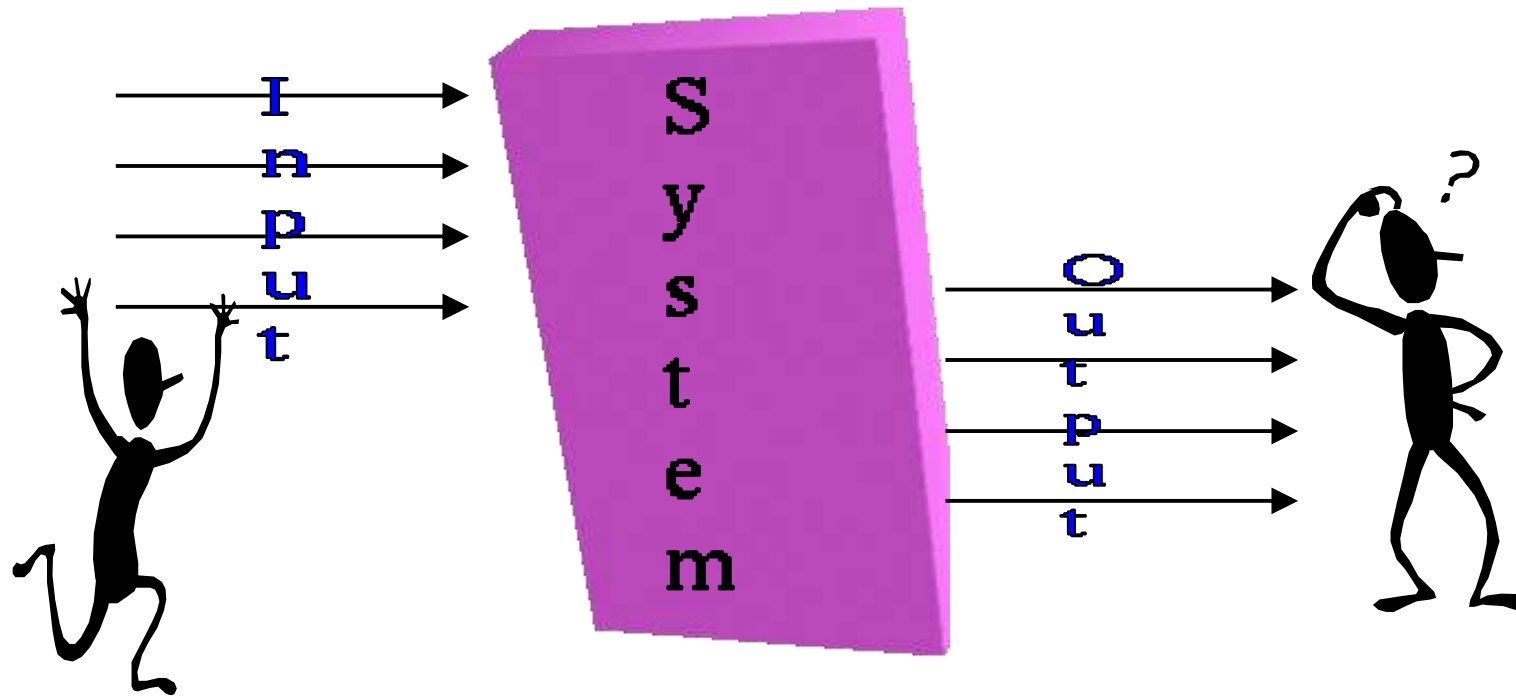


# Testing strategies and tactics

# How Do You Test a Program?

- . Input test data to the program.
- . Observe the output:
  - Check if the program behaved as expected.

# How Do You Test a Program?



# How Do You Test a Program?

- If the program does not behave as expected:
  - Note the conditions under which it failed.
  - Later debug and correct.

# Overview of Testing Activities

- . Test Suite Design
- . Run test cases and observe results to detect failures.
- . Debug to locate errors
- . Correct errors.

# Error, Faults, and Failures

- A failure is a manifestation of an error (aka defect or bug).
  - Mere presence of an error may not lead to a failure.

# Error, Faults, and Failures

- A fault is an incorrect state entered during program execution:
  - A variable value is different from what it should be.
  - A fault may or may not lead to a failure.

# Test cases and Test suites

- Test a software using a set of carefully designed test cases:
  - The set of all test cases is called the test suite



# Test cases and Test suites

- A **test case** is a triplet  $[I, S, O]$ 
  - $I$  is the data to be input to the system,
  - $S$  is the state of the system at which the data will be input,
  - $O$  is the expected output of the system.

# Verification versus Validation

- Verification is the process of determining:
  - Whether output of one phase of development conforms to its previous phase.
- Validation is the process of determining:
  - Whether a fully developed system conforms to its SRS document.

# Verification versus Validation

- Verification is concerned with phase containment of errors,
  - Whereas the aim of validation is that the final product be error free.

# Design of Test Cases

- Exhaustive testing of any non-trivial system is impractical:
  - Input data domain is extremely large.
- Design an **optimal test suite**:
  - Of reasonable size and
  - Uncovers as many errors as possible.

