COMP 3111
SOFTWARE ENGINEERING

LECTURE 13
TESTING

LEARNING OBJECTIVES

1. Understand the purpose of testing.
2. Know three basic types of test cases and how they can be generated.
3. Understand the principles of component-based and system-based testing.
4. Learn different strategies for performing the different types of test cases.

TESTING OUTLINE

Testing Overview

Plan Tests

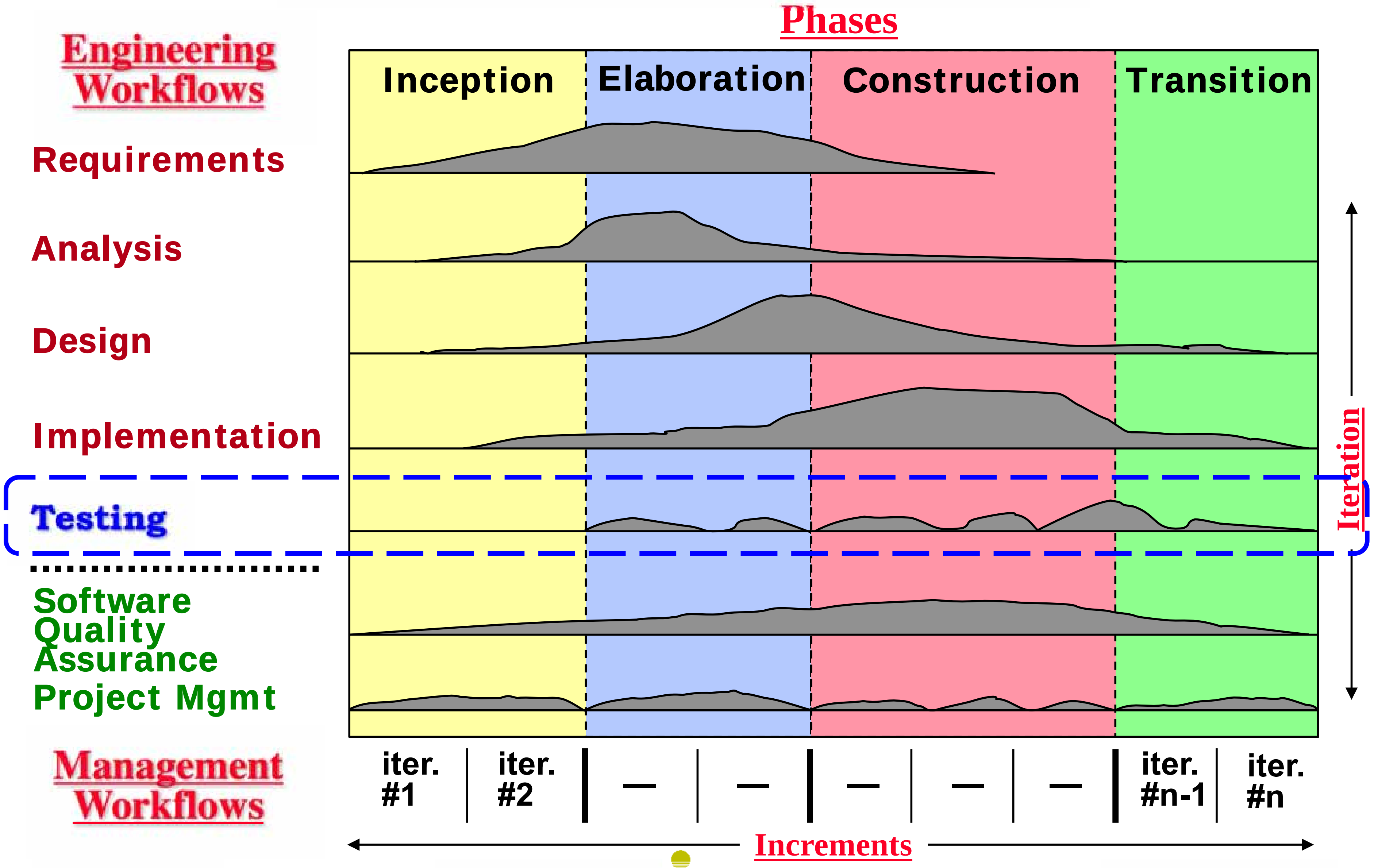
Design Tests
- White/Glass Box
- Black Box
- Regression

