Universitas **Esa Unggul**

Smart, Creative and Entrepreneurial

Software Testing Techniques
Pertemuan-1
Dosen :Kundang K Juman
Prodi Teknik Informatika , Fakultas Imu Komputer

Chapter 14
Software Testing Techniques

- Testing fundamentals

- White-box testing

- Black-box testing

- Object-oriented testing methods

(Source: Pressman, R. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2005)

Characteristics of Testable Software

- Operable
  - The better it works (i.e., better quality), the easier it is to test
- Observable
  - Incorrect output is easily identified; internal errors are automatically detected
- Controllable
  - The states and variables of the software can be controlled directly by the tester
- Decomposable
  - The software is built from independent modules that can be tested independently

(more on next slide)

Characteristics of Testable Software (continued)

- Simple
  - The program should exhibit functional, structural, and code simplicity
- Stable
  - Changes to the software during testing are infrequent and do not invalidate existing tests
- Understandable
  - The architectural design is well understood; documentation is available and organized

Test Characteristics

- A good test has a high probability of finding an error
  - The tester must understand the software and how it might fail
- A good test is not redundant
  - Testing time is limited; one test should not serve the same purpose as another test
- A good test should be "best of breed"
  - Tests that have the highest likelihood of uncovering a whole class of errors should be used
- A good test should be neither too simple nor too complex
  - Each test should be executed separately; combining a series of tests could cause side effects and mask certain errors

Two Unit Testing Techniques

- Black-box testing
  - Knowing the specified function that a product has been designed to perform, test to see if that function is fully operational and error free
  - Includes tests that are conducted at the software interface
  - Not concerned with internal logical structure of the software
- White-box testing
  - Knowing the internal workings of a product, test that all internal operations are performed according to specifications and all internal components have been exercised
  - Involves tests that concentrate on close examination of procedural detail
  - Logical paths through the software are tested
  - Test cases exercise specific sets of conditions and loops

White-box Testing

White-box Testing

- Uses the control structure part of component-level design to derive the test cases

- These test cases
  - Guarantee that <u>all independent paths</u> 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 ensure their validity

"Bugs lurk in corners and congregate at boundaries"

Basis Path Testing

- White-box testing technique proposed by Tom McCabe

- Enables the test case designer to derive a logical complexity measure of a procedural design

- Uses this measure as a guide for defining a basis set of execution paths

- Test cases derived to exercise the basis set are guaranteed to execute <u>every statement</u> in the program <u>at least one time</u> during testing

Flow Graph Notation

- A circle in a graph represents a <u>node</u>, which stands for a <u>sequence</u> of one or more procedural statements
- A node containing a simple conditional expression is referred to as a <u>predicate node</u>
  - Each <u>compound condition</u> in a conditional expression containing one or more Boolean operators (e.g., and, or) is represented by a separate predicate node
  - A predicate node has <u>two</u> edges leading out from it (True and False)
- An <u>edge</u>, or a link, is a an arrow representing flow of control in a specific direction
  - An edge must start and terminate at a node
  - An edge does not intersect or cross over another edge
- Areas bounded by a set of edges and nodes are called <u>regions</u>
- When counting regions, include the area outside the graph as a region, too

