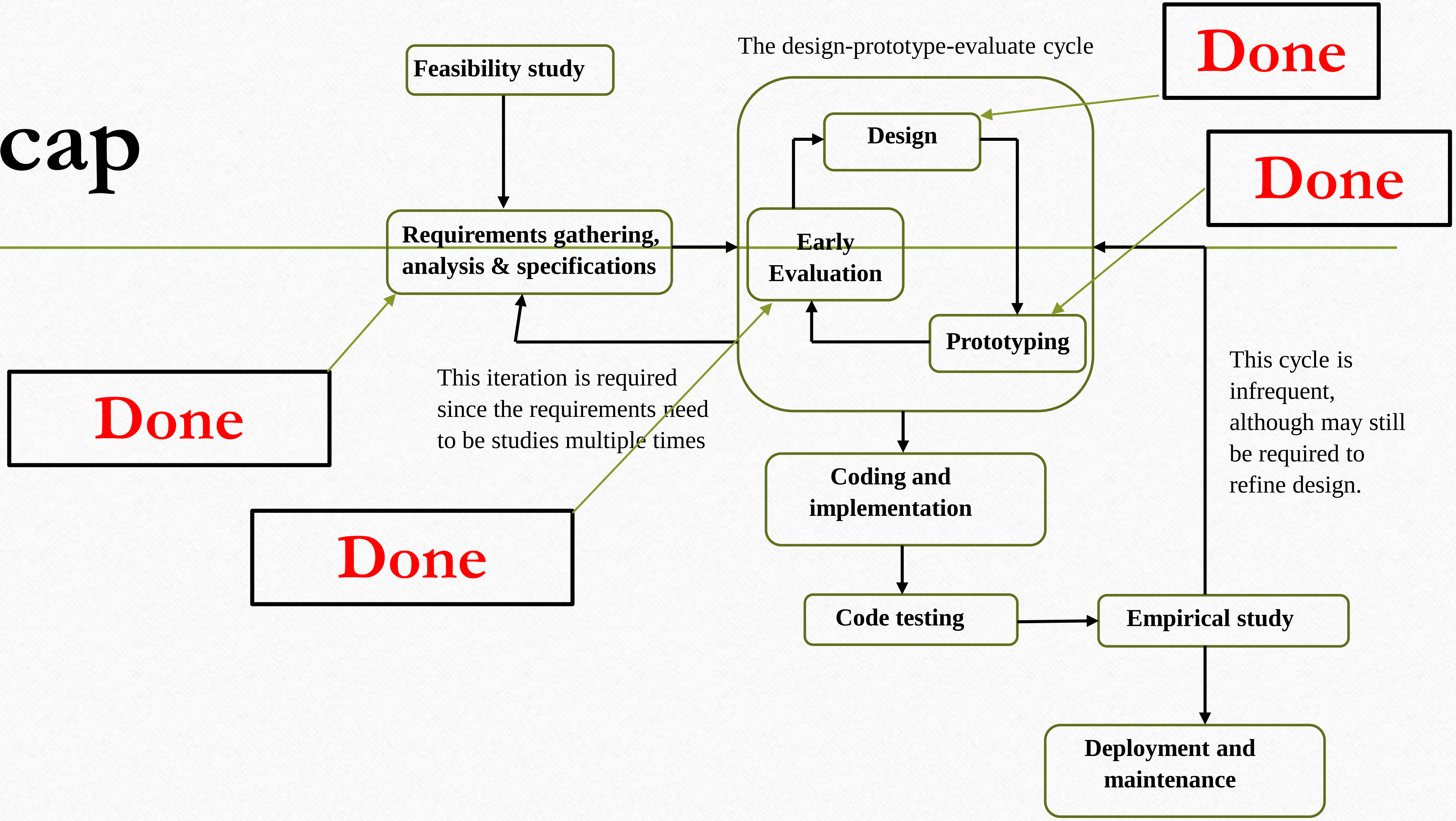
Coding & Code Testing

Dr Samit Bhattacharya

Computer Science and Engineering

IIT Guwahati

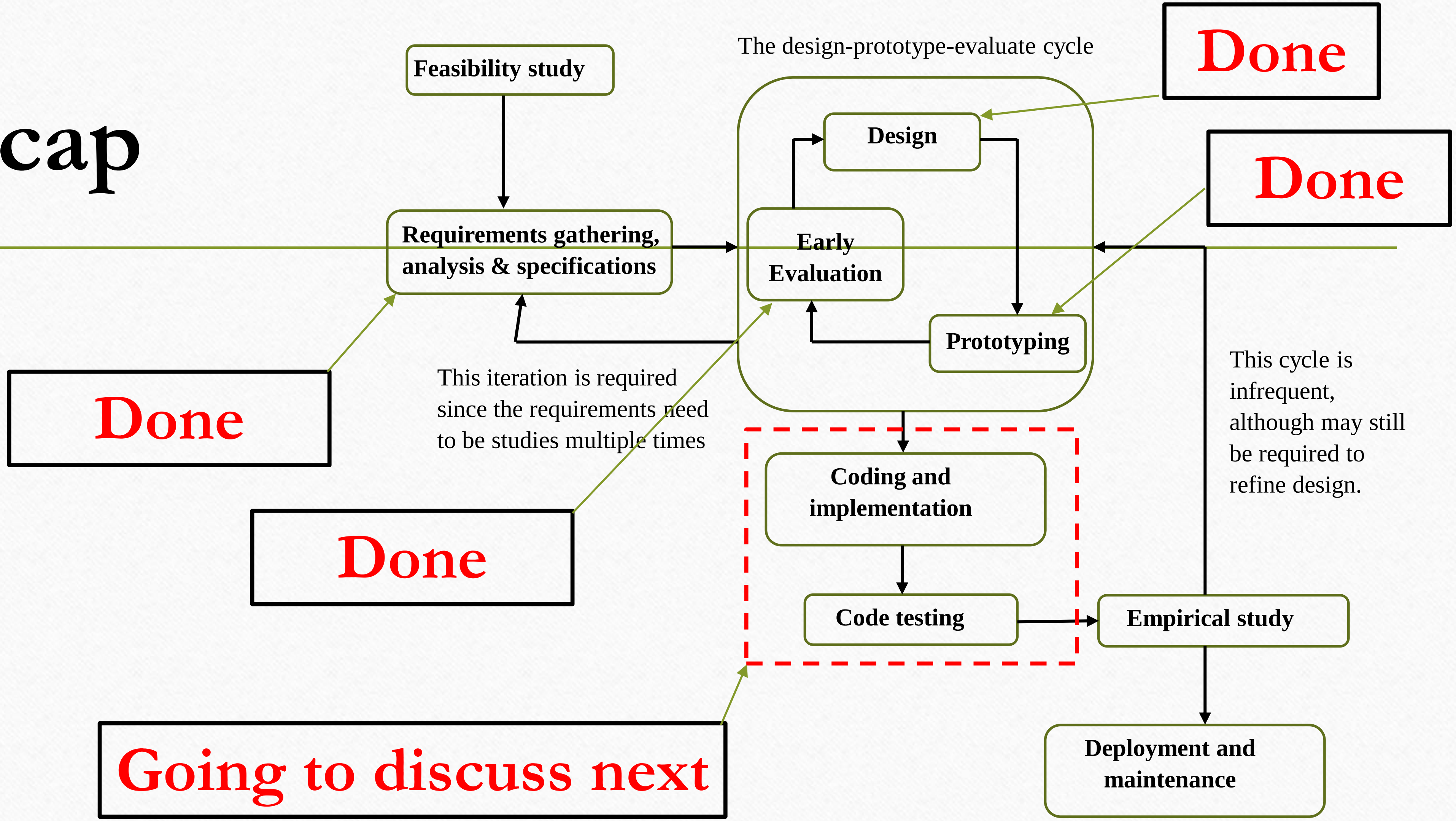
Recap


The design-prototype-evaluate cycle
Done
Done
Feasibility study
Requirements gathering, analysis & specifications
Design
Early Evaluation
Prototyping
This iteration is required since the requirements need to be studies multiple times
Done
Done
Coding and implementation
Code testing
Empirical study
This cycle is infrequent, although may still be required to refine design.
Deployment and maintenance

Next?

- Writing code for implementation
- Code testing (different from prototype testing)

Recap



Feasibility study
The design-prototype-evaluate cycle
Done
Design
Done
Requirements gathering, analysis & specifications
Early Evaluation
Prototyping
Done
This iteration is required since the requirements need to be studies multiple times
This cycle is infrequent, although may still be required to refine design.
Done
Coding and implementation
Code testing
Empirical study
Going to discuss next
Deployment and maintenance

Code Writing

Code Writing

- Let's start with an example
- Quickly write a program to ADD two numbers (only the function)
  - Takes two integers as input
  - Produces the sum as output

Code Writing

```
int f(int a, int b){
int s = 0;
s = a+b;
return (s);
}
```

Is/are there any problem(s) with this code?

Code Writing

```
int f(int a, int b){
int s = 0;
s = a+b;
return (s);
}
```

Function name (not meaningful)

variable name (not meaningful)

No indentation

Things to Remember

Coding Standards/Guidelines

- It helps to follow standard coding practices/guidelines
  - Helps readability and understandability
  - Good for teamwork

Coding Standards/Guidelines

- Organizations may have their own guidelines/standards
- There are some general rules we can follow

Things to Remember

- Indentation
  - One of the very basic thing
  - Improves readability

Things to Remember

- Global variables
  - Naming convention (should be different than others)
  - Number (limited is better)
  - Should be easy to find (may be single/limited number of definition files)

Things to Remember

- Module header (in each file) – typical things
  - Module name
  - Date on which module created
  - Author’s name
  - Modification history
  - Module summary
  - Different functions supported, along with their input/output parameters
  - Global variables accessed/modified by the module …

