Module 44: Software Engineering

Software Testing

Instructors: Abir Das and Jibesh Patra

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

{*abir, jibesh*}*@cse.iitkgp.ac.in*

Slides taken from NPTEL course on Object-Oriented Analysis & Design

by **Prof. Partha Pratim Das**

Table of Contents

Why Test?

- **Ariane 5 Flight 501**

  - Un-manned satellite-launching rocket in 1996
  - Self-destructed 37 seconds after launch
  - Conversion from 64-bit floating point to 16-bit signed integer value had caused an exception (re-used from Ariane 4)
    - The floating point number was larger than 32767
    - Efficiency considerations had led to the disabling of the exception handler

Source: 11 of the most costly software errors in history

Why Test?

- **NASA's Mars Climate Orbiter**
  - Mission to Mars in 1998, $125 Million
  - Lost in space
  - Simple conversion from English units to metric failed
- **EDS Child Support System in 2004**
  - Overpay 1.9 million people
  - Underpay another 700,000
  - US $7 billion in uncollected child support payments
  - Backlog of 239,000 cases
  - 36,000 new cases "stuck" in the system
  - Cost the UK taxpayers over US $1 billion
  - Incompatible software integration
- **Heathrow Terminal 5 Opening**
  - Baggage handling tested for 12K test pieces of luggage
  - Missed to test for *removal of baggage*. In 10 days some 42,000 bags failed to travel with their owners, and over 500 flights were cancelled

Why Test?

- **The Morris Worm**
  - Developed by a Cornell University student for a harmless experiment
  - Spread wildly and crashing thousands of computers in 1988 because of a coding error
  - It was the first widespread worm attack on the fledgling Internet
  - The graduate student, Robert Tappan Morris, was convicted of a criminal hacking offense and fined $10,000
  - Costs for cleaning up the mess may have gone as high as $100 Million
  - Morris, who co-founded the startup incubator Y Combinator, is now a professor at the Massachusetts Institute of Technology
  - A disk with the worm's source code is now housed at the University of Boston

Why Test?

- **Boeing Crash**
  - On March 10, 2019, Ethiopian Airlines Flight 302 crashed just minutes after takeoff. All 157 people on board the flight died
  - On October of 2018, Lion Air Flight 610 also crashed minutes after taking off
  - Both flights involved Boeing's 737 MAX jet
  - The software overpowered all other flight functions trying to mediate the nose lift
  - Many pilots did not know this system existed - they were not re-trained on 737 MAX Jet

Source: Boeing Software Scandal Highlights Need for Full Lifecycle Testing

- **Airbus Crash**
  - On May 9, 2015, the Airbus A400M crashed near Seville after a failed emergency landing during its first flight
  - Electronic Control Units (ECU) on board malfunctioned

Source: Airbus A400M plane crash linked to software fault

Testing a Program

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

What's So Hard About Testing?

- Consider `int proc1(int x, int y)`
- Assuming a 64 bit computer
  - Input space $= 2^{128}$
- Assuming it takes 10secs to key-in an integer pair
  - It would take about a billion years to enter all possible values!
  - Automatic testing has its own problems!

Testing Facts

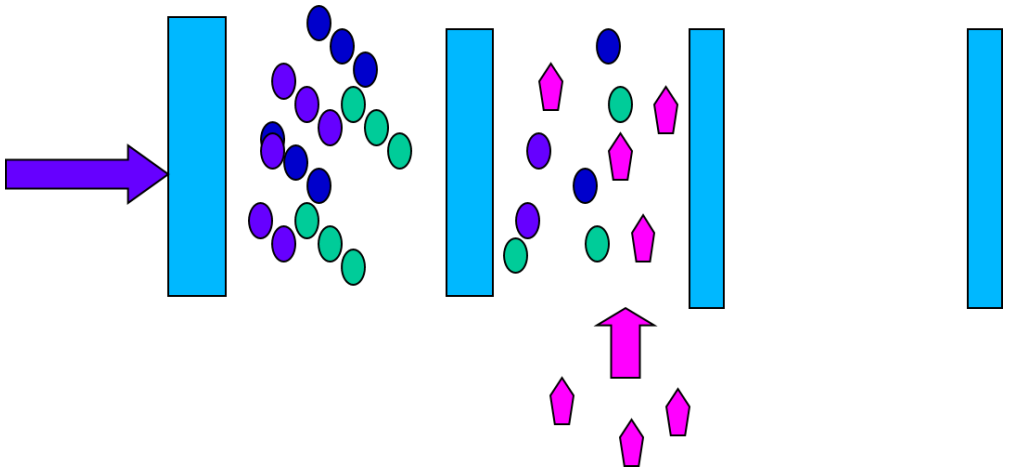
- Consumes largest effort among all phases
  - Largest manpower among all other development roles
  - Implies more job opportunities
- About 50% development effort
  - But 10% of development time?
  - How?
- Testing is getting more complex and sophisticated every year
  - Larger and more complex programs
  - Newer programming paradigms

Overview of Testing

- 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 (also defect or bug)
  - Mere presence of an error may not lead to a failure

Pesticide Effect

- Errors that escape a fault detection technique:
  - Can not be detected by further applications of that technique



- Assume we use 4 fault detection techniques and 1000 bugs:
  - Each detects only 70% bugs
  - How many bugs would remain?
  - $1000 * (0.3)^4 = 81$ bugs

Fault Model

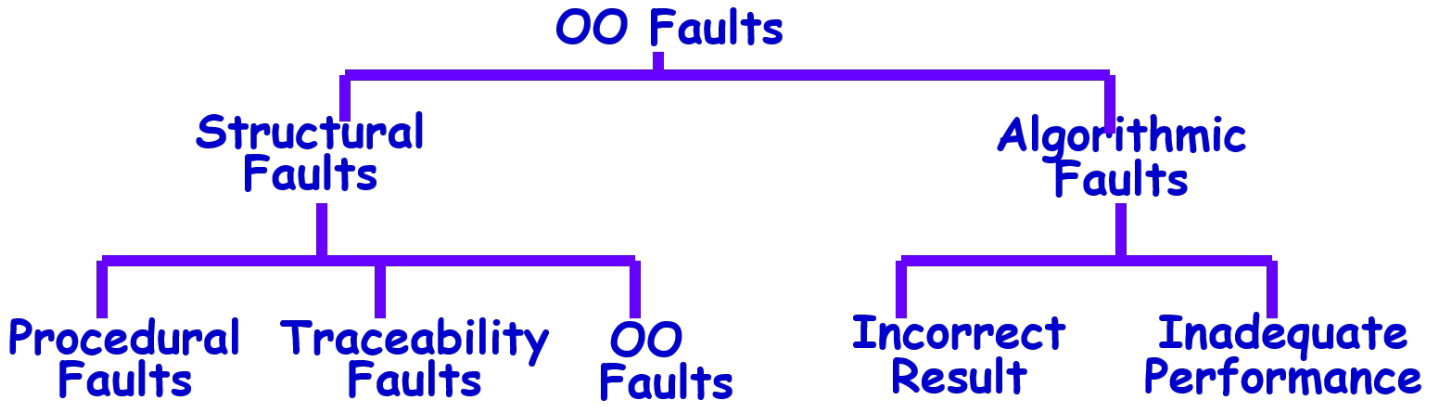
- Types of faults possible in a program
- Some types can be ruled out
  - Concurrency related-problems in a sequential program
  - Consider a singleton in multi-thread

```
#include <iostream>
using namespace std;
class Printer { /* THIS IS A SINGLETON PRINTER -- ONLY ONE INSTANCE */
    bool blackAndWhite_, bothSided_;
    Printer(bool bw = false, bool bs = false) : blackAndWhite_(bw), bothSided_(bs)
    { cout << "Printer constructed" << endl; }
    static Printer *myPrinter_; // Pointer to the Singleton Printer
public:
    ~Printer() { cout << "Printer destructed" << endl; }
    static const Printer& printer(bool bw = false, bool bs = false) {
        if (!myPrinter_) // What happens on multi-thread?
            myPrinter_ = new Printer(bw, bs);
        return *myPrinter_;
    }
    void print(int nP) const { cout << "Printing " << nP << " pages" << endl; }
};
Printer *Printer::myPrinter_ = 0;

int main() {
    Printer::printer().print(10);
    Printer::printer().print(20);
    delete &Printer::printer();
    return 0;
}
```

Fault Model

- Fault Model of an OO Program



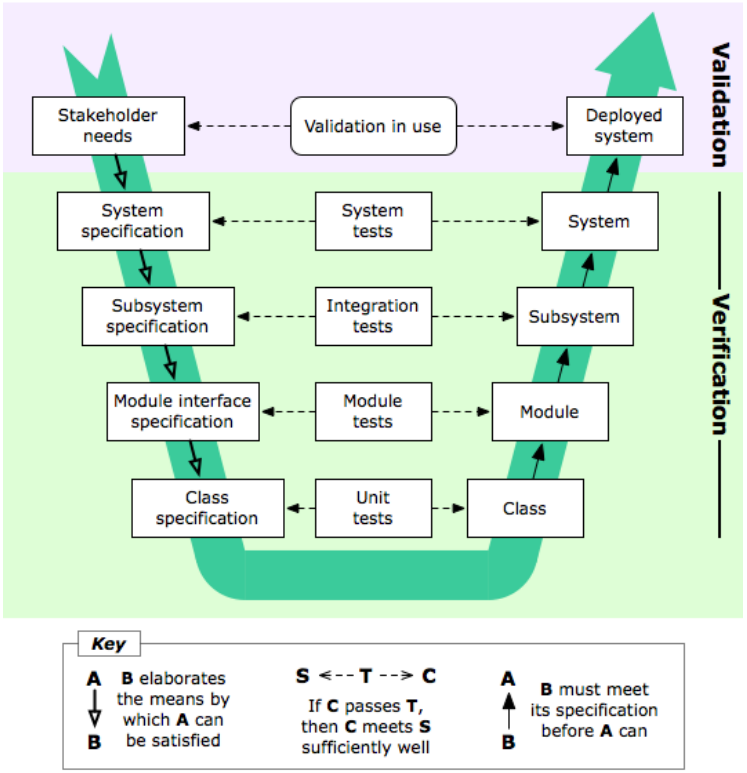
- Hardware Fault-Model
  - Simple:
    - Stuck-at 0
    - Stuck-at 1
    - Open circuit
    - Short circuit
  - Simple ways to test the presence of each
  - Hardware testing is fault-based testing

