

Software Testing

A thick, horizontal yellow brushstroke with a textured, painterly appearance, spanning most of the width of the slide.

Organization of this Lecture

- Introduction to Testing.
 - What is Testing?
 - Why we need Testing?
 - Terminology of Software Testing
 - Context importance in Software Testing
 - Software Defect
 - Testing & Quality in Software domain
- Black-box Testing
- White-box testing:
 - statement coverage
 - path coverage
 - branch testing
 - condition coverage
 - Cyclomatic complexity
- Summary

Testing??

- when we are testing something we are checking whether it is **OK?**



Why we need Testing?

□ Online Banking



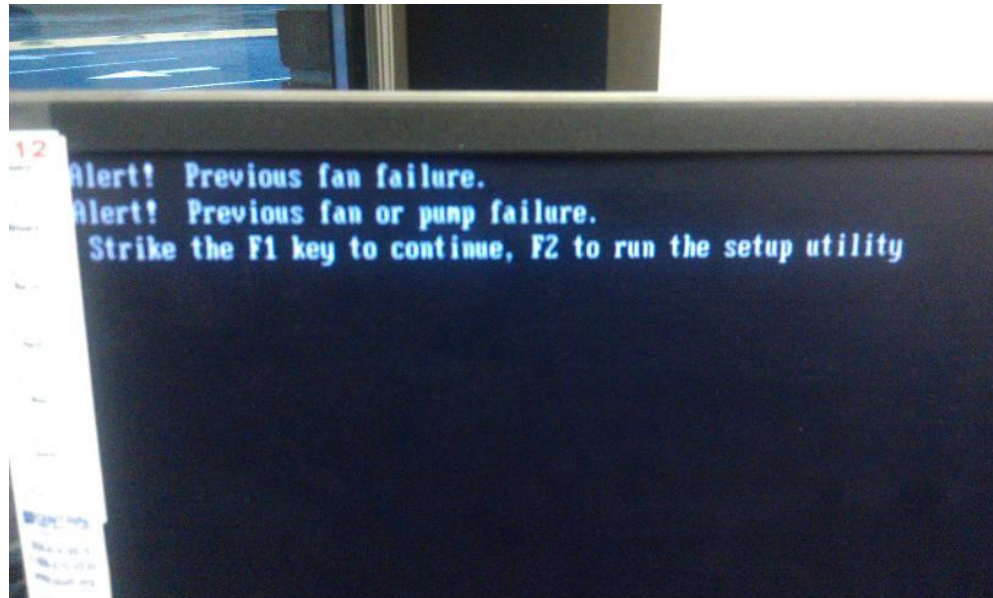
- Communication devices

- Smart controlling devices (Heat, Electricity)



- Auto-pilot Software in Flights, Cars

Some Famous Software fails



Ariane 5 rocket, 1996

- ❑ European space agency rocket called Ariane 5 exploded in 1996.
- ❑ Reason: Software bug
- ❑ 64-bit floating point number into a memory space that is allotted for a 16-bit integer
- ❑ Resulted in transmission of incorrect altitude data to the aircraft



Therac 25



- 6 patients lost their life due to buggy software in a machine that gives radiation therapy to cancer patients.
- Reason: Ray's condition error in this software.
- software ended up calculating more dosage of radiation than what was needed.

