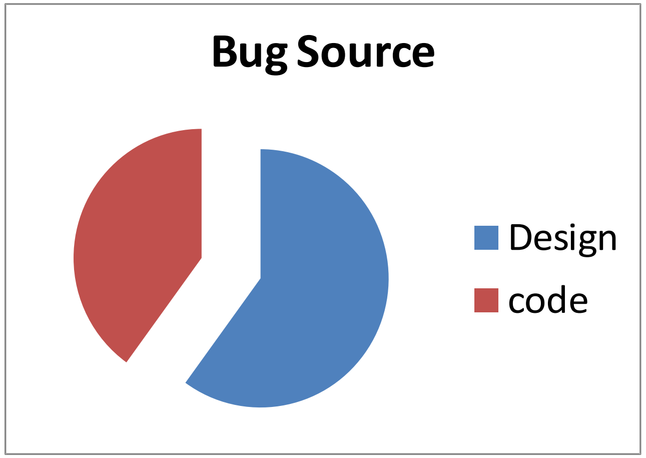
Software Testing

Why test?

- Ariane 5 rocket self-destructed 37 seconds after launch
- Reason: An undetected bug in control software
- Total Cost: over $7 billion
- Total time : 10 years

Facts on error

- Experienced programmers on the average make:
  - 50 errors per 1000 lines of source code
- Extensively tested software contains:
  - About 1 error per 1000 lines of source code.
- Error distribution:
  - 60% spec/design, 40% implementation.



Challenges in testing programs

- Input space too large for most practical programs
  - Exhaustive testing not feasible, not even automatically

- Testing requires maximum effort among all development phases → more job opportunities

- Testing getting more complex with time
  - larger and more complex programs, newer programming paradigms
  - Even after through testing, a practical software cannot be guaranteed to be error-free

Terminologies

- Failure: manifestation of an error / defect / bug
  - Mere presence of an error may not lead to a failure
- Test case: a triplet [I, S, O]
  - I: data to be input to the system
  - S: state of the system at which the data will be input
  - O: expected output of the system
  - Each test case typically tries to establish correct working of some functionality / program elements
- Test suite: set of all test cases to be used

Terminologies

Verification: are we building the product right?

Definition : The process of evaluating work-products (not the actual final product) of a development phase to determine whether they meet the specified requirements for that phase.

Evaluation Items : Plans, Requirement Specs, Design Specs, Code, Test Cases

Validation: are we building the right product?

Definition : The process of evaluating software during or at the end of the development process to determine whether it satisfies specified business requirements.

Evaluation Items : The actual product/software

Design of Test Cases

- ❑ 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

Design of Test Cases

Consider following example:

Find the maximum of two integers x and y.

The code has a simple programming error:

```
int findMax(int x, int y) {
    if (x>y) max = x;
    else max = x;
    return max;
}
```

- 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 two main approaches to design test cases:
  - Black-box approach
  - White-box (or glass-box) approach

Black-Box Testing

- Test cases are designed using only functional specification of the software:
  - Without any knowledge of the internal structure of the software.

- For this reason, black-box testing is also known as functional testing.

White-box Testing

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

Design of test cases

- Exhaustive testing impractical for any non-trivial system

- Required: an optimal test suite
  - Reasonable number of test cases
  - Uncovers as many errors as possible
  - <u>Each test case should aim to detect different errors</u>

- Approaches: Black box testing, White box testing – Both are complimentary to each other

Bit more ...

"Program testing can be used to show the presence of bugs, but never to show their absence !"

