Software Testing

北科大資工系
劉建宏老師

Software Testing

- Software Testing Fundamentals
  - Verification And Validation
  - The V Model
- Testing Principles
- Testing Techniques
  - White-Box Testing Techniques
  - Black-Box Testing Techniques
- Software Testing Strategies
  - Unit Testing
  - Integration Testing
  - Regression Testing
  - System Testing
- Object-oriented Testing

2

Software Testing Fundamentals

- What is software testing?
  - Myers: The process of executing a program with the intent of finding errors.
  - Beizer: The act of designing and executing tests.
  - Whittaker: The process of executing a software system to determine whether it matches its specification and executes in its intended environment
  - IEEE: The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of system or component.

3

Software Testing Fundamentals

- Why do we test?
  - To check if there are any errors in a part or a product.
  - To gain the confidence in the correctness of the product.
  - To ensure the quality and satisfaction of the product
- Testing vs. debugging
  - Testing is to show that a program has bugs
  - Debugging is to locate and correct the error or mis-conception that cause the program failures

4

Software Testing Fundamentals

- Who Tests the Software?
  - Development engineers
    - Understand the system, but test "gently"
    - Driven by "delivery"
    - Only perform unit tests and integration tests
  - Test engineers
    - Need to learn the system, but attempt to break it
    - Driven by "quality"
    - Define test cases, write test specifications, run tests, analyze results
  - Customers
    - Driven by "requirements"
    - Determine if the system satisfies the acceptance criteria

5

How to do Software Testing?

- Software testing can be approached in the following phases
  - Modeling the software's environment by simulation
  - Selecting test scenario
  - Running and evaluating test scenario
  - Measuring testing progress

6

Verification and Validation

- Software testing is one element of a broader topic: verification and validation (V&V)
- Verification – are we building the product correctly?*
  - The set of activities to ensure that software correctly implements specific functions
- Validation – are we building the correct product?*
  - The set of activities to ensure that the developed software is traceable to customer requirements

Note: [Boehm 1981]

7

Verification and Validation

- The definition of V&V encompasses many activities that is referred to as software quality assurance (SQA) including
  - Formal technical review
  - Quality and configuration audits
  - Performance monitoring
  - Simulation
  - Feasibility study
  - Documentation review
  - Database review
  - Algorithm analysis
  - Development testing
  - Qualification testing
  - Installation testing

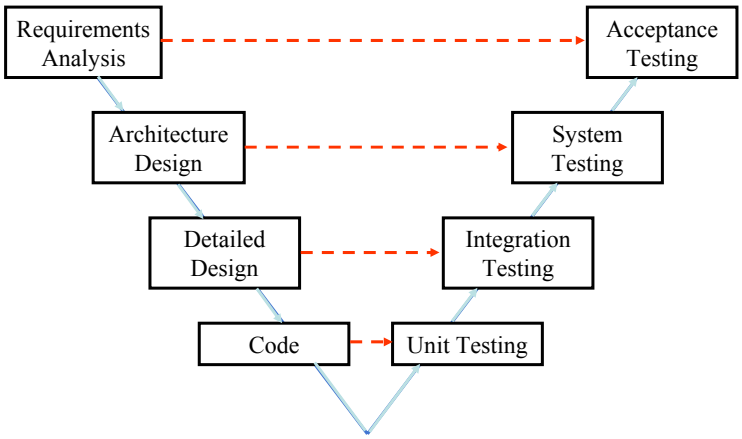
8

The V model

- The V model
  - Emerged in reaction to some waterfall models that showed testing as a single phase following the traditional development phases of requirements analysis, high-level design, detailed design and coding.
  - The V model portrays several distinct testing levels and illustrates how each level addresses a different stage of the software life cycle.
  - The V shows the typical sequence of development activities on the left-hand (downhill) side and the corresponding sequence of test execution activities on the right-hand (uphill) side.

9

The V model



10

The V model

- The V model is valuable because it highlights the existence of several levels of testing and describes how each relates to a different development phase:
  - Unit testing: concentrates on each unit (i.e., component) of the software (white box)
  - Integration testing: focuses on design and the construction of the software architecture (black box, limited amount of white box)

11

The V model

  - System testing: verifies that all elements mesh properly and that overall system function or performance is achieved.
  - Acceptance testing: are ordinarily performed by the business/users to confirm that the product meets the business requirements.

12

Testing Principles

- Davis in his book, *201 Principles of Software Development*, suggests a set of testing principles:
  - All tests should be traceable to customer requirements.
  - Tests should be planned long before testing begins.
  - The Pareto principle applies to software 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".

13

Testing Principles

  - Exhaustive testing is not possible.
  - To be most effective, testing should be conducted by an independent third party.
- Glen Myers suggests a number of testing principles:
  - Testing is a process of executing a program with the intent of finding errors.
  - A good test case is one that has high probability of detecting an as-yet undiscovered error
  - A successful test case is one that detects an as-yet undiscovered error

14

Attributes of A Good Test

- Kaner, Falk, and Nguyen in their book, *Testing Computer Software*, suggest the following attributes of 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

15

Exhaustive Testing

- Is it possible to develop test cases to exercise all logical paths of a program?
  - Even small programs, the number of possible logical paths can be very large
  - Example: Consider a program contains 2 nested loops that each executes 1 to 20 times depending on the input data. The interior loop has 4 if-then-else constructs
    - There are 10^{14} possible paths! If we exercise one test per millisecond, it would take 3170 years to complete test
- Exhaustive testing is impractical for large software system
  - Selective testing is required

16

When Testing Can Stop?

- Team consensus
- Marginal cost
  - If the cost of finding that defect exceeds the loss incurred to the organization, ship the product with that defect
- Test adequate criteria are met
  - Coverage
  - Error discovery rate
- Testing is never done, the burden simply shifts from you to the customer
- When the product has been irrevocably retired

17

Testing Methods

- Two general software testing methods:
  - White-box testing: (logic-driven)
    - Design tests to exercise internal structures of the software to make sure they operates according to specifications and designs
  - Black-box testing: (data-driven or input/output-driven)
    - Design tests to exercise each function of the software and check its errors.
  - White-box and black-box testing approaches can uncover different class of errors and are complement each other

18

White-Box Testing

- White-box testing
  - Also known as glass-box testing or structural testing
  - Has the knowledge of the program's structures
  - A test case design method that uses the control structure of the procedural design to derive test cases
  - Focus on the control structures, logical paths, logical conditions, data flows, internal data structures, and loops.
  - W. Hetzel describes white-box testing as "testing in the small"

19

White-Box Testing

- Using white-box testing methods, we can derive test cases that
  - Guarantee that all independent paths within a module have been exercised at least once.
  - Exercise all logical decisions on their true and false sides.
  - Execute all loops at their boundaries and within their operational bounds.
  - Exercise internal data structures to assure their validity.