Test Cases and Test Suites

- Each test case typically tries to establish correct working of some functionality:
  - Executes (covers) some program elements
  - For restricted types of faults, fault-based testing exists
- Test a software using a set of carefully designed test cases:
  - The set of all test cases is called the test suite
- 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 (called *Golden*)

Verification versus Validation

- **Verification** is the process of determining
  - Whether output of one phase conforms to its previous phase
  - If we are building the system correctly
  - *Verification is concerned with phase containment of errors*
- **Validation** is the process of determining
  - Whether a fully developed system conforms to its SRS document
  - If we are building the correct system
  - *Whereas the aim of validation is that the final product be error free*



Source: Software Testing

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
- 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
- Testing a system using a large number of randomly selected test cases
  - Does not mean that many errors in the system will be uncovered

Design of Test Cases

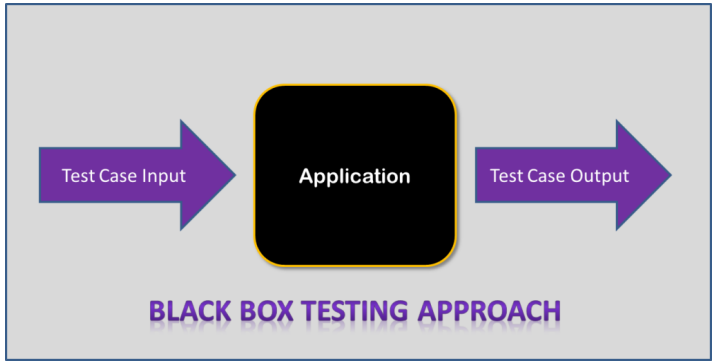
- Consider following example
  - Find the maximum of two integers x and y
- The code has a simple programming error

```
if (x>y)
    max = x;
else
    max = x;
```

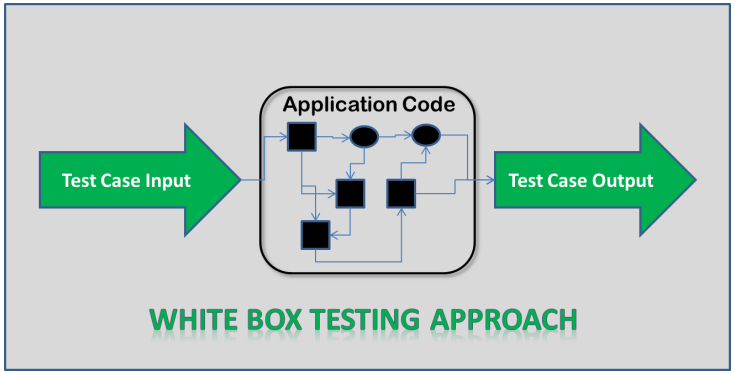
- Test suite {(x=3,y=2); (x=2,y=3)} can detect the error
- A larger test suite {(x=3,y=2); (x=4,y=3); (x=5,y=1)} does not detect the error

Design of Test Cases

- Systematic approaches are required to design an optimal test suite
  - Each test case in the suite should detect different errors
- There are essentially three main approaches to design test cases
  - **Black-box testing** (*Zero Knowledge*)



  - **White-box testing** (*Full Knowledge*)



  - **Grey-box testing** (*Some Knowledge*)

Why Both BB and WB Testing?

Black Box Testing

- Impossible to write a test case for every possible set of inputs and outputs
- Some code parts may not be reachable
- Does not tell if extra functionality has been implemented.

White Box Testing

- Does not address the question of whether or not a program matches the specification
- Does not tell you if all of the functionality has been implemented
- Does not discover missing program logic

Black-Box Testing

- Black-box testing is a method of software testing that examines the functionality of an application without peering into its internal structures or workings
- This method of test can be applied virtually to every level of software testing
  - unit
  - integration
  - system and
  - acceptance
- 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* or *specification-based testing*

White-Box Testing

- White-box testing is a method of software testing that tests internal structures or workings of an application, as opposed to its functionality
- In white-box testing an internal perspective of the system, as well as programming skills, are used to design test cases
- Designing white-box test cases
  - Requires knowledge about the *internal structure* of software
- White-box testing is also called *structural testing*

Grey-Box Testing

- Grey-box testing is a combination of white-box testing and black-box testing
- The aim of this testing is to search for the defects if any due to improper structure or improper usage of applications

Black-Box Testing Strategies

There are several significant approaches to design black box test cases including

- Equivalence class partitioning
- Boundary value analysis
- State Transition Testing
- Decision Table Testing
- Graph-Based Testing
- Error Guessing Technique

Other approaches include:

- Fuzzing Technique
- All Pair Testing
- Orthogonal Array Testing
- and so on

Source: Black Box Testing Techniques with Examples

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

Black-Box Testing: 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
  - If the input data is specified by a range of values
    - For example, numbers between 1 to 5000
    - One valid and two invalid equivalence classes are defined
  - If input is an enumerated set of values
    - For example, { a, b, c }
    - One equivalence class for valid input values
    - Another equivalence class for invalid input values should be defined
  - A program reads an input value in the range of 1 and 5000
    - Computes the square root of the input number
    - One valid and two invalid equivalence classes are defined – The set of negative integers, Set of

Black-Box Testing: Equivalence Class Partitioning

- Max program reads two non-negative integers and spits the larger one

Equivalence Class	Condition	Test Case
EC 1 (Greater)	$x > y$	(5, 2)
EC 2 (Smaller)	$x < y$	(3, 7)
EC 3 (Equal)	$x = y$	(4, 4)

- QES program reads (a, b, c) and solves: $ax^2 + bx + c = 0$

Equivalence Class	Condition	Test Case
Infinite roots	$a = b = c = 0$	(0,0,0)
No root	$a = b = 0; c \neq 0$	(0,0,2)
Single root	$a = 0, b \neq 0$	(0,2,-4)
Repeated roots	$a \neq 0; b * b - 4 * a * c = 0$	(4,4,1)
Distinct roots	$a \neq 0; b * b - 4 * a * c > 0$	(1,-5,6)
Complex roots	$a \neq 0; b * b - 4 * a * c < 0$	(2,3,4)

Black-Box Testing: Boundary Value Analysis (BVA)

- 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

Black-Box Testing: Boundary Value Analysis (BVA)

- 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
- Programmers may improperly use $<$ instead of $\leq$
- Boundary value analysis
  - Select test cases at the boundaries of equivalence classes
- For a function that computes the square root of an integer in the range of 1 and 5000
  - Test cases must include the values: $\{ 0, 1, 5000, 5001 \}$
- QES program reads (a, b, c) and solves: $ax^2 + bx + c = 0$
  - $a = 0$ is a boundary. Check if this test works well
  - $b * b - 4 * a * c = 0$ is a boundary. Check for the test

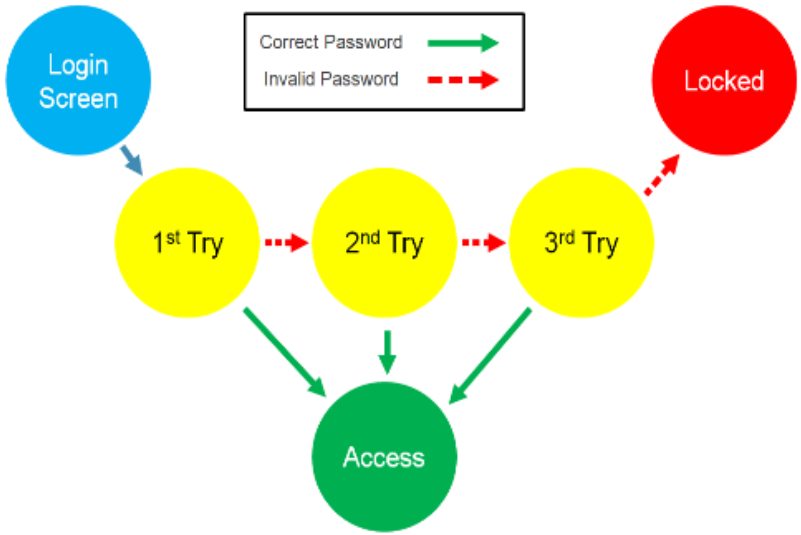
Black-Box Testing: State Transition Testing

- This technique usually considers the state, outputs, and inputs of a system during a specific period
- Based on the type of software that is tested, it checks for the behavioral changes of a system in a particular state or another state while maintaining the same inputs
- The test cases for this technique are created by checking the sequence of transitions and state or events among the inputs
- The whole set of test cases will have the traversal of the expected output values and all states

Source: Black Box Testing Techniques with Examples

Black-Box Testing: State Transition Testing

- **Example**: We need to perform black box testing for a login screen which allows a maximum of three attempts before the login is locked. Assuming that the user-id is correct, design the test suite.