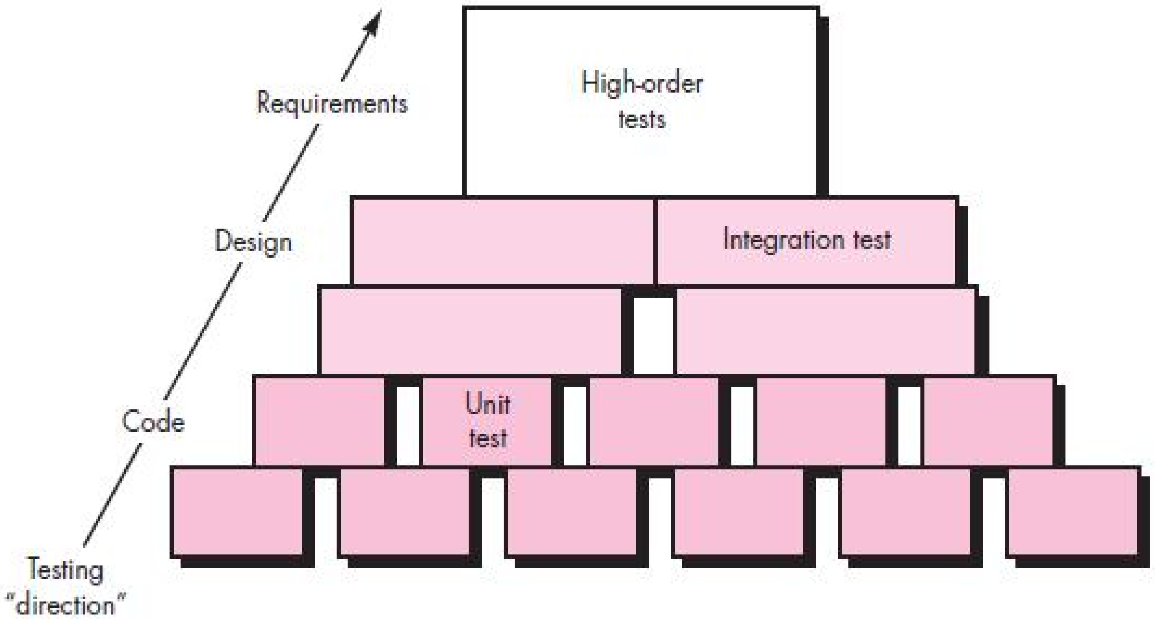
– Dijkstra

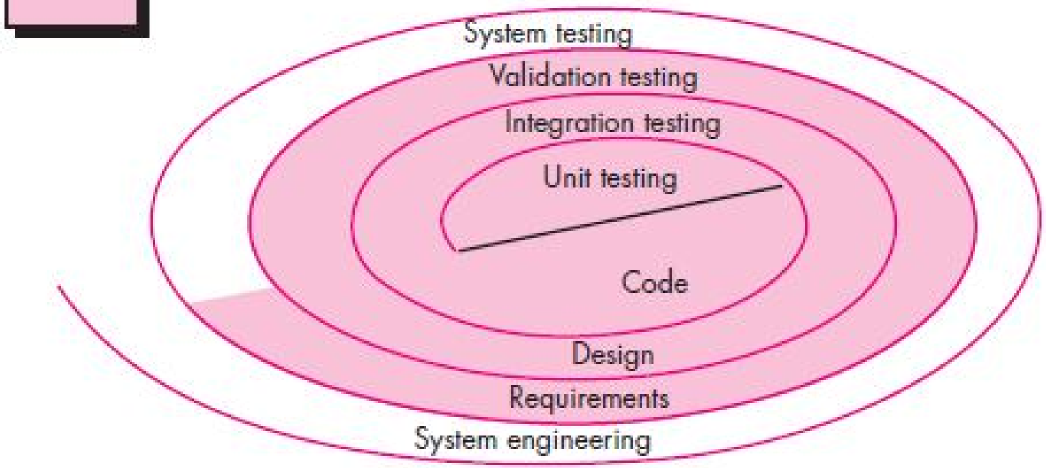
Types of testing

Types of testing

- Unit testing
  - Each module tested in isolation (right after coding)
- Integration testing
  - After all modules have been coded and unit-tested, ...
  - Modules integrated in steps according to integration plan
  - Partially integrated system tested at each integration step
- System testing / Validation Testing
  - Does the fully developed system meet its requirements? (mentioned in SRS)

Software Testing steps and strategy





Unit testing

- To test a particular module M, needed
  - Procedures belonging to other modules that M calls
  - Non-local data structures accessed by M
  - Procedures in other modules that call the procedures in M

- Other modules may not have been coded yet

- Stub procedure
  - A dummy procedure having same I/O parameters as a given procedure, but highly simplified implementation

Unit tests should be FIRST

- **Fast**: have to be run very frequently
- **Independent**: no test should depend on others, so can run any subset in any order
- **Repeatable**: should get same result if run repeatedly
- **Timely**: test cases developed about the same time as the code is written

- Above properties enable automation of testing

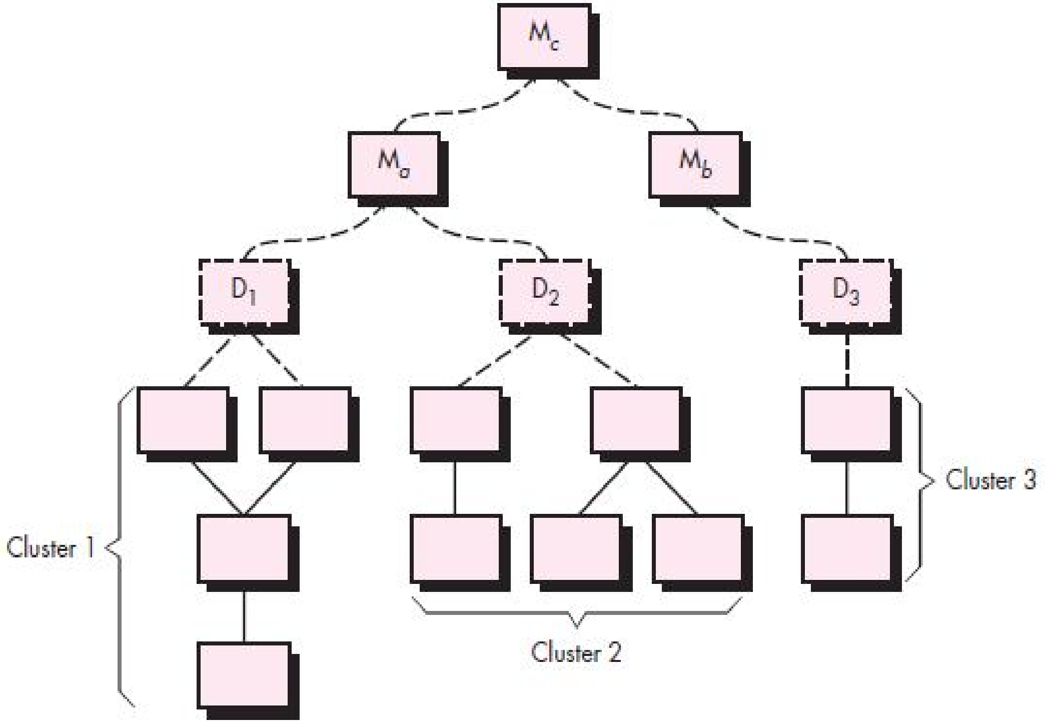
Integration testing

- Integration plan
  - Specifies the steps and the order in which modules are combined to realize the full system
  - Developed by examining **structure chart** (*module dependency graph*)
  - Different approaches: big bang, top-down, bottom-up, mixed

- Big bang approach
  - All modules put together in a single step and tested
  - Difficult to localize errors, practical only for very small systems

Integration testing – Bottom-up approach

Lower-level subsystems tested individually, then combined to form higher-level subsystem