20

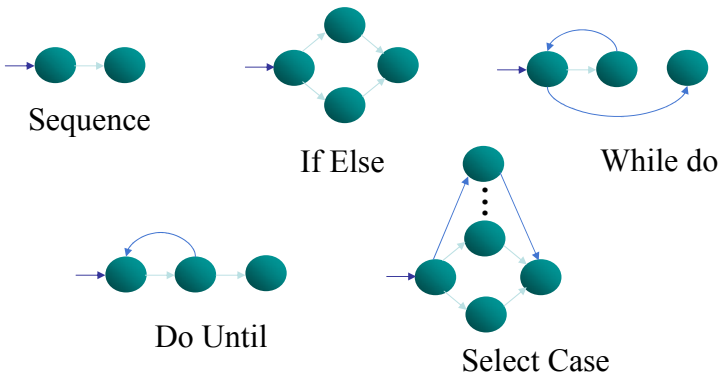
Basis Path Testing

- Basic path testing (a white-box testing technique):
  - First proposed by Tom McCabe.
  - Can be used to derive a logical complexity measure for a procedure design.
  - Used as a guide for defining a basis set of execution path.
  - Guarantee to execute every statement in the program at least one time.

21

Basis Path Testing

- The basic structured-constructs in a flow graph :



22

Basis Path Testing

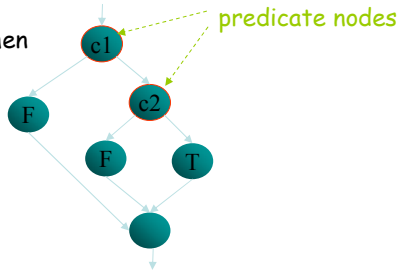
- Flow graph notation (control flow graph)
  - Node represents one or more procedural statements.
    - A sequence of process boxes and a decision diamond can map into a single node
    - A predicate node is a node with two or more edges emanating from it
  - Edge (or link) represents flow of control
  - Region: areas bounded by edges and nodes
    - When counting regions, include the area outside the graph as a region

23

Basis Path Testing

- Compound condition
  - Occurs when one or more Boolean operators (OR, AND, NAND, NOR) is present in a conditional statement
  - A separate node is created for each of the conditions *C1* and *C2* in the statement *IF C1 AND C2*

```
if (c1 AND c2) then
   print T;
else
   print F;
end if;
```



24

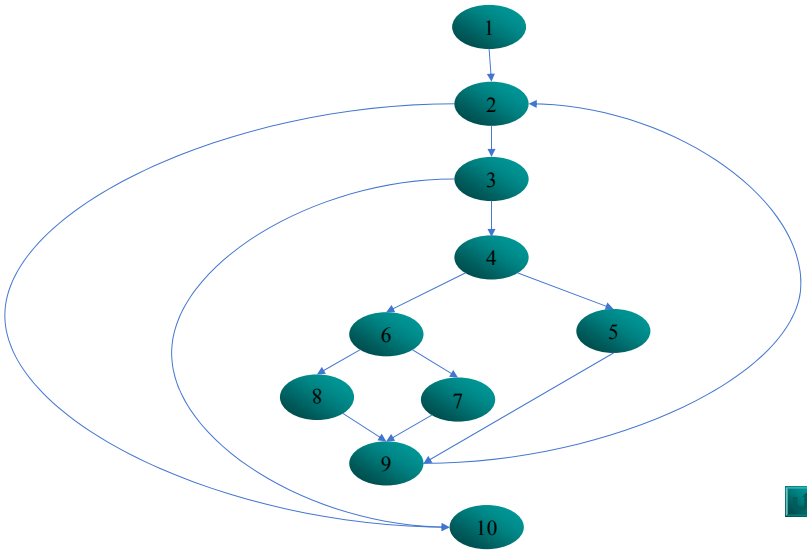
binarySearch() Example

```
public int binarySearch(int sortedArray[ ], int searchValue)
{
    int bottom = 0;
    int top = sortedArray.length - 1;
    int middle, locationOfsearchValue;
    boolean found = flase;
    locationOfsearchValue = -1;        /* the location of searchValue in the sortedArray   */
                                       /* location = -1 means that searchValue is not found */
    while ( bottom <= top && !found)
    {
        middle = (top + bottom)/2;
        if (searchValue == sortedArray[ middle ])
        {
            found = true;
            locationOfsearchValue = middle;
        }
        else if (searchValue < sortedArray[ middle ])
                top = middle - 1;
        else
                bottom = middle + 1;
    } // end while
    return locationOfsearchValue;
}
```

1 2 3 4 5 6 7 8 9 10

25

The CFG of Function binarySearch()



26

Cyclomatic Complexity

- Cyclomatic complexity is a software metric
  - provides a quantitative measure of the global complexity of a program.
  - When this metric is used in the context of the basis path testing
    - the value of cyclomatic complexity defines the number of independent paths in the basis set of a program
    - the value of cyclomatic complexity defines an upper bound of number of tests (i.e., paths) that must be designed and exercised to guarantee coverage of all program statements

27

Cyclomatic Complexity

- Independent path
  - An independent path is any path of the program that introduce at least one new set of procedural statements or a new condition
  - An independent path must move along at least one edge that has not been traversed before the path is defined
    - Examples: consider the CFG of binarySearch()
      - Path 1: 1-2-10
      - Path 2: 1-2-3-4-6-8-9-2-10
      - Path 3: 1-2-3-4-6-8-9-2-3-10

      } independent paths
      - Path 4: 1-2-3-4-6-8-9-2-3-4-6-8-9-2-10 (not an independent path)

28

Cyclomatic Complexity

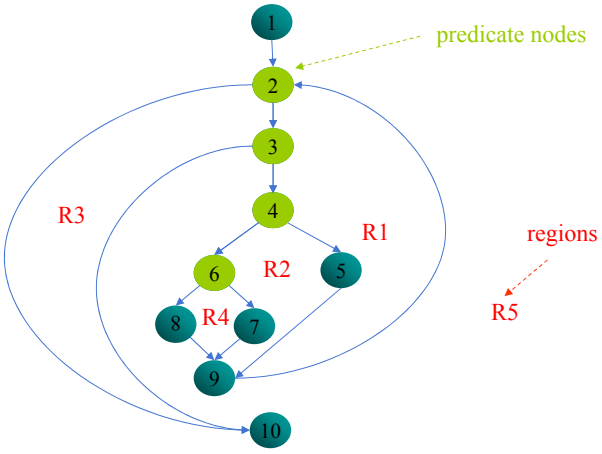
- Three ways to compute cyclomatic complexity:
  - The number of regions of the flow graph correspond to the cyclomatic complexity.
  - Cyclomatic complexity, V(G), for a flow graph G is defined as V(G) = E - N + 2 (13-10+2=5)

    where E is the number of flow graph edges and N is the number of flow graph nodes.
  - Cyclomatic complexity, V(G) = P + 1

    where P is the number of predicate nodes contained in the flow graph G.

