Software Verification and Validation

- *Verification*
  - Does the product meet system specifications?
  - Have you built the product right?
- *Validation*
  - Does the product meet user expectations?
  - Have you built the right product?

Defect

- A defect is a variance from a desired product attribute.
- These attributes may involve system specifications well as user expectations.
- Anything that may cause customer dissatisfaction is a defect.
- Whether these defects are in system specifications or in the software products, it is essential to point them out and fix.

- *“Death and taxes are inevitable.”* -- Haliburton

- *“Death, taxes, and bugs are the only certainties in the life of a programmer.”*-- Kernighan

Software Testing

- Software testing is the process of examining the software product against its requirements.
- It is a process that involves verification of product with respect to its written requirements and conformance of requirements with user needs.
- Software testing is the process of executing software product on test data and examining its output vis-à-vis the documented behavior.

Software Testing Objective

- The correct approach to testing a scientific theory is not to try to verify it, but to seek to refute the theory. That is to prove that it has errors. (Popper 1965)
- The goal of testing is to expose latent defects in a software system before it is put to use.
- A software tester tries to break the system. The objective is to show the presence of a defect not the absence of it.
- Testing cannot show the absence of a defect. It only increases your confidence in the software.
- exhaustive testing of software is not possible

Successful Test

- *“If you think your task is to find problems then you will look harder for them than if you think your task is to verify that the program has none”* – Myers 1979.

- *“A test is said to be successful if it discovers an error”.*

Limitations of Testing

- In order to prove that a formula or hypothesis is incorrect all you have to do to show only one example in which you prove that the formula or theorem is not working.

- On the other hand, million of examples can be developed to support the hypothesis but this will not prove that it is correct.

- You cannot test a program completely because:
  - The domain of the possible inputs is too large to test.
  - There are too many possible paths through the program to test.

- According to Discrete Mathematics
  - To prove that a formula or hypothesis is incorrect you have to show only one example.
  - To prove that it is correct any numbers of examples are insufficient. You have to give a formal proof of its correctness.

Example

- Test a function that compares two strings of characters stored in an array for equality.

