Module 3: Software Testing and Quality (8 hrs) – Comprehension level

Introduction to Software Testing: verification and validation, Test Strategies for conventional Software, Validation Testing, White box Testing: Basis path testing, Black box Testing. Software Quality Assurance : Elements of software quality assurance, SQA Tasks, Goals and Metrics, Software configuration management : SCM process.

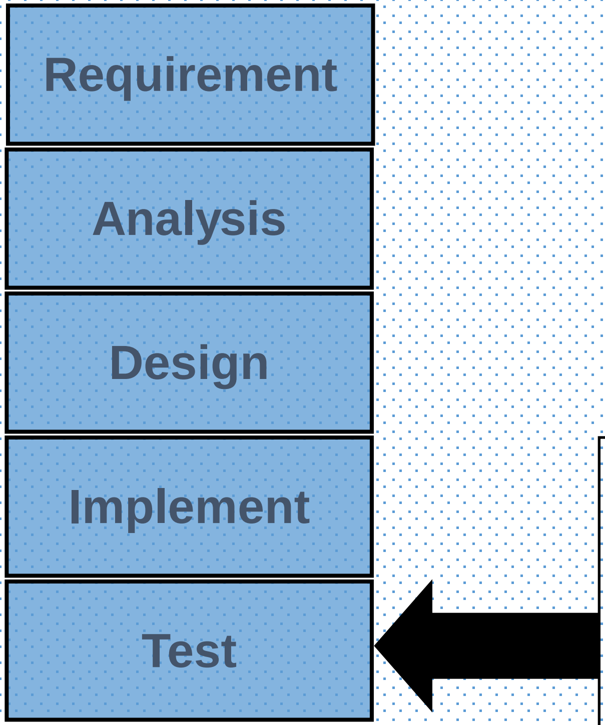
Software Testing Strategies

Contents

Software Testing

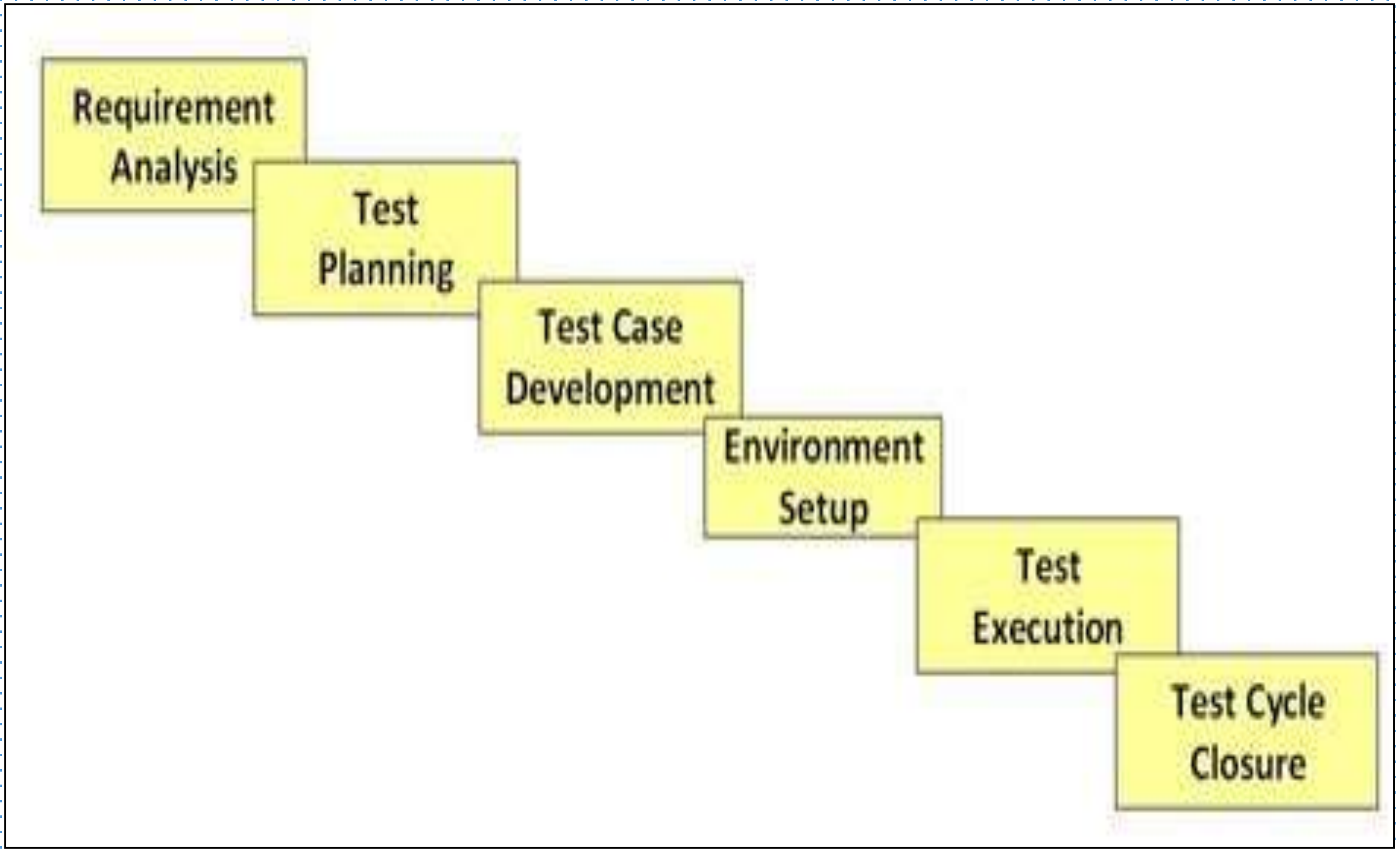
Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

Where are We?

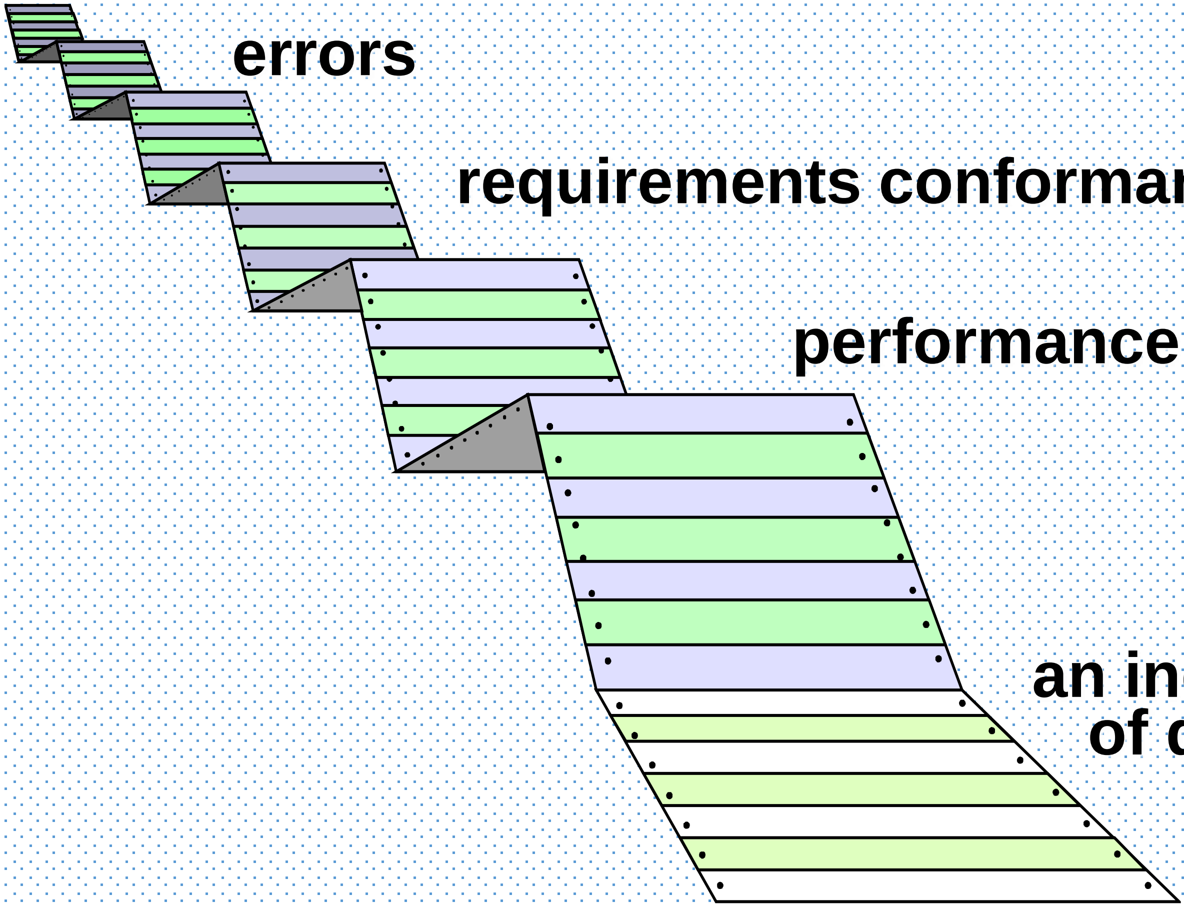


- Evaluating the System

Software Testing Methodology



What Testing Shows



errors

requirements conformance

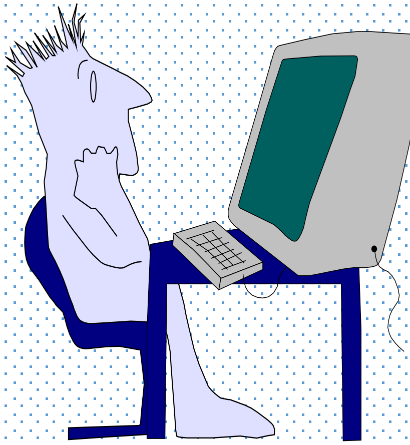
performance

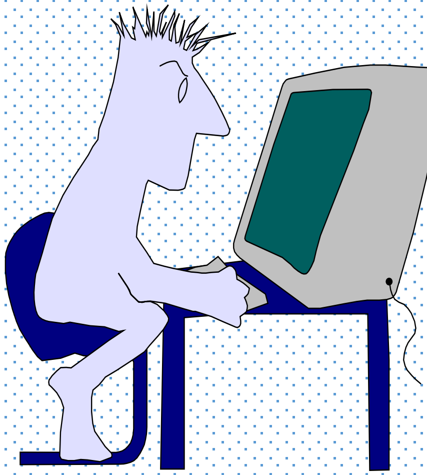
an indication of quality

Verification and Validation

- *Verification* refers to the set of tasks that ensure that software correctly implements a specific function.
- *Validation* refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.
- Boehm states this another way:
  - *Verification:* "Are we building the product right?"
  - *Validation:* "Are we building the right product?"

Who Tests the Software





Developer

Understands the system

but, will test "gently"

and, is driven by "delivery"

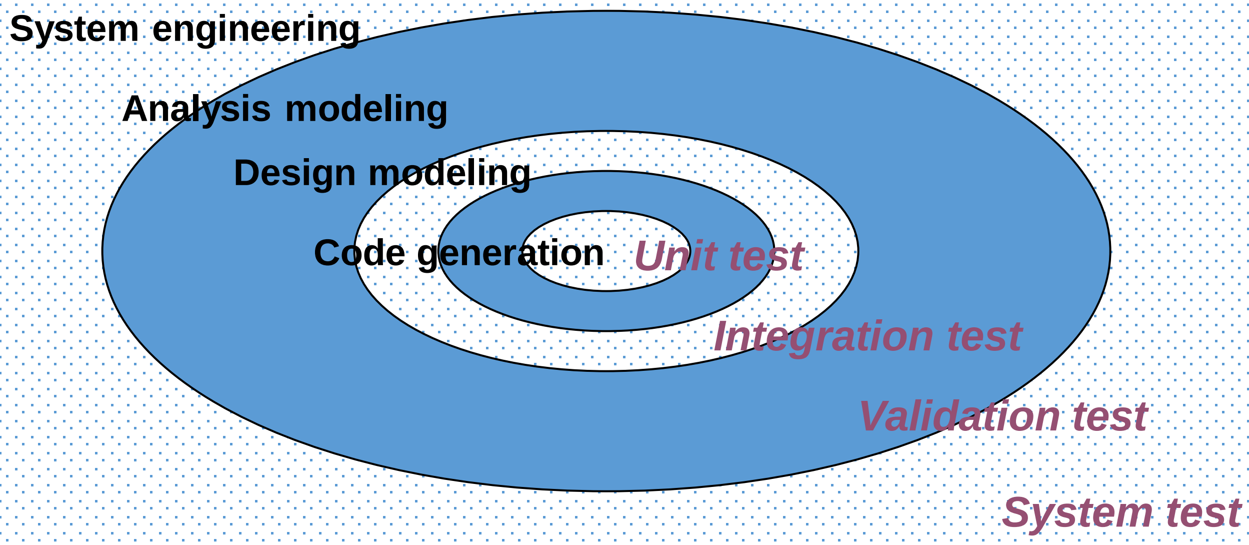
Independent Tester

Must learn about the system,

but, will attempt to break it

and, is driven by quality

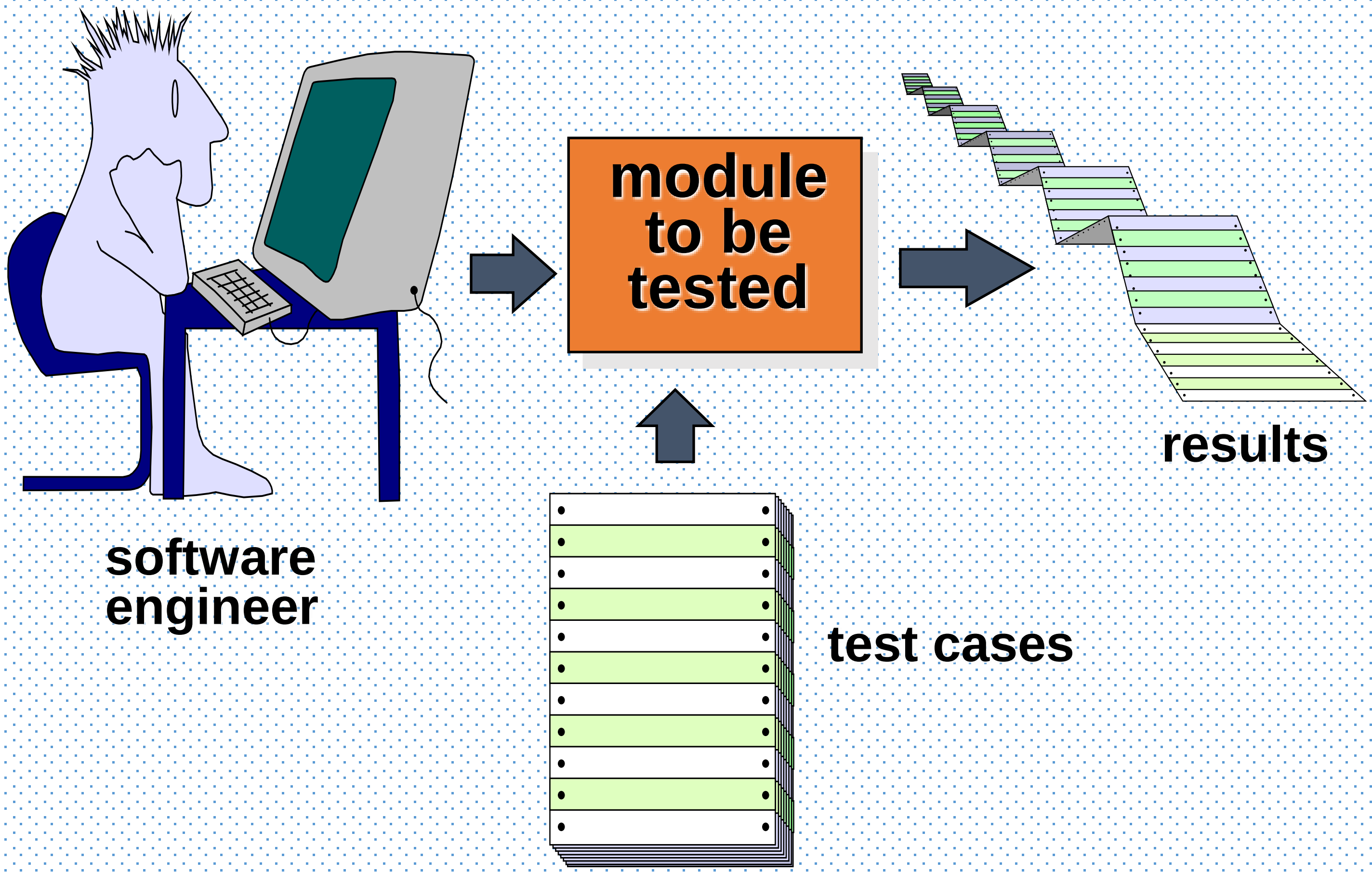
Testing Strategy



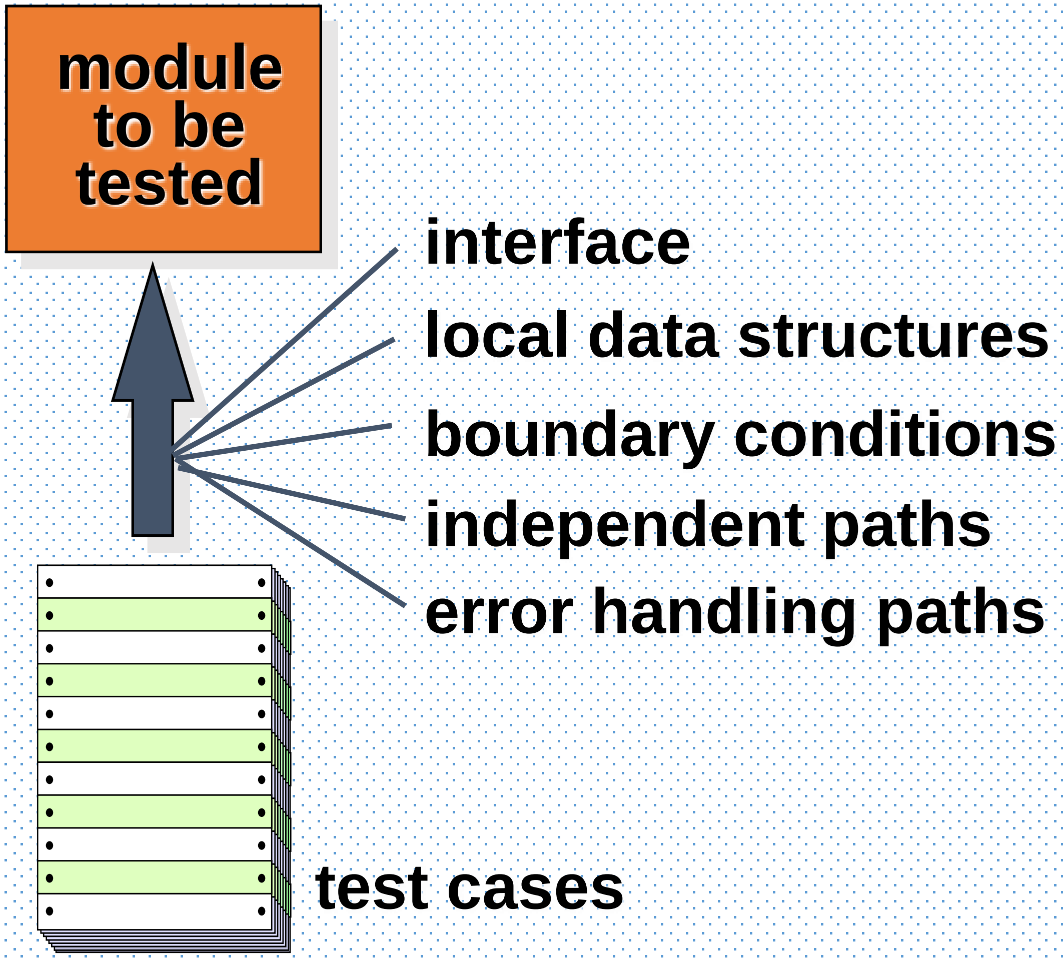
Testing Strategies

- **Unit testing** - testing performed on each module or block of code during development. Unit Testing is normally done by the **programmer** who writes the code.
- **Integration testing** - testing done before, during and after integration of a new module into the main software package. This involves testing of each individual code module. One piece of software can contain several modules which are often created by several different programmers. It is crucial to test each module's effect on the entire program model. Performed by **professional testing team**.
- **System testing** - testing done by a **professional testing team** on the completed software product before it is introduced to the market.
- **Acceptance testing** - beta testing of the product done by the **end-users** of the software.

