Testing and Debugging

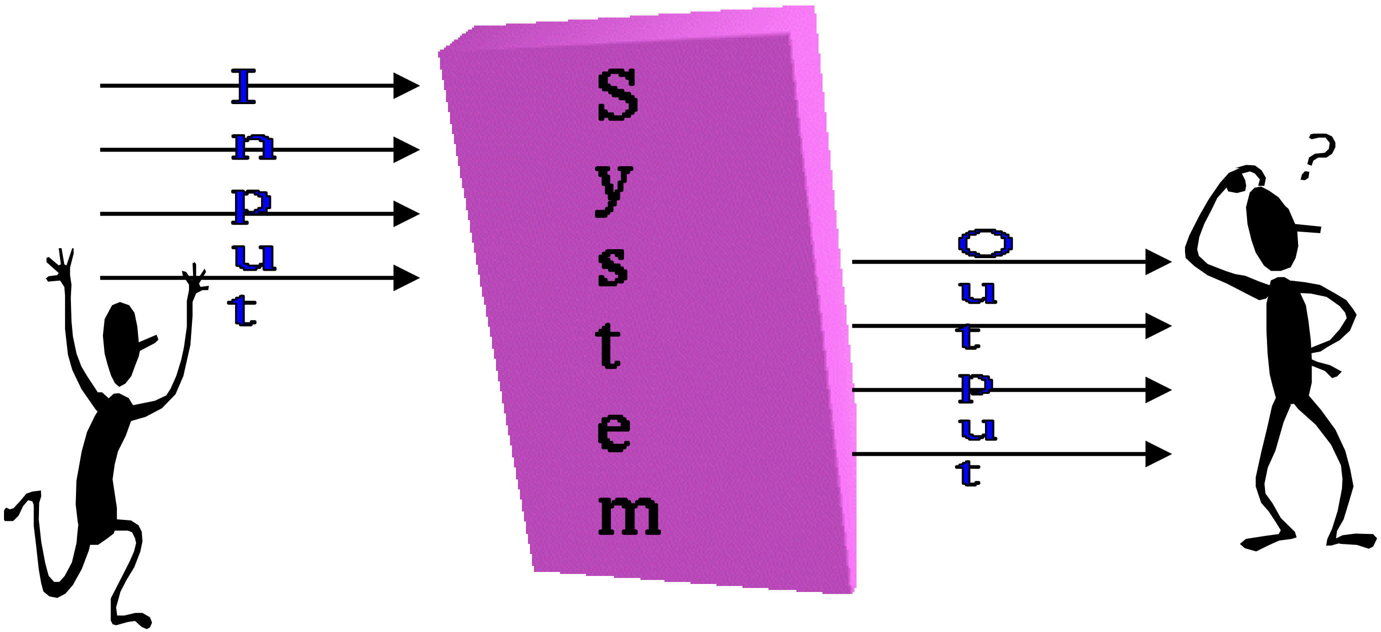
Organization of this lecture

- Important concepts in program testing
- Black-box testing:
  - equivalence partitioning
  - boundary value analysis
- White-box testing
- Debugging
- Unit, Integration, and System testing
- Summary

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 system?



How do you test a system?

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

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 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 <u>test suite</u>

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 an example for finding 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 <u>structural testing</u>.

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

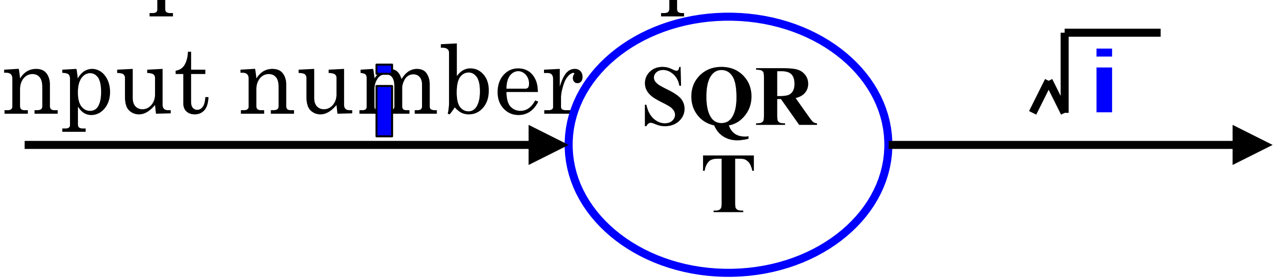
- If the input data to the program is specified by a range of values:
  - e.g. numbers between 1 to 5000.
  - one valid and two invalid equivalence classes are defined.

