Chapter 2

Testing Tactics

Note :

1. **These slides are modified version of slides of Roger Pressman, 6th edition.**
2. **Prepared by Narayan D. G., Faculty,BVBCET Hubli.**

Chapter Plan:

- Introduction
- Black-Box Testing, White-Box testing
- Control structure testing
- loop testing
- OO software testing case design

Testability

- How easily the software can be tested.

Characteristics of 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
- Simplicity—reduce complex architecture and logic to simplify tests
- Stability—few changes are requested during testing
- Understandability—of the design

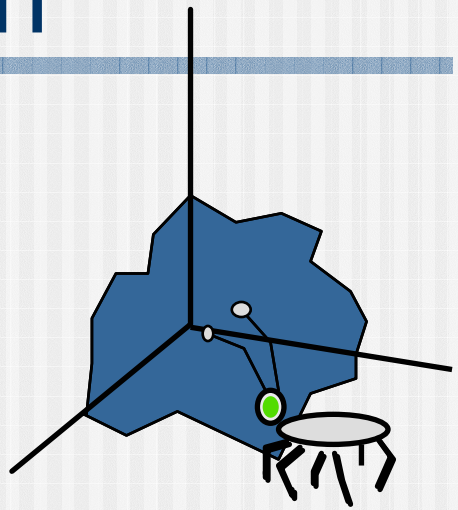
What is a “Good” Test?

- A good test has a high probability of finding an error. Tester must understand the system and develop a mental picture of how it might fail.
- A good test is not redundant; every test should have a different purpose.
- A good test should be “best of breed”; the test that has the highest likelihood of uncovering a whole class of errors should be used.
- A good test should be neither too simple nor too complex.

Test Case Design

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

Boris Beizer

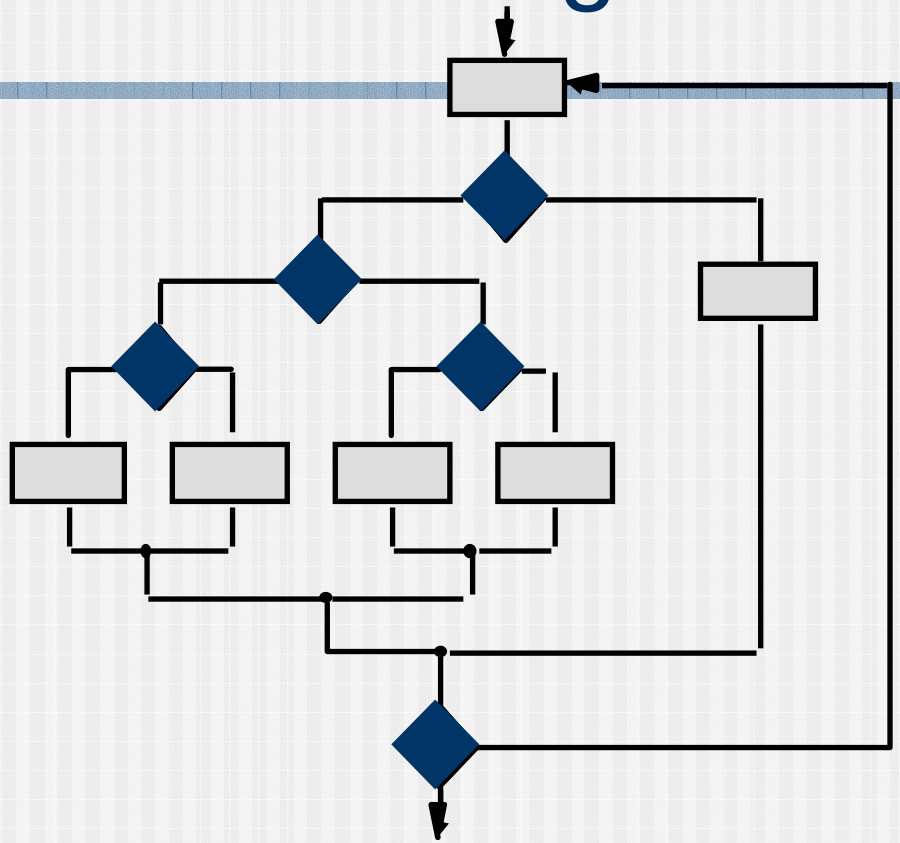


OBJECTIVE **to uncover errors**

CRITERIA **in a complete manner**

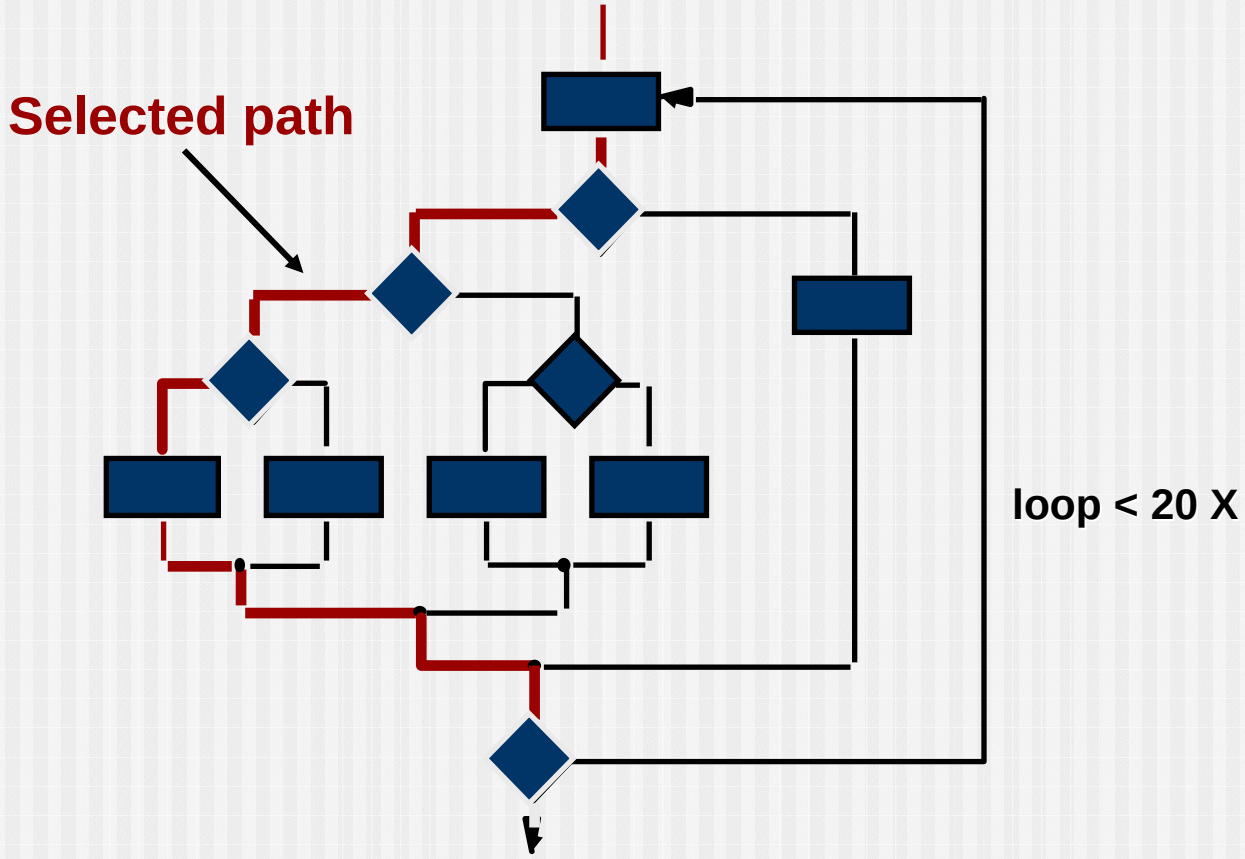
CONSTRAINT **with a minimum of effort and time**