1. Unit Testing



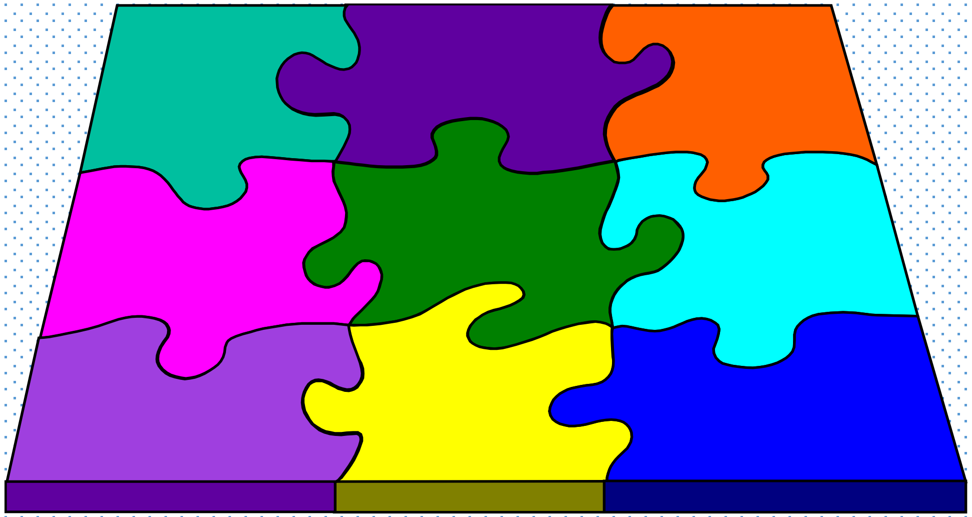
1. Unit Testing



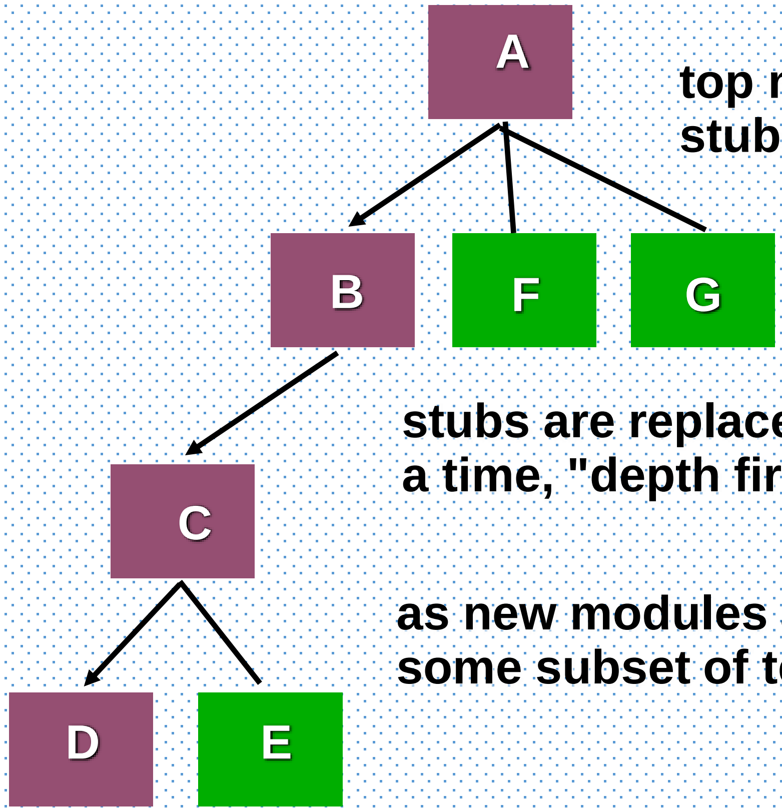
2. Integration Testing Strategies

Options:

- **the "big bang" approach**
- **an incremental construction strategy**



Top Down Integration

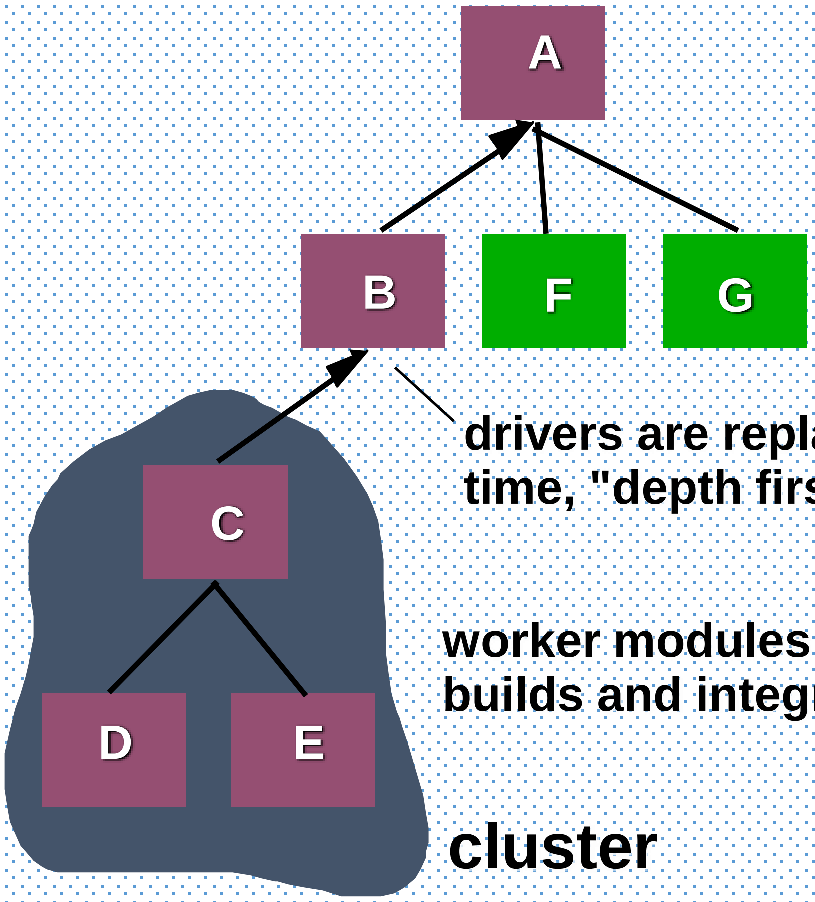


top module is tested with stubs

stubs are replaced one at a time, "depth first"

as new modules are integrated, some subset of tests is re-run

Bottom-Up Integration

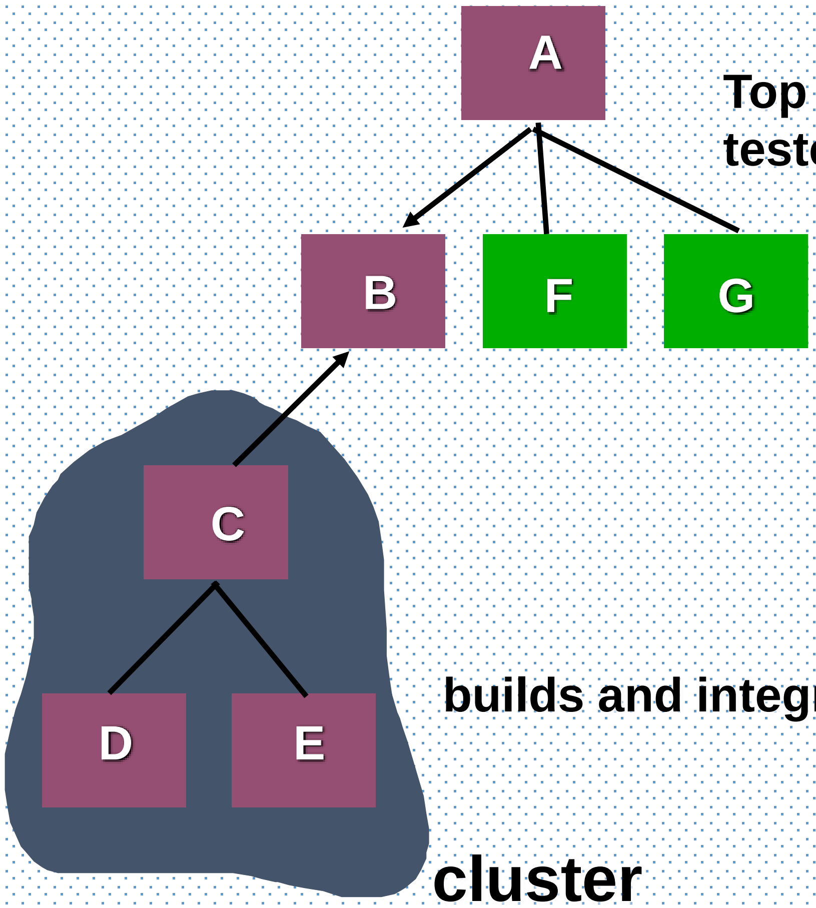


drivers are replaced one at a time, "depth first"

worker modules are grouped into builds and integrated

cluster

Sandwich Testing



Top modules are tested with stubs

builds and integrated

cluster

3. System Testing

System Testing is a level of software testing where a complete and integrated software is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements.

4. Acceptance Testing

Acceptance Testing is a level of software testing where a system is tested for acceptability. The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.

Manual and Automated Testing

- Unit testing, Integration testing, System testing and Acceptance testing can be performed either manually or in automated ways

Manual Testing	Automated Testing
Manual testing requires human intervention for test execution.	Automation Testing is use of tools to execute test cases
Manual testing will require skilled labour, long time & will imply high costs.	Automation Testing saves time, cost and manpower. Once recorded, it's easier to run an automated test suite
Any type of application can be tested manually, certain testing types like ad-hoc and monkey testing are more suited for manual execution.	Automated testing is recommended only for stable systems and is mostly used for System Testing
Manual testing can be become repetitive and boring.	The boring part of executing same test cases time and again, is handled by automation software in Automation Testing.

Regression Testing

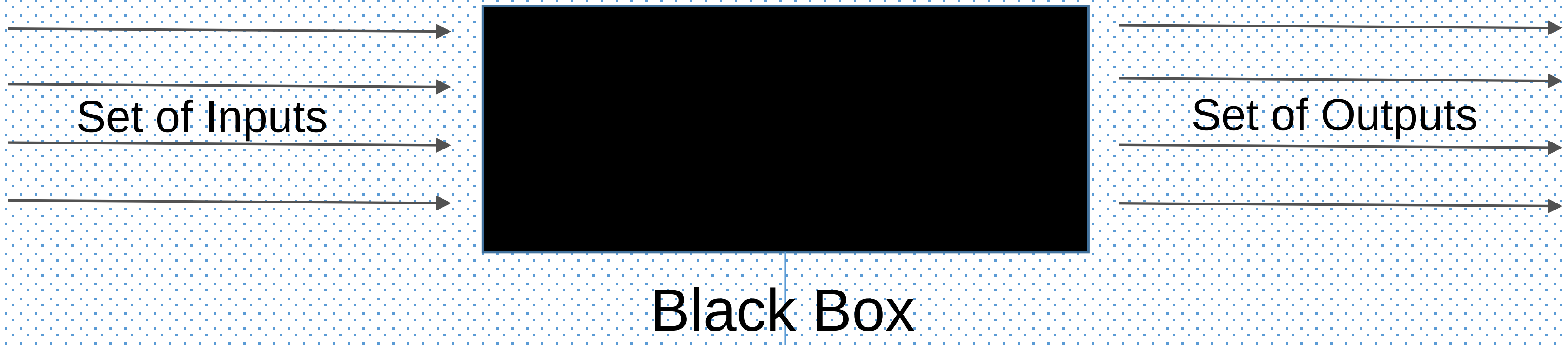
- *Regression testing* is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects
- Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.
- Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.
- Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools.

Smoke Testing

- A common approach for creating "daily builds" for product software
- Smoke testing steps:
  - Software components that have been translated into code are integrated into a "build."
    - A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
  - A series of tests is designed to expose errors that will keep the build from properly performing its function.
    - The intent should be to uncover "show stopper" errors that have the highest likelihood of throwing the software project behind schedule.
  - The build is integrated with other builds and the entire product (in its current form) is smoke tested daily.
    - The integration approach may be top down or bottom up.

Black Box Testing (Functional Testing)

- Uses only the Specification document to identify test cases.
- Knowing the specified function a component has been designed for.
- Tests conducted at the interface of the component.



Black Box Testing

Advantages

- Independent of how the software is implemented.
- If implementation changes, test cases are still useful.
- Test case development can occur in parallel with the implementation.

Disadvantages:

- Redundancies may exist among test cases.

White Box Testing (Structural Testing)

- Structural testing uses the **programs source code (implementation)** as the basis of test case identification.
- Requires knowledge of internal workings of a component.
- Test cases exercise specific sets of condition, loops, etc.

Advantages:

- Provides more coverage in testing conditions and decisions.
- Reveals hidden errors during coding.

Disadvantages:

- Missed cases in the code are obviously missed during testing too.
- In-depth knowledge of the programming language is needed.

Test Case Design

- **White Box**
  - Control Flow Graph
  - Cyclomatic Complexity
  - Basis Path Testing
- **Black Box**
  - Equivalence Classes
  - Boundary Value Analysis

Introduction to White Box Testing

- Test Engineers have access to the source code.
- Typical at the Unit Test level as the programmers have knowledge of the internal logic of code.
- Tests are based on coverage of:
  - Code statements;
  - Branches;
  - Paths;
  - Conditions.
- Most of the testing techniques are based on *Control Flow Graph* (denoted as CFG) of a code fragment.

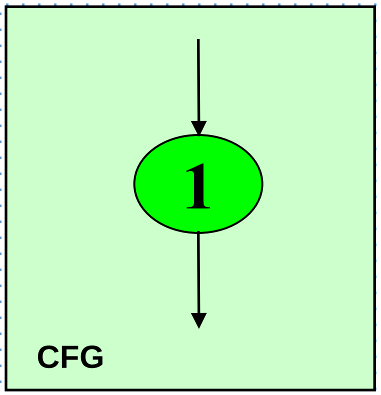
Control Flow Graph

- An abstract representation of a structured program/function/method.
- Consists of two major components:
  - ***Node***:
    - Represents a stretch of sequential code statements with no branches.
  - ***Directed Edge* (also called *arc*):**
    - Represents a branch, alternative path in execution.
- **Path:**
  - A collection of *Nodes* linked with *Directed Edges*.

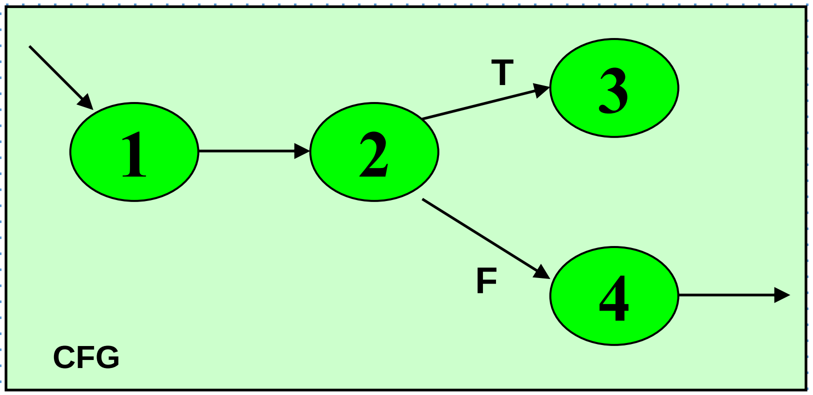
Simple Examples of CFG

```
   Statement1;
   Statement2;
   Statement3;
   Statement4;
```

Can be represented as **one** node as there is no branch.