Test Cases

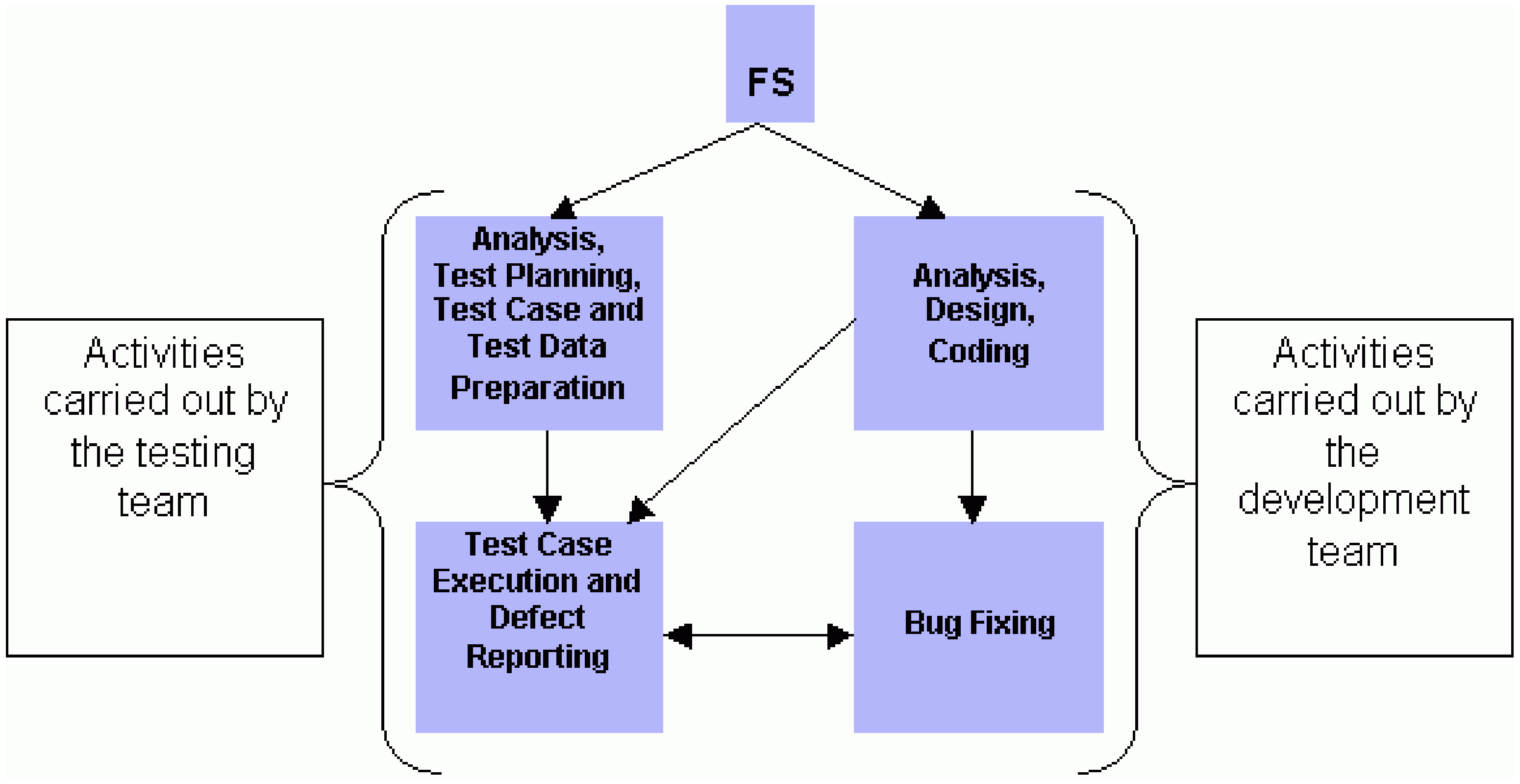
A	**B**	**Expected result**
“cat”	“dog”	False
“”	“”	True
“hen”	“hen”	True
“hen”	“heN”	False
“”	“ball”	False
“cat”	“”	False
“HEN”	“hen”	False
“rat”	“door”	False
“ ”	“ ”	True

```
bool isStringsEqual(char a[], char b[])
{
    bool result;
    if (strlen(a) != strlen(b))
        result = false;
    else {
        for (int i =0; i < strlen(a); i++)
            if (a[i] == b[i])
                result = true;
            else
                result = false;
    }
    return result;
}
```

Test Cases and Test Data

- In order to test a software application, it is necessary to generate test cases and test data that is used in the application.
- Test cases correspond to application functionality such that the tester writes down steps that should be followed to achieve certain functionality.
- Thus a test case involves
  - Input and output specification plus a statement of the function under test.
  - Steps to perform the function
  - Expected results that the software application produces
- Test data includes inputs that have been devised to test the system.

Testing vs. Development


FS
Analysis, Test Planning, Test Case and Test Data Preparation
Analysis, Design, Coding
Test Case Execution and Defect Reporting
Bug Fixing
Activities carried out by the testing team
Activities carried out by the development team

The Developer and Tester

Development	Testing
Development is a creative activity	Testing is a destructive activity
Objective of development is to show that the program works	Objective of testing is to show that the program does not work

Description of Testing Phases

- *Unit Testing:* testing individual components independent of other components.
- *Module Testing:* testing a collection of dependent components – a module encapsulates related components so it can be tested independently.
- *Subsystem Testing:* testing of collection of modules to discover interfacing problems among interacting modules.
- *System Testing:* Integrating subsystems into a system and testing this system as a whole.

Description of Testing Phases

- *Alpha Testing:* acceptance testing for customized projects, in-house testing for products.

- *Beta Testing:* field-testing of product with potential customers who agree to use it and report problem before system is released for general use

Testing Techniques

- Black Box or Functional Testing
- White Box of Structured Testing

Black Box Testing

- a component or system is treated as a black box and it is tested for the required behavior.
- This type of testing is not concerned with how the inputs are transformed into outputs.
- As the system’s internal implementation details are not visible to the tester. He gives inputs using an interface that the system provides and tests the output. If the outputs match with the expected results, system is fine otherwise a defect is found.

Structural Testing (White box)

- we look inside the system and evaluate what it consists of and how is it implemented.

