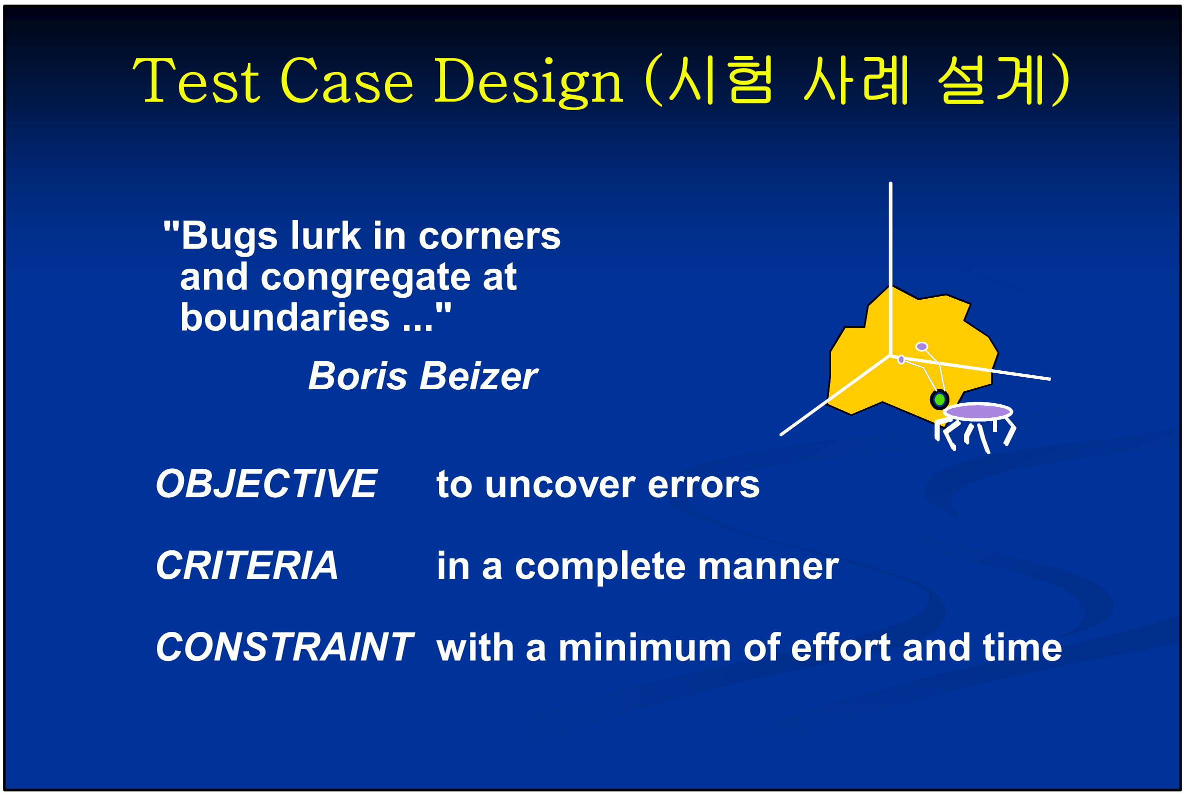
Testability

- Operability — “The better it works, the more efficiently it can be tested”
- Observability — “What you see is what you test”
- Controllability — “The better we can control the software, the more the testing can be automated and optimized”
- Decomposability — “by controlling the scope of the testing, we can more quickly isolate problems and perform smarter retesting”
- Simplicity — “The less there is to test, the more quickly we can test it”
- Stability — “The fewer the changes, the fewer the disruption to testing”
- Understandability — “The more information we have, the smarter we will test”



“훌륭한” 테스트란?

- A good test has a high probability of finding an error
- A good test is not redundant.
- A good test should be “best of breed”
- A good test should be neither too simple nor too complex



Test Case Design (시험 사례 설계)