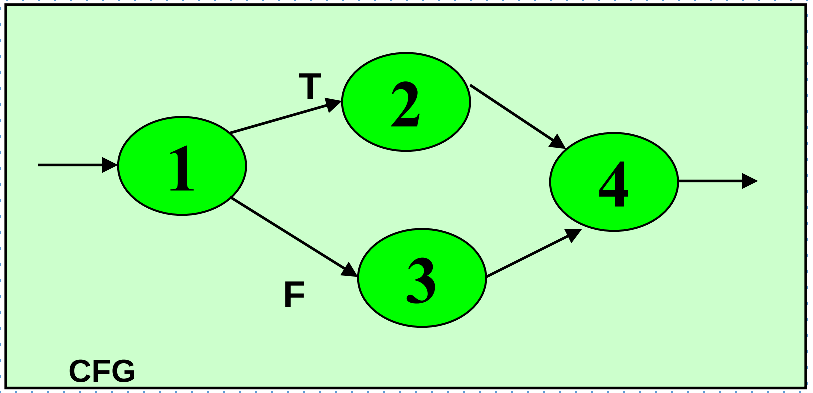


```
   Statement1;       1
   Statement2;

   if X < 10 then    2
       Statement3;   3
   else
   Statement4;       4
```

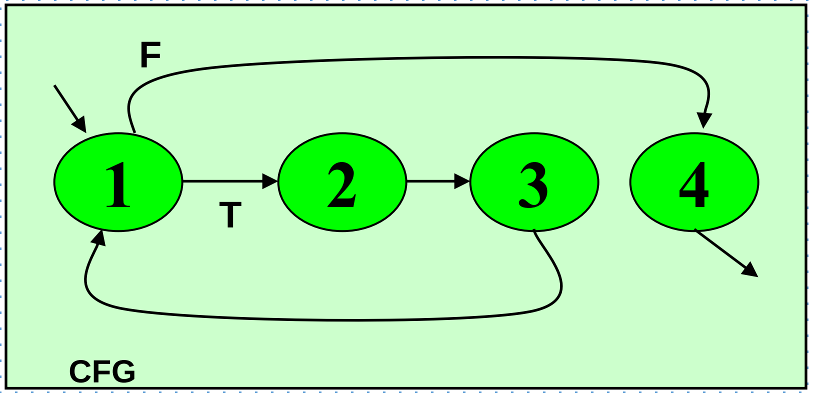


More Examples of CFG

```
if X > 0 then          1
    Statement1;        2
else
    Statement2;        3
```



```
while X < 10 {         1
    Statement1;        2
    X++; }             3
```

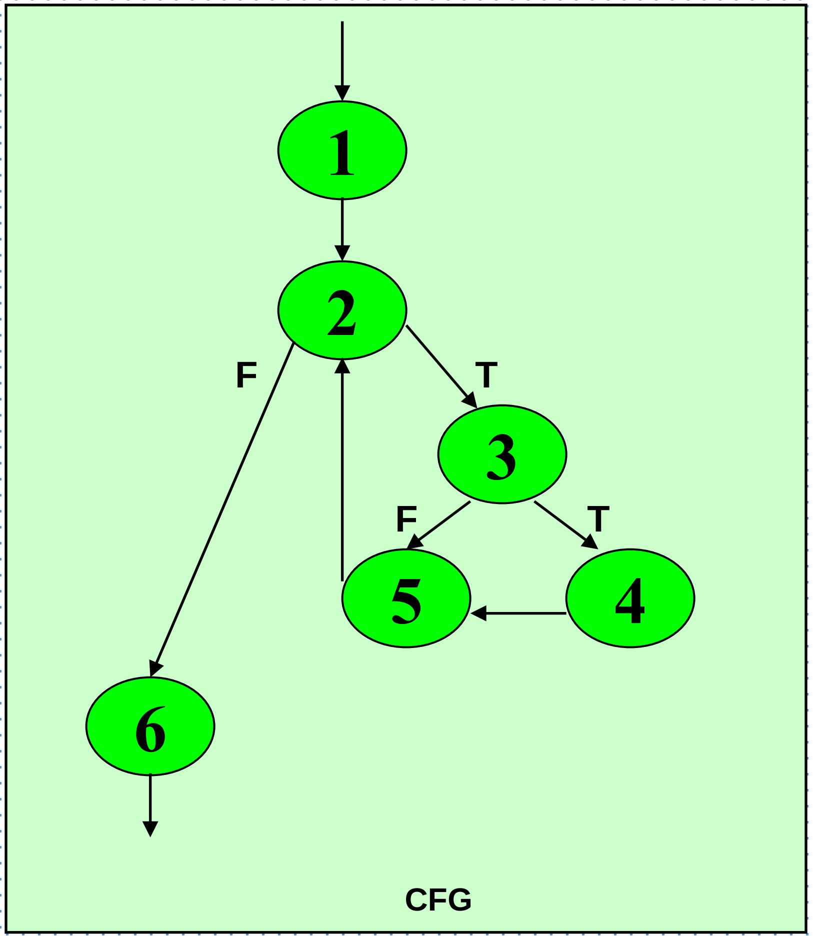


Notation Guide for CFG

- A CFG should have:
  - 1 entry arc (known as a directed edge, too).
  - 1 exit arc.
- All nodes should have:
  - At least 1 entry arc.
  - At least 1 exit arc.
- **A Logical Node** that does not represent any actual statements, can be added as a joining point for several incoming edges.
  - Represents a logical closure.
  - Example:
    - Node 4 in the `if-then-else` example from previous slide.

Example: Minimum Element

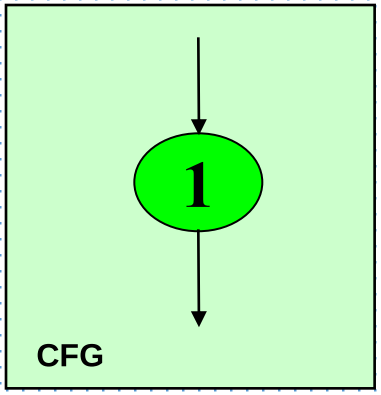
```
min = A[0];              ⎫
I = 1;                   ⎭ 1

while (I < N) {          } 2
      if (A[I] < min)    } 3
         min = A[I];     } 4
      I = I + 1;         } 5
}
print min                } 6
```



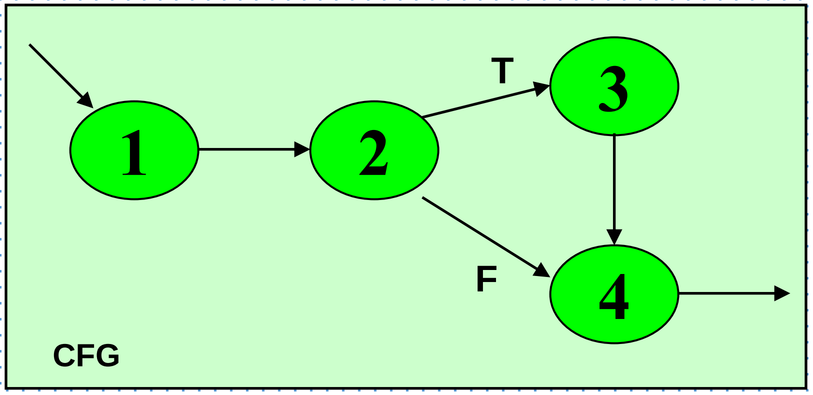
Number of Paths through CFG

- Given a program, how do we exercise all statements and branches at least once?
- Translating the program into a CFG, an equivalent question is:
  - Given a CFG, how do we cover all arcs and nodes at least once?
- Since a path is a trail of nodes linked by arcs, this is similar to ask:
  - Given a CFG, what is the set of paths that can cover all arcs and nodes?

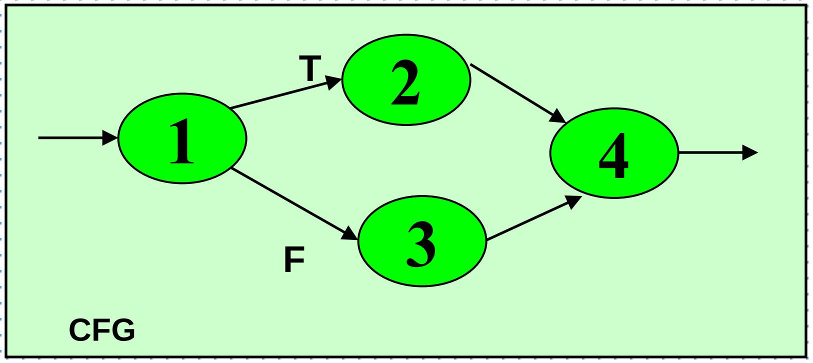
Number of Paths through CFG – Example



- Only **one** path is needed:
  [1]



- **Two** paths are needed:
  [1 – 2 – 4]
  [1 – 2 – 3 – 4]



- **Two** paths are needed:
  [1 – 2 – 4]
  [1 – 3 – 4]

Basis Path Testing

Path Testing: Testing designed to execute all or selected paths through a computer program.

Branch Testing: Testing designed to execute each outcome of each decision point in a computer program.

Basis Path Testing: is a hybrid between path testing and branch testing. It fulfills the requirements of branch testing & also tests all of the independent paths that could be used to construct any arbitrary path through the computer program

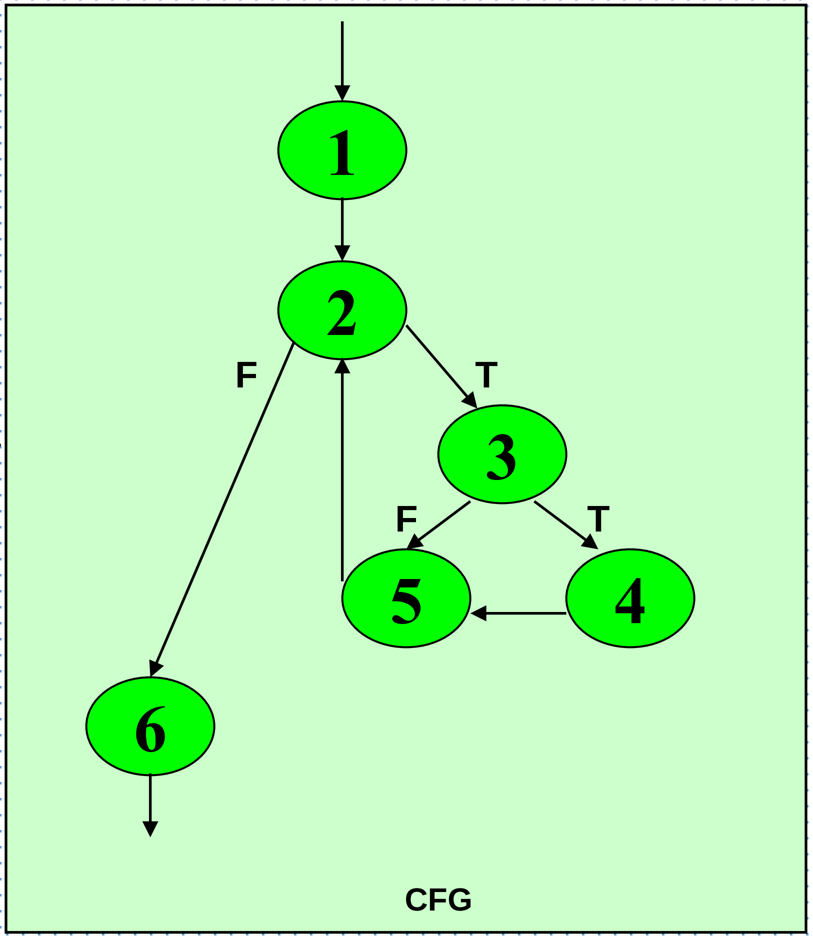
Basis Path Testing

- A generalized technique to find out the number of paths needed (known as "*cyclomatic complexity*") to cover all arcs and nodes in CFG.
- Steps:

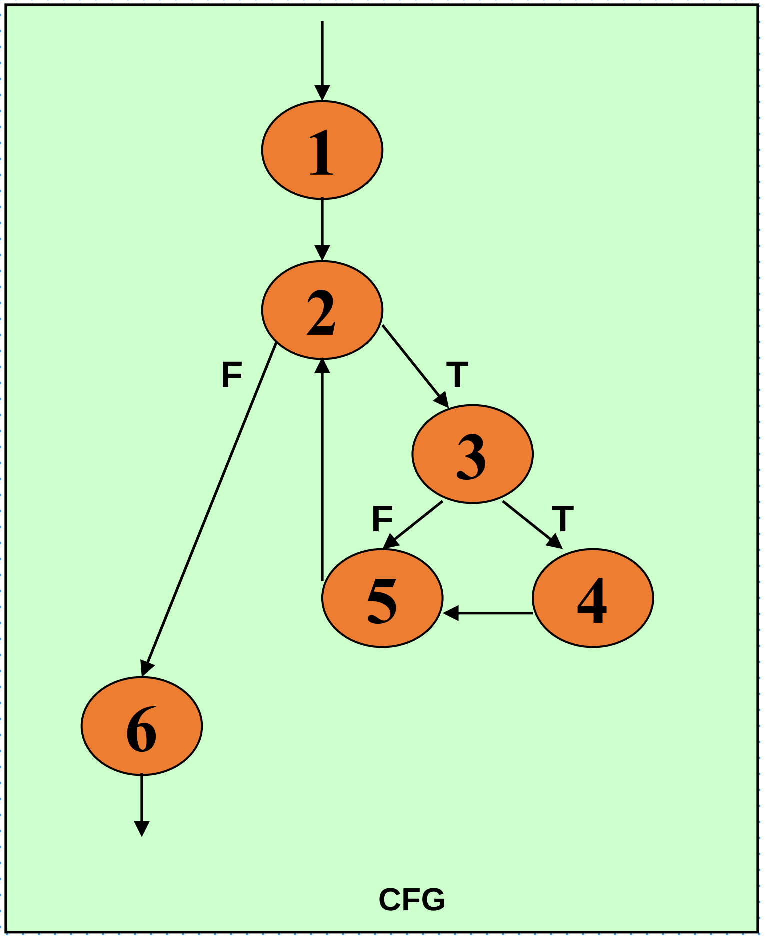
1. Draw the CFG for the code fragment.
2. Compute the *cyclomatic complexity number* ***C***, for the CFG.
3. Find at most ***C*** paths that cover the nodes and arcs in a CFG, also known as **Basis Paths Set.**
4. Design test cases to force execution along paths in the **Basis Paths Set.**

Step 1 – Draw CFG

```
min = A[0];
I = 1;

while (I < N) {
      if (A[I] < min)
         min = A[I];
      I = I + 1;
}
print min
```



Step 2 – Compute Cyclomatic Complexity

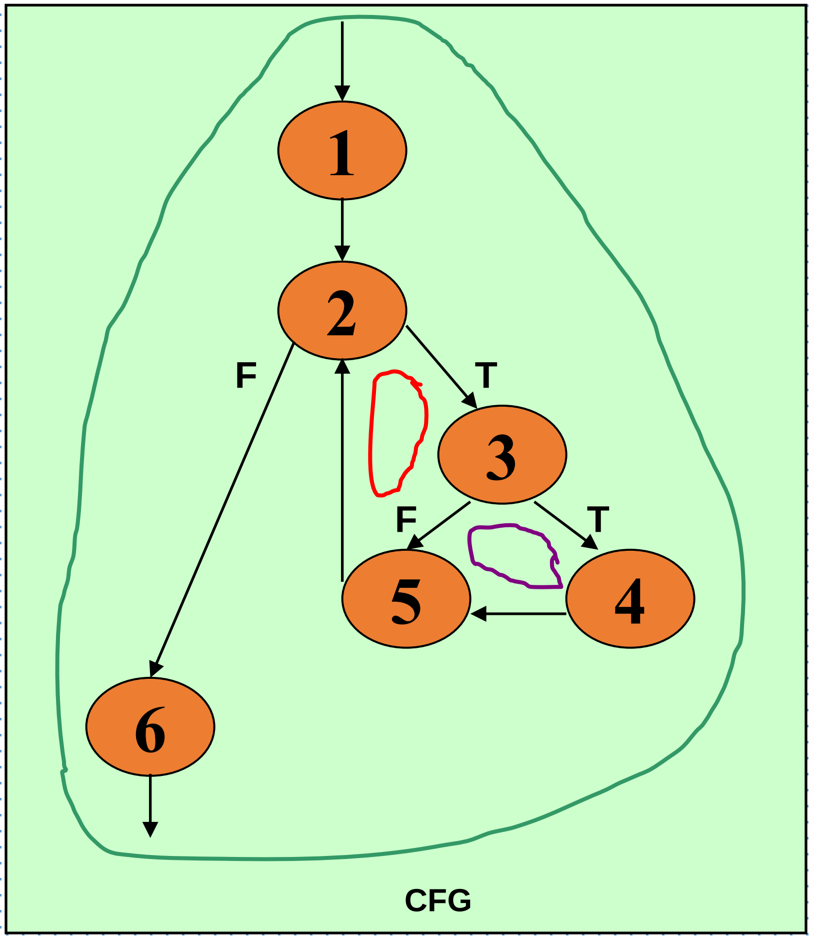


- Cyclomatic complexity =
  - The number of ‘regions’ in the graph;

  OR

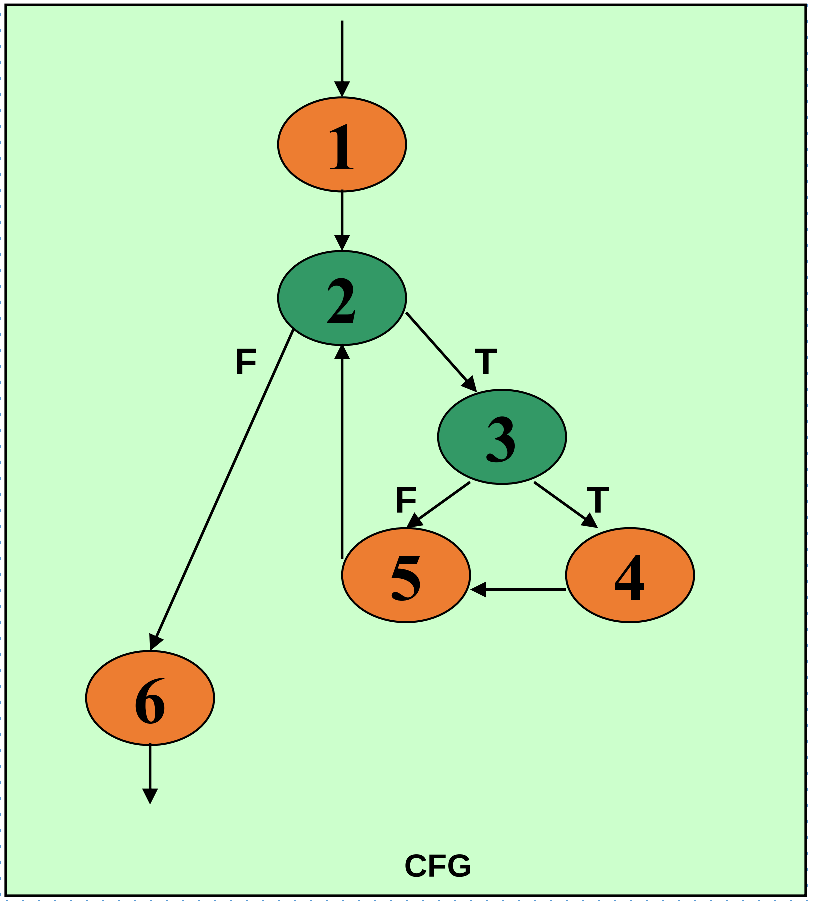
  - The number of predicates + 1.