Testcase #	Password for Trial			State
	Trial 1	Trial 2	Trial 3	*Golden*
(1)	Correct	X	X	Access
(2)	Wrong	Correct	X	Access
(3)	Wrong	Wrong	Correct	Access
(4)	Wrong	Wrong	Wrong	Locked

Source: Black Box Testing Techniques with Examples

Black-Box Testing: Decision Table Testing

- In some instances, the inputs combinations can become very complicated for tracking several possibilities
- Such complex situations rely on decision tables, as it offers the testers an organized view about the inputs combination and the expected output
- This technique is identical to the graph-based testing technique; the major difference is using tables instead of diagrams or graphs
- **Example 1**:

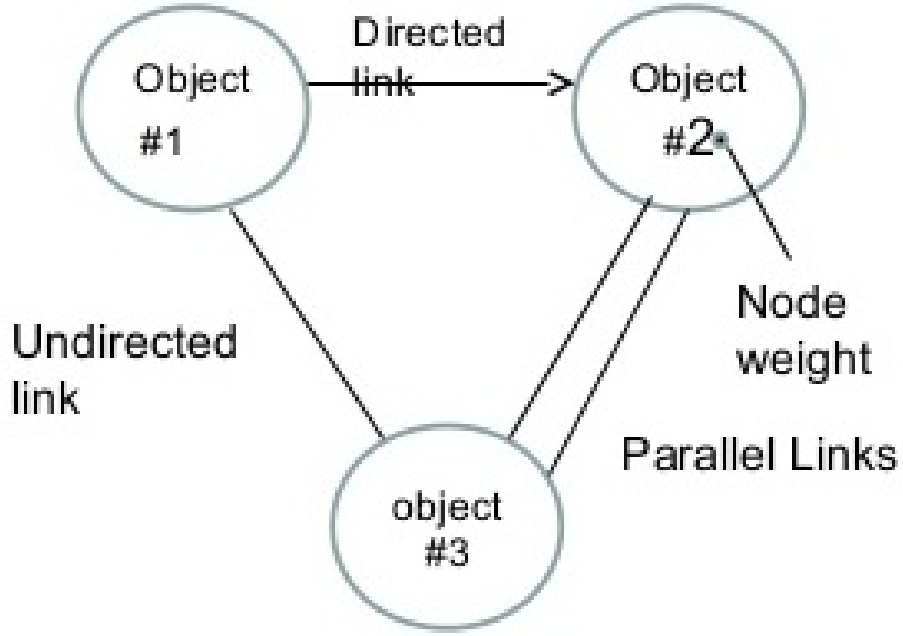
	Rule 1	**Rule 2**	**Rule 3**	**Rule 4**
Condition				
End of month	No	Yes	Yes	Yes
Salary Transferred	N/A	No	Yes	Yes
Provident fund	N/A	N/A	No	Yes
Action				
Income tax	No	No	Yes	Yes
Provident fund	No	No	No	Yes

- **Example 2**: States of various leaves in LMS Testplan

Source: Black Box Testing Techniques with Examples

Black-Box Testing: Graph-Based Testing

- This technique of Black box testing involves a graph drawing that depicts the link between the causes (inputs) and the effects (output), which trigger the effects
- This testing utilizes different combinations of output and inputs. It is a helpful technique to understand the software's functional performance, as it visualizes the flow of inputs and outputs in a lively fashion
- **Example**:



Source: Black Box Testing Techniques with Examples

Black-Box Testing: Error Guessing Technique

- This testing technique is capable of guessing the erroneous output and inputs to help the tester fix it easily
- It is solely based on judgment and perception of the earlier end user experience.

Source: Black Box Testing Techniques with Examples

Types of Black-Box Testing

- **Functional Testing**:
  This type of testing is useful for the testers in identifying the functional requirements of a software or system
- **Regression Testing**:
  This testing type is performed after the system maintenance procedure, upgrades or code fixes to know the impact of the new code over the earlier code
- **Non-Functional Testing**:
  This testing type is not connected with testing for any specific functionality but relates to non-functional parameters like usability, scalability, and performance

White-Box Testing Strategies

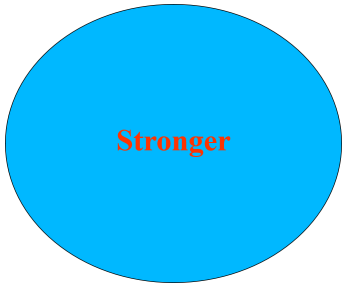
- Coverage-based
  - Design test cases to cover certain program elements
- Fault-based
  - Design test cases to expose some category of faults
- There exist several popular white-box testing methodologies
  - Coverage Based
    - Statement / Line Coverage
    - Function / Call Coverage
    - Branch Coverage
    - Condition Coverage
    - Path Coverage
    - MC/DC Coverage
  - Fault Based
    - Mutation Testing
    - Data Flow Testing

White Box Testing: Coverage-Based Testing vs Fault-Based Testing

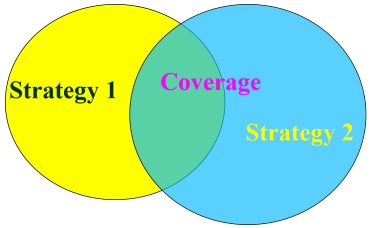
- Idea behind coverage-based testing
  - Design test cases so that certain program elements are executed (or covered)
  - Example: statement coverage, path coverage, etc.
- Idea behind fault-based testing
  - Design test cases that focus on discovering certain types of faults
  - Example: Mutation testing

Stronger, Weaker, and Complementary Testing

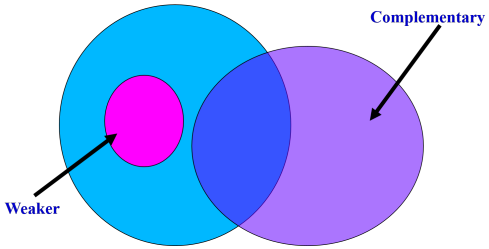
- Stronger and Weaker Testing: Test cases are a super-set of a weaker testing
  - A stronger testing covers at least all the elements of the elements covered by a weaker testing



- Complimentary Testing



- Stronger, Weaker & Complimentary Testing



White Box Testing: Statement / Line Coverage

- Statement coverage methodology
  - Design test cases so that every statement in the program is executed at least once
- The principal idea
  - Unless a statement is executed
  - We do not know if an error exists in that statement
- Observe that a statement behaves properly for one i/p
  - No guarantee that it will behave correctly for all i/p values
- **Line Coverage**
  - Most tools (like `gcov`, `lcov`) actually compute the coverage for the source lines
  - So it multiple statements are written in a single line, the coverage data may be inaccurate. For example, there are two statements in the following line that will be counted as one only

    ```
    x = 5; y = 6;
    ```

  - While the above may be okay, conditional statements should be placed in separate lines for proper statement coverage. For example, for

    ```
    if (x > y) max = x;
    max = y;
    ```

    we would never now if the statement `max = x;` has actually been executed or not
  - Single statement in every source line is a common coding guideline

White Box Testing: Statement / Line Coverage

- Coverage measurement
  - $Statement\ Coverage = \frac{\#executed\ \ statements}{\#statements} * 100\%$
  - $Line\ Coverage = \frac{\#executed\ \ lines}{\#lines} * 100\%$
  - Rationale: a fault in a statement can only be revealed by executing the faulty statement. Consider Euclid's GCD algorithm:

```
int f1(int x, int y) {
    while (x != y) {
        if (x>y)
            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
- Note that { `(x=4,y=3)` } or { `(x=3,y=4)` } itself will cover all lines due the the iterations of the `while` loop. However, it is customary to keep the analysis simple (mostly through a single flow) and include all of them in the test suite

White Box Testing: Function / Call Coverage

- Function Coverage methodology
  - Design test cases so that every function in the program is called at least once
  - This metric reports the call count of a function; it does not pay mind to the execution of the body of the function
  - It is thus generally useful as an initial assessment of a project's coverage, but higher-order metrics are generally required for more in-depth analysis
- Call Coverage methodology
  - Design test cases so that every function call in the program is executed at least once
  - In contrast to Function Coverage (which is about execution of the function itself), Call Coverage mandates that each call found in the code is executed at least once
  - 100% Function Coverage therefore does not imply 100% Call Coverage
  - The reverse is also not necessarily true either - there may be functions that are not called anywhere in the code (unused functions)
- Coverage measurement
  - $\textit{Function Coverage} = \frac{\#\textit{function called at least once}}{\#\textit{total functions}} * 100\%$
  - $\textit{Call Coverage} = \frac{\#\textit{call sites for functions called}}{\#\textit{total call sites of functions}} * 100\%$

White Box Testing: Branch Coverage

- Test cases are designed such that
  - Different branch conditions – are given true and false values in turn
- Branch testing guarantees statement coverage
  - A stronger testing compared to the statement coverage-based testing
  - Why?