- in white box testing the internal structures of the program are analyzed and test cases are devised that can test these structures.

Effective Testing

- The objective of testing is to discover the maximum number of defects with a minimum number of resources before the system is delivered to the next stage.
- Question is: how to increase the probability of finding a defect?
- As, good testing involves much more than just running the program a few times to see whether it works or not. A good tester carries out a thorough analysis of the program to devise test cases that can be used to test the system systematically and effectively. Problem here is how to develop a representative set of test cases that could test a complete program. That is, selection of a few test cases from a huge set of possibilities. What should the sets of inputs be used to test the system effectively and efficiently?

Example, String Equal

- For how many equal strings do I have to test to be in the comfortable zone?
- For how many unequal strings do I have to test to be in the comfortable zone?
- When should I say that further testing is unlikely to discover another error?

Equivalence Classes or Equivalence Partitioning

- Two tests are considered to be equivalent if it is believed that:
  - If one discovers a defect, the other probably will too, and
  - If one does not discover a defect, the other probably won’t either.

- Equivalence classes help you in designing test cases to test the system effectively and efficiently.

Equivalence Partitioning Guidelines

- Organize your equivalence classes. Write them in some order, use some template, sequence, or group them based on their similarities or distinctions. These partitions can be hierarchical or organized in any other manner.

- Boundary conditions: determine boundary conditions. For example, adding in an empty linked list, adding after the last element, adding before the first element, etc.

- You should not forget invalid inputs that a user can give to a system. For example, widgets on a GUI, numeric instead of alphabets, etc.

Example, String Match

- *Organization*
  - For equivalence partitions, we divide the problem in two obvious categories of equal strings and the other one for unequal strings.

Test Cases for Equivalence Partitions - Equal

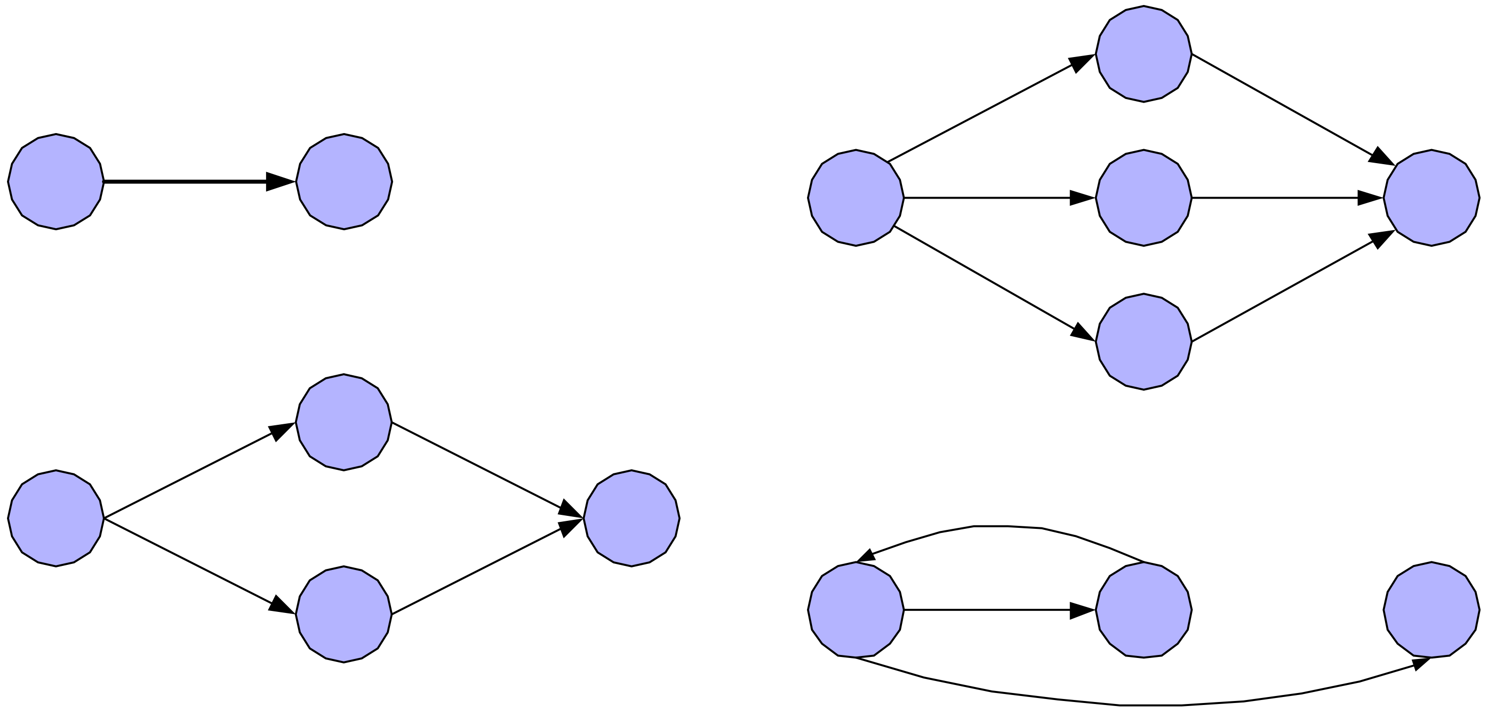
• Two equal strings of arbitrary length		
– All lower case	“cat”	“cat”
– All upper case	“CAT”	“CAT”
– Mixed case	“Cat”	“Cat”
– Numeric values	“123”	“123”
– Two strings with blanks only	“ ”	“ ”
– Numeric and character mixed	“Cat1”	“Cat1”
– Strings with special characters	“Cat#1”	“Cat#1”
• Two NULL strings	“”	“”

