White box Testing

ERFANA SIKDER

White Box Testing: Introduction

- 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(line coverage)
  - Branches
  - Paths
  - Conditions.

What is Dynamic White box or Structural testing

- Dynamic white-box testing, is using information you gain from seeing what the code does and how it works to determine what to test, what not to test, and how to approach the testing.
- Another name commonly used for dynamic white-box testing is *structural testing* because you can see and use the underlying structure of the code to design and run your tests.
- Knowing how the software operates will influence how you test.
- They can help ensure more breadth of testing, in the sense that test cases that achieve 100% **coverage** in any measure will be exercising all parts of the software from the point of view of the items being covered.

What is test coverage?

- Test coverage measures in some specific way the amount of testing performed by a set of tests (derived in some other way, e.g. using specification-based techniques).

$$\text{Coverage} = \frac{\text{Number of coverage items exercised}}{\text{Total number of coverage items}} \times 100\%$$

- where the 'coverage item' is whatever we have been able to count and see whether a test has exercised or used this item. (path,condition,statement)
- 100% coverage does *not* mean 100% tested!
- Coverage techniques measure only one dimension of a multi-dimensional concept.
- Two different test cases may achieve exactly the same coverage but the input data of one may find an error that the input data of the other doesn't.
- One drawback of code coverage measurement is that it measures coverage of what *has* been written, i.e. the code itself; it cannot say anything about the software that has *not* been written.

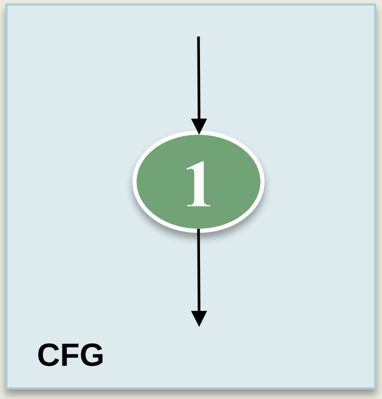
Control Flow Graph: Introduction

- 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.*

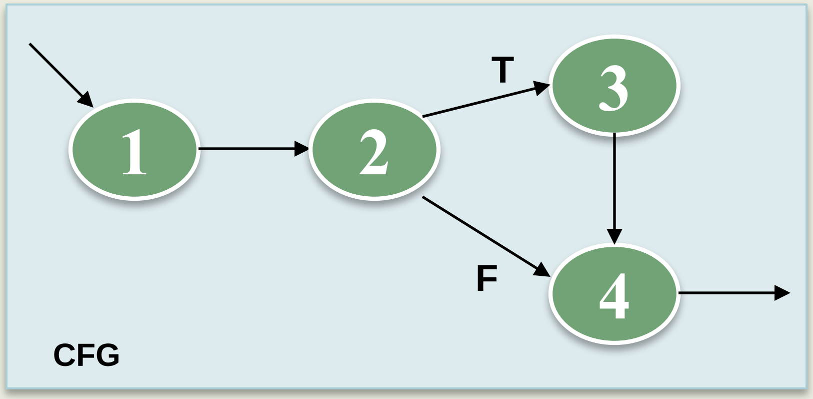
Simple Examples

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

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

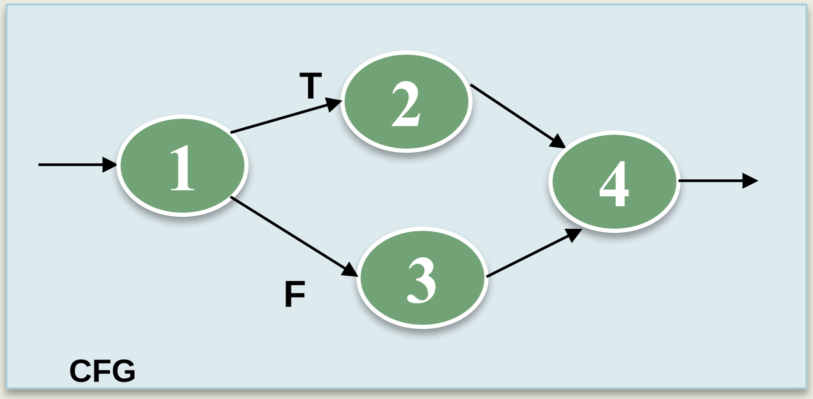


```
Statement1;        } 1
Statement2;

if X < 10 then     } 2
   Statement3;     } 3

Statement4;        } 4
```

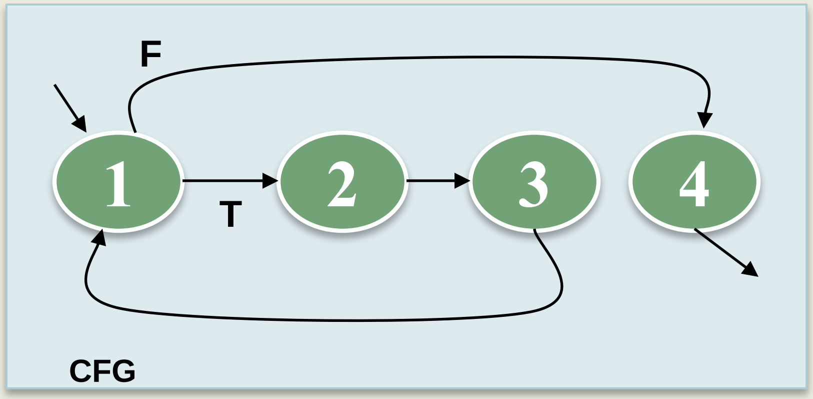


More Examples

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



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



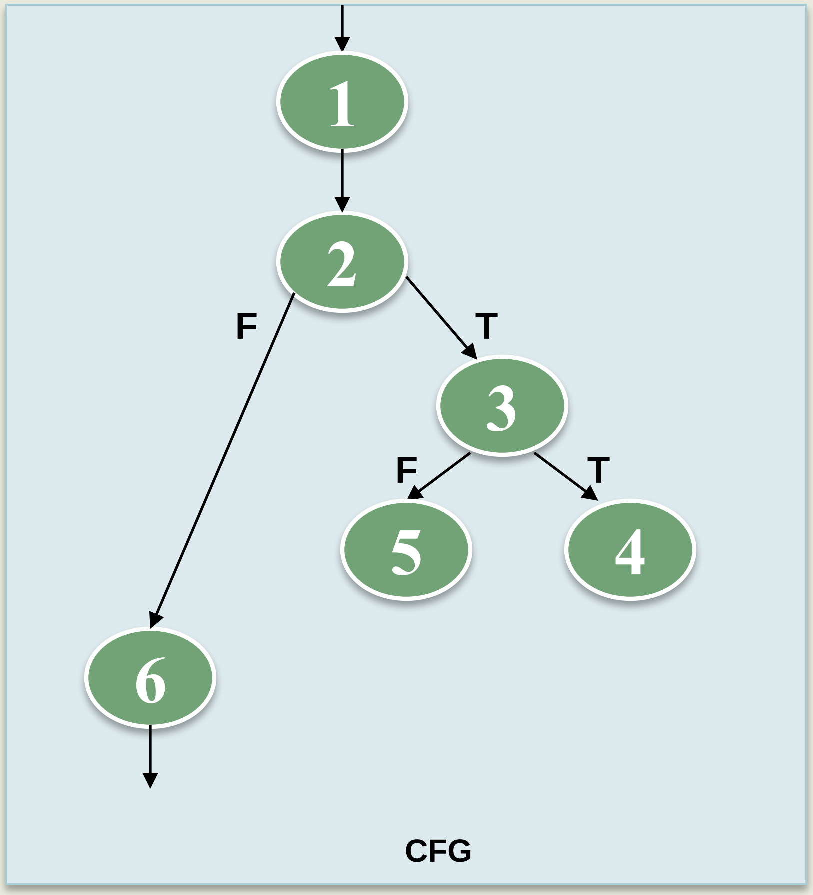
Question: Why is there a node **4** in both CFGs?

