Software testing Strategies

Ruchita Shah

Strategic Approach to S/W Testing

- Planned in advance & conducted systematically
- s/w team should conduct FTR, many errors uncovered before testing
- Begins at component level, move outward toward integration of entire computer-based system
- Different testing techniques at different points
- Conducted by developer as well as independent test group
- Debugging accommodated in testing strategy
- Low-level testing to check source code implemented correctly
- High level testing – major system functions acc to requirements

Verification & Validation

- **Verification** refers to activities that ensure that s/w correctly implements a specific function
- **Validation** refers to activities that ensure that s/w built according to customer requirements
- Encompass FTR, Quality & configuration audits, performance monitoring, simulation, feasibility studies, diff testing etc
- Quality can be assessed thro testing but quality can not be tested

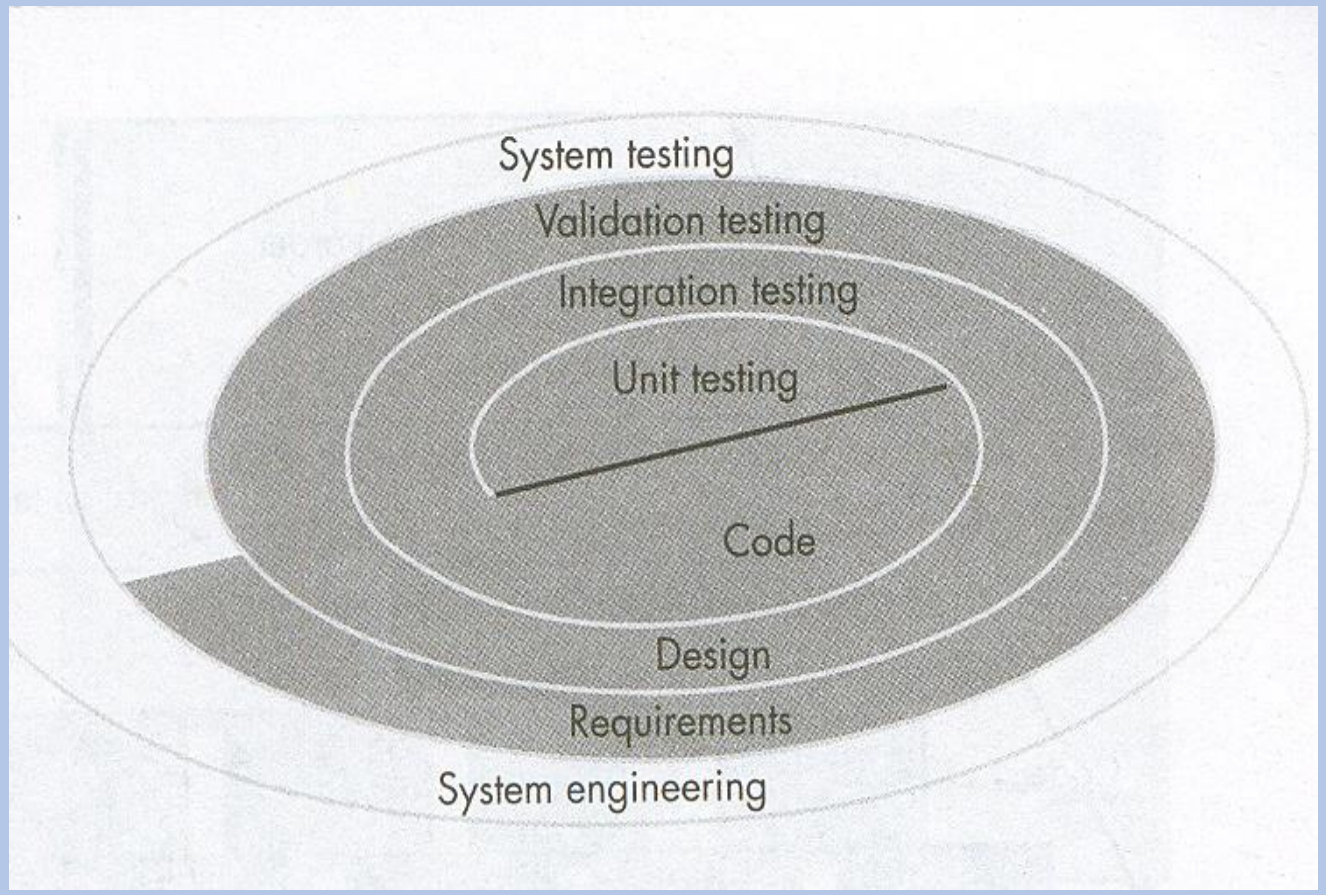
Organizing for S/w Testing

- Developers like to show product as error free and works according to requirements
- s/w has completed on schedule & within budget
- So they try to avoid thorough testing
- Analysis & Design are constructive whereas testing is a destructive task
- S/w developer must test individual units and even perform integration testing
- Independent test group for testing after s/w architecture completes,
- Developer & group work together, group must be part of s/w team