Unequal Strings

- Two different equal strings of arbitrary length
  - Two strings with different length “cat” “mouse”
  - Two strings of same length “cat” “dog”
- Check for case sensitivity
  - Same strings different capitalization “Cat” “caT”
- One string is empty
  - First is NULL “” “cat”
  - Second is NULL “cat” “”

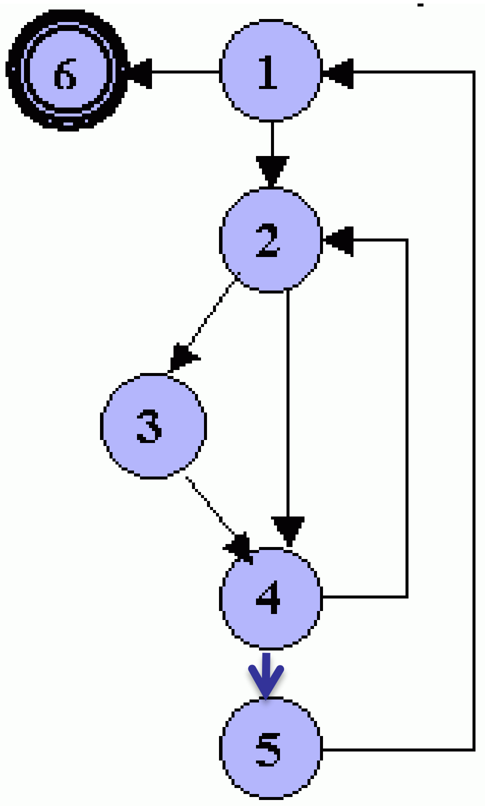
Structural Testing

Basic Code Structures and Flow Graph Notation



Example – Bubble Sort

```
sorted = false;
while (!sorted) {                    //1
    sorted = true;
    for (i=0; i < N-1; i++) {   //2
        if a[i] > a[i+1] {
            swap(a[i], a[i+1]); //3
            sorted = false;
        }                            //4
    }                                //5
}                                    //6
```



```
sorted = false;
while (! sorted)
{
    sorted = true;
    i = 0;
    while (i < size -1)
    {
        if (a[i] > a[i+1])
        {
            sorted = false;
            swap(a[i], a[i+1]);
        }
        i++;
    }
}
```