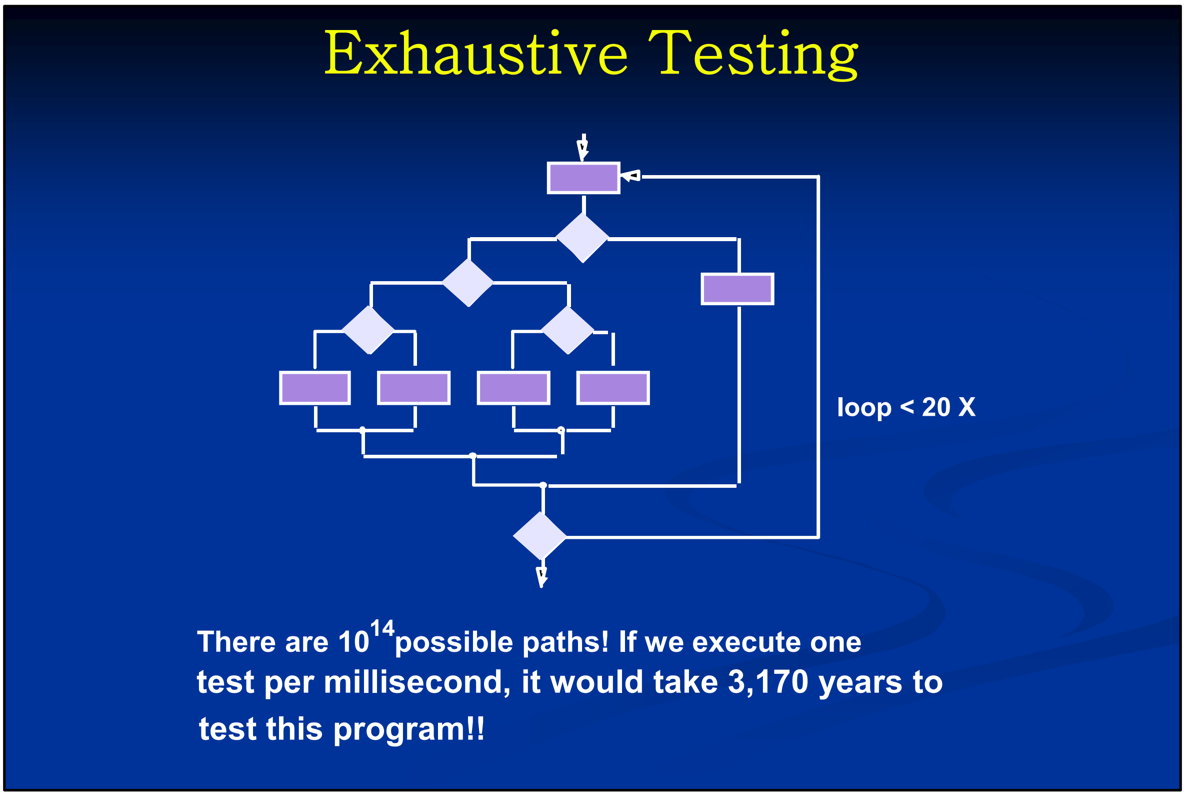
"Bugs lurk in corners
and congregate at
boundaries ..."

