SOFTWARE TESTING-I

Professor Sudip Misra

Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur
http://cse.iitkgp.ac.in/~smisra/

TESTING

- Aims to identify all defects in a program.
- Can reveal the **presence** of errors NOT their **absence**
- Completion of testing does not guarantee error free program
  - Due to large input domain.

Basic Concepts and Terminologies

- Error
  - A mistake committed by the development team during the development phases.
  - Mistake may be in requirement, design or coding phase.
  - Also referred as *fault, bug* or *defect*.
- Failure
  - Manifestation or symptom of an error.
  - Not all errors leads to failure.

Basic Concepts and Terminologies

- Test Case
  - Is a triplet [I, S, O]
  - I: is the data input to the system
  - S: is the state of the system at which data is input
  - O: is the expected output of the system
- Test Suite
  - Set of all test cases with which a given software product is tested.

Basic Concepts and Terminologies

- Verification:
  - The software should conform to its specification.
  - "*Are we building the product right ?*“
- Validation:
  - The software should do what the user really requires.
  - "*Are we building the right product ?*“

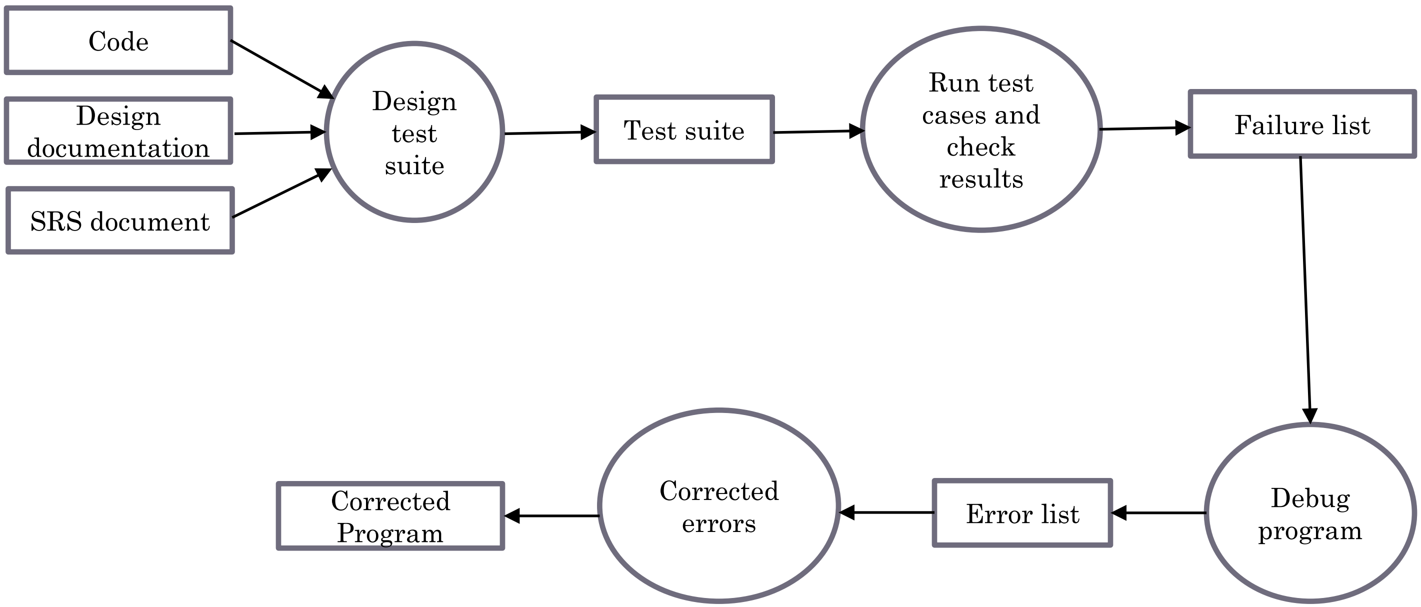
Verification and validation thus starts with **requirements reviews** and **continues through design and code reviews** to **product testing.** [1]

[1] https://www.sqa.org.uk/e-learning/SDPL03CD/page_16.htm

How do we test a system?