# Design of Test Cases

- If test cases are selected randomly:
  - Many test cases would not contribute to the significance of the test suite,
  - Would not detect errors not already being detected by other test cases in the suite.
- Number of test cases in a randomly selected test suite:
  - Not an indication of effectiveness of testing.

# Design of Test Cases

- Testing a system using a large number of randomly selected test cases:
  - Does not mean that many errors in the system will be uncovered.
- Consider following example:
  - Find the maximum of two integers  $x$  and  $y$ .

# Design of Test Cases

- The code has a simple programming error:
- `If (x>y) max = x;`  
`else max = x;`
- Test suite `{(x=3,y=2):(x=2,y=3)}` can detect the error,
- A larger test suite `{(x=3,y=2):(x=4,y=3); (x=5,y=1)}` does not detect the error.

# Design of Test Cases

- Systematic approaches are required to design an **optimal test suite**:
  - Each test case in the suite should detect different errors.



# Design of Test Cases

- There are essentially two main approaches to design test cases:
  - Black-box approach
  - White-box (or glass-box) approach

# Black-Box Testing

- Test cases are designed using only **functional specification** of the software:
  - Without any knowledge of the internal structure of the software.
- For this reason, black-box testing is also known as **functional testing**.

# White-box Testing

- Designing white-box test cases:
  - Requires knowledge about the internal structure of software.
  - White-box testing is also called structural testing.
  - In this unit we will not study white-box testing.

# Black-Box Testing

- There are essentially two main approaches to design black box test cases:
  - Equivalence class partitioning
  - Boundary value analysis

# Equivalence Class Partitioning

- Input values to a program are partitioned into **equivalence classes**.
- Partitioning is done such that:
  - **Program behaves in similar ways to every input value belonging to an equivalence class.**

# Why Define Equivalence Classes?

- Test the code with just one representative value from each equivalence class:
  - As good as testing using any other values from the equivalence classes.

# Equivalence Class Partitioning

- How do you determine the equivalence classes?
  - Examine the input data.
  - Few general guidelines for determining the equivalence classes can be given

# Equivalence Class Partitioning

- If the input data to the program is specified by a **range of values**:
  - e.g. numbers between 1 to 5000.
  - One valid and two invalid equivalence classes are defined.



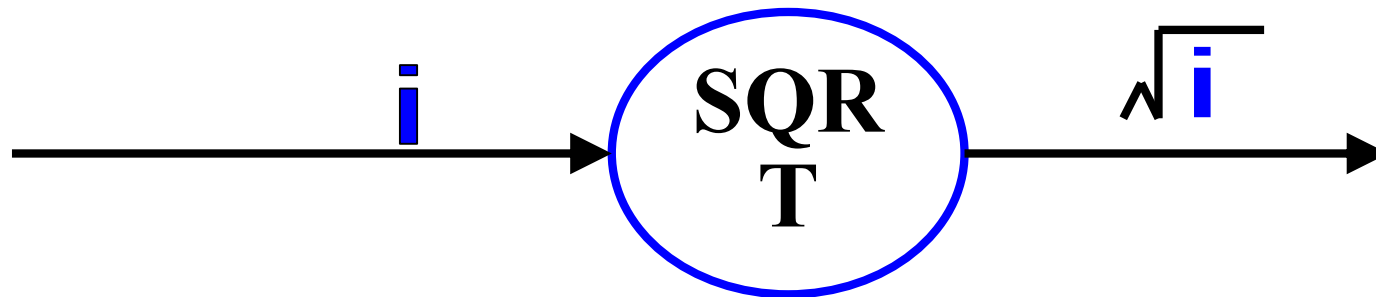


# Equivalence Class Partitioning

- If input is an enumerated set of values:
  - e.g. {a,b,c}
  - One equivalence class for valid input values.
  - Another equivalence class for invalid input values should be defined.

# Example

- A program reads an input value in the range of 1 and 5000:
  - Computes the square root of the input number



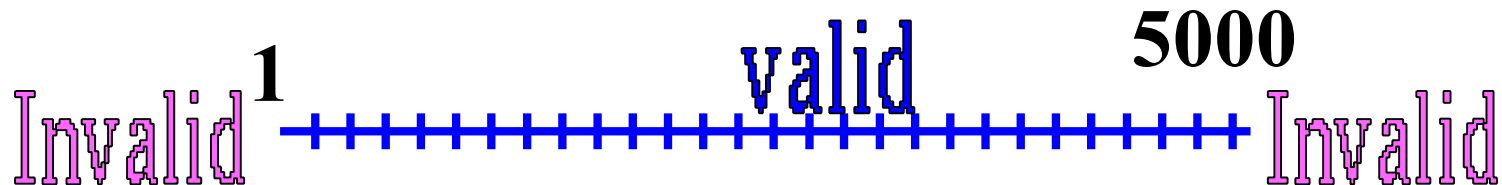
# Example (cont.)

- There are three equivalence classes:
  - The set of negative integers,
  - Set of integers in the range of 1 and 5000,
  - Integers larger than 5000.



# Example (cont.)

- The test suite must include:
  - Representatives from each of the three equivalence classes:
  - A possible test suite can be:  $\{-5, 500, 6000\}$ .