Boris Beizer

OBJECTIVE to uncover errors

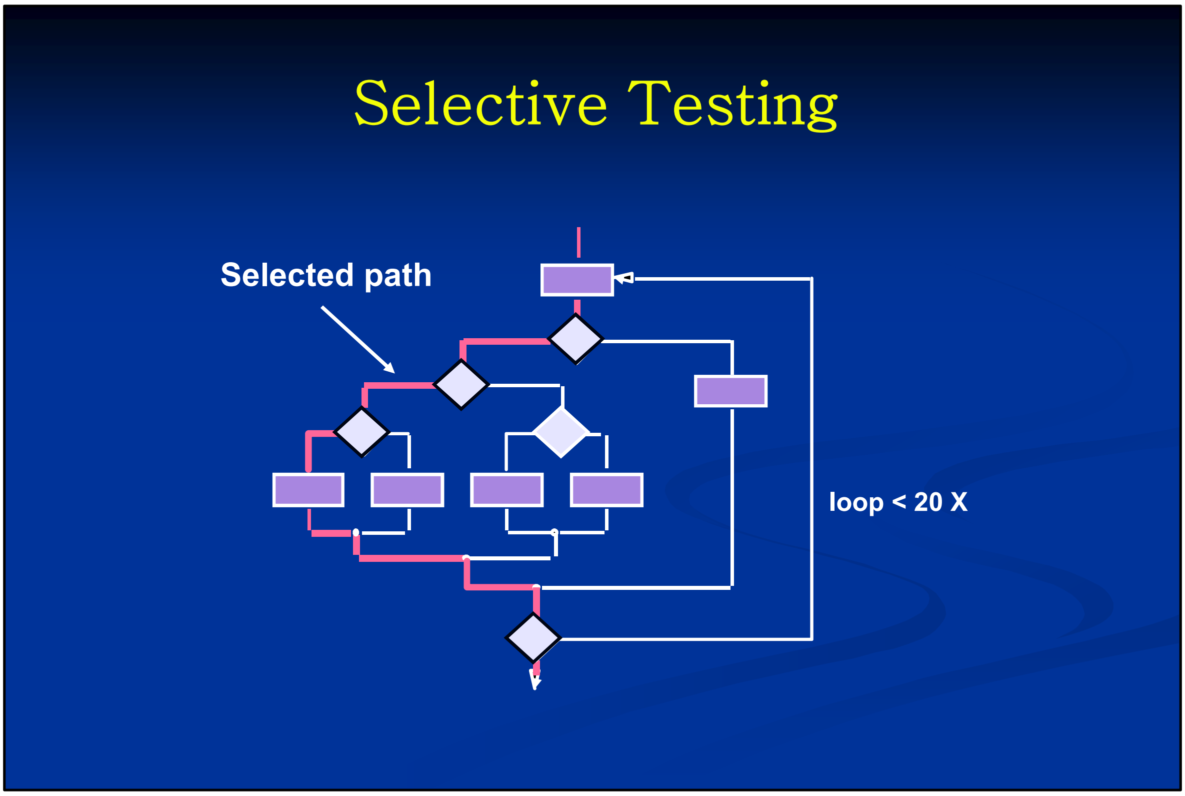
CRITERIA in a complete manner

CONSTRAINT with a minimum of effort and time



Exhaustive Testing

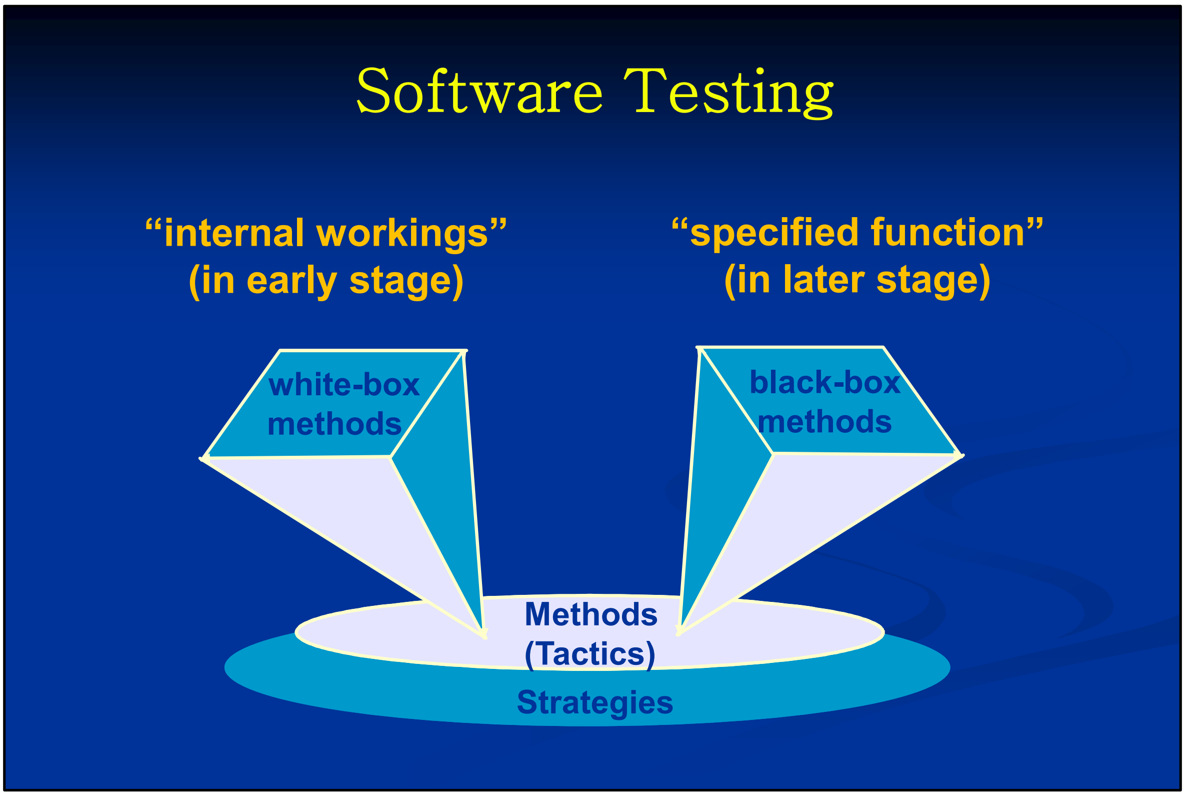
There are 10^{14} possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!