Flow Graph Example

FLOW CHART

FLOW GRAPH

R4

R3

R2

R1

11
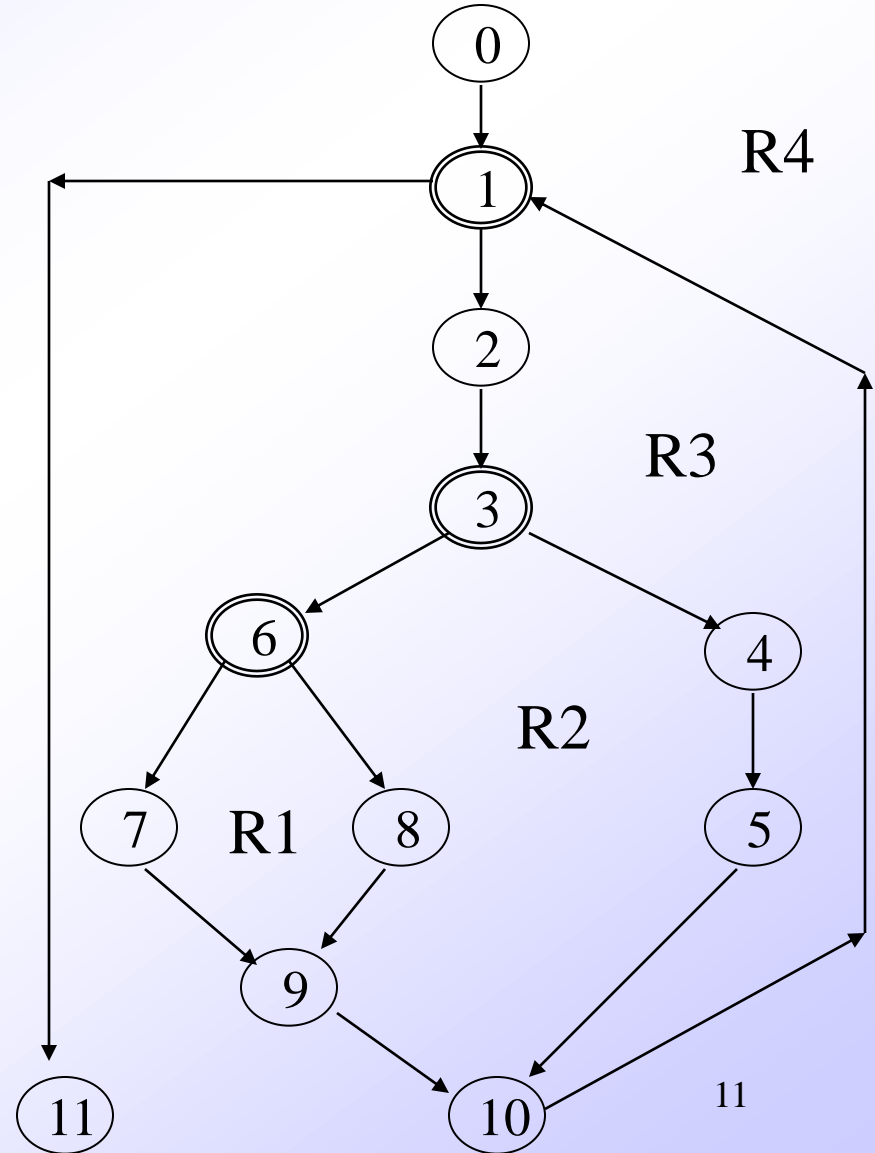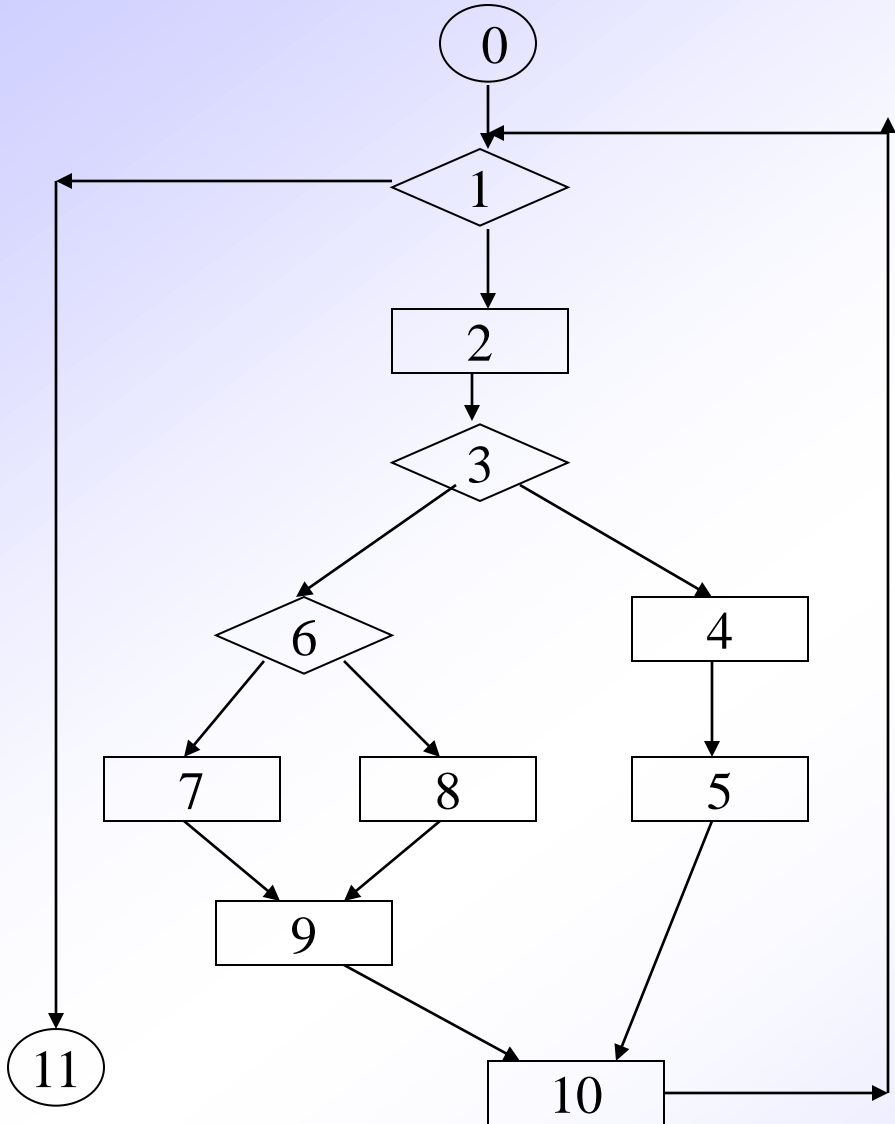
Independent Program Paths