Exhaustive Testing

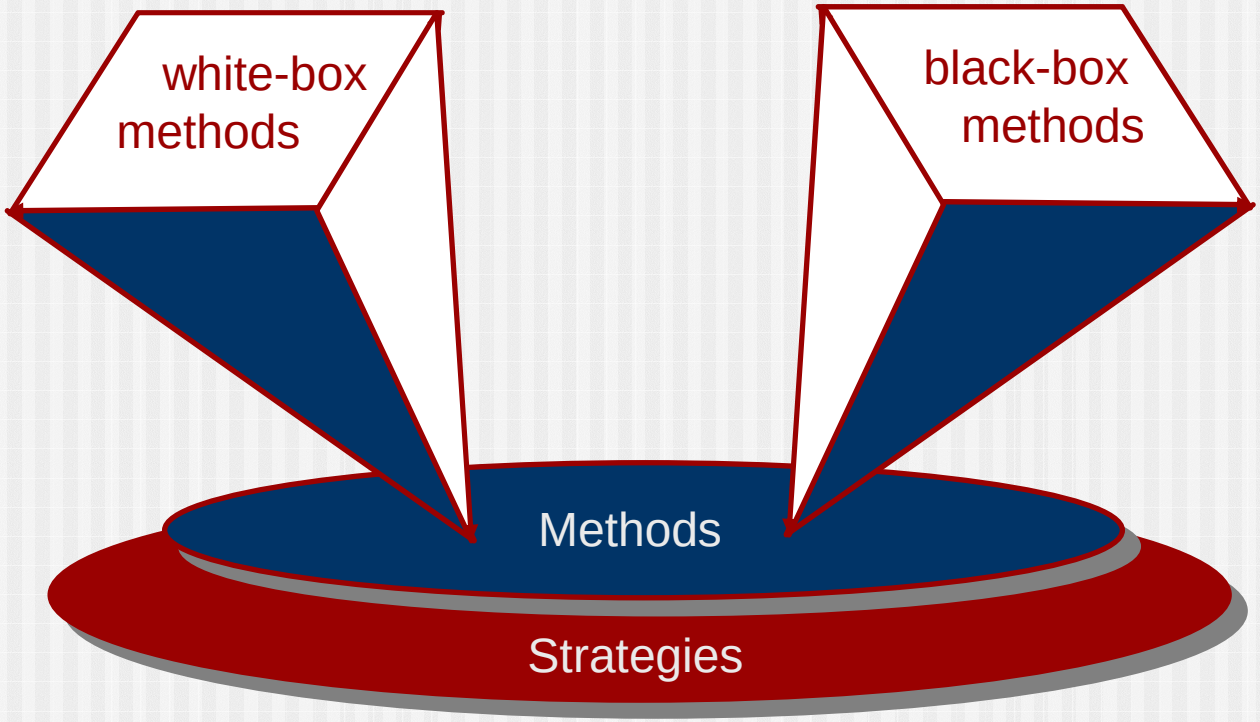


There are 10^{14} possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!

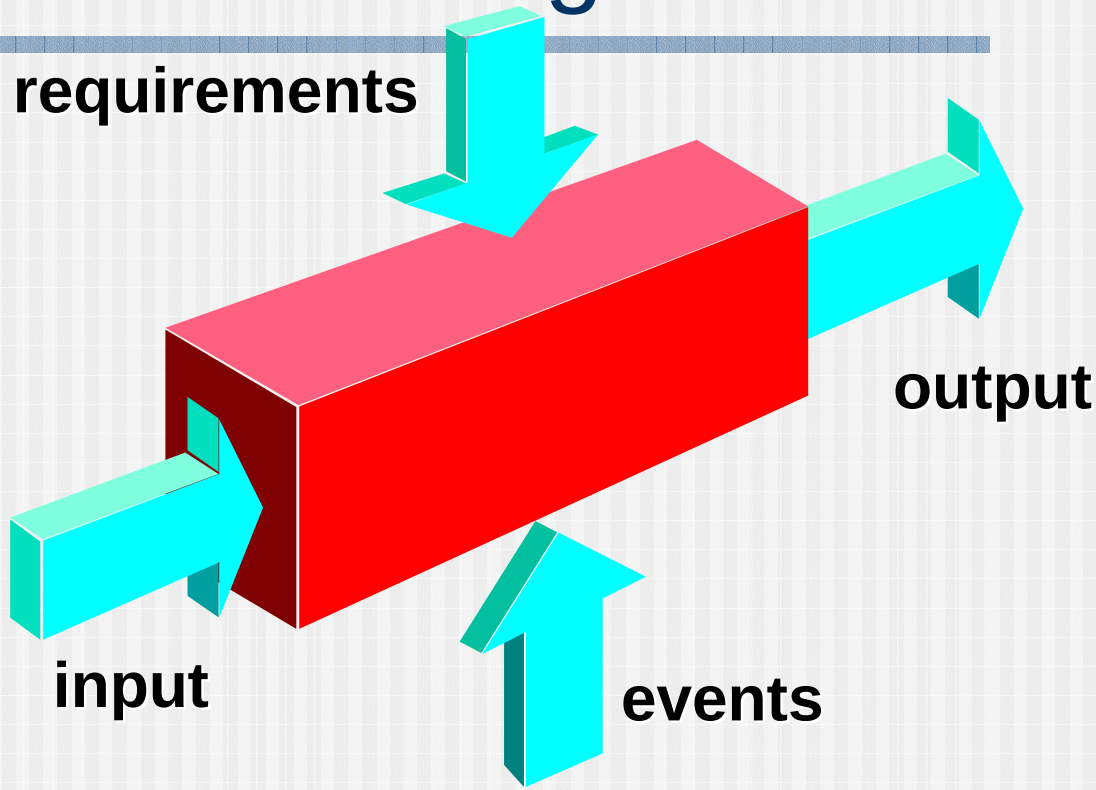
Selective Testing



Software Testing



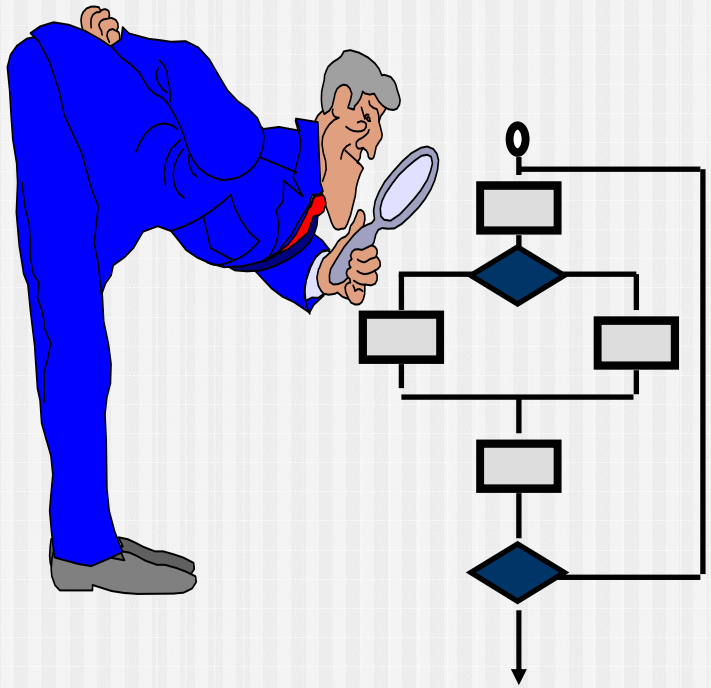
Black-Box Testing



Black box testing

- **Also called as functional testing**
- **No knowledge of internal design or code required.**
- **Tests are based on requirements and functionality**

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**

White box testing

- **Also called as glass-box testing and structural testing**
- **Knowledge of the internal program design and code required.**
- **Tests are based on coverage of code, statements, branches, paths, conditions**

The S/W eng. 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 ensure their validity.**

Basis Path Testing

- It’s a white-box testing technique
- Three steps to derive the test cases using Basis path Testing are as follows.

1. Derive a logical complexity measure of procedural design
   – Convert program OR flowchart to the control graph
   – Identify the number of regions (Cyclomatic Number) which is equivalent to the McCabe’s number

2. Define a basis set of execution paths
   – Determine independent paths

3. Derive test case to exercise (cover) the basis set

Cyclomatic complexity

- Gives a quantitative measure of the logical complexity of the module
- Defines the number of independent paths
- Provides an upper bound to the number of tests that must be conducted to ensure that all the statements are executed at least once.