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
```

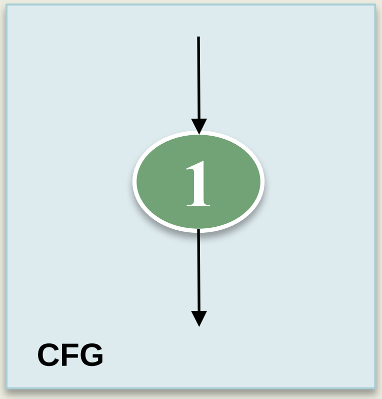


The CFG is **INCOMPLETE**. Try to complete it

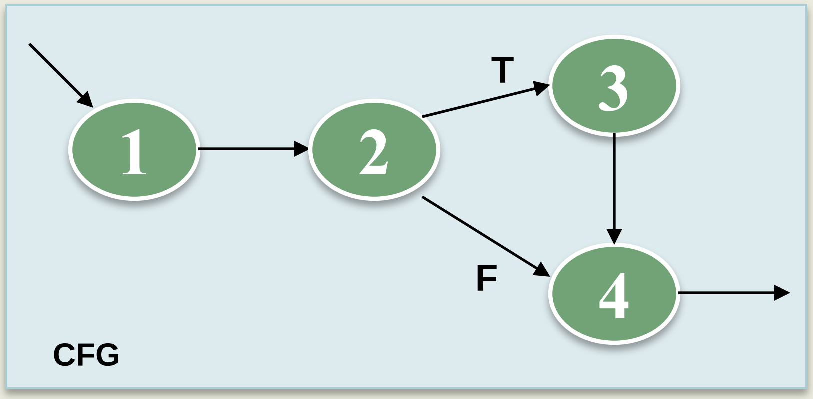
Number of Paths through CFG

- 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?

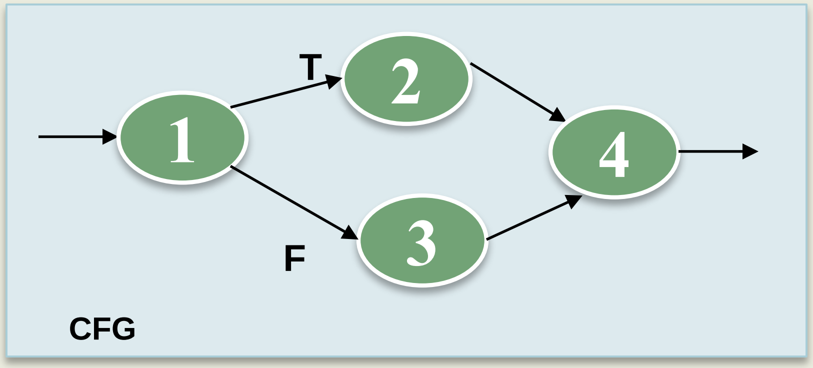
Example



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



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



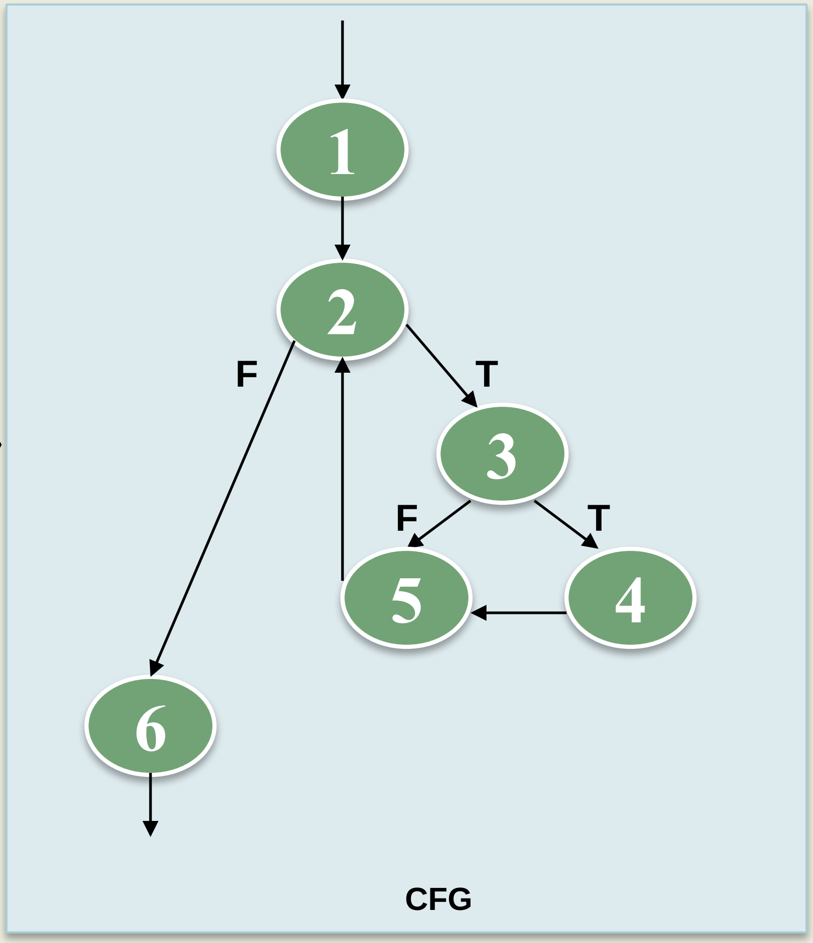
- **Two** paths are needed:
  - **[1 – 2 – 4]**
  - **[1 – 3 – 4]**

White Box Testing: Path Based

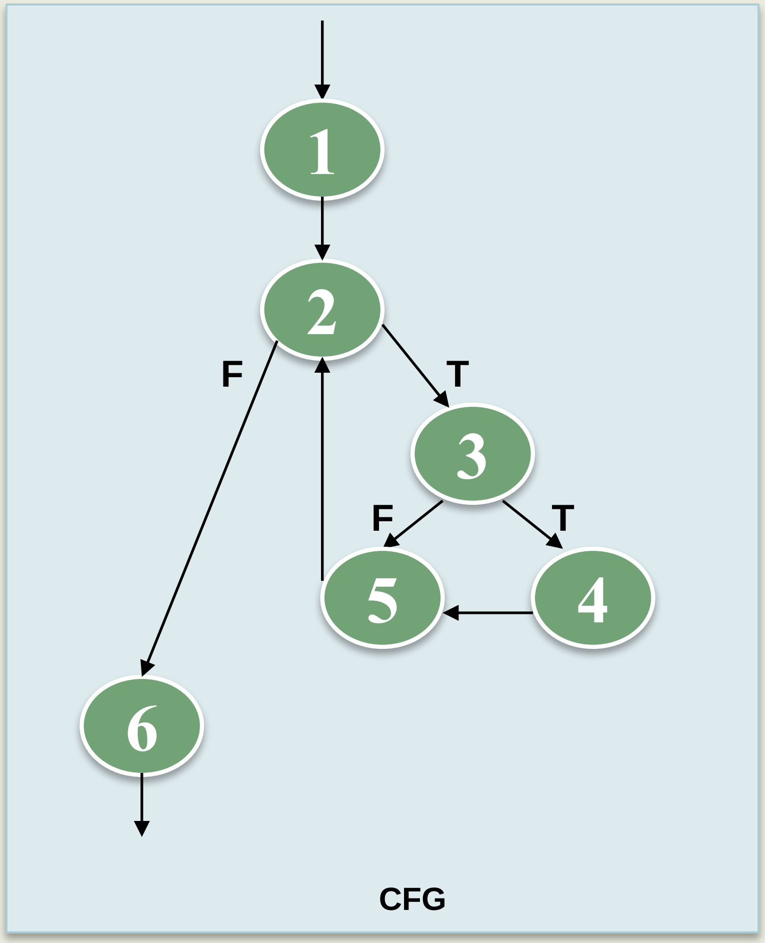
- A generalized technique to find out the number of paths needed (known as *cyclomatic complexity*) to cover all arcs and nodes in CFG.
- Steps:
  1. Draw the CFG for the code fragment.
  2. Compute the *cyclomatic complexity number* ***C***, for the CFG.
  3. Find at most ***C*** paths that cover the nodes and arcs in a CFG, also known as **Basic Paths Set;**
  4. Design test cases to force execution along paths in the **Basic Paths Set.**