- Defined as a path through the program from the start node until the end node that introduces at least one new set of processing statements or a new condition (i.e., new nodes)
- Must move along <u>at least one</u> edge that has not been traversed before by a previous path
- Basis set for flow graph on previous slide
  - Path 1: 0-1-11
  - Path 2: 0-1-2-3-4-5-10-1-11
  - Path 3: 0-1-2-3-6-8-9-10-1-11
  - Path 4: 0-1-2-3-6-7-9-10-1-11
- The <u>number of paths</u> in the basis set is determined by the <u>cyclomatic complexity</u>
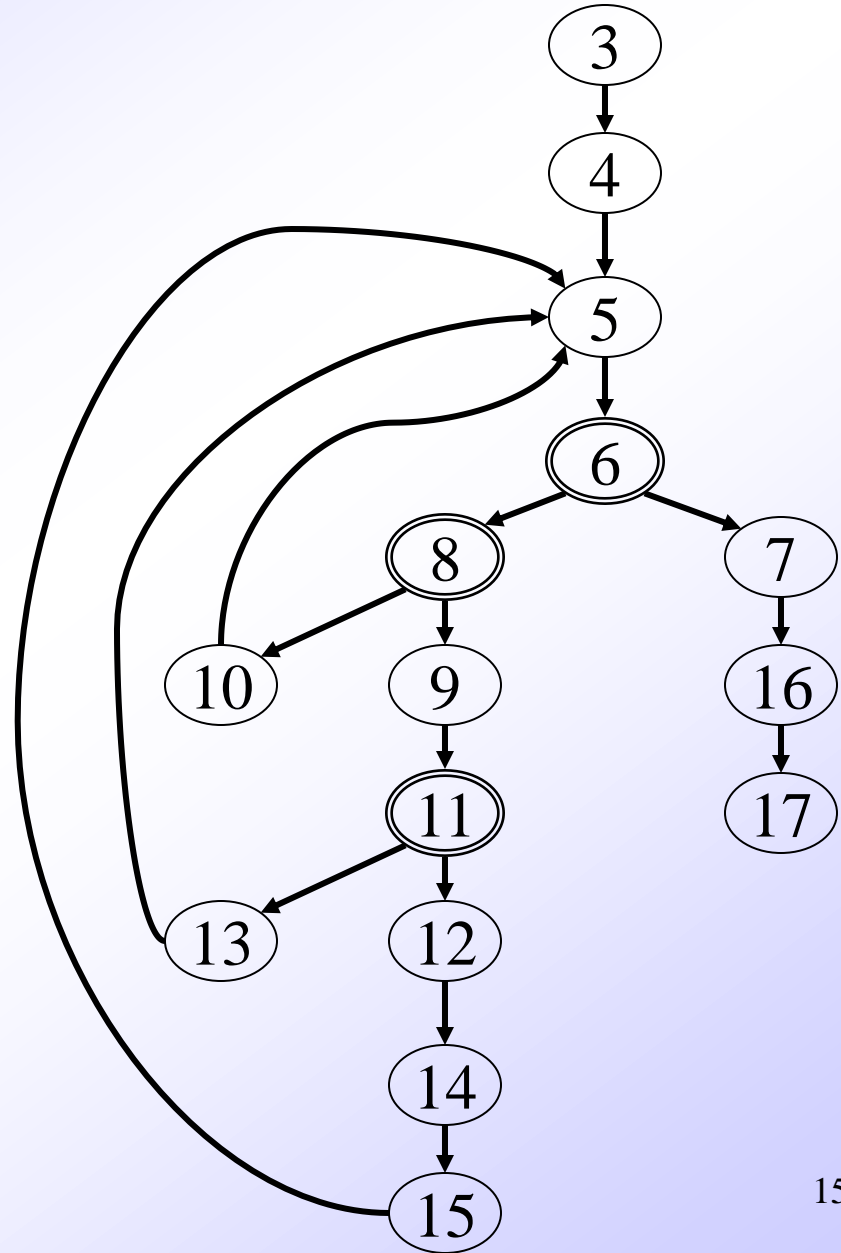
Cyclomatic Complexity