- Input test data to the system.
- Observe the output:
  - Check if the system behaved as expected.
- If the program does not behave as expected:
  - Note the conditions under which it failed.
  - Later debug and correct.

How do we test a system?



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
  - to uncover as many errors as possible.
- If test cases are selected randomly:
  - many test cases do not contribute to the significance of the test suite.
  - many test cases detect errors already detected by other test cases in the suite.

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 an example:
  - finding the maximum of two integers x and y.

Design of Test Cases

- If (x>y) max = x;
  else max = x;
- The code has a simple error:
- 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.

*"In contrast to **random test suite** we need carefully designed set of test cases such that, each test case helps detect different errors i.e.* ***minimal test suite.***"

Design of Test Cases

Black-box approach	White (glass)-box approach
Test cases are designed using only the functional specification of the software - *knowledge of internal structure not required.*	Test cases **requires thorough knowledge** about the internal structure of software.
For this reason, black-box testing is known as **functional testing**.	white-box testing is known as called **structural testing**

These approaches are complementary. A program has to be tested using the test cases designed by both the approaches.

Black-Box Testing

- Test cases are designed from an examination of the input/output values only (knowledge of code or deign not required)
- Two approach for design black-box test cases:
  - Equivalence class partitioning
  - Boundary value analysis

Equivalence class partitioning

- Domain of input values, to the program under test, is partitioned into a set of equivalence class.
- For every input data belonging to the same equivalence class, the program behave similarly.
  - Testing the program against any one input of given equivalence class suffice the test for that class.
- Eliminate the time required for exhaustive testing (testing for each input).
- Equivalence class test suite is a set of any one test cases from each equivalence classes.

Equivalence class partitioning

- Guidelines for designing the equivalence classes
  - If input data values is Range
    - Then one valid equivalence class & two invalid equivalence classes need to be defined. e.g. input data values [1, 10]
      - valid equivalence class: [1, 10]
      - invalid equivalence class: [-∞, 0] and [11, ∞].
  - If input data values is specific set
    - Then one valid equivalence class & one invalid equivalence classes need to be defined. e.g. input data values {A, B, C}
      - valid equivalence class: {A, B, C}
      - invalid equivalence class: U - {A, B, C}.

Equivalence class partitioning

- Equivalence class partitioning test suite for a function that reads a character string of size less than five characters and displays whether it is a palindrome.
  - **Step 1**: Identify the input domain:
    - In this case string, set of discrete members
  - **Step 2**: Equivalence class based on input
    - valid equivalence class:
      - set of string of length five or less.
    - invalid equivalence class:
      - set of string of length six or more.
  - **Step 3**: Equivalence class based on Input and output
    - valid equivalence class:
      - set of string of length five or less and palindrome.
      - set of string of length five or less and non-palindrome.
    - invalid equivalence class:
      - set of string of length six or more.
- Hence required test suite: { aba, abc, abcdef }

Practice Problem: Equivalence Class

- Validity of Date of Journey
- Interface
  - Boolean isValidDOJ(date current_date, date input_date);
- Specification
  - The input_date should not be a past date.
  - The input_date should not exceed 90 days from current_days.
- Write all possible equivalence classes for the specification

Practice Problem: Equivalence Class

- Counting Characters in a String
- Interface
  - void CharCount(int VoCount, int InCount);
- Specification
  - the procedure keeps on reading Input from the keyboard; it stops when a non-alphabet (English) character or some upper value Max-Size has been reached
  - if the input is an alphabet character, then the counter InCount is incremented; if it is a vowel, then the counter VoCount is incremented
  - both counters are input and output parameters
  - the invariant VoCount <= InCount holds
- Write all possible equivalence classes for the specification

