# White Box Testing

# White box testing

- Black box testing focuses only on functionality
  - What the program does; not how it is implemented
- White box testing focuses on implementation
  - Aim is to exercise different program structures with the intent of uncovering errors
- Is also called *structural testing*
- Various criteria exist  for test case design
- Test cases  have to be selected to satisfy coverage criteria

# Types of structural testing

- Control flow based criteria
  - looks at the coverage of the control flow graph
- Data flow based testing
  - looks at the coverage in the definition-use graph
- Mutation testing
  - looks at various mutants of the program

# Control flow based criteria

- Considers the  program as control flow graph
  - Nodes represent code blocks – i.e. set of statements always executed together
  - An edge (i,j) represents a possible transfer of control from i to j
- Assume a start node and an end node
- A path is a sequence of nodes from start to end

# Statement Coverage Criterion

- Criterion: Each statement is executed at least once during testing
- I.e. set of paths executed during testing should include all nodes
- Limitation: does not require a decision to evaluate to false if no else clause
- E.g.: abs (x) : if ( x>=0) x = -x; return(x)
  - The set of test cases {x = 0} achieves 100% statement coverage, but error not detected
- Guaranteeing 100% coverage not always possible due to possibility of unreachable nodes

# Statement Coverage

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

# Statement Coverage

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

# Statement Coverage Criterion

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

# Example

```
int f1(int x, int y){
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=3,y=3),(x=4,y=3), (x=3,y=4)}
  - All statements are executed at least once.

# Branch coverage

- Criterion: Each edge should be traversed at least once during testing
- i.e. each decision must evaluate to both true and false during testing
- Branch coverage implies stmt coverage
- If multiple conditions in a decision, then all conditions need not be evaluated to T and F

# Branch Coverage

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

# Stronger Testing

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

# Example

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

# Example

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

# Condition Coverage

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

# Example

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

# Branch Testing

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

# Branch testing

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

# Condition coverage

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

# Path Coverage

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

# Path Coverage-Based Testing

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

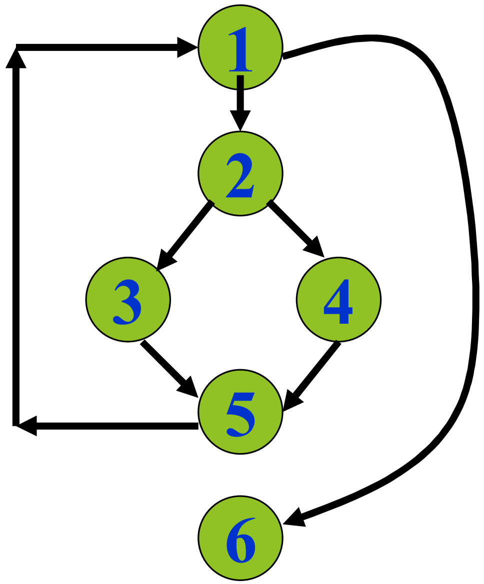
## How to Draw Control Flow Graph?

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

# Example

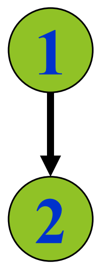
```
int f1(int x,int y){
while (x != y){
   if (x>y) then
      x=x-y;
   else y=y-x;
}
return x;        }
```

# Example Control Flow Graph



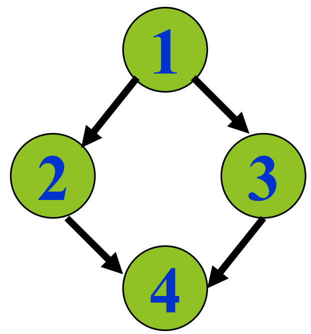
# How to draw Control flow graph?

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



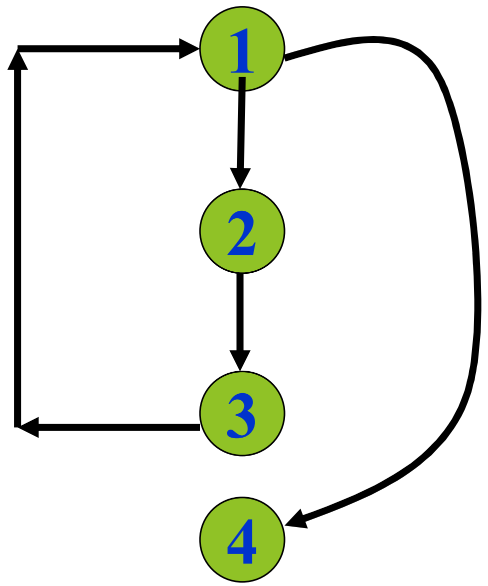
# How to draw Control flow graph?

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