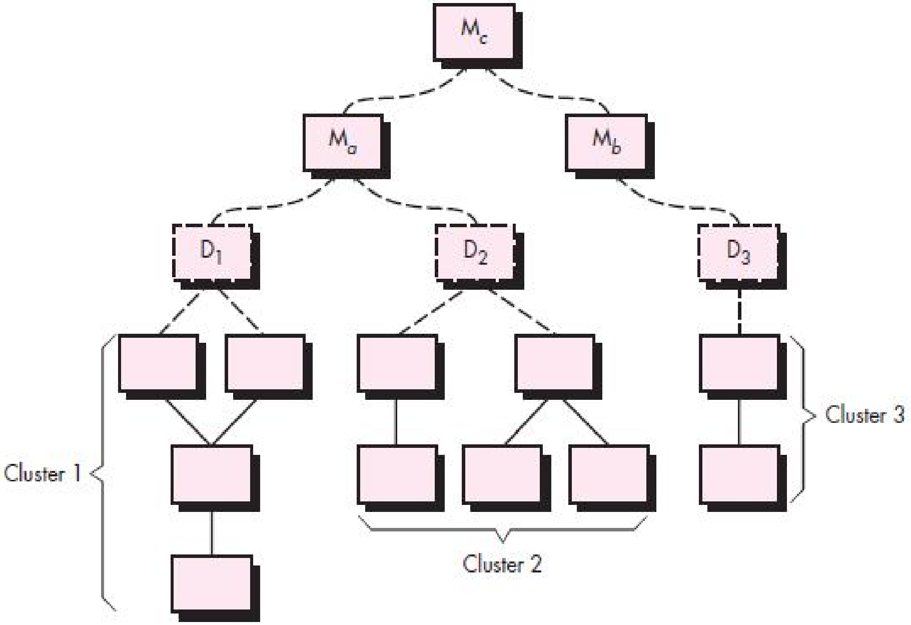
IT : Bottom-up approach - Steps

1. Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software subfunction.

2. A *driver* (a control program for testing) is written to coordinate test case input and output.

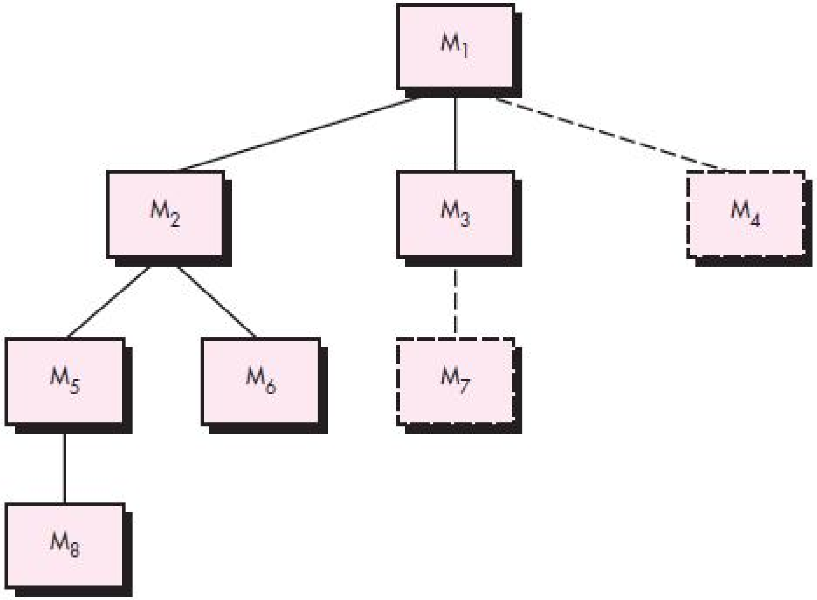
3. The cluster is tested.

4. Drivers are removed and clusters are combined moving upward in the program structure.



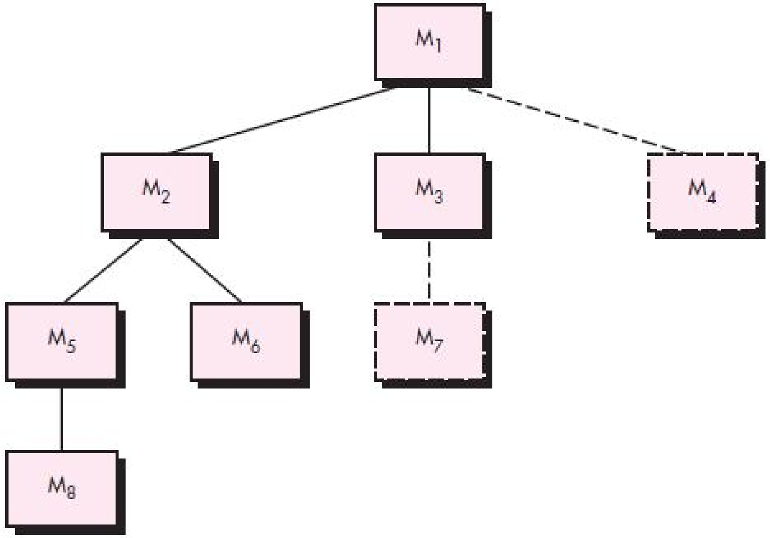
Integration testing – Top-down approach

- Testing starts with main module, few lower-level modules integrated at each step and tested
- Stubs used to simulate the lower-level modules that are called by the modules currently being tested



IT : Top-down approach - Steps

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.



2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.

3. Tests are conducted as each component is integrated.

4. On completion of each set of tests, another stub is replaced with the real component.

5. Regression testing may be conducted to ensure that new errors have not been introduced.

Integration testing

- Mixed or sandwiched approach

Integrate and test modules as they become available

Most commonly used

Sometime called "*Sandwich Testing*"

System testing / Validation Testing

- Objective – validate a fully developed system against its requirements

- Alpha testing
  - carried out by the test team within the developing organization
- Beta testing
  - performed by a select group of friendly customers
- Acceptance testing
  - performed by the customer himself to determine whether to accept or reject the delivered product

