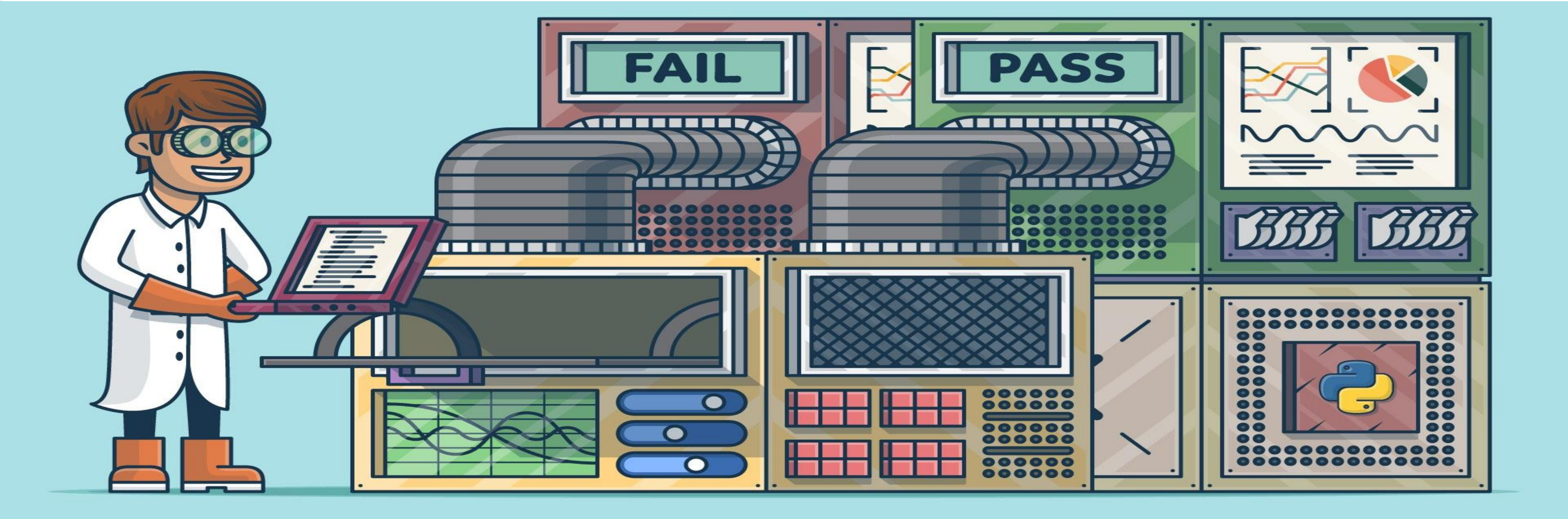
Advanced Software Engineering

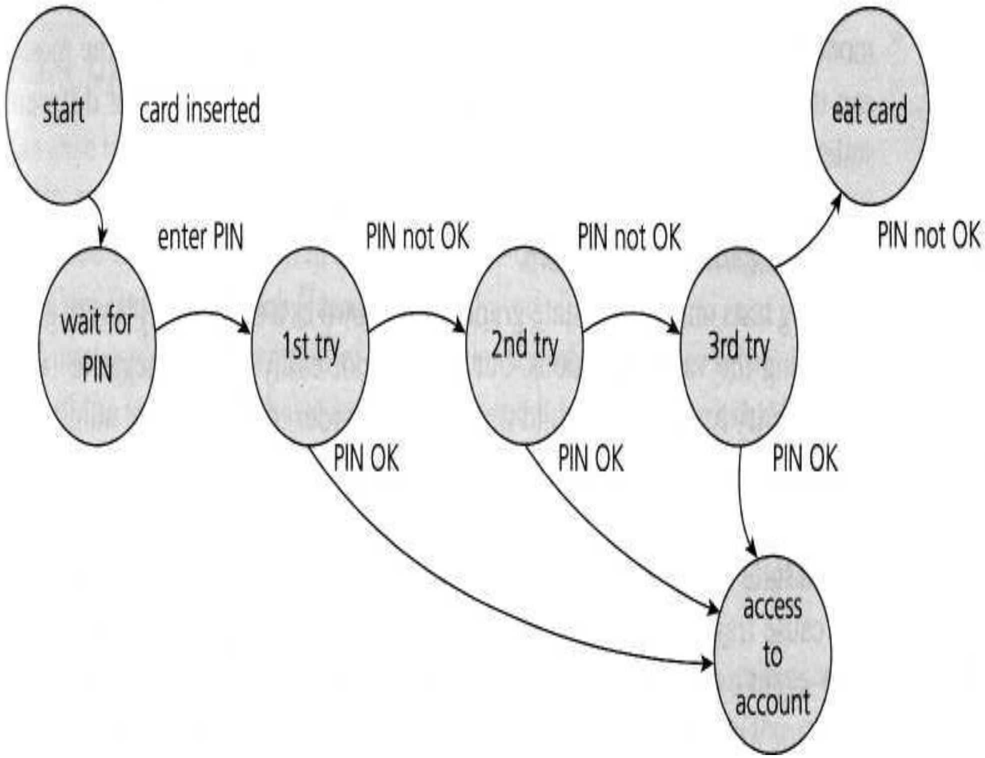
CSE608

Software Testing



Dr. Islam El-Maddah

Testing State Diagrams

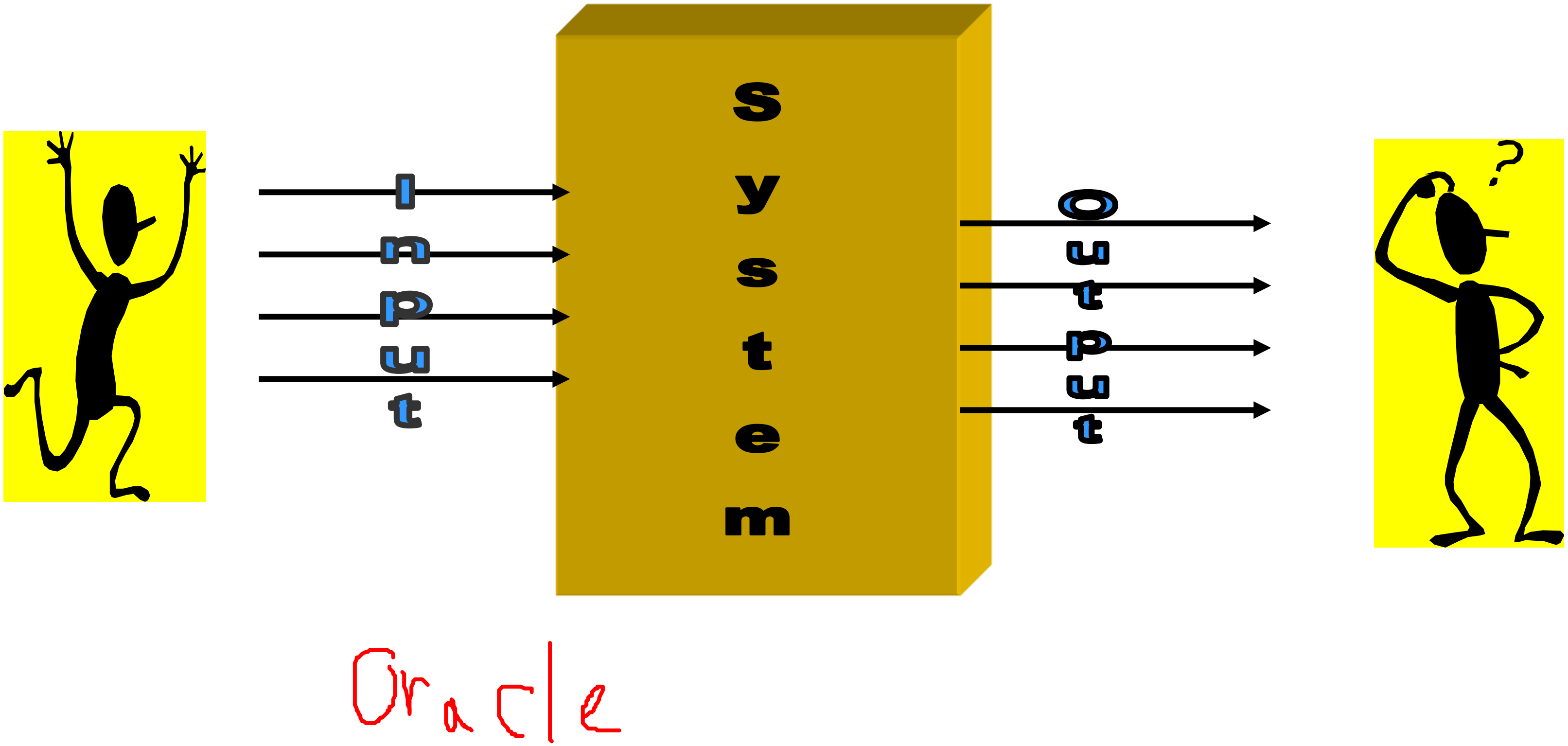


	Insert card	Valid PIN	Invalid PIN
S1) Start state	S2	-	-
S2) Wait for PIN	-	S6	S3
S3) 1st try invalid	-	S6	S4
S4) 2nd try invalid	-	S6	S5
S5) 3rd try invalid	-	-	S7
S6) Access account	-	?	?
S7) Eat card	S1 (for new card)	-	-

How do you test a system?

- Input test data to the system.
- Observe the output:
  - Check if the system behaved as expected.

How do you test a system?



How do you test a system?

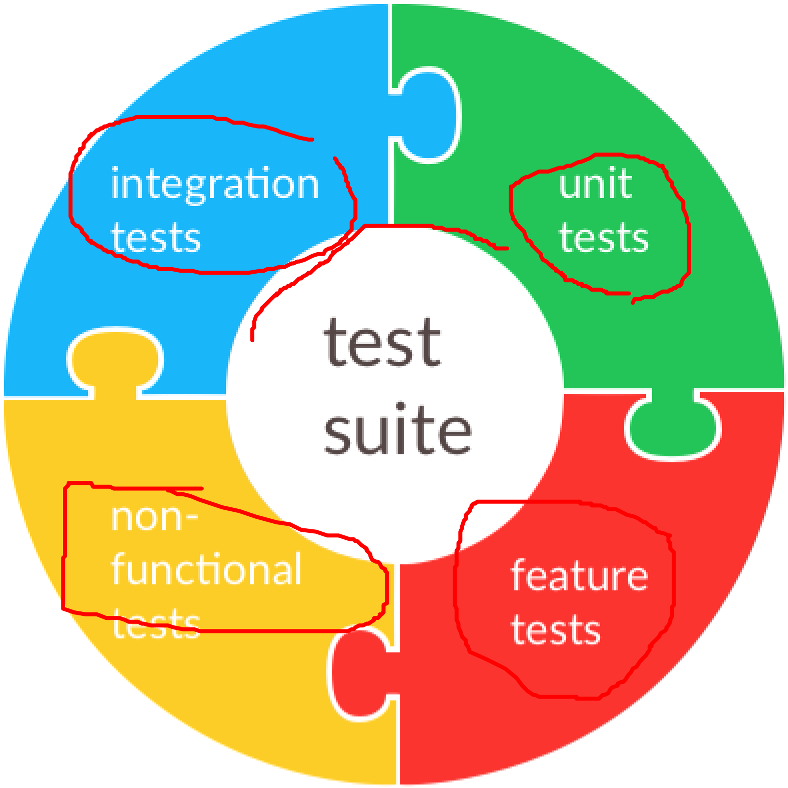
- If the program does not behave as expected:
  - note the conditions under which it failed.
  - later debug and correct.

Errors and Failures

- A failure is a manifestation of an error (aka defect or bug).
  - mere presence of an error may not lead to a failure.

Test cases and Test suite

- Test a software using a set of carefully designed test cases:
  - the set of all test cases is called the test suite



Test cases and Test suite

- A test case is a triplet [I,S,O]:
  - I is the data to be input to the system,
  - S is the state of the system at which the data is input,
  - O is the expected output from the system.

Verification vs Validation

- Verification is the process of determining:
  - whether output of one phase of development conforms to its previous phase.
- Validation is the process of determining
  - whether a fully developed system conforms to its SRS document.

Verification & Validation

Are we building the product right?	Are we building the right product?

Verification	Validation
• Verify the intermediary products like requirement documents, design documents, ER diagrams, test plan and traceability matrix • Developer point of view • Verified without executing the software code • Techniques used: Informal Review, Inspection, Walkthrough, Technical and Peer review	• Validate the final end product like developed software or service or system • Customer point of view • Validated by executing the software code • Techniques used: Functional testing, System testing, Smoke testing, Regression testing and Many more

Design of Test Cases

- Exhaustive testing of any non-trivial system is impractical:
  - input data domain is extremely large.
- Design an optimal test suite:
  - of reasonable size
  - to uncover as many errors as possible.