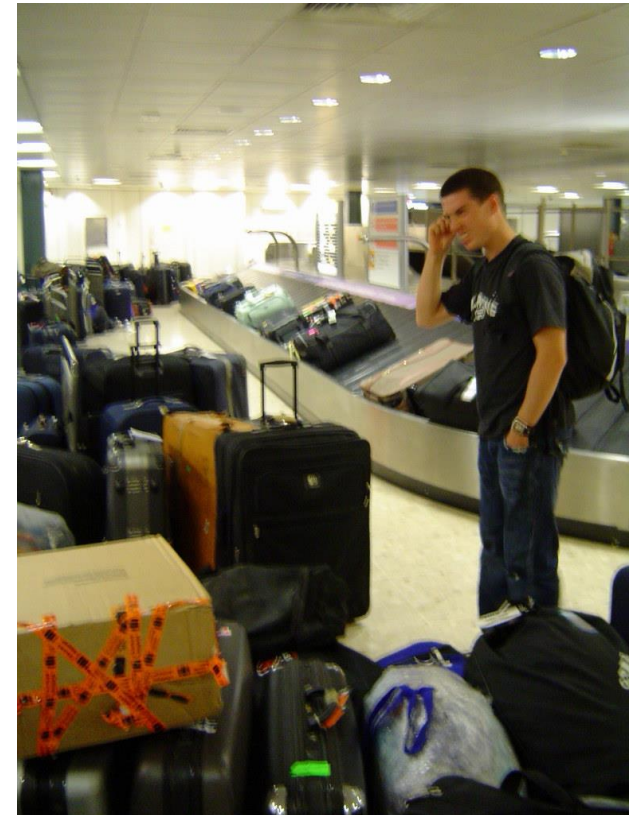
3. The Morris Worm, 1988

- A Cornell University student created a worm as part of an experiment
- ended up spreading and crashing tens of thousands of computers due to a coding error
- Graduate student Robert Tappan Morris was eventually charged and convicted.
- Morris is a professor at MIT and the worm's source code has been kept as a museum piece



4. Heathrow Terminal 5 Opening, 2008

- ❑ The problem lay with a new baggage handling system
- ❑ Performed well on test runs, but failed miserably in real-life.
- ❑ Caused massive disruptions like malfunctioning luggage belts and thousands of items being lost or sent to the wrong destinations.
- ❑ Over the next 10 days, some 42,000 bags were lost and more than 500 flights canceled, costing more than £16 million.



Why Context is important?



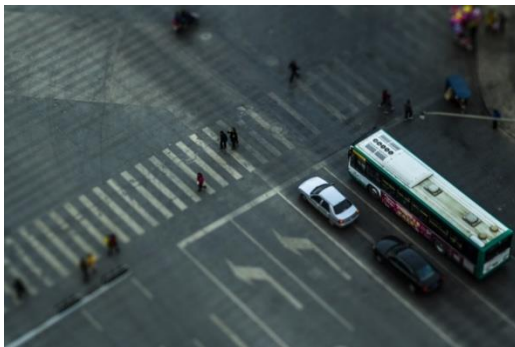
- We have various types of Software
 - System Software
 - Application Software
 - Web Applications
 - Enterprise level Software
 - Embedded Software

- Not all software systems carry the same level of **risk** and not all problems have the same impact when they occur.

Risk



- An event that has not happened yet and it may never happen.
- A potential threat for Software
- We must have the idea about the likelihood and severity of the risk.