Solution

- Equivalence classes for *InCount*
  1. $InCount < 0$ [invalid]
  2. $0 \le InCount < \text{MaxSize}$ [valid]
  3. $InCount = \text{MaxSize}$ [valid]
  4. $InCount > \text{MaxSize}$ [invalid]
- Equivalence class for *VoCount*
  1. $VoCount < 0$ [invalid]
  2. $0 \le VoCount \le InCount$ [valid]
  3. $VoCount > InCount$ [Invalid]
- Equivalence classes for keyboard *Input*
  1. $Input < \text{A}$
  2. $Input < a$
  3. $Input > Z$
  4. $Input > z$

  [invalid] non-English alphabets

  5. $\text{A} \le Input \le Z$
  6. $\text{a} \le Input \le z$
  7. $Input = \text{A}$
  8. $Input = E$
  9. $Input = I$
  10. $Input = O$
  11. $Input = U$
  12. $Input = a$
  13. $Input = e$
  14. $Input = i$
  15. $Input = o$
  16. $Input = u$

  [valid]

Boundary Value Analysis

- programming error frequently occurs at the boundaries of different equivalence classes of inputs.
  - For example, programmers may improperly use < instead of <=, or conversely <= .
  - The requirements are generally vague at the boundaries. e.g. different **Tax Rate** on different **Income Slab**.
  - Confusion in using loops and conditions checks (related to coding).
- Boundary value analysis leads to selection of test cases at the boundaries of the different equivalence classes.

Boundary Value Analysis

- For a function that computes the tax based on the income.

Income Slab	Tax Rate
Income up to Rs. 3,00,000	No Tax
Income from Rs. 3,00,000 – Rs. 5,00,000	5%
Income from Rs. 5,00,000 – 10,00,000	20%
Income more than Rs. 10,00,000	30%

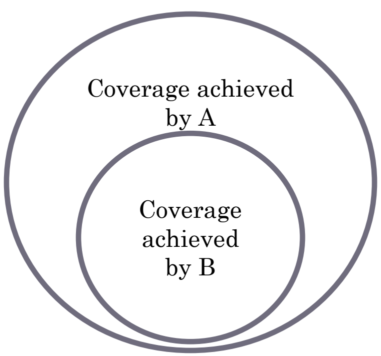
- The boundary value test suite is {299999, 300000, 499999, 500000, 999999, 1000000}

White-box Testing

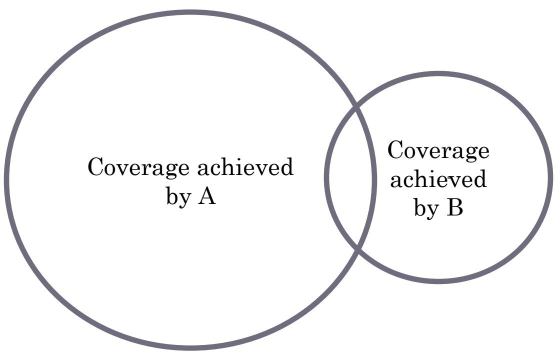
- Test cases are designed based on an analysis of the internal structure of the component or system.
- White-box testing strategy can be
  - **Coverage based testing**: Attempts to execute (i.e. cover) certain elements of the program.
    - Statement Coverage
    - Branch Coverage
    - Path-Coverage
    - condition coverage
  - **Fault-based testing**: Attempts to enhanced the existing test suite to detect certain types of faults.
    - Mutation Testing.

Stronger, Weaker and Complementary Testing

- Testing strategy A is said to be **stronger** than testing strategy B if A covers all type of program elements covers by B. (B is **weaker** than A)
- If neither A is stronger than B nor B is stronger than A. A and B said to be **complementary**.
- If a stranger testing has been performed, then a weaker testing **need not** be carried out.



A is stronger than B



A and B are complementary

Statement Coverage

- Aims to design test cases so as to execute every statement in a program at least once.
- The principal idea:
  - unless a statement is executed,
  - we have no way of knowing if an error exists in that statement.
- Observing that a statement behaves properly for one input value:
  - no guarantee that it will behave correctly for all input values.