Things to Remember

- Naming conventions
  - Global variable names may always start with capital letter
  - Local variable names may made up of small letters
  - Constant names may always be capital letters

Things to Remember

- Error and exception handling
  - Error conditions reporting should be standard (e.g., functions may return a 0 for error or 1 otherwise, consistently)
  - Exception handling should be must

Things to Remember

- Code should be well-documented
  - A rule of thumb - at least one ‘concise and explanatory’ comment line on average for every 3-source lines
  - Good idea to create ‘user manual’ and ‘technical manual’

Things to Avoid

DO NOT's

- Use coding style too clever or too difficult to understand
  - Many coders actually take pride in writing cryptic and incomprehensible code (e.g., writing entire code in one line)
  - Can obscure meaning and hamper understanding
  - Makes maintenance difficult

DO NOT's

- Create **obscure side effects**
  - Side effects (of a function call) - modification of parameters, modification of global variables, and I/O operations
  - Obscure side effect - not obvious from casual code examination
  - Affect code understanding (e.g., global variable changed obscurely in a called module or some file I/O performed which is difficult to infer from function name and header information)

DO NOT's

- Use identifier for multiple purposes - we often use same identifier to denote several temporary entities (e.g.., a temporary loop variable for computing and storing final result) for 'memory efficiency'
  - Variable should have descriptive name indicating purpose - not possible if used for multiple purposes (affects readability and understanding)

DO NOT's

- Write 'lengthy' functions
  - Function length should not exceed 10 lines
  - Lengthy functions usually difficult to understand (likely to have different functions coded together)
  - Likely to have larger number of bugs

DO NOT's

- Use ‘goto’ (branching) statements indiscriminately
  - Makes a program unstructured and difficult to understand

Code Testing

Testing

- TWO ways
  - Review-based
  - Execution-based

Code Review

Basics

- Relative less rigorous and quick testing method
  - Carried out after a module is successfully compiled and all syntax errors eliminated
- Cost-effective strategies for reduction in coding errors and produce high quality code

Basics

- TWO types
  - Code walkthrough (similar to cognitive walkthrough)
  - Code inspection (similar to heuristic evaluation)

Code Walkthrough

Basics

- An informal code analysis technique
- Main objective - discover algorithmic and logical errors in the code

Steps

- Few members of development team are given code to read and understand
- Each member
  - Selects **some** test cases
  - Simulates execution by hand (i.e. trace execution through each statement and function execution)
- Members note down their findings and discuss in a walkthrough meeting in presence of coder(s)

(Some) Guidelines

- Team performing walkthrough should not be either too big or too small (ideally, 3-5 members)
- Use ‘representative’ test cases
- Discussion should focus on discovery of errors and **not how to fix errors**

Code Inspection

Basics

- Aim is to discover some common types of errors caused due to oversight and improper programming (e.g., uninitialized variables)
- Also checks adherence to coding standards

Basics

- Typically companies collect statistics regarding different types of errors commonly committed by their engineers and identify type of errors most frequently committed
  - Such a list can be used during code inspection
- Similar to heuristic evaluation (evaluation with a checklist)

Some Common Errors

- Use of uninitialized variables
- Jumps into loops
- Nonterminating loops
- Incompatible assignments

Some Common Errors

- Improper storage allocation and deallocation
- Use of incorrect logical operators
- Incorrect precedence among operators
- Improper modification of loop variables

Execution-Based Testing

Code Testing

- Review-based testing informal – mostly evaluates code qualitatively
- Good for early evaluation to ‘clean up’ the code before more rigorous and formal testing is done

(Formal) Program Testing

- Consists of
  - Providing program a set of test inputs (or test cases)
  - Observing program behavior (and/or output)

(Formal) Program Testing

- If program fails to behave as expected (or output mismatch), the conditions under which failure occurs are noted for later debugging and correction

Terminology

- Test case – a triplet [I,S,O]
  - I = data input
  - S = system state at input time
  - O = expected output
- For simplicity, we will consider only the doublet [I,O]

Terminology

- Test suite - set of all test cases with which a given software is to be tested

Note

- Aim of testing - to identify ALL defects in a software product
- In practice, not possible to guarantee software is *completely error free* after testing
  - Input data domain of most software products very large
  - Not practical to test software exhaustively with respect to every possible input

Note

- Then why to go for testing at all!