Path Based Testing: Step 1

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

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

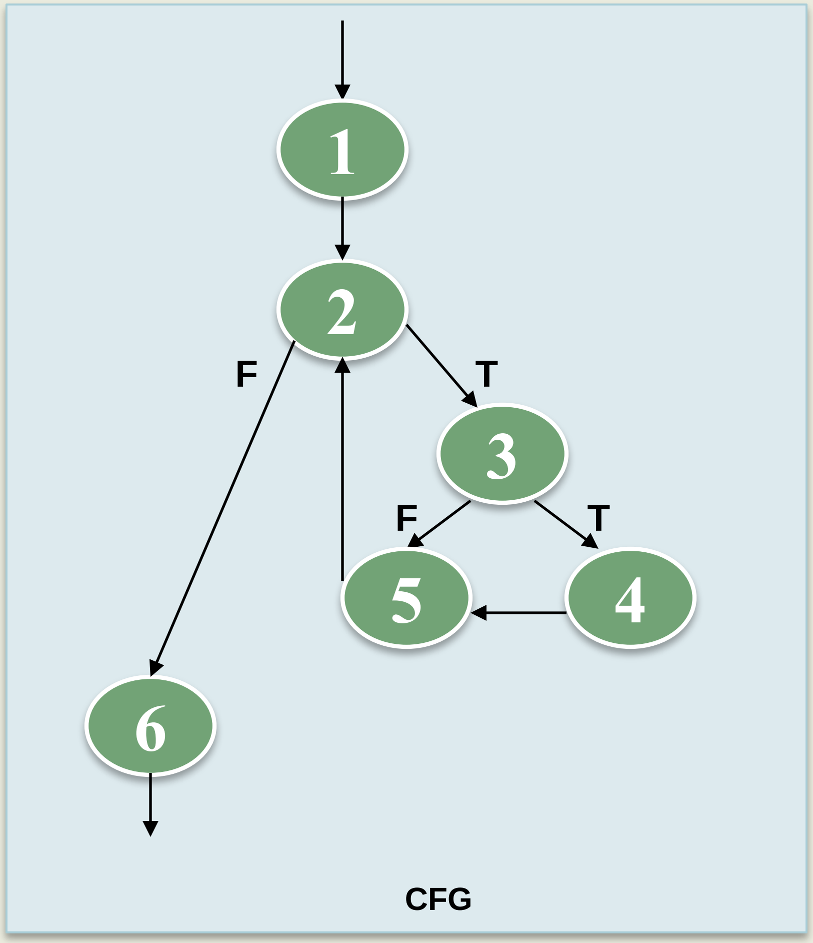


Path Base Testing: Step 2



- Cyclomatic complexity =
  - The number of ‘regions’ in the graph;   OR
  - The number of predicates + 1.

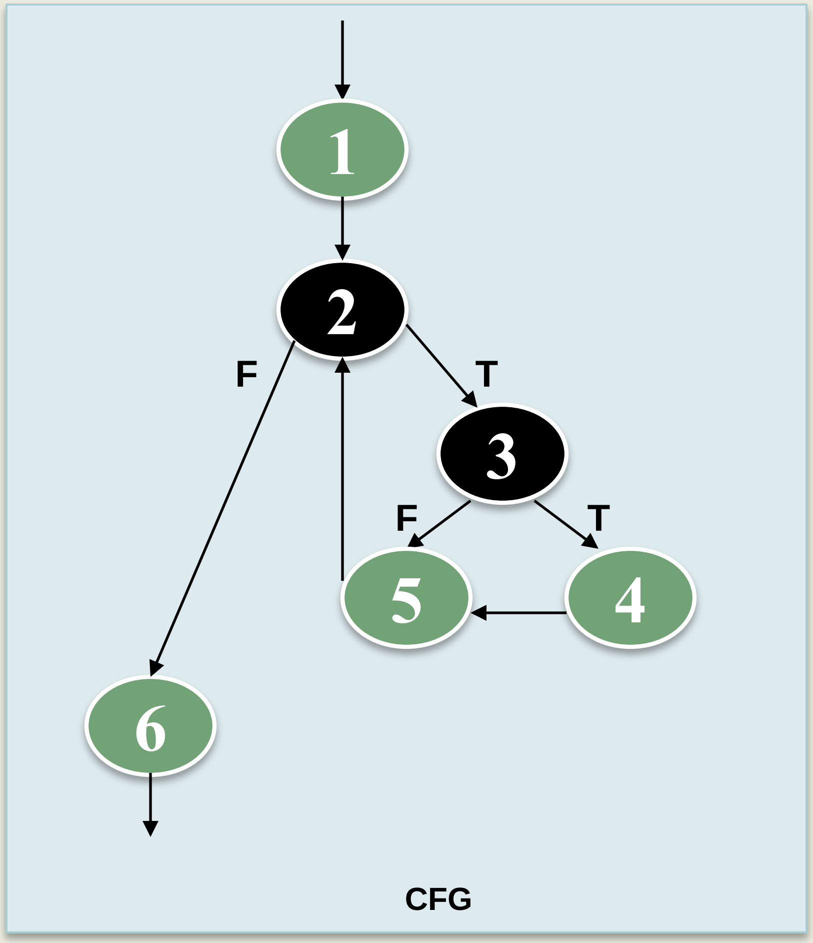
Path Base Testing: Step 2



CFG

- **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.

Path Base Testing: Step 2



CFG

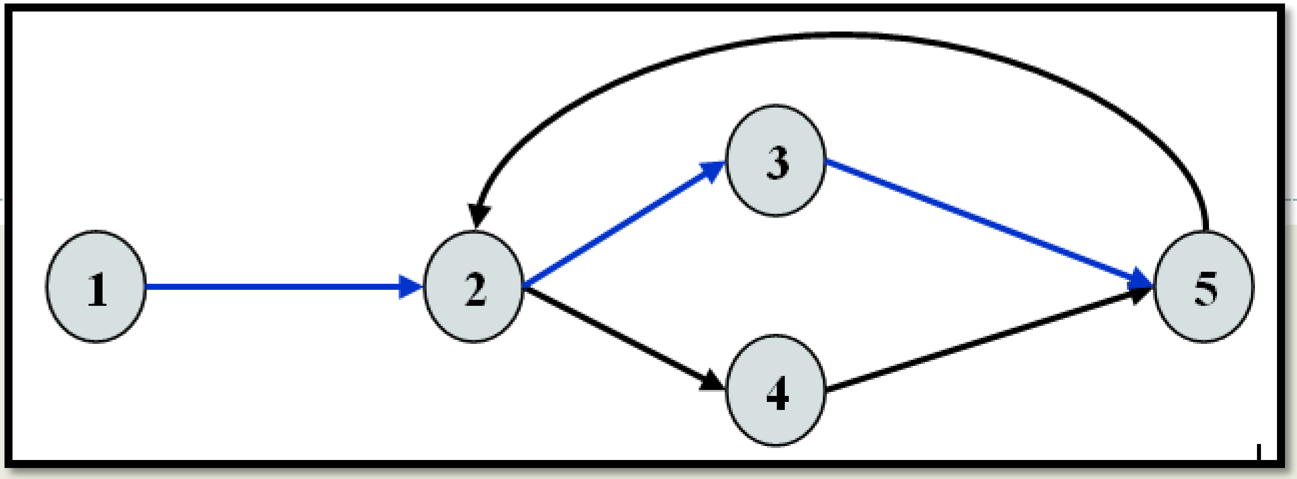
- **Predicates**:
  - Nodes with multiple exit arcs.
  - Corresponds to branch/conditional statement in program.
- In this example:
  - Predicates = 2
    - (Node 2 and 3)
  - Cyclomatic Complexity

    = 2 + 1

    = 3

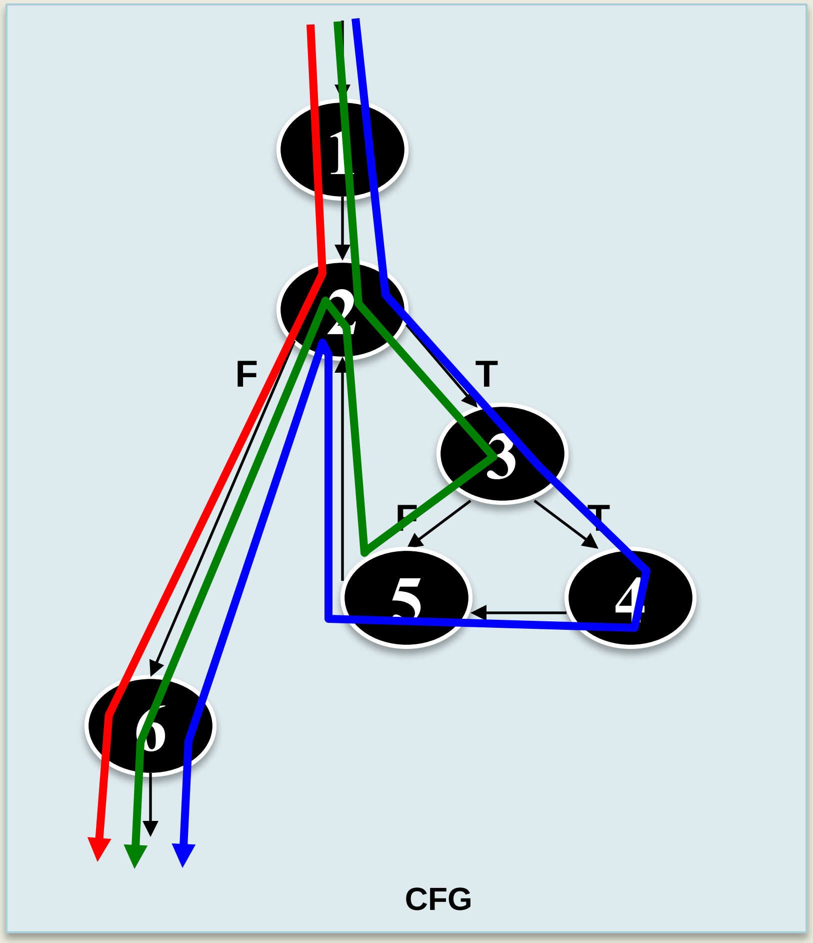
Path Base Testing: Step 3

- 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.



- 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.

Path Base Testing: 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]

Path Base Testing: Step 4

- 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