- Provides a quantitative measure of the <u>logical complexity</u> of a program
- Defines the <u>number of independent paths</u> in the basis set
- Provides an <u>upper bound</u> for the number of tests that must be conducted to ensure <u>all statements</u> have been executed <u>at least once</u>
- Can be computed <u>three</u> ways
  - The number of regions
  - $V(G) = E - N + 2$, where E is the number of edges and N is the number of nodes in graph G
  - $V(G) = P + 1$, where P is the number of predicate nodes in the flow graph G
- Results in the following equations for the example flow graph
  - Number of regions = 4
  - $V(G) = 14$ edges $- 12$ nodes $+ 2 = 4$
  - $V(G) = 3$ predicate nodes $+ 1 = 4$

Deriving the Basis Set and Test Cases

1) Using the design or code as a foundation, draw a corresponding flow graph

2) Determine the cyclomatic complexity of the resultant flow graph

3) Determine a basis set of linearly independent paths

4) Prepare test cases that will force execution of each path in the basis set

A Second Flow Graph Example

```
1    int functionY(void)
2    {
3        int x = 0;
4        int y = 19;

5    A: x++;
6        if (x > 999)
7            goto D;
8        if (x % 11 == 0)
9            goto B;
10       else goto A;

11   B: if (x % y == 0)
12           goto C;
13       else goto A;

14   C: printf("%d\n", x);
15       goto A;

16   D: printf("End of list\n");
17       return 0;
18   }
```


15

A Sample Function to Diagram and Analyze

```
1    int functionZ(int y)
2    {
3    int x = 0;

4    while (x <= (y * y))
5       {
6       if ((x % 11 == 0) &&
7           (x % y == 0))
8          {
9          printf("%d", x);
10         x++;
11         } // End if
12      else if ((x % 7 == 0) ||
13               (x % y == 1))
14         {
15         printf("%d", y);
16         x = x + 2;
17         } // End else
18      printf("\n");
19      } // End while

20   printf("End of list\n");
21   return 0;
22   } // End functionZ
```

