Khulna University
Computer Science & Engineering Discipline

SK Alamgir Hossain

Assistant Professor
Computer Science & Engineering Discipline
Khulna University, Khulna, Bangladesh

http://alamgirhossain.com

CSE3203 - Software Engineering and Information System Design

Chapter 12

– Software Testing

Chapter Outline

- Software Testing Strategies
  - A strategic approach to software testing
  - Unit Testing
  - Integration Testing
  - Validation Testing
  - System Testing
  - The ART of Debugging
- Software Testing Techniques
  - Testability
  - White-Box Testing
  - Basis Path Testing
  - Control Structure Testing
  - Black-Box Testing
  - Comparison Testing
  - Orthogonal Array Testing
  - Testing Patterns

Lecture 22

Software Testing

- A strategic approach to software testing
- Unit Testing
- Integration Testing
- Validation Testing
- System Testing
- The ART of Debugging
- Summary

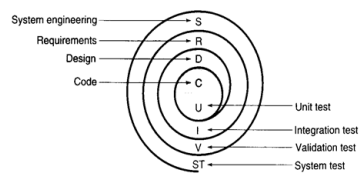
Verification and Validation

Figure: Achieving software quality

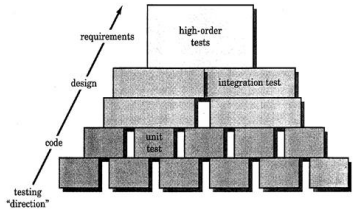
- Verification -- Does the product meet its specifications?
- Validation -- Does the product perform as desired?

Software Testing Strategy

A Software Testing Strategy (spiral view)



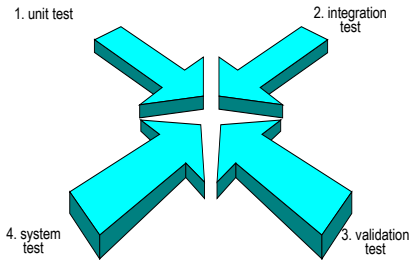
Software Testing Strategy



A Software Testing Strategy (Procedural view)

- Generic characteristics of software testing strategies:-
- Testing begins at module level and works outward towards the of integration entire computer based system.
- Different testing techniques are required at different points in time.
- Testing is conducted by the s/w developer and ITG (Independent Test Group) for large projects.
- Testing and Debugging are different and Debugging is essential in any testing strategy.

Testing Strategy



Testing Strategy

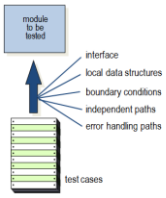
- We begin by 'testing-in-the-small' and move toward 'testing-in-the-large'
- For conventional software
  - The module (component) is our initial focus
  - Integration of modules follows
- For OO software
  - our focus when "testing in the small" changes from an individual module (the conventional view)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

Strategic Issues

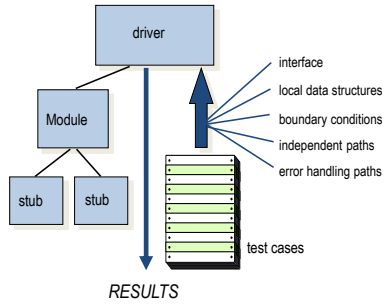
- State testing objectives explicitly.
- Understand the users of the software and develop a profile for each user category.
- Develop a testing plan that emphasizes "rapid cycle testing."
- Build "robust" software that is designed to test itself
- Use effective formal technical reviews as a filter prior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

1. Unit Testing

- Unit testing -- focuses on the smallest element of software design viz. the module.
  - Corresponds to class testing in the OO context.
- Makes heavy use of white-box testing.



Unit Test Environment



Unit Test Generation

- Interface considerations
  - # of input parameters = # arguments?
  - Parameter and argument attributes match?
  - Parameter and argument units match?
  - Order correct (if important)?
  - Number and order of arguments for built-ins?
  - References to parms not associated with entry point?
  - Attempt to modify input-only arguments?
  - Global variable definitions consistent?
  - Constraints passed as arguments?

Unit Test Generation

- External I/O considerations
  - Files attributes correct?
  - OPEN/CLOSE correct?
    Format specification matches I/O statement?
  - Buffer size matches record size?
  - Files opened before use?
  - EOF handled correctly?
    I/O errors handled?
  - Textual errors in output?