# How to draw Control flow graph?

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



# Path

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

# Linearly Independent Path

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

# Independent path

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

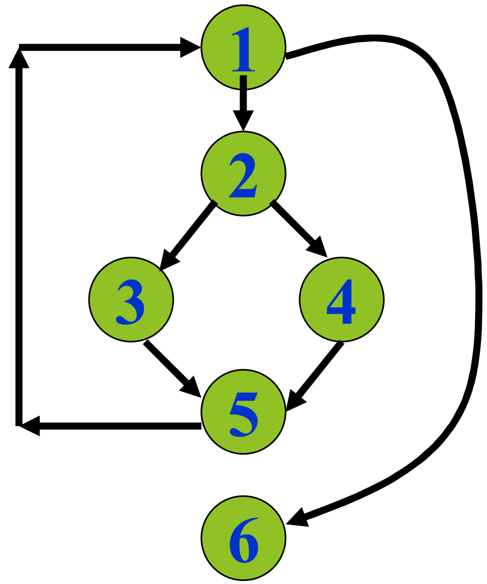
# McCabe's Cyclomatic Metric

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

# McCabe's Cyclomatic Metric

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

# Example Control Flow Graph

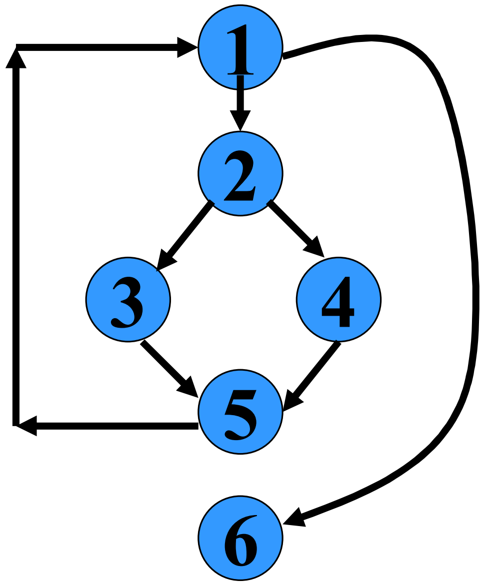



Cyclomatic complexity = 7-6+2 = 3.

# Cyclomatic Complexity

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

# Example Control Flow Graph



# Example

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

# Cyclomatic complexity

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

# Cyclomatic Complexity

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

# Cyclomatic complexity

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

# Cyclomatic Complexity

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

# Path Testing

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

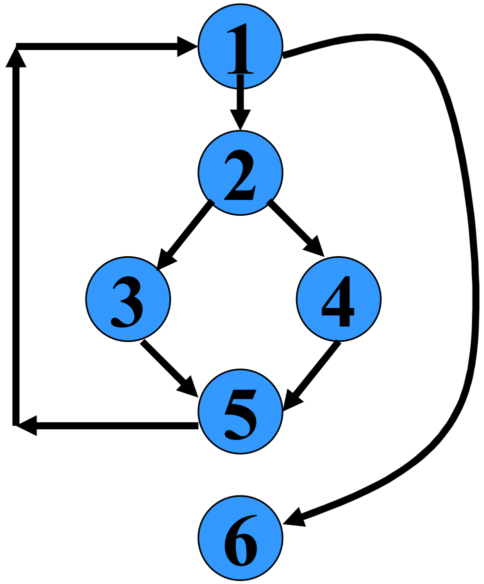
# Derivation of Test Cases

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

# Example

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

# Example Control Flow Diagram



# Derivation of Test Cases

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

## An interesting application of cyclomatic complexity

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

# Cyclomatic Complexity

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

# Cyclomatic Complexity

- From maintenance perspective,
  - limit cyclomatic complexity
    - of modules to some reasonable value.
  - Good software development organizations:
    - restrict cyclomatic complexity of functions to a maximum of 10 or so.

# Control flow based…

- There are other criteria too - path coverage, predicate coverage, cyclomatic complexity based, ...
- None is sufficient to detect all types of defects (e.g. a program missing some paths cannot be detected)
- They provide some quantitative handle on the breadth of testing
- More used to evaluate the level of testing rather than selecting test cases

# Tool support and test case selection

- Two major issues for using these criteria
  - How to determine the coverage
  - How to select test cases to ensure coverage
- For determining coverage - tools are essential
- Tools also tell which branches and statements are not executed
- Test case selection is mostly manual - test plan is to be augmented based on coverage data

# In a Project

- Both functional and structural should be used
- Test plans are usually determined using functional methods; during testing, for further rounds, based on the coverage, more test cases can be added
- Structural testing is useful at lower levels only; at higher levels ensuring coverage is difficult
- Hence, a combination of functional and structural at unit testing
- Functional testing (but monitoring of coverage) at higher levels

# Comparison

| | Code Review | Structural Testing | Functional Testing |
|---|---|---|---|
| Computational | M | H | M |
| Logic | M | H | M |
| I/O | H | M | H |
| Data handling | H | L | H |
| Interface | H | H | M |
| Data defn. | M | L | M |
| Database | H | M | M |

# System Testing

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

# Alpha Testing

- **System testing is carried out by the test team within the developing organization.**