Step 2 – Compute Cyclomatic Complexity



- **Region**: Enclosed area in the CFG.
  - Do not forget the outermost region.
- In this example:
  - 3 Regions (see the circles with different colors).
  - Cyclomatic Complexity = 3
- Alternative way in next slide.

Step 2 – Compute Cyclomatic Complexity



- **Predicates**:
  - Nodes with multiple exit arcs.
  - Corresponds to branch/conditional statement in program.
- In this example:
  - Predicates = 2
    - (Node 2 and 3)
  - Cyclomatic Complexity

    = 2 + 1

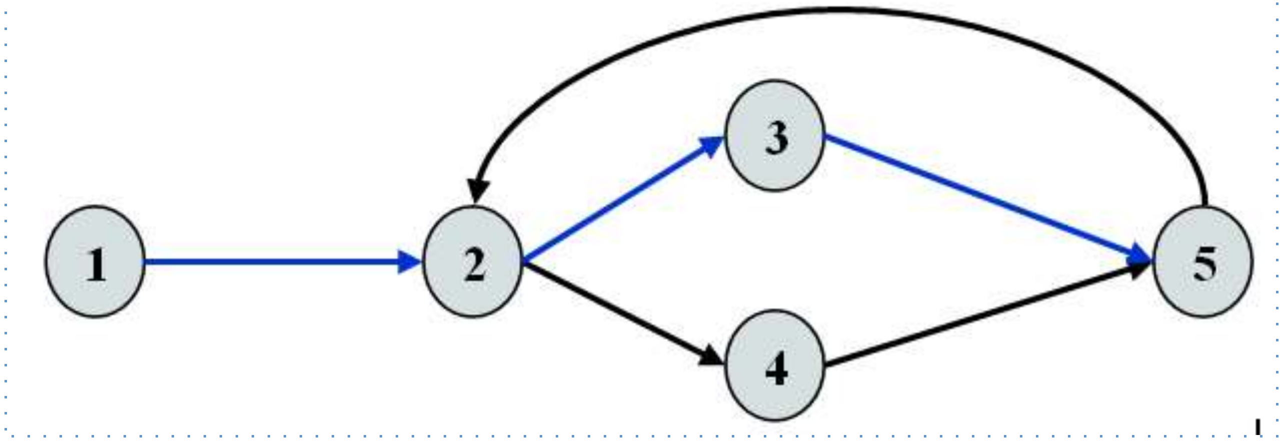
    = 3

Step 3 – Find Basis Path Set

- Independent path:
  - An **executable** or **realizable path** through the graph from the start node to the end node that has not been traversed before.
  - **Must** move along **at least one arc** that has not been yet traversed (an unvisited arc).
  - The objective is to cover all statements in a program by independent paths.
- The number of independent paths to discover

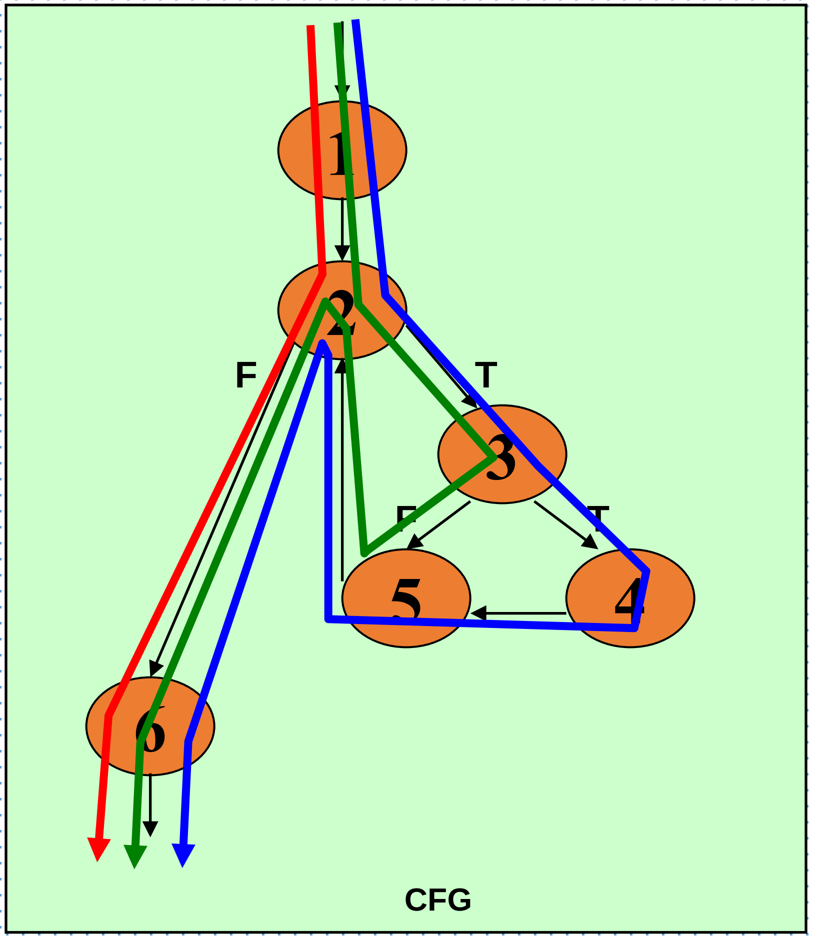
  <= cyclomatic complexity number.
- Decide the Basis Path Set:
  - It is the maximal set of *independent paths* in the flow graph.
  - **NOT** a unique set.

Step 3 – Example



- 1-2-3-5 can be the first independent path; 1-2-4-5 is another; 1-2-3-5-2-4-5 is one more.
- There are only these 3 independent paths. The basis path set is then having 3 paths.
- Alternatively, if we had identified 1-2-3-5-2-4-5 as the first independent path, there would be no more independent paths.
- The number of independent paths therefore can vary according to the order we identify them.

Step 3



- Cyclomatic complexity = 3.
- Need at most **3** independent paths to cover the CFG.
- In this example:
  - [1 – 2 – 6]
  - [1 – 2 – 3 – 5 – 2 – 6]
  - [1 – 2 – 3 – 4 – 5 – 2 – 6]

Step 4 – Design Test Cases

- Prepare a test case for each independent path.
- In this example:
  - Path: [1 – 2 – 6]
    - Test Case: A = { 5, …}, N = 1
    - Expected Output: 5
  - Path: [1 – 2 – 3 – 5 – 2 – 6]
    - Test Case: A = { 5, 9, … }, N = 2
    - Expected Output: 5
  - Path: [1 – 2 – 3 – 4 – 5 – 2 – 6]
    - Test Case: A = { 8, 6, … }, N = 2
    - Expected Output: 6
- These tests will result a complete decision and statement coverage of the code.

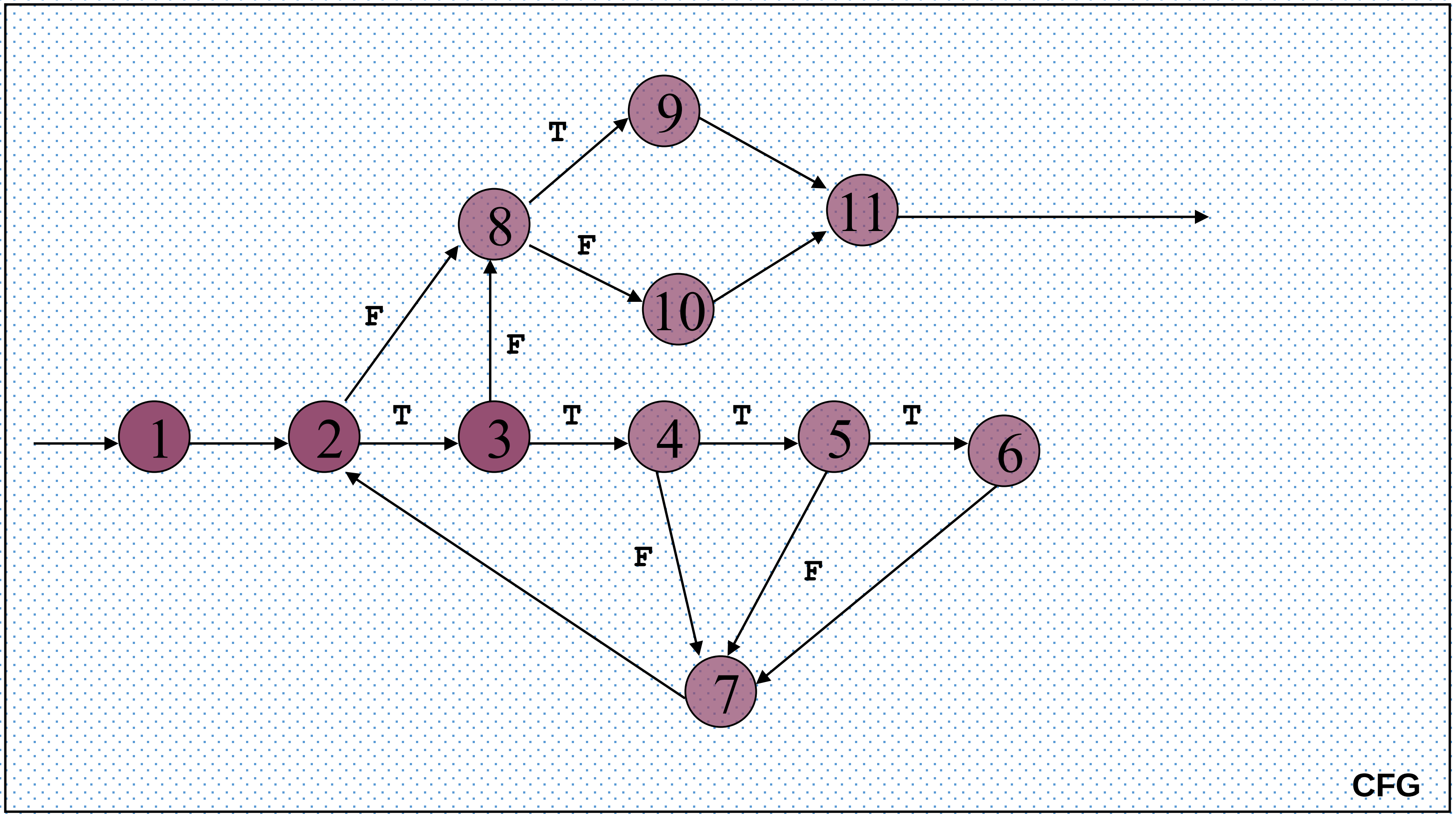
Another Example

```
int average (int[ ] value, int min, int max, int N) {
   int i, totalValid, sum, mean;
   i = totalValid = sum = 0;
   while ( i < N && value[i] != -999 ) {
       if (value[i] >= min && value[i] <= max){
              totalValid += 1; sum += value[i];
       }
       i += 1;
   }
   if (totalValid > 0)
       mean = sum / totalValid;
   else
       mean = -999;
   return mean;
}
```

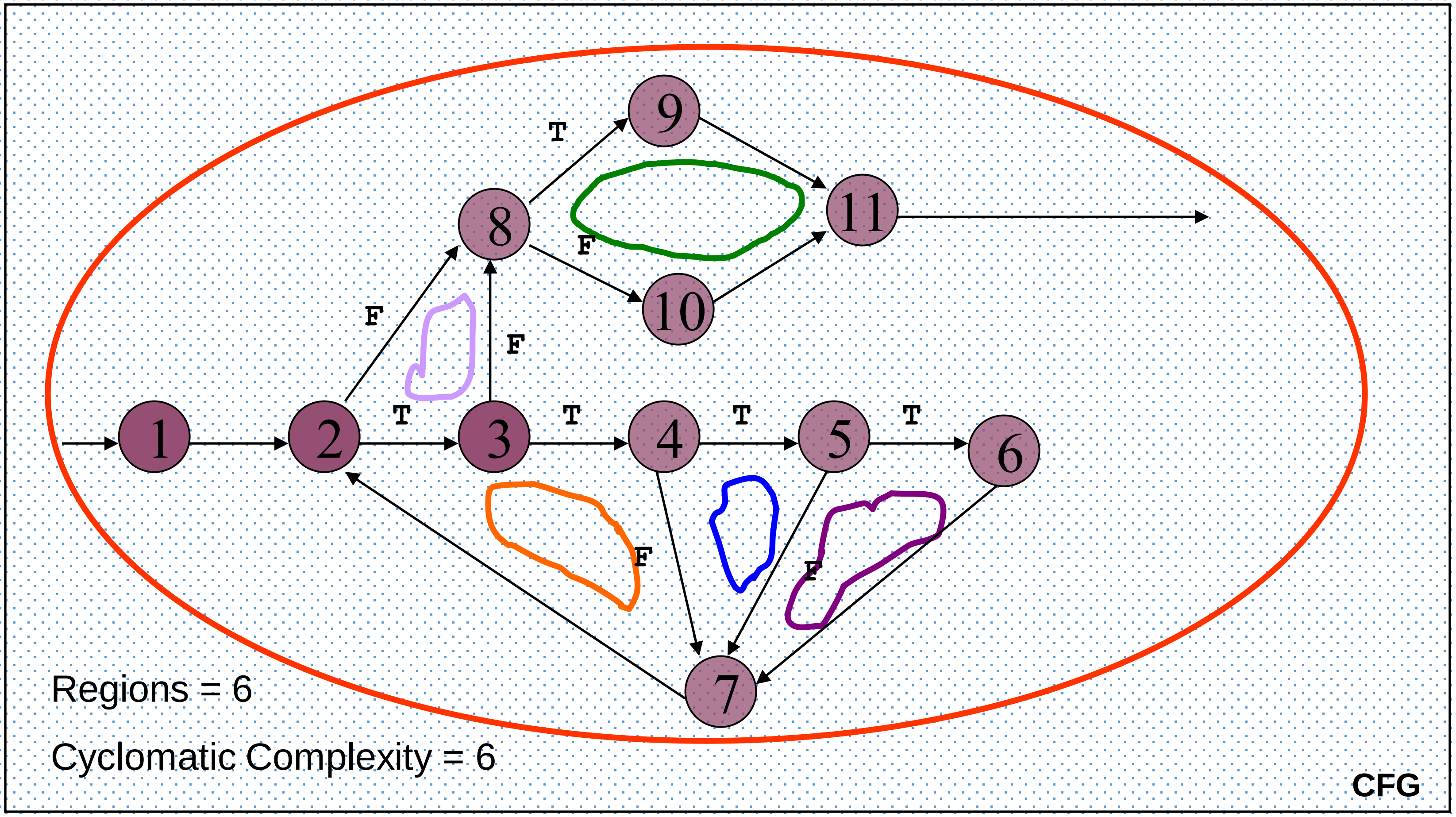
Step 1: Draw CFG

```
int average (int[ ] value, int min, int max, int N) {
   int i, totalValid, sum, mean;
   i = totalValid = sum = 0;                          } 1
   while ( i < N && value[i] != -999 ) {
             2        4        3          5
        if (value[i] >= min && value[i] <= max){
                totalValid += 1; sum += value[i];     6
        }
        i += 1;    7
   }
   if (totalValid > 0)   8
        mean = sum / totalValid;   9
   else
        mean = -999;   10
   return mean;  11
}
```

Step 1: Draw CFG



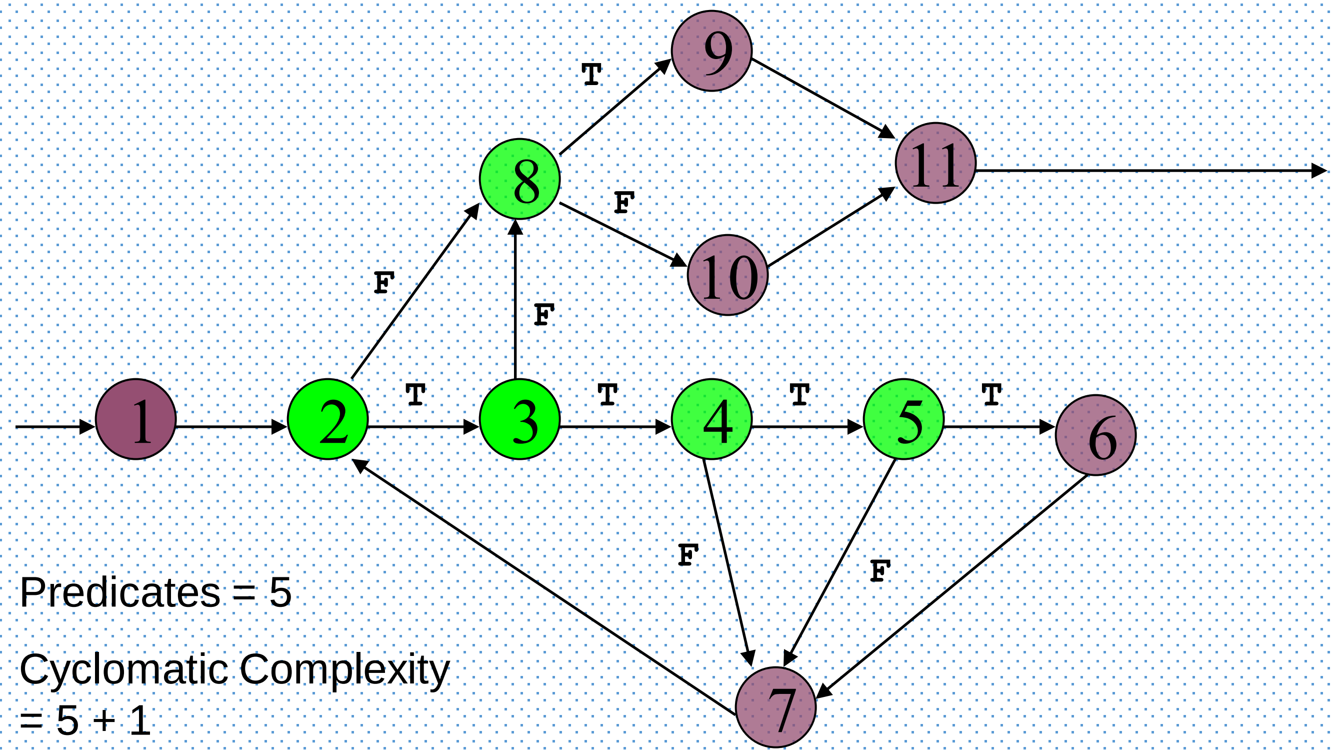
Step 2: Find Cyclomatic Complexity



Regions = 6

Cyclomatic Complexity = 6

Step 2: Find Cyclomatic Complexity



Predicates = 5

Cyclomatic Complexity
= 5 + 1
= 6

CFG

Step 3: Find Basis Path Set

- Find at most 6 independent paths.
- Usually, simpler path == easier to find a test case.
- However, some of the simpler paths are not possible (not realizable):
  - Example: [1 – 2 – 8 – 9 – 11].
    - Not Realizable (i.e., impossible in execution).
    - Verify this by tracing the code.
- Basic Path Set:
  - [1 – 2 – 8 – 10 – 11].
  - [1 – 2 – 3 – 8 – 10 – 11].
  - [1 – 2 – 3 – 4 – 7 – 2 – 8 – 10 – 11].
  - [1 – 2 – 3 – 4 – 5 – 7 – 2 – 8 – 10 – 11].
  - [1 – (2 – 3 – 4 – 5 – 6 – 7) – 2 – 8 – 9 – 11].
- In the last case, (…) represents possible repetition.