Note

- Testing does expose many (most) defects (important ones) if done properly and systematically
  - Practical way of reducing defects in a system
  - Increases confidence in a developed system

Choice of Test Cases (Suite)

- Exhaustive testing impractical - possible input data values extremely large or infinite
  - We must design test suite that is of reasonable size and can uncover as many errors existing in the system as possible

Choice of Test Cases (Suite)

- Randomly selected test cases not necessarily contribute to significance of test suite - they need not detect additional defects not already detected by other test cases
  - Number of test cases not indication of **testing effectiveness**
  - Large number of test cases selected at random does not guarantee all (or even most) of the errors will be uncovered

Choice of Test Cases (Suite)

- Example - code to find greater of two integers

  If (x>y) max = x;

  else max = x;

  (code has a simple programming error)

Choice of Test Cases (Suite)

If (x>y) max = x;

else max = x;

- Consider test suite,

  Case 1: (x=3,y=2), 3

  Case 2: (x=2,y=3), 3

- Can detect the error

Choice of Test Cases (Suite)

If (x>y) max = x;

else max = x;

- Consider a larger test suite

Case 1: (x=3,y=2), 3

Case 2: (x=4,y=3), 4

Case 3: (x=5,y=1), 5

- Can't detect the error → larger test suite not necessarily better always

Choice of Test Cases (Suite)

- Implication - test suite should be carefully designed (not decided randomly)
- Require systematic approaches

(Systematic) Code Testing

- Broadly of TWO types
  - Functional testing - test cases designed using only functional specification of software, i.e. without any knowledge of internal structure [**Black-box testing]**
  - Structural testing - test cases designed using knowledge of internal structure of software **[white-box testing]**

Black-Box Testing

Idea

- Design test cases based on input/output values ONLY [no knowledge of design or code required]
- TWO main approaches to design test cases
  - Equivalence class partitioning
  - Boundary value analysis

Equivalence Class & Partitioning

- Domain of input values partitioned into sets – each called 'equivalence class'
  - Program behavior similar for every input data belonging to an equivalence class
- Testing code with **any ONE value** of an equivalence class is as good as testing with **ALL** input values belonging to that class

Example

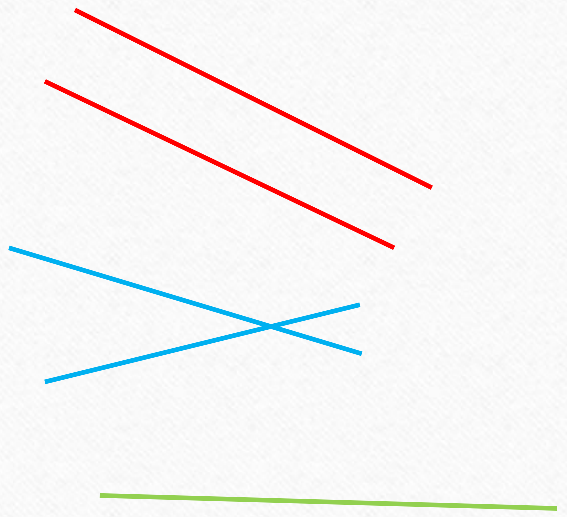
- Consider a code to compute square root of an input integer in the range [0, 5000]
- As per rule, we should define THREE classes
    - Set of negative integers
    - Set of integers in the range of [0, 5000]
    - Integers larger than 5000
- Test cases must include representative input (and corresponding output) for each of the three equivalence classes [e.g., (-5,op),(500,op),(6000,op)]

Example

- Consider another program to compute intersection point of two straight lines and displays the result
- Input - two integer pairs (m1, c1) and (m2, c2); each pair defines a straight line of the form y=mx + c
- What are the equivalence classes?

Example

- Parallel lines ($m1=m2$, $c1\neq c2$)
- Intersecting lines ($m1\neq m2$)
- Coincident lines ($m1=m2$, $c1=c2$)
- Anything else …
- Ex test suit [(2, 2) (2, 5); (5, 5) (7, 7); (10, 10) (10, 10)]



Question

- What would be the equivalence classes if, instead of lines, we now consider line segments?
  - Input of the form: (x11,y11),(x21,y21); (x12,y12),(x22,y22) [each pair indicate one end point]
  - Output – intersection point or NULL

Boundary Value Analysis

- Programming error frequently occurs at the boundaries of equivalence classes
- Due to oversight - programmers fail to notice special processing required for inputs at class boundaries
  - E.g., may improperly use < instead of <= or conversely <= for <
- Boundary value analysis - selection of test cases at the boundaries

Boundary Value Analysis

- Ex - reconsider function to compute square root of integer values in the range of 0 and 5000
- Earlier we considered test cases for 3 classes
  - Set of negative integers
  - Set of integers in the range of [0, 5000]
  - Integers larger than 5000
- Along with those, we should include {0, -1,5000,5001}

Note

- Equivalence classes are sets → should use set notations to represent
- Test cases should contain **both input and expected output**
- Remember to include test case(s) from valid class(es), invalid class(es) and boundary cases in black-box test suites

White-Box Testing

Basic Idea

- Tests internal structure of the code
- Knowledge of internal structure required
- Harder than black-box testing!