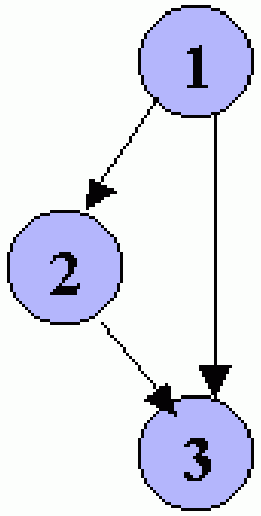
Path

- A sequence of edges in the flow graph form input to output

White Box Testing - Coverage

- Statement Coverage
- Branch Coverage
- Path Coverage

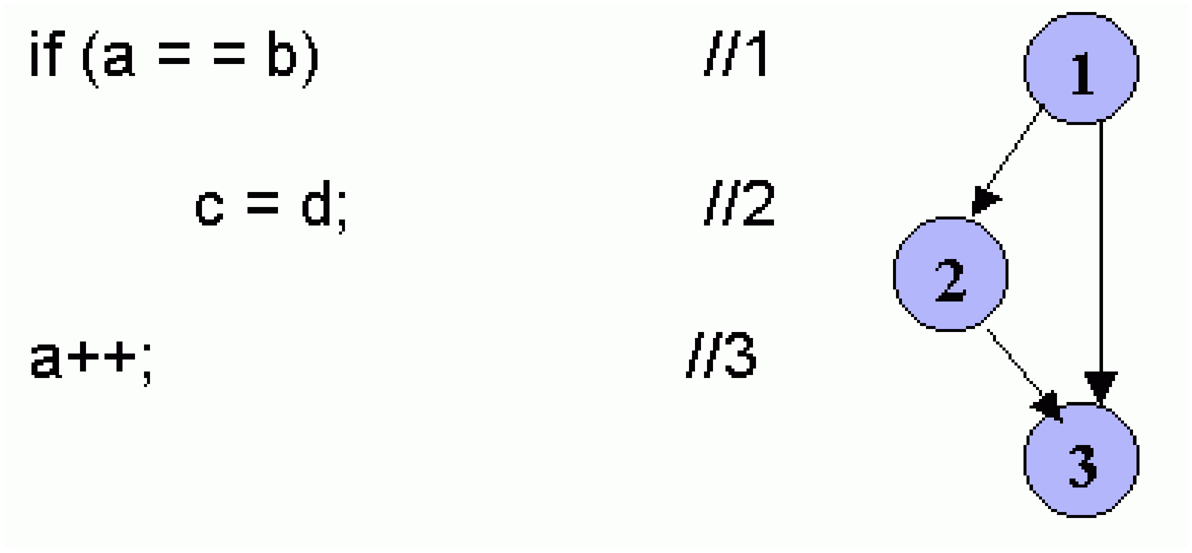
```
if (a = = b)                //1
        c = d;              //2
a++;                        //3
```



- statement coverage
  - test cases

  Input: a = 2, b = 2, d = 3

  expected result: a = 3; b = 2; c = 3; d = 3



```
if (a = = b)                //1

      c = d;                //2

a++;                        //3
```