S/w Testing Strategy for Conventional S/W Architecture

- Testing is a spiral activity
- Initially system engineering defines s/w role
- Next s/w requirement analysis establishes info domain, function, behavior, performance, constraints & validation criteria
- Next design & then coding
- Than begin unit testing, each unit tested
- Next integration testing focus on design & construction of architecture
- Validation testing where requirements validated
- finally system testing, s/w & other elements tested as a whole

S/w Testing Strategy for Conventional S/W Architecture



S/w Testing Strategy for Conventional S/W Architecture

- Testing is a series of 4 steps
- Initially unit-testing focus on units, exercise specific paths, ensure complete coverage & max error detection
- Integrated component tested by integration testing
- Testing for verification & program construction, focus on i/p & o/p
- After s/w integrated high order testing- validation testing for functional, behavioral & performance requirements conformance
- S/w combined with other system elements, system testing for overall system function & performance

S/W Testing Strategy for OO Architectures

- Testing broadened to include error discovery techniques ex FTR
- Completeness & consistency of objects assessed as constructed
- Classes are integrated into OO architecture
- Regression testing performed to uncover errors in communication & collaboration b/w classes
- Lastly system as a whole

Criteria for Completion of Testing

- When testing is complete? No definite answer
- Every time a user executes a program, it is tested
- Statistical modeling & s/w reliability theory used to model s/w failure as function of execution time
- S/w failure as a function of time is calculated for this

f(t) = (1/p) ln[Io pt + 1]

f(t) = no of failures expected after s/w tested for execution time t

Io = initial s/w failure intensity at beginning of testing

P = exponential reduction in failure intensity

Strategic Issues

For successful s/w testing

- Specify product requirements in a quantifiable manner long before testing commences. Good testing also assess product for quality attributes such as portability, maintainability etc, specified in measurable way
- State testing objectives explicitly. Objectives stated in measurable terms ex test effectiveness, test coverage, mean time to failure, cost to find & fix defects etc in test plan

Strategic Issues

For successful s/w testing

- Understand the users of s/w & develop user profile for each user category. Use-case for interaction scenario, reduce testing efforts, focus on actual use of product
- Develop a testing plan that emphasizes rapid cycle testing. Rapid cycle tests, increments of functionality, feed back used for quality control & test strategy
- Build robust s/w that is designed to test itself. Use of antibugging techniques in s/w, design accommodate automated & regression testing

Strategic Issues

For successful s/w testing

- Use effective FTR as a filter prior to testing. Uncovers errors, reduce testing effort & time
- Conduct FTR to assess the test strategy & test cases themselves. Uncovers inconsistencies, omissions & errors, saves time & improves quality
- Develop a continuous improvement approach for the testing process, Collection of Metrics

Testing Strategies for Conventional S/W

UNIT TESTING

- Verification of smallest unit
- Imp control paths tested to uncover errors within boundary of module
- Test & errors uncovered are limited
- Focus on internal process logic & data structure