Visualizing a Program

- To understand white-box testing strategies, program (flow) visualization helps
- Can do so with Control Flow Graph (CFG)

Control Flow Graph (CFG)

- CFG graphically represents sequence of instruction execution (how the control flows through the program)

Control Flow Graph (CFG)

- How to draw CFG
  - Assign numbers (in sequence) to all statements of a program
  - Create a ‘node’ in the CFG for each numbered statement
  - Add an ‘edge’ from one node to another if execution of the statement representing first node results in transfer of control to other node

CFG (Sequence)

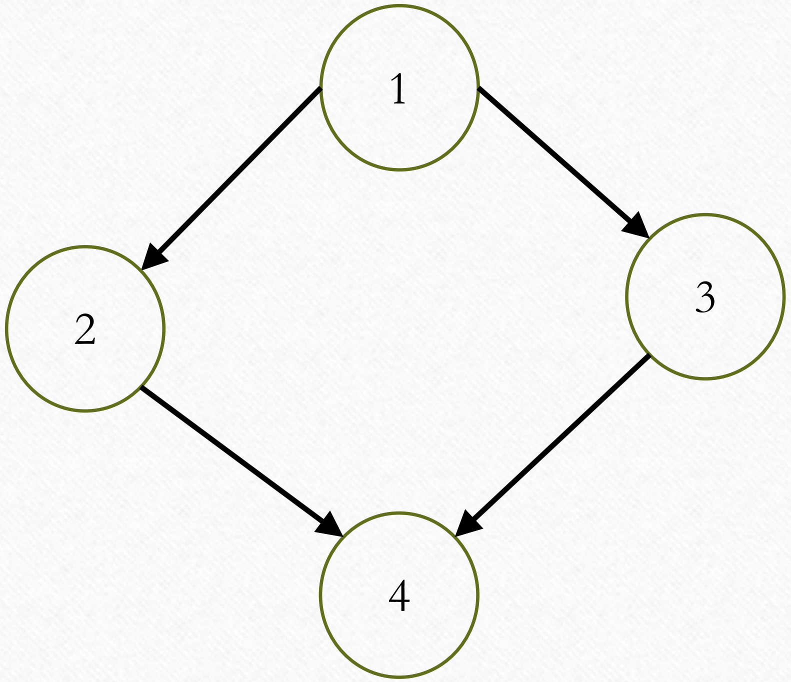
- Code segment

1. A=5;
2. B=A+10;

CFG (Selection/Condition)

- Code segment

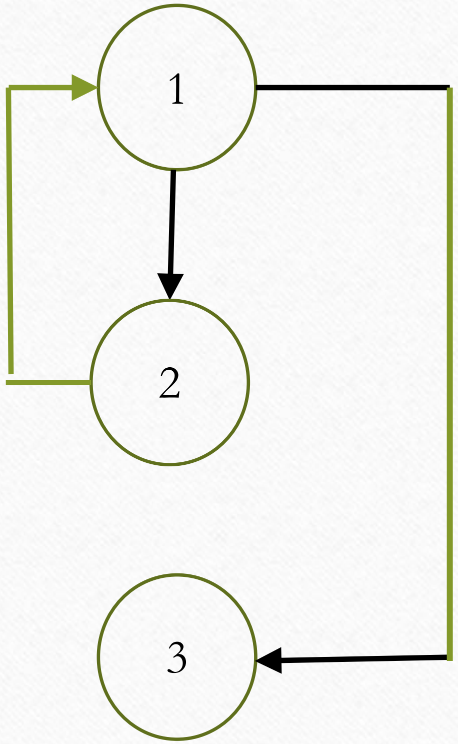
1. If (A>10)
   2. C = 1;
3. Else C = 0;
4. C=C*10;



CFG (Iteration/Loop)

- Code segment

1. While (A<B){
    2. B = B+1;}
3. C=A+B;



Test Case Design

- Core concern – code coverage
  - Test cases should ‘cover’ as much code as possible
  - Coverage – extent of code accessed during execution

Test Case Design

- Many approaches
  - Statement coverage
  - Branch coverage
  - Condition coverage
  - Path coverage
  - Control flow testing
  - Data flow testing ….