Example: Euclid's GCD Algorithm

```
int computeGCD(int x, int y)
{
    while (x != y)
    {
      if (x>y) then
                        x=x-y;
             else y=y-x;
    }
    return x;
}
```

Example: Euclid's GCD Algorithm

```
int computeGCD(int x, int y)
{
    while (x != y)
    {
      if (x>y) then
                  x=x-y;
            else y=y-x;
     }
     return x;
 }
```

- 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.
- Each edge of program's control flow graph (CFG) is traversed at least once --- **edge testing.**

BRANCH COVERAGE

```
int f1(int x,int y)
{
   while (x != y)
   {
      if (x>y) then
         x=x-y;
      else y=y-x;
   }
   return x;
}
```

BRANCH COVERAGE

```
int f1(int x,int y)
{
    while (x != y)
    {
        if (x>y) then
            x=x-y;
        else y=y-x;
    }
    return x;
}
```

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

Branch Coverage

- Branch Coverage-based testing is **stronger** than statement coverage-based testing.
  - Branch coverage ensures statement coverage, but not vice versa.

Condition Coverage

- Test cases are designed such that:
  - each component of a composite conditional expression
    - given both true and false values.
- 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.
- Consider a Boolean expression having n components:
  - for condition coverage we require 2^n test cases.
- practical only if **n** (the number of component conditions) is small.
  - Number of test cases increases exponentially.

Condition Coverage

- Condition testing
  - stronger testing than branch testing.

Path Coverage

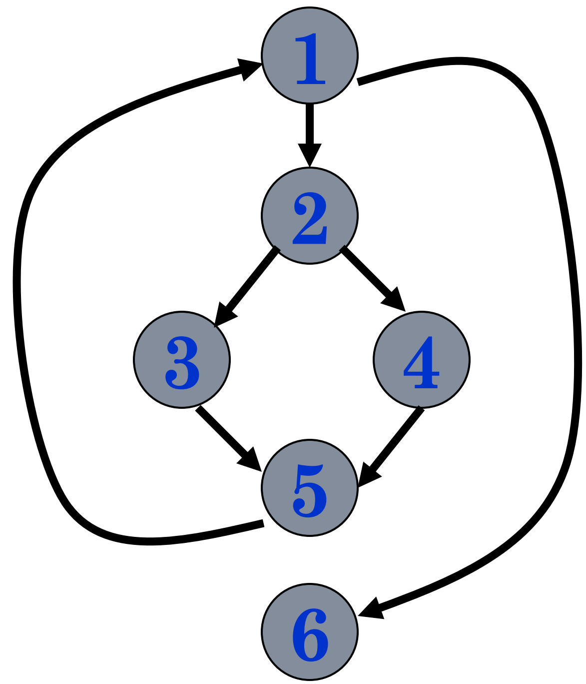
- Design test cases such that:
  - All **linearly independent paths** in the program are executed at least once.
- To understand the path coverage-based testing:
  - we need to learn how to draw control flow graph (CFG) of a program.

Control flow graph (CFG)

- A control flow graph (CFG) describes:
  - the sequence in which different instructions of a program get executed.
  - the way control flows through the program.
- Formally CFG is Directed Graph $G(N, E)$
  - Each node $n \in N$ corresponds to a unique program statement.
  - An edge $(n_i, n_j) \in E$ if control can transfer from statement n_i to statement n_j.

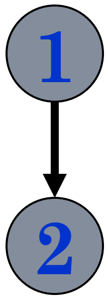
Example

```
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;
}
```



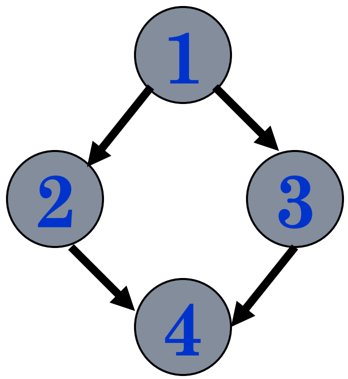
How to draw Control flow graph?