# Boundary Value Analysis

- Some typical programming errors occur:
  - At boundaries of equivalence classes
  - Might be purely due to psychological factors.
- Programmers often fail to see:
  - Special processing required at the boundaries of equivalence classes.

# Boundary Value Analysis

- Programmers may improperly use `<` instead of `<=`
- Boundary value analysis:
  - Select test cases at the boundaries of different equivalence classes.

# Example

- For a function that computes the square root of an integer in the range of 1 and 5000:
  - Test cases must include the values: {0,1,5000,5001}.



# Debugging

- Once errors are identified:
  - It is necessary identify the precise location of the errors and to fix them.
- Each debugging approach has its own advantages and disadvantages:
  - Each is useful in appropriate circumstances.



# Brute-Force method

- This is the most common method of debugging:
  - Least efficient method.
  - Program is loaded with print statements
  - Print the intermediate values
  - Hope that some of printed values will help identify the error.

# Symbolic Debugger

- Brute force approach becomes more systematic:
  - With the use of a symbolic debugger,
  - Symbolic debuggers get their name for historical reasons
  - Early debuggers let you only see values from a program dump:
    - Determine which variable it corresponds to.

# Symbolic Debugger

- Using a symbolic debugger:
  - Values of different variables can be easily checked and modified
  - Single stepping to execute one instruction at a time
  - Break points and watch points can be set to test the values of variables.

# Backtracking

- This is a fairly common approach.
- Beginning at the statement where an error symptom has been observed:
  - Source code is traced backwards until the error is discovered.

# Example

```
int main(){  
    int i,j,s;  
    i=1;  
    while(i<=10){  
        s=s+i;  
        i++; j=j++;}  
    printf(“%d”,s);  
}
```

# Backtracking

- Unfortunately, as the number of source lines to be traced back increases,
  - the number of potential backward paths increases
  - becomes unmanageably large for complex programs.

# Cause-elimination method

- Determine a list of causes:
  - which could possibly have contributed to the error symptom.
  - tests are conducted to eliminate each.
- A related technique of identifying error by examining error symptoms:
  - software fault tree analysis.

# Program Slicing

- This technique is similar to back tracking.
- However, the search space is reduced by defining slices.
- A slice is defined for a particular variable at a particular statement:
  - set of source lines preceding this statement which can influence the value of the variable.



# Example

```
int main(){  
    int i,s;  
    i=1; s=1;  
    while(i<=10){  
        s=s+i;  
        i++;}  
    printf(“%d”,s);  
    printf(“%d”,i);  
}
```

# Debugging Guidelines

- Debugging usually requires a thorough understanding of the program design.
- Debugging may sometimes require full redesign of the system.
- A common mistake novice programmers often make:
  - not fixing the error but the error symptoms.

# Debugging Guidelines

- Be aware of the possibility:
  - an error correction may introduce new errors.
- After every round of error-fixing:
  - regression testing must be carried out.

# Program Analysis Tools

- An automated tool:
  - takes program source code as input
  - produces reports regarding several important characteristics of the program,
  - such as size, complexity, adequacy of commenting, adherence to programming standards, etc.

# Program Analysis Tools

- Some program analysis tools:
  - Produce reports regarding the adequacy of the test cases.
- There are essentially two categories of program analysis tools:
  - Static analysis tools
  - Dynamic analysis tools

# Static Analysis Tools

- Static analysis tools:
  - Assess properties of a program without executing it.
  - Analyze the source code
    - Provide analytical conclusions.

# Static Analysis Tools

- Whether coding standards have been adhered to?
  - Commenting is adequate?
- Programming errors such as:
  - uninitialized variables
  - mismatch between actual and formal parameters.
  - Variables declared but never used, etc.

# Static Analysis Tools

- Code walk through and inspection can also be considered as static analysis methods:
  - However, the term static program analysis is generally used for automated analysis tools.



# Dynamic Analysis Tools

- Dynamic program analysis tools require the program to be executed:
  - its behavior recorded.
  - Produce reports such as adequacy of test cases.

# Testing

- . The aim of testing is to identify all defects in a software product.
- . However, in practice even after thorough testing:
  - one cannot guarantee that the software is error-free.

# Testing

- The input data domain of most software products is very large:
  - It is not practical to test the software exhaustively with each input data value.

# Testing

- Testing does however expose many errors:
  - Testing provides a practical way of reducing defects in a system
  - Increases the users' confidence in a developed system.

# Testing

- Testing is an important development phase:
  - requires the maximum effort among all development phases.
- In a typical development organization:
  - maximum number of software engineers can be found to be engaged in testing activities.

# Testing

- Many engineers have the wrong impression:
  - testing is a secondary activity
  - it is intellectually not as stimulating as the other development activities, etc.

# Testing

- Testing a software product is in fact:
  - as much challenging as initial development activities such as specification, design, and coding.
- Also, testing involves a lot of creative thinking.

# Testing

- Software products are tested at three levels:
  - Unit testing
  - Integration testing
  - System testing



# Unit testing

- . During unit testing, modules are tested in isolation:
  - If all modules were to be tested together:
    - . it may not be easy to determine which module has the error.