```
1: cin >> x;
2: if (0 == x)
3:     x = x + 1;
4: y = 5;
```

Note that, {(x = 0)} covers lines {1, 2, 3, 4} while {(x = 1)} covers only lines {1, 2, 4}. So with {(x = 0)}, we get 100% statement coverage. But then, did we check for the jump from line 2 to 4 for the false condition? This condition did not get tested. So we need {(x = 0), (x = 1)} for 100% branch coverage and it obviously leads to 100% statement coverage.

How do we get 100% branch coverage for:

```
1: if (true)
2:     x = x + 1;
3: y = 5;
```

White Box Testing: Branch Coverage

- Example:

```
0: int f1(int x, int y) {
1:     while (x != y) {
2:         if (x > y)
3:             x = x - y;
4:         else y = y - x;
5:     }
6:     return x;
7: }
```

Branches: {1-2, 1-6, 2-3, 2-4, 3-5, 4-5, 5-1}

- Test cases for branch coverage can be
  - (x=3,y=3): {1-6}: {1, 6}
  - (x=4,y=3): {1-2, 2-3}: {1, 2, 3, 5}
  - (x=3,y=4): {1-2, 2-4}: {1, 2, 4, 5}
- *Adequacy criterion*: Each branch (in CFG) must be executed at least once
  - $Branch\ Coverage = \frac{\#executed\ branches}{\#branches} * 100\%$
- Traversing all edges of a graph causes all nodes to be visited
  - So test suites that satisfy the branch adequacy criterion for a program also satisfy the statement adequacy criterion for the same program
- The converse is not true
  - A statement-adequate (or node-adequate) test suite may not be branch-adequate (edge-adequate)

White Box Testing: Condition Coverage

All Branches can still miss conditions

- Sample fault: missing operator (negation)

```
digit_high == 1 || digit_low == -1
```

- Branch adequacy criterion can be satisfied by varying only `digit_low`
  - The faulty sub-expression might not be tested
  - Even though we test both outcomes of the branch

White Box Testing: 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
    - ▷ That is, given true and false values
- Basic condition testing
  - Adequacy criterion: each basic condition must be executed at least once
- Coverage
  - $Condition\ Coverage = \frac{\#truth\ values\ taken\ by\ all\ basic\ conditions}{2*\#basic\ conditions} * 100\%$

White Box Testing: Branch Testing

- Branch testing is the simplest condition testing strategy
  - Compound conditions appearing in different branch statements
    - ▷ Are given true and false values
  - Condition testing
    - ▷ Stronger testing than branch testing
  - Branch testing
    - ▷ Stronger than statement coverage testing

White Box 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
- Commonly known as **Multiple Condition Coverage** (MCC), **Multicondition Coverage** and **Condition Combination Coverage**

White Box Testing: Modified Condition / Decision (MC/DC)

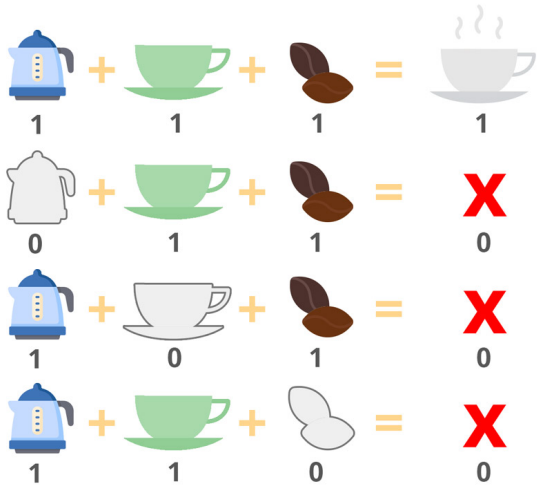
- **Motivation**
  - Effectively test important combinations of conditions, without exponential blowup in test suite size
  - *Important* combinations means: Each basic condition shown to independently affect the outcome of each decision
- **Requires**
  - For each basic condition C, two test cases obtained
  - Values of all evaluated conditions except C are the same
  - Compound condition as a whole evaluates to true for one and false for the other
- **MC/DC** stands for Modified Condition / Decision Coverage
- A kind of **Predicate Coverage** technique
  - Condition: Leaf level Boolean expression.
  - Decision: Controls the program flow
- **Main Idea**
  - Each condition must be shown to independently affect the outcome of a decision, that is, the outcome of a decision changes as a result of changing a single condition

White Box Testing: MC/DC in action: The Cup of Coffee Example

To make a cup of coffee, we would need ALL of the following: a kettle, a cup and coffee. If any of the components were missing, we would not be able to make our coffee. Or, to express this another way:

```
if (kettle && cup && coffee)
    return cup_of_coffee;
else
    return false;
```

Or to illustrate it visually:



	Inputs			Outputs
Test	Kettle	Mug	Coffee	Result
1	0	0	0	0
2	0	0	1	0
3	0	1	0	0
4	0	1	1	0
5	1	0	0	0
6	1	0	1	0
7	1	1	0	0
8	1	1	1	1

- **Tests 4 & 8 demonstrate that 'kettle' can independently affect the outcome**
- **Tests 6 & 8 demonstrate that 'mug' can independently affect the outcome**
- **Tests 7 & 8 demonstrate that 'coffee' can independently affect the outcome**

Source: What is MC/DC?

White Box Testing: MC/DC in action: Flow Check

A sample C/C++ function with a decision composed of OR and AND expressions illustrates the difference between Modified Condition/Decision Coverage and a coverage of all possible combinations as required by MCC:

```
bool isSilent(int *line1, int *line2)
{
    if ((!line1 || *line1 <= 0) && (!line2 || *line2 <= 0))
        return true;
    else
        return false;
}
```

Or to illustrate it visually:

Source: Modified Condition/Decision Coverage (MC/DC)

White Box Testing: Modified Condition / Decision (MC/DC)

Cond = `(((a || b) && c) || d) && e`

Condition Coverage Test Cases

Every condition in the decision has taken all possible outcomes at least once

#	a	b	c	d	e	Cond
0:	F	F	F	F	F	F
1:	F	F	F	F	T	F
2:	F	F	F	T	F	F
3:	F	F	F	T	T	T
4:	F	F	T	F	F	F
5:	F	F	T	F	T	F
6:	F	F	T	T	F	F
7:	F	F	T	T	T	T
8:	F	T	F	F	F	F
9:	F	T	F	F	T	F
10:	F	T	F	T	F	F
11:	F	T	F	T	T	T
12:	F	T	T	F	F	F
13:	F	T	T	F	T	T
14:	F	T	T	T	F	F
15:	F	T	T	T	T	T

#	a	b	c	d	e	Cond
16:	T	F	F	F	F	F
17:	T	F	F	F	T	F
18:	T	F	F	T	F	F
19:	T	F	F	T	T	T
20:	T	F	T	F	F	F
21:	T	F	T	F	T	T
22:	T	F	T	T	F	F
23:	T	F	T	T	T	T
24:	T	T	F	F	F	F
25:	T	T	F	F	T	F
26:	T	T	F	T	F	F
27:	T	T	F	T	T	T
28:	T	T	T	F	F	F
29:	T	T	T	F	T	T
30:	T	T	T	T	F	F
31:	T	T	T	T	T	T

MC/DC Coverage Test Cases

Every condition in the decision independently affects the decision's outcome

#	a	b	c	d	e	Cond	Cases
1:	T	X	T	X	T	T	21, 23, 29, 31
2:	F	T	T	X	T	T	13, 15
3:	T	X	F	T	T	T	19, 27
4:	T	X	T	X	F	F	20, 22, 28, 30
5:	T	X	F	F	X	F	16, 17, 24, 25
6:	F	F	X	F	X	F	0, 1, 4, 5

White Box Testing: Modified Condition / Decision (MC/DC)

- **MC/DC is**
  - basic condition coverage (C)
  - branch coverage (DC)
  - plus one additional condition (M): every condition must independently affect the decision's output
- It is subsumed by compound conditions and subsumes all other criteria discussed so far
  - stronger than statement and branch coverage
- A good balance of thoroughness and test size (and therefore widely used)
- MC/DC code coverage criterion is commonly used software testing. For example, DO-178C software development guidance in the aerospace industry requires MC/DC for the most critical software level (DAL A).
- **MC/DC vs. MCC**
  - MCC testing is characterized as number of tests = 2^C. In coffee example we have 3 conditions (kettle, cup and coffee) therefore tests = $2^3 = 8$
  - MC/DC requires significantly fewer tests (C + 1). In coffee example we have 3 conditions, therefore $3 + 1 = 4$
  - In a real-world setting, most aerospace projects would include some decisions with 16 conditions or more. So the reduction would be from $2^16 = 65{,}536$ to $16 + 1 = 17$. That is, 65,519/65,536 = 99.97%

Source: What is MC/DC?

White Box Testing: Different Types of Code Coverage

Coverage Criteria	SC	DC	MC/DC	MCC
Every statement in the program has been invoked at least once	X			
Every point of entry and exit in the program has been invoked at least once		X	X	X
Every control statement (that is, branch-point) in the program has taken all possible outcomes (that is, branches) at least once		X	X	X
Every non-constant Boolean expression in the program has evaluated to both a True and False result		X	X	X
Every non-constant condition in a Boolean expression in the program has evaluated to both a True and False result			X	X
Every non-constant condition in a Boolean expression in the program has been shown to independently affect that expression's outcome			X	X
Every combination of condition outcomes within a decision has been invoked at least once				X

- **SC**: Statement Coverage
- **DC**: Decision Coverage
- **MC/DC**: Modified Condition / Decision Coverage
- **MCC**: Multiple Condition Coverage

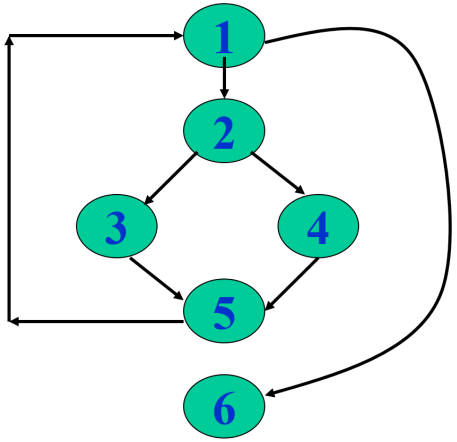
Source: What is MC/DC?

White Box Testing: 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
- 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
- Number all statements of a program
- Numbered statements
  - Represent nodes of 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

White Box Testing: Path Coverage: CFG

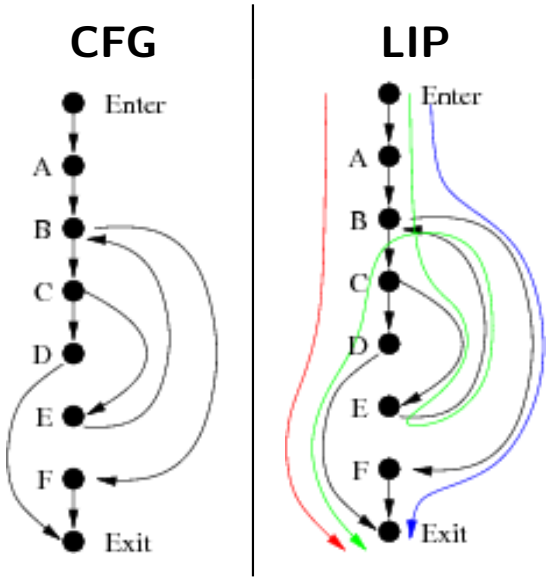