Step 4: Derive Test Cases

- Path:
  - [1 – 2 – 8 – 10 – 11]
- Test Case:
  - `value = {…}` irrelevant.
  - `N = 0`
  - `min, max` irrelevant.
- Expected Output:
  - `average = -999`

```
... i = 0;                    1

while (i < N &&                   2
      value[i] != -999) {
     ......
}
if (totalValid > 0)                  8
     ......
else
     mean = -999;                 10

return mean;              11
```

Step 4: Derive Test Cases

- Path:
  - [1 – 2 – 3 – 8 – 10 – 11]
- Test Case:
  - `value = {-999}`
  - `N = 1`
  - `min, max` irrelevant
- Expected Output:
  - `average = -999`

```
... i = 0;                      1

while (i < N &&                 2
       value[i] != -999)        3
     ......
}
if (totalValid > 0)             8
     ......
else
     mean = -999;               10
                                11
return mean;
```

Step 4: Derive Test Cases

- Path:
  - [1 – 2 – 3 – 4 – 7 – 2 – 8 – 10 – 11]
- Test Case:
  - A single value in the `value[ ]` array which is smaller than *min.*
  - `value = { 25 }, N = 1, min = 30, max` irrelevant.
- Expected Output:
  - `average = -999`

- Path:
  - [1 – 2 – 3 – 4 – 5 – 7 – 2 – 8 – 10 – 11]
- Test Case:
  - A single value in the `value[ ]` array which is larger than *max.*
  - `value = { 99 }, N = 1, max = 90, min` irrelevant.
- Expected Output:
  - `average = -999`

Step 4: Derive Test Cases

- Path:
  - [1 – 2 – 3 – 4 – 5 – 6 – 7 – 2 – 8 – 9 – 11]
- Test Case:
  - A single valid value in the `value[ ]` array.
  - `value = { 25 }, N = 1, min = 0, max = 100`
- Expected Output:
  - `average = 25`

- Path:
  - [1 – 2 – 3 – 4 – 5 – 6 – 7 – 2 – 3 – 4 – 5 – 6 – 7 – 2 – 8 – 9 – 11]
- Test Case:
  - Multiple valid values in the `value[ ]` array.
  - `value = { 25, 75 }, N = 2, min = 0, max = 100`
- Expected Output:
  - `average = 50`

Case Studies

Write code fragments for **Binary Search** and **Bubble Sort** programs and perform “Basis Path Testing” by doing the following:

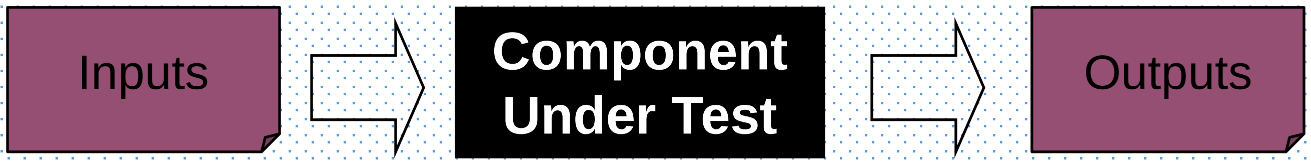
1. Draw the CFG for the code fragment.
2. Compute the cyclomatic complexity number for the CFG.
3. Find the Basis Paths Set.
4. Design test cases along paths in the Basis Paths Set.

Summary of Basis Path Testing

- A simple test that:
  - Cover all statements.
  - Exercise all decisions (conditions).
- The cyclomatic complexity is an **upperbound** of the independent paths needed to cover the CFG.
  - If more paths are needed, then either cyclomatic complexity is wrong, or the paths chosen are incorrect.
- Although picking a complicated path that covers more than one unvisited edge is possible all times, it is not encouraged:
  - May be hard to design the test case.

Black Box Testing

- Test Engineers have no access to the source code or documentation of internal working.
- The "Black Box" can be:
  - A single unit.
  - A subsystem.
  - The whole system.
- Tests are based on:
  - Specification of the "Black Box".
  - Providing **inputs** to the "Black Box" and inspect the **outputs.**



Black Box Testing

- Two techniques will be covered for the black box testing in this course:
  - Equivalence Partition;
  - Boundary Value Analysis.

Equivalence Partition

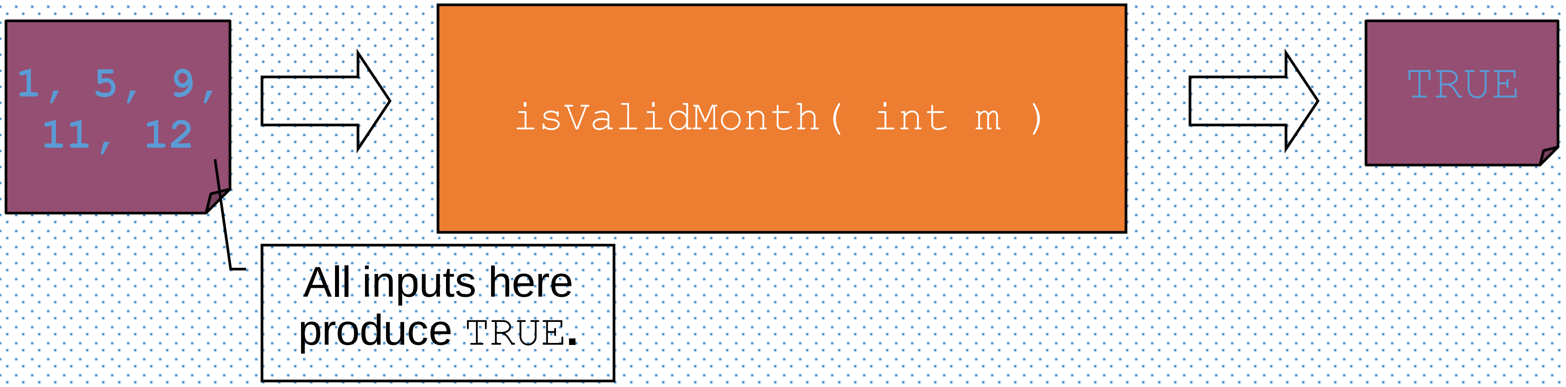
- To ensure the correct behavior of a “black box”, both valid and invalid cases need to be tested.
- Example:
  - Given the method below:

```
boolean isValidMonth(int m)

Functionality: check m is [1..12]
Output:
       - true if m is 1 to 12
       - false otherwise
```

Is there a better way to test other than testing **all** integer values [-2^{31}, …, $2^{31}-1$] ?

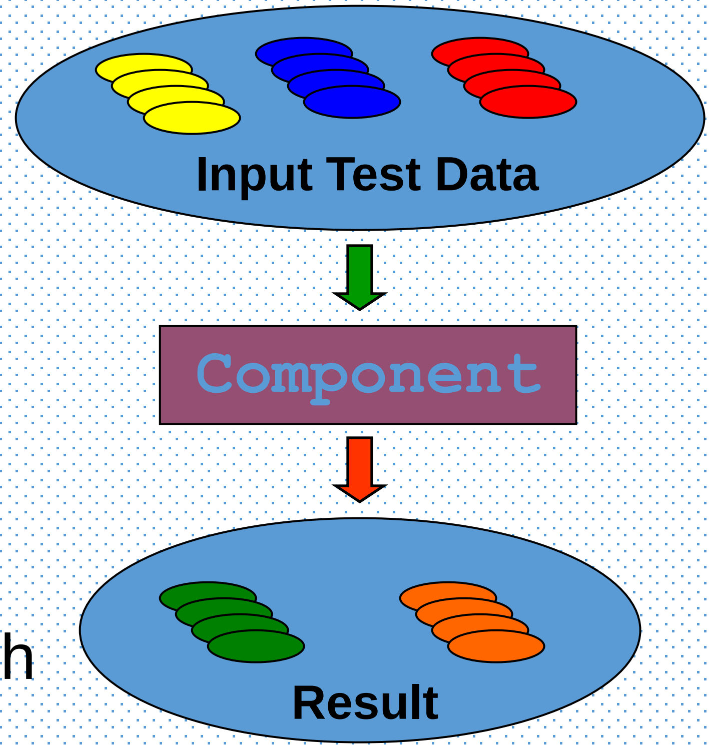
Equivalence Partition

- Observations:
  - For a method, it is common to have a number of inputs that produce similar outcomes.
  - Testing one of the inputs *should be* as good as exhaustively testing all of them.
  - So, pick only a few test cases from each “category” of input that produce the same output.



Equivalence Partition

- Partition input data into *equivalence classes.*
- Data in each equivalence class:
  - Likely to be treated equally by a reasonable algorithm.
  - Produce same output state, i.e., valid/invalid.
- Derive test data for each class.



Example isValid(Month)

- For the *is ValidMonth* example:
  - Input value `[1 … 12]` should get a similar treatment.
  - Input values lesser than `1`, larger than `12` are two other groups.
- Three partitions:
  - `[ -∞ ... 0 ]` should produce an **invalid** result
  - `[ 1 … 12 ]` should produce a **valid** result
  - `[ 13 … ∞ ]` should produce an **invalid** result
- Pick one value from each partition as test case:
  - E.g., `{-12, 5, 15}`
  - Reduce the number of test cases significantly.

Example - 2

Problem statement for Triangle Problem

The program reads three input numbers that represent the lengths of the three sides of a triangle. Based on these three input values, the program determines whether the triangle is scalene (that is, it has three unequal sides), isosceles (two equal sides), or equilateral (three equal sides). The program displays the result on the screen.

Triangle Problem

Input: Three integers a, b, c which represent the sides of a triangle.

Output: Type of triangle: “Equilateral”, “Isosceles”, “Scalene”, or “Not a Triangle”.

The integers must satisfy the following conditions:

C1: $a<b+c$

C2: $b<a+c$

C3: $c<a+b$

If all three sides are equal -> Equilateral triangle

If exactly one pair of sides are equal -> Isosceles triangle

No pair of sides are equal -> Scalene triangle

If conditions C1 or C2 or C3 fails -> Not a Triangle

Triangle Problem

- Example: High-level **classes** for the Triangle Problem:

- Class 1: **Equilateral Triangle**. Test case is triple (5,5,5)
- Class 2: **Isosceles Triangle**. Test case is triple (5,5,6)
- Class 3: **Scalene Triangle**. Test case is triple (5,6,7)
- Class 4: **Not a Triangle**. Test case is triple (3,15,11)

Boundary Value Analysis

- It has been found that most errors are caught at the **boundary** of the equivalence classes.
- Not surprising, as the end points of the boundary are usually used in the code for checking:
    - E.g., checking `K` is in range [`X ... Y`):

```
if (K >= X && K <= Y)
    ...
```

Easy to make mistake on the comparison.

Using Boundary Value Analysis

- If the component specifies a range, [`X ... Y`]
  - Four values should be tested:
    - Valid: `X` and `Y`
    - Invalid: Just below `X` (e.g., `X - 1`)
    - Invalid: Just above `Y` (e.g., `Y + 1`)
  - E.g., [`1 ... 12`]
    - Test Data: `{0, 1, 12, 13}`
- Similar for open interval (`X ... Y`), i.e., `X` and `Y` not inclusive.
  - Four values should be tested:
    - Invalid: `X` and `Y`
    - Valid: Just **above** `X` (e.g., `X + 1`)
    - Valid: Just **below** `Y` (e.g., `Y - 1`)
  - E.g., (`100 ... 200`)
    - Test Data: `{100, 101, 199, 200}`

PRESIDENCY UNIVERSITY

Boundary Value Test Cases for Triangle Problem

Triangle Side boundary conditions:

Each side between 1 and 200

A, B, C sides are taking values alternatively, while other two sides are having nominal values

- Min (1)
- Min+1 (2)
- Nominal (100)
- Max-1 (199)
- Max (200)

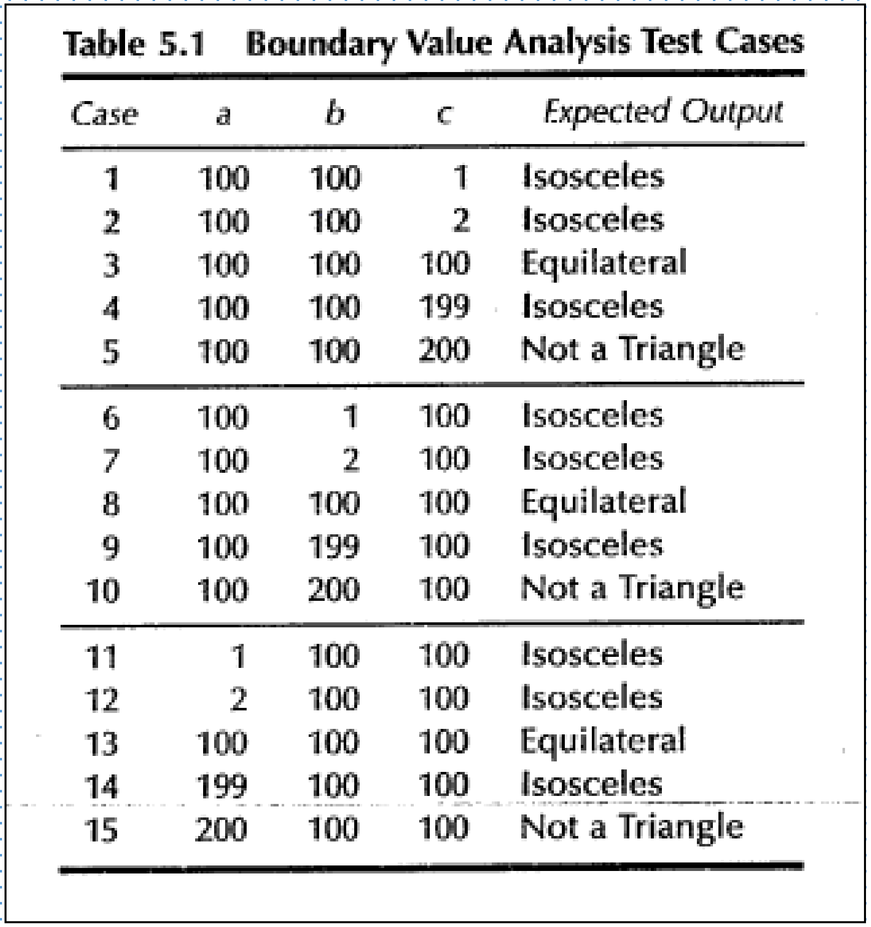


Table 5.1 Boundary Value Analysis Test Cases

Case	*a*	*b*	*c*	*Expected Output*
1	100	100	1	Isosceles
2	100	100	2	Isosceles
3	100	100	100	Equilateral
4	100	100	199	Isosceles
5	100	100	200	Not a Triangle
6	100	1	100	Isosceles
7	100	2	100	Isosceles
8	100	100	100	Equilateral
9	100	199	100	Isosceles
10	100	200	100	Not a Triangle
11	1	100	100	Isosceles
12	2	100	100	Isosceles
13	100	100	100	Equilateral
14	199	100	100	Isosceles
15	200	100	100	Not a Triangle

Example 1 - Searching

```
boolean Search(
        List aList, int key)
```

Precondition:

- `aList` has at least one element

Postcondition:

- `true` if `key` is in the `aList`
- `false` if `key` is not in `aList`

Equivalence Classes for Search

- Sequence with a single value:
  - key found.
  - key not found.
- Sequence of multi values:
  - key found:
    - First element in sequence.
    - Last element in sequence.
    - “Middle” element in sequence.
  - key not found.

Test Data for Search

Test Case	aList	Key	Expected Result
1	[123]	123	True
2	[123]	456	False
3	[1, 6, 3, -4, 5]	1	True
4	[1, 6, 3, -4, 5]	5	True
5	[1, 6, 3, -4, 5]	3	True
6	[1, 6, 3, -4, 5]	123	False

Example 2 – Simple Shopping Scenario

Let us consider a simple shopping scenario -

- Shop for $1000 and get 5% discount
- Shop for $2000 and get 7% discount
- Shop for $3000 and above and get 10% discount.