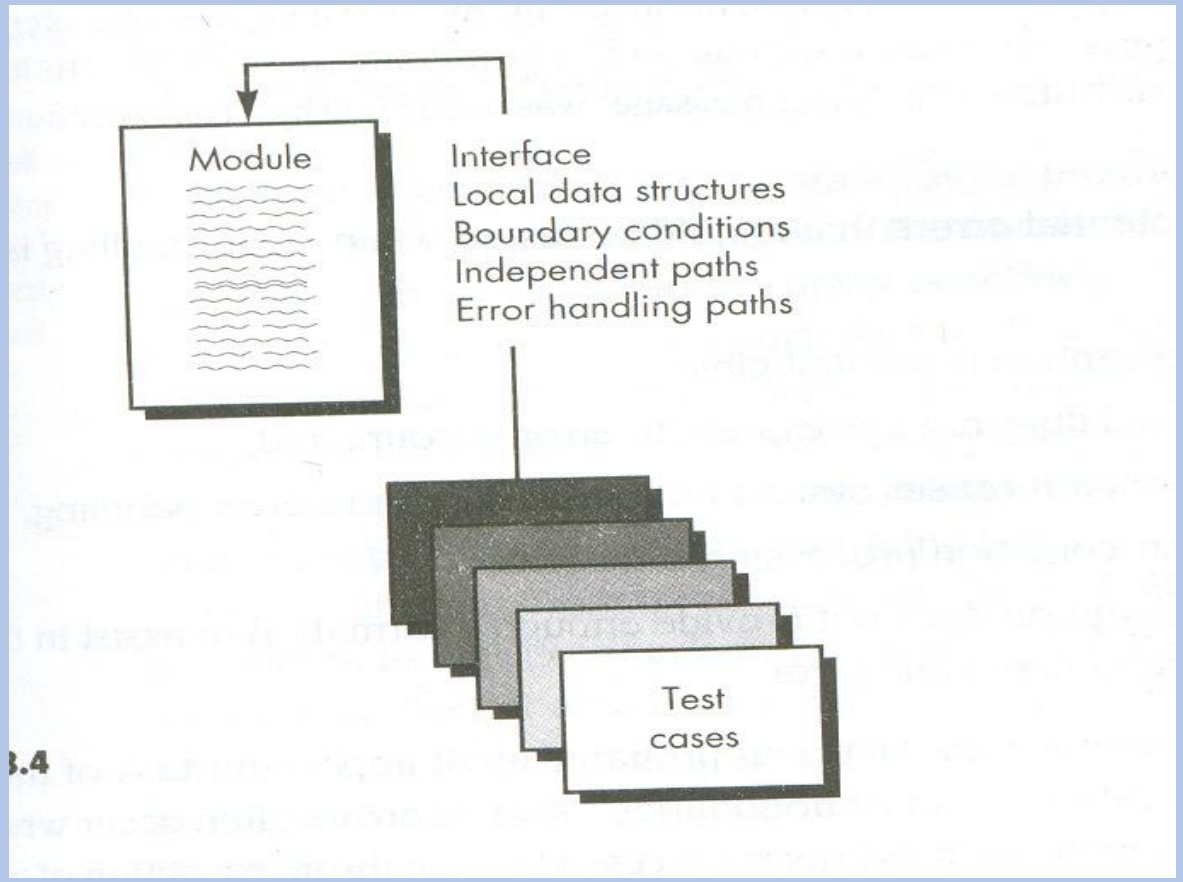
UNIT TESTING

Unit Test Consideration

- Module i/f tested for proper information flow into & out of module
- Local DS to check data maintains integrity in all steps
- Impact of local DS to global DS
- Boundary conditions tested to check operations at boundary
- All independent paths exercised to ensure all statements executed at least once
- All error-handling paths tested

UNIT TESTING

Unit Test Consideration



UNIT TESTING

Unit Test Consideration

- Test must uncover errors due to erroneous computation, incorrect comparisons or improper control flow
- **computational errors** are

1. Misunderstood or incorrect arithmetic precedence
2. Mixed mode operation
3. Incorrect initialization
4. Precision inaccuracy
5. Incorrect symbolic representation

UNIT TESTING

Unit Test Consideration

- **Test for incorrect comparison & improper control flow errors** are

1. Comparison of different data types
2. Incorrect logical operator or precedence
3. Expected equality when not possible
4. Incorrect comparison of variables
5. Improper or nonexistence of loop termination
6. Exit not provided in diverse condition
7. Improperly modified loop variables

UNIT TESTING

Unit Test Consideration

- Boundary testing most imp as s/w often fails at boundary
- Test case for DS, control flow & data values just below, at & just above max & min
- Antibugging technique during programming : Errors are anticipated & error-handling paths setup or terminate the processing cleanly when error occurs