Unit Test Generation

- Data structure considerations
  - Improper or inconsistent typing?
  - Erroneous initialization or default values?
  - Incorrect variable names?
  - Inconsistent data types?
  - Underflow, overflow and addressing exceptions?

Unit Test Generation

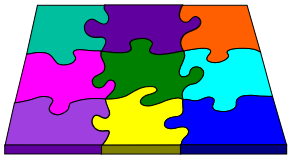
- Test cases must cover all execution paths
- Common computational errors to be checked:
  - incorrect arithmetic
  - mixed mode operations
  - incorrect initialization
  - precision inaccuracy
  - incorrect symbolic representation of expression
- Other tests needed
  - incompatible data types in comparisons
  - incorrect logical operators or precedence
  - comparison problems (e.g., == on floats)
  - loop problems

Unit Test Generation

- Error handling tests
  - Exception-handling is incorrect?
  - Error description is unintelligible, insufficient or incorrect?
  - Error condition causes system interrupt before error handling completed?

2. Integration Testing Strategies

Options:
- the "big bang" approach
- an incremental construction strategy

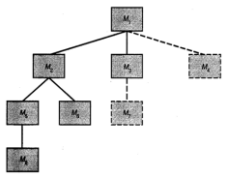


Integration Testing

- A systematic approach for constructing program structure while conducting tests to uncover errors associated with interfacing.
- Tendency for Non-Incremental integration.. Big Bang approach …. Chaos !! (usually).
- Incremental integration - program is constructed and tested in small segments.
  - Top-Down Integration testing
  - Bottom-Up Integration testing

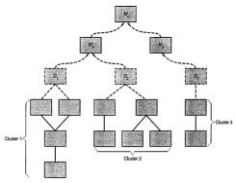
Top-Down Integration testing

- Top-Down Approach
- Begin construction and testing with main module.
  - Stubs are substituted for all subordinate modules.
- Subordinate stubs are replaced one at a time by actual modules.
- Tests are conducted as each module is integrated.
- On completion of each set of tests, another stub is replaced with the real module.
- Regression testing may be conducted to ensure that new errors have not been introduced.



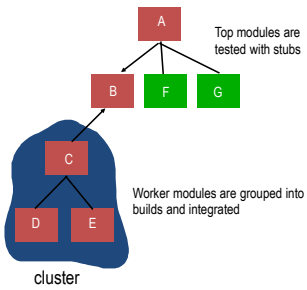
Bottom-Up Integration testing

- Bottom Up Approach :
  - This approach begins construction and testing with modules at the lowest levels in the program structure.
    - Low-level modules are combined into clusters.
    - A driver is written to coordinate test case input and output.
    - The cluster is tested.
    - Drivers are removed and clusters are combined moving upward in the program hierarchy.



- Bottom Up Approach
  - Advantages:
    - Easier test case design and lack of stubs.
  - Disadvantages:
    - The program as an entity is does not exist until the last module is added.
- Sandwich Testing:- combined approach
  - Top down strategy for upper levels and Bottom up strategy for subordinate levels.

Sandwich Testing



3. Regression Testing

- Regression Testing
  - Re-execution of some subset of tests already conducted to ensure that the new changes do not have unintended side effects.
- The Regression test suite should contain three different classes of test cases :
  - A representative sample of tests that will exercise all software functions
  - Additional tests that focus on functions that are likely to be affected by the change.
  - Tests that focus on software components that have changed.

Object-Oriented Testing

- begins by evaluating the correctness and consistency of the OOA and OOD models
- testing strategy changes
  - the concept of the 'unit' broadens due to encapsulation
  - integration focuses on classes and their execution across a 'thread' or in the context of a usage scenario
  - validation uses conventional black box methods
- test case design draws on conventional methods, but also encompasses special features

Broadening the View of "Testing"

It can be argued that the review of OO analysis and design models is especially useful because the same semantic constructs (e.g., classes, attributes, operations, messages) appear at the analysis, design, and code level. Therefore, a problem in the definition of class attributes that is uncovered during analysis will circumvent side effects that might occur if the problem were not discovered until design or code (or even the next iteration of analysis).