Impact of Risk

□ Time



₹

- Money

- Business reputation



- Physical damage

Software Defect

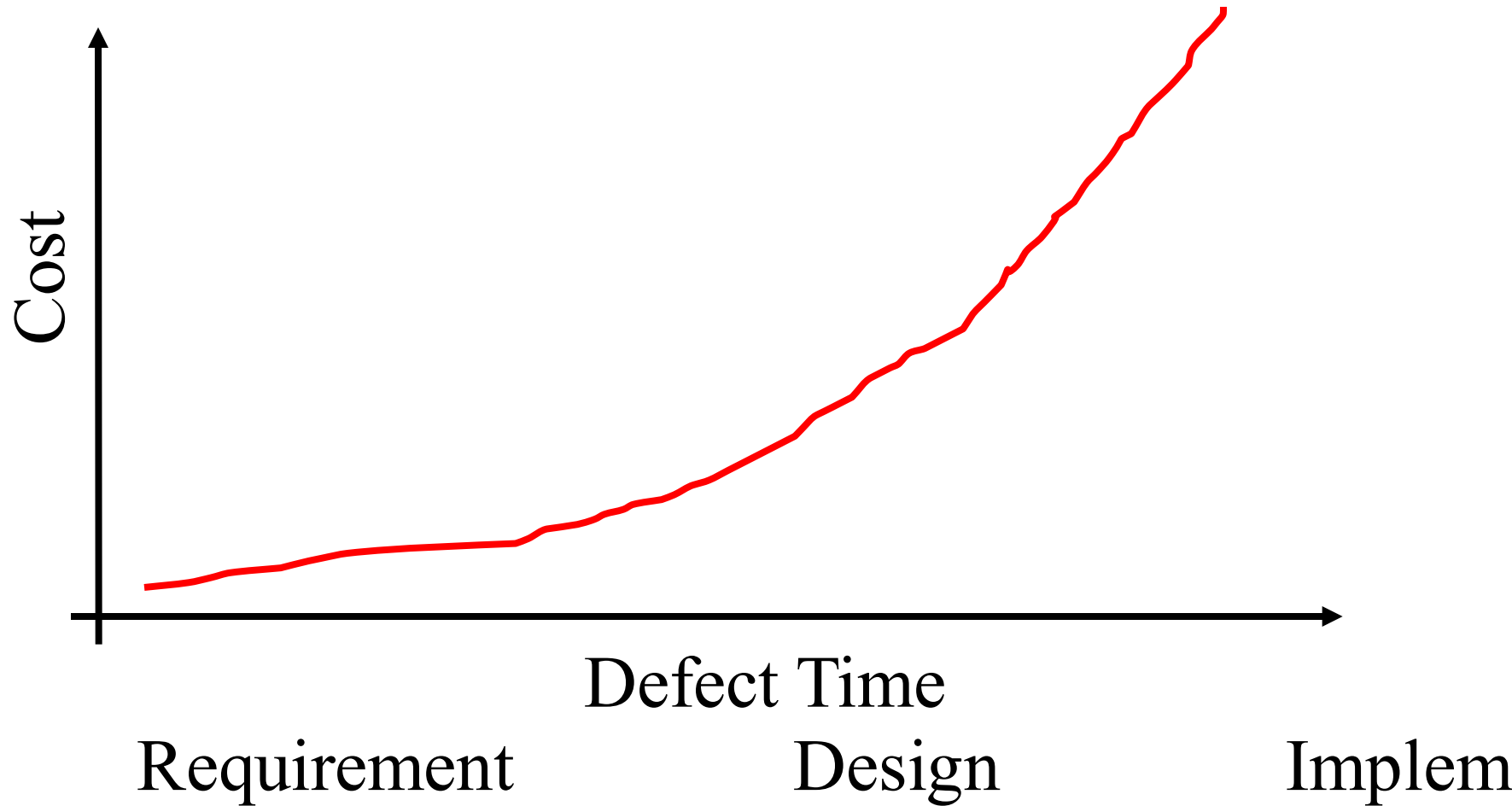
- **Errors** while designing and building the software is called **defects** or sometimes bugs or faults.
- Defects can be presented at any level in the software
 - Requirement
 - Design
 - Code
 - Test cases
- Any defect may cause the system to fail.
- Not all defects result in failures.

Sources of Software Defect



- ❑ Errors in the specification, design and implementation of the software and system
- ❑ Errors in use of the system
- ❑ Environmental conditions
- ❑ Intentional damage
- ❑ Potential consequences of earlier errors, intentional damage, defects and failures.

Cost of the Defect



Software and the Quality



- Measurement of the functional and non-functional characteristics of the Software
- A poor test may uncover few defects and presents a false sense of security.
- A well-designed test will uncover defects and we can assert that the overall level of risk of using the system has been reduced.
- When test detects the defects, the **quality** of the software system increases