- **Its main purpose is to discover software bugs that were not found before.**
- **At the stage of alpha testing, software behavior is verified under real-life conditions by imitating the end-users' actions.**
- **The alpha phase includes the following testing types: smoke, sanity, integration, systems, usability, UI (user interface), acceptance, regression, and functional testing.**
- **If an error is detected, then it is immediately addressed to the development team.**
- **Alpha testing helps to discover issues missed at the stage of requirement gathering.**
- **The alpha release is the software version that has passed alpha testing.**
- **The next stage is beta testing.**

# Beta Testing

- System testing performed by a select group of friendly customers.
- All the testing activities are performed outside the organization that has developed the product.
- Beta checking helps to identify the gaps between the stage of requirements gathering and their implementation.
- Beta testing can be called pre-release testing.
- It can be conducted by a limited number of end-users called beta testers before the official product delivery.
- The main purpose of beta testing is
  - to verify software compatibility with different software and hardware configurations, types of network connection, and to get the users' feedback on software usability and functionality.

# Beta Testing

- There are two types of beta testing:
  - **open beta** is available for a large group of end-users or to everyone interested
  - **closed beta** is available only to a limited number of users that are selected especially for beta testing.
- During beta testing, end users detect and report bugs they have found.
- The product version that has passed beta testing is called beta release.
- After the beta phase comes gamma testing.

# Gamma Testing

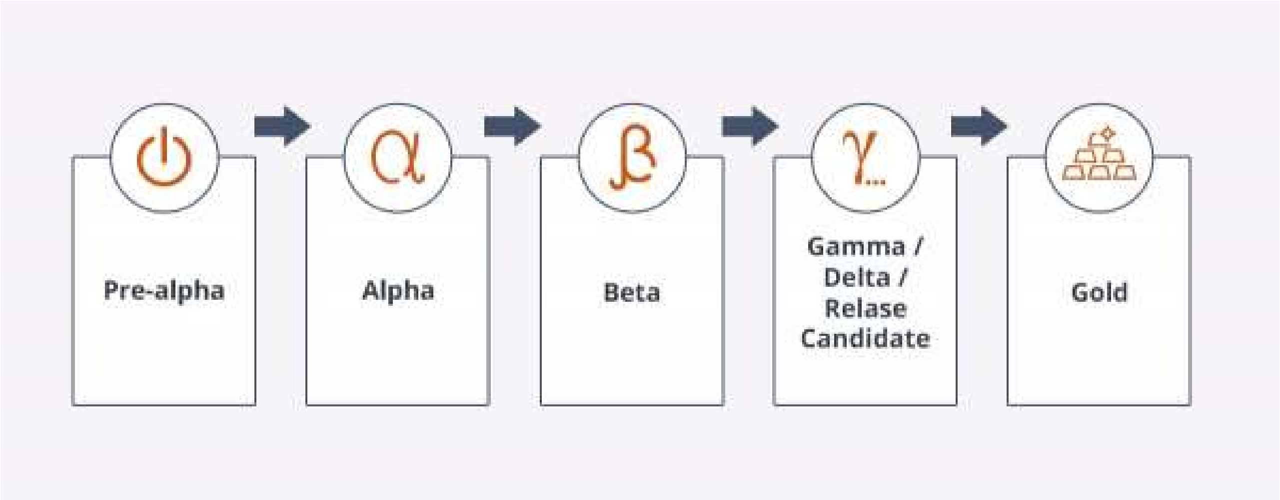
- Gamma testing is the final stage of the testing process conducted before software release.
- It makes sure that the product is ready for market release according to all the specified requirements.
- Gamma testing focuses on software security and functionality.
- But it does not include any in-house Quality Assurance (QA) activities.
- During gamma testing, the software does not undergo any modifications unless the detected bug is of a high priority and severity.
- Only a limited number of users perform gamma testing, and testers do not participate.
- The checking includes the verification of certain specifications, not the whole product.
- Feedback received after gamma testing is considered as updates for upcoming software versions.
- But, because of a limited development cycle, gamma testing is usually skipped.

# Acceptance Testing

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

# software release life cycle

- **pre-alpha stage** that consists of activities done before the QA and testing phase.



# Metrics

# Data

- Defects found are generally logged
- The log forms the basic data source for metrics and analysis during testing
- Main questions of interest for which metrics can be used
  - How good is the testing that has been done so far?
  - What is the quality or reliability of software after testing is completed?

# Coverage Analysis

- Coverage is very commonly used to evaluate the thoroughness of testing
- This is not white box testing, but evaluating the overall testing through coverage
- Organization sometimes have guidelines for coverage, particularly at unit level (say 90% before checking code in)
- Coverage of requirements also checked – often by evaluating the test suites against requirements