16

A Sample Function to Diagram and Analyze

```
1    int functionZ(int y)
2    {
3    int x = 0;

4    while (x <= (y * y))
5        {
6        if ((x % 11 == 0) &&
7            (x % y == 0))
8            {
9            printf("%d", x);
10           x++;
11           } // End if
12       else if ((x % 7 == 0) ||
13               (x % y == 1))
14           {
15           printf("%d", y);
16           x = x + 2;
17           } // End else
18       printf("\n");
19       } // End while

20   printf("End of list\n");
21   return 0;
22   } // End functionZ
```
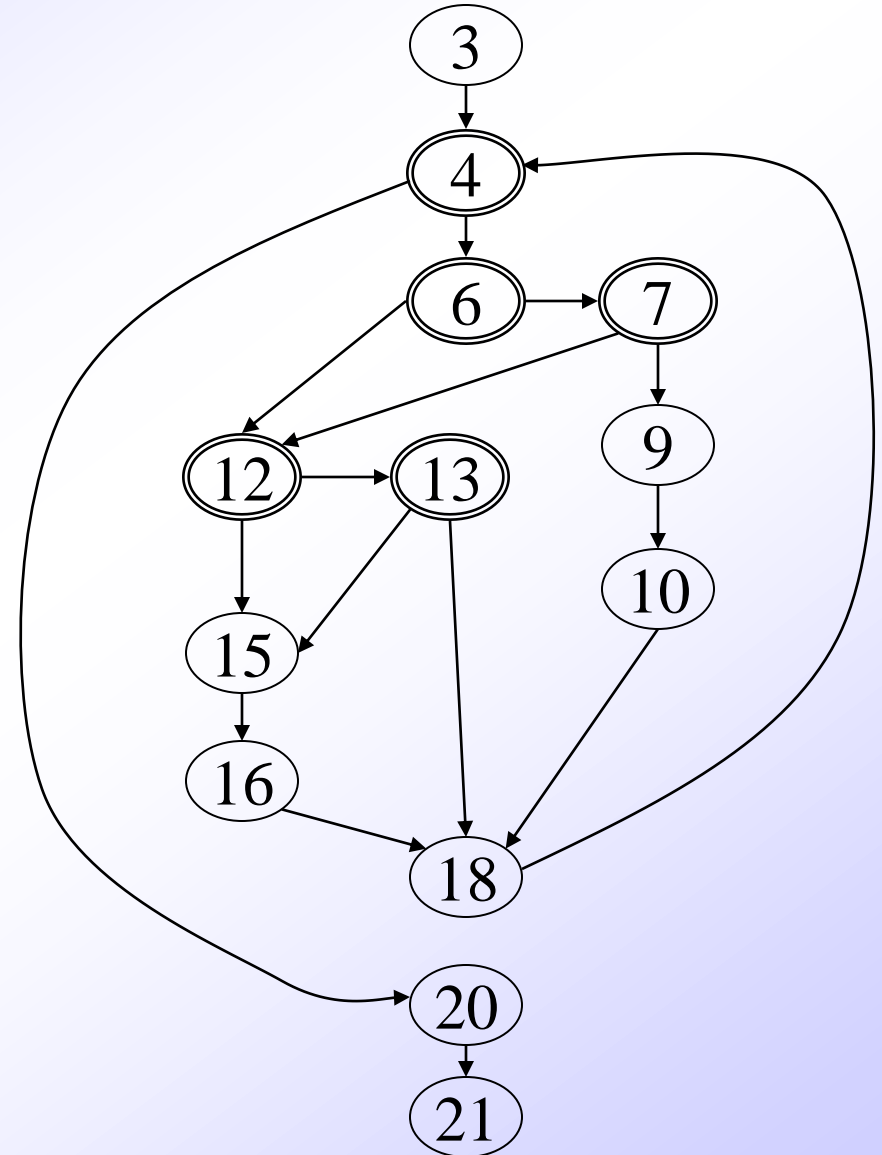
Loop Testing - General

- A white-box testing technique that focuses exclusively on the validity of loop constructs
- Four different classes of loops exist
  - Simple loops
  - Nested loops
  - Concatenated loops
  - Unstructured loops
- Testing occurs by varying the loop boundary values
  - Examples:

    ```
    for (i = 0; i < MAX_INDEX; i++)

    while (currentTemp >= MINIMUM_TEMPERATURE)
    ```

Testing of 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, where $m < n$

5) $n - 1$, n, $n + 1$ passes through the loop

'n' is the maximum number of allowable passes through the loop

Testing of Nested Loops

1) Start at the <u>innermost</u> loop; set all other loops to <u>minimum</u> values

2) Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter values; add other tests for out-of-range or excluded values

3) <u>Work outward</u>, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values

4) Continue until all loops have been tested

Testing of Concatenated Loops

- For independent loops, use the same approach as for simple loops
- Otherwise, use the approach applied for nested loops

Testing of Unstructured Loops

- <u>Redesign</u> the code to reflect the use of structured programming practices
- Depending on the resultant design, apply testing for simple loops, nested loops, or concatenated loops

Black-box Testing

Black-box Testing

- <u>Complements</u> white-box testing by uncovering different classes of errors
- Focuses on the functional requirements and the information domain of the software
- Used during the <u>later stages</u> of testing after white box testing has been performed
- The tester identifies a set of input conditions that will fully exercise all functional requirements for a program
- The test cases satisfy the following:
  - Reduce, by a count greater than one, the number of additional test cases that must be designed to achieve reasonable testing
  - Tell us something about the presence or absence of <u>classes of errors</u>, rather than an error associated only with the specific task at hand

Black-box Testing Categories

- Incorrect or missing functions

- Interface errors

- Errors in data structures or external data base access

- Behavior or performance errors

- Initialization and termination errors

Questions answered by Black-box Testing

- How is functional validity tested?
- How are 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 boundary values of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

Equivalence Partitioning

- A black-box testing method that <u>divides the input domain</u> of a program <u>into classes</u> of data from which test cases are derived

- An ideal test case <u>single-handedly</u> uncovers a <u>complete class</u> of errors, thereby reducing the total number of test cases that must be developed

- Test case design is based on an evaluation of <u>equivalence classes</u> for an input condition

- An equivalence class represents a <u>set of valid or invalid states</u> for input conditions

- From each equivalence class, test cases are selected so that the <u>largest number</u> of attributes of an equivalence class are exercise at once

Guidelines for Defining Equivalence Classes

- If an input condition specifies <u>a range</u>, one valid and two invalid equivalence classes are defined

  - Input range: 1 – 10   Eq classes: {1..10}, {x < 1}, {x > 10}

- If an input condition requires <u>a specific value</u>, one valid and two invalid equivalence classes are defined

  - Input value: 250   Eq classes: {250}, {x < 250}, {x > 250}

- If an input condition specifies <u>a member of a set</u>, one valid and one invalid equivalence class are defined

  - Input set: {-2.5, 7.3, 8.4}   Eq classes: {-2.5, 7.3, 8.4}, {any other x}

- If an input condition is <u>a Boolean value</u>, one valid and one invalid class are define

  - Input: {true condition}   Eq classes: {true condition}, {false condition}

Boundary Value Analysis

- A greater number of errors occur at the <u>boundaries</u> of the input domain rather than in the "center"

- Boundary value analysis is a test case design method that <u>complements</u> equivalence partitioning
  - It selects test cases at the <u>edges</u> of a class
  - It derives test cases from both the input domain and output domain

Guidelines for Boundary Value Analysis

- 1. If an input condition specifies a <u>range</u> bounded by values *a* and *b*, test cases should be designed with values *a* and *b* as well as values just above and just below *a* and *b*

- 2. If an input condition specifies a <u>number of values</u>, test case should be developed that exercise the minimum and maximum numbers. Values just above and just below the minimum and maximum are also tested

- Apply guidelines 1 and 2 to output conditions; produce output that reflects the minimum and the maximum values expected; also test the values just below and just above

- If internal program data structures have prescribed boundaries (e.g., an array), design a test case to exercise the data structure at its minimum and maximum boundaries

Object-Oriented Testing Methods

Introduction

- It is necessary to test an object-oriented system at a variety of different levels
- The goal is to uncover errors that may occur as classes collaborate with one another and subsystems communicate across architectural layers
  - Testing begins "in the small" on methods within a class and on collaboration between classes
  - As class integration occurs, <u>use-based testing</u> and <u>fault-based testing</u> are applied
  - Finally, use cases are used to uncover errors during the software validation phase
- Conventional test case design is driven by an <u>input-process-output</u> view of software
- Object-oriented testing focuses on designing appropriate sequences of methods to exercise the <u>states</u> of a class