OOT Strategy

- class testing is the 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

Smoke Testing

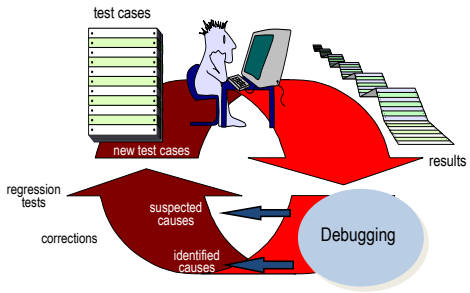
- A common approach for creating "daily builds" for product software
- Smoke testing steps:
  - Software components that have been translated into code are integrated into a "build."
    - A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
  - A series of tests is designed to expose errors that will keep the build from properly performing its function.
    - The intent should be to uncover "show stopper" errors that have the highest likelihood of throwing the software project behind schedule.
  - The build is integrated with other builds and the entire product (in its current form) is smoke tested daily.
    - The integration approach may be top down or bottom up.

High Order Testing

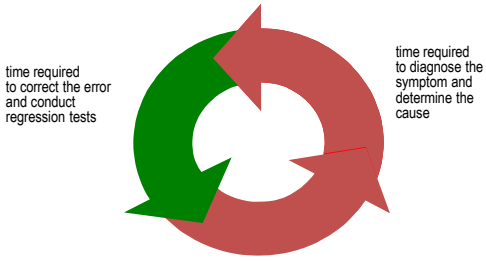
- Validation testing
  - Focus is on software requirements
- System testing
  - Focus is on system integration
- Alpha/Beta testing
  - Focus is on customer usage
- Recovery testing
  - forces the software to fail in a variety of ways and verifies that recovery is properly performed
- Security testing
  - 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

Debugging: A Diagnostic Process

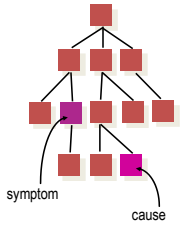
The Debugging Process



Debugging Effort

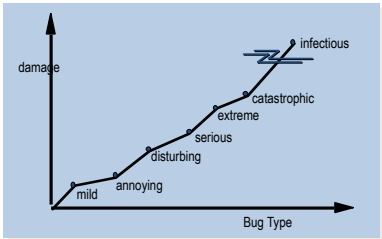


Symptoms & Causes



- 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 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.

Lecture 22

Software Testing Techniques

- Testability
- White-Box Testing
- Basis Path Testing
- Control Structure Testing
- Black-Box Testing
- Comparison Testing
- Orthogonal Array Testing
- Testing Patterns

Testability

- Operability—it operates cleanly
- Observability—the results of each test case are readily observed
- Controllability—the degree to which testing can be automated and optimized
- Decomposability—testing can be targeted
- Simplicity—reduce complex architecture and logic to simplify tests
- Stability—few changes are requested during testing
- Understandability—of the design

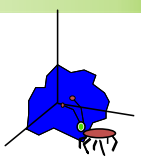
What is a "Good" Test?

- 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

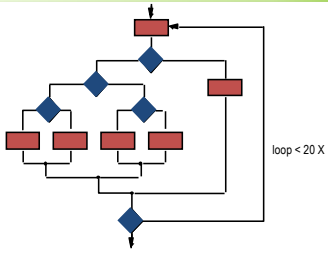
"Bugs lurk in corners and congregate at boundaries ..."

Boris Beizer



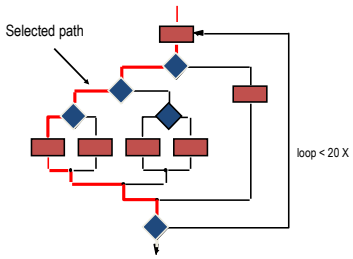
OBJECTIVE	to uncover errors
CRITERIA	in a complete manner
CONSTRAINT	with a minimum of effort and time

Exhaustive Testing

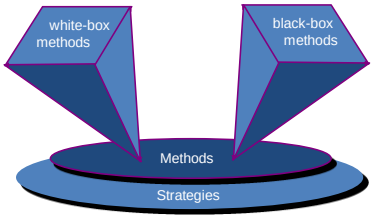


There are 10^{14} possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!