UNIT TESTING

Unit Test Consideration

- These error-handling also must be tested
- errors encountered are

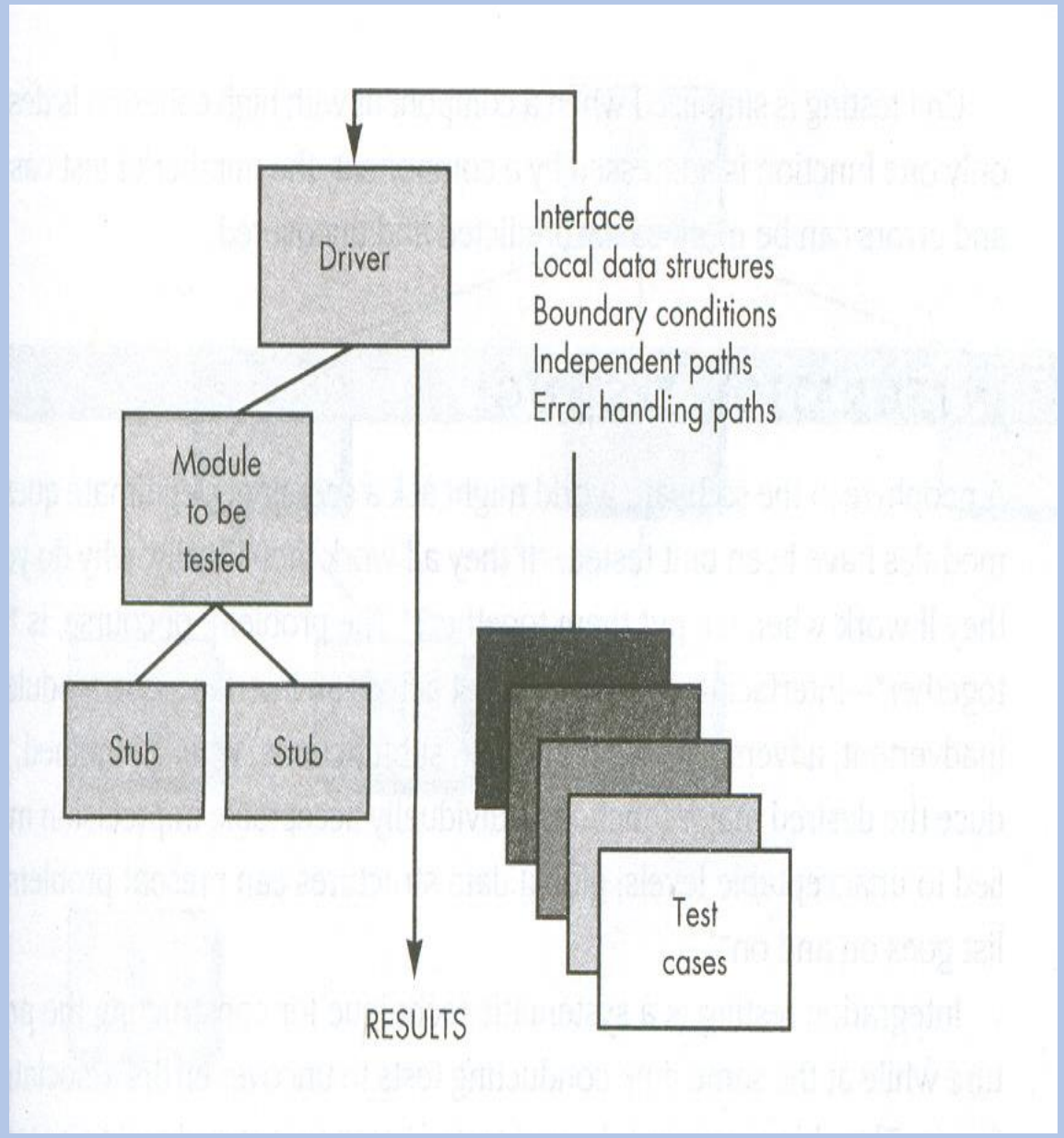
1. Unintelligible error description
2. No correspondence to error noted & error occurred
3. Error cause system intervention before error handle executes
4. Exception condition processing improper

Unit Test Procedures

- Unit test after source code developed, reviewed & verified
- Each test case coupled with expected results
- A module is not a stand alone
- A driver or stub s/w developed for each unit test
- Driver : a main program that accepts test data, passes to test module & prints results
- Stub : replace subordinate modules of test module, a dummy subprogram, do minimal data manipulation, print verification & returns control to test module

Unit Test Procedures

- Drivers & stubs are overhead to program
- Keep them simple
- Many compo not tested adequately with driver & stub then postpone until integration
- Unit testing simple if module with high cohesion
- less no of test cases & errors easily uncovered



INTEGRATION TESTING

- Data can be lost across i/f
- One module may have adverse effect on others
- Combined subprograms may not produce desired result
- Acceptable imprecision might get magnified
- Problem in global DS
- For these we need Integration testing
- Systematically construct program structure & conducts tests for i/f errors