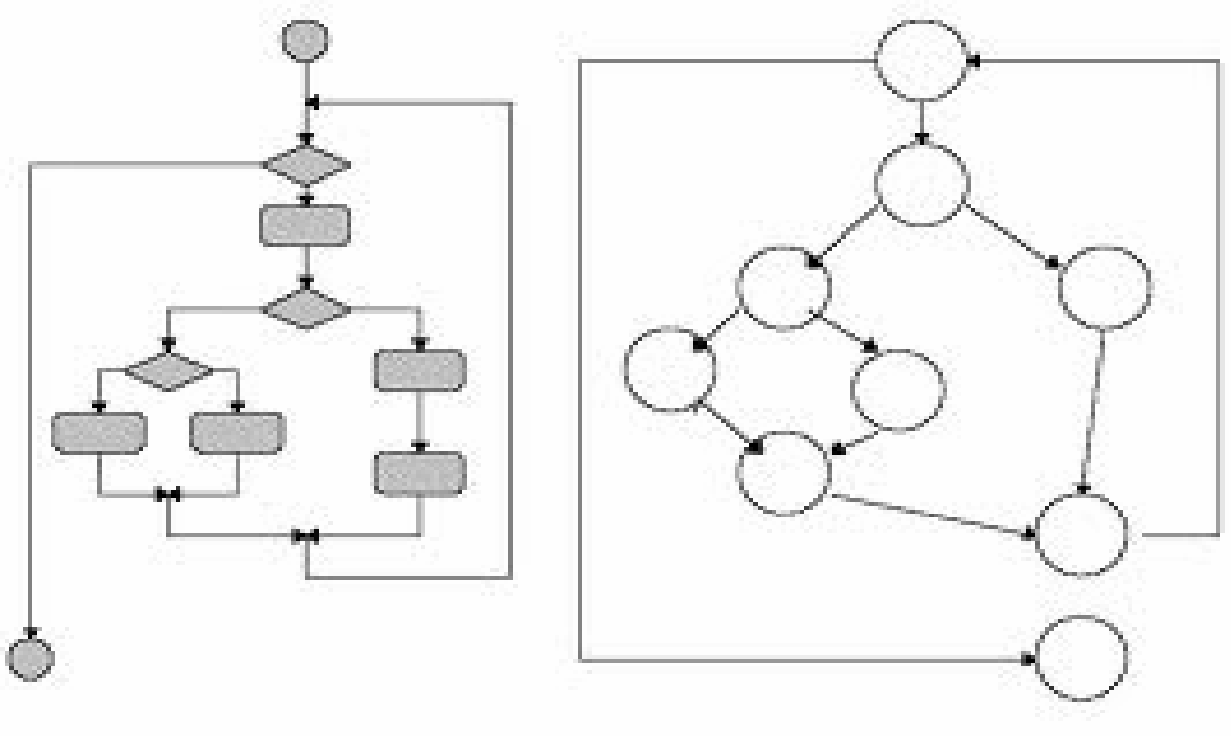
Complexity of a flow graph ‘G’, V(G), is computed in one of three ways:

– V(G) = No. of regions of G

– V(G) = E - N +2 (E: No. of edges & N: No. of nodes)

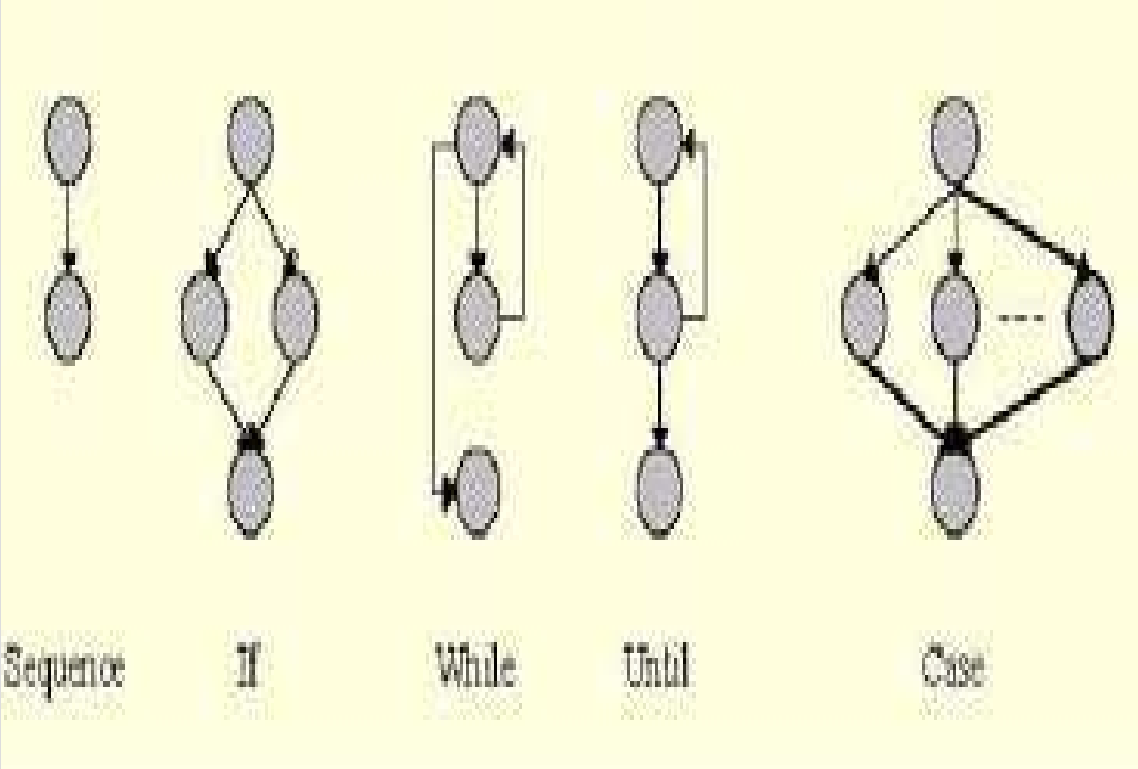
– V(G) = P + 1 (P: No. of predicate nodes in G or No. of conditions in the code)

Example: C(G)



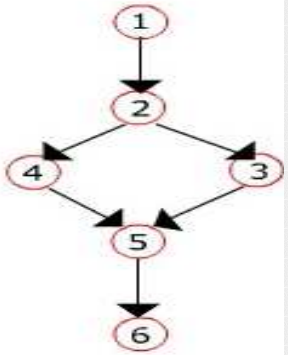
What is V(G)

Flow Graph Notations



Example 1

```
begin
1 read a;
  read b;
  2
  if (a>b) then
  3 a++;
  else
  4 b++;
5 print a, b;
6 end
```



Cyclomatic Complexity = No. of Regions = 2
Cyclomatic Complexity = E - N +2 = 6 – 6 + 2 = 2
Cyclomatic Complexity = P + 1 = 1 + 1 = 2

The independents paths in the graph are:
i) 1-2-3-5-6
ii) 1-2-4-5-6

Example 2:

Find C(G), independent paths for the following program.

```
public void howComplex(int i) {

    while (i<10) {
        System.out.printf("i is %d", i);
        if (i%2 == 0) {
            System.out.println("even");
        } else {
            System.out.println("odd");
        }
    }  // End while
  return 0;
}
```

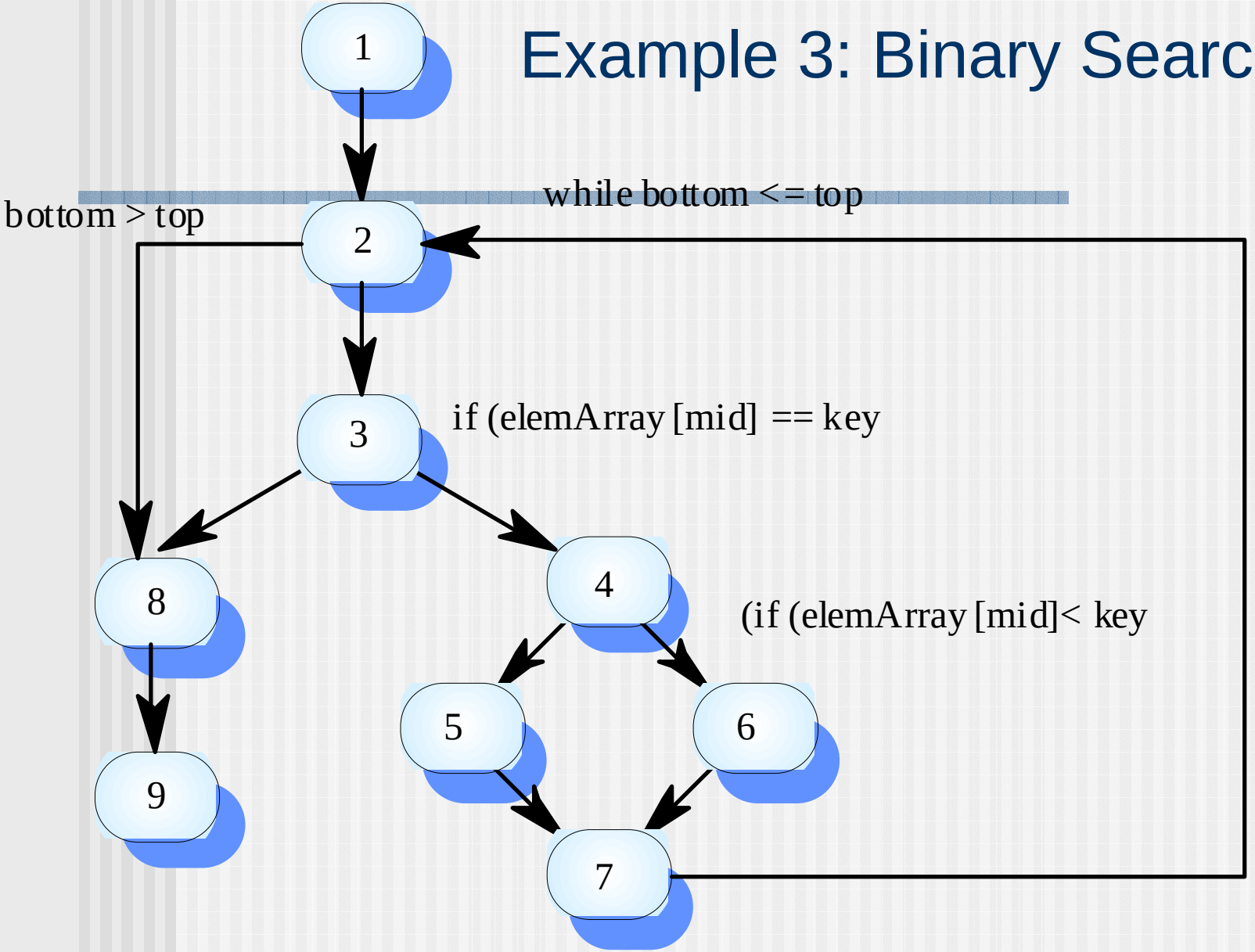
Example:

```
while records remain
    read record;
    if record field 1 = 0
            then process record;
           store in buffer;
           increment counter;
    else if record field 2 = 0
             then reset record;
    else process record;
             store in file;
    endif;
    endif;
enddo;
end;
```