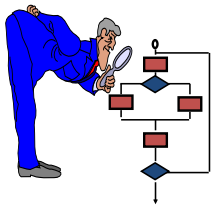
Selective Testing



Software Testing



White-Box Testing

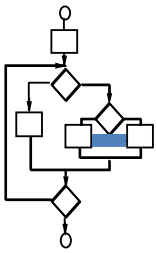


... our goal is to ensure that all statements and conditions have been executed at least once ...

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



First, we compute the cyclomatic complexity:

number of simple decisions + 1

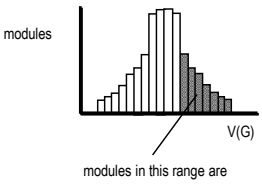
or

number of enclosed areas + 1

In this case, V(G) = 4

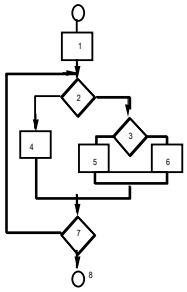
Cyclomatic Complexity

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

Basis Path Testing



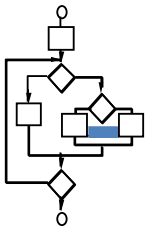
Next, we derive the independent paths:

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

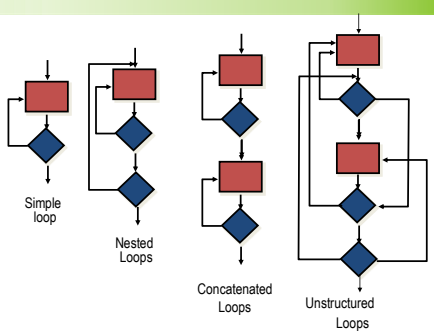
Graph Matrices

- A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on a flow graph
- Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.
- By adding a link weight to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing

Control Structure Testing

- Condition testing — a test case design method that exercises the logical conditions contained in a program module
- Data flow testing — selects test paths of a program according to the locations of definitions and uses of variables in the program

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

Loop Testing: Nested Loops

Nested Loops

Start at the innermost loop. Set all outer loops to their minimum iteration parameter values.

Test the min+1, typical, max-1 and max for the innermost loop, while holding the outer loops at their minimum values.

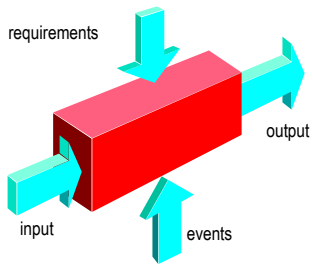
Move out one loop and set it up as in step 2, holding all other loops at typical values. Continue this step until the outermost loop has been tested.

Concatenated Loops

If the loops are independent of one another
then treat each as a simple loop
else* treat as nested loops
endif*

for example, the final loop counter value of loop 1 is used to initialize loop 2.

Black-Box Testing



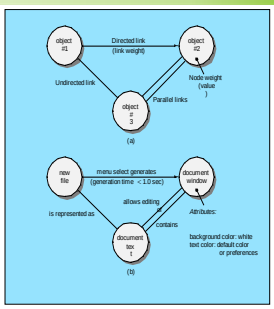
Black-Box Testing

- 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?

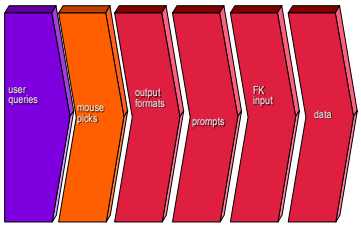
Graph-Based Methods

To understand the objects that are modeled in software and the relationships that connect these objects

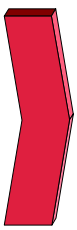
In this context, we consider the term "objects" in the broadest possible context. It encompasses data objects, traditional components (modules), and object-oriented elements of computer software.



Equivalence Partitioning



Sample Equivalence Classes



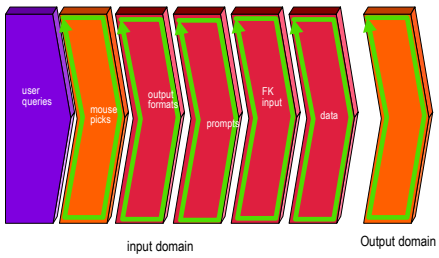
Valid data

- user supplied commands
- responses to system prompts
- file names
- computational data
  - physical parameters
  - bounding values
  - initiation values
- output data formatting
- 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