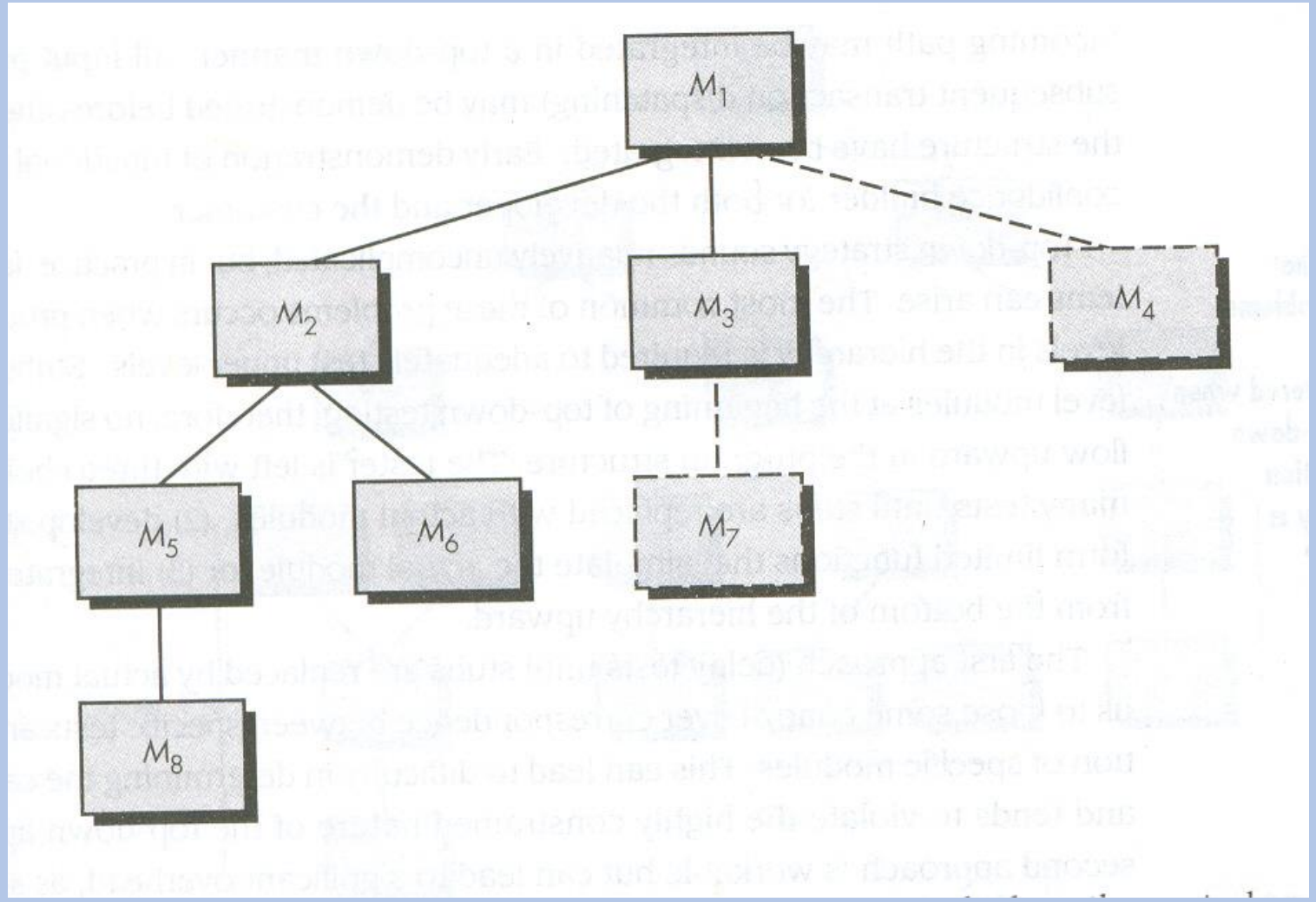
INTEGRATION TESTING

- 2 methods
  - Chaotic - combine all subprograms & construct whole structure & then test, chaos results, difficult to isolate causes of errors
  - Incremental integration better

Top-down Integration

- Incremental program structure by moving downward
- Begin with main program
- Integrate subordinate modules in depth-first or breadth-first manner
- Depth first : integrate all modules on a major control path
- Breadth first : incorporate modules level by level

Top-down Integration



Top-down Integration

- Integration process

1. Main control module as a test driver & stubs substituted for direct subordinate of main module
2. Depending on approach, subordinate stubs replaced with actual module, one at a time
3. Test as each component integrated
4. Another stub is replaced
5. Regression testing to ensure new errors not introduced

Top-down Integration

- Top-down approach verifies major control points early in test
- Early recognition of problem in major control
- Depth first checks complete function of s/w