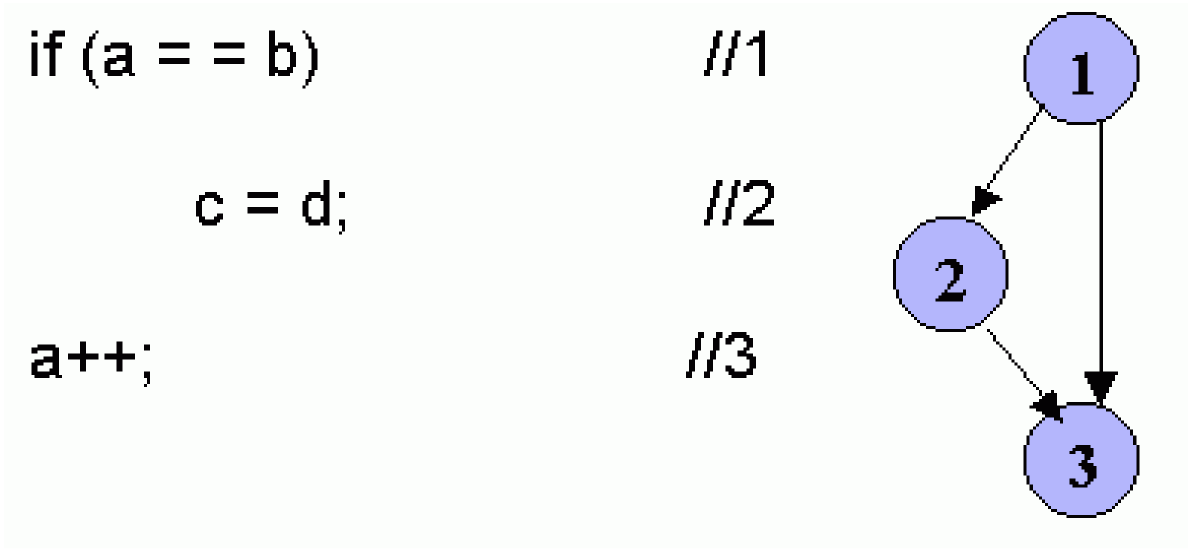
- branch coverage
  - test cases

Input: a = 2; b = 2; c = 3; d = 4

expected result: a = 3; b = 2; c = 4; d = 4

input: a = 1, b = 2; c = 3; d = 4

expected result: a = 2; b = 2; c = 3; d = 4



```
if (a = = b)              //1

      c = d;              //2

a++;                      //3
```