Example 2 – Simple Shopping Scenario

- **Equivalence partitioning** - We can divide the input into four partitions, amount<0, 0 to 1000, 1001 to 2000, 2001 to 3000 and then 3001 and above.
- We will keep things simple for now and will not go into details like the max amount one can shop for, or what about cents and all that stuff.
- Adding the flavor of **boundary value to the above partitions** - the boundary values would be 0, 1000, 1001, 2000, 2001, 3000 and 3001. In boundary value analysis, immediate upper and lower values too are recommended to be tested, so we will include -1, 1 and 999. These are the values that can tell behavior the software for almost entire range of input values, but there can be other issues in software. For that reason, other input values too can be considered.

Example 3 – Quadratic Equation

- For example, suppose we are testing a function that uses the quadratic formula to determine the two roots of a second-degree polynomial $\mathbf{a}x^2\mathbf{+b}x\mathbf{+c}$. For simplicity, assume that we are going to work only with real numbers, and print an error message if it turns out that the two roots are complex numbers (numbers involving the square root of a negative number).

- We can come up with test data for each of the four cases, based on values of the polynomial's *discriminant* ($\mathbf{b^2-4ac}$).

Example 3 – Quadratic Equation

- Easy data (discriminant is a perfect square)
- Typical data (discriminant is positive):

a	b	c	Roots
1	2	1	-1, -1
1	3	2	-1, -2

a	b	c	Roots
1	4	1	-3.73205, -0.267949
2	4	1	-1.70711, -0.292893

Example 3 – Quadratic Equation

- Boundary / extreme data (discriminant is zero):

a	b	c	Roots
1	1	1	square root of negative number
0	1	1	division by zero

- Bogus data (discriminant is negative, or **a** is zero):

a	b	c	Roots
2	-4	2	1, 1
2	-8	8	2, 2

Example 4 – Factorial of n: n!

- Equivalence partitioning – break the input domain into different classes:
  - Class1: n<0
  - Class2: n>0 and n! doesn’t cause an overflow
  - Class3: n>0 and n! causes an overflow
- Boundary Value Analysis:
  - n=0 (between class1 and class2)

Factorial of n: n!
Test Cases

- Test case = (ins, expected outs)

- Equivalence partitioning – break the input domain into different classes:

  1. From Class1: ((n = -1), " function not defined for n negative")
  2. From Class2: ((n = 3), 6)
  3. From Class3: ((n=100), " input value too big")

- Boundary Value Analysis:

  4. ((n=0), 1)

Module 3: Software Testing and Quality (8 hrs) – Comprehension level

Introduction to Software Testing: verification and validation, Test Strategies for conventional Software, Validation Testing, White box Testing: Basis path testing, Black box Testing. Software Quality Assurance : Elements of software quality assurance, SQA Tasks, Goals and Metrics, Software configuration management : SCM process.

Software Quality Assurance (SQA)

Department of Computer Science and Engineering

School of Engineering, Presidency University

Software Quality Assurance (SQA)

Contents

Elements of Software Quality Assurance (SQA)

- **Standards**
- **Reviews and Audits**
- **Testing**
- **Error/defect collection and analysis**
- **Change management**
- **Education**
- **Vendor management**
- **Security management**
- **Safety**
- **Risk management**

Role of the SQA Group - I

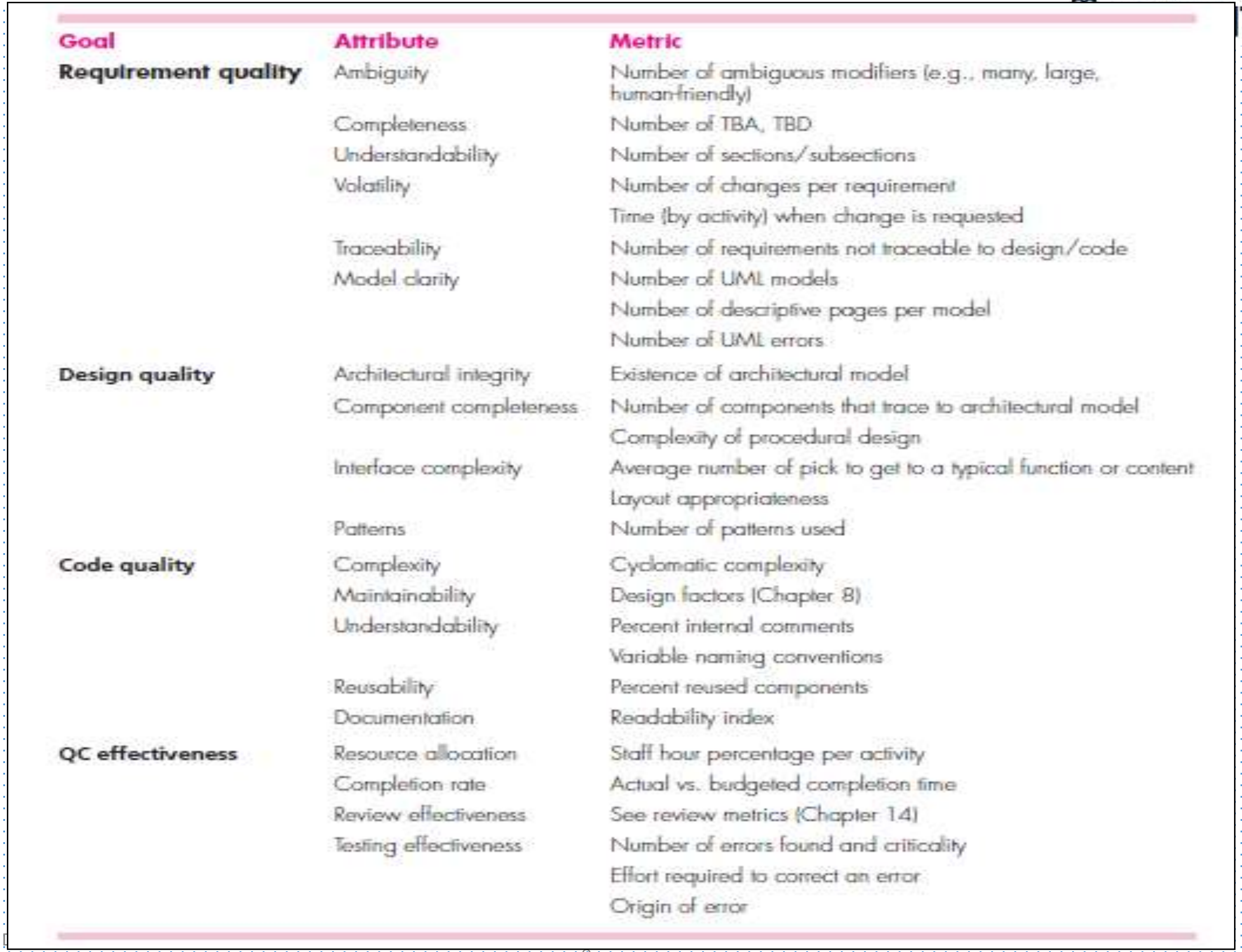
- **Prepares an SQA plan for a project.**
  - The plan identifies
    - evaluations to be performed
    - audits and reviews to be performed
    - standards that are applicable to the project
    - procedures for error reporting and tracking
    - documents to be produced by the SQA group
    - amount of feedback provided to the software project team
- **Participates in the development of the project’s software process description.**
  - The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan.

Role of the SQA Group - II

- **Reviews software engineering activities to verify compliance with the defined software process.**
  - identifies, documents, and tracks deviations from the process and verifies that corrections have been made.
- **Audits designated software work products to verify compliance with those defined as part of the software process.**
  - reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made
  - periodically reports the results of its work to the project manager.
- **Ensures that deviations in software work and work products are documented and handled according to a documented procedure.**
- **Records any noncompliance and reports to senior management.**
  - Noncompliance items are tracked until they are resolved.

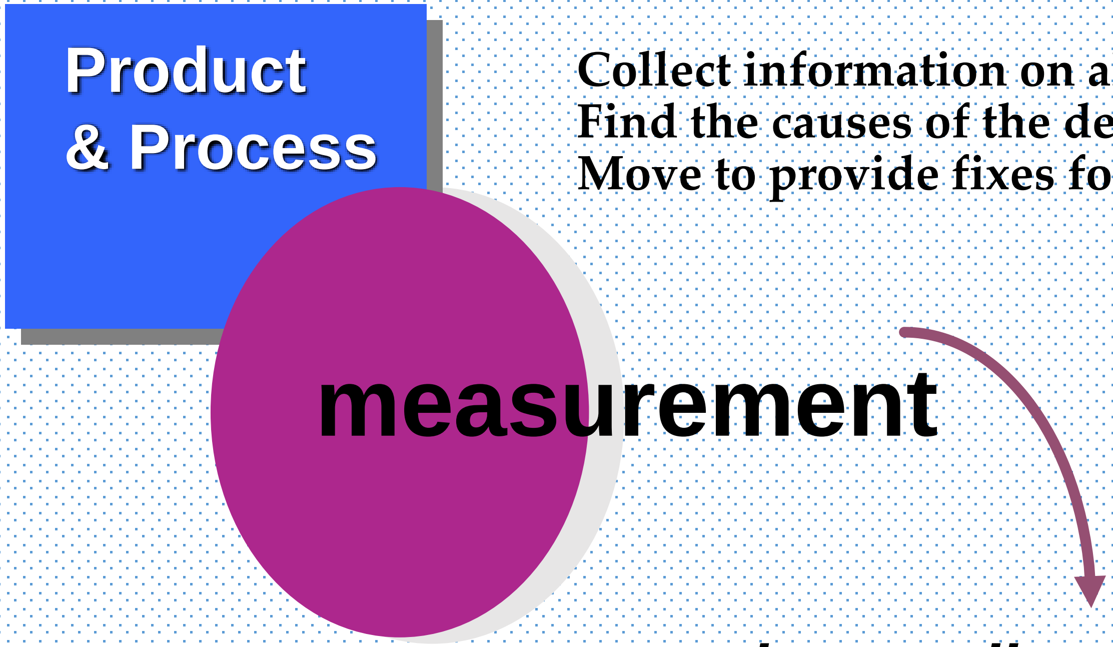
SQA Goals

- **Requirements quality.** The correctness, completeness, and consistency of the requirements model will have a strong influence on the quality of all work products that follow.
- **Design quality.** Every element of the design model should be assessed by the software team to ensure that it exhibits high quality and that the design itself conforms to requirements.
- **Code quality.** Source code and related work products (e.g., other descriptive information) must conform to local coding standards and exhibit characteristics that will facilitate maintainability.
- **Quality control effectiveness.** A software team should apply limited resources in a way that has the highest likelihood of achieving a high quality result.



Goal	Attribute	Metric
Requirement quality	Ambiguity	Number of ambiguous modifiers (e.g., many, large, human-friendly)
	Completeness	Number of TBA, TBD
	Understandability	Number of sections/subsections
	Volatility	Number of changes per requirement
		Time (by activity) when change is requested
	Traceability	Number of requirements not traceable to design/code
	Model clarity	Number of UML models
		Number of descriptive pages per model
		Number of UML errors
Design quality	Architectural integrity	Existence of architectural model
	Component completeness	Number of components that trace to architectural model
		Complexity of procedural design
	Interface complexity	Average number of pick to get to a typical function or content
		Layout appropriateness
	Patterns	Number of patterns used
Code quality	Complexity	Cyclomatic complexity
	Maintainability	Design factors (Chapter 8)
	Understandability	Percent internal comments
		Variable naming conventions
	Reusability	Percent reused components
	Documentation	Readability index
QC effectiveness	Resource allocation	Staff hour percentage per activity
	Completion rate	Actual vs. budgeted completion time
	Review effectiveness	See review metrics (Chapter 14)
	Testing effectiveness	Number of errors found and criticality
		Effort required to correct an error
		Origin of error

Statistical SQA



Collect information on all defects
Find the causes of the defects
Move to provide fixes for the process

... an understanding of how to improve quality ...

Statistical SQA

- Information about software errors and defects is collected and categorized.
- An attempt is made to trace each error and defect to its underlying cause (e.g., non-conformance to specifications, design error, violation of standards, poor communication with the customer).
- Using the **Pareto principle** (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the *vital few*).
- Once the vital few causes have been identified, move to correct the problems that have caused the errors and defects.

Six Sigma for Software Engineering

- The term “six sigma” is derived from six standard deviations - 3.4 instances (defects) per million occurrences - implying an extremely high quality standard.
- The Six Sigma methodology defines five core steps:
  - *Define* customer requirements and deliverables and project goals via well-defined methods of customer communication.
  - *Measure* the existing process and its output to determine current quality performance (collect defect metrics).
  - *Analyze* defect metrics and determine the vital few causes.
  - *Improve* the process by eliminating the root causes of defects.
  - *Control* the process to ensure that future work does not reintroduce the causes of defects.

Software Safety

- *Software safety* is a software quality assurance activity that focuses on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail.
- If hazards can be identified early in the software process, software design features can be specified that will either eliminate or control potential hazards.

Case Study

Assume that you are a student member of “Software Quality - Assurance” Cell in Presidency University.

List the activities that you would perform to ensure that the functioning of the CSE department is as per standard quality requirements.

Software Quality Concepts

Contents

What is Quality and Views of Quality

- The *American Heritage Dictionary* defines *quality* as
  - "a characteristic or attribute of something."

Views of Quality

- The *transcendental view* argues that quality is something that you immediately recognize, but cannot explicitly define.
- The *user view* sees quality in terms of an end-user's specific goals. If a product meets those goals, it exhibits quality.
- The *manufacturer's view* defines quality in terms of the original specification of the product. If the product conforms to the spec, it exhibits quality.
- The *product view* suggests that quality can be tied to inherent characteristics (e.g., functions and features) of a product.

Why is Quality of "Software" Important

- In 2005, *ComputerWorld* lamented that
  - "bad software plagues nearly every organization that uses computers, causing lost work hours during computer downtime, lost or corrupted data, missed sales opportunities, high IT support and maintenance costs, and low customer satisfaction.
- A year later, *InfoWorld* wrote about the
  - "the sorry state of software quality" reporting that the quality problem had not gotten any better.
- Today, software quality remains an issue, but who is to blame?
  - Customers blame developers, arguing that sloppy practices lead to low-quality software.
  - Developers blame customers (and other stakeholders), arguing that irrational delivery dates and a continuing stream of changes force them to deliver software before it has been fully validated.

Definition of Software Quality

- Software quality can be defined as:
  - *An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.*
  - Key terms are:
    - Effective Software Process
    - Useful Product
    - Measurable Value

Effective Software Process

- An *effective software process* establishes the infrastructure that supports any effort at building a high quality software product.
- The management aspects of process create the checks and balances that help avoid project chaos - a key contributor to poor quality.
- Software engineering practices allow the developer to analyze the problem and design a solid solution - both critical to building high quality software.
- Finally, umbrella activities such as change management and technical reviews have as much to do with quality as any other part of software engineering practice.

Useful Product

- A *useful product* delivers the content, functions, and features that the end-user desires.
- But as important, it delivers these assets in a reliable, error free way.
- A useful product always satisfies those requirements that have been explicitly stated by stakeholders.
- In addition, it satisfies a set of implicit requirements (e.g., ease of use) that are expected of all high quality software.

Measureable Value

- By *adding measurable value for both the producer and user* of a software product, high quality software provides benefits for the software organization and the end-user community.
- The software organization gains added value because high quality software requires less maintenance effort, fewer bug fixes, and reduced customer support.
- The user community gains added value because the application provides a useful capability in a way that expedites some business process.

Quality Dimensions

- David Garvin:
  - **Performance Quality.** Does the software deliver all content, functions, and features that are specified as part of the requirements model in a way that provides value to the end-user?
  - **Feature quality.** Does the software provide features that surprise and delight first-time end-users?
  - **Reliability.** Does the software deliver all features and capability without failure? Is it available when it is needed? Does it deliver functionality that is error free?
  - **Conformance.** Does the software conform to local and external software standards that are relevant to the application? Does it conform to de facto design and coding conventions? For example, does the user interface conform to accepted design rules for menu selection or data input?

Quality Dimensions

- **Durability.** Can the software be maintained (changed) or corrected (debugged) without the inadvertent generation of unintended side effects? Will changes cause the error rate or reliability to degrade with time?
- **Serviceability.** Can the software be maintained (changed) or corrected (debugged) in an acceptably short time period. Can support staff acquire all information they need to make changes or correct defects?
- **Aesthetics**. Most of us would agree that an aesthetic entity has a certain elegance, a unique flow, and an obvious "presence" that are hard to quantify but evident nonetheless.

Software Quality Dilemma

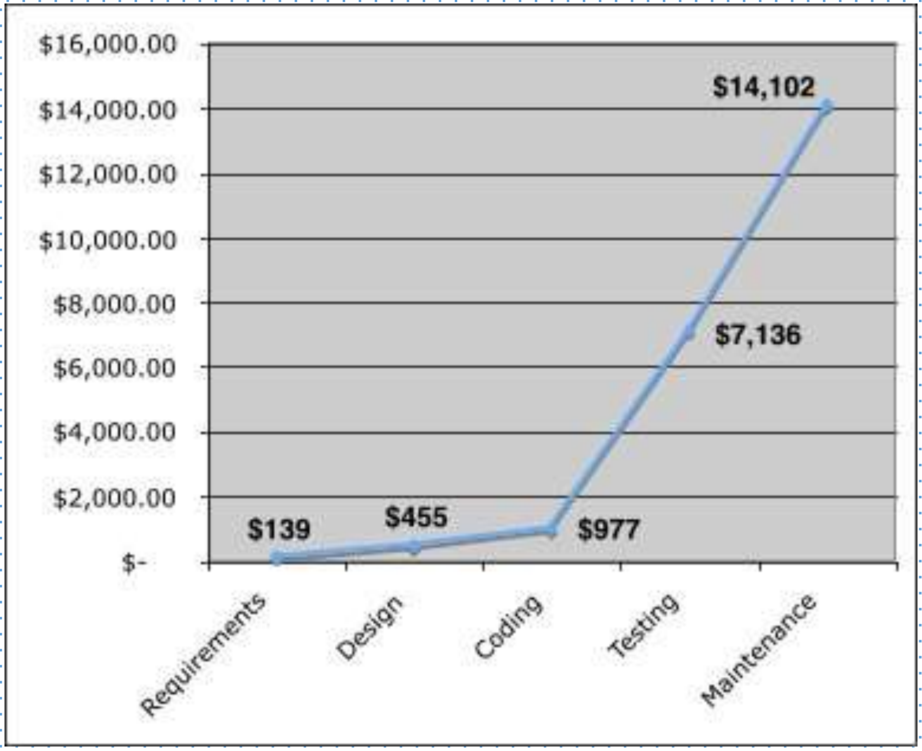
- If you produce a software system that has **terrible quality**, you lose because no one will want to buy it.
- If on the other hand you spend infinite time, extremely large effort, and huge sums of money to build the **absolutely perfect piece of software**, then it's going to take so long to complete and it will be so expensive to produce that you'll be out of business anyway.
- Either you missed the market window, or you simply exhausted all your resources.
- **Magical Middle Ground -** So people in industry try to get to that *magical middle ground* where the product is good enough not to be rejected right away, such as during evaluation, but also not the object of so much perfectionism and so much work that it would take too long or cost too much to complete.