Design of test cases

White box testing

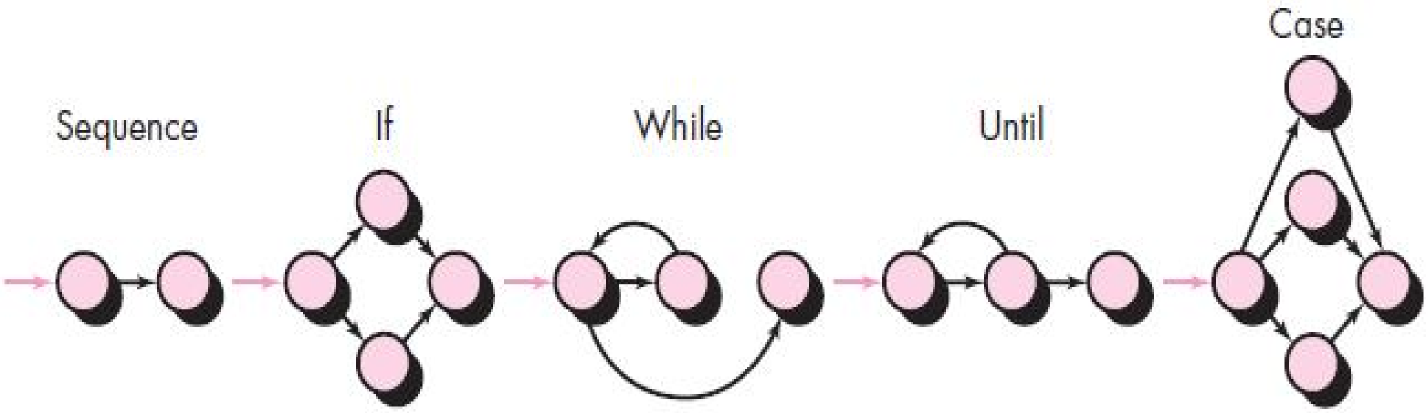
- Knowledge about the internal structure of software used to develop test cases
- Also called structural testing
- White box testing strategies
  - Coverage-based: Design test cases so that certain program elements are covered (executed)
    - Statement coverage, Branch coverage, Path coverage etc
  - Fault-based: Design test cases that focus on discovering certain specific category of faults
  - Data flow testing: Derive test cases based on definition and use of variables in a program

White box testing strategies

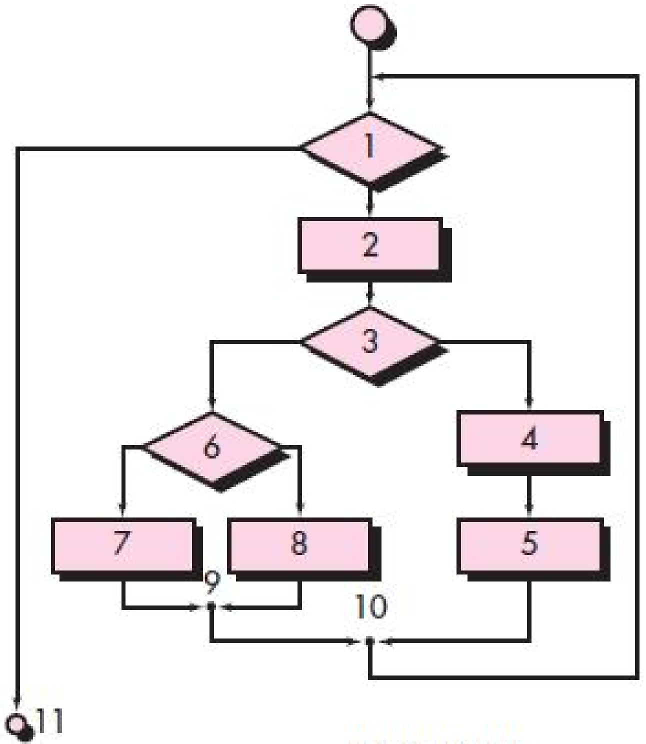
- Path coverage / Basis path
  - Design test cases such that all linearly independent paths in the Control Flow Graph(CFG) of the program are executed at least once
  - Path is from the starting node to a terminal node in CFG
  - Linearly Independent Path (LIP) – a path that introduces at least one new edge that is not included in any other linearly independent path.
  - Prepare test cases that will force execution of each path in the Basis set (set of all LIPs).
  - May not be easy to derive LIPs from CFG of complex programs

Control Flow Graph

- A CFG describes how the control flows though the program
- Each structured construct has a corresponding flow graph symbol