```
int f1(int x,int y) {
1 while (x != y){
2    if (x>y) then
3         x=x-y;
4    else y=y-x;
5 }
6 return x;        }
```



- 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
- Any path through the program that introduces at least one new edge is a *Linearly Independent Path* (LIP).
- A set of paths are linearly independent if none of them can be created by combining the others in some way.
  - It is straight forward to identify linearly independent paths of simple programs; but not so for complicated programs
- LIP in the above example:
  - 1,6
  - 1,2,3,5,1,6
  - 1,2,4,5,1,6

White Box Testing: Path Coverage: LIP

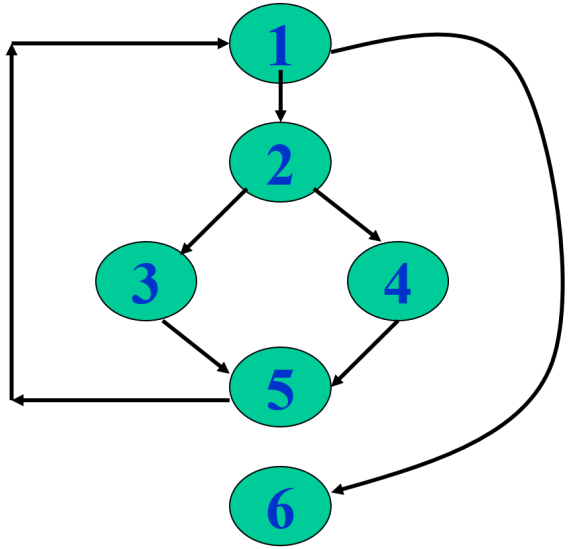

```
    public static boolean isPrime(int n) {
A       int i = 2;
B       while (i < n) {
C           if (n % i == 0) {
D               return false
            }
E           i++;
        }
F       return true;
    }
```



Source: The 'Linearly Independent Paths' Metric for Java

White Box Testing: Path Coverage: McCabe's Cyclomatic Metric

- An upper bound for the number of linearly independent paths of a program – a practical way of determining the maximum number of LIP
- 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
- Alternately, inspect control flow graph to determine number of bounded areas (any region enclosed by a nodes and edge sequence) in the graph $V(G)$ = Total number of bounded areas + 1
- Example: Cyclomatic complexity $= 7 - 6 + 2 = 3 = 2 + 1$



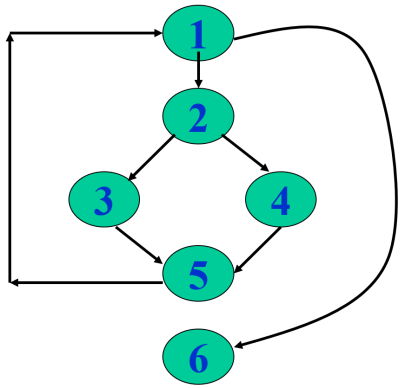
White Box Testing: Path Coverage: McCabe's Cyclomatic Metric

- 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
- 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
- A measure of the number of independent paths in a program
- Provides a lower bound
  - for the number of test cases for path coverage
- 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

White Box Testing: Path Coverage: Practical Path Testing

- Tester proposes initial set of test data using her experience & judgement
- A dynamic program analyzer is used to measures which parts of the program have been tested
- Result used to determine when to stop testing
- 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: Number of independent paths: 3

```
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;        }
```

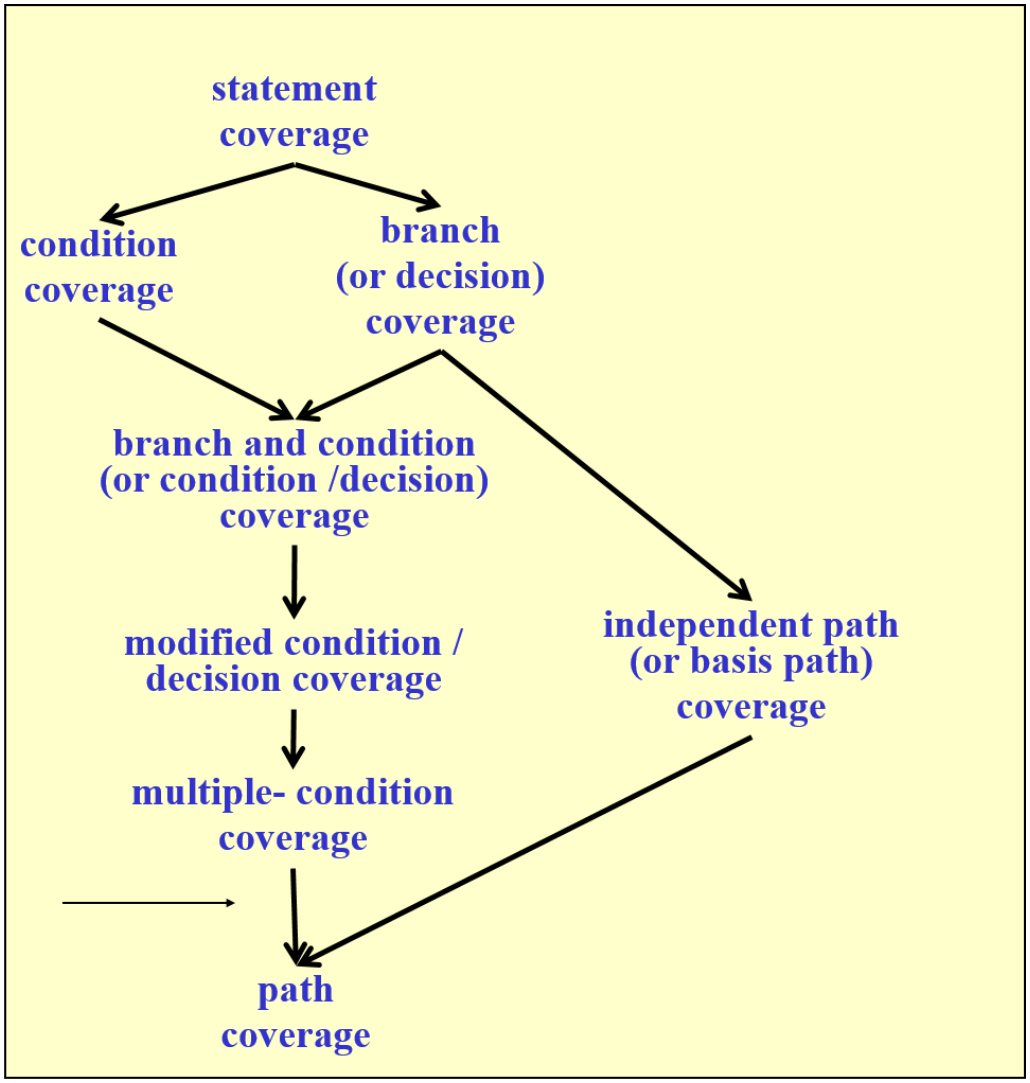


  - 1,6: test case (x=1, y=1)
  - 1,2,3,5,1,6: test case (x=1, y=2)
  - 1,2,4,5,1,6: test case (x=2, y=1)

White Box Testing: An 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 of a program:
  - Also indicates the psychological complexity of a program
  - Difficulty level of understanding the program
- 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

White-Box Testing: Coverage Testing Summary



White Box Testing: 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
- Good software development organizations:
  - Restrict cyclomatic complexity of functions to a maximum of ten or so
- Insert faults into a program:
  - Check whether the tests pick them up
  - Either validate or invalidate the tests
  - Example:

```
1: cin >> x;
2: if (0 == x)
3:     x = x + 1;
4: y = 5;
```

Note that, {(x = 0)} covers lines {1, 2, 3, 4} while {(x = 1)} covers only lines {1, 2, 4}. So with {(x = 0)}, we get 100% statement coverage. But then, did we check for the jump from line 2 to 4 for the false condition? This condition did not get tested. So we need {(x = 0), (x = 1)} for 100% branch coverage and it obviously leads to 100% statement coverage.
How do we get 100% branch coverage for:

```
1: if (true)
2:     x = x + 1;
3: y = 5;
```

White Box Testing: Mutation Testing

- Insert faults into a program:
  - Check whether the tests pick them up
  - Either validate or invalidate the tests
- Each time the program is changed
  - it is called a **mutated program**
  - the change is called a **mutant**
- 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**
- 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 pre-defining a set of primitive changes that can be applied to the program
- The primitive changes can be
  - Deleting a statement
  - Altering an arithmetic operator
  - Changing the value of a constant
  - Changing a data type, etc.

White Box Testing: Mutation Testing: Error Seeding

- 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
- Let
  - N be the total number of errors in the system and n of these errors be found by testing
  - S be the total number of seeded errors and s of the seeded errors be found during testing
  - $\frac{n}{N} = \frac{s}{S}$
  - $N = S * \frac{n}{s}$
  - Remaining defects:

$$N - n = n * \frac{S - s}{s}$$

- Example:
  - 100 errors were introduced
  - 90 of these errors were found during testing
  - 50 other errors were also found
  - Remaining errors:

$$50 * \frac{100 - 90}{90} = 6$$

- 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
  - even after all test cases have been exhausted, the test suite is enhanced to kill the mutant

White Box Testing: Data Flow Testing

Data Flow Testing is a type of structural testing. It is a method that is used to find the test paths of a program according to the locations of definitions and uses of variables in the program

- It is concerned with:
  - Statements where variables receive values
  - Statements where these values are used or referenced
- To illustrate the approach of data flow testing, assume that each statement in the program assigned a unique statement number. For a statement number S:
  - DEF(S) = {X | statement S contains a definition of X}
  - USE(S)= {X | statement S contains a use of X}
  - Example: 1: a = b; DEF(1) = {a}, USE(1) = {b}
  - Example: 2: a = a + b; DEF(2) = {a}, USE(2) = {a,b}
- If a statement is a loop or if condition then its DEF set is empty and USE set is based on the condition of statement s.
- Data Flow Testing uses the control flow graph to find the situations that can interrupt the flow of the program.
- Reference or define anomalies in the flow of the data are detected at the time of associations between values and variables. These anomalies are:
  - A variable is defined but not used or referenced
  - A variable is used but never defined
  - A variable is defined twice before it is used

White Box Testing: Data Flow Testing

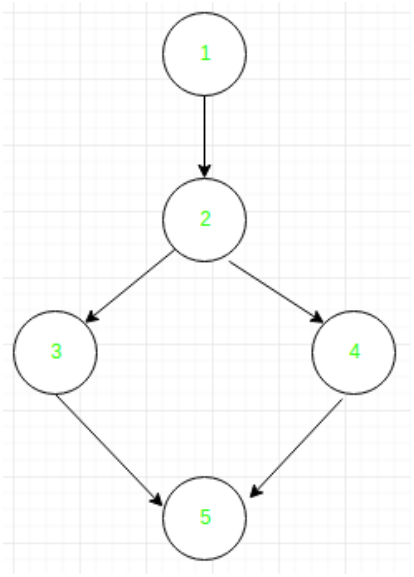
- **Advantages** of Data Flow Testing
  - To find a variable that is used but never defined
  - To find a variable that is defined but never used
  - To find a variable that is defined multiple times before it is use
  - Deallocating a variable before it is used
- **Disadvantages** of Data Flow Testing
  - Time consuming and costly process
  - Requires knowledge of programming languages

White Box Testing: Data Flow Testing

- Example:

```
1. read x, y;
2. if (x > y)
3. a = x + 1
   else
4. a = y - 1
5. print a;
```

- CFG



- Define/use of variables

Variable	Defined at node	Used at node
x	1	2, 3
y	1	2, 4
a	3, 4	5

Source: Data Flow Testing

Data Object Categories

- (d) Defined, Created, Initialized. An object (like variable) is defined when it:
  - appears in a data declaration
  - is assigned a new value
  - is a file that has been opened
  - is dynamically allocated
  - ...
- (k) Killed, Undefined, Released
- (u) Used:
  - (c) Used in a calculation
  - (p) Used in a predicate
  - An object is used when it is part of a computation or a predicate
    - A variable is used for a computation (c) when it appears on the RHS (sometimes even the LHS in case of array indices) of an assignment statement
    - A variable is used in a predicate (p) when it appears directly in that predicate

Source: Topics in Software Dynamic White-box Testing: Part 2: Data-flow Testing

White Box Testing: Data Flow-Based Testing: Definition and Use