Selective Testing

Selected path

loop < 20 X



Software Testing

“internal workings”
(in early stage)

“specified function”
(in later stage)

white-box
methods

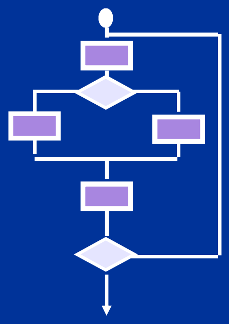
black-box
methods

Methods
(Tactics)

Strategies

White-Box Testing

- Glass-box testing



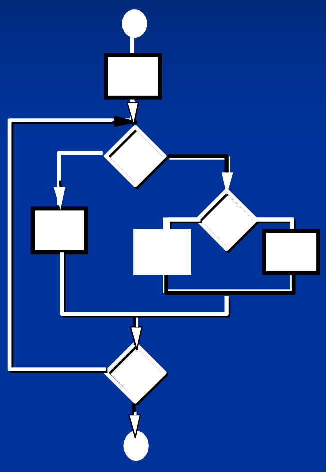
(1) Independent path

(2) Logical decision

(3) All loop

(4) Data structure

Basis Path Testing



First, we compute the cyclomatic complexity:

✓ number of simple decisions + 1

or

✓ number of enclosed areas + 1

In this case, V(G) = 4

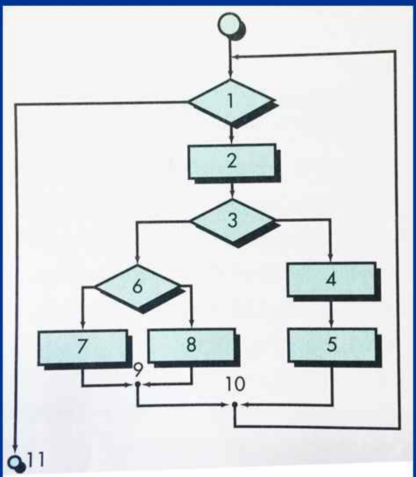
Basis Path Testing

White-box testing technique proposed by Tom McCabe.

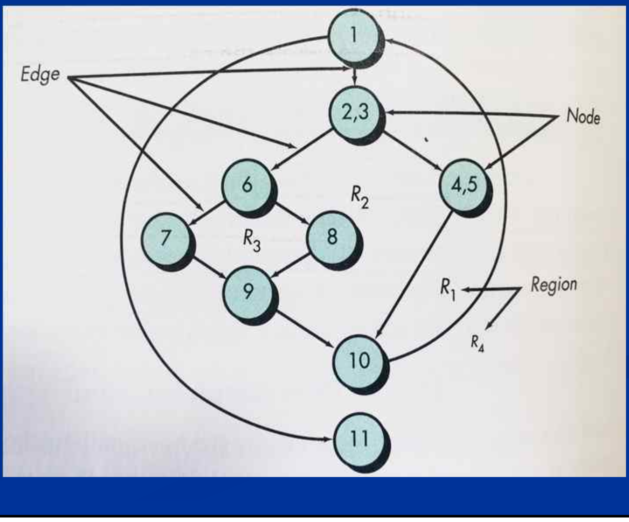
It enables the test-case designer to derive a logical complexity measure of a procedural design and use this measures as a guide for designing a basis set of execution path.

