# Chapter 16
# Software Testing Techniques

## SW Testing Techniques

- SW Testing
  - is critical element of SW quality assurance
  - presents the ultimate review of specification, design and coding
- It is not unusual for a SW development organization to expend btw 30 and 40 percent of total project effort on testing
- Testing is a destructive process rather than constructive

## Testing Objectives

1. Testing is the process of executing a program with the intent of finding errors
2. A good test case is one that has a high probability of finding an as-yet discovered errors
3. A successful test is one that uncovers an as-yet undiscovered error.

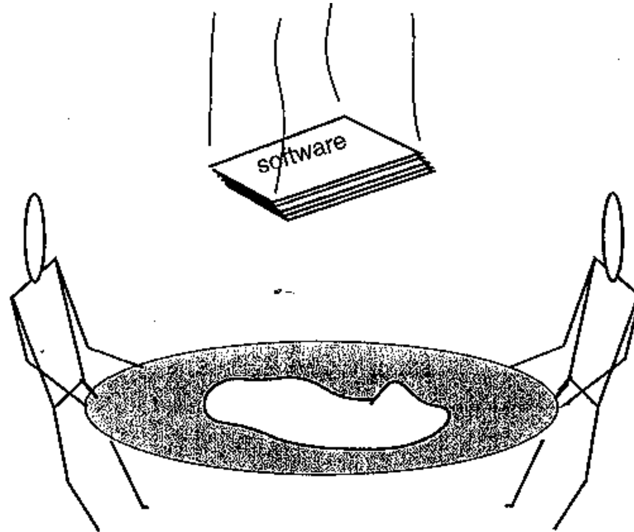
Our objective is to design tests that systematically uncover different classes of errrors and do so with a minimum amount of time and effort.

Testing cannot show the absence of defects, it can only show that SW errors are present.

## Testing Principles

- All tests should be traceable to customer requirements
- Tests should be planned long before testing beins. (with requirement model)
- *The Pareto* principle applies to SW testing: 80 % of all errors uncovered during testing will likely be traceable to 20 % of all program modules.
- Testing should begin “in the small” and progress toward testing “in the large”. Modules →clusters →entire system
- Exhaustive testing is not possible. Huge number of combinations
- Testing should be conducted by an independent third party.

# Software Quality & Testing




# Software Testing

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

Glen Myers

# Testability

- **operability-** it operates cleanly
- **observability-** the results are easy to see
- **controllability-** processing can be controlled
- **decomposability-** testing can be targeted
- **simplicity-** no complex architecture and logic
- **stability-** few changes are requested during testing
- **Understandability** – test design is well unserstood

# Who Tests the software?

developer

understands the system but, will test “gently” and, is driven by “delivery”

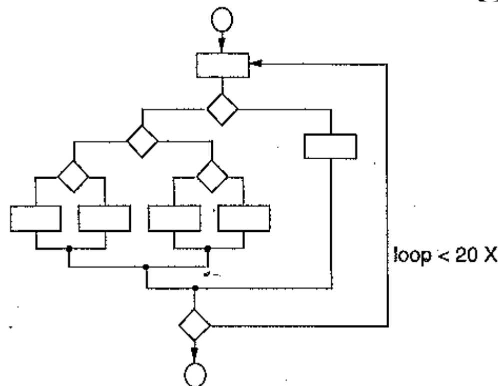
independent tester

must learn about the system, but, will attempt to break it and, is driven by equality

# A good test

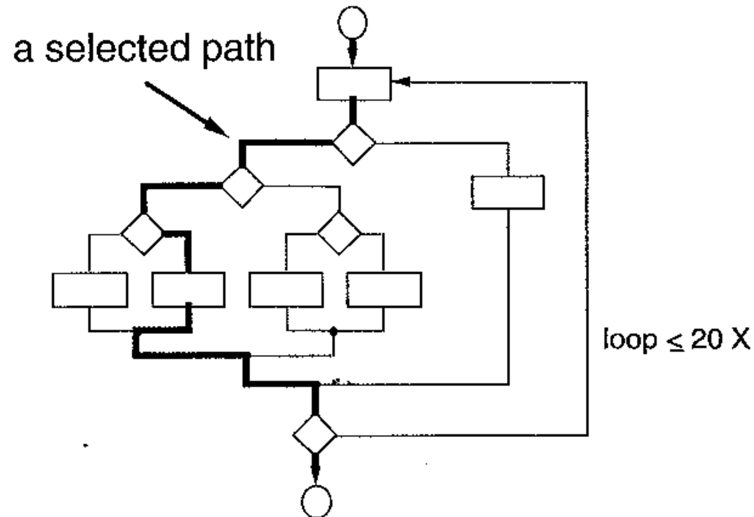
- Has a high probability of finding an error
- is not redundant – testing time and resources are limited
- Should be a “best of breed” – the test that has the highest likelihood of uncoveringa whole class of errors should be used
- Should be neither too simple nor too complex

# Exhaustive Testing



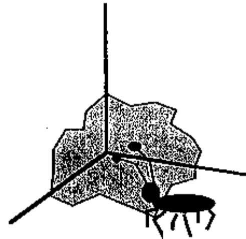

There are $10^{14}$ possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!!