Test Case Design

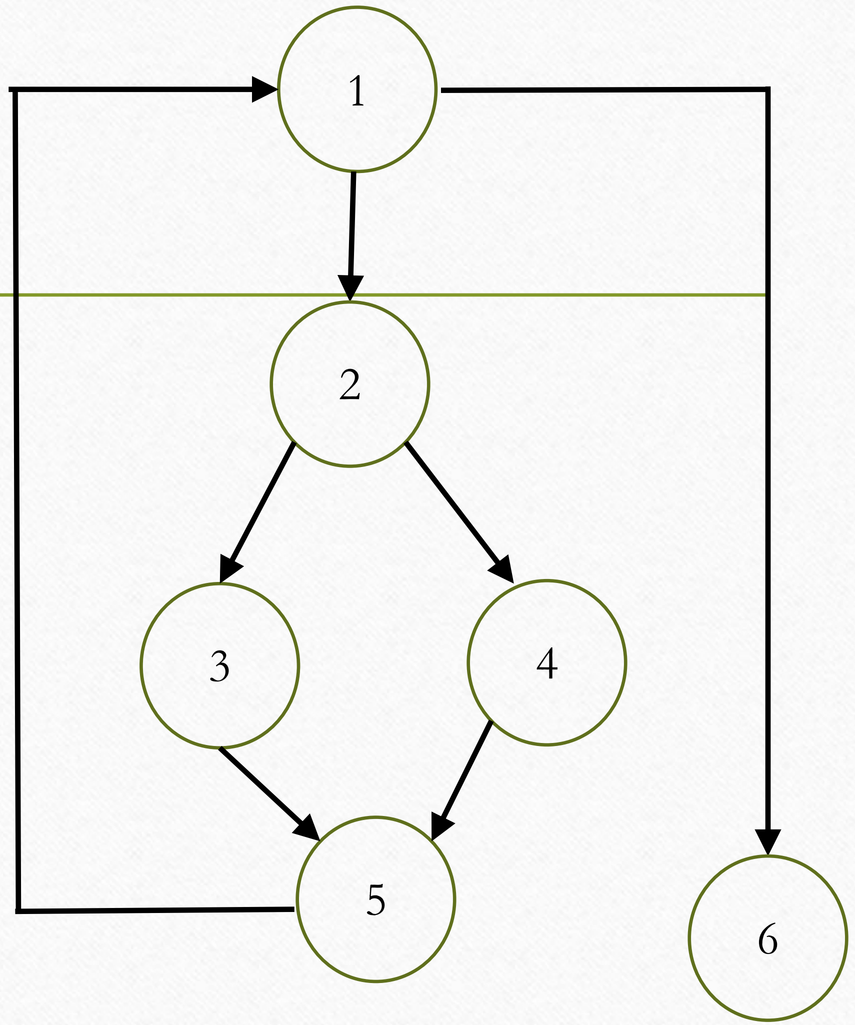
- Many approaches
  - Statement coverage
  - Branch coverage
  - Condition coverage
  - Path coverage
  - Control flow testing
  - Data flow testing ….

Test Case Design (Statement Coverage)

- Aims to design test cases such that every statement is executed at least once
  - Idea - unless a statement is executed, hard to determine existence of error in that statement
- Note - executing a statement once and observing ‘correct’ behavior for ‘some’ input does not guarantee ‘correct’ behavior for all input values

Test Case Design (Statement Coverage)

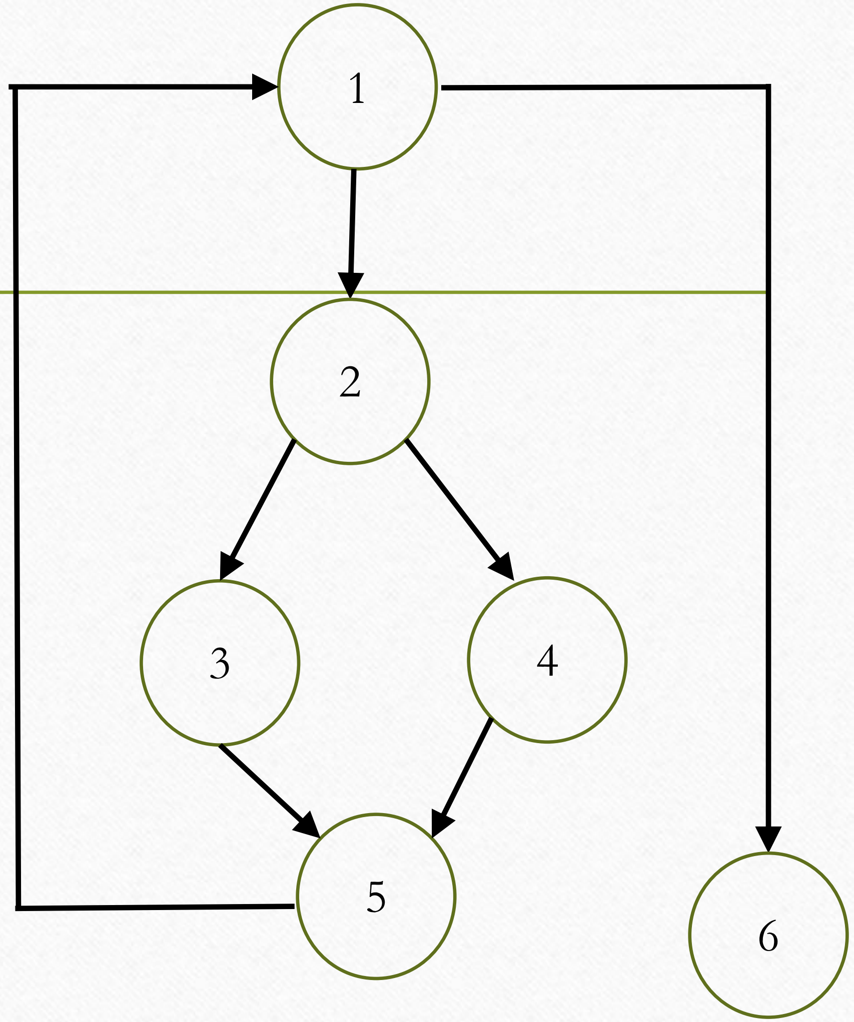
```
int doSomething (int x, int y){
1.  while (x != y){
2.      if (x > y)
3.          x = x - y;
4.      else y = y - x;
5.  }
6.  return x;
```