Testing Implications for Object-Oriented Software

- Because attributes and methods are encapsulated in a class, testing methods from outside of a class is generally unproductive

- Testing requires reporting on the state of an object, yet encapsulation can make this information somewhat difficult to obtain

- Built-in methods should be provided to report the values of class attributes in order to get a snapshot of the state of an object

- Inheritance requires retesting of each new context of usage for a class
  - If a subclass is used in an entirely different context than the super class, the super class test cases will have little applicability and a new set of tests must be designed

Applicability of Conventional Testing Methods

- White-box testing can be applied to the operations defined in a class
    - Basis path testing and loop testing can help ensure that every statement in an method has been tested
- Black-box testing methods are also appropriate
    - Use cases can provide useful input in the design of black-box tests

Fault-based Testing

- The objective in fault-based testing is to design tests that have a high likelihood of uncovering <u>plausible faults</u>

- Fault-based testing begins with the analysis model
  - 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

- If the analysis and design models can provide insight into what is likely to go wrong, then fault-based testing can find a significant number of errors

Fault-based Testing (continued)

- Integration testing looks for plausible faults in method calls or message connections (i.e., client/server exchange)
- <u>Three types</u> of faults are encountered in this context
    - Unexpected result
    - Wrong method or message used
    - Incorrect invocation
- The behavior of a method must be examined to determine the occurrence of plausible faults as methods are invoked
- Testing should exercise the attributes of an object to determine whether proper values occur for distinct types of object behavior
- The focus of integration testing is to determine whether <u>errors</u> exist in the <u>calling code</u>, not the called code

Fault-based Testing
vs. Scenario-based Testing

- Fault-based testing misses two main types of errors
  - Incorrect specification: subsystem doesn't do what the user wants
  - Interactions among subsystems: behavior of one subsystem creates circumstances that cause another subsystem to fail
- A solution to this problem is <u>scenario-based testing</u>
  - It 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 as tests
  - Scenario-based testing tends to exercise multiple subsystems in a single test

Random Order Testing (at the Class Level)

- Certain methods in a class may constitute a minimum <u>behavioral life history</u> of an object (e.g., open, seek, read, close); consequently, they may have implicit order dependencies or expectations designed into them

- Using the methods for a class, a variety of method sequences are generated randomly and then executed

- The goal is to detect these order dependencies or expectations and make appropriate adjustments to the design of the methods

Partition Testing (at the Class Level)

- Similar to equivalence partitioning for conventional software
- Methods are grouped based on one of three partitioning approaches
- <u>State-based</u> partitioning categorizes class methods based on their ability to change the state of the class
  - Tests are designed in a way that exercise methods that change state and those that do not change state
- <u>Attribute-based</u> partitioning categorizes class methods based on the attributes that they use
  - Methods are partitioned into those that read an attribute, modify an attribute, or do not reference the attribute at all
- <u>Category-based</u> partitioning categorizes class methods based on the generic function that each performs
  - Example categories are initialization methods, computational methods, and termination methods

Multiple Class Testing

- <u>Class collaboration testing</u> can be accomplished by applying random testing, partition testing, scenario-based testing and behavioral testing

- The following sequence of steps can be used to generate multiple class random test cases

  1) For each client class, use the list of class methods to generate a series of random test sequences; use these methods to send messages to server classes

  2) For each message that is generated, determine the collaborator class and the corresponding method in the server object

  3) For each method in the server object (invoked by messages from the client object), determine the messages that it transmits

  4) For each of these messages, determine the next level of methods that are invoked and incorporate these into the test sequence

Tests Derived from Behavior Models

- The <u>state diagram</u> for a class can be used to derive a sequence of tests that will exercise the <u>dynamic behavior</u> of the class and the classes that collaborate with it

- The test cases should be designed to achieve coverage of <u>all</u> states
  - Method sequences should cause the object to transition through all allowable states

- More test cases should be derived to ensure that all behaviors for the class have been exercised based on the behavior life history of the object

- The state diagram can be traversed in a "breadth-first" approach by exercising only a single transition at a time
  - When a new transition is to be tested, only previously tested transitions are used