Top-down Integration

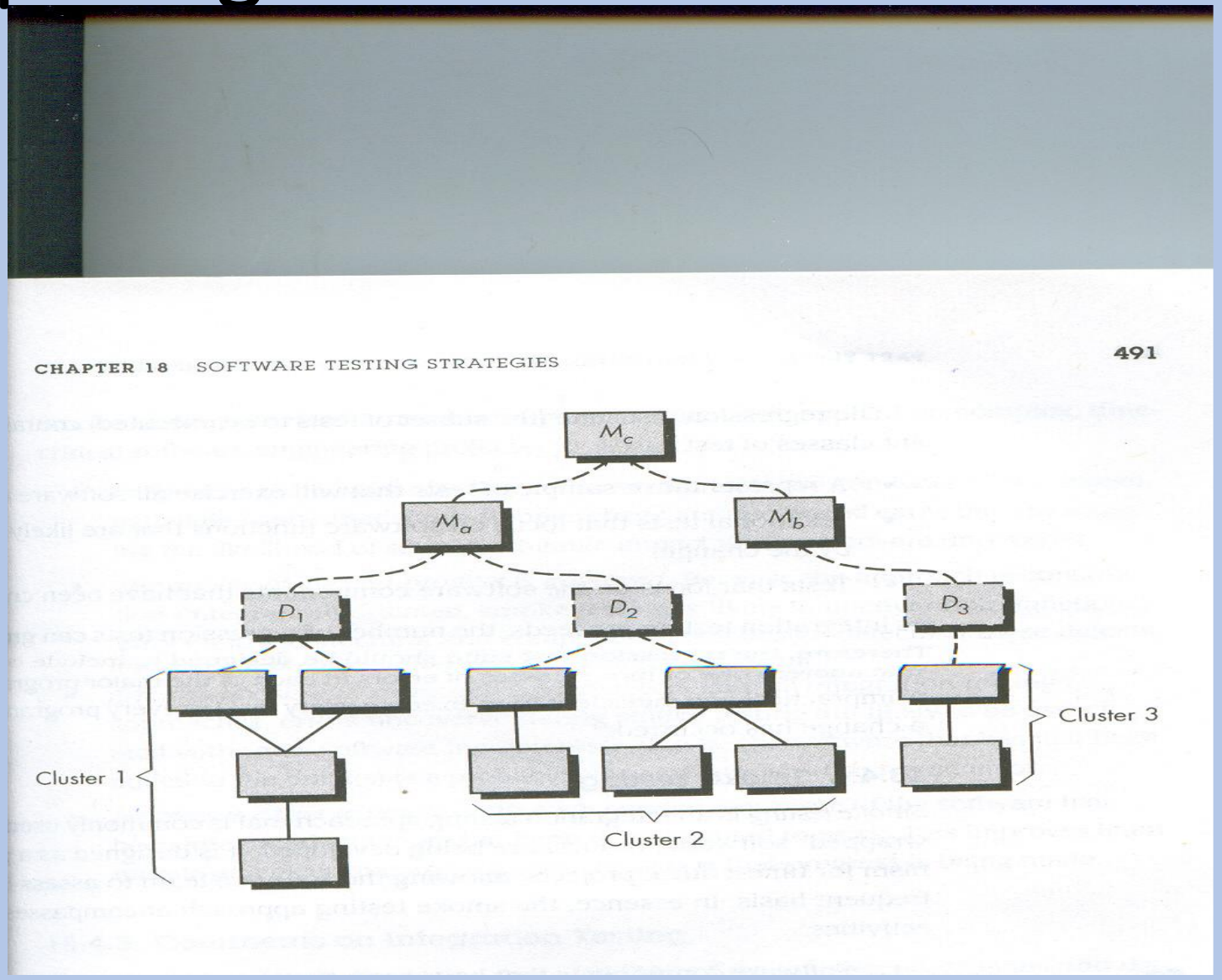
- Problem in top-down approach
- When processing at low level require adequate testing of upper levels
- stubs in place of subordinate, actual data do not flow upward
- Either delay some test until after integration of actual subordinate or develop stubs with limited functionality of actual module or use bottoms up integration

Bottom-up Integration

- Starts construction & testing from lowest level
- Need for stub eliminated
- Steps are

1. Low level components combined into clusters that perform a specific sub function
2. Driver for i/p/& o/p
3. Cluster tested
4. Drivers removed & clusters combined moving upward

Bottom-up Integration



Bottom-up Integration

- Components combined to form 1,2,& 3 clusters
- Tested using driver
- Super ordinate of 1 & 2 i.e. Ma in place of D1 & D2
- Mb replaces D3, then after Ma & Mb integrated to Mc
- Moving upward drivers in less no

Regression Testing

- Each new module added for integration testing changes s/w
- New data paths, new I/O, new control logic, may cause problem with functions working flawlessly earlier
- Regression testing is re-execution of subset of tests conducted
- To ensure that changes do not propagate side effects
- Whenever s/w debugged, configuration changes
- Regression testing to ensure new errors not introduced
- Done manually by executing subset of all test cases or automated capture/playback tools

Regression Testing

- Enables to capture tests & results
- Regression test suite contains 3 classes of test cases
  - Tests that will exercise all s/w functions
  - Additional tests that focus on functions likely to be affected by change
  - Tests of components that are changed
- With progression of integration testing, Regression test suite can grow large
- Include only those tests that address errors in major functions