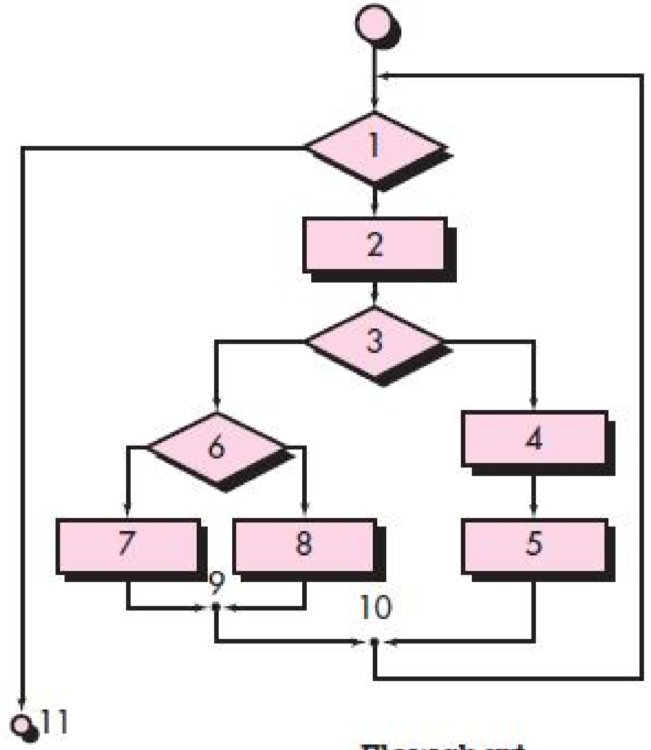


Drawing CFG

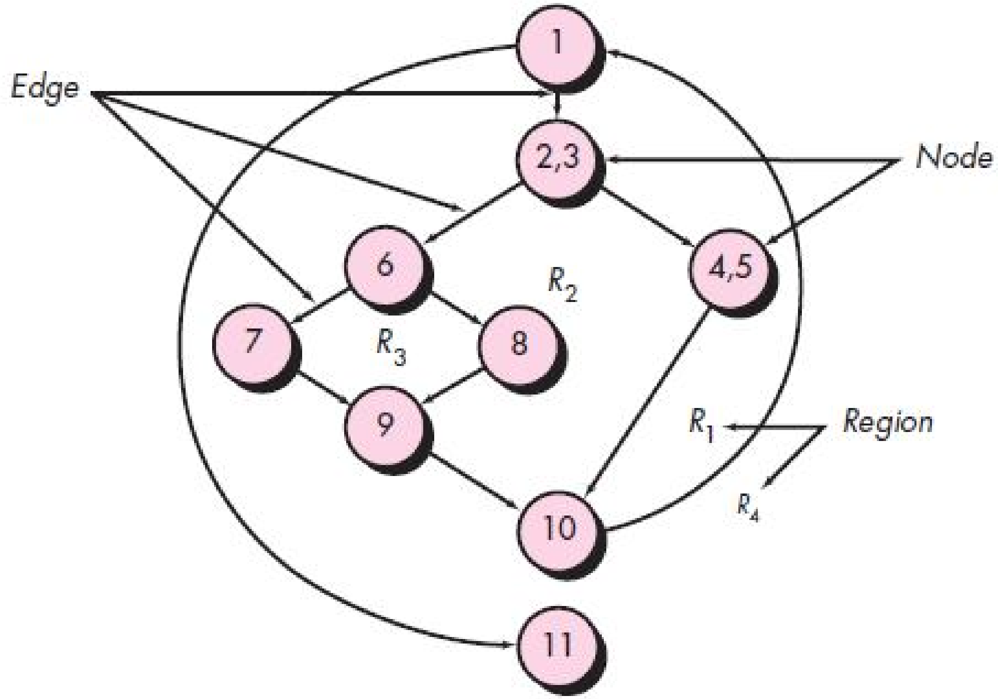


Flowchart

Drawing CFG

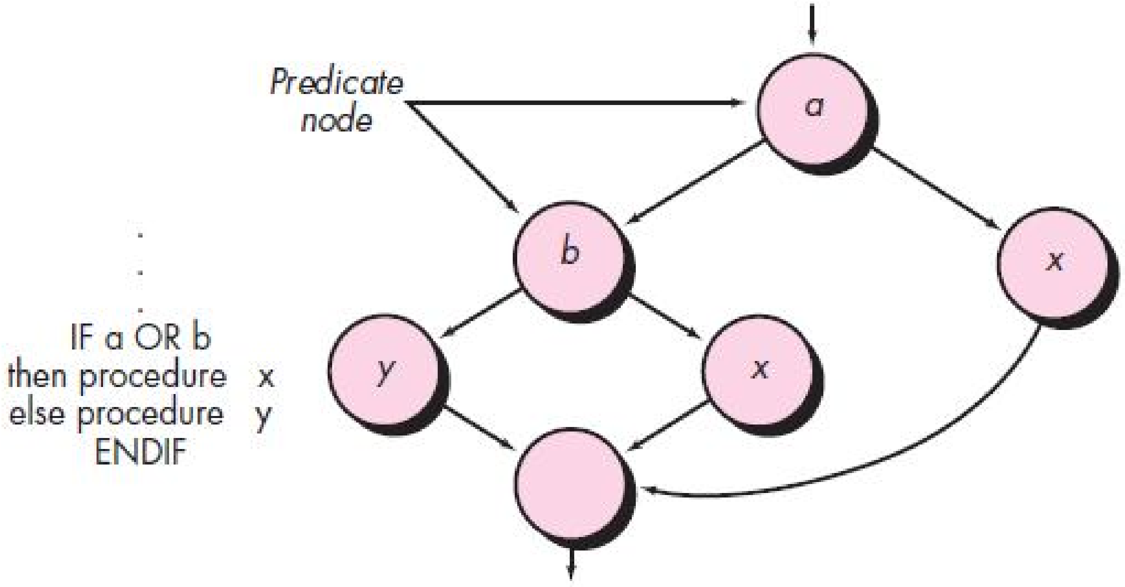


Flowchart



Flow graph

Drawing CFG with Compound Logic

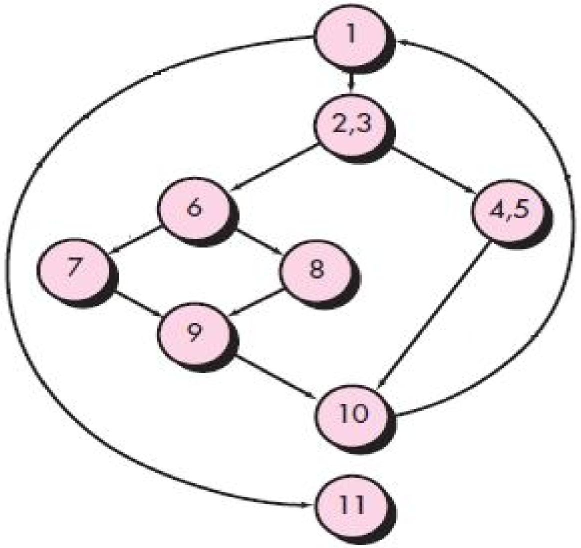


```
.
.
.
IF a OR b
then procedure  x
else procedure  y
ENDIF
```

Predicate node : Each node that contains a condition is called a *Predicate node* and is characterized by two or more edges emanating from it.

Linearly Independent Path (LIP)

Any path through the program that introduces **at least one new set of processing statements** or **a new condition** → An independent path must move along **at least one edge that has not been traversed before** the path is defined.



Linearly Independent Path (LIP)

Any path through the program that introduces **at least one new set of processing statements** or **a new condition** → An independent path must move along **at least one edge that has not been traversed before** the path is defined.