Cost of Quality

- *Prevention costs* include
  - quality planning
  - formal technical reviews
  - test equipment
  - Training
- *Internal failure costs* include
  - rework
  - repair
  - failure mode analysis
- *External failure costs* are
  - complaint resolution
  - product return and replacement
  - help line support
  - warranty work

Cost of Quality

- The relative costs to find and repair an error or defect increase dramatically as we go from prevention to detection to internal failure to external failure costs.



Quality and Risk

- *People bet their jobs, their comforts, their safety, their entertainment, their decisions, and their very lives on computer software. It better be right."*
- Example:
  - *Throughout the month of November, 2000 at a hospital in Panama, 28 patients received massive overdoses of gamma rays during treatment for a variety of cancers. In the months that followed, five of these patients died from radiation poisoning and 15 others developed serious complications. What caused this tragedy? A software package, developed by a U.S. company, was modified by hospital technicians to compute modified doses of radiation for each patient.*

Quality and Security

Gary McGraw comments:

- "Software security relates entirely and completely to quality. You must think about security, reliability, availability, dependability - at the beginning, in the design, architecture, test, and coding phases, all through the software life cycle [process]. Even people aware of the software security problem have focused on late life-cycle stuff. The earlier you find the software problem, the better. And there are two kinds of software problems. One is bugs, which are implementation problems. The other is software flaws - architectural problems in the design. People pay too much attention to bugs and not enough on flaws."

Software Configuration Management (SCM)

Department of Computer Science and Engineering

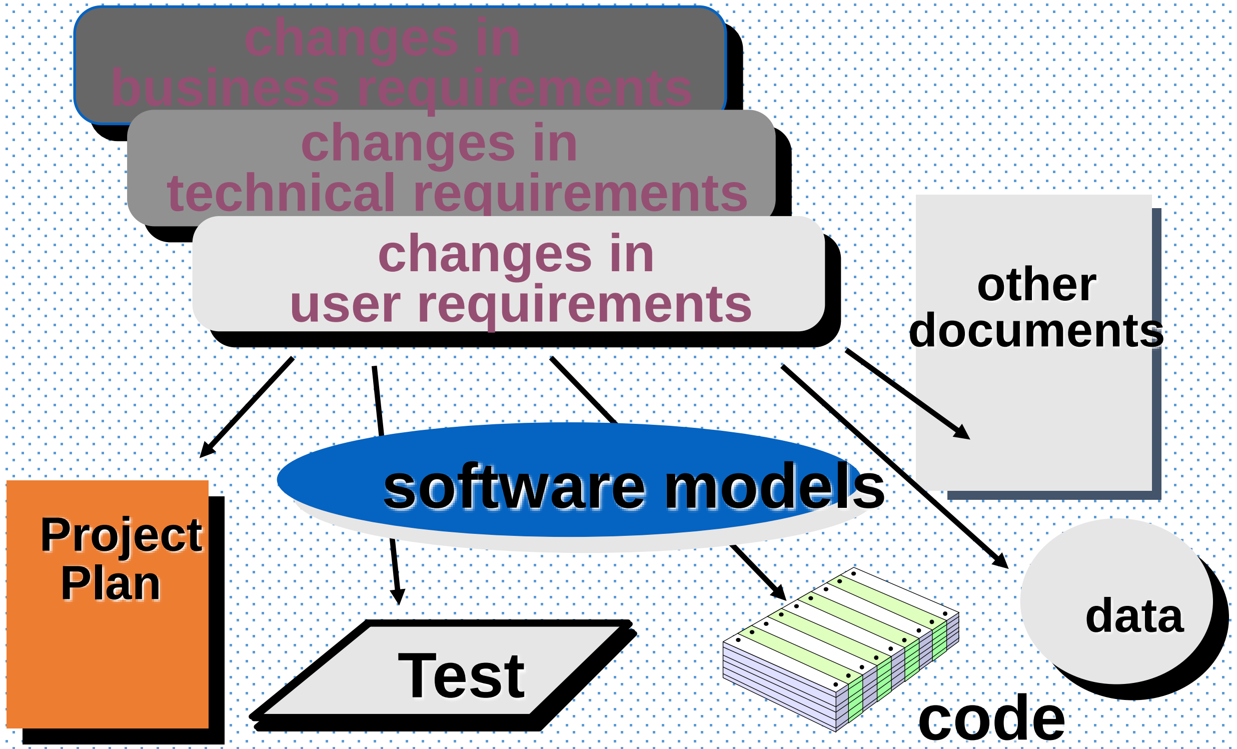
School of Engineering, Presidency University

What is Software Configuration Management (SCM)

SCM is the task of **tracking and controlling changes** in the **software**.

If something in the project goes wrong, SCM can determine what was changed and who changed it.

What can Change in a Project?


changes in business requirements
changes in technical requirements
changes in user requirements
other documents
software models
Project Plan
Test
data
code

Key aspects of SCM

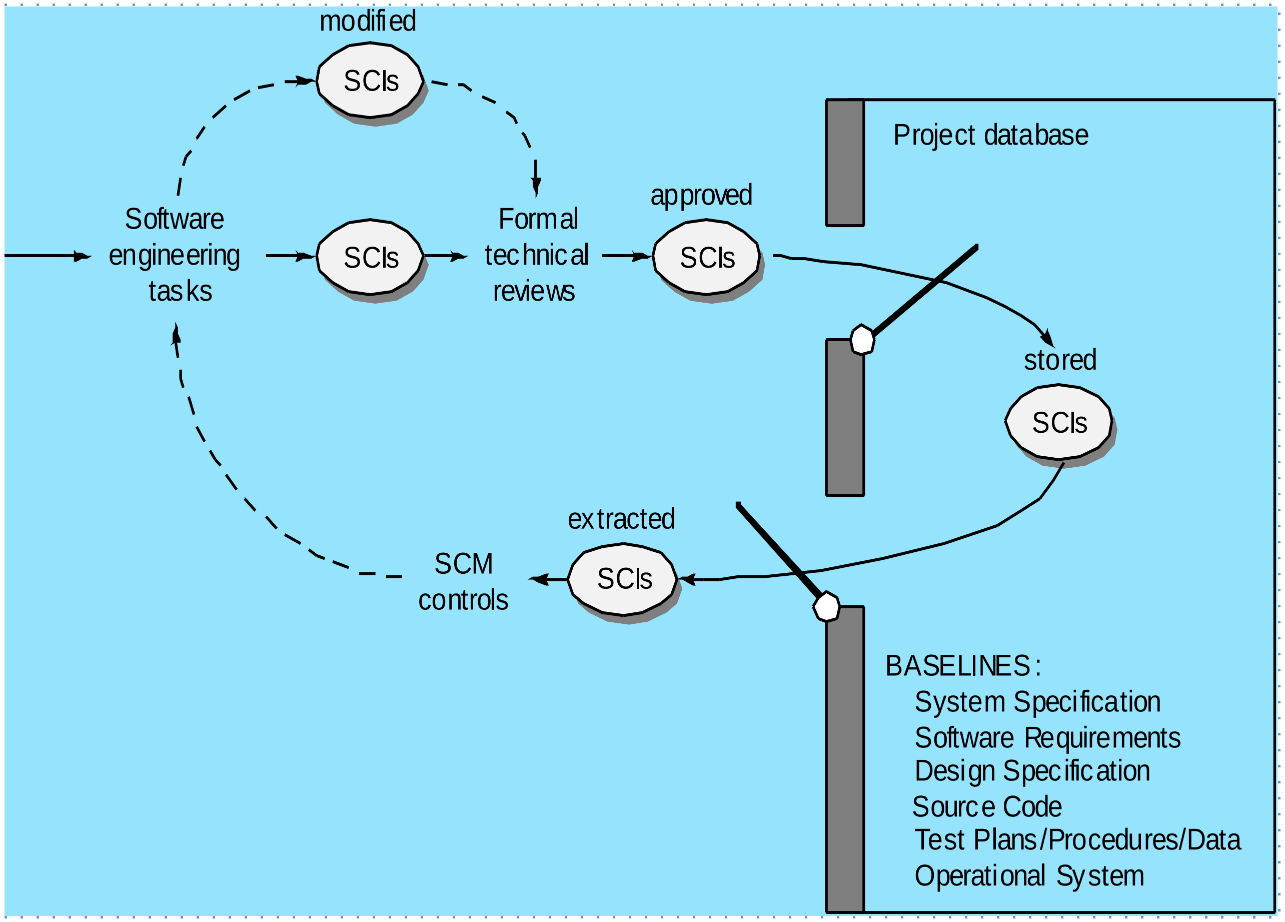
Key aspects of SCM include:

1. Establishing a Baseline for the Project
2. Establishing a SCM Repository for the Project
3. Establishing the SCM Process which includes
    a. Maintaining Change Control
    b. Maintaining Version Control

1. Establishing a Baseline

- The IEEE (IEEE Std. No. 610.12-1990) defines a baseline as:
  - A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.
- A Baseline is a milestone in the development of software that is marked by the delivery of one or more software configuration items and the approval of these SCIs that is obtained through a formal technical review

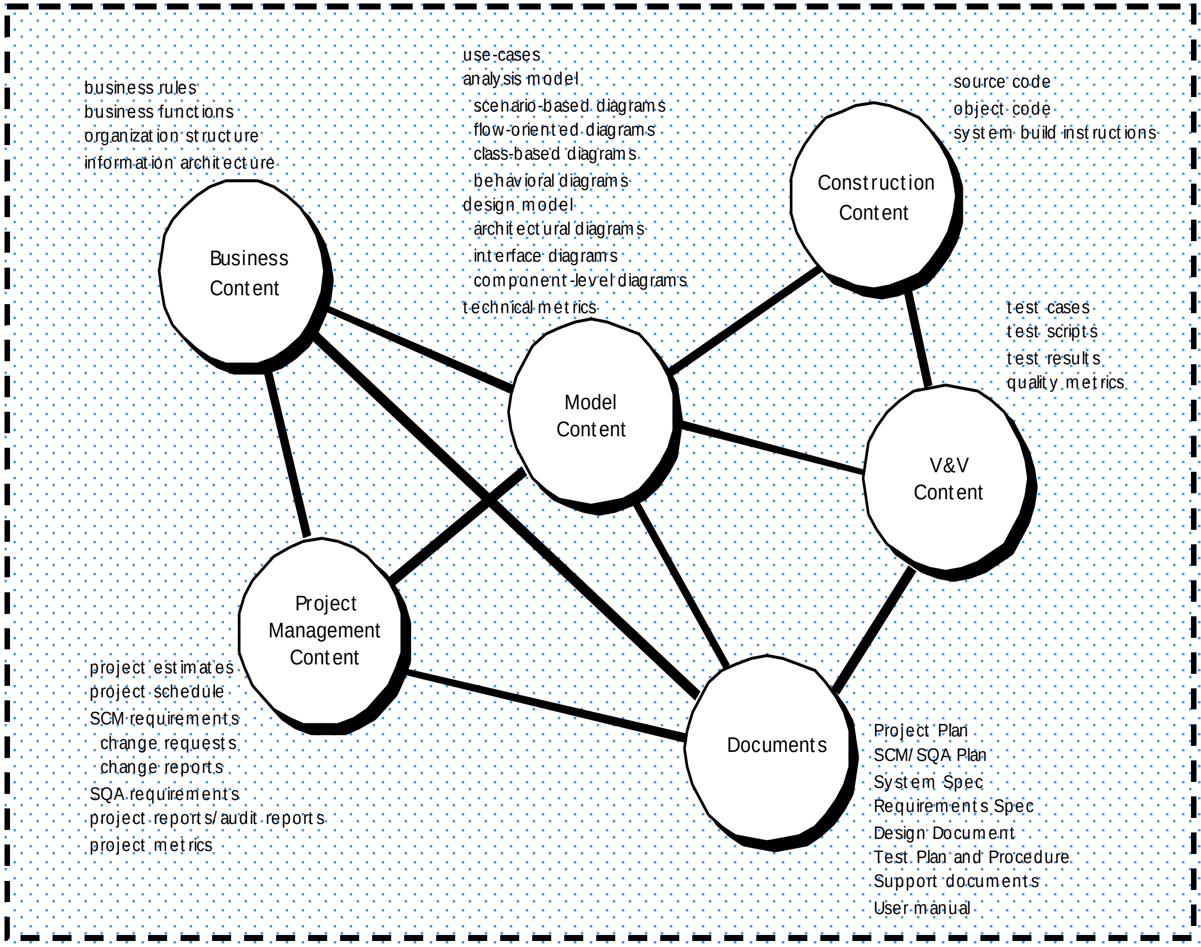
1. Establishing a Baseline



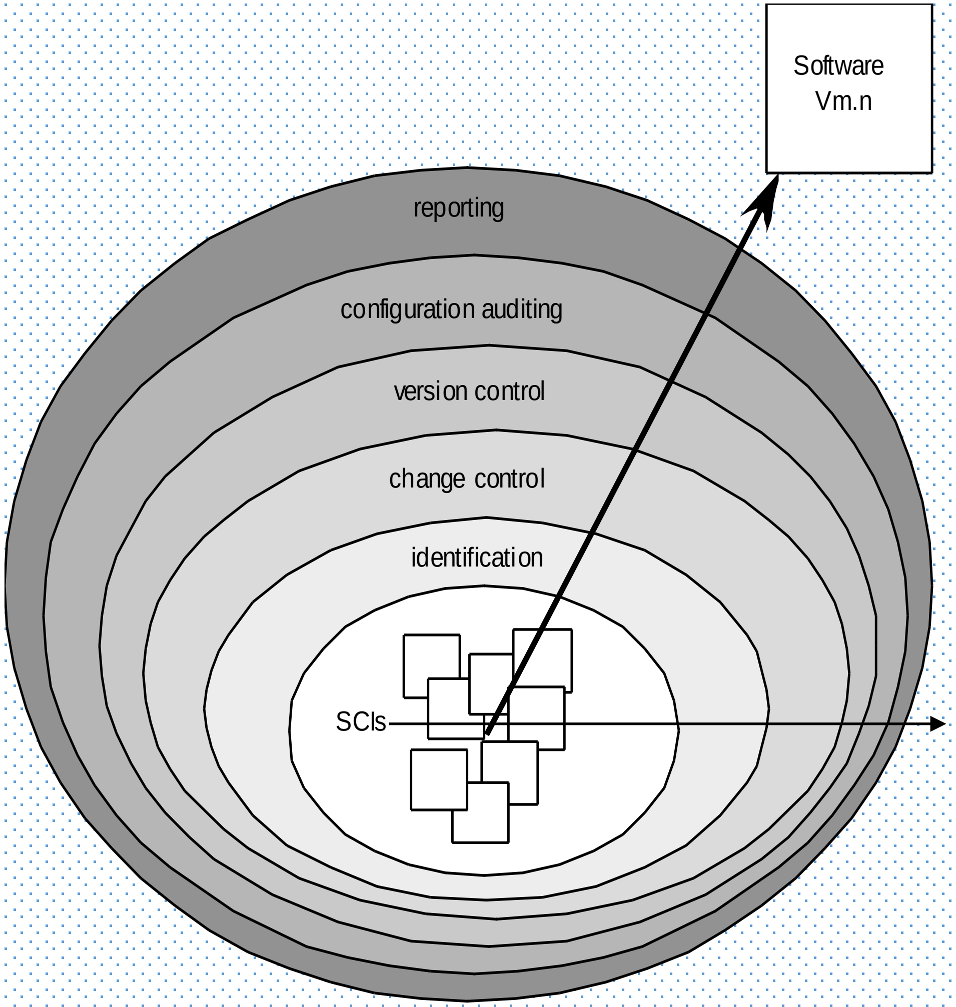
2. Establishing a SCM Repository

- The SCM repository is the set of mechanisms and data structures that allow a software team to manage change in an effective manner
- The repository performs or precipitates the following functions:
    - Data integrity
    - Information sharing
    - Tool integration
    - Data integration
    - Document standardization