Design of Test Cases

- If test cases are selected randomly:
  - many test cases do not contribute to the significance of the test suite,
  - do not detect errors not already detected by other test cases in the suite.
- The number of test cases in a randomly selected test suite:
  - not an indication of the effectiveness of the testing.

Design of Test Cases

- Testing a system using a large number of randomly selected test cases:
  - does not mean that many errors in the system will be uncovered.
- Consider an example:
  - finding the maximum of two integers x and y.

Design of Test Cases

- ```
  If (x>y) max = x;
              else max = x;
  ```

- The code has a simple error:
- test suite {(x=3,y=2);(x=2,y=3)} can detect the error,
- a larger test suite {(x=3,y=2);(x=4,y=3); (x=5,y=1)} does not detect the error.
  ```

Design of Test Cases

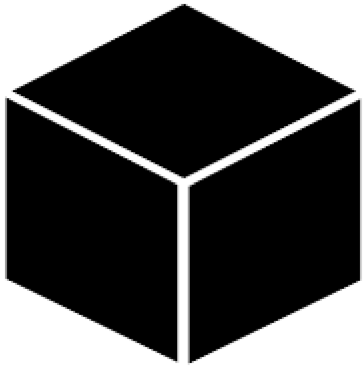
- Systematic approaches are required to design an optimal test suite:
  - each test case in the suite should detect different errors.

Design of Test Cases

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

Black-box Testing

- Test cases are designed using only functional specification of the software:
  - without any knowledge of the internal structure of the software.
- For this reason, black-box testing is also known as functional testing.



White-box Testing

- Designing white-box test cases:
  - requires knowledge about the internal structure of software.
  - white-box testing is also called <u>structural testing</u>.

Black-box Testing

- Two main approaches to design black box test cases:
  - Equivalence class partitioning
  - Boundary value analysis

White-Box Testing

- There exist several popular white-box testing methodologies:
  - Statement coverage
  - Branch coverage
  - Path coverage
  - Condition coverage
  - Mutation testing
  - Data flow-based testing

Statement Coverage

- Statement coverage methodology:
  - design test cases so that
    - every statement in a program must be executed at least once.

Statement Coverage

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

Statement coverage criterion

- Based on the observation:
  - an error in a program can not be discovered:
    - unless the part of the program containing the error is executed.

Statement coverage criterion

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

Example

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

Euclid's GCD Algorithm

Euclid's GCD computation algorithm

- By choosing the test set {(x=3,y=3),(x=4,y=3), (x=3,y=4)}