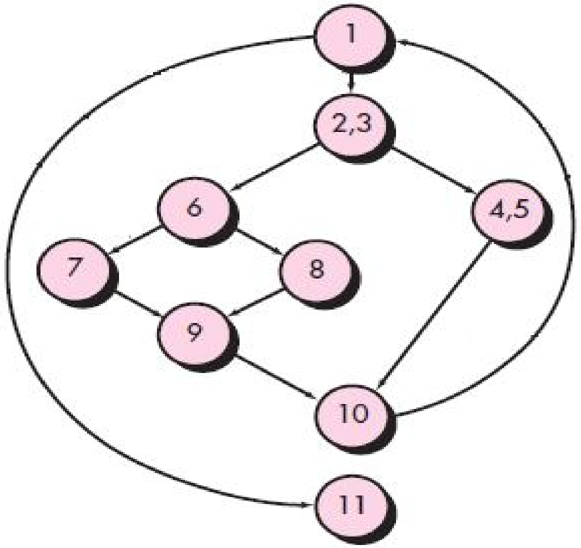
Path 1: 1-11

Path 2: 1-2-3-4-5-10-1-11

Path 3: 1-2-3-6-7-9-10-1-11

Path 4: 1-2-3-6-8-9-10-1-11

<ins>Note that each new path introduces a new edge.</ins>



The path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 is not considered to be an independent path

Because it is simply a combination of already specified paths and does not traverse any new edges

Estimating number of LIPs

- MeCabe's Cyclomatic complexity metric C
  - Gives upper bound for the number of LIPs
  - Also known as structural complexity metric

Computing C : Three ways of determining C

Given a CFG G, where N: #nodes, E: #edges, P : # decision statements / # predicate nodes

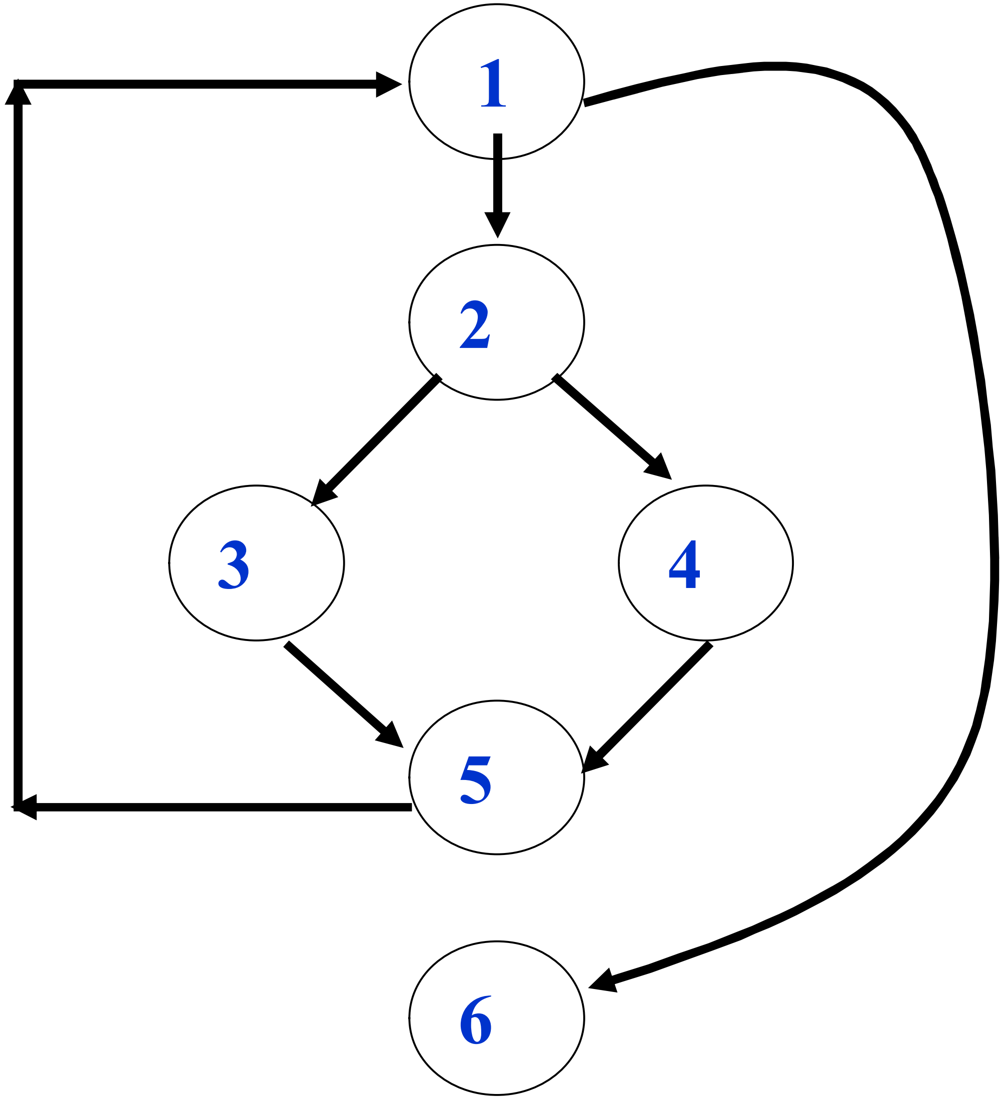
$C = E - N + 2$ [*ex : 11 – 9 + 2 = 4*]

Example of code and CFG

```
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; }
```

Example of code and CFG

```
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; }
```



Linearly independent paths:

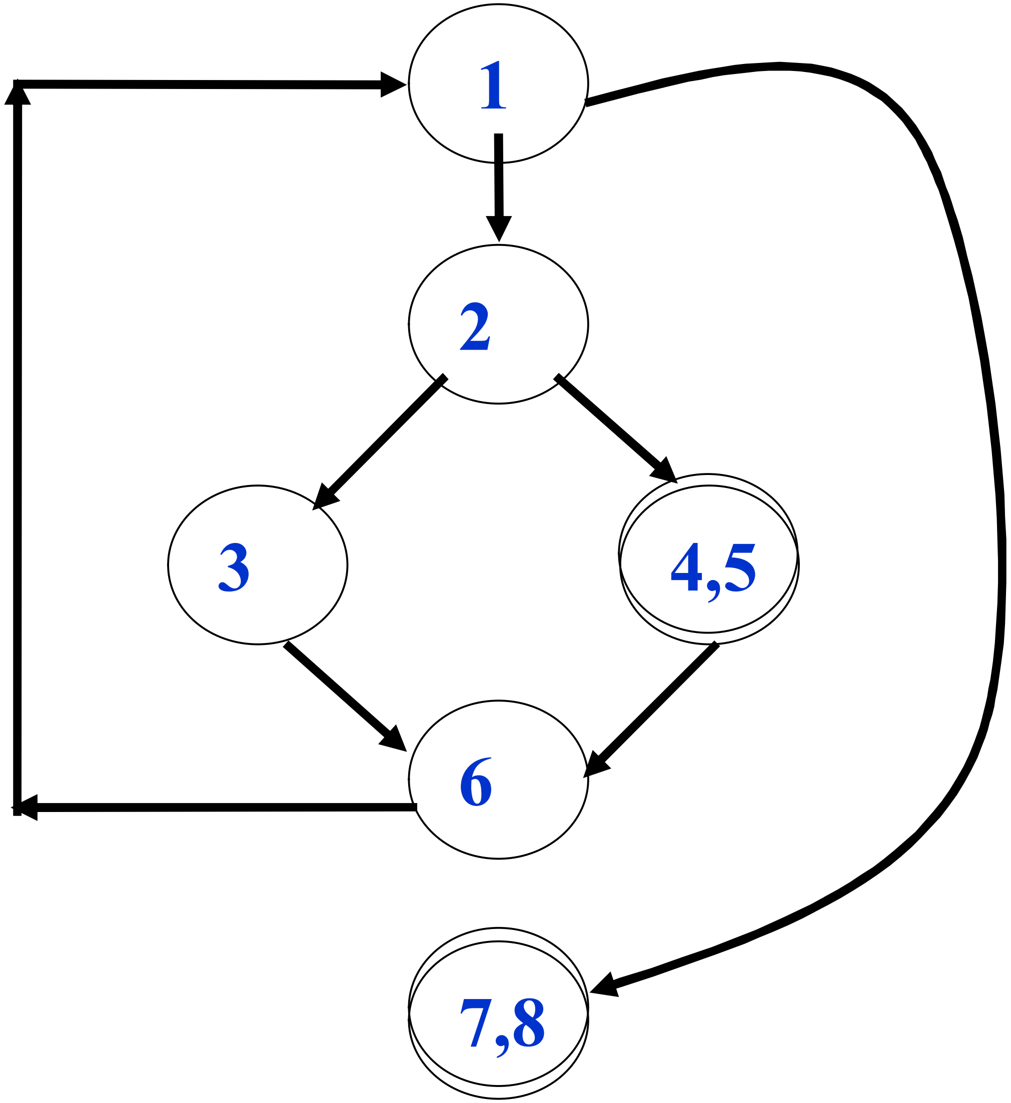
1, 6

1,2,3,5,1,6

1,2,4,5,1,6

Example of code and CFG

```
int f1(int x,int y){
1  while (x != y){
2    if (x>y)
3      x=x-y;
4    else
5      y=y-x;
6  }
7  return x;
8}
```



Linearly independent paths:

1, 7,8

1,2,3,6,1,7,8

1,2,4,5,6,1,7,8

Path coverage

- Cyclomatic complexity of a program
  - Provides an estimate of the number of test cases to be designed to guarantee coverage of all LIPs
  - Does not make it any easier to derive the test cases – gives an indication of the minimum number of testcases required for path coverage.
- Practical path coverage testing
  - Testers propose initial set of test cases using their experience and judgment
  - Dynamic program analyzer used to estimate percentage of LIPs covered by this test suite
  - If less than 90% of the estimated number of LIPs covered, expand test suite

Use of Cyclomatic complexity

- Experimental studies indicate C is related with
  - number of errors existing in the code
  - time required to find and correct the errors
- Generally accepted that
  - C is an indication of the psychological complexity or level of difficulty in understanding a program
- Advised: limit C of a module to some reasonable value
  - Good software development organizations limit C to ~10

Black box testing

- Test cases designed using only functional specification of the module / software
    - No knowledge of the internal structure of the software used
    - Also known as functional testing

- Two approaches to black box testing
    - Equivalence class partitioning
    - Boundary value analysis
    - Both approaches are complementary to each other

Equivalence class partitioning

- Input space partitioned into equivalence classes such that …
  - Program behaves similarly to every input belonging to the same equivalence class
  - Test with just one representative value from each equivalence class

- How to determine the equivalence classes?
  - If input space is a range of values
  - If input space is an enumerated set of values

Boundary value analysis

- Some typical programming errors occur at boundaries of equivalence classes
  - Failure to notice special processing required at the boundaries, e.g. use of < instead of <=

- Select test cases at the boundaries of different equivalence classes

Another black box testing strategy

- Testing would be a lot easier if test cases could be automatically generated from requirements

- Cause-effect graphing
  - Methodology to automatically derive test cases from the functional requirements in SRS document
  - Work done at IBM
  - Details not being discussed

Test summary report

- Document that describes the testing process
  - Generated towards the end of testing phase

- For each subsystem, a summary of the tests which have been applied to this subsystem
  - Number of tests applied, how many successful or unsuccessful

Software Reliability

Software reliability

- Reliability: probability of a product working correctly over a given period of time
  - Reliability is an important attribute that determines quality of the product
  - Customer may want quantitative estimation of reliability before buying software
- Software reliability difficult to measure accurately
  - No simple relationship between observed system reliability and number of errors