# Selective Testing




# Test Case Design

"Bugs lurk in corners
and congregate at
boundaries ..."
*Boris Beizer*

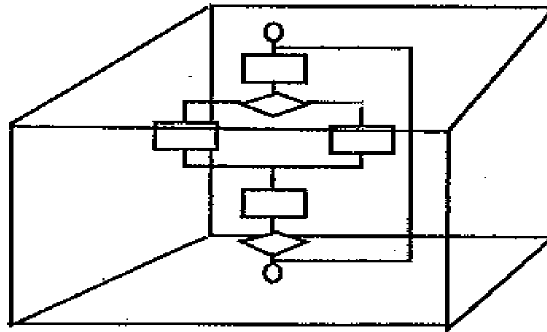


| | |
|---|---|
| OBJECTIVE | to uncover errors |
| CRITERIA | in a complete manner |
| CONSTRAINT | with a minimum of effort and time |

# Test Case Design

- Black-box testing
  - Conducted at the SW interface
  - Used to demonstrate that Sw functions are operational; that input is properly accepted and output is correctly produced, and thar the integrity of the SW is maintained
- White-box testing
  - close examination of procedural details.
  - Logical paths through the SW is tested by providing test cases that exercise sprecific sets of conditions andor loops.

# White-Box Testing



... Our goal is to ensure that all statements and conditions have been executed at least once ...

## White-Box Testing methods

1. Guarantee that all *independent paths* within a module have been exercise at least once
2. Exercise all logical decisions on their *true* and *false* sides
3. Execute all loops at their boundaries and within their operational bounds
4. Exercise internal data structures to assure their validity

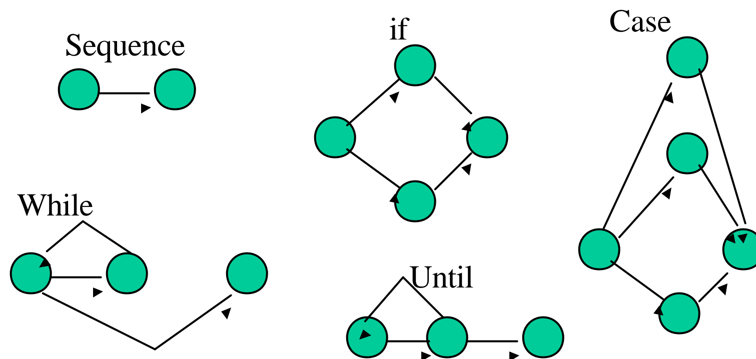
## Why Cover?

- ❏ logic errors and incorrect assumptions are inversely proportional to a path's execution probability
- ❏ we often believe that a path is not likely to be executed; in fact, reality is often counter intuitive
- ❏ typographical errors are random; it's likely that untested paths will contain some

# Basis Path Testing

- A white-box testing proposed by McCabe
- Basis path method enables the test case designer to derive a *logical complexity* measure of a procedural design and to define a *basis set* of execution paths

# Flow Graph Notation




Each circle represents one or more nonbranching PDL or source code statements

# Flow chart – Flow graph

# Basis Path Testing

First, we compute the cyclomatic complexity :

number of simple decisions + 1

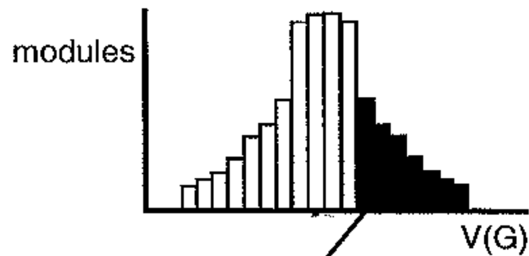
or

number of enclosed areas + 1

in this case, V(G) = 4

# Cyclomatic Complexity

A number of industry studies have indicated that the higher V(G), the higher the probability or errors.




modules in this range are more error prone

# Cyclomatic Complexity

- The number of regions of the Flow Graph
- V(G) = E –N + 2
  - E: # of flow graph edges
  - N: # of flow graph nodes
- V(G) = P + 1
  - P: # of predicate nodes in flow Graph G

# 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.

# Basis Path Testing Notes

- you don't need a flow chart, but the picture will help when you trace program paths
- count each simple logical test, compound tests count as 2 or more
- basis path testing should be applied to critical modules