```
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
```

- These tests will result a complete decision and statement coverage of the code.

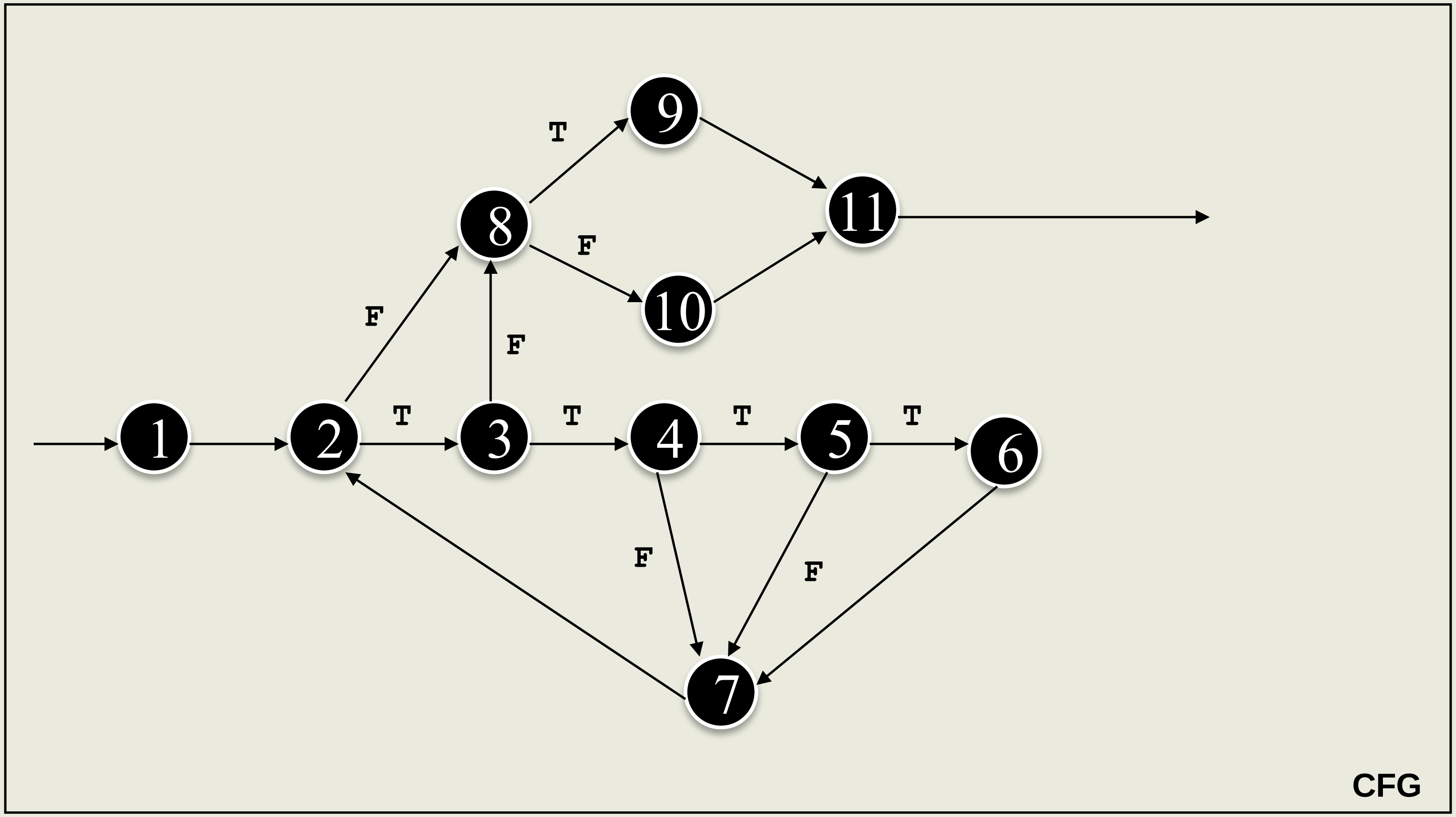
Another Example