29

Cyclomatic Complexity of Function binarySearch()



30

Deriving Basis Test Cases

- The following steps can be applied to derive the basis set:
  1. Using the design or code as a foundation, draw the corresponding flow graph.
  2. Determine the cyclomatic complexity of the flow graph.
     - V(G) = 5 regions
     - V(G) = 13 edges – 10 nodes + 2 = 5
     - V(G) = 4 predicate nodes + 1 = 5

31

Deriving Basis Test Cases

3. Determine a basis set of linearly independent paths.
   - Path 1: 1-2-10
   - Path 2: 1-2-3-10
   - Path 3: 1-2-3-4-5-9-2- …
   - Path 4: 1-2-3-4-6-7-9-2-…
   - Path 5: 1-2-3-4-6-8-9-2-…
4. Prepare test cases that force the execution of each path in the basis set
   - Path 1 test case:
     - Inputs: sortedArray = { }, searchValue = 2
     - Expected results: locationOfSearchValue = -1

32

Deriving Basis Test Cases

- Path 2 test case: cannot be tested stand-alone!
  - Inputs: sortedArray = {2, 4, 6}, searchValue = 8
  - Expected results: locationOfSearchValue = -1
- Path 3 test case:
  - Inputs: sortedArray = {2, 4, 6, 8, 10}, searchValue = 6
  - Expected results: locationOfSearchValue = 2
- Path 4 test case:
  - Inputs: sortedArray = {2, 4, 6, 8, 10}, searchValue = 4
  - Expected results: locationOfSearchValue = 1
- Path 5 test case:
  - Inputs: sortedArray = {2, 4, 6, 8, 10}, searchValue = 10
  - Expected results: locationOfSearchValue = 4

33

Deriving Basis Test Cases

- Each test cases is executed and compared to its expected results.
- Once all test cases have been exercised, we can be sure that all statements are executed at least once
- Note: some independent paths cannot be tested stand-alone because the input data required to traverse the paths cannot be achieved
  - In binarySearch(), the initial value of variable *found* is `FALSE`, hence path 2 can only be tested as part of path 3, 4, and 5 tests

34

Graph Matrices

- A graph matrix
  - A tabular representation of a flow graph
  - A square matrix with a size equal to the number of nodes on the flow graph ➔ row
  - Matrix entries correspond to the edges between nodes
  - Adding link weight to each edge to represent
    - The connection between nodes
    - The probability of the edge to be executed
    - The resource (e.g., processing time or memory) required for traversing the edge

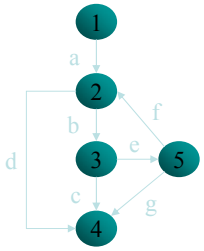
35

Graph Matrices

- A connection matrix
  - A graph matrix with the link weight is 1 (representing a connection exists) or 0 (representing a connection does not exist)
  - Each row of the matrix with two or more entries represents a predicate node
  - Provide another method for computing the cyclomatic complexity of a flow graph

36

Graph Matrices



Flow Graph

Node \ Connected to node	1	2	3	4	5
1		a			
2			b	d	
3				c	e
4					
5		f		g	

Graph Matrix

	1	2	3	4	5
1		1			
2			1	1	
3				1	1
4					
5		1		1	

Connection Matrix

Connections
1 - 1 = 0
2 - 1 = 1
2 - 1 = 1
2 - 1 = 1
3 + 1 = 4

Cyclomatic complexity

37

Condition Testing

- Condition Testing
  - Test case design method that exercise the logical conditions in a program module
- Logical conditions
  - Simple condition:
    - (a rel-op b) where rel-op={<, ≤, =, ≠, ≥, >} (may be negated with NOT)
    - Example: a ≤ b; NOT(a ≤ b)

38

Condition Testing

- Compound condition:
  - Two or more simple conditions connected with AND, OR
  - Example: (a > b) AND (c < d)
- Relational expression:
  - (E1 rel-op E2) where E1 and E2 are arithmetic expressions
  - Example: ((a*b+c)>(a+b+c))
- Boolean expression
  - A condition without relational expressions

- If a condition is incorrect, then at least one component of the condition is incorrect

39

Condition Testing

- The type of errors in a condition include :
  - Boolean operator (incorrect/missing/extra Boolean operators)
  - Boolean variable error
  - Boolean parenthesis error
  - Relational operator error
  - Arithmetic expression error
- The purpose of condition testing is to detect not only errors in the conditions of a program but also other errors in the program.
  - Detect faults in the condition, statements executed before the condition, and the statements executed after the condition

40

Condition Testing

- Branch testing
  - The simplest condition testing strategy that requires each decision point and each possible branch to be executed at least once
  - For a compound condition C, test (1) true and false branches of C and (2) true and false branches of every simple condition of C at least once
  - Example: for C = (a>b) AND (c<d) we test for
    - C TRUE and FALSE
    - a>b TRUE and FALSE
    - c<d TRUE and FALSE

41

BRO Testing

- K.C. Tai suggests a condition testing strategy called branch and relational operator (BRO) testing.
  - Can detect branch and relational operator errors in a condition provided that all Boolean variables and relational operators in the condition occur only once and have no common variable
  - Uses condition constraint for a condition C which is defined as $(D_1, D_2, \ldots, D_n)$ where D_i specifies the constraint on the outcome of i-th condition of C

42

BRO Testing

- Boolean variables have constraint on the outcome that must be either True (t) or False (f)
- For relational expression, symbols <, >, = are used to specify the constraints on the outcome of the expression
- A condition constraint D for condition C is said to be covered by an execution of C if, during the execution of C, the outcome of each simple condition in C satisfies the corresponding constraint in D.

43

BRO Testing Examples

- Example C1: B1 & B2
  - Constraint set = (D1, D2) = {(t,t), (t,f), (f,t)}
  - If C1 is incorrect due to one or more Boolean operator errors, at least one of the constraint set will force C1 to fail
- Example C2: B1 & (E3=E4)
  - Constraint set = (D1, D2) = {(t,=), (t,>), (t,<), (f,=)}
  - Coverage of the preceding constraint set will detect the Boolean and relational operator errors in C2
- Example C3: (E1 > E2) & (E3=E4)
  - Constraint set = {(>,=), (<,=), (>,>), (>,<), (=,=)}
  - Coverage of the preceding constraint set will detect the relational operator errors in C3

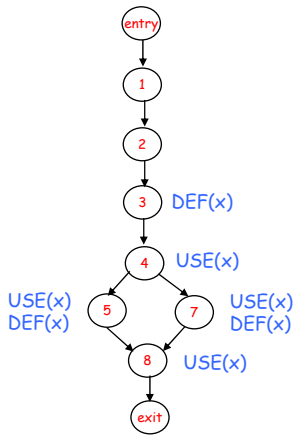
44

Data Flow Testing

- Data flow testing
  - A testing technique that selects test paths of a program according to the locations of definitions and uses of variables in the program.
  - DEF(S) = {X| statement S contains a definition of X}
  - USE(S) = {X| statement S contains a use of X}
  - The definition of variable X at statement S is said to be live at statement S' if there exists a path from statement S to statement S' that contains no other definition of X
  - A definition-use (DU) chain of variable X is of the form [X, S, S'], where S and S' are statement numbers, X is in DEF(S) and USE(S'), and the definition of X in statement S is live at statement S'.