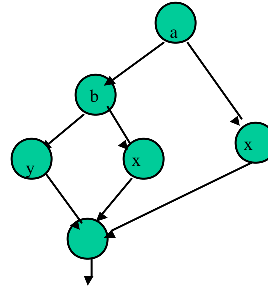
# Compound Logic

If a OR b

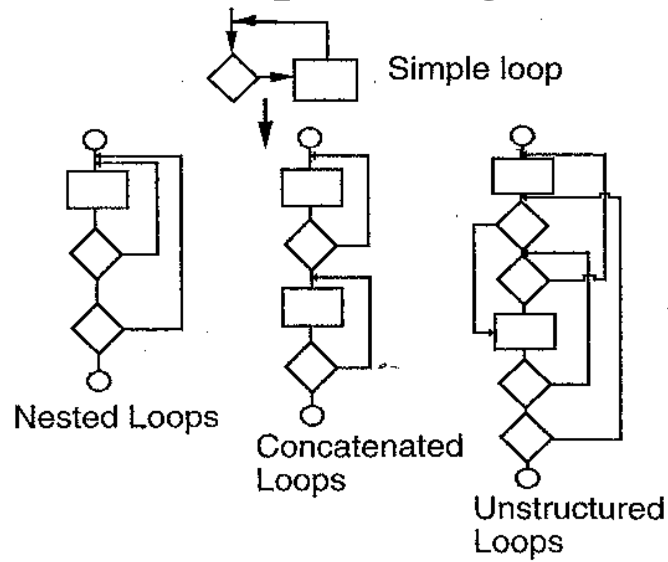
then procedure x

else procedure y

ENDIF




# Loop Testing



## Loop Testing : Simple Loops

*Minimum conditions- Simple Loops*

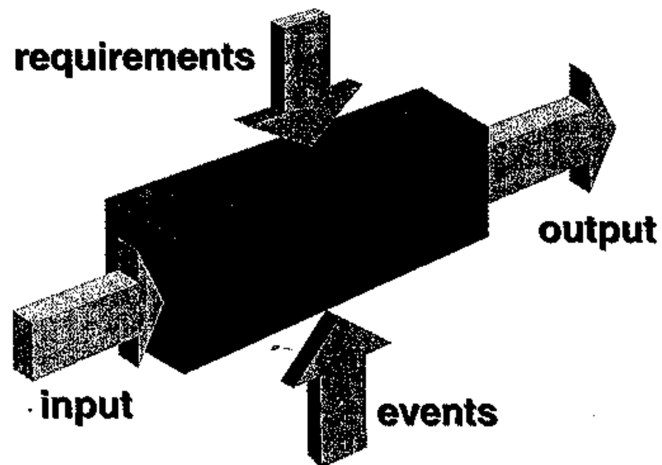
1. skip the loop entirely
2. only one pass through the loop
3. two passes through the loop
4. m passes through the loop m<n
5. (n-1), n, and (n+1) passes through the loop

where n is the maximum number of allowable passes

## Black Box Testing

- Focuses on the functional requirements
- Attemp to find errors on
  - Incorrect of missing functions
  - Interface errors
  - Errors in data structures or external data base access
  - Performance errors
  - Initialaization and termination errors

# Black Box Testing




# Equivalence Partitioning

- A black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived.
- Test case design for equivalance partitioning is based on an evaluation of equivalance classes for an *input condition.*
- An equivalence class represents a set of valid or invalid states for input conditions.
- Input condition is either a specific numeric value, a range of values, a set of related values or a Boolean condition.

# Equivalence Partitioning

Guidelines for Equivalent Classes (EC)

1. If an input condition specifies a *range* , one valid and two invalid EC are defined
2. If an input condition requires a specific *value,* one valid and two invalid EC are defined
3. If an input condition specifies a member of a *set,* one valid and one invalid EC are defined
4. If an input condition is *Boolean*, one valid and one invalid EC are defined

# Equivalence Partitioning

*Example:*

*area code – blank or three digit number*

Input condition, *Boolean – may or may not be present*

Input condition, *range – values defined btw 200 and 999*

*prefix – three digit number beginning with 0 or 1*

Input condition, *range – specified value > 200 with no 0 digits*

*suffix – four digit number*

Input condition, *value – four digit length*

*password – six digit alphanumeric value*

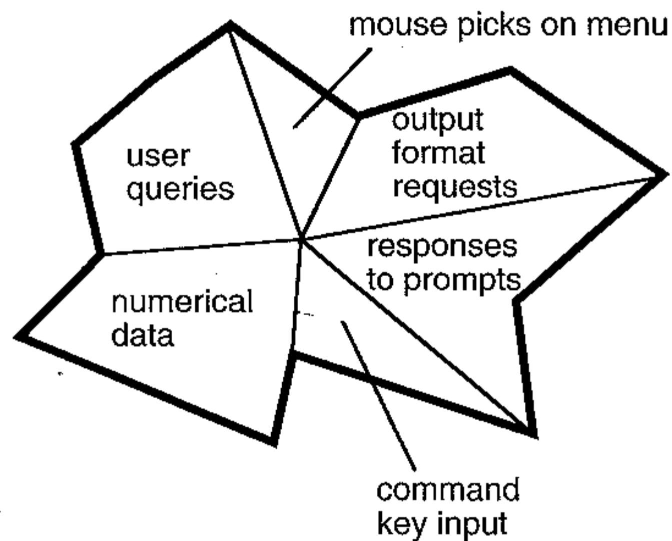
Input Condition *Boolean- a password may or may not be present*