# Unit testing

- Unit testing reduces debugging effort several folds.
  - Programmers carry out unit testing immediately after they complete the coding of a module.

# Integration testing

- After different modules of a system have been coded and unit tested:
  - modules are integrated in steps according to an integration plan
  - partially integrated system is tested at each integration step.

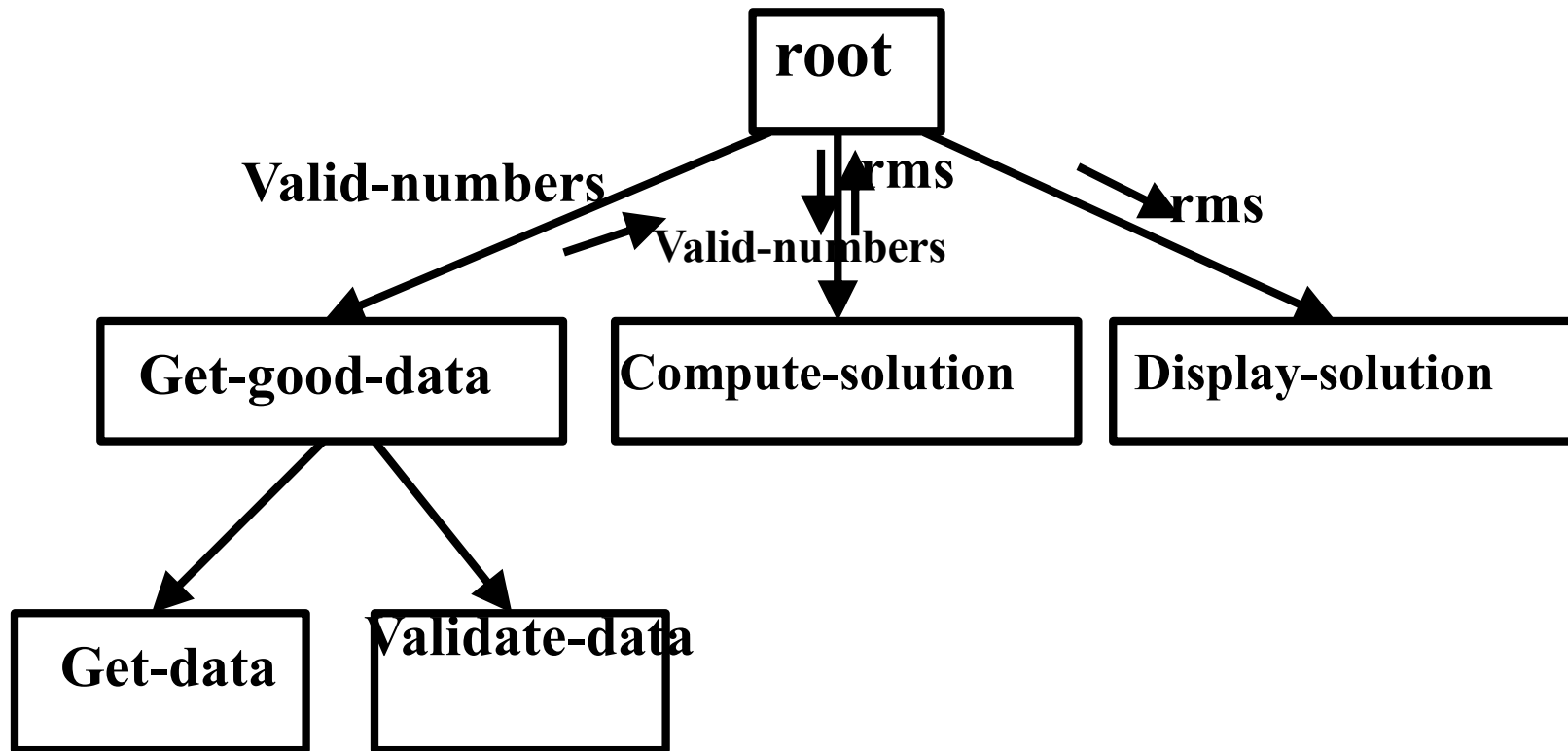
# System Testing

- System testing involves:
  - validating a fully developed system against its requirements.

# Integration Testing

- Develop the integration plan by examining the structure chart :
  - big bang approach
  - top-down approach
  - bottom-up approach
  - mixed approach

# Example Structured Design



# Big Bang Integration Testing

- Big bang approach is the simplest integration testing approach:
  - all the modules are simply put together and tested.
  - this technique is used only for very small systems.

# Big Bang Integration Testing

- Main problems with this approach:
  - If an error is found:
    - It is very difficult to localize the error
    - The error may potentially belong to any of the modules being integrated.
  - Debugging errors found during big bang integration testing are very expensive to fix.



# Bottom-up Integration Testing

- Integrate and test the bottom level modules first.
- A disadvantage of bottom-up testing:
  - when the system is made up of a large number of small subsystems.
  - This extreme case corresponds to the big bang approach.

# Top-down integration testing

- Top-down integration testing starts with the main routine:
  - and one or two subordinate routines in the system.
- After the top-level 'skeleton' has been tested:
  - immediate subordinate modules of the 'skeleton' are combined with it and tested.