45

Data Flow Testing Example




```
public void Odd() {
 1    int x;
 2    printf("Enter a number: ");
 3    scanf("%d", &x);
 4    if x%2 == 0
 5      x=x+1;
 6    else
 7      x=2*x;
 8    printf ("x = %d\n", x);
}
```

46

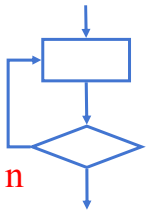
Data Flow Testing Example

- DU testing strategy
  - A strategy for selecting the derived DU chains as test cases
    - All use coverage: every DU chain to be covered at least once
    - DU Chains of the Odd() Example
      - (x, 3, 4), (x, 3, 5), (x, 3, 7)
      - (x, 5, 8), (x, 7, 8)
    - (x, 3, 8) is NOT a DU chain since the value of x at Line 3 is redefined at Lines 5 and 7 before it reaches the use at Line 8
    - Test paths selected according to all use coverage:
      - path1 1-2-3-4-7-8 cover (x, 3, 4), (x, 3, 7), (x, 7, 8)
      - path2 1-2-3-4-5-8 cover (x, 3, 4), (x, 3, 5), (x, 5, 8)

47

Loop Testing

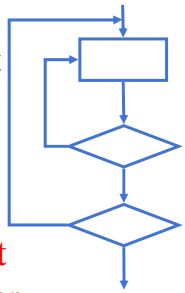
- Simple Loops (n iterations)
  - skip the loop entirely
  - only one pass through the loop
  - two passes through the loop
  - m passes through the loop where m < n
  - n-1, n, n+1 passes through the loop



48

Loop Testing

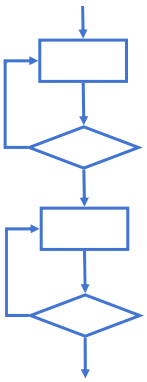
- Nested Loops
  - Start from the innermost loop; set all other loops to minimum values
  - conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter values; test out-of-range values
  - work outward, conducting tests for the next innermost loop while keeping all other outer loops at their min. iteration and other nested loops at their typical values
  - continue until all loops have been tested



49

Loop Testing

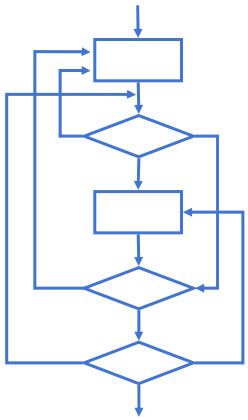
- Concatenated Loops
  - independent loops using simple loop testing
  - dependent loops (i.e., loop counter value of loop1 is used in loop2) using nested loop testing



50

Loop Testing

- Unstructured Loops
  - Whenever possible, redesign the code to reflect the use of structured programming constructs.
  - Then apply the appropriate loop testing strategy.



51

Black-Box Testing

- Black-box testing
  - Also known as functional testing, behavioral testing, or specification-based testing
  - Does not have the knowledge of the program's structures
  - Discover program errors based on program requirements and product specifications
  - Derive sets of inputs to fully exercise all functional requirements for a program
  - Complementary to white-box testing

52

Black-Box Testing

- Focuses on the functional requirements of the software including functions, operations, external interfaces, external data and information
- Attempts to find errors in the following:
  - Incorrect of missing functions
  - Interface errors
  - Errors in data structures or external database access
  - Performance errors
  - Initialization and termination errors

53

Random Testing

- Random testing is a method that select test cases randomly from the entire input domain of the program.
  - The input values of each test case are randomly generated
  - The overall distribution of the test cases need to conform to the distribution of input domain or operational profile.
    - Example: for an ATM system that 80% transactions are withdrawal transactions and 20% transactions are transfer transactions
    - 80% test cases will be generated randomly to test withdrawal transaction and 20% test cases will be generated randomly to test transfer transaction
  - Random testing is considered to be efficient since test cases can be generated automatically
  - However, to obtain a precise operational profile is very difficult

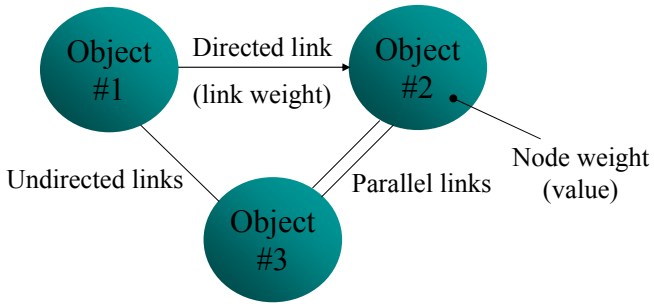
54

Graph-Based Testing Methods

- Graph-Based testing is a method that derives test cases from a graphical model representing the objects and their relationships of a program
  - Create a graphical model to represent the programs in terms of objects and relationships
  - Derive test cases by traversing the graph and covering each object and relationship to verify "all objects have the expected relationship to one another"
- A graph consists of
  - nodes representing objects
  - links representing relationships between nodes
  - node weights describing properties of a node
  - link weight describing some characteristic of a link

55

Graph Notation

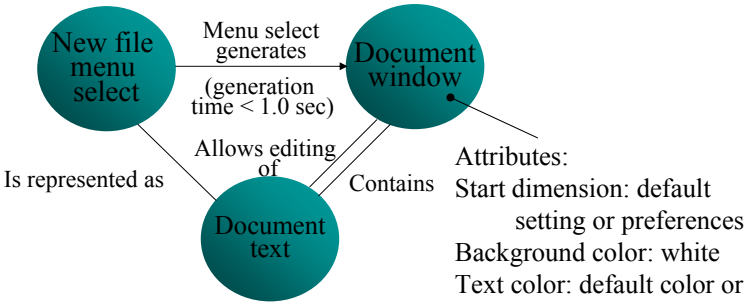


- Directed link: indicates that a relationship moves in only one direction
- Bidirectional link (or symmetric link): implies that the relationship applies in both directions
- Parallel link: indicates a number of different relationships are established between graph nodes

56

Note: Redraw from [Pressman 2004]

A Simple Example



Note: Redraw from [Pressman 2004]

57

Graph-Based Testing Methods

- Graph can be used in a number of behavioral testing
  - Transaction flow modeling
    - Nodes representing steps in some transaction
    - Links representing logical connection between steps
  - Finite state modeling
    - Nodes representing user observable states of the software
    - Links representing transition that occur to move from state to state

58

Graph-Based Testing Methods

- Data flow modeling
  - Nodes representing data object
  - Links representing the transformations that occur to translate one data object into another
- Timing modeling
  - Nodes representing program objects
  - Links representing sequential connections between objects
  - Link weights used to specify required execution times

59

Partition Testing

- Partition testing
  - The input domain of the program is partitioned into different disjointed subdomains
  - Ideally, the partition divides the domain into subdomains with property that within each subdomain, either the program produces the correct answer or the program produce an incorrect answer for every element
  - Only an element randomly selected from each subdomain is needed for testing the program in order to determine program faults