Input condition, *value – six character string*

*commands – {check deposit, bill pay }*

Input condition, *set – containing commands noted above*

# Equivalence Partitioning




# Equivalence Classes

*Valid data*

- user supplied commands
- responses to system prompts
- file names
- computational data
  - physical parameters
  - bounding values
  - initiation values
- output data formatting commands
- responses to error messages
- graphical data (e.g., mouse picks)

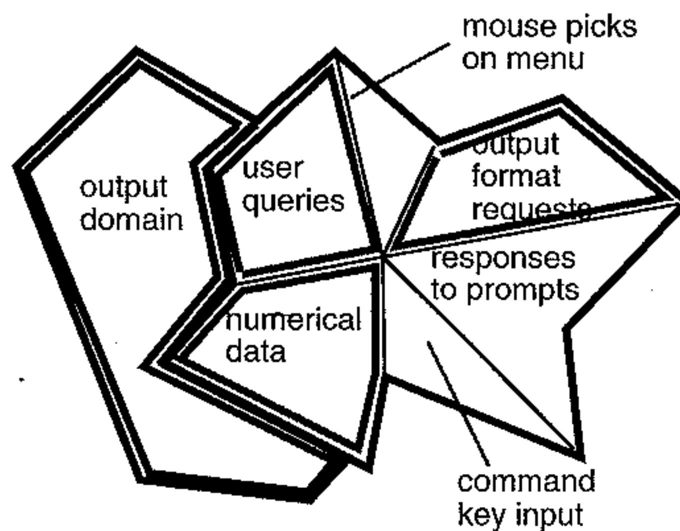
*Invalid data*

- data outside bounds of the program
- physically impossible data
- proper value supplied in wrong place

# Boundary Value Analysis (BVA)

- A great number of errors tend to occur at the boundaries of the input domain than in the center.
- BVA complements Equivalance partitioning
- BVA leads to the selection of test cases at the edges of the class.

# Boundary Value Analysis



# Boundary Value Analysis

- Guidelines for BVA:
  - If an input condition specifies range bounded by values a and b, test cases should be designed with values a and b, just above and just below a and b, respectively.
  - If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum values. Values just above and below minimum and maximum are also tested.
  - If internal program data structures have prescibed boundaries (e.g., an array has a defined limit of 100 entries) be certain to design a test case to exercise the data structure at its boundary.

# Chapter 17
# Software Testing Strategies

## SW Testing Strategy

- A strategy for SW testing integrates SW test case design methods into a well-planned series of steps that result in the successfull construction of SW
- These approaches and philosophies are what we shall call *strategy.*

## A Strategic approach to SW Testing

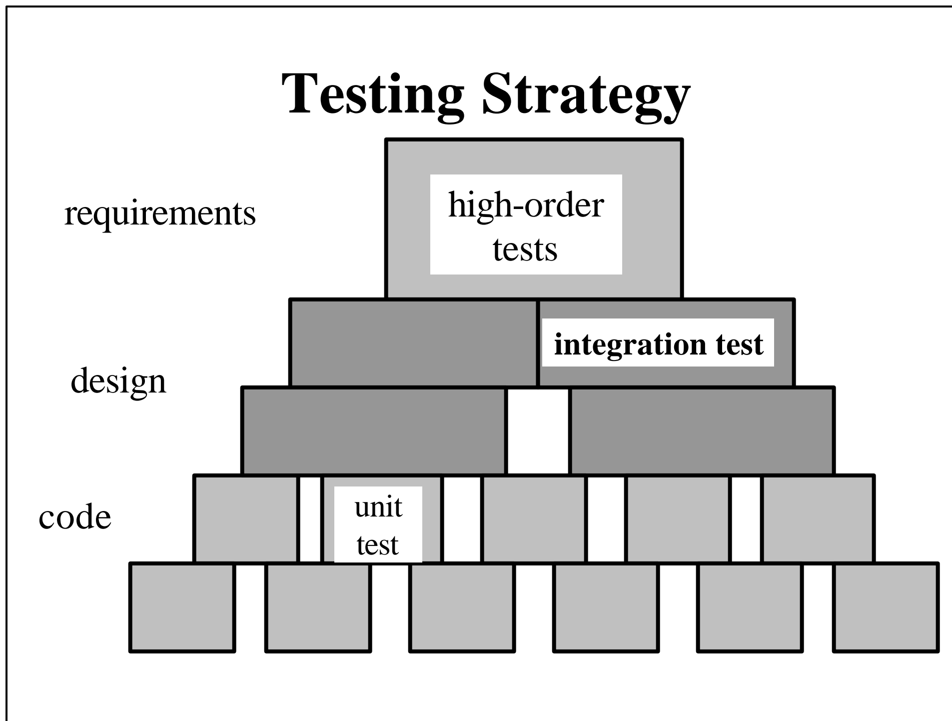
- Testing begins at the module level and works outward toward the integration of the entire computer-based system.
- Different testing techniques are appropriate at different points in time
- Testing is conducted by the developer of the SW and an independent test group
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy

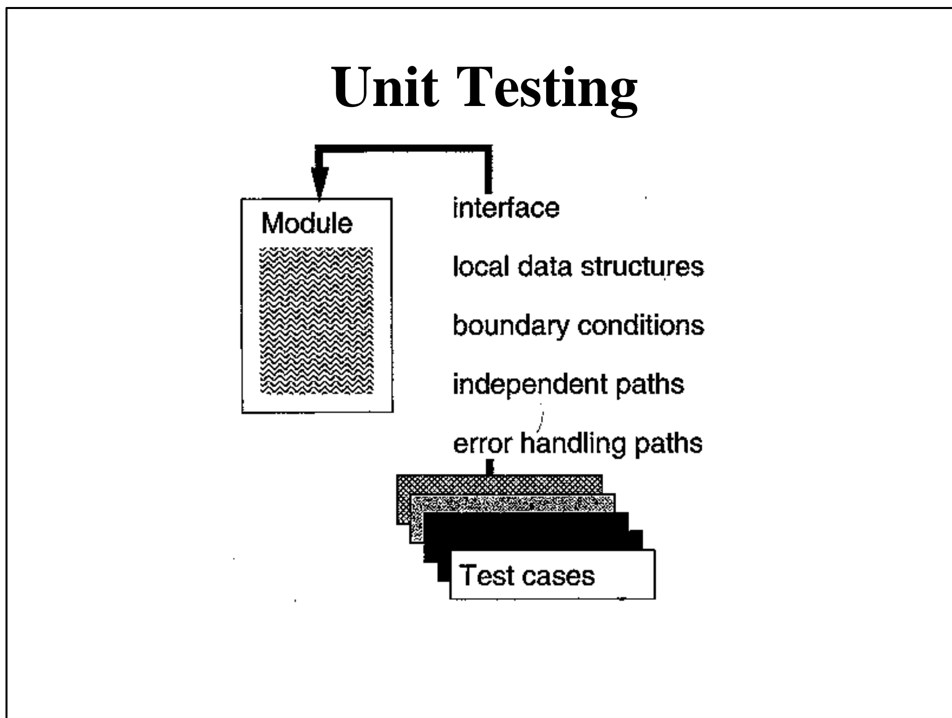
## Verification and Validation

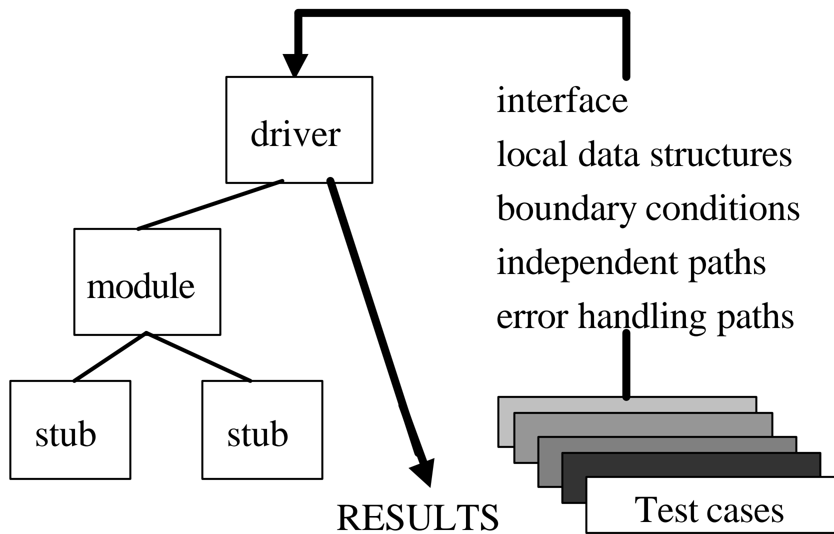
- Verification  - Are we building the product right?
- Validation – Are we building the right product ?