# Mixed Integration Testing

- **Mixed (or sandwiched) integration testing:**
  - **uses both top-down and bottom-up testing approaches.**
  - **Most common approach**

# Integration Testing

- In top-down approach:
  - testing waits till all top-level modules are coded and unit tested.
- In bottom-up approach:
  - testing can start only after bottom level modules are ready.

# System Testing

- . There are three main kinds of system testing:
  - Alpha Testing
  - Beta Testing
  - Acceptance Testing

# Alpha Testing

- System testing is carried out by the test team within the developing organization.

# Beta Testing

- System testing performed by a select group of friendly customers.

# Acceptance Testing

- System testing performed by the customer himself:
  - to determine whether the system should be accepted or rejected.



# Stress Testing

- Stress testing (aka endurance testing):
  - impose abnormal input to stress the capabilities of the software.
  - Input data volume, input data rate, processing time, utilization of memory, etc. are tested beyond the designed capacity.

# How Many Errors are Still Remaining?

- Seed the code with some known errors:
  - artificial errors are introduced into the program.
  - Check how many of the seeded errors are detected during testing.

# Error Seeding

- Let:
  - $N$  be the total number of errors in the system
  - $n$  of these errors be found by testing.
  - $S$  be the total number of seeded errors,
  - $s$  of the seeded errors be found during testing.

# Error Seeding

- $n/N = s/S$
- $N = S \ n/s$
- remaining defects:  
$$N - n = n \ ((S - s)/s)$$

# Example

- 100 errors were introduced.
- 90 of these errors were found during testing
- 50 other errors were also found.
- Remaining errors=  
 $50 (100-90)/90 = 6$

# Error Seeding

- The kind of seeded errors should match closely with existing errors:
  - However, it is difficult to predict the types of errors that exist.
- Categories of remaining errors:
  - can be estimated by analyzing historical data from similar projects.

# Summary

- Exhaustive testing of almost any non-trivial system is impractical.
  - we need to design an optimal test suite that would expose as many errors as possible.

# Summary

- If we select test cases randomly:
  - many of the test cases may not add to the significance of the test suite.
- There are two approaches to testing:
  - black-box testing
  - white-box testing.



# Summary

- Black box testing is also known as **functional testing**.
- Designing black box test cases:
  - **Requires understanding only SRS document**
  - **Does not require any knowledge about design and code.**
- Designing white box testing requires knowledge about design and code.

# Summary

- We discussed black-box test case design strategies:
  - Equivalence partitioning
  - Boundary value analysis
- We discussed some important issues in integration and system testing.

# Program Testing

- Introduction to Testing.
- White-box testing:
  - statement coverage
  - path coverage
  - branch testing
  - condition coverage
  - Cyclomatic complexity
- Summary

# Black-box Testing

- Test cases are designed using only **functional specification** of the software:
  - Without any knowledge of the internal structure of the software.
- For this reason, black-box testing is also known as **functional testing**.

# White-Box Testing

- Designing white-box test cases:
  - Requires knowledge about the internal structure of software.
  - White-box testing is also called structural testing.

# Black-Box Testing

- Two main approaches to design black box test cases:
  - Equivalence class partitioning
  - Boundary value analysis

# White-Box Testing

- There exist several popular white-box testing methodologies:
  - Statement coverage
  - Branch coverage
  - Path coverage
  - Condition coverage
  - Mutation testing
  - Data flow-based testing



# Coverage-Based Testing Versus Fault-Based Testing

- Idea behind coverage-based testing:
  - Design test cases so that certain program elements are executed (or covered).
  - Example: statement coverage, path coverage, etc.
- Idea behind fault-based testing:
  - Design test cases that focus on discovering certain types of faults.
  - Example: Mutation testing.

# Statement Coverage

- Statement coverage methodology:
  - Design test cases so that every statement in the program is executed at least once.

# Statement Coverage

- The principal idea:
  - Unless a statement is executed,
  - We have no way of knowing if an error exists in that statement.

# Statement Coverage Criterion

- Observing that a statement behaves properly for one input value:
  - No guarantee that it will behave correctly for all input values.

# Example

```
. int f1(int x, int y){  
. 1 while (x != y){  
. 2     if (x>y) then  
. 3         x=x-y;  
. 4     else y=y-x;  
. 5 }  
. 6 return x; }
```

Euclid's GCD Algorithm

# Euclid's GCD Computation Algorithm

- By choosing the test set  $\{(x=3, y=3), (x=4, y=3), (x=3, y=4)\}$ 
  - All statements are executed at least once.

# Branch Coverage

- Test cases are designed such that:
  - Different branch conditions
    - Given true and false values in turn.

# Branch Coverage

- Branch testing guarantees statement coverage:
  - A stronger testing compared to the statement coverage-based testing.



# Stronger Testing

- Test cases are a superset of a weaker testing:
  - A stronger testing covers at least all the elements of the elements covered by a weaker testing.