Quality

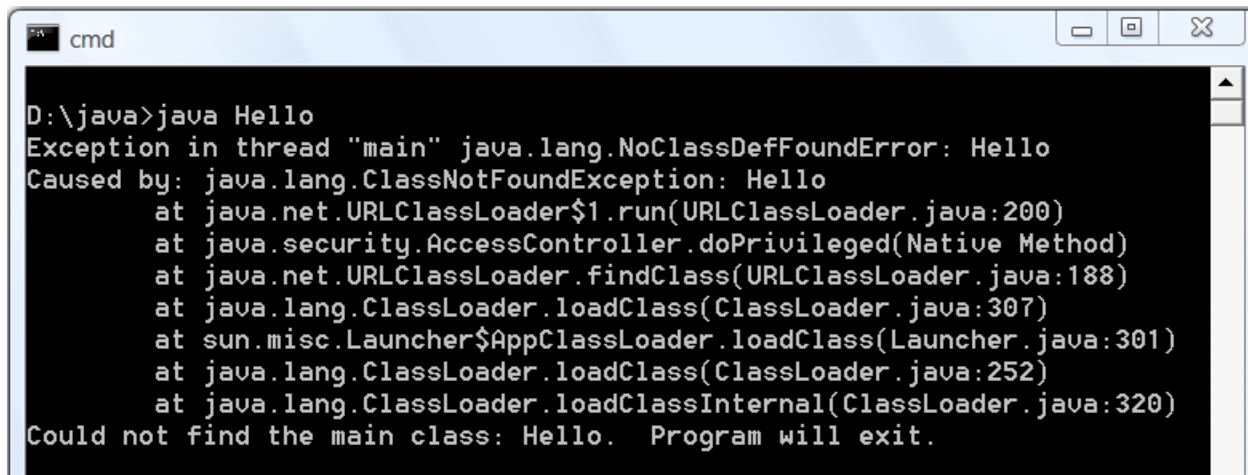


- For a Software development team
 - The software meets its defined specification, is technically excellent and has few bugs in it.
- For Customer
 - A cost effective and Timely solution of their problem.
- If the customer wants a cheap car for a 'run-about' and has a small budget then an expensive sports car or a military tank are not quality solutions, however well built they are.

Viewpoint of expectation & Quality

Viewpoint	Software	Ice-Cream
By looking Attributes of the product	LOC (lines of Code) MTTF (Mean Time to Failure) Fan-in, Fan-out etc.	Flavor, Cones, Toppings
Fitness of the product	Ask user to test the Software	This Ice-Cream is perfect for after-meal
Manufacturing process of the product	Use of a planned Testing process in SDLC (Software Development Life Cycle)	Ice-Cream is prepared with Pure milk and chocolate
Value for money	Time-boxing the Testing process	Prize is affordable, and we can keep it for X number of days.
Transcendent feelings	Working with a desired Software team. Use of a favorite Tool	XYZ company makes the best Ice-Creams

Identify root cause for better Quality

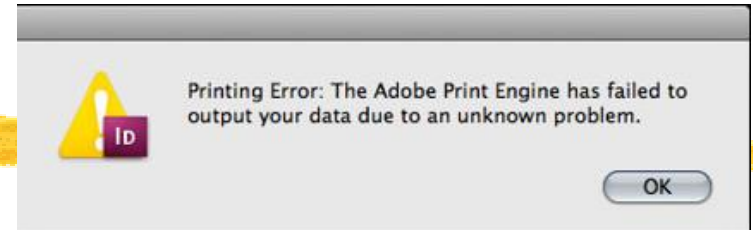
A screenshot of a Windows command prompt window titled 'cmd'. The window has a black background with white text. The text shows a command 'D:\java>java Hello' followed by an exception message: 'Exception in thread "main" java.lang.NoClassDefFoundError: Hello'. Below this, it says 'Caused by: java.lang.ClassNotFoundException: Hello' and lists several stack trace entries with line numbers. The final line states 'Could not find the main class: Hello. Program will exit.'

```
D:\java>java Hello
Exception in thread "main" java.lang.NoClassDefFoundError: Hello
Caused by: java.lang.ClassNotFoundException: Hello
    at java.net.URLClassLoader$1.run(URLClassLoader.java:200)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:188)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:307)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:301)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:252)
    at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:320)
Could not find the main class: Hello. Program will exit.
```

Root cause analysis

Example

- Printer is unable to print
- Possible causes
 - Printer runs out of supplies (ink or paper).
 - Printer driver software fails.
 - Printer room is too hot for the printer and it seizes up.
- Root cause for runs out of supplies
 - No-one is responsible for checking the paper and ink levels in the printer
 - possible root cause: no process for checking printer ink/paper levels before use.
 - Some staff don't know how to change the ink cartridges
 - possible root cause: staff not trained or given instructions in looking after the printers.
 - There is no supply of replacement cartridges or paper
 - possible root cause:
no process for stock control and ordering



Black Box Testing



- ❑ Black box testing treats the system as a "black-box", so it doesn't explicitly use Knowledge of the internal structure.
- ❑ It focuses on the functionality part of the module. It is the testing based on an analysis of the specification of a piece of software without reference to its internal workings.
- ❑ The goal is to test how well the component conforms to the published requirements for the component.
- ❑ Specifically, this technique determines whether combinations of inputs and operations produce expected results.

Characteristics



It attempts to find:

- ❑ Incorrect or missing functions
- ❑ Interface errors
- ❑ Errors in data structures or external database access
- ❑ Performance errors
- ❑ Initialization and termination errors

Black-box Testing



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

Equivalence Partitioning (EP)

Equivalence Partitioning (EP) test cases for a program that solves a quadratic equation of the form: $ax^2 + bx + c = 0$

1. Identify Input Equivalence Classes: Inputs are: a, b, c.

For a valid quadratic equation, $a \neq 0$.