- all statements are executed at least once.

Branch Coverage

- Test cases are designed such that:
  - different branch conditions
    - given true and false values in turn.

Branch Coverage

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

Stronger testing

- Test cases are a superset of a weaker testing:
  - discovers at least as many errors as a weaker testing
  - contains at least as many significant test cases as a weaker test.

Example

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

Example

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

Condition Coverage

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

Example

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

Branch testing

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

Branch testing

- Condition testing
  - stronger testing than branch testing:
- Branch testing
  - stronger than statement coverage testing.

Condition coverage

- Consider a Boolean expression having n components:
  - for condition coverage we require 2^n test cases.

Condition coverage

- Condition coverage-based testing technique:
  - practical only if n (the number of component conditions) is small.

Path Coverage

- Design test cases such that:
  - all linearly independent paths in the program are executed at least once.

Linearly independent paths

- Defined in terms of
  - control flow graph (CFG) of a program.

Path coverage-based testing

- To understand the path coverage-based testing:
  - we need to learn how to draw control flow graph of a program.

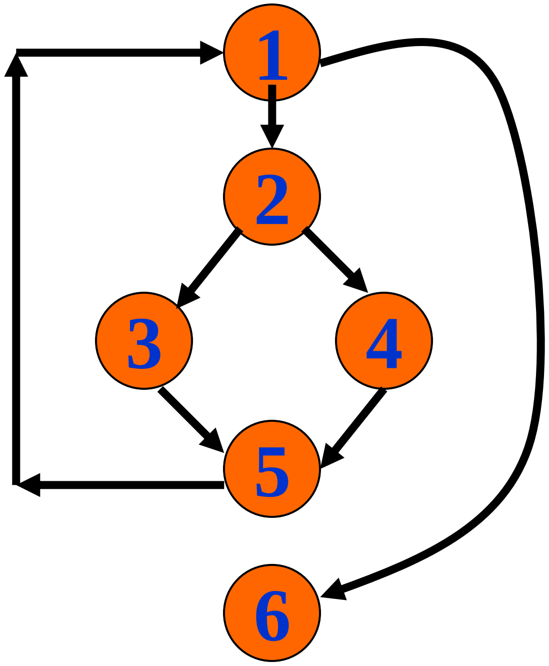
Control flow graph (CFG)

- A control flow graph (CFG) describes:
  - the sequence in which different instructions of a program get executed.
  - the way control flows through the program.

Example

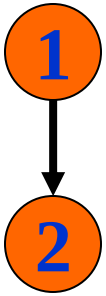
```
int f1(int x,int y){
while (x != y){
    if (x>y) then
        x=x-y;
    else y=y-x;
}
return x;          }
```

Example Control Flow Graph



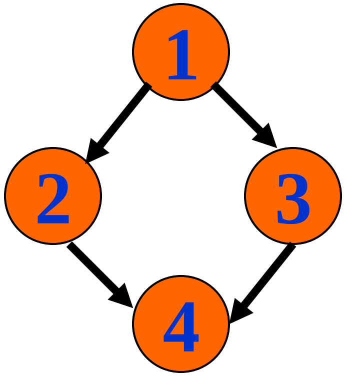
How to draw Control flow graph?

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



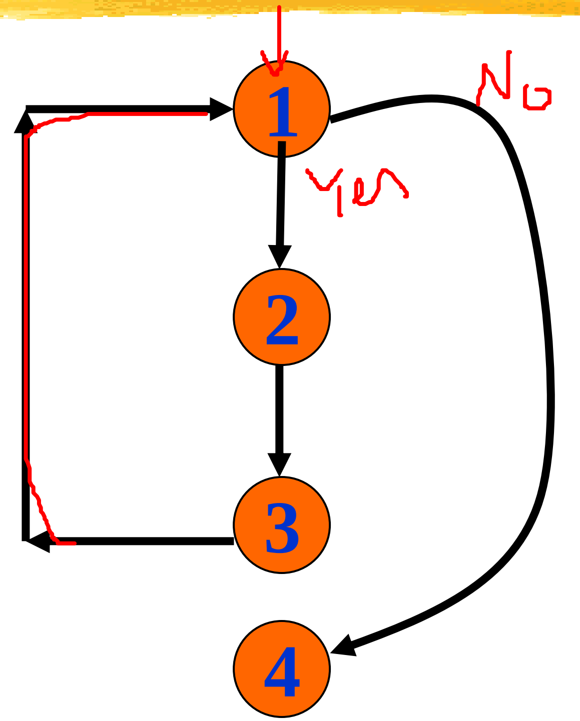
How to draw Control flow graph?

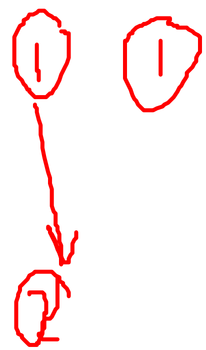
- Selection:
  - 1 if(a>b) then
  - 2 c=3;
  - 3 else c=5;
  - 4 c=c*c;



How to draw Control flow graph?

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





C.C. = ~~nodes~~ edges - nodes + 2

Path

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

Independent path