Basis Path Testing

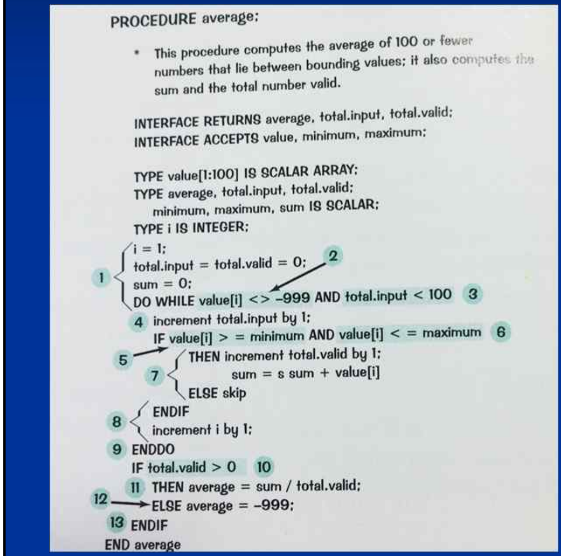
Flow Chart



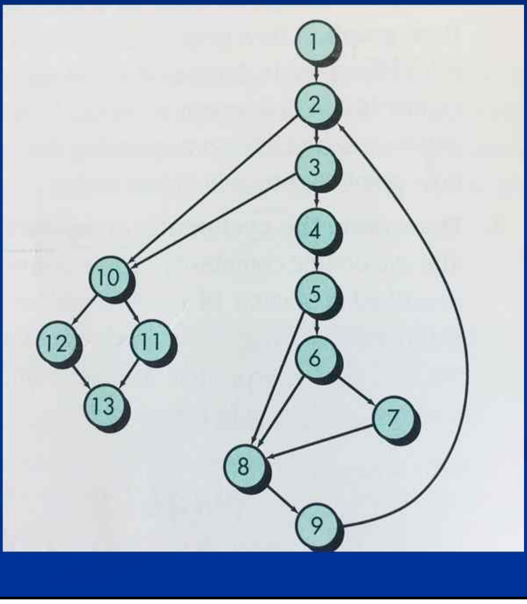
Flow Graph



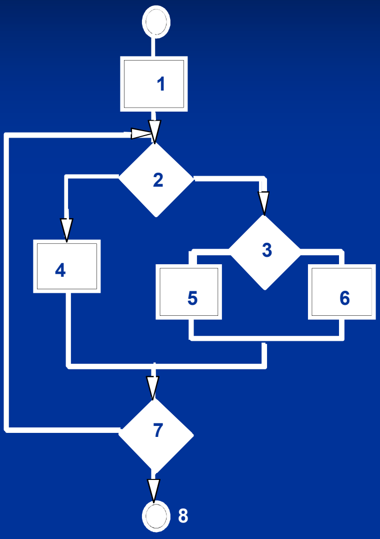
Basis Path Testing



```
PROCEDURE average;

    *  This procedure computes the average of 100 or fewer
       numbers that lie between bounding values; it also computes the
       sum and the total number valid.

   INTERFACE RETURNS average, total.input, total.valid;
   INTERFACE ACCEPTS value, minimum, maximum;

   TYPE value[1:100] IS SCALAR ARRAY;
   TYPE average, total.input, total.valid;
       minimum, maximum, sum IS SCALAR;
   TYPE i IS INTEGER;
     ⎧i = 1;
     ⎪total.input = total.valid = 0;     2
  1 <  sum = 0;
     ⎩DO WHILE value[i] <> -999 AND total.input < 100   3
   4 increment total.input by 1;
      IF value[i] > = minimum AND value[i] < = maximum   6
  5 →     ⎧THEN increment total.valid by 1;
       7 <          sum = s sum + value[i]
          ⎩ELSE skip
    8  ⎧ENDIF
      <  increment i by 1;
  9 ENDDO
    IF total.valid > 0   10
    11 THEN average = sum / total.valid;
 12 → ELSE average = -999;
 13 ENDIF
END average
```



Basis Path Testing



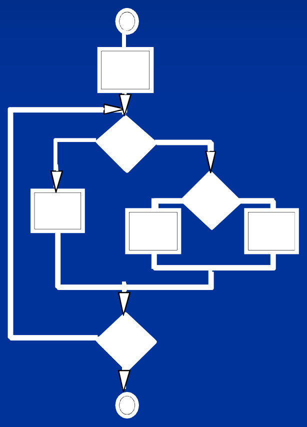
Next, we derive the independent paths:

Since V(G) = 4, there are four paths

Path 1: 1,2,3,6,7,8
Path 2: 1,2,3,5,7,8
Path 3: 1,2,4,7,8
Path 4: 1,2,4,7,2,4,7,8