60

Equivalence Partitioning

- Equivalence partitioning
  - The input domain of a program is partitioned into a finite number of equivalence classes from which test cases are derived
  - An equivalence class consists of a set of data that is treated the same by the program or that should generate the same result
  - Test case design is based on an evaluation of equivalence classes for an input condition
  - Can reduce the total number of test cases to be developed

61

Equivalence Partitioning

- The equivalence classes are identified based on the set of valid or invalid states for each input condition
- An input condition can be
  - A specific numeric value
  - A range of values
  - A set of related values
  - A Boolean condition

62

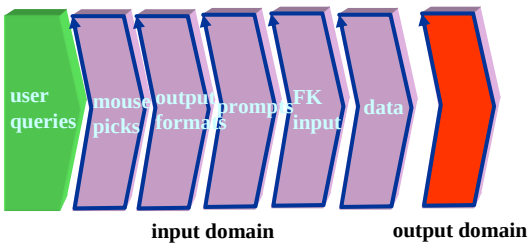
Equivalence Partitioning

- The following guidelines can help to define the equivalence classes [Pressman 2004]:
  - If an input condition specifies a range, one valid and two invalid equivalence class are defined.
  - If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
  - If an input condition specifies a member of a set, one valid and one invalid equivalence classes are defined.
  - If an input condition is Boolean, one valid and one invalid classes are defined.

63

Boundary Value Testing

- Based on programming experience, more errors are found at the boundaries of an input/output domain than in the "center".
- In addition to select test data "inside" an equivalence class, data at the "edges" of the equivalence class also need to be examined



Note: Redraw from [Pressman 2004]

64

Equivalence Partitioning Example

- Consider we are writing a program for computing letter grades based on the numerical scores of students, where the input variable is *Score*. The rule of computing the grade is as follows:

Score	Grade
90~100	A
80~89	B
70~79	C
60~69	D
0~59	F

65

Equivalence Partitioning Example

- The input domain of *Score* can be partitioned into 5 valid equivalence classes and 2 invalid equivalence classes
  - Valid classes: 0~59, 60~69, 70~79, 80~89, 90~100
  - Invalid classes: smaller than 0 and greater than 100
- Any data value within a class is considered equivalence in terms of testing
- Using the equivalence partitioning testing, we can reduce the test cases from 100 (assume 0 <= score <= 100) to 7

66

Boundary Value Testing

- Boundary value analysis (BVA)
  - A test case design technique complements to equivalence partition
  - Rather selecting any element from an equivalence class, BVA leads to the selection of test cases that exercise bounding values ("edge" of the class)
  - Unlike equivalence partition that derives test cases only from input conditions, BVA derives test cases from both input conditions and output domain

67

Boundary Value Testing

- Guidelines [Pressman 2004]:
  1. If an input condition specifies a range [*a*, *b*], test cases should be designed with value *a* and *b*, just above and below *a* and *b*
     - Example: Integer D with input condition [-3, 5], BVA test values are -3, 5, -2, 6, -1, 4
  2. If an input condition specifies a number values, test cases should be developed to exercise the minimum, number, maximum number, and values just above and below minimum and maximum
     - Example: Enumerate data E with input condition: {3, 5, 100, 102}, BVA test values are 3, 102, 2, 4, 101, 103

68

Boundary Value Testing

3. Guidelines 1 and 2 are applied to output condition
4. If internal program data structures have prescribed boundaries, be certain to design a test case to exercise the data structure at its boundary
   - Array input condition:
     - Empty, single element, full element, out-of-boundary
   - Search output condition:
     - Element is inside array or the element is not

Boundary Value Testing Example

- Consider the letter grade assignment program in previous example
- The input domain of *Score* can be partitioned into 5 valid equivalence classes and 2 invalid equivalence classes
  - Valid classes: 0~59, 60~69, 70~79, 80~89, 90~100
  - Invalid classes: smaller than 0 and greater than 100
- With BVA, we can obtain the following test values

Classes	Just below minimum	Minimum	Just above minimum	Just below maximum	Maximum	Just above maximum
>100	100	101	102	-	-	-
90~100	89	90	91	99	100	101
80~89	79	80	81	88	89	90
70~79	69	70	71	78	79	80
60~69	59	60	61	68	69	70
0~59	-1	0	1	58	59	60
<0	-	-		-2	-1	0

Error Guessing

- Identify potential errors and design test cases based on intuition and experiences
- Test cases can be derived by making a list of possible errors or error-prone situations
  - Empty or null lists
  - Zero instances or occurrences
  - Blanks or null strings
  - Negative numbers
  - Historical defects (need to maintain defect history)

71

A Strategic Approach to Software 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 software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy

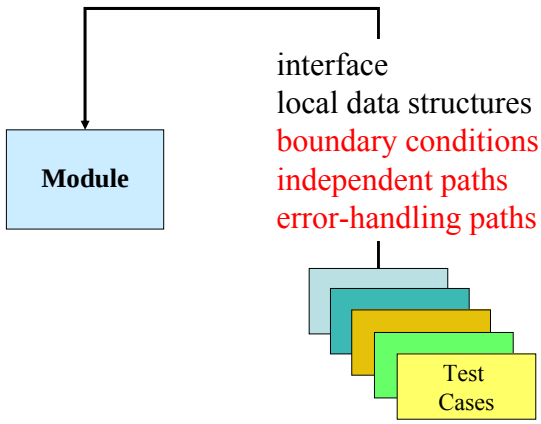
72

Unit Testing

- Unit testing
  - Is normally considered as an adjunct to the coding step
  - Focuses verification effort on the smallest unit of software design – the software component or module
  - Using the component-level design description as a guide
  - Provide a release criterion for a programming task
  - Encourage early defect discovery
  - Discourage big-bang integration with its late defect discovery
  - Find "side effect" defects, especially in highly coupling designs.
  - Is white-box oriented and the step can be conducted in parallel for multiple components

73

Unit Testing



Note: Redraw from [Pressman 2004]

74

Considerations of Unit Test

- Interface
  - Ensures that information properly flows in and out of the component
- Local data structures
  - Ensures that data stored temporarily maintains its integrity during execution
- Boundary conditions
  - Ensures that the component operates properly at boundaries established to limit or restrict processing
- Independent paths (basis paths)
  - Ensures that all paths in a component have been executed at least once
- Error-handling paths
  - Ensures that errors are correctly handled

75

Unit Test Cases

- Unit test cases should be designed to uncover errors due to erroneous computation, incorrect comparisons, or improper control flow
- Common errors in computation
  - Misunderstood or incorrect arithmetic precedence
  - Mixed mode operations
  - Incorrect initialization
  - Precision inaccuracy
  - Incorrect symbolic representation of an expression

76

Unit Test Cases

- Common errors in comparison and control flow
  - Comparison of different data types
  - Incorrect logical operators or precedence
  - Expectation of equality when precision error makes equality unlikely
  - Incorrect comparison of variables
  - Improper or nonexistent loop termination
  - Failure to exit when divergent iteration is encountered
  - Improperly modified loop variables