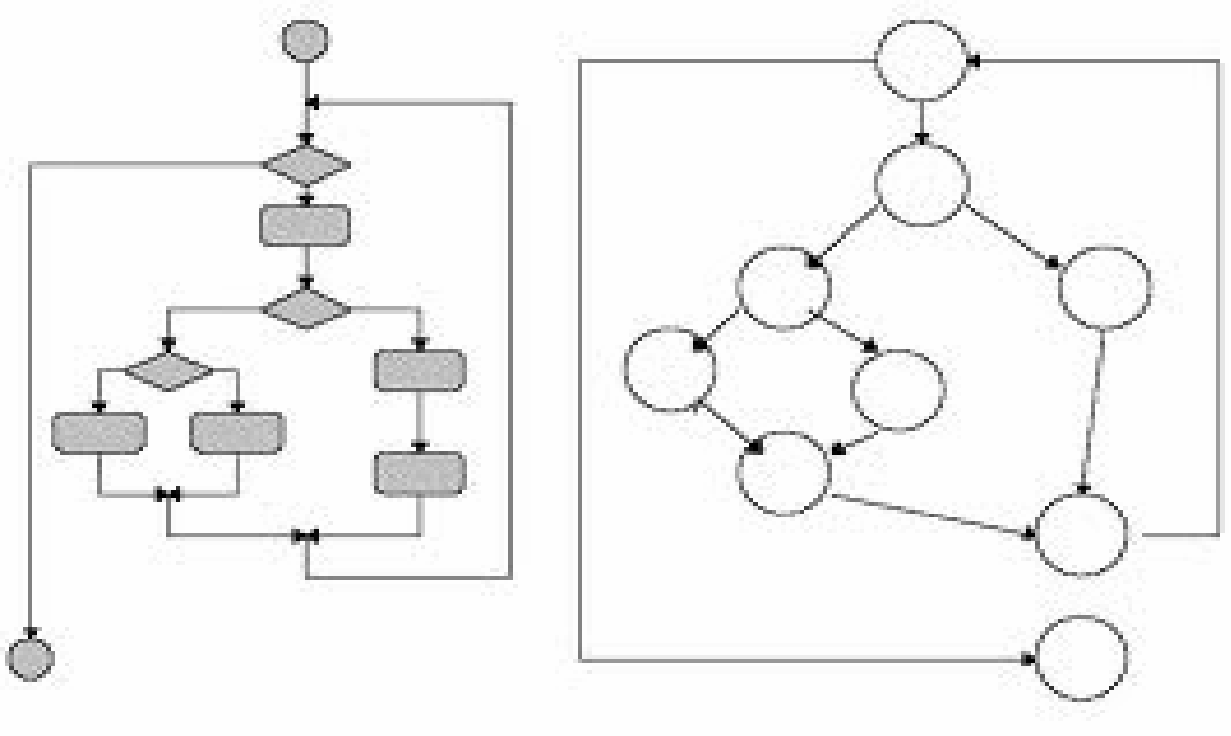
Example 3: Binary search

```
int search ( int key, int [] elemArray)
{
   int bottom = 0;
   int top = elemArray.length - 1;
   int mid;
   int result = -1;
   while ( bottom <= top )
   {
        mid = (top + bottom) / 2;
        if (elemArray [mid] == key)
        {
          result = mid;
          return result;
        } // if part
        else
        {
          if (elemArray [mid] < key)
             bottom = mid + 1;
          else
             top = mid - 1;
        }
   } //while loop
   return result;
} // search
```

Example 3: Binary Search



Converting from Flow chart to Flow Graph



Example4: Compound Condition

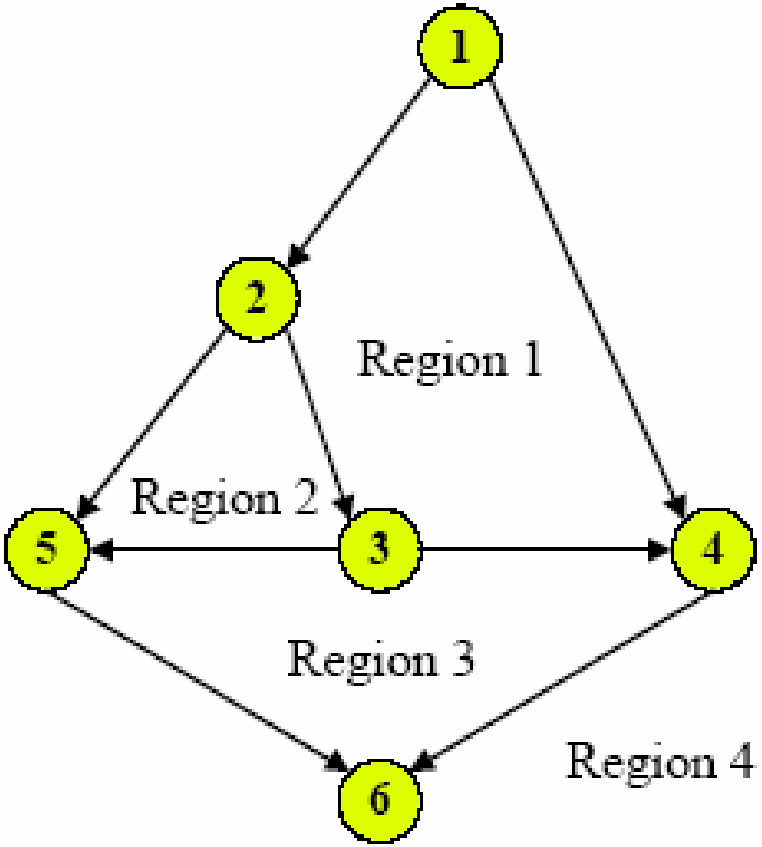
Procedure: reserveVideoCopy return(result)

1. If (status = “available”) OR ((status = “rented”) AND (returnDate $\leq$ requestDate))
2. status = “reserved”
3. link video copy instance to member instance
4. result = “success”
5. Else
6. result = “failure”
7. Endif

End

FLOW GRAPH

Procedure: reserveVideoCopy



V(G)=4

Flow graph node to program statement mapping:

1. 1a
2. 1b
3. 1c
4. 2, 3, 4
5. 5, 6
6. 7

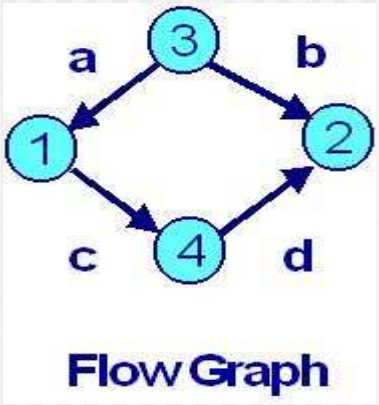
Basis set:

1. 1 4 6
2. 1 2 3 4 6
3. 1 2 3 5 6
4. 1 2 5 6

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

Example:



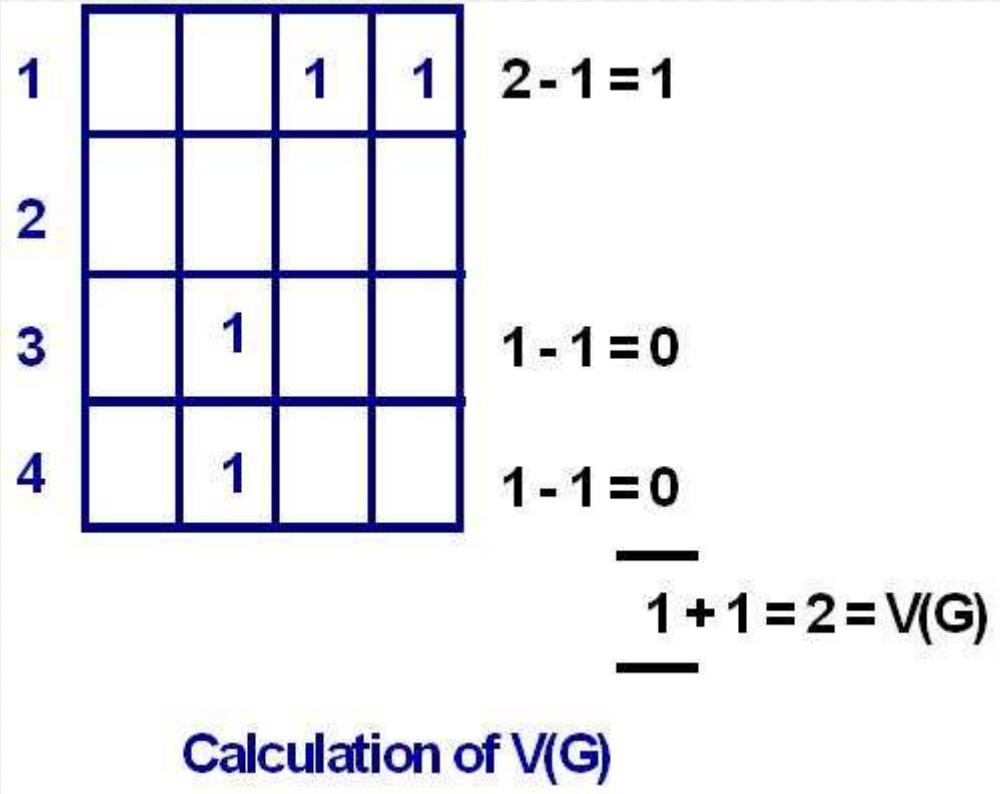
	1	2	3	4
1			a	c
2				
3		b		
4		d		

Graph Matrix

Link weight can be

1. The prob. That link will be executed
2. Processing time expended during traversal of a link
3. The memory/resources required during traversing of a link

V(G) using Graph matrix



Control Structure Testing

- “Errors are much more common in the neighborhood of logical conditions than they are in the locus of sequential processing statements”
- Condition testing — a test case design method that exercises the logical conditions contained in a program module
- Logical condition may define
  - Relational expression: (E1 op E2), where E1 and E2 are arithmetic expressions.
  - Simple condition: Boolean variable or relational expression, possibly proceeded by a NOT operator.
  - Compound condition: composed of two or more simple conditions, Boolean operators and parentheses.
  - Boolean expression : Condition without Relational expressions.

Example:

1. **i <= n**

   This is clearly an example of a relational expression. It involves integers, and integer is an ordered type - so there are three cases that should be covered by tests:

   - **i < n**
   - **i == n**
   - **i > n**

2. **(j >= 1) && (A[j] > A[j+1])**

This is a compound condition, combinations that should be considered:

- **(j < 1) && (A[j] < A[j+1])**
- **(j < 1) && (A[j] == A[j+1])**
- **(j < 1) && (A[j] > A[j+1])**
- **(j == 1) && (A[j] < A[j+1])**
- **(j == 1) && (A[j] == A[j+1])**
- **(j == 1) && (A[j] > A[j+1])**
- **(j > 1) && (A[j] < A[j+1])**
- **(j > 1) && (A[j] == A[j+1])**
- **(j > 1) && (A[j] > A[j+1])**

Data Flow Testing

- **The data flow testing method [Fra93] selects test paths of a program according to the locations of definitions and uses of variables in the program.**
  - **Assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables. For a statement with *S* as its statement number**
    - **DEF(*S*) = {*X* | statement *S* contains a definition of *X*}**
    - **USE(*S*) = {*X* | statement *S* contains a use 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*‘**
  - ***This type of testing is used in targeted fashion for areas of software that are suspect.***

Example:

Pseudo-code Sample

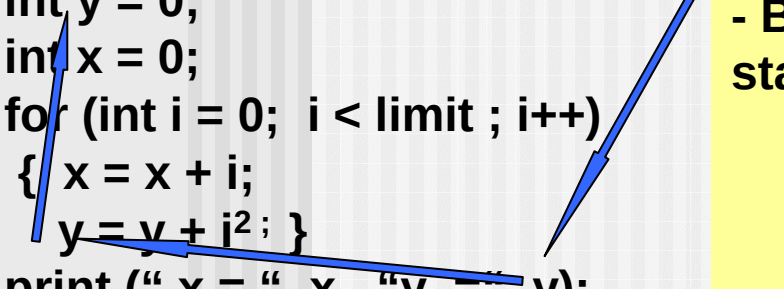
1. **int a, b**
2. **input (a)**
3. **Input (b)**
4. **if (a > 10)**
5. **then Output (a+10)**
6. **else Output (b+10)**
7. **end**

DU(a,2,3), DU(b,3,6)

A simple example of Program Slice

Pseudo code example

```
1. int limit = 10;
2. int y = 0;
3. int x = 0;
4. for (int i = 0;  i < limit ; i++)
5.  {  x = x + i;
6.     y = y + i2 ; }
7. print (“ x = “, x , “y  = “, y);
```

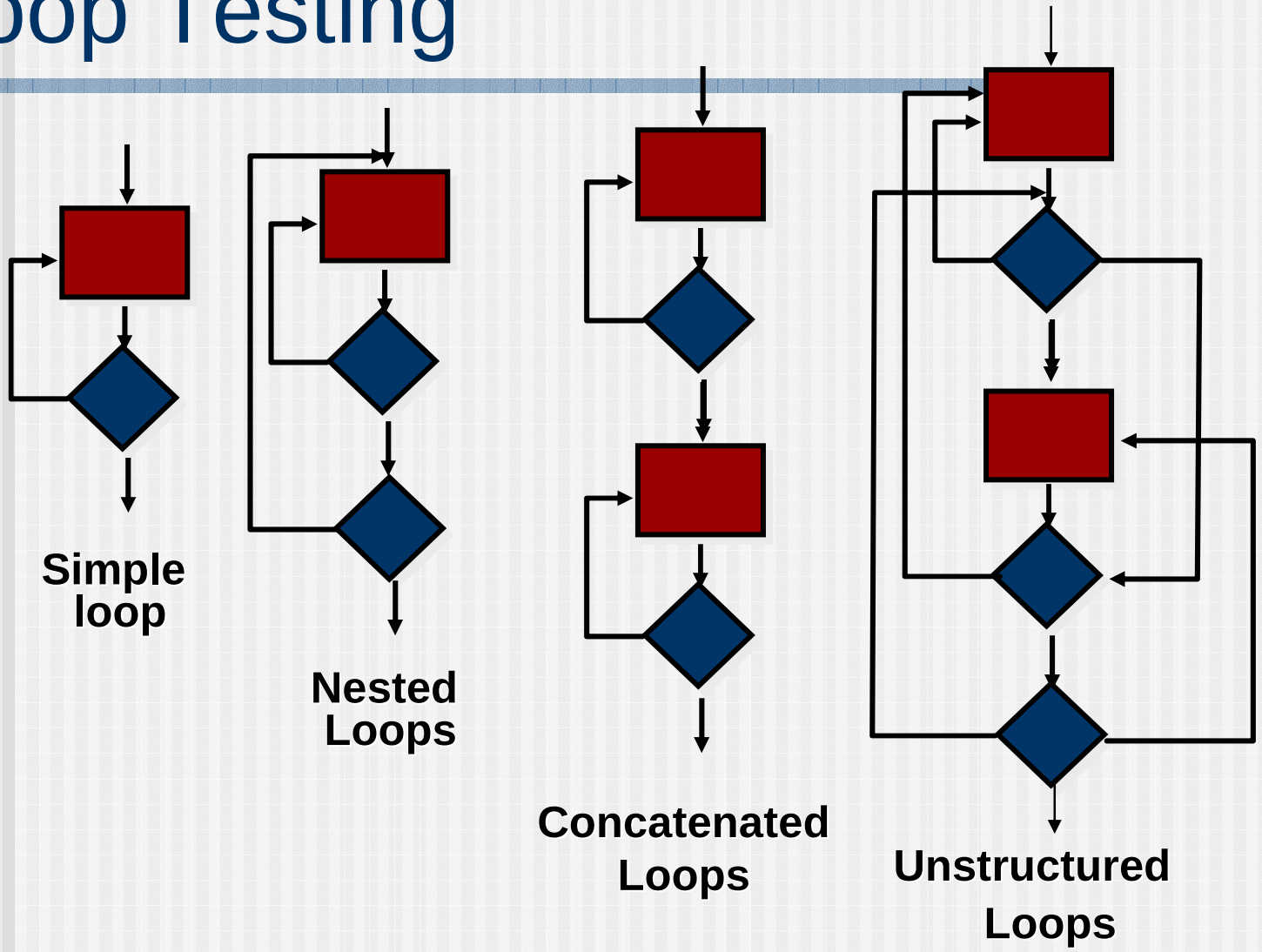


- Consider that our variable of interest is y at statement 7.