# Example

```
. int f1(int x,int y){  
. 1 while (x != y){  
. 2     if (x>y) then  
. 3         x=x-y;  
. 4     else y=y-x;  
. 5 }  
. 6 return x;      }
```

# Example

- Test cases for branch coverage can be:
- $\{(x=3, y=3), (x=3, y=2), (x=4, y=3), (x=3, y=4)\}$

# Condition Coverage

- Test cases are designed such that:
  - Each component of a composite conditional expression
    - Given both true and false values.

# Example

- Consider the conditional expression
  - $((c1.and.c2).or.c3)$ :
- Each of  $c1$ ,  $c2$ , and  $c3$  are exercised at least once,
  - i.e. given true and false values.

# Branch Testing

- Branch testing is the simplest condition testing strategy:
  - Compound conditions appearing in different branch statements
    - Are given true and false values.

# Branch testing

- **Condition testing**
  - Stronger testing than branch testing.
- **Branch testing**
  - Stronger than statement coverage testing.

# Condition coverage

- Consider a boolean expression having  $n$  components:
  - For condition coverage we require  $2^n$  test cases.
- Condition coverage-based testing technique:
  - Practical only if  $n$  (the number of component conditions) is small.



# Path Coverage

- Design test cases such that:
  - All linearly independent paths in the program are executed at least once.
- Defined in terms of
  - Control flow graph (CFG) of a program.

# Path Coverage-Based Testing

- To understand the path coverage-based testing:
  - we need to learn how to draw control flow graph of a program.
- A control flow graph (CFG) describes:
  - the sequence in which different instructions of a program get executed.
  - the way control flows through the program.

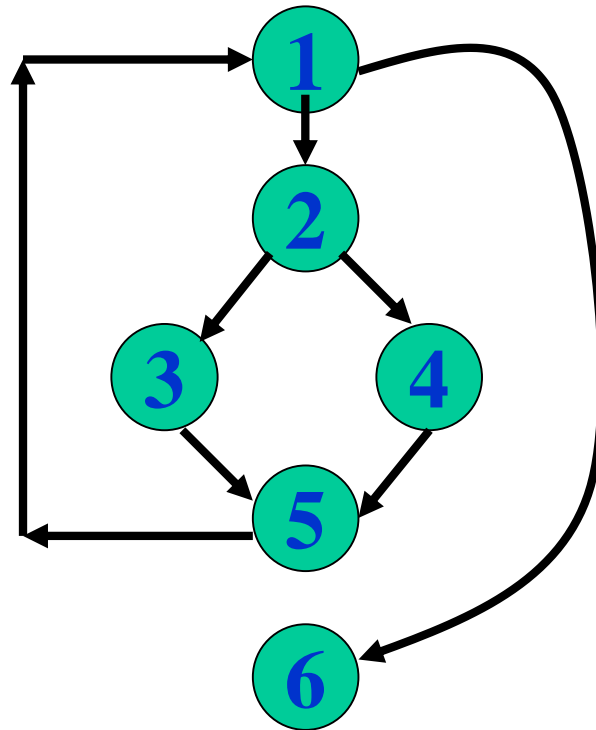
# How to Draw Control Flow Graph?

- Number all the statements of a program.
- Numbered statements:
  - Represent nodes of the control flow graph.
- An edge from one node to another node exists:
  - If execution of the statement representing the first node
    - Can result in transfer of control to the other node.

# Example

```
. int f1(int x,int y){  
. 1 while (x != y){  
. 2     if (x>y) then  
. 3         x=x-y;  
. 4     else y=y-x;  
. 5 }  
. 6 return x;      }
```

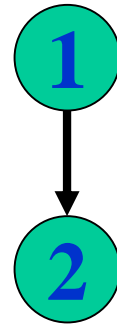
# Example Control Flow Graph



# How to draw Control flow graph?

. Sequence:

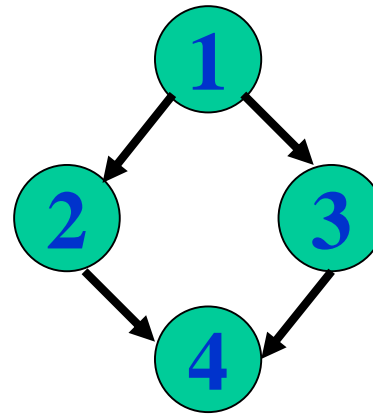
- 1 a=5;
- 2 b=a\*b-1;



# How to draw Control flow graph?

## . Selection:

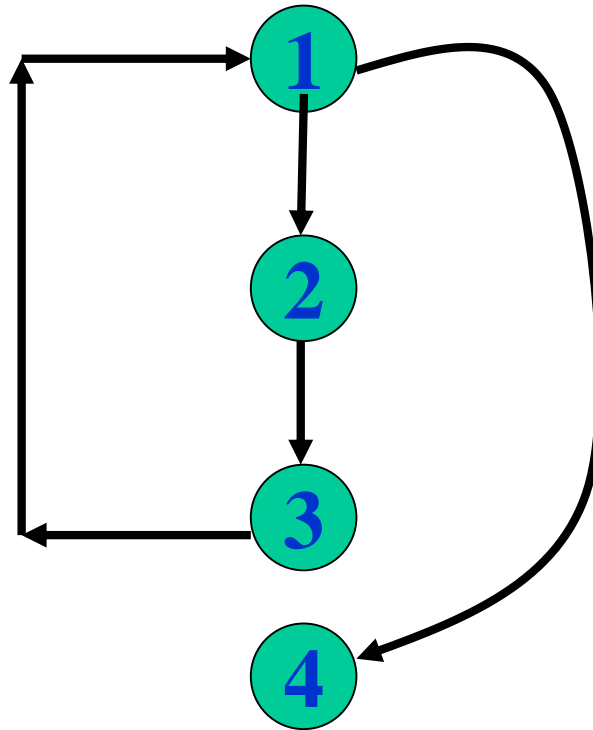
- 1 if(a>b) then
- 2           c=3;
- 3 else     c=5;
- 4 c=c\*c;