2. SCM Repository Contents



3. Establishing the SCM Process for the Project



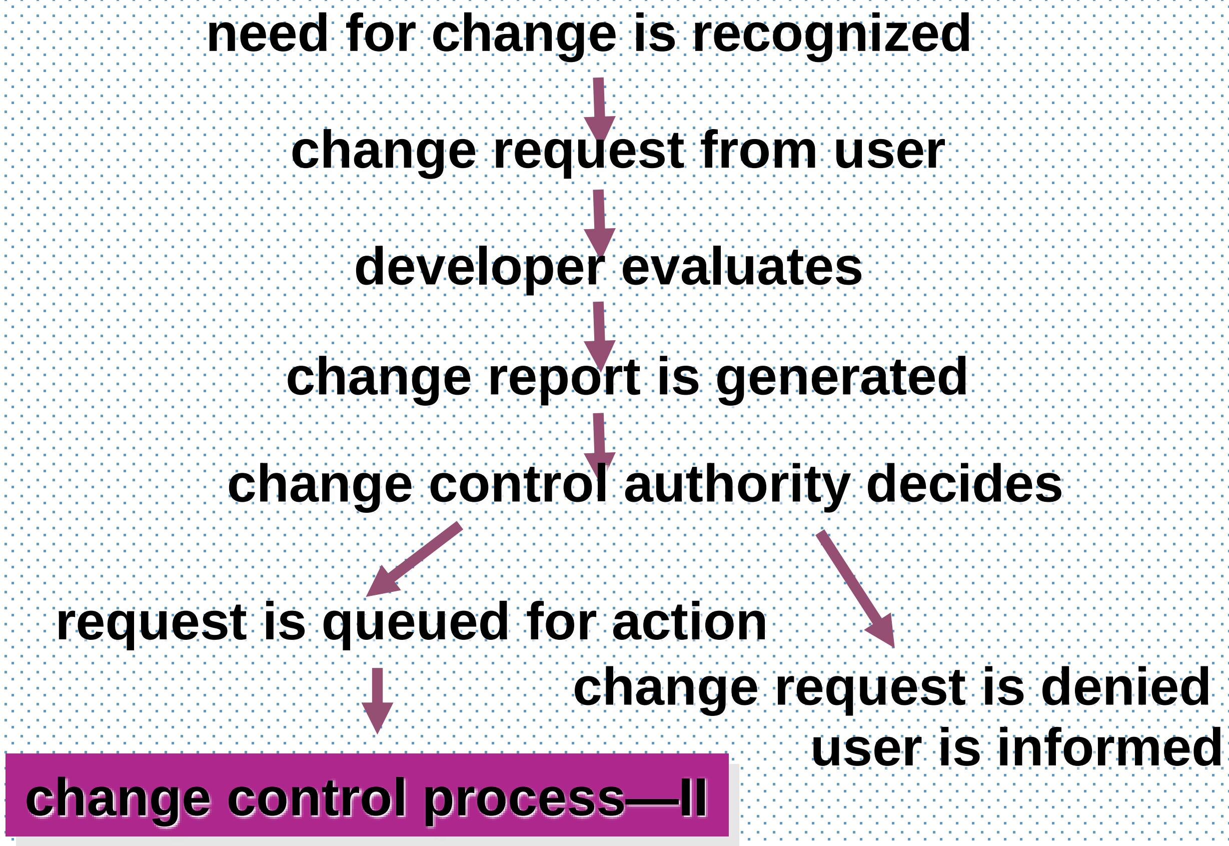
SCI – Software Configuration Item

3.1 Identification of SCI

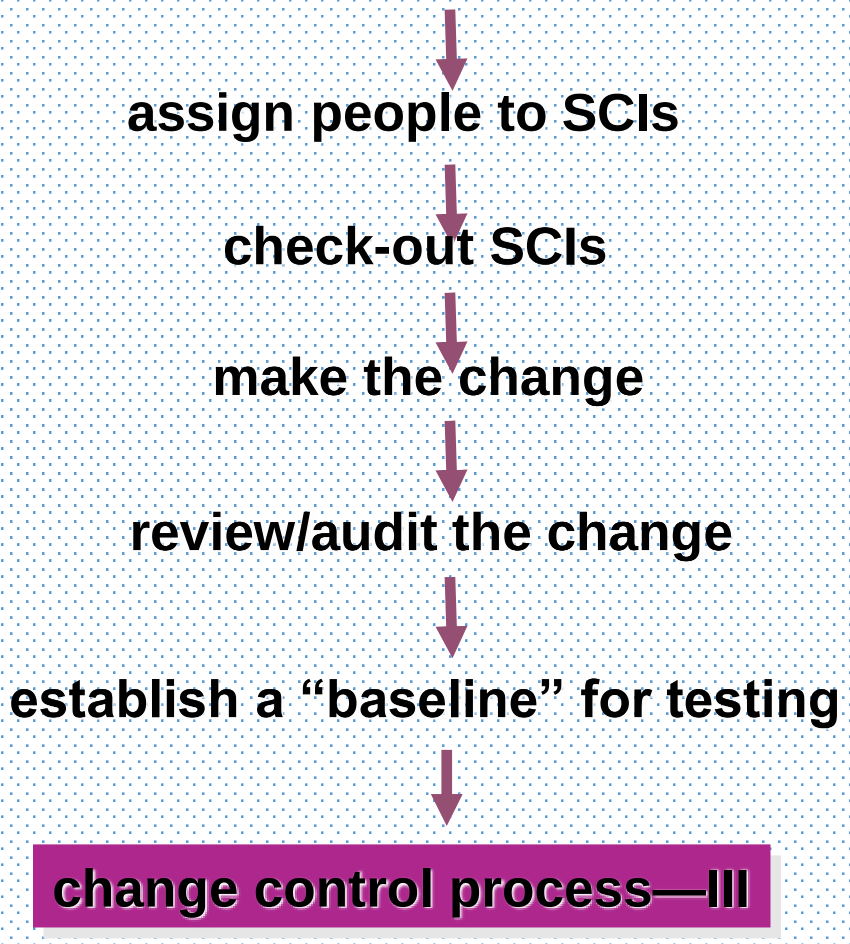
- SCI can include:
  - Business content
  - Model content
  - Construction content
  - V&V (Verification and Validation) content
  - Project Documents

[Refer ‘SCM Repository Content’ diagram on Slide 8]

3.2 Change Control Process


need for change is recognized
change request from user
developer evaluates
change report is generated
change control authority decides
request is queued for action
change request is denied
user is informed
change control process—II

3.2 Change Control Process



3.2 Change Control Process

perform SQA and testing activities

check-in the changed SCIs

promote SCI for inclusion in next release

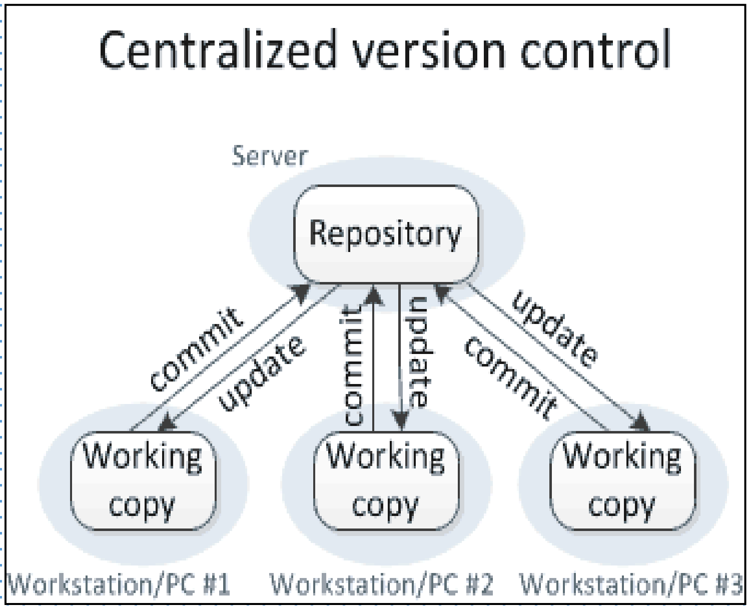
rebuild appropriate version

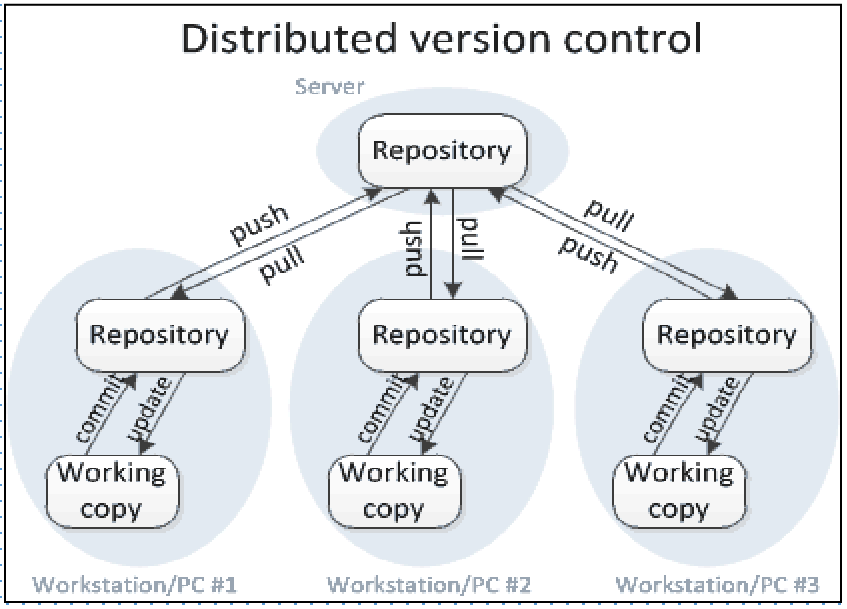
review/audit the change

include all changes in release

3.3 Version Control

Version control is a system that records changes to a file or set of files over time so that you can recall specific **versions** later.
Types: Centralized and Distributed Version Controls

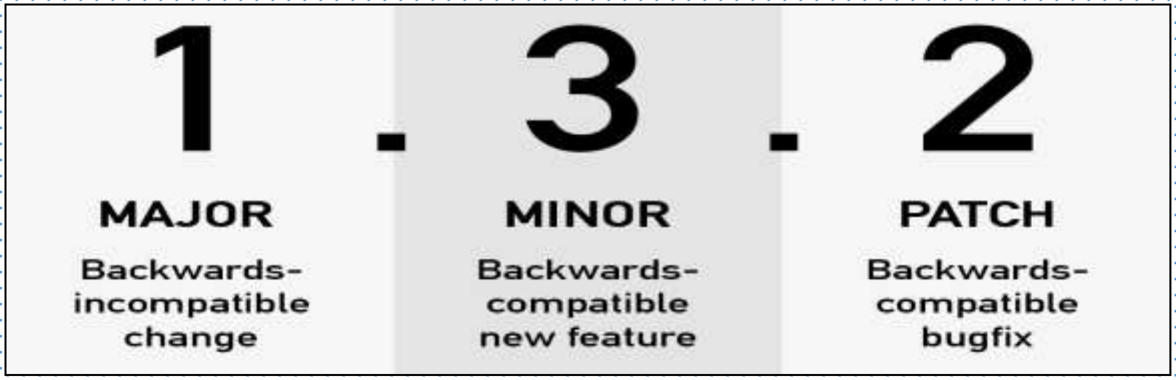




3.3 Numbers in a Version

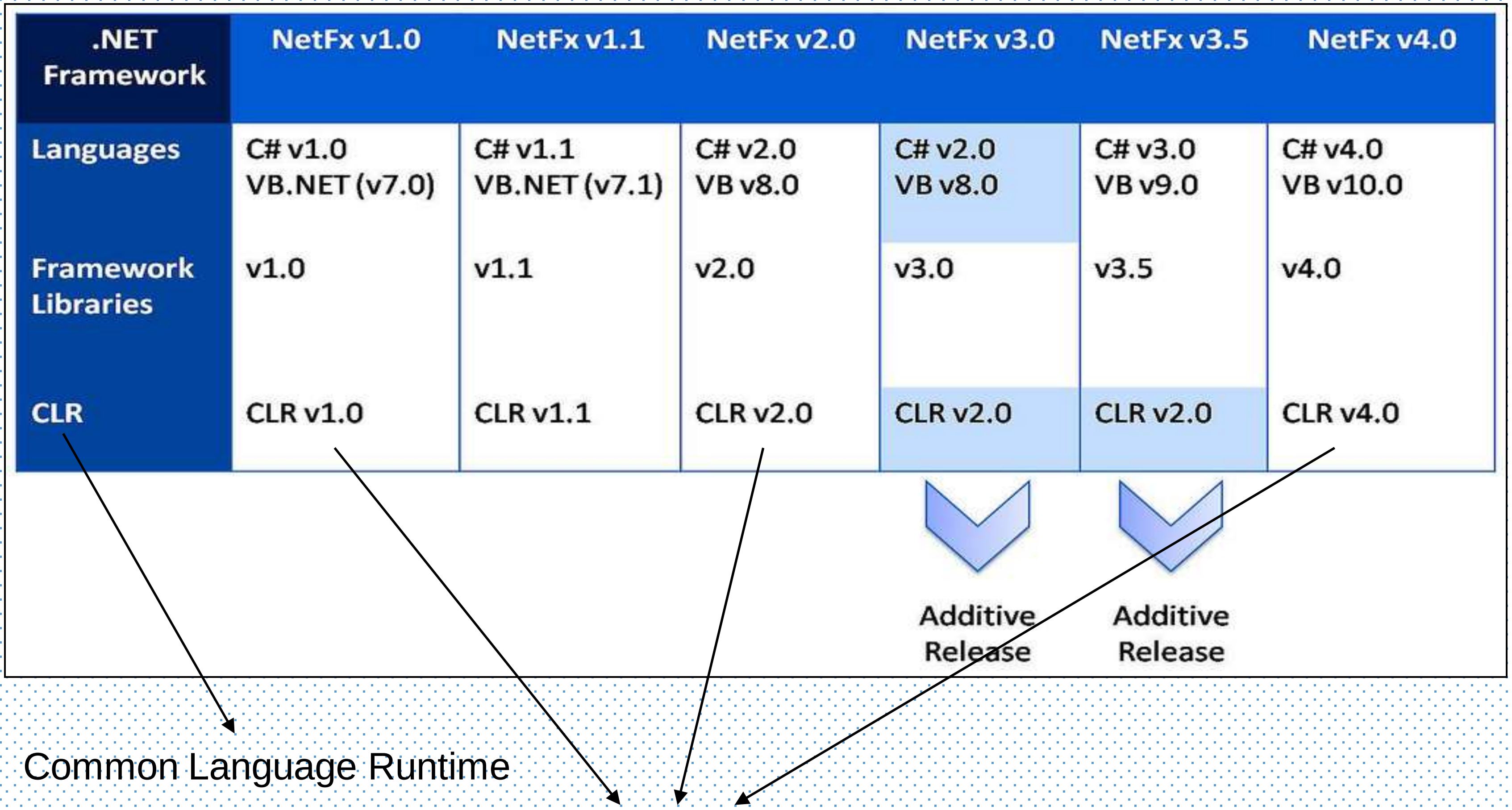
A Version typically comprises of three numbers:

- Major Version Number
- Minor Version Number
- Patch Number


1 . 3 . 2
MAJOR
Backwards-incompatible change
MINOR
Backwards-compatible new feature
PATCH
Backwards-compatible bugfix

Version Example

3.3 Versioning Example



.NET Framework	NetFx v1.0	NetFx v1.1	NetFx v2.0	NetFx v3.0	NetFx v3.5	NetFx v4.0
Languages	C# v1.0 VB.NET (v7.0)	C# v1.1 VB.NET (v7.1)	C# v2.0 VB v8.0	C# v2.0 VB v8.0	C# v3.0 VB v9.0	C# v4.0 VB v10.0
Framework Libraries	v1.0	v1.1	v2.0	v3.0	v3.5	v4.0
CLR	CLR v1.0	CLR v1.1	CLR v2.0	CLR v2.0	CLR v2.0	CLR v4.0
				Additive Release	Additive Release	

Common Language Runtime

Versions

3.3 Popular Version Control Tools

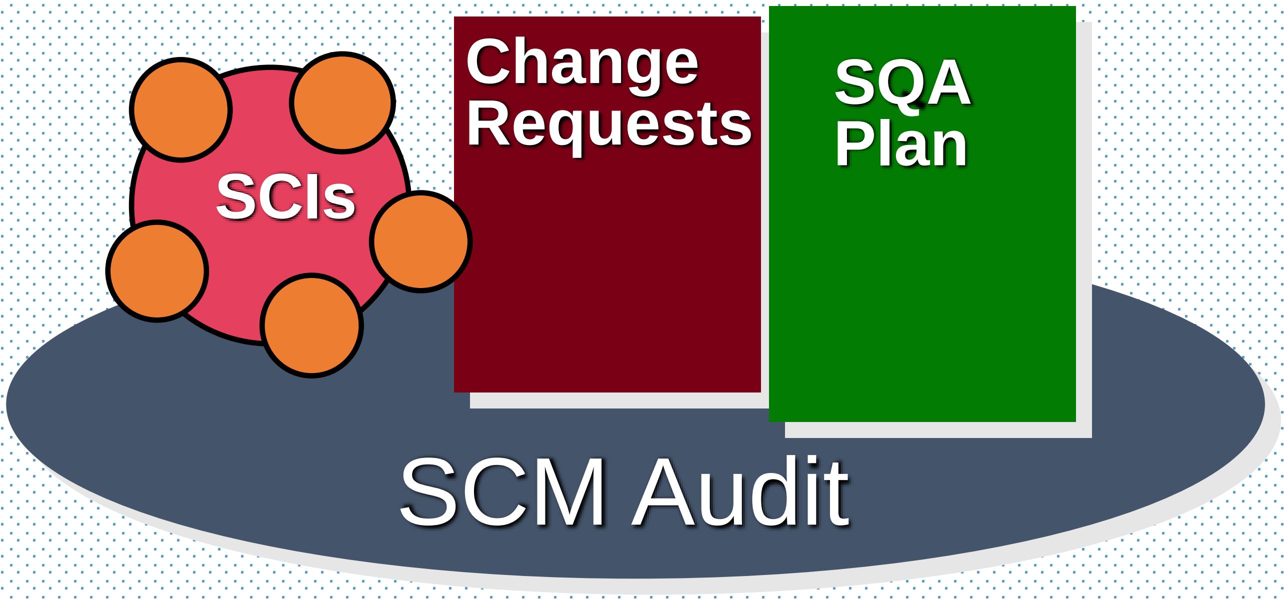
Popular Version Control tools:

1. GIT
2. Team Foundation Server (Microsoft)
3. Rational Clear Case (IBM)
4. Visual SourceSafe (Microsoft)

3.4 SCM Auditing

SCM Auditing answers the following questions:

- What is the SQA Plan followed for project?
- How many SCIs were identified?
- How many change requests were handled?



3.5 Reporting

The **reporting** aspect of the SCM process provides timely information on the **status** of the changes requested and the SCIs to the people who may be affected by the changes.

These could be the people requesting for changes, the developers, the project manager, and the senior management.

Case Study

For the University Project,

1. What SCIs would you identify?
2. Explain the change control process that you would adopt? Explain for a single identified SCI?
3. How many versions of your project is required, before final submission. Justify your answer.