- path coverage
  - test cases

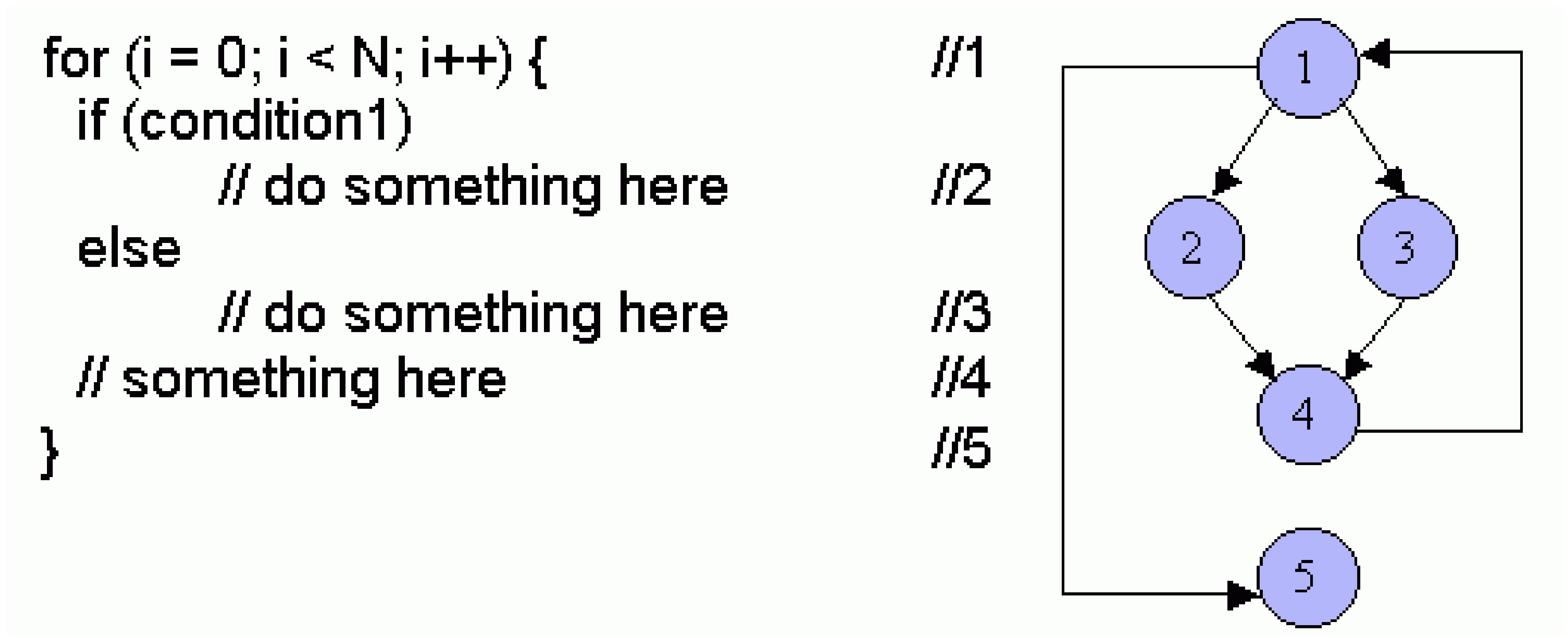
Input: a = 2; b = 2; c = 3; d = 4

expected result: a = 3; b = 2; c = 4; d = 4

input: a = 1, b = 2; c = 3; d = 4

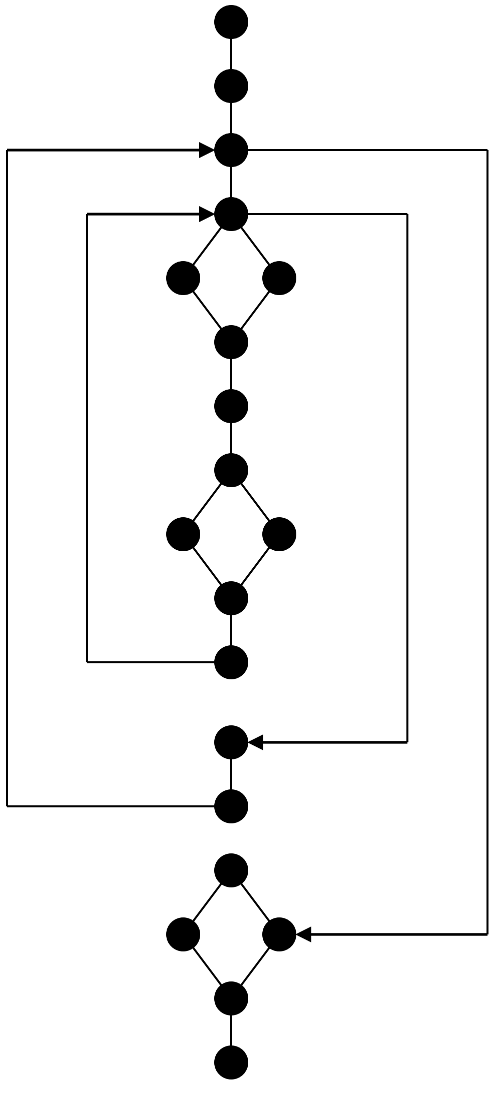
expected result: a = 2; b = 2; c = 3; d = 4

Paths in a Program Containing Loops



```
for (i = 0; i < N; i++) {              //1
  if (condition1)
          // do something here        //2
  else
          // do something here        //3
  // something here                   //4
}                                     //5
```

A simple graph with many paths



Cyclomatic Complexity

- Measure of complexity of code
- Gives an upper bound on the test cases needed for branch coverage
- See chapter for details