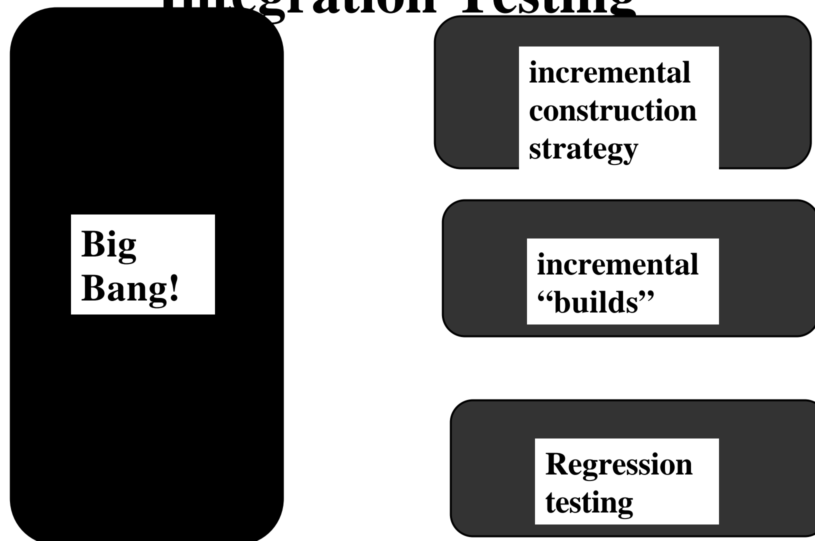
## Testing Strategy

| | |
|---|---|
| • System Engineering | System test |
| • Requirements | Validation Test |
| • Design | Integration test |
| • Code → | Unit Test |

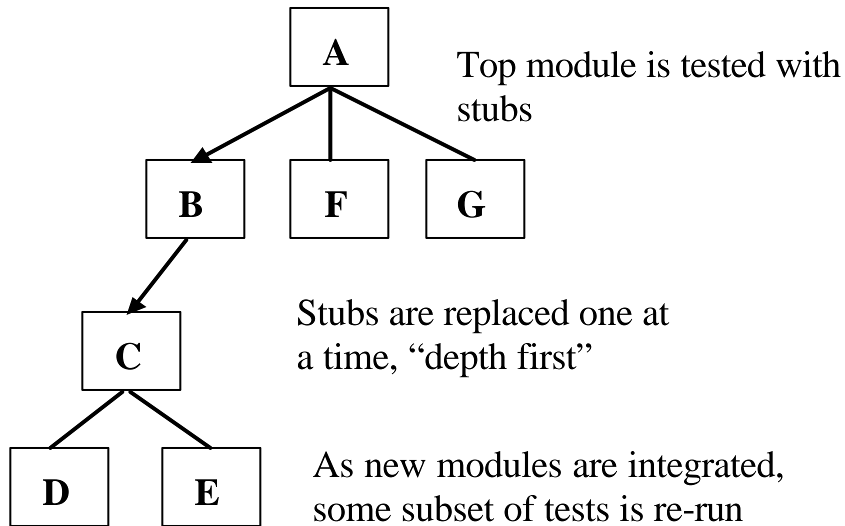

Testing Strategy
requirements
high-order
tests
integration test
design
code
unit
test


Unit Testing
Module
interface
local data structures
boundary conditions
independent paths
error handling paths
Test cases

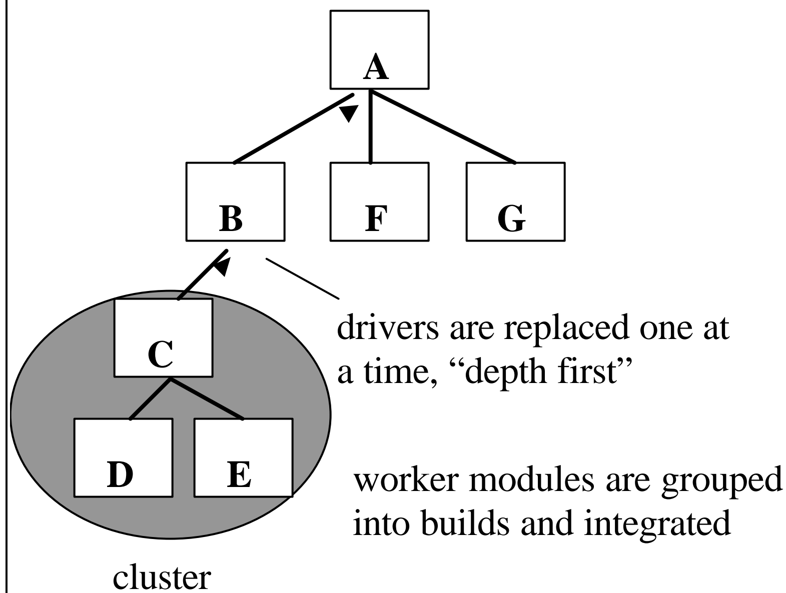

Unit Testing Environment
driver
module
stub
stub
interface
local data structures
boundary conditions
independent paths
error handling paths
Test cases
RESULTS


Integration Testing
Big
Bang!
incremental
construction
strategy
incremental
“builds”
Regression
testing