```
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;
}
```

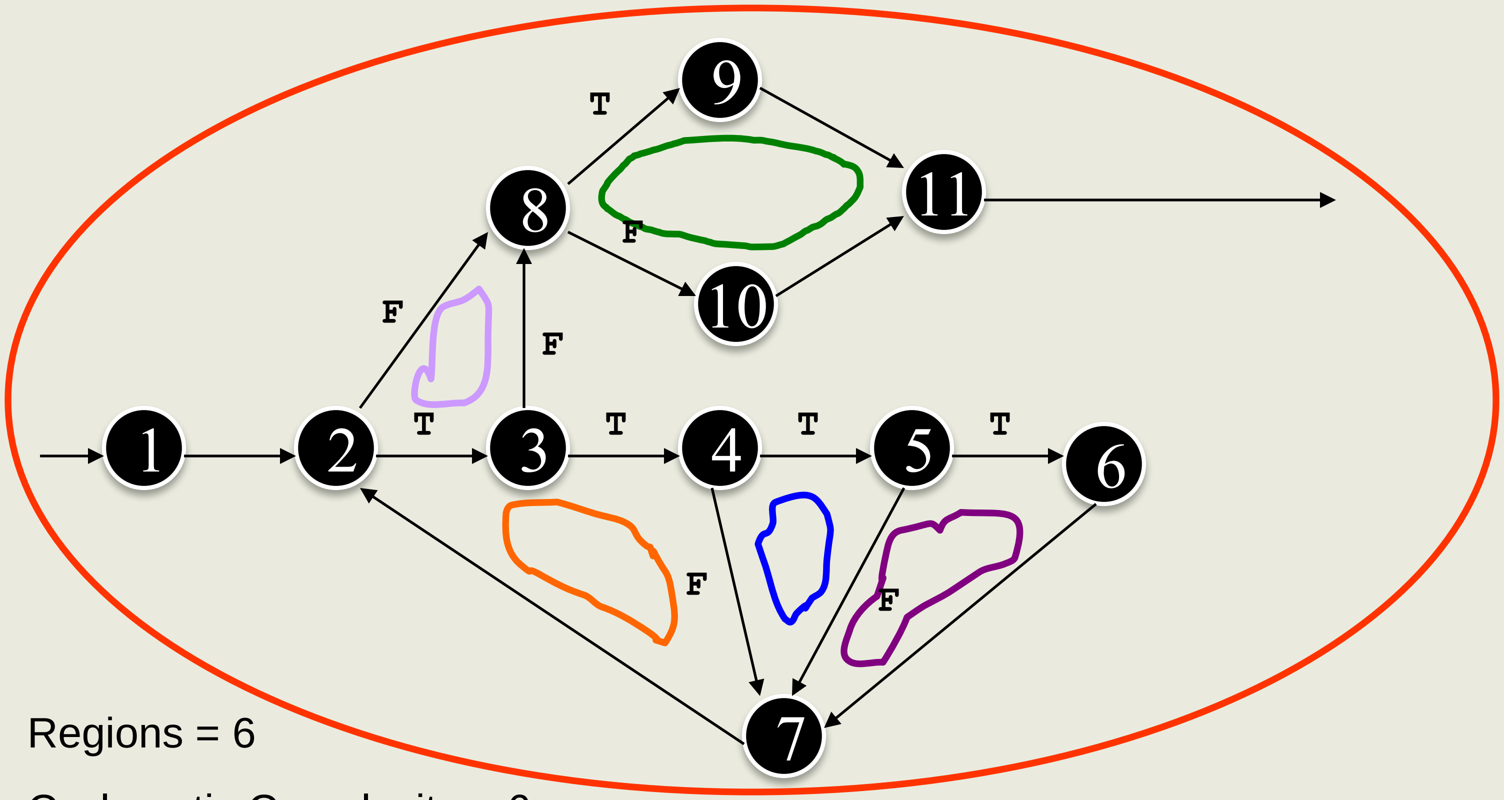
Step 1: Draw CFG