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

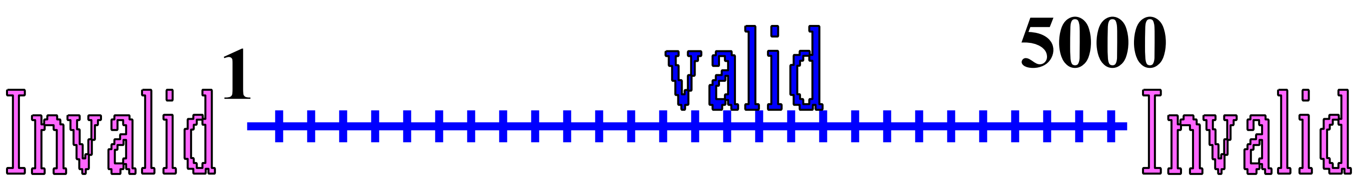


EXAMPLE (CONT.)

- 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}.



Problem

- Design the Equivalence Class Test Suite for a program that reads two integer pairs (m_1,c_1) and (m_2,c_2) defining two straight lines of the form $y=mx+c$. The program computes the intersection of the two lines.

Boundary Value Analysis

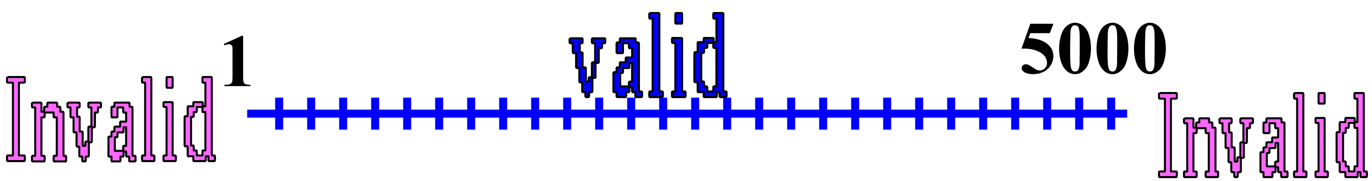
- Some typical programming errors occur:
  - at boundaries of equivalence classes
  - might be purely due to psychological factors.
- Programmers often fail to see:
  - special processing required at the boundaries of equivalence classes.

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}.



White-box Testing

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

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

Statement Coverage

- Statement coverage methodology:
  - design test cases so that every statement in a 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

- Based on the observation:
  - an error in a program can not be discovered:
    - unless the part of the program containing the error is executed.

Example

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

Euclid's GCD Algorithm

Euclid's GCD computation algorithm

- By choosing the test set {(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:
  - discovers at least as many errors as a weaker testing
  - contains at least as many significant test cases as a weaker test.

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=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.

CONDITION COVERAGE

- Consider a boolean expression having n components:
  - for condition coverage we require 2^n test cases.

CONDITION COVERAGE

- 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.

LINEARLY INDEPENDENT PATHS

- 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.

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.

How to draw Control flow graph?

- Number all the statements of a program.
- Numbered statements:
  - represent nodes of the control flow graph.

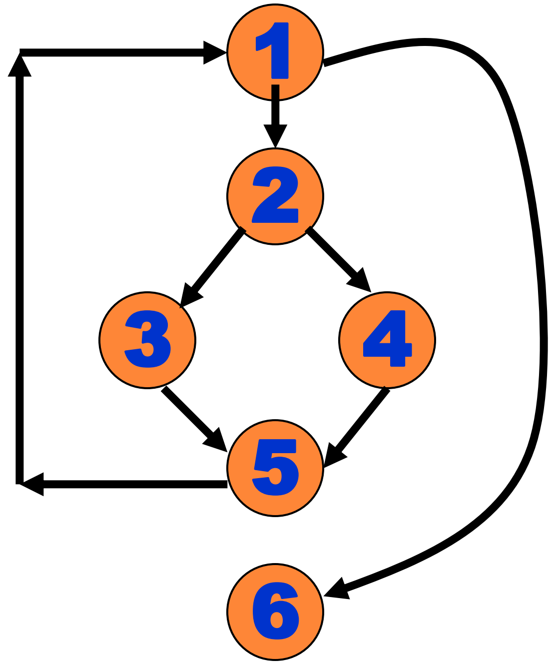
How to draw Control flow graph?