Smoke Testing

- Used when shrink-wrapped s/w is developed
- For time critical projects
- Allows to assess project frequently
- Activities are

1. S/w components coded are integrated into build, includes all data files, libraries, reusable modules & constructed modules to implement program function
2. Tests designed to expose errors in that function i.e. in build, errors that can affect project schedule are identified
3. Build integrated with other builds, smoke tested daily, diff integration approach

Smoke Testing

- Beneficial on complex, time critical s/w
- integration risk minimized. Incompatibilities & stopping errors uncovered early, reduce serious schedule impact
- Quality of end product is improved. Constructive approach, uncovers functional, architectural & component design defects
- Error diagnosis & correction are simplified. Errors uncovered are associated with newly attached s/w
- Progress is easier to assess. Each day more s/w integrated & tested, improves team morale

Strategic options

- Advantage of one strategy may be disadvantage of other
- Disadvantage of top-down is requirements of stubs
- Adv is major control functions tested early
- Disadvantage of bottom-up is program as an entity do not exist until last module
- Adv is easy test case design & removal of stubs
- Strategy depends on s/w characteristics, project schedule

Integration Test Documentation

- Plan for integration of s/w & description of tests in test specification
- Contains test plan & test procedure
- Part of s/w configuration
- Testing divided into phases & builds
- Address specific functions of s/w
- Criteria & tests applied are
  - Interface integrity : Internal & external i/f tested with module integration
  - Functional validity : test for functional errors
  - Information content : tests for errors in local & global DS
  - Performance : to verify performance bounds

Integration Test Documentation

- Also contain schedule of integration, overhead s/w etc
- Start & end date of each phase
- Test env & resources
- Detail testing procedure to accomplish test plan
- Order of integration & tests
- List of test cases & expected results etc
- Actual test results
- Problem or peculiarity recorded in test specification

Test Strategies for OO S/W

- Objective of testing to find more no of errors with manageable amount of efforts in realistic time span
- Testing strategy & tactics diff for OO s/w

Unit testing in OO context

- Concept of unit with OO changes
- Each class & its instance combine attributes & operations
- Unit testing focuses on encapsulated class
- Operation within class are smallest testable unit
- A class contain no of diff operations & an operation exist as part of diff classes
- Tactic for unit test changes

Unit testing in OO context

- A single operation can not be tested isolation
- Ex. An operation X defined in superclass, inherited by a no of subclasses, each subclass uses operation X but within context of private attributes & operations of that subclass, thus operation X varies in each subclass, operation X to be tested in context of each subclass, Standalone testing will be ineffective
- Unit testing focus on algorithmic details & data of module
- OO testing focus on operations encapsulated in the class & state behavior of class

Integration Testing in OO Context

- OO s/w do not have obvious hierarchical control structure
- Top-down & bottom-up integration meaningless
- Integrating classes one at a time impossible, because direct & indirect interactions b/w components of class
- 2 diff strategies
  - Thread-based strategy- integrated set of classes that respond to one i/p or event for system, each thread integrated & tested individually
  - User-based testing- begins construction of system by testing those classes that use very few server classes, these classes are independent classes

Integration Testing in OO Context

- After testing independent classes next layer of classes called dependent classes which use independent classes are tested
- These sequence of testing layers of dependent classes continue until entire system is constructed
- Use of drivers & stubs also changes while testing OO system
- Drivers used to test operations at lowest level & testing whole group of classes
- Driver replaces user i/f to check system functionality before implementation

Integration Testing in OO Context

- Stubs used when collaboration of classes is required but collaborating class is not implemented
- Cluster testing if integration testing of OO s/w
- Cluster of collaborating classes tested to uncover errors in collaboration

Validation Testing

- After integration testing s/w is assembled
- Validation testing checks if s/w functions according to customer requirements
- Validation criteria defined in s/w requirements specification provides base for testing

Validation Test Criteria

- It is a Black-box testing
- Conformance to requirements
- Test plan & test procedures defines classes of tests & test cases
- Checks if functional requirements satisfied
- Behavioral characteristics achieved
- Human engineer & other requirements such as error recovery, compatibility, maintainability meet

Validation Test Criteria

- After each validation test, 2 possible conditions
  - Function & performance confirms
  - Deviation from specification
- Deviation or error at this stage rarely corrected before schedule
- Configuration Review : To ensure all s/w configuration elements developed & cataloged

Alpha & Beta testing

- Impossible to assess how customer will use program
  - Instruction misinterpreted
  - strange combination of data
  - o/p not understandable to user
- Acceptance test conducted
- Customer validates requirements
- Conducted by end users
- Conducted over period of time to uncover errors that might degrade the product