```
int average (int[ ] value, int min, int max, int N) {
    int i, totalValid, sum, mean;   } 1
    i = totalValid = sum = 0;
              2              3
    while ( i < N && value[i] != -999 ) {
                   4                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

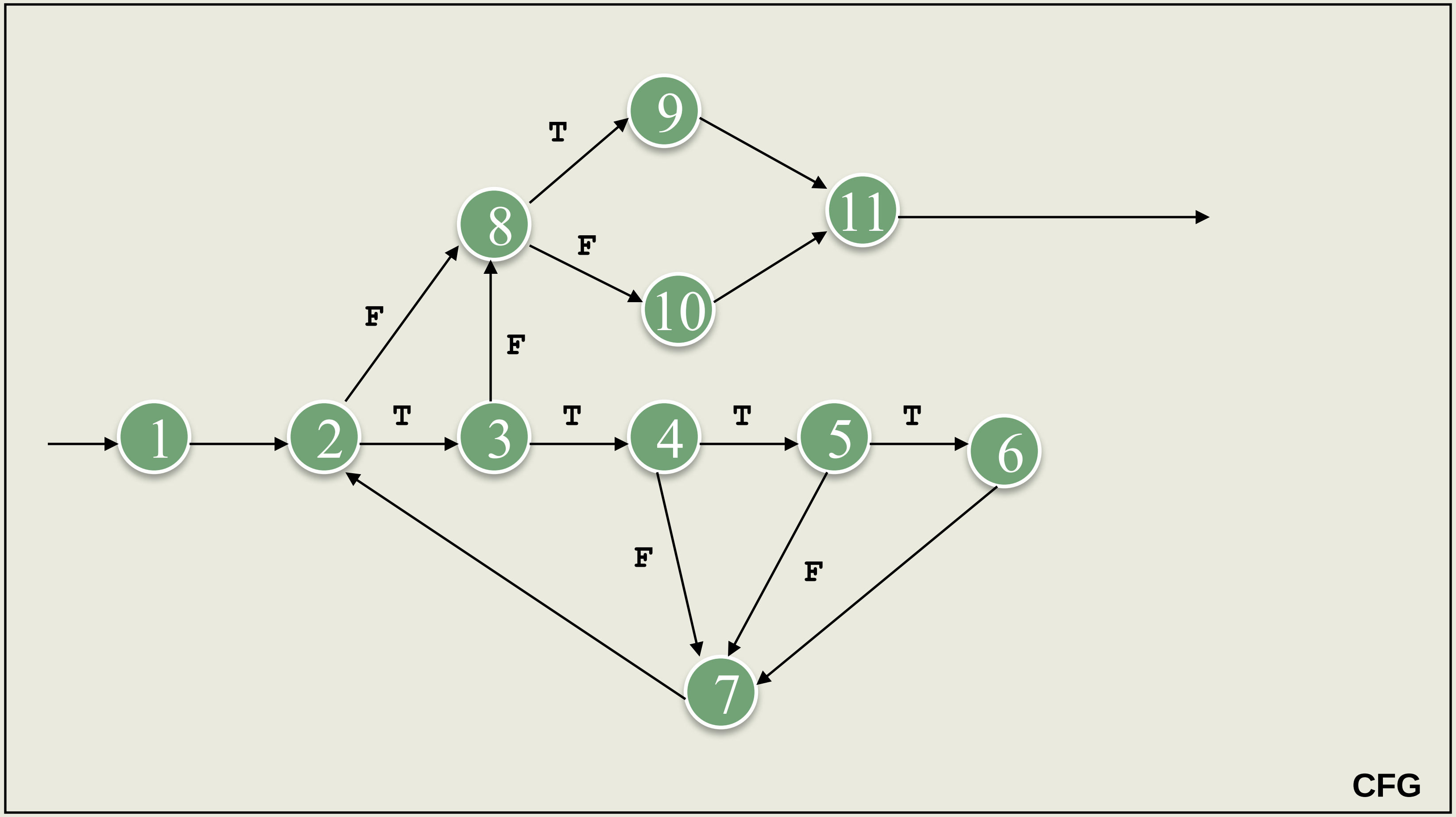


Regions = 6

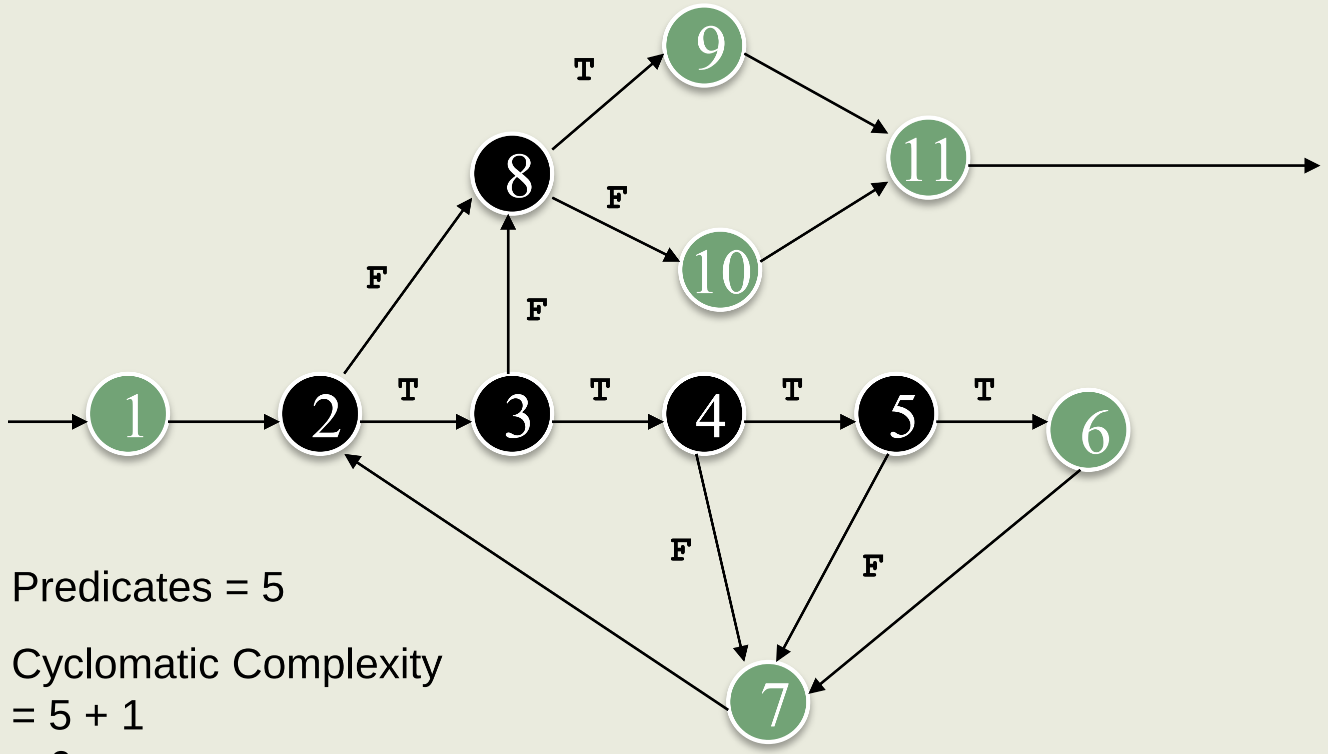
Cyclomatic Complexity = 6

CFG

Step 2: Find Cyclomatic Complexity



Step 2: Find Cyclomatic Complexity



Predicates = 5

Cyclomatic Complexity
= 5 + 1
= 6

CFG

Step 3: Find Basic 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).
- 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

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

OR

- 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`

Summary: Path Base White Box 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.

Example

Imperial Taxi Services (ITS) serves one-time passengers and regular clients (identified by a taxi card). The ITS taxi fares for one-time passengers are calculated as follows:

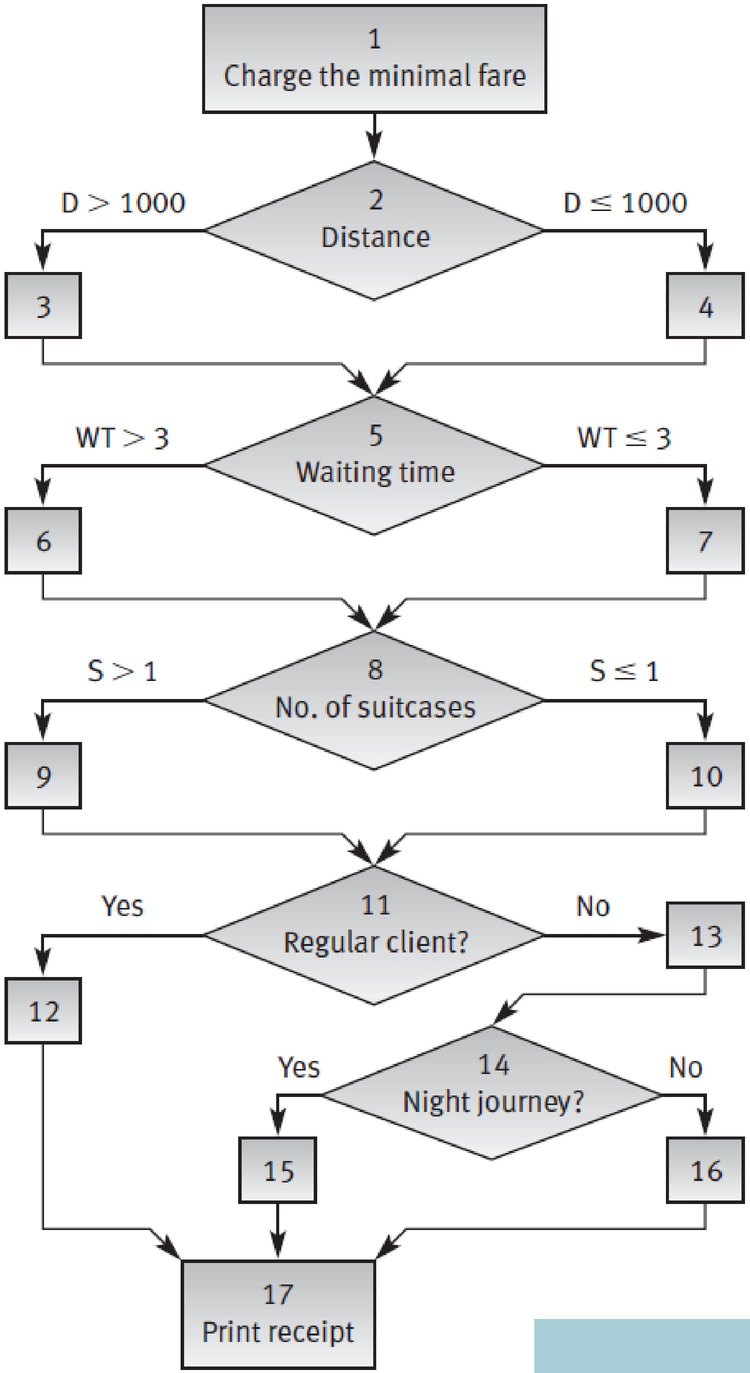
(1) Minimal fare: $2. This fare covers the distance traveled up to 1000 yards and waiting time (stopping for traffic lights or traffic jams, etc.) of up to 3 minutes.