- 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

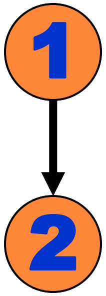
- int f1(int x,int y){
- 1 while (x != y){
- 2   if (x>y) then
- 3   x=x-y;
- 4   else y=y-x;
- 5 }
- 6 return x;   }

Example Control Flow Graph



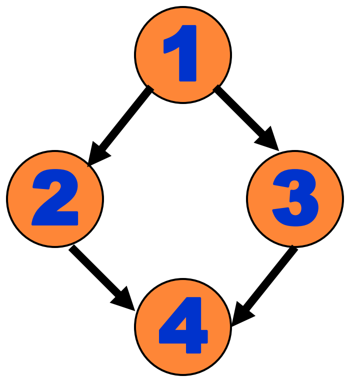
How to draw Control flow graph?

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



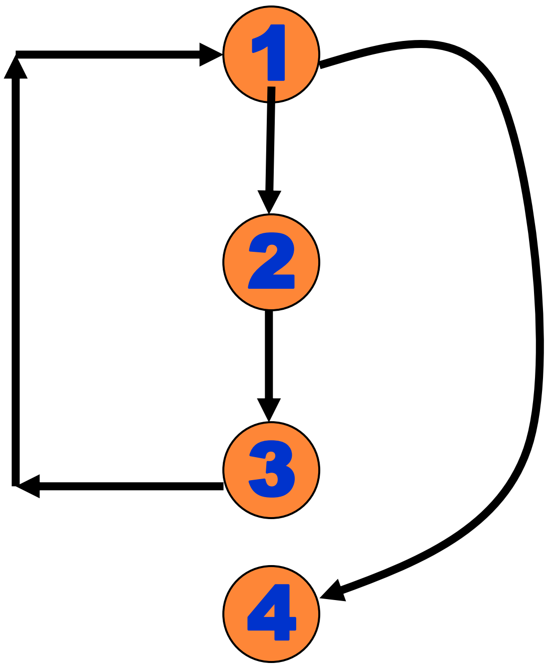
How to draw Control flow graph?

- 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.

INDEPENDENT PATH

- Any path through the program:
  - introducing at least one new node:
    - 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.