- But , on second look, we would pick up statements:
  - 7
  - 6
  - 4 (because the loop influences statement 6)
  - 2
  - 1 (because limit influences statement 4)

-Statements <1,2,4,6,7 > form a ***program slice related to variable, y***

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

- Start at the innermost loop. Set all other loops to minimum values.
- Conduct simple loop tests for the innermost loop while holding the outer loop at their minimum iteration parameter value.
- Work outward, performing tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to “typical” values.
- Continue until all loops have been tested

Loop Testing:

- ***Concatenated Loops***

  If the loops are independent of one another

  then treat each as a simple loop

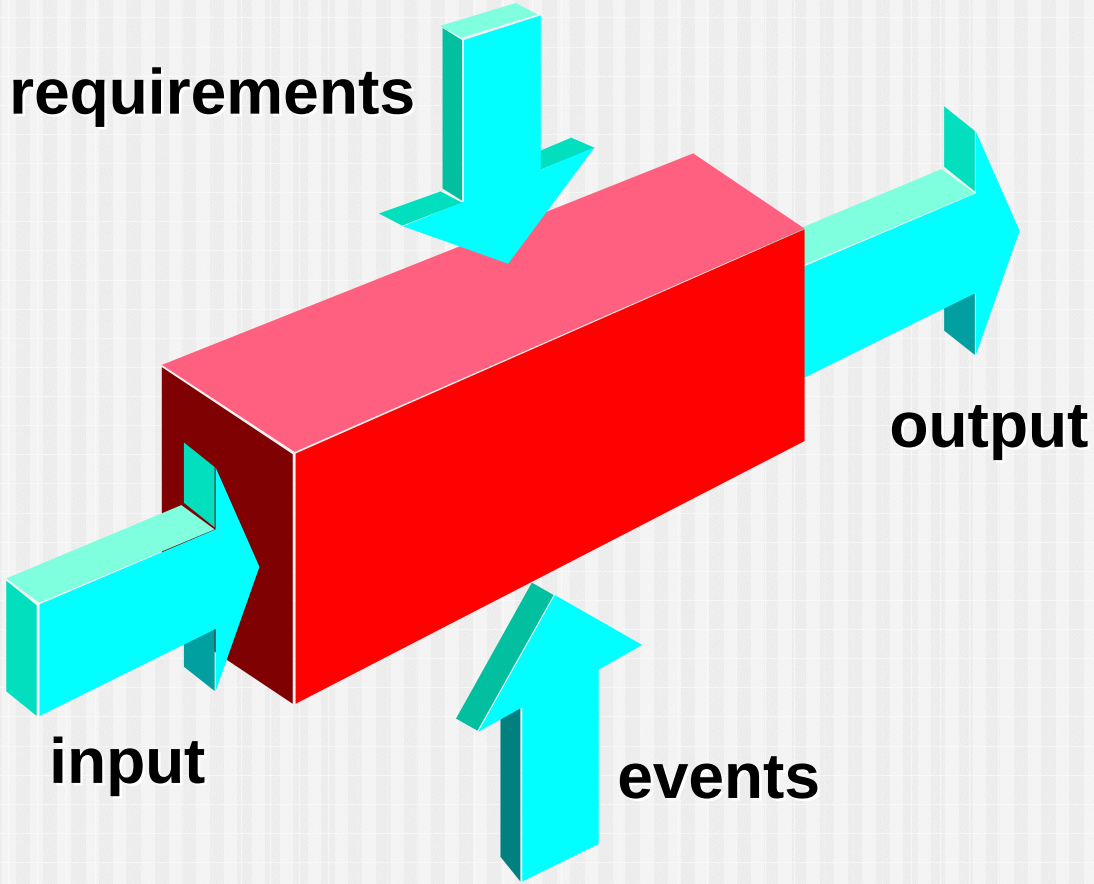
  else treat as nested loops

  end if

- **Unstructured loops**

  This class of loop should be redesigned to reflect the use of the structured programming constructs

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?

Equivalence Class Partitioning

- Black-box testing method that divides the input domain of program into classes of data.
- Each of these classes is an equivalence partition where the program behaves in an equivalent way for each class member
- Test cases should be chosen from each partition.

Example 1

- In a computer store, the computer item can have a quantity between -500 to +500. What are the equivalence classes?
- **Answer**: Valid class: -500 <= QTY <= +500
  Invalid class: QTY > +500
  Invalid class: QTY < -500

Example 2

- Bank account balance can be 1000 to 1500 or 0 to 499 or 2000 (the field type is integer). What are the equivalence classes?

Answer

- Valid class: 0 <= account <= 499
- Valid class: 1000 <= account <= 1500
- Valid class: 2000 <= account <= 2000
- Invalid class: account < 0
- Invalid class: 499 <account<1000
- Invalid class: 1500 < account < 2000
- Invalid class: account > 2000

Boundary Value Analysis

- Errors occur at boundaries of the input domain
- BVA is a test case design technique that complements Equivalence Partitioning.
- Rather than selecting any element of equivalence class, it selects the test cases at the edge of equivalence class.

Ex: IF Bank account can be 500 to 1000. Then test cases are

499, 500, 501, 999,1000,1001

Guidelines

1. Design the test cases at boundaries of data structure.
2. BVA can be applied for output conditions also. Ex. Temp Vs Pressure report in the form of Table. Min and Max allowable entries.

Example 3:

Given the Percentage, Program find the grade of a student as follows.

A Grade – 90 to 100
B Grade – 80 to 89
C Grade – 70 to 79
D Grade – 60 to 69
E Grade – 50 to 59
F Grade - 0 to 50

Derive the test cases for the above specification.

Example 4: Search routine

procedure Search (Key : ELEM ; T: ELEM_ARRAY;
Found : **in out** BOOLEAN; L: **in out** ELEM_INDEX) ;

Pre-condition

-- the array has at least one element
T'FIRST <= T'LAST

Post-condition

-- the element is found and is referenced by L
(Found and T (L) = Key)

or

-- the element is not in the array

Search routine - input partitions

- **Inputs which conform to the pre-conditions**
- **Inputs where a pre-condition does not hold**
- **Inputs where the key element is a member of the array**
- **Inputs where the key element is not a member of the array**

Testing guidelines - sequences

- Test software with sequences which have only a single value
- Use sequences of different sizes in different tests
- Derive tests so that the first, middle and last elements of the sequence are accessed
- Test with sequences of zero length

Search routine - input partitions

Array	**Element**
Single value	In sequence
Single value	Not in sequence
More than 1 value	First element in sequence
More than 1 value	Last element in sequence
More than 1 value	Middle element in sequence
More than 1 value	Not in sequence

Input sequence (T)	**Key** (Key)	**Output** (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

Example 4: Sorting example

- Example: sort (lst, n)
  - Sort a list of numbers
  - The list is between 2 and 1000 elements
- Domains:
  - The list has some item type (of little concern)
  - n is an integer value (sub-range)
- Equivalence classes;
  - n < 2
  - n > 1000
  - 2 <= n <= 1000

Sorting example

- What do you test?
- Not all cases of integers
- Not all cases of positive integers
- Not all cases between 1 and 1001

- Highest payoff for detecting faults is to test around the boundaries of equivalence classes.

- Test n=1, n=2, n=1000, n=1001, and say n= 10
- Five tests versus 1000.

Example 5:

- Myers [MYE79] uses the following program as a self-assessment of one's ability to specify adequate testing: A program reads three integer values. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral. Develop a set of test cases that you feel will adequately test this program.

Graph-Based techniques

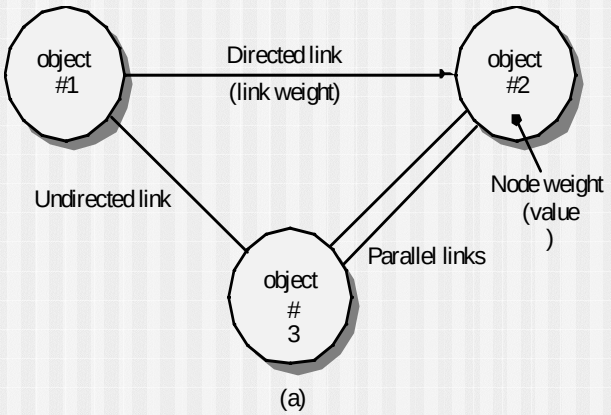
- A graph represents the relationships between

  Data objects and program objects, which helps to derive test cases that search for errors associated with these relationships.