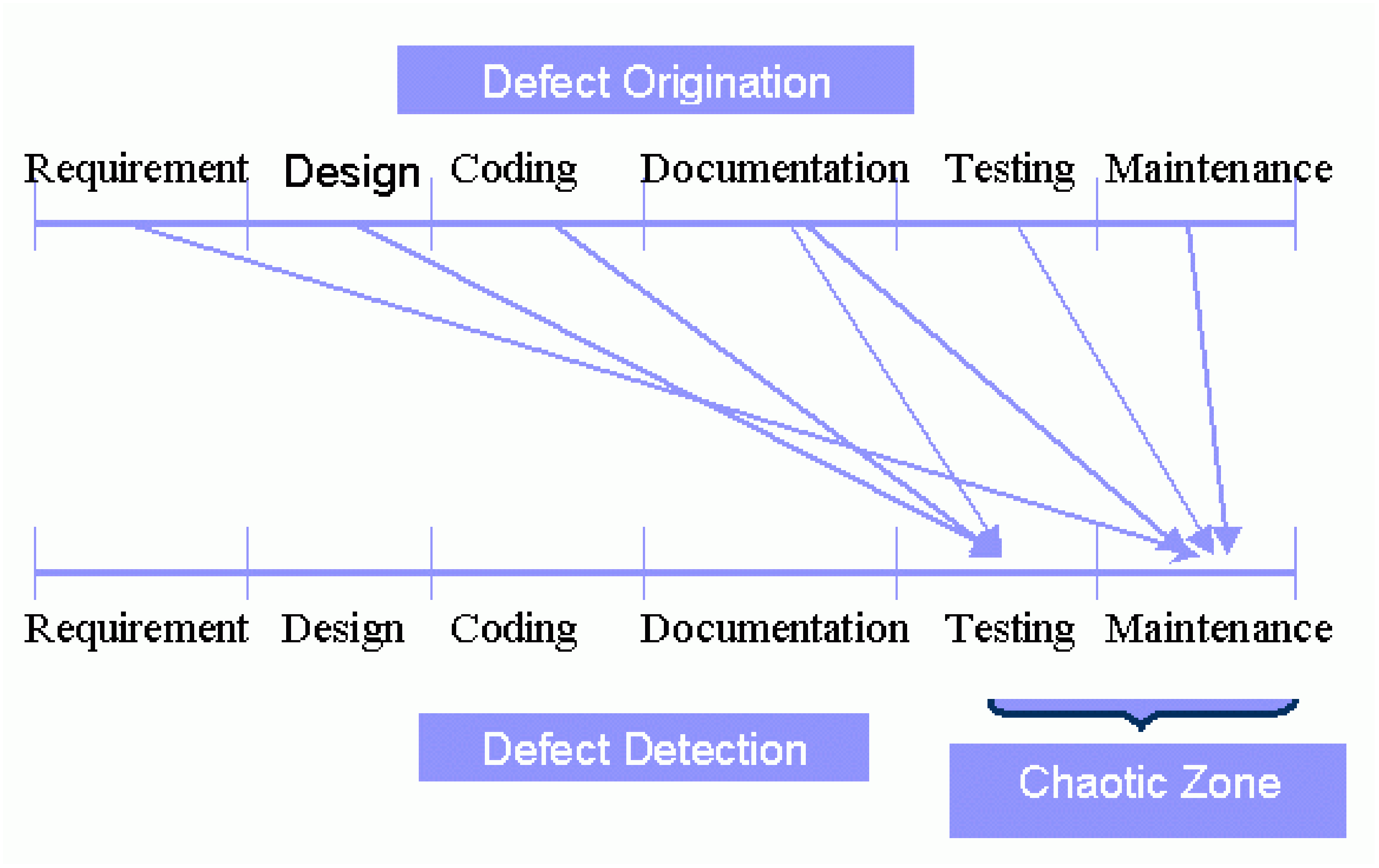
Defect Removal Efficiency

Design Inspection					●		●	●	●
Code Inspection				●		●		●	●
Quality Assurance			●			●	●		●
Testing		●	●	●	●	●	●	●	●
Worst	30%	37%	50%	55%	65%	75%	77%	85%	95%
Median	40%	53%	65%	70%	80%	87%	90%	97%	99%
Best	50%	60%	75%	80%	87%	93%	95%	99%	99.9%

Defect Origination

- In inspections the emphasis is on early detection and fixing of defects from the program.
  - Requirements
  - Design
  - Coding
  - User documentation
  - Testing itself can cause defects due to bad fixes
  - Change requests at the maintenance or initial usage time

Inspection and Chaotic Zone



Inspection versus Testing

- Inspections and testing are complementary and not opposing verification techniques.
- Both should be used during the verification and validation process.
- Inspection does not require execution of program and they maybe used before implementation.
- Inspections may be applied to any representation of the system (requirements, design, test data, etc.)
- Inspections can check conformance with a specification but not conformance with the customer’s real requirements.
- Inspections cannot check non-functional characteristics such as performance, usability etc.

Inspection versus Testing

- Many different defects may be discovered in a single inspection.
- In testing, one defect may mask another so several executions are required for inspections, checklists are prepared that contain information regarding defects.
- Reuse domain and programming knowledge of the viewers likely to help in preparing these checklists.
- Inspections involve people examining the source representation with the aim of discovering anomalies and defects.

Inspection pre-Conditions

- A precise specification must be available before inspections.
- Team members must be familiar with the organization standards.
- In addition to it, syntactically correct code must be available to the inspectors.
- Inspectors should prepare a checklist that can help them during the inspection process.

Inspection Checklists

- Checklist of common errors in a program should be developed and used to drive the inspection process.
- checklists are programming language dependent