# Reliability Estimation

- High reliability is an important goal to be achieved by testing
- Reliability is usually quantified as a probability or a failure rate or mean time to failure
  - $R(t) = P(X > t)$
  - MTTF = mean time to failure
  - Failure rate- failures per unit time
- For a system reliability can be measured by counting failures over a period of time
- Measurement often not possible for software as due to fixes reliability changes, and with one-off, not possible to measure

# Reliability Metrics

- Reliability metrics are used to quantitatively express the reliability of the software product.
- Mean Time to Failure (MTTF)
  - **MTTF** is described as the time interval between the two successive failures.
  - The time units are entirely dependent on the system & it can even be stated in the number of transactions.
  - To measure **MTTF**, we can evidence the failure data for n failures. Let the failures appear at the time instants $\mathbf{t_1, t_2 \ldots\ldots t_n}$.
  - **MTTF can be calculated as**

$$\sum_{i=1}^{n} \frac{t_{i+1} - t_i}{(n-1)}$$

# Reliability Metrics

- Mean Time to Repair (MTTR)
  - Once failure occurs, some-time is required to fix the error.
  - **MTTR** measures the average time it takes to track the errors causing the failure and to fix them.
- Mean Time Between Failure (MTBR)
  - We can merge **MTTF** & **MTTR** metrics to get the MTBF metric.

    **MTBF = MTTF + MTTR**
  - Thus, an **MTBF** of 300 denoted that once the failure appears, the next failure is expected to appear only after 300 hours.
  - In this method, the time measurements are real-time & not the execution time as in **MTTF**.

# Reliability Estimation

- Simple method of measuring reliability achieved during testing
  - Failure rate, measured by no of failures in some duration
- for using this for prediction, assumed that during this testing software is used as it will be by users
- Execution time is often used for failure rate, it can be converted to calendar time

# Reliability Estimation…

- Sw reliability estimation models are used to model the failure followed by fix model of software
- Data about failures and their times during the last stages of testing is used by these model
- These models then use this data and some statistical techniques to predict the reliability of the software
- Software reliability growth models are quite complex and sophisticated

# Defect removal efficiency

- Basic objective of testing is to identify defects present in the programs
- Testing is good only if it succeeds in this goal
- Defect removal efficiency of a QC activity = % of present defects detected by that QC activity
- High DRE of a quality control activity means most defects present at the time will be removed

# Defect removal efficiency …

- DRE for a project can be evaluated only when all defects are know, including delivered defects
- Delivered defects are approximated as the number of defects found  in some duration after delivery
- The *injection stage* of a defect is the stage in which it was introduced in the software, and *detection stage* is when it was detected
  - These stages are typically logged for defects
- With injection and detection stages of all defects, DRE for a QC activity can be computed

# Defect Removal Efficiency …

- DREs of different QC activities are a process property - determined from past data
- Past DRE can be used as expected value for this project
- Process followed by the project must be improved for better DRE

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