77

Unit Test Cases

- What should be tested when error handling is evaluated?
  - Error description is unintelligible
  - Error noted does not correspond to error encountered
  - Error condition causes system intervention prior to error handling
  - Exception-condition processing is incorrect
  - Error description does not provide enough information to assist in the location of the cause of the error

78

Unit Test Cases

- Boundary testing
  - An important task of the unit test step
  - Test cases that exercise data structure, control flow, and data values just below, at, and just above maxima and minima are vary likely to uncover errors

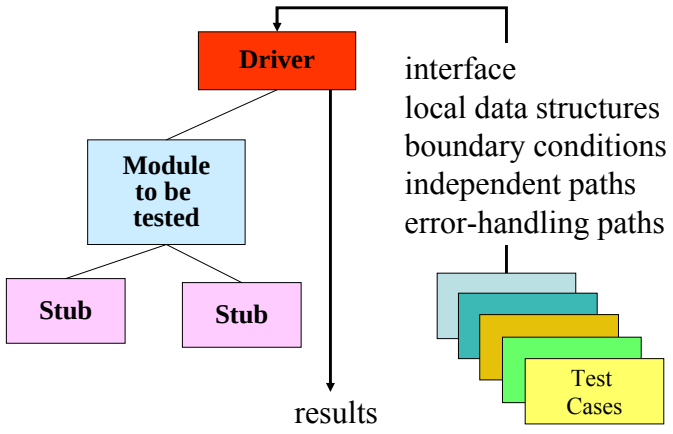
79

Unit Test Procedure

- Review of design information (provided as guidance) to establish test cases that are likely to uncover errors previous noted
- Each test case should be couple with a set of expected results
- Develop driver and/or stub software for each unit test
  - Driver: nothing more than a “main program” that accept test case data, passes such data to the component to be tested, and prints relevant results
  - Stub: serves to replace modules that are subordinate called by the component to be tested.
  - Drivers and stubs represent overhead
- Execute and evaluate the unit test

80

Unit Test Environment



Note: Redraw from [Pressman 2004]

81

JUnit

- JUnit
  - An open source Unit Test Framework for Java (http://www.junit.org)
    - Provides the test drivers for unit testing
    - Provides automatic test runs
    - Provides automatic result checks
  - The steps of using JUnit
    - Write a test case
    - Run the test
    - Verify the result

82

JUnit

- Write a test case
  - Create your own test case as a subclass of JUnit `TestCase`.
  - Override the `setUp()` method to initialize object(s) under test
  - Override the `tearDown()` method to release object(s) under test
- Run the test
  - Define a public `test???()` method for exercising the object(s) under test

83

JUnit

- Verify the result
  - Assert expected result of the test case using `assertEquals()`
- Error vs. Failures
  - Error: unanticipated problem like an `ArrayIndexOutOfBoundsException`
  - Failure: is anticipated and can be checked with assertions

84

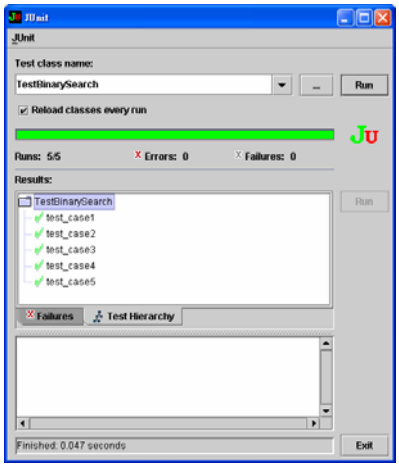
JUnit Example for binarySearch

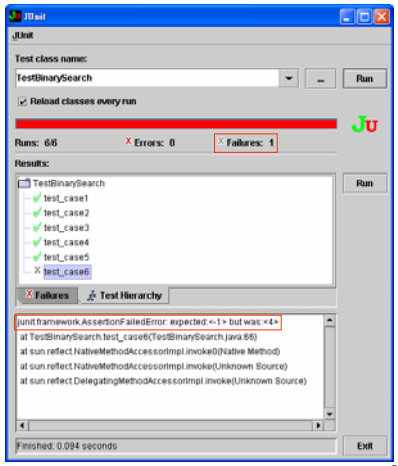
```
import junit.framework.TestCase;
public class TestBinarySearch extends
    TestCase {
    BinarySearch search;
    int sortedArray[];
    int sortedArray2[]={2,4,6};
    int sortedArray3[]={2,4,6,8,10};

    public TestBinarySearch(String name) {
        super(name);
    }
    public static void main(String[] args) {
    junit.swingui.TestRunner.run(TestBinar
    ySearch.class);
    }
    void setUp() throws Exception {
        super.setUp();
        search=new BinarySearch();
        sortedArray=new int[0];
    }
    protected void tearDown() throws
    Exception {
        super.tearDown();
    }
```

```
  public void test_case1() {

    assertEquals(search.binarySearch(sortedArray,
    2),-1);
  }
  public void test_case2() {

    assertEquals(search.binarySearch(sortedArray2
    ,8),-1);
  }
  public void test_case3() {

    assertEquals(search.binarySearch(sortedArray3
    ,6),2);
  }
  public void test_case4() {

    assertEquals(search.binarySearch(sortedArray3
    ,4),1);
  }
  public void test_case5() {

    assertEquals(search.binarySearch(sortedArray3
    ,10),4);
  }
  // test case for showing the JUnit failure
  public void test_case6() {

    assertEquals(search.binarySearch(sortedArray3
    ,7),4);
  }
}
```

85

JUnit Example for binarySearch





86

Integration Testing

- Big bang (non-incremental integration)
  - All components are combined in advance. The entire program is tested as a whole
  - When a set of errors is encountered, correction is difficult because isolation of causes is complicated by the vast expanse of entire program
  - Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop

87

Integration Testing

- Incremental integration
  - The program is constructed and tested in small increments, where errors are easier to isolate and correct
  - Interfaces are more likely to be tested completely
  - A systematic test approach may be applied
    - Top-down integration
    - Bottom-up integration
    - Sandwich testing (combination of above two approaches)

88

Top Down Integration

- Modules are integrated by moving download through the control hierarchy
  - Beginning with the main control module (main program)
  - Modules subordinate (and ultimately subordinate) to main control module are incorporated into the structure in either a depth-first or breadth-first manner

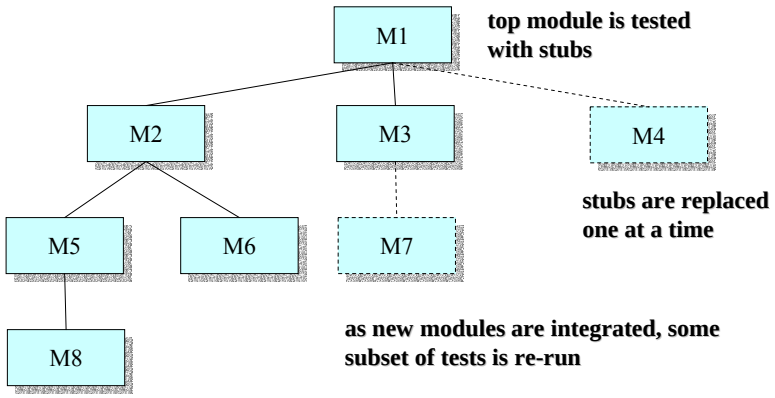
89

Top Down Integration

- The integration process:
  - The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module
  - Depending the depth-first or breadth-first, subordinate stubs are replaced one at a time with actual components
  - Test are conducted as each component is integrated
  - On completion of each set of tests, another stub is replaced with the real component
  - Regression testing may be conducted to ensure that new errors have not been introduced
  - Repeat from step 2 until the entire program structure is built