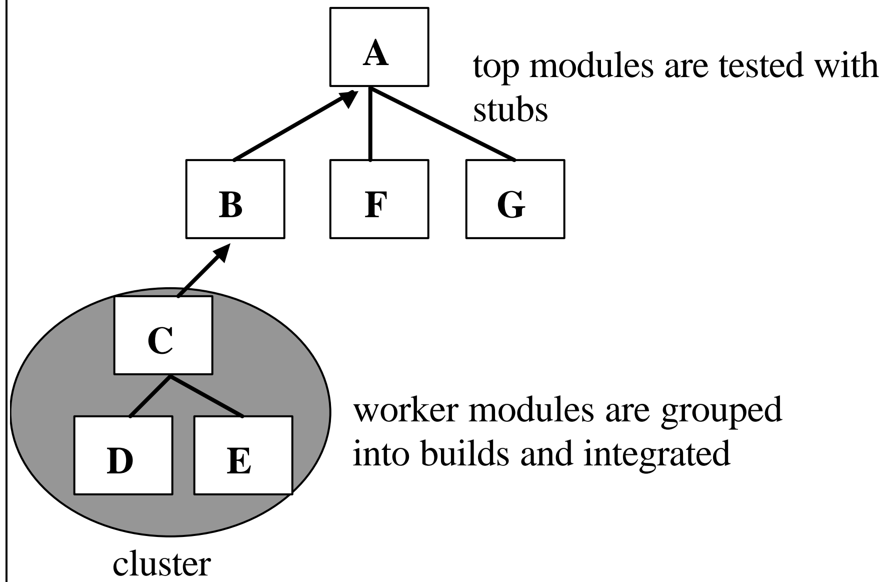
# Top-Down Integration




# Bottom-Up Integration



# Sandwich Testing

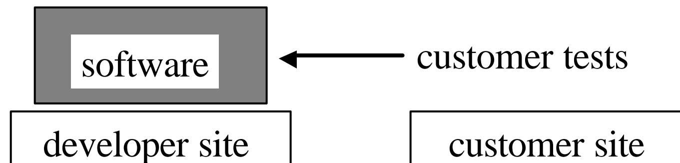



# High-Order Testing

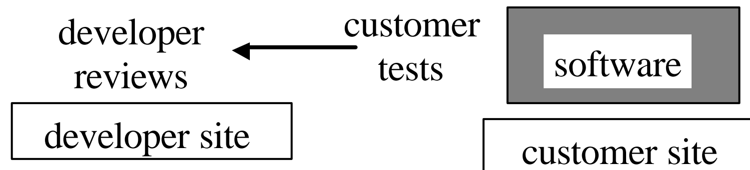
- ❑ validation test
- ❑ system test
- ❑ alpha and beta test
- ❑ other specialised testing

# Alpha & Beta Test

Alpha Test




Beta Test




# Test Specification

1. **Scope of Testing**
2. **Test Plan**
   - **A. test phases and builds**
   - **B. Schedule**
   - **C. overhead software**
   - **D. environment and resources**
3. **Test Procedure *n* (description of test for build *n*)**
   - **A. order of integration**
     1. **purpose**
     2. **modules to be tested**

# Test Specification(Cont'd)

B. unit tests for modules in build
- 1. description of tests for module n
- 2. overhead software description
- 3. expected results

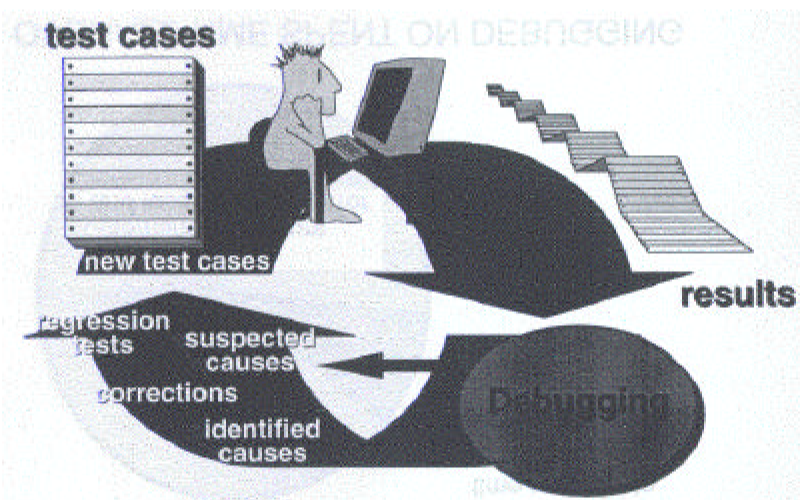
C. test environment
- 1. special tools or techniques
- 2. overhead software description