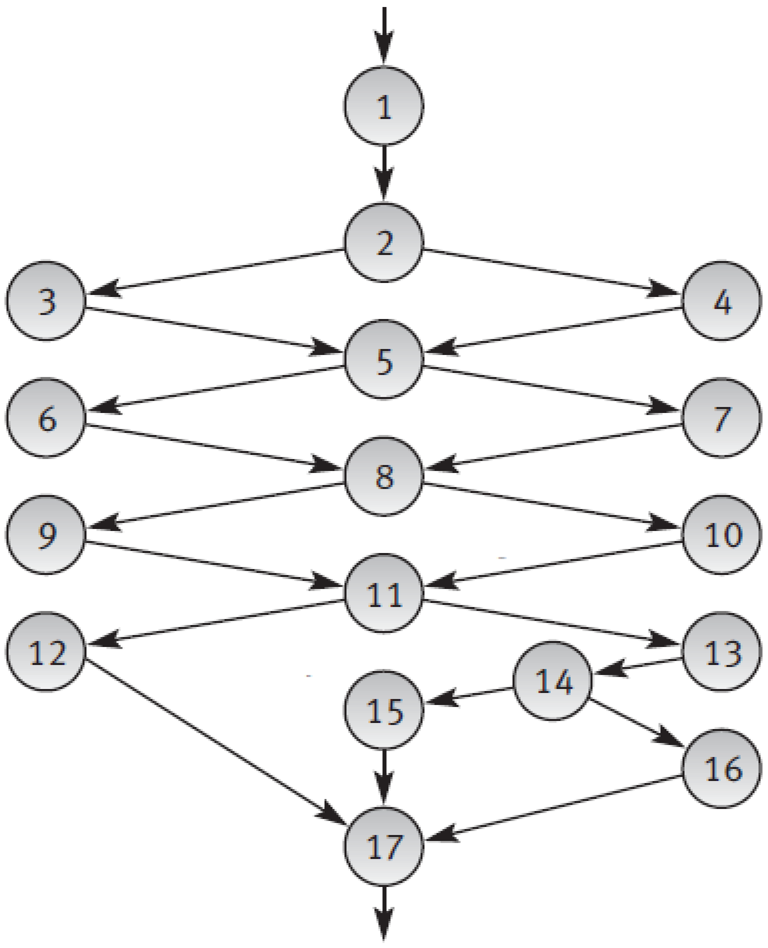
(2) For every additional 250 yards or part of it: 25 cents.

(3) For every additional 2 minutes of stopping or waiting or part thereof: 20 cents.

(4) One suitcase: no charge; each additional suitcase: $1.

(5) Night supplement: 25%, effective for journeys between 21.00 and 06.00.
Regular clients are entitled to a 10% discount and are not charged the night supplement.


1
Charge the minimal fare
2
Distance
D > 1000
D ≤ 1000
3
4
5
Waiting time
WT > 3
WT ≤ 3
6
7
8
No. of suitcases
S > 1
S ≤ 1
9
10
11
Regular client?
Yes
No
12
13
14
Night journey?
Yes
No
15
16
17
Print receipt


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17

Example

Find out the Cyclometic complexity and
Basic path set

Advantages and disadvantages of white box testing

- **Advantages:**

– Direct determination of software correctness as expressed in the processing paths, including algorithms.

– Allows performance of line coverage follow up.

– Ascertains quality of coding work and its adherence to coding standards.

- **Disadvantages :**

– The vast resources utilized, much above those required for black box testing of the same software package.

– The inability to test software performance in terms of availability (response time), reliability, load durability, etc.

EXPERIENCE-BASED TECHNIQUES

EXPLORATORY TESTING

ERROR GUESSING

Exploratory testing

- Cem Kaner
- a style of software testing that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the quality of his/her work by treating test-related learning, test design, test execution, and test result interpretation as mutually supportive activities that run in parallel throughout the project.

http://en.wikipedia.org/wiki/Exploratory_testing
http://www.satisfice.com/articles/what_is_et.shtml
http://www.kaner.com/pdfs/ETatQAI.pdf

Error Guessing

- A complementary method with other formal techniques
- Based on tester's own experience
- No specific rule for error guessing
- list possible defects or failures and to design tests that attempt to produce them.
- The tester is encouraged to think of situations in which the software may not be able to cope.
  - Typical conditions to try include division by zero, blank (or no) input, empty files and the wrong kind of data (e.g. alphabetic characters where numeric are required)
  - While the page is loading trying to submit form again
  - If computation is involved play around the data type and machine culture.
  - If time/ date is involved then think about machine culture
  - If anyone ever says of a system or the environment in which it is to operate 'That could never happen', it might be a good idea to test that condition
  - Behave like a dumb user
  - Think about concurrency
  - Think about race condition

Testing technique choosing procedure

Which process to choose?

- Depends on a number of factors
  - Type of system
  - Customer or contractual requirement
  - Level & type of risk
  - Test objectives
  - Knowledge of the tester
  - Documentation available
  - Time and budget
  - Development life cycle
  - Previous experience or types of defects found
- Best testing technique is no single testing technique.