- Sequence:
  1. a=5;
  2. b=a*b-1;



How to draw Control flow graph?

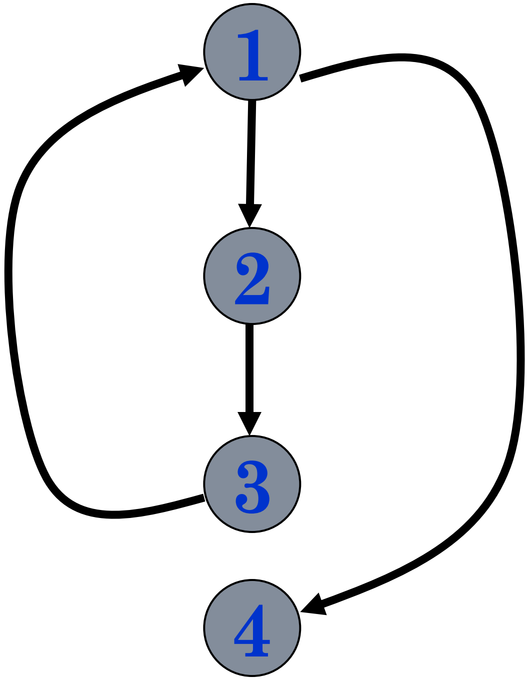
- Selection:
  1. if(a>b) then
  2. c=3;
  3. else c=5;
  4. c=c*c;



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.
- There can be an *infinite number of paths* e.g. 12314, 12312314, 12312312314, ….. .
  - Coverage of all paths requires infinite many test cases.
  - Path coverage-based testing subset of paths – *Linearly independent paths* (basic paths).

Linearly Independent path

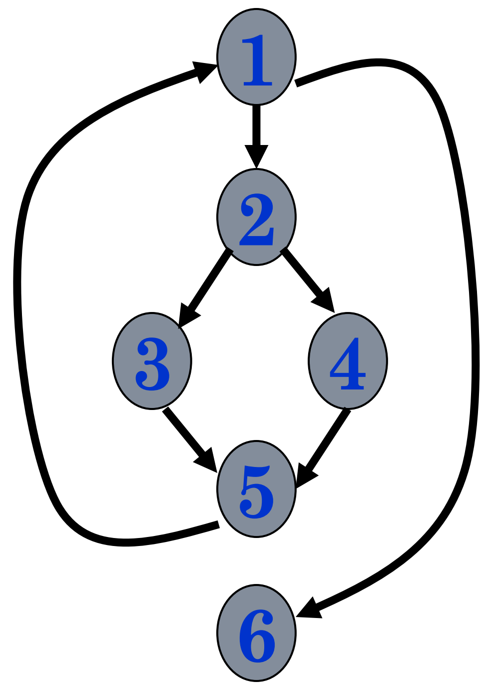
- Any path through the program:
  - introducing at least one new node:
    - that is not included in any other linear independent paths.
- Collection of such path is the set of linear independent paths.
  - For any given path in such set, its sub-path cannot be in that set.
- It is straight forward:
  - to identify linearly independent paths of simple programs.
- For complicated programs:
  - it is not so easy to determine the **number of linear 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 [Method-1]