Alpha & Beta testing

- Alpha & Beta testing to uncover errors that only end-users can find
- Alpha test conducted at developer’s site by user
- Controlled env
- Natural setting of developer
- Looking over the shoulder
- Records errors & usage problems

Alpha & Beta testing

- Beta test at customer site by end users
- Developer not present
- Live application of s/w
- Env not controlled
- customer records problems & reports to developer

System Testing

- S/w incorporated with other system elements, h/w, people, env after system integration & validation tests
- Errors not solely by s/w engineers
- System testing problems leads to finger-pointing
- When error uncovered each developer blame others
- Engineers must
  - Design error-handling paths
  - Conduct series of tests
  - Record results of test
  - Participate in planning & design of system elements

System Testing

- A series of different tests
- Fully exercise computer based system
- Verify system elements integrated properly & perform allocated functions

System Testing - Recovery Testing

- Computer-based system must recover from faults
- Resume processing within pre specified time,
- Fault tolerance – processing faults do not cause overall system function to cease
- Recovery testing forces s/w to fail in diff ways & verify recovery is performed
- If recovery automatic, re-initialization, checkpoint mechanism, data recovery, restart evaluated for correctness are tested
- If require human intervention, MTTR evaluated.

System Testing - Security Testing

- Computer-based system that have sensitive info is target for illegal penetration
- Spans a broad range of activities – hackers who attempt to penetrate system fro fun, disgruntle employee attempt for revenge, dishonest for illicit personal gain
- Security testing verify protection mechanism
- Test play role of individual who penetrates the system
- Tester may acquire password through external clerical means

System Testing - Security Testing

- May attack system with custom s/w designed to break down any defense
- Confuse the system & stopping service to others
- Purposely cause some system errors & penetrate during recovery
- Browse insecure data
- Given time & resources good security testing penetrates a system
- Cost of penetration more than value of info

System Testing - Stress Testing

- White box & black-box testing evaluate normal program functions & performance
- Stress test confront program with abnormal situation
- Execute system demanding abnormal quantity, frequency or volume of resources, ex tests generated
  - Demand ten interrupts in second
  - I/p data rate increased
  - Require max memory or other resources
  - Excessive disk data fetching etc

System Testing - Stress Testing

- Sensitivity testing – some time small range of data within bounds may cause extreme or erroneous processing & performance degradation, sensitivity testing uncovers such data combinations

System Testing - Performance Testing

- For real & embedded systems
- S/w provides required functions but not conform performance requirements
- Performance testing test run-time performance of an integrated system
- Occur throughout all tests
- Performance of individual module to full system
- Require special h/w & s/w
- Monitor execution intervals, log events, machine states

The Art of Debugging

- As a consequence of testing
- Begins with test
- Removal of errors
- External errors & internal cause may not have obvious relationship

Debugging Process

- Begins with testing
- Result assessed & checked with expected results
- If not same there is problem
- Underlying cause must be found & error correction
- Either cause found & corrected or not found
- Design special test case & validate

Debugging Process

- Debugging is difficult because

1. Symptom & cause geographically remote
2. Symptom disappear when another error corrected
3. Symptom may caused by non-errors
4. May caused by human error, not easily traceable
5. May result of timing problem rather than processing
6. Difficult to reproduce i/p conditions
7. Intermittent symptom common in embedded system
8. Causes distributed across a no of tasks running on diff processors

Debugging Process

- Effect may range from mild to catastrophic
- Pressure increase with errors
- Introduction of more errors while fixing one error

Debugging Approaches

- Objective is to find & correct cause of error
- Require systematic evaluation, intuition & luck
- 3 approaches : (1) brute force (2) backtracking & (3) cause elimination
- Brute force : More common
  - Less efficient
  - When all else fail
  - Memory dumps taken
  - Run-time traces invoked
  - program loaded with write statements to find clue to cause of errors
  - waste of time & effort

Debugging Approaches

- Backtracking : for small programs
  - Begins at symptom
  - Source code traced backward until cause found
  - if large program no of paths to trace are unmanageable
- Cause elimination : work on binary partitioning
  - Data related to error organized
  - cause hypothesis devised & data used to prove or disprove hypothesis
  - Alternately list of all possible causes developed & tests conducted to eliminate each

Debugging Approaches

- Debugging tools : like debugging compilers aid
- Automatic test case generators etc available
- A fresh viewpoint, unclouded by frustration do wonders
- Bugs found are corrected but may introduce new errors (1) cause of bug reproduces in other parts of program (2) new bug introduction while fixing (3) prevention of bugs