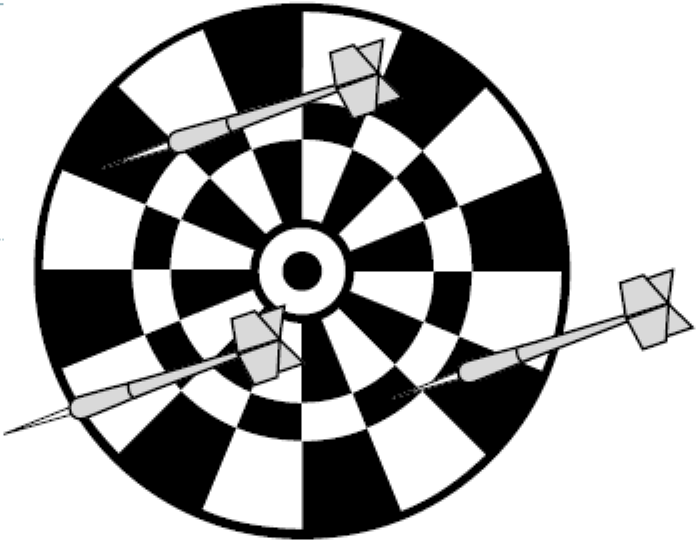
Internal factors the influence testing technique choosing

- *Models used*
  - Model : developed and used during the specification, design and implementation of the system
  - E.g. :If the specification contains a state transition diagram, state transition testing would be a good technique to use.
- *Tester knowledge /experience*
  - How much testers know about the system and about testing techniques
  - This knowledge will in itself be influenced by their experience of testing and of the system under test.
- *Likely defects*
  - Knowledge of the likely defects will be very helpful
  - This knowledge could be gained through experience of testing a previous version of the system and previous levels of testing on the current version.
- *Test objective*
  - If the test objective is simply to gain confidence that the soft ware will cope with typical operational tasks then use cases would be a sensible approach.
  - If the objective is for very thorough testing then more rigorous and detailed should be chosen.
- *• Documentation*
  - Whether or not documentation (e.g. a requirements specification) exists and whether or not it is up to date will affect the choice of testing techniques.
  - The content and style of the documentation will also influence the choice of techniques (for example, if decision tables or state graphs have been used then the associated test techniques should be used).
- *• Life cycle model*
  - A sequential life cycle model will lend itself to the use of more formal techniques whereas an iterative life cycle model may be better suited to using an exploratory testing approach.

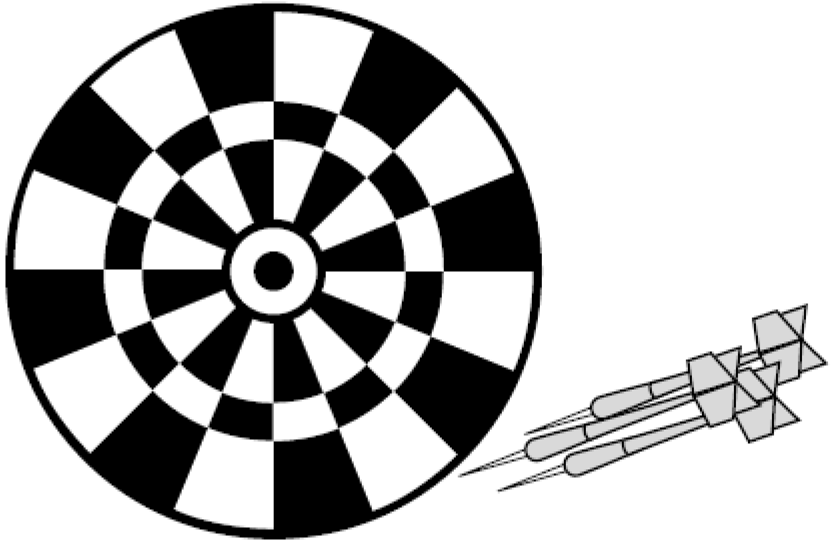
External factors the influence testing technique choosing

- *Risk*
  - The greater the risk (e.g. safety-critical systems), the greater the need for more thorough and more formal testing.
- *Customer/contractual requirements*
  - Sometimes contracts specify particular testing techniques to use (most commonly statement or branch coverage).
- *Type of system*
  - The type of system (e.g. embedded, graphical, financial, etc.) will influence the choice of techniques.
  - For example, a financial application involving many calculations would benefit from boundary value analysis.
- *Regulatory requirements*
  - Some industries have regulatory standards or guidelines that govern the testing techniques used.
  - For example, the aircraft industry requires the use of equivalence partitioning, boundary value analysis and state transition testing for high integrity systems together with statement depending on the level of software integrity required.
- *Time and budget*
  - *U*ltimately how much time there is available will always affect the choice of testing techniques.
  - When more time is available we can afford to select more techniques and when time is severely limited we will be limited to those that we know have a good chance of helping us find just the most important defects.

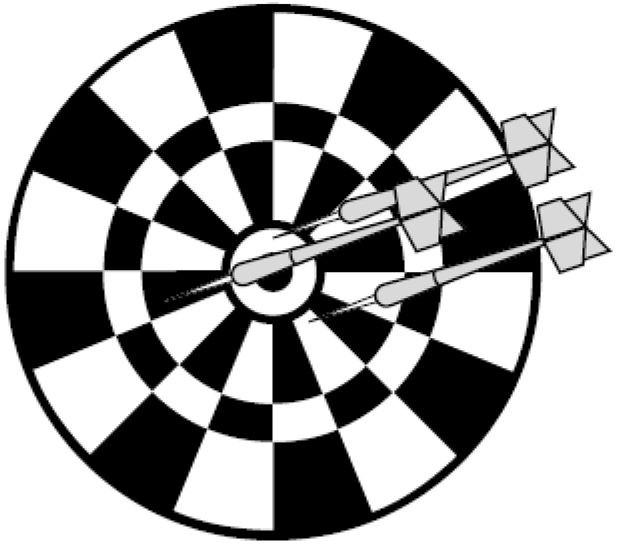
Verification & Validation example



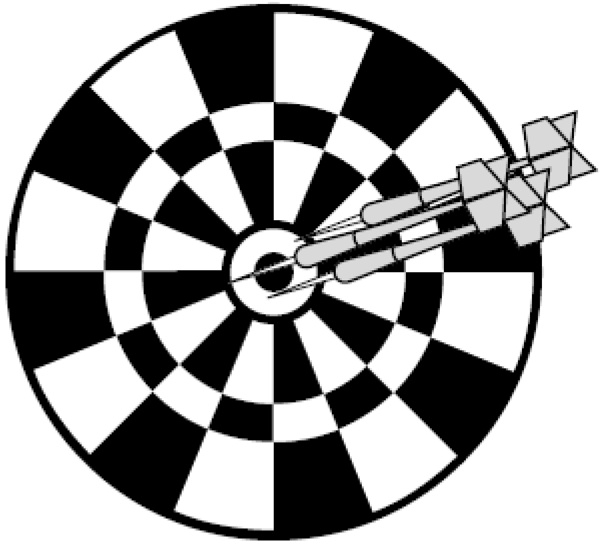
Neither Accurate nor Precise



Precise, but not Accurate



Accurate, but not Precise



Accurate and Precise

In April 1990, the Hubble space telescope was launched into orbit around the Earth. As a reflective telescope, Hubble uses a large mirror as its primary means to magnify the objects it's aiming at. The construction of the mirror was a huge undertaking requiring extreme precision and accuracy. Testing of the mirror was difficult since the telescope was designed for use in space and couldn’t be positioned or even viewed through while it was still on Earth. For this reason, the only means to test it was to carefully measure all its attributes and compare the measurements with what was specified. This testing was performed and Hubble was declared fit for launch.

Unfortunately, soon after it was put into operation, the images it returned were found to be out of focus. An investigation discovered that the mirror was improperly manufactured. The mirror was ground according to the specification, but the specification was wrong. The mirror was extremely *precise*, but it wasn’t *accurate*. Testing had confirmed that the mirror met the spec—*verification*—but it didn’t confirm that it met the original requirement—*validation*.

In 1993, a space shuttle mission repaired the Hubble telescope by installing a “corrective lens” to refocus the image generated by the improperly manufactured mirror.