# How to draw Control flow graph?

## . Iteration:

- 1 while(a>b){
- 2       b=b\*a;
- 3       b=b-1;}
- 4 c=b+d;





# Path

- A path through a program:
  - A node and edge sequence from the starting node to a terminal node of the control flow graph.
  - There may be several terminal nodes for program.

# Linearly Independent Path

- Any path through the program:
  - Introducing at least one new edge:
    - That is not included in any other independent paths.

# Independent path

- It is straight forward:
  - To identify linearly independent paths of simple programs.
- For complicated programs:
  - It is not so easy to determine the number of independent paths.

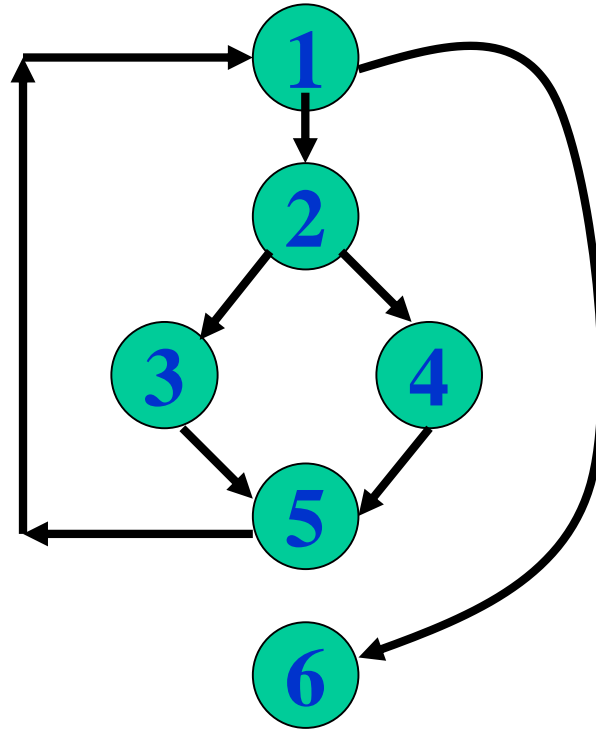
# McCabe's Cyclomatic Metric

- An upper bound:
  - For the number of linearly independent paths of a program
- Provides a practical way of determining:
  - The maximum number of linearly independent paths in a program.

# McCabe's Cyclomatic Metric

- Given a control flow graph  $G$ , cyclomatic complexity  $V(G)$ :
  - $V(G) = E - N + 2$ 
    - $N$  is the number of nodes in  $G$
    - $E$  is the number of edges in  $G$

# Example Control Flow Graph

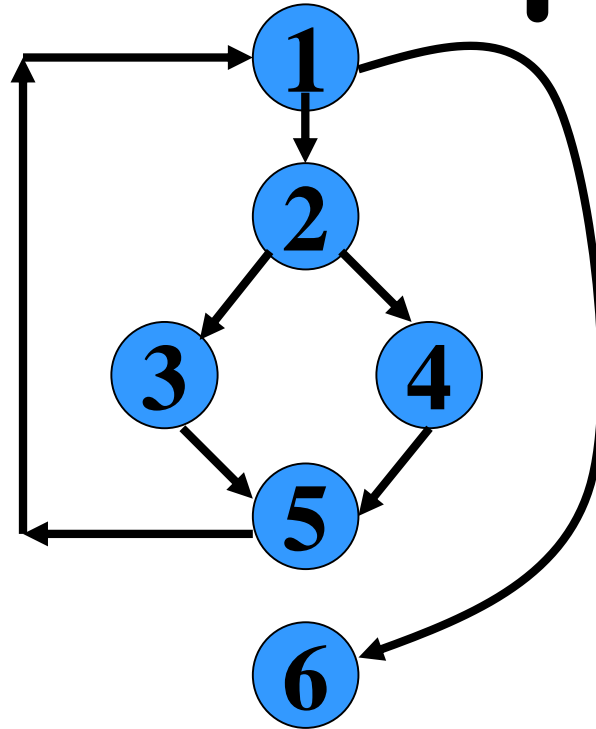


Cyclomatic  
complexity =  
 $7 - 6 + 2 = 3$ .

# Cyclomatic Complexity

- Another way of computing cyclomatic complexity:
  - inspect control flow graph
  - determine number of bounded areas in the graph
- $V(G) = \text{Total number of bounded areas} + 1$ 
  - Any region enclosed by a nodes and edge sequence.