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

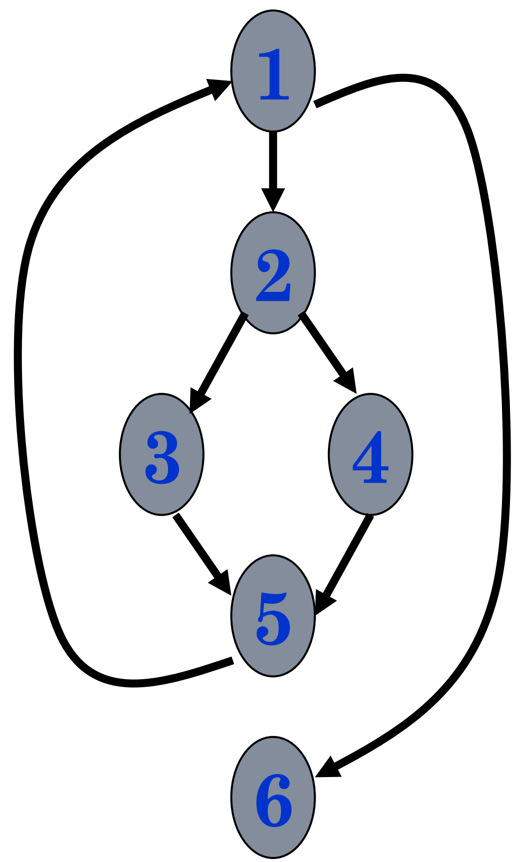


McCabe's cyclomatic metric [Method-2]

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

McCabe's cyclomatic metric [Method-2]

- From a visual examination of the CFG:
  - the number of bounded areas is 2.
  - cyclomatic complexity = 2+1=3.
- This method would not work for non planer graph.



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).
- 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

- 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

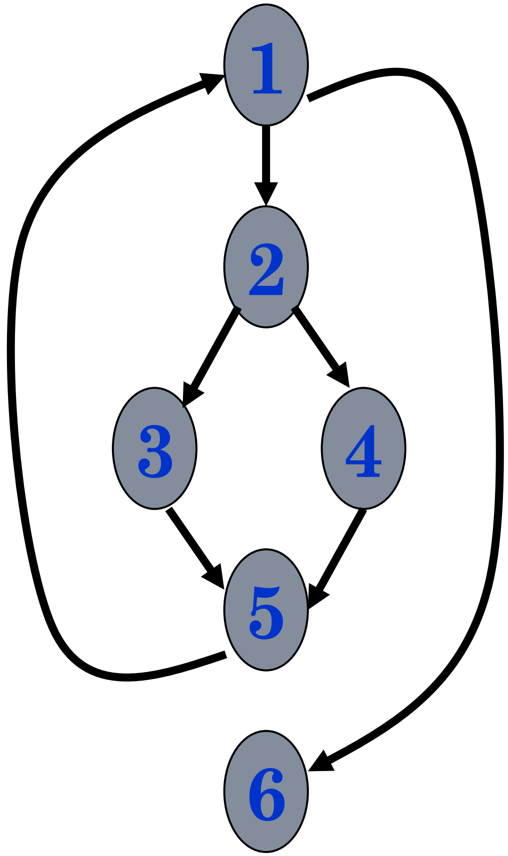
- The tester proposes:
  - an initial set of test data using his experience and judgment.
- A dynamic program analyzer is used:
  - to indicate which parts of the program have been tested
  - the output of the dynamic analysis
    - used to guide the tester in selecting additional test cases.

Derivation of Test Cases

- 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;
}
```



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)

Other Applications of McCabe's cyclomatic metric

- Estimation of structural complexity of code.
  - McCabe's metric is based on code structure
  - Intuitively, it correlates the psychological complexity or difficulty level of understanding the program.
  - Good software development organizations:
    - restrict cyclomatic complexity of functions to a maximum of **ten** or so.
- Estimation of testing efforts.
  - McCabe's metric is a measures of maximum number of basic paths.
  - Implies, minimum number of **test case required** for path coverage.
  - Hence in turn estimate the test efforts.
  - restrict cyclomatic complexity of functions to **seven** to reduce testing effort.

Other Applications of McCabe's cyclomatic metric

- Estimation of program reliability
  - Study indicates
    - the number of errors latent in the code after testing has direct relation with McCabe's metric.
    - The relationship possibly due to McCabe's metric is based on structural complexity of the code.
    - Usually, larger the structural complexity, the more difficult to test and debug.

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.

Summary

- 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.

SUMMARY

- 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.

Summary

- 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.

Summary

- 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.

THANK YOU