Finally, we derive test cases to exercise these paths

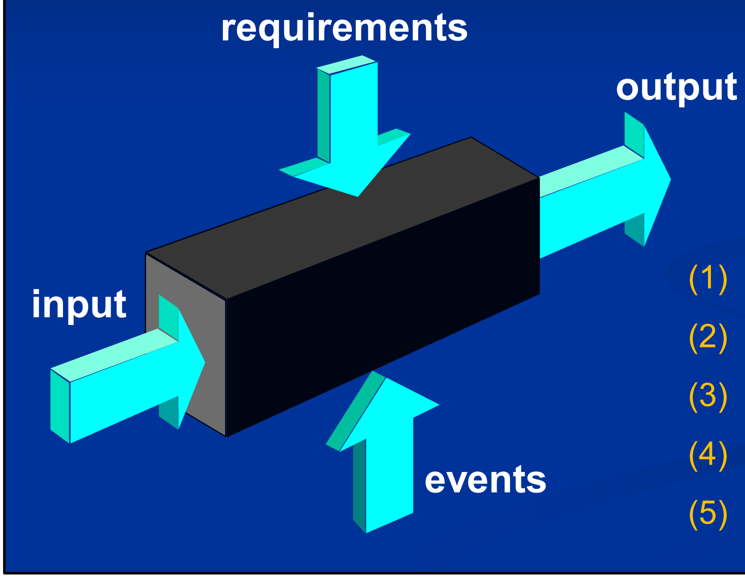
Deriving Test cases



1. Draw flow graph
2. Determine cyclomatic complexity V(G)
3. Determine independent paths
4. Prepare test cases

Black-Box Testing

- Behavioral testing



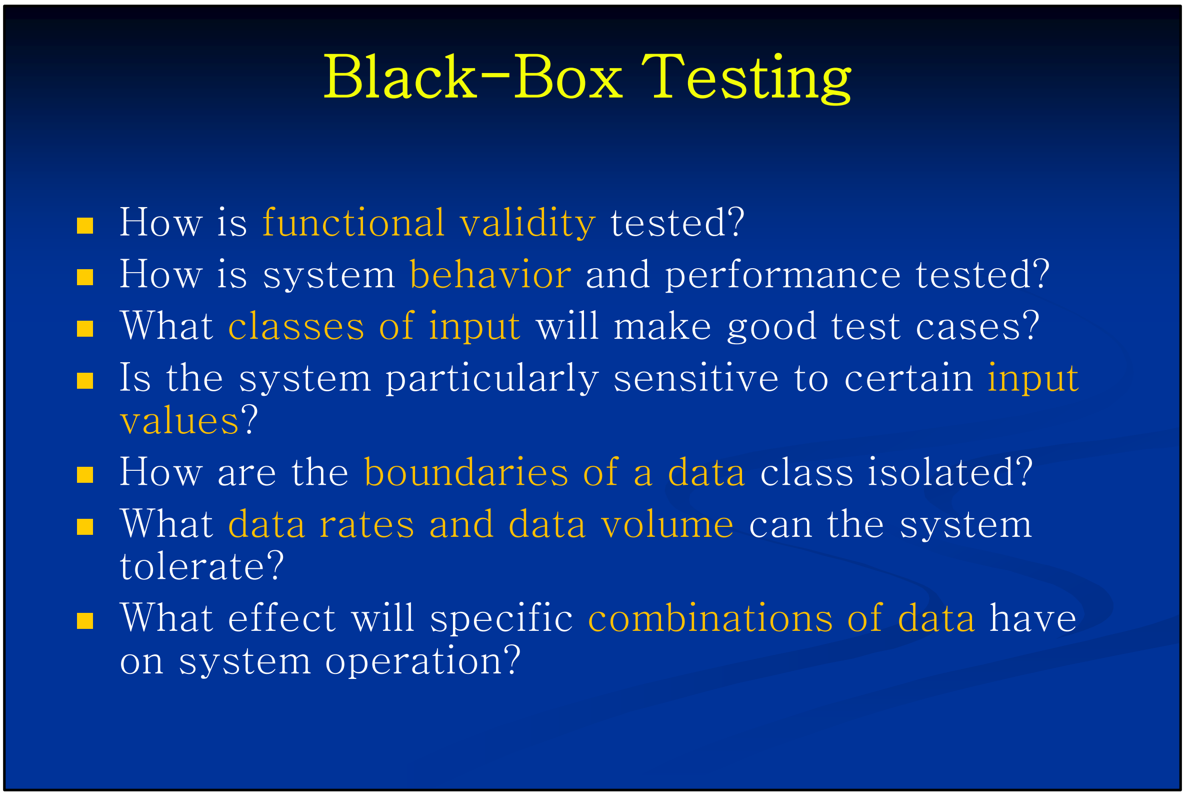
(1) Incorrect/missing function

(2) Interface errors

(3) Data structure errors

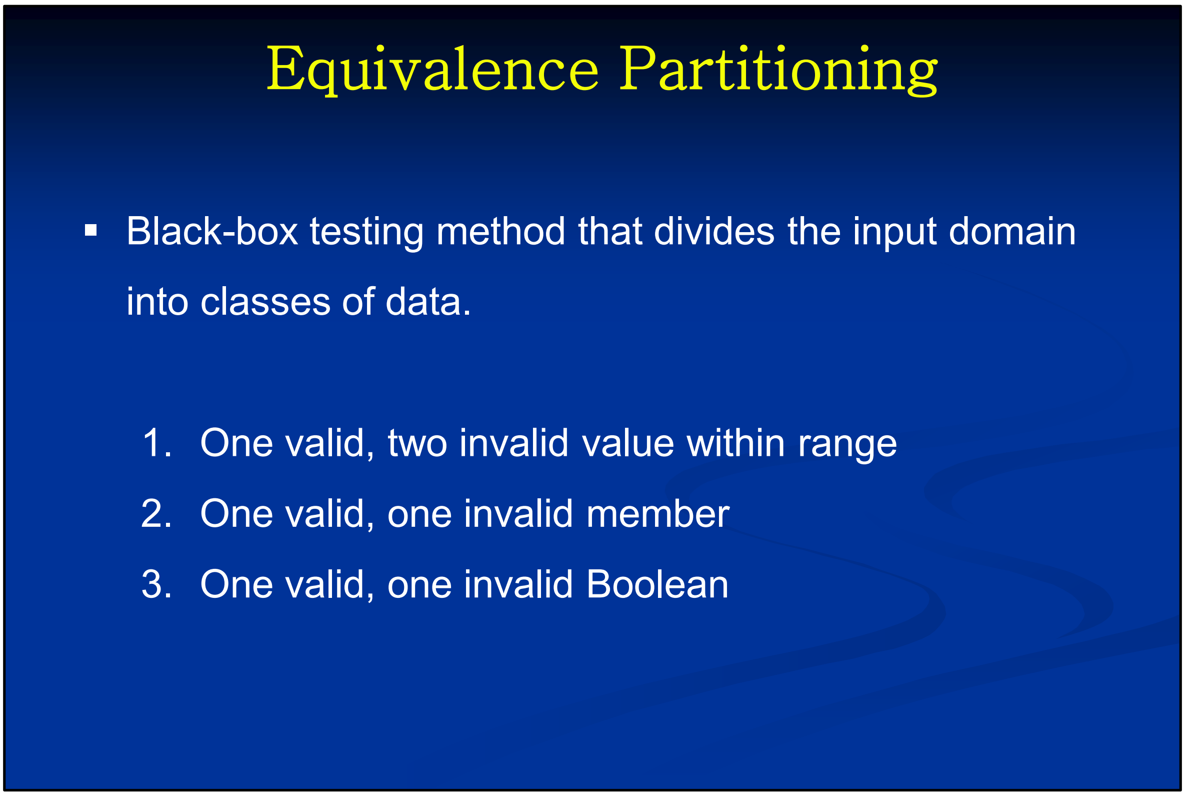
(4) Behavioral errors

(5) Initiation/termination errors



Black-Box Testing

- How is functional validity tested?
- How is system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?



Equivalence Partitioning

- Black-box testing method that divides the input domain into classes of data.

1. One valid, two invalid value within range
2. One valid, one invalid member
3. One valid, one invalid Boolean

Boundary Value Analysis

- Greater number of errors occurs at the boundaries of the input domain rather than in the “center”
- compliments Equivalence Partitioning

1. Value a and b and just above and below a and b
2. Min and max number, just above and below min & max
3. Apply 1, 2 to output
4. Data structure boundary