# Example Control Flow Graph





# Example

- From a visual examination of the CFG:
  - the number of bounded areas is 2.
  - cyclomatic complexity =  $2+1=3$ .

# Cyclomatic complexity

- McCabe's metric provides:
  - A quantitative measure of testing difficulty and the ultimate reliability
- Intuitively,
  - Number of bounded areas increases with the number of decision nodes and loops.

# Cyclomatic Complexity

- The first method of computing  $V(G)$  is amenable to automation:
  - You can write a program which determines the number of nodes and edges of a graph
  - Applies the formula to find  $V(G)$ .

# Cyclomatic complexity

- The cyclomatic complexity of a program provides:
  - A lower bound on the number of test cases to be designed
  - To guarantee coverage of all linearly independent paths.

# Cyclomatic Complexity

- Defines the number of independent paths in a program.
- Provides a lower bound:
  - for the number of test cases for path coverage.

# Cyclomatic Complexity

- Knowing the number of test cases required:
  - Does not make it any easier to derive the test cases,
  - Only gives an indication of the minimum number of test cases required.

# Path Testing

- The tester proposes:
  - An initial set of test data using his experience and judgement.
- A dynamic program analyzer is used:
  - To indicate which parts of the program have been tested
  - The output of the dynamic analysis
    - used to guide the tester in selecting additional test cases.

# Derivation of Test Cases

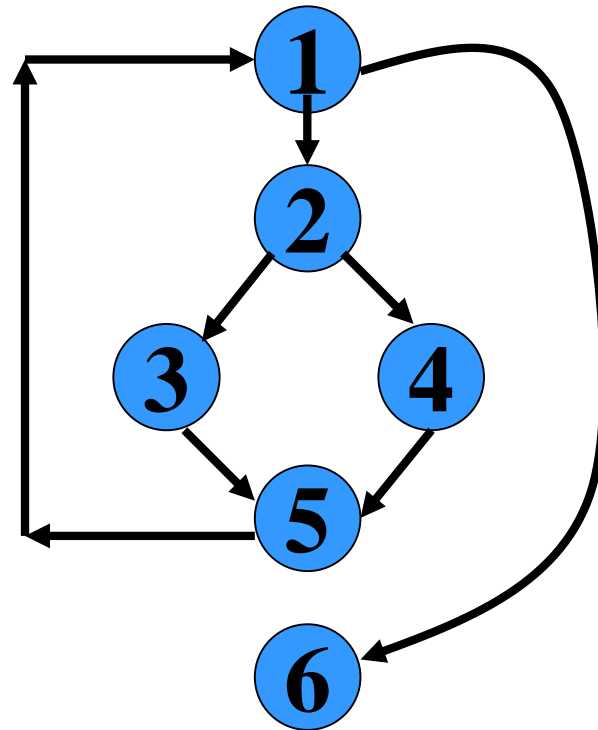
- Let us discuss the steps:
  - to derive path coverage-based test cases of a program.
- Draw control flow graph.
- Determine  $V(G)$ .
- Determine the set of linearly independent paths.
- Prepare test cases:
  - to force execution along each path.



# Example

```
. int f1(int x,int y){  
. 1 while (x != y){  
. 2     if (x>y) then  
. 3         x=x-y;  
. 4     else y=y-x;  
. 5 }  
. 6 return x;      }
```

# Example Control Flow Diagram



# Derivation of Test Cases

- Number of independent paths: 3
  - 1,6      test case (x=1, y=1)
  - 1,2,3,5,1,6      test case(x=1, y=2)
  - 1,2,4,5,1,6      test case(x=2, y=1)

# An interesting application of cyclomatic complexity

- Relationship exists between:
  - McCabe's metric
  - The number of errors existing in the code,
  - The time required to find and correct the errors.

# Cyclomatic Complexity

- Cyclomatic complexity of a program:
  - Also indicates the psychological complexity of a program.
  - Difficulty level of understanding the program.

# Cyclomatic Complexity

- From maintenance perspective,
  - limit cyclomatic complexity
    - of modules to some reasonable value.
  - Good software development organizations:
    - restrict cyclomatic complexity of functions to a maximum of ten or so.

# Summary

- Exhaustive testing of non-trivial systems is impractical:
  - We need to design an optimal set of test cases
    - Should expose as many errors as possible.
- If we select test cases randomly:
  - many of the selected test cases do not add to the significance of the test set.

# Summary

- There are two approaches to testing:
  - black-box testing and
  - white-box testing.
- Designing test cases for black box testing:
  - does not require any knowledge of how the functions have been designed and implemented.
  - Test cases can be designed by examining only SRS document.



# Summary

- White box testing:
  - requires knowledge about internals of the software.
  - Design and code is required.
- We have discussed a few white-box test strategies.
  - Statement coverage
  - branch coverage
  - condition coverage
  - path coverage

# Summary

- A stronger testing strategy:
  - provides more number of significant test cases than a weaker one.
  - Condition coverage is strongest among strategies we discussed.
- We discussed McCabe's Cyclomatic complexity metric:
  - provides an upper bound for linearly independent paths
  - correlates with understanding, testing, and debugging difficulty of a program.