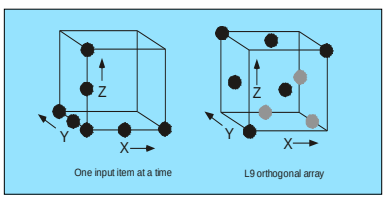


Comparison Testing

- Used only in situations in which the reliability of software is absolutely critical (e.g., human-rated systems)
  - Separate software engineering teams develop independent versions of an application using the same specification
  - Each version can be tested with the same test data to ensure that all provide identical output
  - Then all versions are executed in parallel with real-time comparison of results to ensure consistency

Orthogonal Array Testing

- Used when the number of input parameters is small and the values that each of the parameters may take are clearly bounded



OOT—Test Case Design

Berard [BER93] proposes the following approach:

1. Each test case should be uniquely identified and should be explicitly associated with the class to be tested,
2. The purpose of the test should be stated,
3. A list of testing steps should be developed for each test and should contain [BER94]:
   a. a list of specified states for the object that is to be tested
   b. a list of messages and operations that will be exercised as a consequence of the test
   c. a list of exceptions that may occur as the object is tested
   d. a list of external conditions (i.e., changes in the environment external to the software that must exist in order to properly conduct the test)
   e. supplementary information that will aid in understanding or implementing the test.

Testing Methods

- Fault-based testing
  - The tester looks for plausible faults (i.e., aspects of the implementation of the system that may result in defects). To determine whether these faults exist, test cases are designed to exercise the design or code.
- Class Testing and the Class Hierarchy
  - Inheritance does not obviate the need for thorough testing of all derived classes. In fact, it can actually complicate the testing process.
- Scenario-Based Test Design
  - Scenario-based testing concentrates on what the user does, not what the product does. This means capturing the tasks (via use-cases) that the user has to perform, then applying them and their variants as tests.

OOT Methods: Random Testing

- Random testing
  - identify operations applicable to a class
  - define constraints on their use
  - identify a miminum test sequence
    - an operation sequence that defines the minimum life history of the class (object)
  - generate a variety of random (but valid) test sequences
    - exercise other (more complex) class instance life histories

OOT Methods: Partition Testing

- Partition Testing
  - reduces the number of test cases required to test a class in much the same way as equivalence partitioning for conventional software
  - state-based partitioning
    - categorize and test operations based on their ability to change the state of a class
  - attribute-based partitioning
    - categorize and test operations based on the attributes that they use
  - category-based partitioning
    - categorize and test operations based on the generic function each performs

OOT Methods: Inter-Class Testing

- Inter-class testing
  - For each client class, use the list of class operators to generate a series of random test sequences. The operators will send messages to other server classes.
  - For each message that is generated, determine the collaborator class and the corresponding operator in the server object.
  - For each operator in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits.
  - For each of the messages, determine the next level of operators that are invoked and incorporate these into the test sequence

OOT Methods: Behavior Testing

The tests to be designed should achieve all state coverage [KIR94]. That is, the operation sequences should cause the Account class to make transition through all allowable states

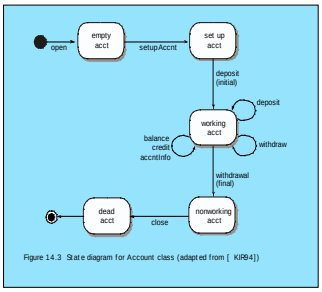


Figure 14.3 State diagram for Account class (adapted from [KIR94])

Testing Patterns

Khulna University

Pattern name: pair testing

Abstract: A process-oriented pattern, pair testing describes a technique that is analogous to pair programming (Chapter 4) in which two testers work together to design and execute a series of tests that can be applied to unit, integration or validation testing activities.

Pattern name: separate test interface

Abstract: There is a need to test every class in an object-oriented system, including "internal classes" (i.e., classes that do not expose any interface outside of the component that used them). The separate test interface pattern describes how to create "a test interface that can be used to describe specific tests on classes that are visible only internally to a component." [LAN01]

Pattern name: scenario testing

Abstract: Once unit and integration tests have been conducted, there is a need to determine whether the software will perform in a manner that satisfies users. The scenario testing pattern describes a technique for exercising the software from the user's point of view. A failure at this level indicates that the software has failed to meet a user visible requirement. [KAN01]

THANK YOU