```
1.   read (x, y);
2.   z = x + 2;
3.   if (z < y)
4        w = x + 1;
     else
5.       y = y + 1;
6.   print (x, y, w, z);
```

Def	C-use	P-use
x, y		
z	x	
		z, y
w	x	
y	y	
	x, y, w, z	

- To find a variable that is used but never defined
- To find a variable that is defined but never used
- To find a variable that is defined multiple times before it is use
- Deallocating a variable before it is used

Processes for Development

Development involves a number of processes:

- Design & Coding (discussed in Modules 2 & 4)
- Debugging
- Issue / Bug Tracking
- Testing
- Documentation
- Release (discussed as a part of Maintenance)
- Version Control (discussed as a part of Maintenance)

Testing Processes

- 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
- The input data domain of most software products is very large
  - It is not practical to test the software exhaustively with each input
- Testing however exposes many errors
  - Testing provides a practical way of reducing defects in a system
  - Increases the users' confidence in the system
- Testing
  - Is a continual process
  - Needs significant automation (especially to repeats tests already done when new stuff is added). Usually achieved through
    - ▷ **Regression Testing**
  - Has to happen at all phases and all levels of abstraction – both for development and for maintenance
- Software products are tested at three levels
  - **Unit testing**
  - **Integration testing**
  - **System testing**

Regression Testing

- **Regression**[1] **testing** is re-running functional and non-functional tests to ensure that previously developed and tested software still performs after a change
  - If not, that would be called a regression
  - The software *Regresses*
- Regression testing
  - Is usually done through auotmated processes after each change to the system after each bug fix
  - Ensures that no new bug has been introduced due to the change or the bug fix – the new system's performance is at least as good as the old system
  - Is always used during incremental system development
- Regression test is:
  - Required before every code check-in (no regression, that is)
  - Used at every level for every kind of test
  - Most powerful tool for quality control for software development and maintenance

[1] a return to a former or less developed state

Regression Testing: Positive & Negative Test Cases

	Execution Outcome	
Type of Test	**Successful**	**Unsuccessful**
Positive Case (Success)	PASS	FAIL
Negative Case (Failure)	FAIL	PASS

Regression outcome is typically shown by:

$$\frac{\#\ of\ PASS\ cases}{\#\ of\ PASS + FAIL\ (total)\ cases} * 100\%$$

How to do Regression Testing? Approach

- We need to first debug the code to identify the bugs
- Once the bugs are identified and fixed, then the regression testing is done by selecting relevant test cases from the test suite that covers both modified and affected parts of the code
- Regression Testing can be carried out using the following strategies:
  - Retest All
    - Needed for all version major / minor releases
  - Regression Test Selection
    - Test cases which have frequent defects
    - Functionalities which are more visible to the users
    - Test cases which verify core features of the product
    - Test cases of Functionalities which has undergone more and recent changes
    - All Integration Test Cases & all Complex Test Cases
    - Boundary value test cases
    - Samples of Successful test cases & Failure test cases
  - Prioritization of Test Cases
    - Depending on business impact, critical & frequently used functionalities
    - *General prioritization*: Beneficial on subsequent versions
    - *Version-specific prioritization*: Beneficial for a particular version of the software
  - Hybrid
    - This technique is a hybrid of regression test selection and test case prioritization

Source: What is Regression Testing? Definition, Test Cases (Example)

How to do Regression Testing? Automation

Regression Testing relies on intense automation to reduce menial manual testing efforts and costs

- Scripting for Regression Testing
  - Scripting is the biggest handle for Regression Automation
  - Unix: Shell, Bash, sed, awk etc.
  - Windows: VBScript
  - Perl
  - Python
- Regression Testing Tools
  - Selenium: Open Source portable framework for automating web applications (browser-based regression). It provides a playback tool for authoring functional tests without the need to learn a test scripting language (Selenium IDE)
  - Micro Focus Unified Functional Testing (UFT One) (formerly known as QuickTest Professional (QTP)): An tool to automate functional and regression test cases. UFT One Intelligent test automation with embedded AI-based capabilities that accelerates testing across desktop, web, mobile, mainframe, composite and packaged enterprise-grade apps.
  - IBM Rational Functional Tester (RFT): An automated functional testing and regression testing tool. This software provides automated testing capabilities for functional, regression, GUI and data-driven testing. It supports a range of applications, such as web-based, .Net, Java, Siebel, SAP, terminal emulator-based applications, PowerBuilder, Ajax, Adobe Flex, Dojo Toolkit, GEF, Adobe PDF documents, zSeries, iSeries and pSeries.

Source: What is Regression Testing? Definition, Test Cases (Example)

Retesting and Regression Testing

- **Retesting** means testing the functionality or bug again to ensure the code is fixed
  - If it is not fixed, defect needs to be re-opened
  - If fixed, defect is closed
- **Regression testing** means testing your software application when it undergoes a code change to ensure that the new code has not affected other parts of the software.

Regression Testing	Retesting
Confirms that a recent program or *code change has not adversely affected existing features*	Confirms that the *test cases that failed in the final execution are passing after the defects are fixed*
Ensures that new *code changes do not have any side effects* to existing functionalities	Is done *on the basis of the Defect fixes*
Defect verification **is not** in scope	*Defect verification* **is** in scope
May be carried out *parallel with Re-testing*	Must be carried out *before regression testing*
May be automated *Manual Testing is expensive*	*Cannot automate the test cases for Retesting*
Generic testing	*Planned testing*
Done for **passed** *test cases*	*Done only for* **failed** *test cases*
Checks for *unexpected side-effects*	Makes sure that *the original fault has been corrected*
Done for any *mandatory modification or changes in an existing project*	Executes a defect with *the same data and the same environment with different inputs with a new build*
Test cases for **can be** *obtained* from the *functional specification, user tutorials and manuals, and defect reports in regards to corrected problems*	Test cases for retesting **cannot be** *obtained before start testing*

Source: Difference Between Retesting and Regression Testing

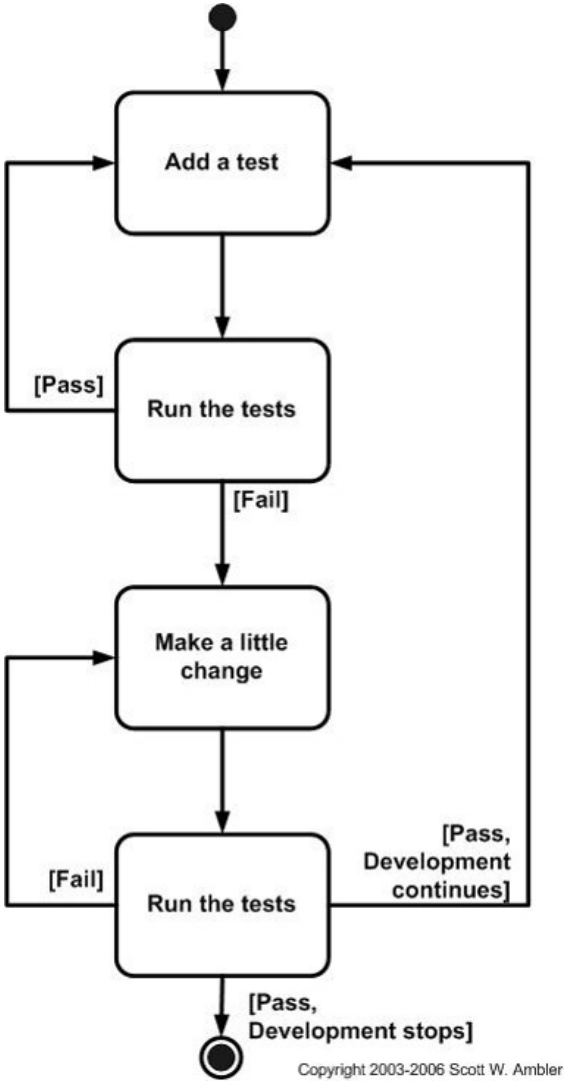
Regression Test Suite

- To repeat old tests and runs
- Test Case with Golden
  - Unit Tests
  - API / Application Tests
  - Directed Tests, Corner Cases, Customer Tests
  - Random & Huge Tests
  - Performance Tests
    - Time
    - Resources: Low Memory Tests
  - Coverage Tests
  - ...
- Folder Structure
  - Uniformity names of test case files: Critical for automation scripts
- Run script

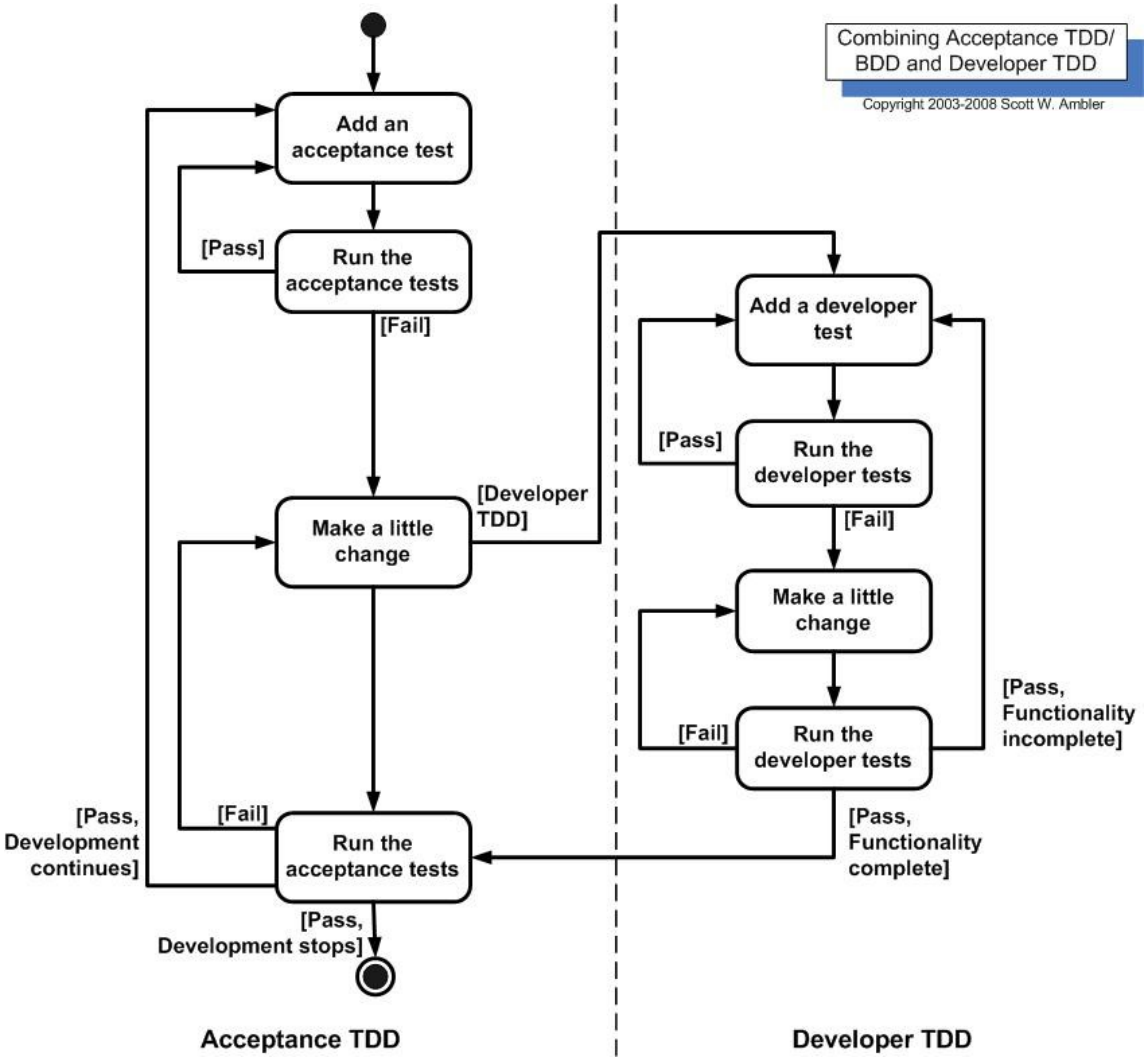
Unit Testing

- 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 reduces debugging effort several folds
  - Programmers carry out unit testing immediately after they complete the coding of a module
- Unit testing drives development in Test-Driven Development (TDD)

TDD



TDD


Combining Acceptance TDD/ BDD and Developer TDD
Copyright 2003-2008 Scott W. Ambler
Add an acceptance test
[Pass]
Run the acceptance tests
[Fail]
Make a little change
[Developer TDD]
[Pass, Development continues]
[Fail]
Run the acceptance tests
[Pass, Development stops]
Add a developer test
[Pass]
Run the developer tests
[Fail]
Make a little change
[Fail]
Run the developer tests
[Pass, Functionality incomplete]
[Pass, Functionality complete]
Acceptance TDD
Developer TDD

Unit Testing

- For a class attach a (test) unit (typically JUnit in Java or CPPUnit in C++)) that tests the class
- D Language provides specific support for unit testing
- Unit testing is a main feature of D
- Unit testing in D
  - Unit tests can be added to a class - they are run upon program start-up
  - Aids in verifying that class implementations weren't inadvertently broken
  - Unit tests is a part of the code for a class
  - Creating tests is a part of the development process

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
  - Validate a fully developed system against its requirements
- Develop the integration plan by examining the structure chart:
  - **Big bang approach**
  - **Top-down approach**
  - **Bottom-up approach**
  - **Mixed approach**

Integration Testing

- **Big Bang** is the simplest integration testing approach
  - All the modules are simply put together and tested
  - This technique is used only for very small systems
  - 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

System tests are designed to validate a fully developed system

- To assure that it meets its requirements
- Functional requirements are validated through functional tests
- Non-functional requirements validated through performance tests

Testing	Release	Features
Alpha	Alpha	• All functionality has been implemented • Reasonable testing has been done – *Regression passes with known exceptions* • *Testing is carried out by the test team within the company* • No preview expected from *real* users yet
Beta	Beta	• All functionality has been thoroughly tested • Extensive testing has been done – *Regression passes with some (documented) exceptions* • Releases made to friendly customers (or non-developer internal groups). Or beta programs launched • *Testing performed by a select group of friendly customers* • Must include *real* users of a system
Acceptance	FCS	• FCS: **First Customer Shipment** • Company has gained fair confidence on the quality – *Regression is clean with rare (documented) exceptions* • Customer will use this release now till (Minor) bug fix releases are done or new (Major) version is produced • *Testing performed by the customer to determine whether the system should be accepted or rejected*

Test Plans: LMS

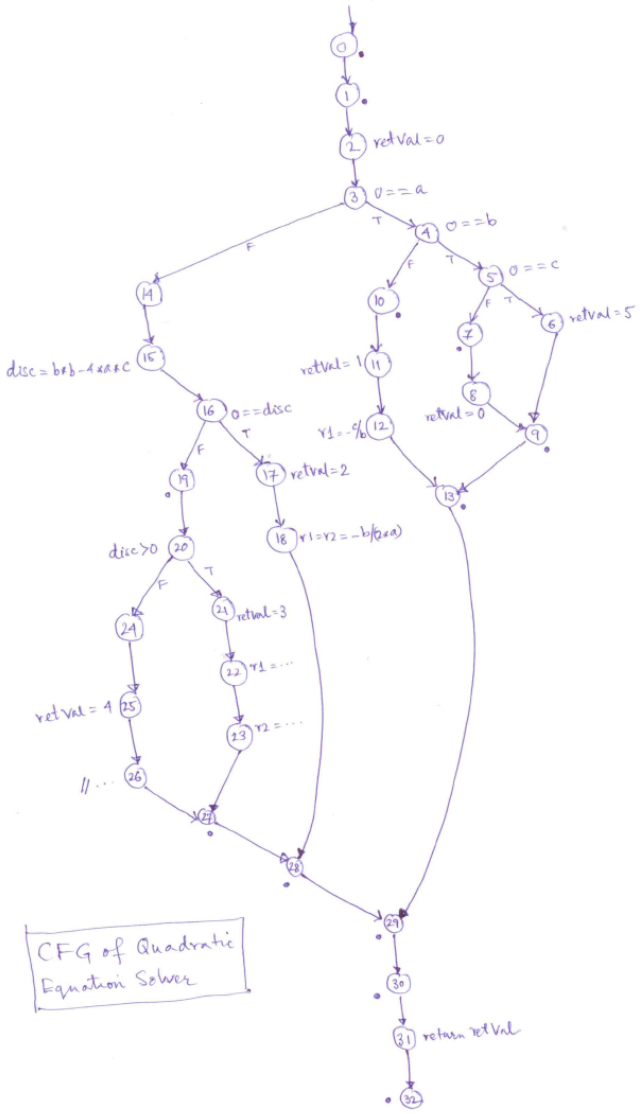
Leave Quota								
Test	CL	EL	DL	SL	ML	PL	LWP	Remarks
Credit			X					Checks if leave is rightly credited
Carry Forward			X					Checks if leave is rightly carried forward to next period
Brought Over			X					Checks if leave is rightly brought over from previous period
Encashment	X		X	X	X	X	X	Checks if leave is rightly transferred for encashment
Leave Period								
Test	CL	EL	DL	SL	ML	PL	LWP	Remarks
Duration								Checks if the leave is within permissible number of days
Holiday Mix								Checks if the leave mixes right with holidays - before, after, during, etc
Leave-Leave Mix								Checks if the leave mixes right with other leaves
Leave Financial								
Test	CL	EL	DL	SL	ML	PL	LWP	Remarks
Encashment Credit	X		X	X	X	X	X	Checks if leave is rightly encashed to salary
Salary Deduction	X	X	X	X	X	X		Checks if salary is rightly deducted / held back for exceeding leave quota
Leave Conditions								
Test	CL	EL	DL	SL	ML	PL	LWP	Remarks
Pre-Approval								Checks if a leave is usually pre-approved
Post-Approval		X			X	X	X	Checks if a leave is post-approved under special circumstances
Medical	X	X	X			X		Checks the medical conditions and certificates for leave
Maternity	X	X	X	X				Checks the maternity conditions and certificates for leave
Parental	X	X	X	X			X	Checks the parental conditions and certificates for leave
Exigency	X	X	X	X	X	X		Checks the exigency conditions and documents for leave

Test Plans: QES: Code

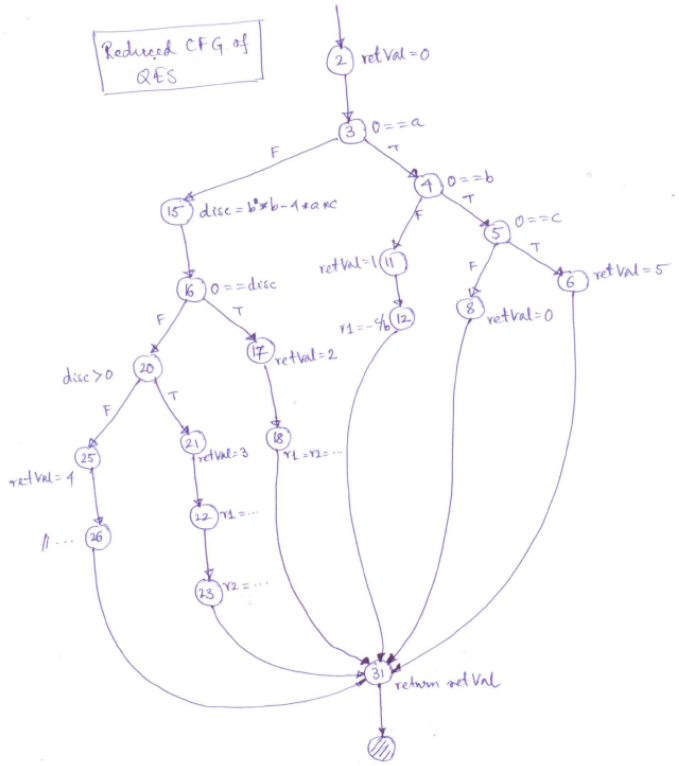