Class A (Based on coefficient 'a'):

□	Class Description	Valid/Invalid
□	A1 $a = 0$ (not quadratic)	Invalid
□	A2 $a > 0$	Valid
□	A3 $a < 0$	Valid
	= $b^2 - 4ac$):	
	Class B (Based on discriminant D	

□	Class Description	Valid
□	B1 $D > 0 \rightarrow$ two distinct real roots	Valid
□	B2 $D = 0 \rightarrow$ one repeated real root	Valid
□	B3 $D < 0 \rightarrow$ complex/imaginary roots	Valid

□ Class C (Input types):

□	Class Description	
□	C1 Valid numeric inputs	
□	C2 Non-numeric/empty/null inputs (invalid)	

Equivalence Partitioning (EP)

2. Generate Test Cases Using Equivalence Partitions

Test Case 1: Invalid – Not a Quadratic Equation

| TC | Input (a,b,c) | Partition Covered | Expected Result|

| TC1 | a=0, b=5, c=3 | A1 | Invalid input: Not a quadratic equation |

Test Case 2: Valid – Two Distinct Real Roots ($D > 0$)

| TC | Input | Partition | Explanation | Expected Output |

| TC2 | a=1, b=5, c=6 | A2 + B1 + C1 | $D = 25 - 24 = 1 > 0$ | Two real roots: -2, -3 |

Test Case 3: Valid – One Real Root ($D = 0$)

| TC | Input | Partition | Explanation | Expected Output |

| TC3 | a=1, b=2, c=1 | A2 + B2 + C1 | $D = 4 - 4 = 0$ | One repeated root: -1 |

Test Case 4: Valid – Complex Roots ($D < 0$)

| TC | Input | Partition | Explanation | Expected Output |

| TC4 | a=1, b=2, c=5 | A2 + B3 + C1 | $D = 4 - 20 = -16$ | Complex roots |

Test Case 5: Valid – Negative 'a' but still quadratic**

| TC | Input | Partition | Expected Output |

| TC5 | a=-2, b=4, c=1 | A3 + B1 | Two real roots |

Test Case 6: Non-numeric input (Invalid)

| TC | Input | Partition | Expected |

| TC6 | a="x", b=2, c=1 | C2 | Error: Invalid numeric input |

Test Case 7: Empty or Null Input

| TC | Input | Partition | Expected |

| TC7 | a="", b=2, c=3 | C2 | Error: Missing input |

Test Case 8: Large numbers (valid numeric)

| TC | Input | Partition | Expected |

| TC8 | a=1000, b=2000, c=1 | A2 + C1 | Valid result with correct root calculation |

Summary of All EP Test Cases

TC No.	a	b	c	EP Classes Covered	Outcome
TC1	0	5	3	A1	Invalid (not quadratic)
TC2	1	5	6	A2, B1	Two real roots
TC3	1	2	1	A2, B2	One repeated real root
TC4	1	2	5	A2, B3	Complex roots
TC5	-2	4	1	A3, B1	Two real roots
TC6	"x"	2	1	C2	Invalid input
TC7	""	2	3	C2	Invalid input
TC8	1000	2000	1	C1	Valid large-number calculation

White Box Testing



- White box testing involves looking at the structure of the code.
- All internal components should be adequately exercised.
- It is the testing based on an analysis of internal workings and structure of a piece of software.
- It is also known as Structural Testing / Glass Box Testing / Clear Box Testing.

White-box Testing

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

Characteristics



white box testing tends to involve the coverage of the specification in the code.

- Aims to establish that the code works as designed.
- Examines the internal structure and implementation of the program.
- Target specific paths through the program.
- Needs accurate knowledge of the design, implementation and code.

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.

Statement coverage criterion



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

Example



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

Euclid's GCD computation algorithm



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

Branch Coverage



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

Branch Coverage



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

Stronger testing

- Test cases are a superset of a weaker testing:
 - 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:
- $\{(x=3, y=3), (x=3, y=2), (x=4, y=3), (x=3, y=4)\}$

Condition Coverage



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

Example



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

Branch testing

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

Branch testing



- Condition testing
 - stronger testing than branch testing:
- Branch testing
 - stronger than statement coverage testing.