Test Case Design (Statement Coverage)

```
int doSomething (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;
```

- **x=3, y=3**
- **x=3, y=2**
- **x=3, y=4**



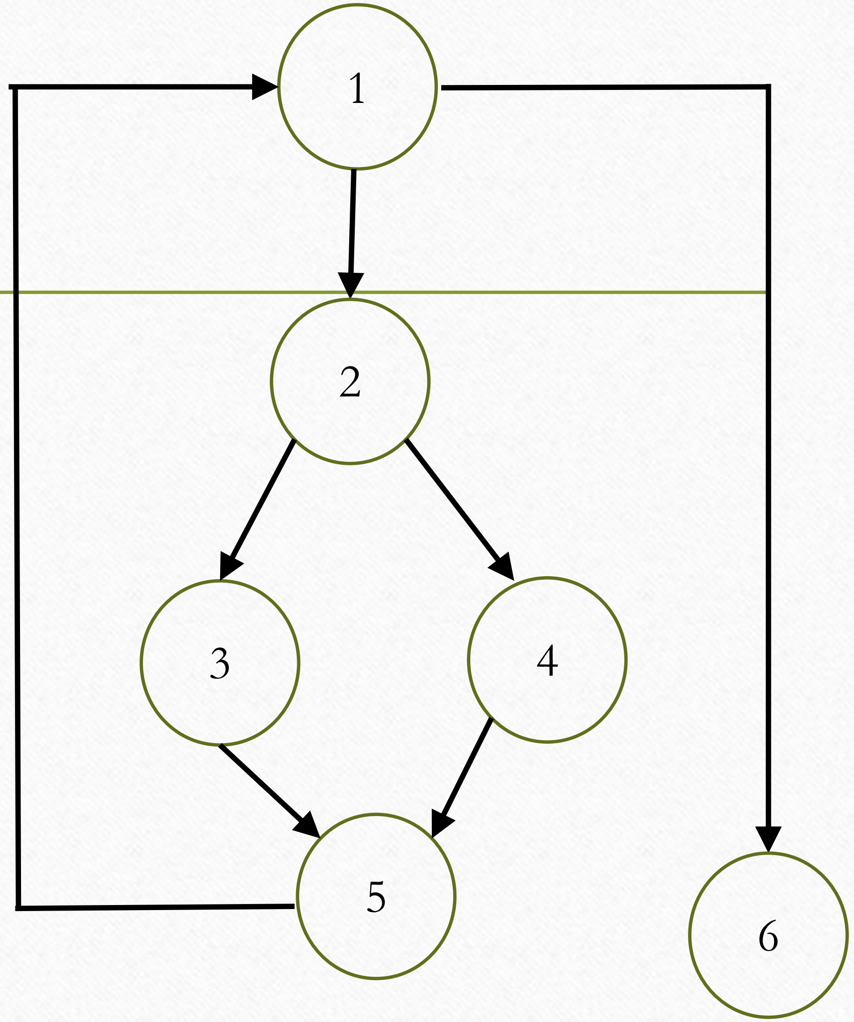
Test Case Design (Branch Coverage)

- Test cases designed to cover each branch condition (both true and false values) - also known as ‘edge testing’
- Guarantees statement coverage - stronger strategy compared to statement coverage

Test Case Design (Branch Coverage)

```
int doSomething (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;
```

- **x=3, y=3**
- **x=3, y=2**
- **x=3, y=4**



Test Case Design (Condition Coverage)

- Test cases designed to make each component of a composite conditional expression assume both true and false values
  - E.g., ((c1 AND c2) OR c3) – test cases should make c1, c2 and c3 each assume true and false
  - n components → 2^n test cases for each composite condition

Test Case Design (Condition Coverage)

- Branch testing – simplest condition coverage strategy (true/false values considered for whole condition rather than individual components)
- Guarantees branch and statement coverage - stronger strategy compared to both (may be impractical if conditions are complex)

Test Case Design (Path Coverage)

- Test cases should ensure all **linearly independent paths** in the code executed at least once
- Path - a node and edge sequence from starting node to a terminal node of CFG of a program (note – CFG can have more than one terminal node)