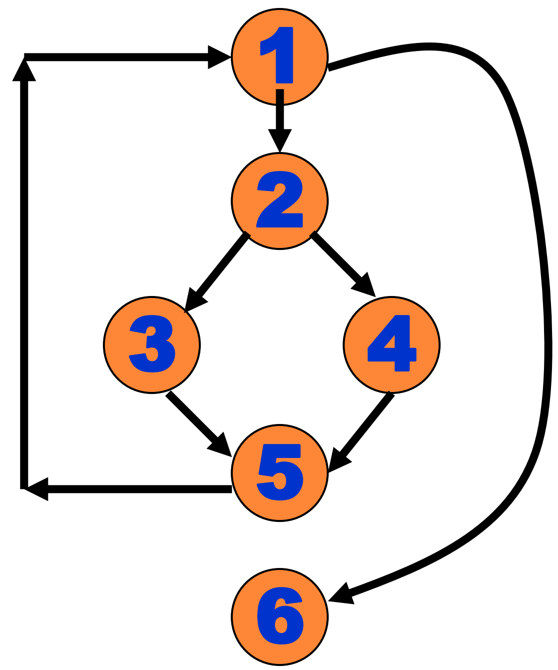
McCabe's cyclomatic metric

- 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



Example

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

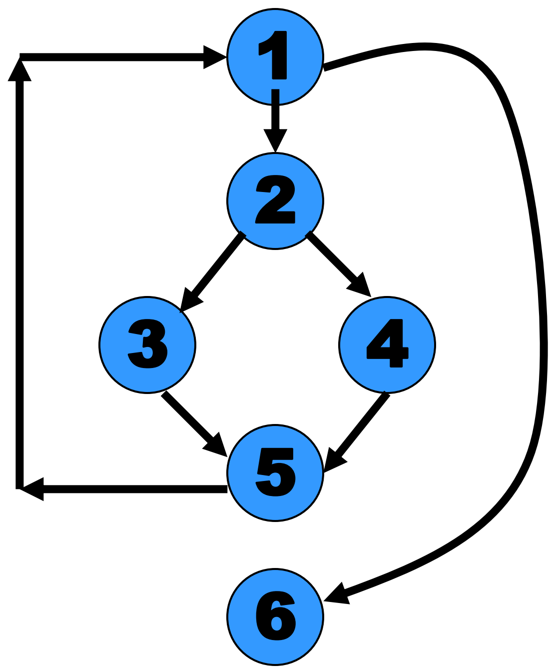
Cyclomatic complexity

- Another way of computing cyclomatic complexity:
  - inspect control flow graph
  - determine number of bounded areas in the graph
- V(G) = Total number of bounded areas + 1

BOUNDED AREA

- 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

- 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.

Path testing

- 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.

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.

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.

Data Flow-Based Testing

- Selects test paths of a program:
  - according to the locations of
    - definitions and uses of different variables in a program.

Data Flow-Based Testing

- For a statement numbered S,
  - DEF(S) = {X/statement S contains a definition of X}
  - USES(S)= {X/statement S contains a use of X}
  - Example: 1: a=b; DEF(1)={a}, USES(1)={b}.
  - Example: 2: a=a+b; DEF(2)={a}, USES(2)={a,b}.

Data Flow-Based Testing

- A variable X is said to be live at statement S1, if
  - X is defined at a statement S:
  - there exists a path from S to S1 not containing any definition of X.

DU Chain Example

```
X(){
 a=5; /* Defines variable a */
 While(C1) {
    if (C2)
          b=a*a;   /*Uses variable a */
    else a=a-1; /* Defines variable a */
    }
  print(a); } /*Uses variable a */
```

Definition-use chain (DU chain)

- [X,S,S1],
  - S and S1 are statement numbers,
  - X in DEF(S)
  - X in USES(S1), and
  - the definition of X in the statement S is live at statement S1.

Data Flow-Based Testing

- One simple data flow testing strategy:
  - every DU chain in a program be covered at least once.

Data Flow-Based Testing

- Data flow testing strategies:
  - useful for selecting test paths of a program containing nested if and loop statements

Definition-Use based Testing

- A definition-use (DU) chain of a variable X is of form [X,S,S1], where

  - X Є DEF(S)
  - X Є USES(S1)

  And the definition of X in the statement S is live at statement S1.

- ***DU*** or ***data flow testing strategy*** is to require that every DU chain be covered at least once.

Definition-Use based Testing

```
int gcd(int a, int b){
        int c = a;
        int d = b;
        if(c == 0)
                return d;
        while(d != 0){
                if(c > d)
                        c = c - d;
                else
                        d = d - c;
        }
        return c;
}
```

DU Chains

1. [d, d=b, return d]
2. [d, d=b, while(d!=0)]
3. [d, d=b, if(c>d)]
4. [d, d=b, c=c-d]
5. [d, d=b, d=d-c]
6. [d, d=d-c, while(d!=0)]
7. [d, d=d-c, if(c>d)]
8. [d, d=d-c, c=c-d]
9. [d, d=d-c, d=d-c]

Mutation Testing

- The software is first tested:
  - using an initial testing method based on white-box strategies we already discussed.
- After the initial testing is complete,
  - mutation testing is taken up.
- The idea behind mutation testing:
  - make a few arbitrary small changes to a program at a time.

Mutation Testing

- Each time the program is changed,
  - it is called a mutated program
  - the change is called a mutant.

MUTATION TESTING

- A mutated program:
  - tested against the full test suite of the program.
- If there exists at least one test case in the test suite for which:
  - a mutant gives an incorrect result,
  - then the mutant is said to be dead.

Mutation Testing

- If a mutant remains alive:
  - even after all test cases have been exhausted,
  - the test suite is enhanced to kill the mutant.
- The process of generation and killing of mutants:
  - can be automated by predefining a set of primitive changes that can be applied to the program.

Mutation Testing

- The primitive changes can be:
  - altering an arithmetic operator,
  - changing the value of a constant,
  - changing a data type, etc.

MUTATION TESTING

- A major disadvantage of mutation testing:
  - computationally very expensive,
  - a large number of possible mutants can be generated.

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.

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.

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.

Dynamic Analysis Tools

- Code Instrumentation
  - Achieved by inserting additional statements to the code
  - To collect the execution trace of the program.

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.