90

Top Down Integration



91

Note: Redraw from [Pressman 2004]

Top Down Integration

- Advantages
  - Is better at discovering errors in the system architecture (verifies major control or decision points early)
  - Can demonstrate a complete function of the system early (if depth-first integration is selected)
- Disadvantages
  - Logistical problems can raise
  - Need to write and test stubs (costly)
- To solve the logistical problem:
  - Delay many tests until stubs are replaced with actual modules
  - Develop stubs that perform limited functions that simulate the actual module (requires significant overhead)
  - Integrate the software from the bottom of the hierarchy upward

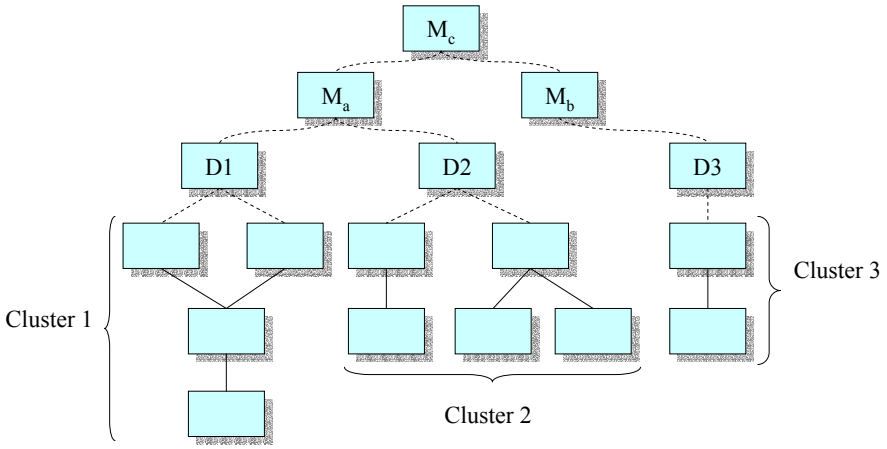
92

Bottom Up Integration

- Begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure).
- The integration process:
  - Low-level components are combined into clusters (sometimes called builds) that perform a specific software subfunction
  - A test driver is written to coordinate test case input and output
  - The cluster is tested
  - The test drivers are removed and clusters are combined moving upward in the program structure.
- As the integration moves upward, the need for separate test drivers lessens.

93

Bottom Up Integration



94

Note: Redraw from [Pressman 2004]

Bottom Up Integration

- Advantages
  - Easier test case design and no need for stubs
  - Can starts at an early stage in the development process (not require to wait until the architectural design of the system to be completed)
  - Potentially reusable modules are adequately tested
- Disadvantages
  - User interface components are tested last
  - The program as an entity does not exist until the last module is added
  - Major design faults show up late

95

Sandwich Integration Testing

- The advantages of one integration strategy tend to result in disadvantages for the other strategy.
- In practice, most integration involves a combination of both the top-down and bottom-up integration strategies, i.e., sandwich integration and testing
- The sandwich integration and testing
  - The decision modules (logic modules) are integrated and tested top-down so that major design faults can be revealed early
  - The worker modules (operational modules) are integrated and tested bottom-up so that the potentially reusable modules are adequately tested while reducing the constructions of test stubs
  - The sandwich integration and testing has the strengths of both top-down and bottom-up integration strategies while avoiding their weaknesses

96

Regression Testing

- Each time a new module is added as part of integration testing, or a bug is fixed (as the results of uncovering errors through testing) in the software, the software changes.
- These changes may cause problems with functions that previously worked.
- Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not generated unintended side effects.

97

Regression Testing

- Regression testing may be conducted
  - Manually by re-executing a subset of all test cases
  - Using automated capture/playback tools – enable software engineer to capture test cases and results for subsequent playback and comparison.
- The regression test suite contains three different classes of test cases:
  - A representative sample of tests that will exercise all software functions
  - Additional tests that focus on software functions that are likely to be affected by the change
  - Tests that focus on the software components that have been changed

98

Regression Testing

- As integration testing proceeds, the number of regression tests can grown quite large
  - The regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions.
  - It is impractical and inefficient to re-execute every test for every program function once a change has occurred

99

System Testing

- System testing
  - The software is tested as a whole. It verifies all elements mesh properly to make sure that all system functions and performance are achieved in the target environment.
- The focus areas are
  - System functions and performance
  - System reliability and recoverability (recovery test)
  - System installation (installation test)
  - System behavior in the special conditions (stress and load test)
  - System user operations (acceptance test/alpha test)
  - Hardware and software integration and collaboration
  - Integration of external software and the system
- System testers: test engineers in ITG or SQA people.

100

System Testing

- Recovery testing
  - Verify that the system can recover when forced to fail in various ways
  - Database recovery is particularly important (measure time to recover (MTTR))
  - Measure mean-time-to-repair (MTTR) if recovery requires human intervention
- Stress testing
  - Verify that the system can continue functioning when confronted with many simultaneous requests (abnormal situations)
  - Execute the system by demanding resource in abnormal quantity, frequency, or volume (subject to extreme data & event traffic)
  - Excessive interrupt, high input data rate, maximum memory, …
  - How high can we go? Do we fail-soft or collapse?
  - Sensitivity testing (a variation of stress testing)
  - Attempts to uncover data combinations within valid input classes that may cause instability or improper processing (performance degradation)

101

System Testing

- Security testing
  - Verify that system built-in access protection mechanisms work
  - To test system for invulnerability from frontal attack as well as flank or rear attack
  - Verify that the penetration cost is more than the value of the information obtained. (measure average time to break in)
- Performance testing
  - Is designed to test the run-time performance of software (real-time and embedded systems) within the context of an integrated system
  - Measure speed, resource utilization under various circumstances.
  - Is often coupled with stress testing and usually requires both hardware and software instrumentation
  - Occurs throughout all steps in the testing process

102

Alpha and Beta Testing

- Alpha and beta testing
  - When software is developed as a product to be used by many customers, a process called alpha and beta testing is used to uncover errors that only end-user seems able to find.
- Alpha Testing
  - Conducted at developer's site
  - Conducted by customer
  - Developer looks over the shoulder
  - Developer records the errors and problems
  - Conducted in a controlled environment

103

Alpha and Beta Testing

- Beta Testing
  - Conducted at one or more customer sites
  - Conducted by end-user
  - Developer is not present
  - Uncontrolled environment
  - Errors may be real or imagined
  - Customer reports the errors
  - Accordingly modifications are made
  - Final product is released
- For commercial products α - β testing is done