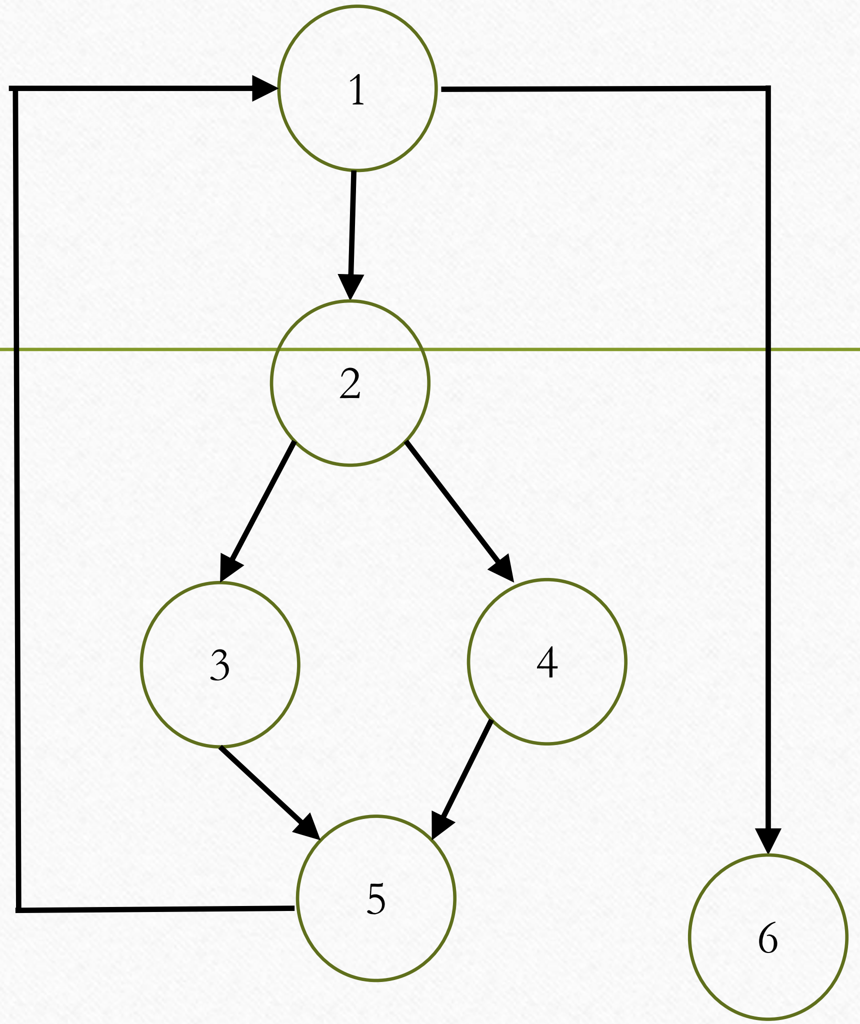
Test Case Design (Path Coverage)

- Linearly independent path - any path with at least one new edge/node not included in any other linearly independent paths of the CFG
  - Sub path of another path not linearly independent

Example

int doSomething (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;



- Identify the paths?
  - 1→2→3→5→1→6
  - 1→2→4→5→1→6
  - 1→6

Test Case Design (Path Coverage)

- **Cyclomatic complexity** – a measure of upper bound of number of linearly independent paths for a CFG
  - One way to compute: E-N+2 (E = no of edges, N = no of nodes)
  - Another way: D+1 (D = no of decision statements)
- In previous example: E=7, N=6, D=2, complexity = 3

Integration & System Testing

Integration & Testing

- System consists of subsystems (modules and units)
- Testing whole system at a time difficult
- Alternative approaches required

Integration & Testing

- TWO broad approaches
  - Bottom-up testing
  - Top-down testing

Bottom-up Testing

- Each subsystem tested separately and then full system tested
- A subsystem may consist of many modules communicate through well-defined interfaces
  - Primary purpose is to test the interfaces
  - Both control and data interfaces are tested
  - Test cases should exercise all interfaces in all possible manners

Top-down Testing

- Testing starts with the main routine
  - After top-level ‘skeleton’ tested, the immediate subroutines of the ‘skeleton’ are combined with it and tested

Stubs & Drivers

- Such approaches may (likely) require
  - ‘Stubs’ - simulate effect of lower-level routines called by the routines under test (in top-down approach)
  - ‘Driver’ routines – used during bottom-up testing to ‘simulate’ behavior of upper level modules that are not yet integrated

System Testing

- THREE main stages
  - **Alpha testing -** carried out by test team within organization
  - **Beta testing -** performed by a select group of 'friendly customers' (may be specially 'recruited')
  - **Acceptance testing -** performed by customer

System Testing

- What is tested?
  - Functionality
  - Performance

System Testing

- Functionality test
  - Test software functionality w.r.t SRS document

System Testing

- Performance test – tests non-functional requirements
- Some important tests
  - Stress testing - evaluates system performance under abnormal/illegal input conditions (in short time periods)
  - Volume testing – tests system performance for large input
  - Configuration testing – done to analyze system behavior in various hardware and software configurations specified in the requirements

System Testing

- Performance test – tests non-functional requirements
- Some important tests
  - Compatibility testing – checks if the system interfaces properly with other systems
  - Regression testing – tests backward compatibility of software with older platforms/systems
  - Recovery testing - tests system response to faults such as loss of power, devices, services, data, and so on

System Testing

- Performance test – tests non-functional requirements
- Some important tests
  - Documentation testing – tests various manuals and documents created
  - Usability testing – empirical testing (more on it later)

Reference

- Rajib Mall – Fundamentals of S/W Engineering
- Roger Pressman –S/W Engineering: A Practitioner's Approach