- Any path through the program:
  - introducing at least one new node:
    - that is not included in any other independent paths.

Independent path

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

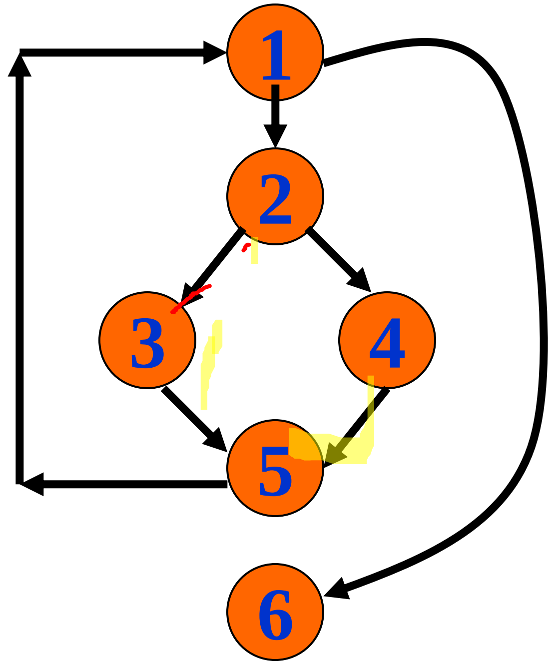
McCabe's cyclomatic metric

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

McCabe's cyclomatic metric

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

Example Control Flow Graph



Example

- Cyclomatic complexity = 7-6+2 = 3.

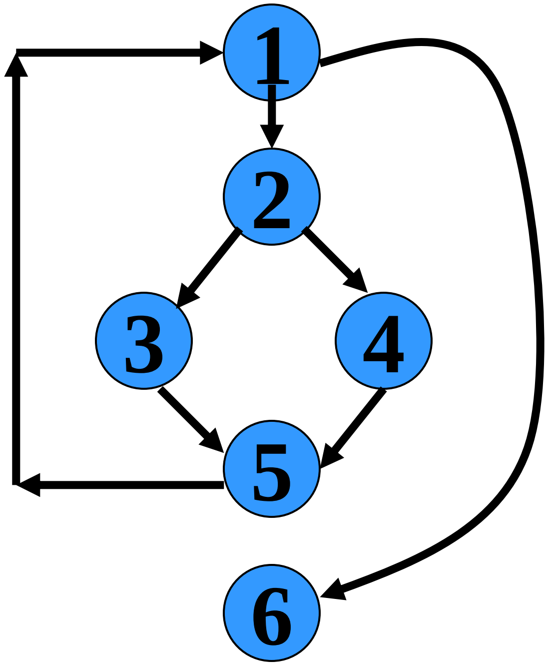
Cyclomatic complexity

- Another way of computing cyclomatic complexity:
  - inspect control flow graph
  - determine number of bounded areas in the graph
- V(G) = Total number of bounded areas + 1

Bounded area

- Any region enclosed by a nodes and edge sequence.

Example Control Flow Graph



Example

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

Cyclomatic complexity

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

Cyclomatic complexity

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

Cyclomatic complexity

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

Cyclomatic complexity

- Defines the number of independent paths in a program.
- Provides a lower bound:
  - for the number of test cases for path coverage.

Cyclomatic complexity

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

Path testing

- The tester proposes:
  - an initial set of test data using his experience and judgement.

Path testing

- A dynamic program analyzer is used:
  - to indicate which parts of the program have been tested
  - the output of the dynamic analysis
    - used to guide the tester in selecting additional test cases.

Derivation of Test Cases

- Let us discuss the steps:
  - to derive path coverage-based test cases of a program.

Derivation of Test Cases

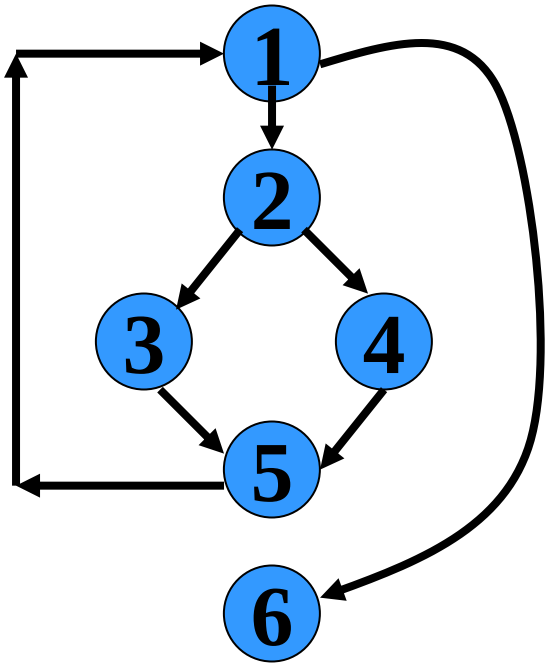
- Draw control flow graph.
- Determine V(G).
- Determine the set of linearly independent paths.
- Prepare test cases:
  - to force execution along each path.

Example

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

S ✓
B ✓
C
D

Derivation of Test Cases



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

An interesting application of cyclomatic complexity

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

Cyclomatic complexity

- Cyclomatic complexity of a program:
  - also indicates the psychological complexity of a program.
  - difficulty level of understanding the program.

Cyclomatic complexity

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