```
00: unsigned int Solve(double a, double b, double c, double& r1, double& r2)
01: {
02:     unsigned int retVal = 0;
03:     if (0 == a) {
04:         if (0 == b) {
05:             if (0 == c) { // Infinite solutions
06:                 retVal = 5;
07:             } else { // Inconsistent equation
08:                 retVal = 0;
09:             }
10:         } else { // Linear equation
11:             retVal = 1;
12:             r1 = -c/b;
13:         }
14:     } else {
15:         double disc = b*b - 4*a*c;
16:         if (0 == disc) { // Repeated roots
17:             retVal = 2;
18:             r1 = r2 = -b/(2*a);
19:         } else {
20:             if (disc > 0) { // Real distinct roots
21:                 retVal = 3;
22:                 r1 = (-b + sqrt(disc))/(2*a);
23:                 r2 = (-b - sqrt(disc))/(2*a);
24:             } else { // Complex conjugate roots
25:                 retVal = 4;
26:                 r1 = (-b)/(2*a); r2 = (sqrt(-disc))/(2*a);
27:             }
28:         }
29:     }
30:
31:     return retVal;
32: }
```

Test Plans: QES: CFG

CFG of QES



Reduced CFG of QES



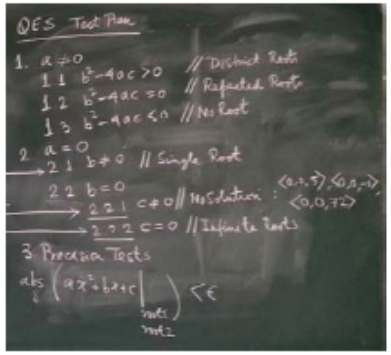
Test Plans: QES


```
#include <iostream>
using namespace std;

00  unsigned int Solve(double a, double b, double c, double& r1, double& r2)
01  {
02          unsigned int retVal = 0;
03          if (0 == a) {
04                  if (0 == b) {
05                          if (0 == c) { // Infinite solutions
06                                  retVal = 5;
07                          } else { // Inconsistent equation
08                                  retVal = 0;
09                          }
10                  } else { // Linear equation
11                          retVal = 1;
12                          r1 = -c/b;
13                  }
14          } else {
15                  double disc = b*b - 4*a*c;
16                  if (0 == disc) { // Repeated roots
17                          retVal = 2;
18                          r1 = r2 = -b/(2*a);
19                  } else {
20                          if (disc > 0) { // Real distinct roots
21                                  retVal = 3;
22                                  r1 = (-b + sqrt(disc))/(2*a);
23                                  r2 = (-b - sqrt(disc))/(2*a);
24                          } else { // Complex conjugate roots
25                                  retVal = 4;
26                                  // ...
27                          }
28                  }
29          }
30
31          return retVal;
32  }
```

Equivalence Classes of Test Cases:

a	b	c	Case
0	0	0	Infinite roots
0	0	2	No root
0	2	-4	Single root
4	4	1	Repeated roots
1	-5	6	Distinct roots
2	3	4	Complex roots



a	b	c	Equivalence Class	Statements Covered	Branches Covered	Paths Covered
0	0	0	Infinite roots	2,3,4,5,6,31	2-3,3-4,4-5,5-6,6-31	2-3-4-5-6-31
0	0	2	No root	2,3,4,5,8,31	2-3,3-4,4-5,5-8,8-31	2-3-4-5-8-31
0	2	-4	Single root	2,3,4,11,12,31	2-3,3-4,4-11,11-12,12-31	2-3-4-11-12-31
4	4	1	Repeated roots	2,3,15,16,17,18,31	2-3,3-15,15-16,16-17,17-18,18-31	2-3-15-16-17-18-31
1	-5	6	Distinct roots	2,3,15,16,20,21,22,23,31	2-3,3-15,15-16,16-20,20-21,21-22,22-23,23-31	2-3-15-16-20-21-22-23-31
2	3	4	Complex roots	2,3,15,16,20,25,26,31	2-3,3-15,15-16,16-20,20-25,25-26,26-31	2-3-15-16-20-25-26-31

Test Plans: QES


```
int main()
{
        double a, b, c;
        double r1, r2;
        unsigned int retVal = 0;
        unsigned int moreInputs = 0;

        do {
                cout << "Input coefficients: a, b and c" << endl;
                cin >> a >> b >> c;
                cout << endl;

                switch (retVal = Solve(a, b, c, r1, r2)) {
                        case 0: cout << "No root" << endl; break;
                        case 1: cout << "Linear Eqn: r1 = " << r1 << endl;  break;
                        case 2: cout << "Repeated real roots: r1 = " << r1 << " r2 = " << r2 << endl;  break;
                        case 3: cout << "Distinct real roots: r1 = " << r1 << " r2 = " << r2 << endl;  break;
                        case 4: cout << "Complex conjugate roots" << endl;  break;
                        case 5: cout << "Infinite roots" << endl;  break;
                        default: cout << "Something wrong" << endl;  break;
                }

                cout << "Continue Solving? Input 1" << endl;
                cin >> moreInputs;

        } while (1 == moreInputs);

        return 0;
}
```