Condition coverage



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

Condition coverage



- 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?

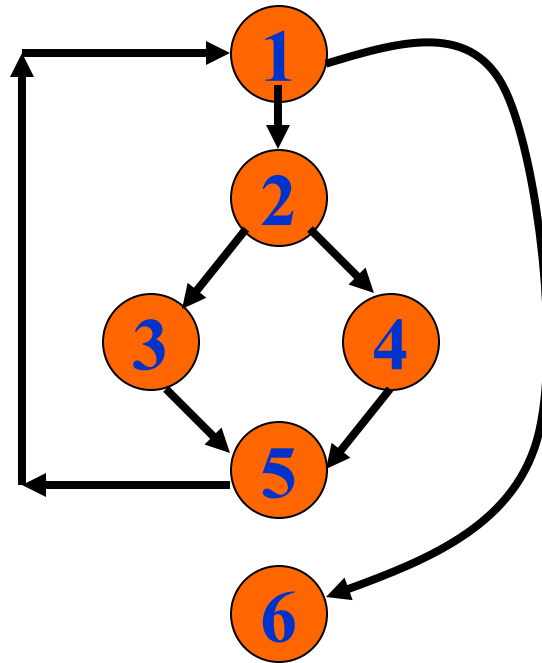
- An edge from one node to another node exists:
 - if execution of the statement representing the first node
 - can result in transfer of control to the other node.

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

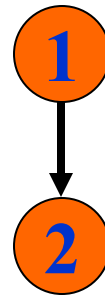


How to draw Control flow graph?

□ Sequence:

□ 1 $a=5;$

□ 2 $b=a*b-1;$



How to draw Control flow graph?

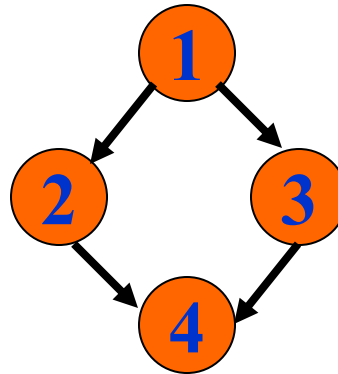
□ Selection:

□ 1 if($a > b$) then

□ 2 $c = 3;$

□ 3 else $c = 5;$

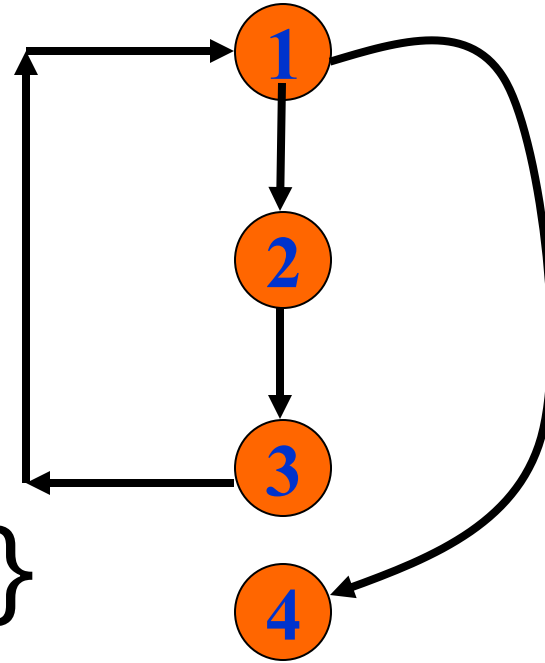
□ 4 $c = c * c;$



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

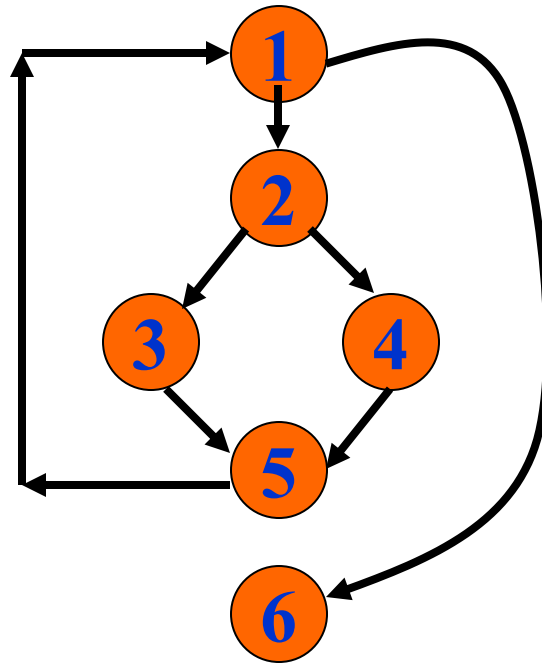


- An upper bound:
 - for the number of linearly independent paths of a program
- Provides a practical way of determining:
 - the maximum number of linearly independent paths in a program.