104

Acceptance Testing

- Acceptance tests are performed after system testing if the software is being developed for a specific client
- Acceptance tests are usually carried out by the clients or end users
- Acceptance test cases are based on requirements
  - User manual can be an additional source for test cases
  - System test cases can be reused

105

Acceptance Testing

- The software must run under real-world conditions on operational hardware/software
- The clients determine if the software meet their requirements

106

Object-Oriented Testing

- Although there are similarities between testing conventional systems and Object-Oriented systems, Object-Oriented testing has significant differences
- The characteristics of Object-Oriented systems influence both testing strategy and testing methods.
  - The class become the natural unit for test case design
  - Implications of Object-Oriented concepts such as inheritance, encapsulation, and polymorphism pose testing challenges
  - Testing the state-dependent behavior of Object-Oriented systems become important since no clear control flow in Object-Oriented programs
  - Integration strategies change significantly since no obvious 'top' module to the system for top-down integration

107

Unit Testing in the OO Context

- Smallest testable unit is the encapsulated class
- A single operation needs to be tested as part of a class hierarchy because its context of use may differ subtly
  - Inheritance allows an operation exists in many different classes where the operation is applied within the context of different attributes and is used in various subtle ways
- Class testing is the equivalent of unit testing in conventional software
- Approach:
  - Methods within the class are tested
  - The state behavior of the class is examined
- Unlike conventional unit testing which focuses on the input-process-output view of software, algorithm detail of a module, and the data that flow across the module interface, class testing focuses on designing sequences of methods to exercise the states of a class
- But white-box methods can still be applied

108

Integration Testing in OO Context

- The OO integration strategy changes dramatically
  - OO does not have a hierarchical control structure so conventional top-down and bottom-up integration tests have little meaning.
  - Integrating operations one at a time into a class is often impossible because the "direct and indirect interactions of the components that make up the class"
- OO integration applied three different incremental strategies
- Thread-based testing: integrates classes required to respond to one input or event.
  - each thread is integrated and tested individually.
- Use-based testing: integrates the set of classes required to respond to use dependency
  - independent classes: use very few (if any) of server classes
  - dependent classes: the classes use independent classes
  - testing the independent classes first and then dependent classes
- Cluster testing: integrates classes required to demonstrate one collaboration
  - A cluster of collaborating classes is tested to uncover the collaborating errors

109

Validation Testing in an OO Context

- At the validation or system level, the details of class connections disappear
- Like conventional validation, the OO validation focuses on user-visible actions and user-recognizable outputs
- Approach:
  - Use-case scenarios from the analysis model has a high likelihood to uncover errors of user interaction requirements
  - Conventional black-box testing methods can be used to create a deficiency list
  - Test cases may be driven from the object-behavior model and from event flow diagram created as part of OOA
  - Acceptance tests through alpha (at developer's site) and beta (at customer's site) testing with actual customers

110

OOT—Test Case Design

- Berard in his book, *Essays on Object-Oriented Software Engineering*, proposes the following approach for OO test case design:
  - Each test case should be uniquely identified and should be explicitly associated with the class to be tested
  - The purpose of the test should be stated
  - A list of testing steps should be developed for each test and should contain:
    - A list of specified states for the object that is to be tested
    - A list of messages and operations that will be exercised as a consequence of the test
    - A list of exceptions that may occur as the object is tested
    - 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)
    - Supplementary information that will aid in understanding or implementing the test

111

OOT—Test Case Design

- 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.

112

OOT Methods: Random Testing

- Random testing
  - Identify operations applicable to a class
  - Define constraints on their use
  - Identify a minimum 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

113

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

114

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

115

OOT Methods: State-Based Testing

- State-based testing
  - Basic idea is to derive tests from the object-behavioral analysis model to check the state behaviors of a system.
  - A state machine (state transition diagram or statechart) is used to define test cases and test criteria to uncover the incorrect state behavior of a program.
- Coverage criteria:
  - State node coverage
  - Transition path coverage
- Applications:
  - Very useful to check protocol-based program communications.
  - Good to check problems in state-driven or event-driven application programs.
  - Can be used at both specification and code levels

116

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

117

Exercises

- Consider the following program and (1) draw the control flow diagram (CFG) of the program; (2) compute the cyclomatic complexity of the program; (3) derive the independent paths of the program; (4) design test cases to exercise the derived paths; and (5) based on the designed test cases, write JUnit code to test the program.

```
void SelectionSort(int n,int List[])
{
  int j,k, minPosition;
  int temp;

  temp=0;
  for(k=0;k<=n-1;k++)
  {
      minPosition=k;
      for(j=k+1;j<=n;j++)
         if(List[j]<List[minPosition])
             minPosition= j;
         temp=List[k];
         List[k]=List[minPosition];
         List[minPosition]=temp;
  }
}
```

118

Exercises

- Design test cases using black-box testing techniques, such as equivalence partitioning and boundary value analysis, for a program that takes a 3 digit string as input and converts the string into an integer.
- What are the differences between white-box and black-box testing techniques. Why they complement each other?
- What is integration testing? Describe and discuss three common integration processes
- What is regression testing? Why regression testing is required? How to select test cases for regression testing?
- What is system testing? What are the differences between alpha and beta testing?
- What are the differences between testing object-oriented programs and traditional programs?
- Describe three incremental integration strategies for object-oriented program.

119

Reference

- [Beizer 1990] B. Beizer, *Software Testing Techniques*, 2nd Edition, Van Nostrand-Reinhold, 1990.
- [Boehm 1981] B. Boehm, Software Engineering Economics, Prentice-Hall, 1981.
- [Davis 1995] A. Davis, *201 Principles of Software Development*, McGraw-Hill, 1995.
- [Gao 2003] Jerry Gao, Jacob Tsao, and Ye Wu, *Testing and Quality Assurance for Component-Based Software*, Artech House, 2003.
- Kaner 1999] C. Kaner, J. Falk, and H. Q. Nguyen, Testing Computer Software, 2nd Edition, Wiley, 1999.
- [Myers 1979] G. Myers, *The Art of Software Testing*, Wiley, 1979.
- [Pressman 2004] Roger S. Pressman, *Software Engineering – A practitioner's Approach*, 6th Edition, McGraw Hill, 2004.
- [Sommerville 2004] Ian Sommerville, *Software Engineering*, 7th Edition, Addison Wesley, 2004.

120

Reference

- [Hetzel 1984] W. Hetzel, *The Complete Guide to Software Testing*, QED Information Sciences, 1984.
- [Tai 1989] K. C. Tai, "What to Do Beyond Branch Testing," *ACM Software Engineering Notes*, vol. 14, no. 2, April 1989, pp. 58-61.
- [Whittaker 2000] James A. Whittaker, "What Is Software Testing? And Why Is It So Hard," *IEEE Software*, 2000.
- [Weyuker 1991] E. Weyuker and B. Jeng, "Analyzing Partition Testing Strategies," *IEEE Transaction on Software Engineering*, Vol. 17, No. 7, 1991, pp. 703-711.

121