Difficulty in measuring reliability

- 90-10 rule: 90% of total execution time spent in executing only 10% of the instructions in program
  - Most used 10% - core of the program
  - Reliability improvement due to correction of a bug depends on location of bug (in core or non-core part)
- Perceived reliability depends to a large extent on operation profile
  - If parts having no error most frequently used, perceived reliability will be high
  - Different users may use different parts of the software → reliability is observer-dependent

Difficulty in measuring reliability

- Software reliability keeps on changing throughout the life of the product
  - Each time a bug is detected and corrected, reliability may increase or decrease

- Different categories of software products have different reliability requirements
  - Reliability requirement for a software may be stated in SRS

Reliability metrics

- Help to quantitatively express reliability
- ROCOF: rate of occurrence of failure
  - Observe product in operation over a time interval and note total number of failures in the interval
- POFOD: probability of failure on demand
  - Likelihood of the system failing when service requested
- Availability: how likely is the system to be available for use over a given period of time

Reliability metrics

- MTTF: mean time to failure
  - Average time between two successive failures
  - Only run-time considered, not time to fix the error, reboot time, etc

- MTTR: mean time to repair
  - Average time it takes to fix faults after failure observed

- MTBF: mean time between failures
  - MTTF + MTTR

Failure classes

- Previously discussed reliability metrics do not consider the consequences / severity of the failures
- Failure classes
  - Cosmetic – only minor irritations, do not lead to incorrect functioning / results
  - Transient – occur only for certain input values while invoking a function of the system
  - Permanent – occur for all inputs while invoking a function
  - Recoverable – when failure occurs, system recovers with or without operator intervention
  - Unrecoverable – system restart needed after failure

Measuring reliability: Statistical testing

- Objective: determine reliability of software product
  - Not aimed at discovering errors
- Step 1: determine operation profile of product
  - Different users use system in different ways
  - Formally, probability distribution of the input / operations
- Step 2: generate set of test data corresponding to operational profile
- Step 3: apply test data to product, record failures, time of failures, etc
- Must continue till a statistically significant number of errors have been observed

Statistical testing

- Advantages
  - Thoroughly tests the core (most likely to be used)
  - Gives more accurate estimate of reliability than many other methods
- Disadvantages – difficult to perform
  - No automatic way of generating operation profile
  - Statistical uncertainty – need to continue testing until a statistically significant number of faults are observed

Program analysis tools

Program analysis tools

- Automated tools to facilitate testing
  - Input: source code / executable code
  - Output: important characteristics of the code, e.g. size, complexity, adequacy of commenting, …

- Two types of tools
  - Static analysis tools
  - Dynamic analysis tools

Static analysis tool

- Assess properties of a code without executing it
- E.g. analyze source code to
  - Evaluate whether coding standards have been adhered to, adequacy of commenting
  - Check for errors such as uninitialized variables, variables declared but never used, mismatch between formal parameters and function arguments
- Cannot evaluate dynamic memory references
- Results often summarized in Kiviat chart
  - Polar chart showing cyclomatic complexity, number of source lines, % of comment lines, Halstead's metrics, …

Dynamic analysis tools

- Dynamic analysis tools
  - Program is executed and its behavior recorded
  - Produces reports such as adequacy of test cases, e.g. structural coverage that was achieved by a certain test suite

Debugging

Debugging

- Once errors have been identified, need to identify precise location and fix

- Debugging guidelines
  - Requires thorough understanding of program design
  - May sometimes require full redesign of the system
  - Fix the error, not the error symptoms
  - An error correction may introduce more errors
  - After every round of error-fixing, regression testing must be carried out

Debugging techniques

- Brute-force debugging
  - Use print statements for intermediate values
  - Least efficient

- Use symbolic debugger
  - Early debuggers let you only see values from a program dump
  - Modern debuggers: single stepping, set breakpoints to check values of variables, ...

Debugging techniques

- Backtracking
  - Beginning at the statement where an error symptom has been observed, ...
  - Source code is traced backwards until the error is discovered
  - As number of source code lines to be traced back increases, ...
    - number of potential backward paths increases ...
    - becomes unmanageable

Debugging techniques

- Program slicing
  - Similar to backtracking, but search space reduced by defining slices
  - Slice defined for a particular variable at a particular statement – set of source lines which can influence the value of the variable at that statement

- Cause elimination method
  - Determine a list of causes which could possibly have contributed to the error symptom
  - Tests conducted to eliminate each
  - Related technique – software fault tree analysis

EXTRA

Types of Integration Testing

- Smoke testing
  - The intent is to uncover “showstopper” errors in a time-critical project
  - It may ask basic questions like "Does the program run?", "Does it open a window?", or "Does clicking the main button do anything?"
  - The process aims to determine whether the application is **so badly broken** as to make further immediate testing unnecessary.

- Regression testing
  - Running an old test suite after a system is changed or some bug is fixed
  - To ensure that no new bug has been introduced due to the change
  - Always used during incremental system development and during maintenance

System testing / Validation Testing What are tested?

- In all types of testing, two types of tests carried out

- Functionality tests
  - To check whether the system satisfies the functional requirements
  - Primarily to check whether system behaves correctly

- Performance tests
  - To check conformance with non-functional requirements

Performance tests

- Stress testing / endurance testing
  - Impose abnormal input conditions to test capabilities of software under stress
  - Input data volume, input data rate, utilization of memory, etc. beyond the designed capacity are applied

- Configuration testing
  - Test system behaviour in various hardware and software configurations

- Usability testing
  - Test user interface, e.g. display screens, error messages

Performance tests

- Compatibility testing
  - Required when the system interfaces with other systems
  - Check whether the interface functions as required
- Recovery testing
  - Check response of system to faults, loss of power, etc
- Documentation testing
  - Whether user manual, technical manuals exist, whether they are consistent, whether they are properly formatted
- Maintenance testing
  - Whether the software is easily maintainable
  - Whether diagnostic tools are supplied, which help to find problems that may arise