D. test case data

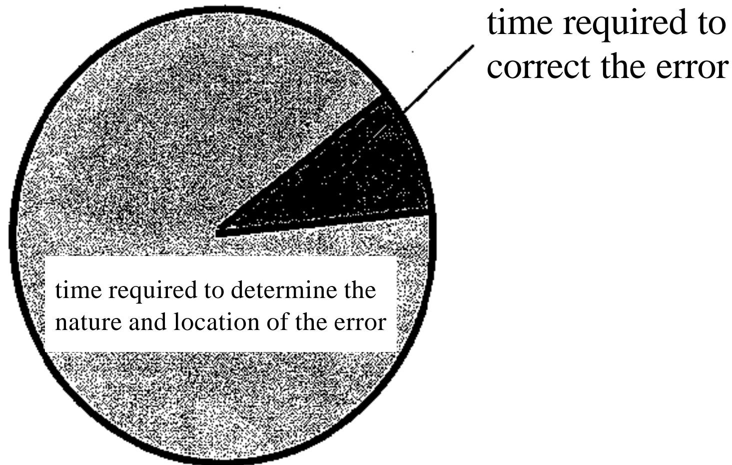
E. expected results for build n

4. Actual Test Results
5. References
6. Appendices

# The Debugging Process

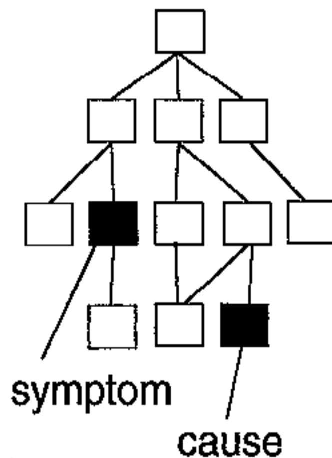


# Debugging : A Two Part Process



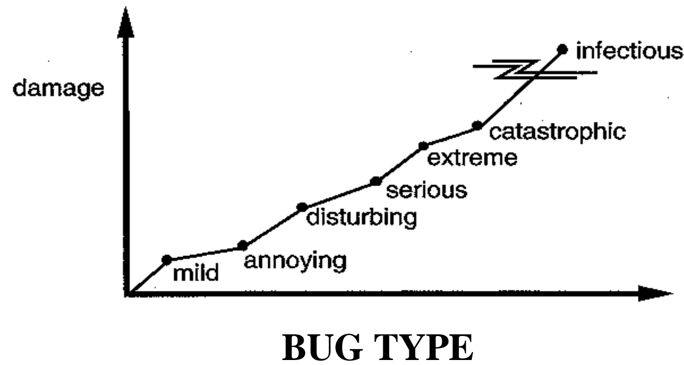

**OVERALL TIME SPENT ON DEBUGGING**

# Symptoms & Causes




- ❑ 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

## Consequences of Bugs




**BUG TYPE**

*Bug Categories :* function-related bugs, system-related bugs, data bugs, coding bugs, design bugs, documentation bugs, standards violations, etc.

## Debugging Techniques

- ❑ brute force / testing
- ❑ backtracking
- ❑ induction
- ❑ deduction

## Debugging : Final Thoughts

**1.** Don't run off half-cocked, think about the symptom you're seeing.

**2.** Use tools (e.g., dynamic debugger) to gain more insight.

**3.** If at an impasse, get help from someone else.

**4.** Be absolutely sure to conduct regression tests when you do "fix" the bug.

# Chapter 18
# Technical Metrics for Software

# McCall's Triangle of Quality

# A Comment

McCall's quality factors were proposed in the early 1970s. They are as valid today as they were in that time. It's likely that software built to conform to these factors will exhibit high quality well into the 21st century, even if there are dramatic changes in technology.

## Formulation Principles

- ❑ The objective of measurement should be established before data collection begins;
- ❑ Each technical metric should be defined in an unambiguous manner;
- ❑ Metrics should be derived based on a theory that is valid for the domain of application (e.g., metrics for design should draw upon basic design concepts and principles and attempt to provide an indication of the presence of an attribute that is deemed desirable);
- ❑ Metrics should be tailored to best accommodate specific products and processes [BAS84]

## Collection and Analysis Principles

- ❑ Whenever possible, data collection and analysis should be automated;
- ❑ Valid statistical techniques should be applied to establish relationships between internal product attributes and external quality characteristics
- ❑ Interpretative guidelines and recommendations should be established for each metric

## Attributes of Technical Metrics

- ❑ **simple and computable**. It should be relatively easy to learn how to derive the metric, and its computation should not demand inordinate effort or time
- ❑ **empirically and intuitively persuasive**. The metric should satisfy the engineer's intuitive notions about the product attribute under consideration
- ❑ **consistent and objective**. The metric should always yield results that are unambiguous.

## Attributes of Technical Metrics(Cont'd)

- ❑ **consistent in its use of units and dimensions**. The mathematical computation of the metric should use measures that do not lead to bizarre combinations of unit.
- ❑ **Programming language independent**. Metrics should be based on the analysis model, the design model, or the structure of the program itself.
- ❑ **An effective mechanism for quality feedback**. That is, the metric should provide a software engineer with information that can lead to a higher quality end product.