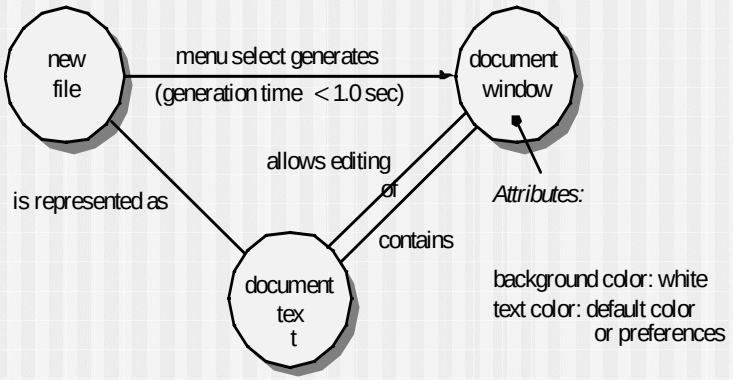
Graph-Based Methods

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.



(a)



(b)

Graph-Based Methods

Behavioral testing methods that can make use of graphs:

1. **Transaction flow modelling:**
   - Nodes represent steps in transaction and the links represent logical connection between steps.
   - DFD's can be used to assist in creating graphs of these types.

2. Finite State Modelling:

   - Nodes represents observable states of software and links represent the transition from one state to other.
   - State transition diagrams can be used for creating graphs.

Graph-Based Methods

Behavioral testing methods that can make use of graphs:

3. **Data flow modelling:**
   - Nodes are data objects and are the transformation that occur to translate one object to another.
   - Example: FICA tax withheld (FTW) is computerd from gross wages (GW) using relationship

     FTW=.62 X GW

Object- Oriented Testing Methods

Challenges of Class Testing

- Encapsulation:
  - Encapsulated attributes and operations create minor obstacles.
  - Difficult to obtain a snapshot of a class without building extra methods which display the classes' state
- Inheritance and polymorphism:
  - Each new context of use (subclass) requires re-testing because a method may be implemented differently (polymorphism).
  - Other unaltered methods within the subclass may use the redefined method and need to be tested
- White box tests:
  - Basis path, condition, data flow and loop tests can all apply to individual methods, but don't test interactions between methods

Class Testing: Random Class Testing

1. **Identify methods applicable to a class**
2. **Define constraints on their use – e.g. the class must always be initialized first**
3. **Identify a minimum test sequence – an operation sequence that defines the minimum life history of the class**
4. **Generate a variety of random (but valid) test sequences – this exercises more complex class instance life histories**

- **Example:**
  1. **An account class in a banking application has *open*, *setup*, *deposit*, *withdraw*, *balance*, *summarize* and *close* methods**
  2. **The account must be opened first and closed on completion**
  3. ***Open – setup – deposit – withdraw – close : Min test sequence***
  4. ***Open – setup – deposit –* [deposit | withdraw | balance | summarize] – withdraw – close*. Generate random test sequences using this template**

Class 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

1. state-based partitioning

categorize and test operations based on their ability to change the state of a class

Ex: Deposit, Withdraw – State operations

Balance, Summarize, creditlimit – nonstate operations.

Test case p1

open, setup, deposit, deposit, withdraw, withdraw, close

Test case p2

open, setup, deposit, summarize, creditlimit, withdraw, close.

Class Testing: Partition Testing

2. Attribute-based partitioning

categorize and test operations based on the attributes that they use

Ex: Attributes balance and creditlimit can be used to define partitions.

- Operations that use creditlimit
- Operations that modify creditlimit
- Operations that do not use/modify creditlimit

Class Testing: Partition Testing

3. category-based partitioning

categorize and test operations based on the generic function each performs e.g.

- initialization
- state changing
- queries
- Termination

Ex: Initialization – open, setup
computational operations- deposit, withdraw,
queries- balance, summarize
Termination- close

OO Integration Testing

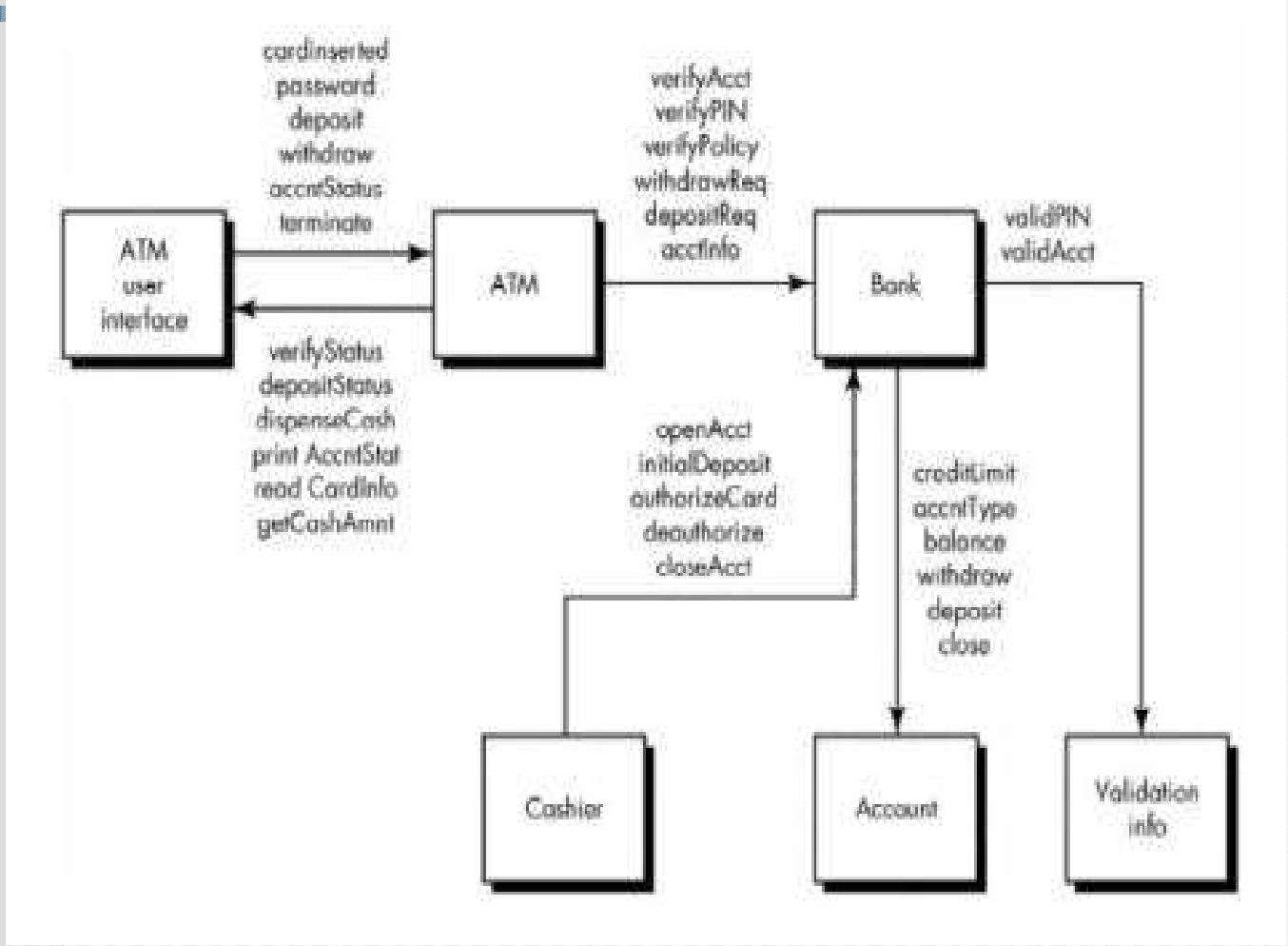
- OO does not have a hierarchical control structure so conventional top-down and bottom-up integration tests have little meaning
- Integration applied three different incremental strategies:
  - Thread-based testing: integrates classes required to respond to one input or event
  - Use-based testing: integrates classes required by one use case
  - Cluster testing: integrates classes required to demonstrate one collaboration

• **What integration testing strategies will you use?**

Random Integration Testing

- Multiple Class Random Testing
  1. For each client class, use the list of class methods to generate a series of random test sequences. Methods will send messages to other server classes.
  2. For each message that is generated, determine the collaborating class and the corresponding method in the server object.
  3. For each method in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits
  4. For each of the messages, determine the next level of methods that are invoked and incorporate these into the test sequence

Class collaboration diagram



Example:

Consider a sequence of operations for the **bank** class relative to an **ATM** class:

verifyAcct•verifyPIN•[[verifyPolicy•withdrawReq]|depositReq|acctInfoREQ]

o A random test case for the **bank** class might be

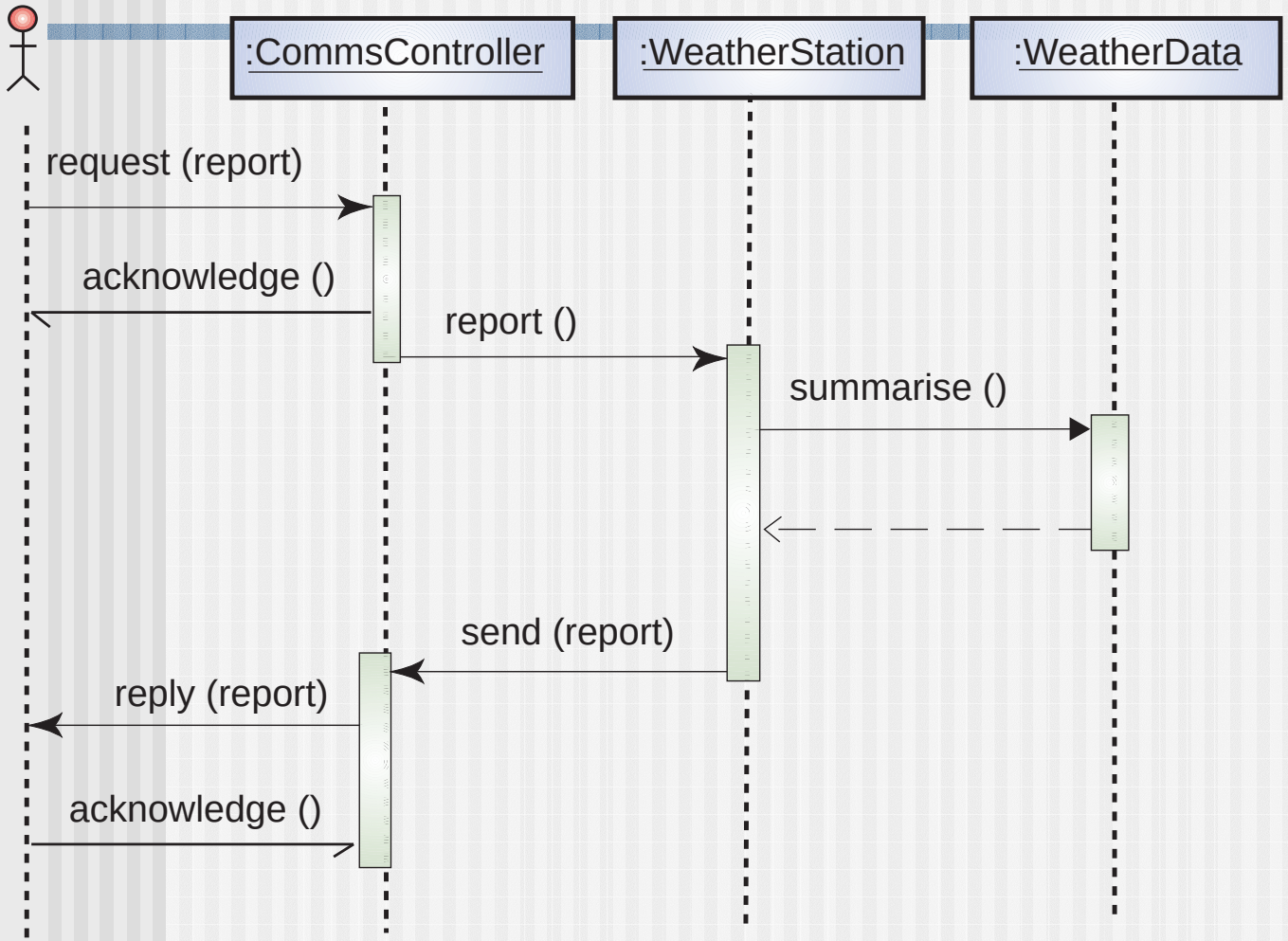
o To test collaboration the messages associated with each of the operations noted in test case *r*3 are considered.

- **Bank** must collaborate with **ValidationInfo** to execute the *verifyAcct* and *verifyPIN.*
- **Bank** must collaborate with **account** to execute *depositReq.*

Scenario-based testing

- Identify scenarios from use-cases and supplement these with interaction diagrams that show the objects involved in the scenario
- Consider the scenario in the weather station system where a report is generated

Collect weather data



Weather station testing

- Thread of methods executed
  - CommsController:request → WeatherStation:report → WeatherData:summarise
- Inputs and outputs
  - Input of report request with associated acknowledge and a final output of a report
  - Can be tested by creating raw data and ensuring that it is summarised properly
  - Use the same raw data to test the WeatherData object

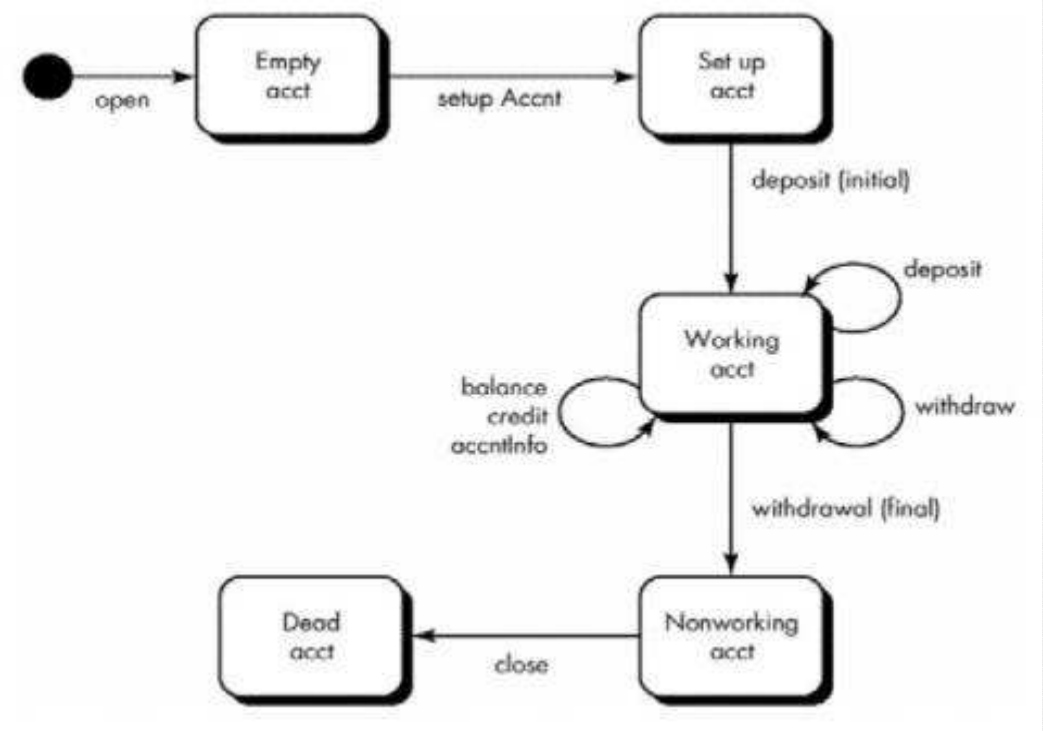
- Hence, a new test case that exercises these collaborations is

$$\text{test case } r_4 = \text{verifyAcct}_{\text{Bank}}[\text{validAcct}_{\text{ValidationInfo}}] \bullet \text{verifyPIN}_{\text{Bank}} \bullet [\text{validPin}_{\text{ValidationInfo}}] \bullet \text{depositReq} \bullet [\text{deposit}_{\text{account}}]$$

Behavioural Integration Testing

- Derive tests from the object-behavioural analysis model
- Each state in a State diagram should be visited in a “breadth-first” fashion.
  - Each test case should exercise a single transition
  - When a new transition is being tested only previously tested transitions are used
  - Each test case is designed around causing a specific transition
- Example:
  - A credit card can move between *undefined*, *defined*, *submitted* and *approved* states
  - The first test case must test the transition out of the start state *undefined* and not any of the other later transitions

Example:


Empty
acct
open
setup Accnt
Set up
acct
deposit (initial)
deposit
Working
acct
balance
credit
accntInfo
withdraw
withdrawal (final)
Dead
acct
close
Nonworking
acct

test case s_1: open • setupAccnt • deposit (initial) • withdraw (final) • close

Description

- State transition diagrams represent the dynamic behavior of a class.
- STD for a class can be used to help derive a sequence of tests that will exercise the dynamic behavior of the class (and those classes that collaborate with it).
- Above STD for the **account** class
  - Initial transitions move through the *empty acct* and *setup acct* states.
  - Majority of all behavior for instances of the class occurs while in the *working acct* state.
  - A final withdrawal and close cause the account class to make transitions to the *nonworking acct* and *dead acct* states

Tests should achieve all state coverag

test case s_1: open • setupAccnt • deposit (initial) • withdraw (final) • close