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

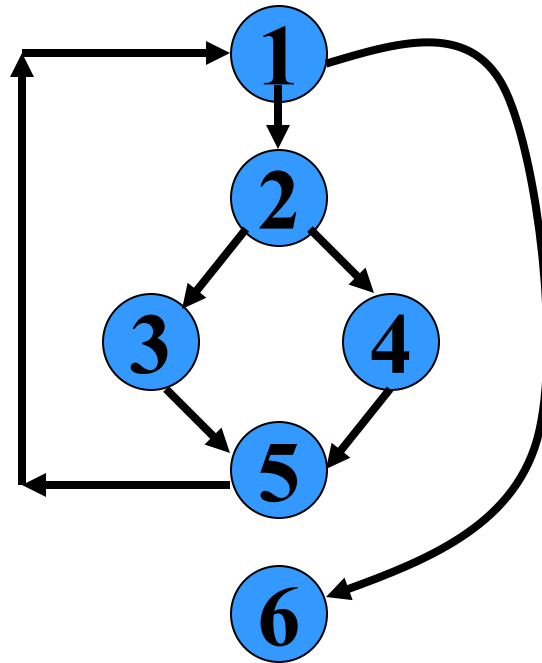
- Another way of computing cyclomatic complexity:
 - inspect control flow graph
 - determine number of bounded areas in the graph
- $V(G) = \text{Total number of bounded areas} + 1$

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



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

Cyclomatic complexity



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

Path testing



- The tester proposes:
 - an initial set of test data using his experience and judgement.

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



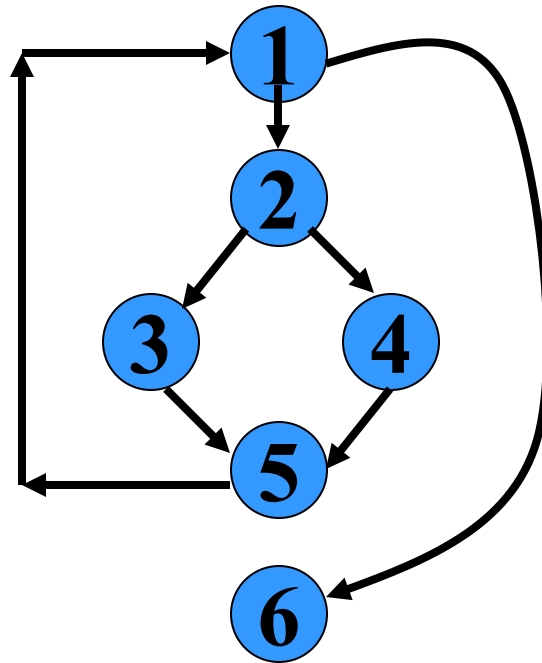
- Draw control flow graph.
- Determine $V(G)$.
- Determine the set of linearly independent paths.
- Prepare test cases:
 - to force execution along each path.

Example



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

Example Control Flow Diagram



Derivation of Test Cases

□ Number of independent paths:
3

□ 1,6 test case (x=1, y=1)

□ 1,2,3,5,1,6 test case(x=1, y=2)

□ 1,2,4,5,1,6 test case(x=2,
y=1)

An interesting application of cyclomatic complexity



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

Cyclomatic complexity



- Cyclomatic complexity of a program:
 - also indicates the psychological complexity of a program.
 - difficulty level of understanding the program.

Cyclomatic complexity



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

Automated Testing Tools

- Mercury Interactive
 - Quick Test Professional: Regression testing
 - WinRunner: UI testing
- IBM Rational
 - Rational Robot
 - Functional Tester
- Borland
 - Silk Test
- Compuware
 - QA Run
- AutomatedQA
 - TestComplete

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.