Implement Tests

Perform Tests
- Unit
- Integration
- System

Evaluate Tests

TESTING LIFE CYCLE ROLE



TESTING LIFE CYCLE ROLE

- In the Unified Process, testing is primarily in focus both during elaboration (when the executable architectural baseline is tested) and during construction (when the bulk of the system is implemented).
- During inception, elaboration and construction, the focus is primarily on integrating and system testing each build.
- During transition, the focus is on fixing defects (bugs) and doing regression testing.
- Previously developed test cases are used later as regression test cases.
- The test model is maintained throughout the life cycle, but it evolves:
  - obsolete test cases are removed.
  - some test cases are refined into regression test cases.
  - new test cases are created for each new build.

ARIANE 5 ROCKET

- The rocket self-destructed 37 seconds after launch.

  Reason: *A control software defect that went undetected.*

  – 64-bit floating point to 16-bit signed integer conversion caused an exception.
    - ➢ Floating point number was larger than 32,767 (max 16-bit signed integer).

  – Efficiency considerations had led to the *disabling of the exception handler*.

☞ **The program crashed → the rocket exploded.**

Total cost: over US$1 billion.

MARS POLAR LANDER

MARS POLAR LANDER: AN EXPEDITION TO THE SOUTH POLAR REGION
METEOROLOGY MAST (MET)
THERMAL EVOLVED GAS ANALYZER (TEGA)
ROBOTIC ARM
STEREO SURFACE IMAGER (SSI)
MEDIUM GAIN ANTENNA
MVACS PAYLOAD ELECTRONICS BOX
LIGHT DETECTION AND RANGING (LIDAR)
MARS DESCENT IMAGER (MARDI)
PROPELLANT TANK
SOLAR ARRAY WING
METEOROLOGY SUBMAST
DESCENT THRUSTERS

- The lander crashed because a sensor signal falsely indicated that it had touched down even though it was still 130 feet above the surface.

  Reason: *Software defect and insufficient testing.*

  – Software intended to ignore touchdown indications prior to the enabling of the touchdown sensing logic *was not properly implemented* and *not tested*.

  – The error was traced to *a single bad line of code*.

  ☞ **The descent engines shut down prematurely → the lander crashed.**

Total cost: US$120 million.

TESTING IS FOR EVERY SYSTEM

- These two examples show particularly costly errors.
- However, every little error adds up!
  - In 1982 the Vancouver Stock Exchange instituted a new stock index initialized to a value of 1000.000. The index was updated after each transaction. Twenty two months later it had fallen to 520.

  ☞ **Cause → updated value was truncated rather than rounded!**

  - The rounded calculation gave a value of 1098.892!
- Insufficient software testing costs $22-60 billion per year in the U.S.

☞ **If your software is worth writing, it's worth testing to make sure it is correct!**

THE PURPOSE OF TESTING

Testing tries to find differences between the specified (*expected*) and the observed (*actual*) system behavior.

- **Validation** makes sure that we have built the right product.

  We check that the system meets its stated requirements and each system function derives from some requirement.

  ☞ **Acceptance tests deal mainly with validation.**

- **Verification** makes sure that we have built the product right.

  We check the quality of the implementation by ensuring that each function works correctly and has no defects.

  ☞ **Most testing is targeted at doing verification.**

Goal: To design tests that will systematically find defects.

THE REALITY OF TESTING

- ☞ Testing *cannot* show the absence of software errors.

  (But it can increase quality and confidence!)

- ☞ It is impossible to completely test a nontrivial system.

  (Most systems will have bugs in them that either users uncover or that are *never uncovered*!)

- ☞ Testing is a destructive activity.

  (We try to make the software fail!)

- ☞ It is often difficult for software engineers to effectively test their own software.

  (Since they have no incentive to make their software fail!)

IS EXHAUSTIVE TESTING FEASIBLE?

Why do I need to plan my tests?

Why not just try my program with all inputs and see if it works?

```
int proc1 (int x, int y, int z)
    // requires: 1 ≤ x,y,z ≤ 10000
    // effects: computes some f(x,y,z)
```

How many runs are needed to *exhaustively* test this program?

Conclusion

It is imperative to have a plan for testing if we want to uncover as many defects as possible in the shortest possible time.

PLAN TESTS

Goal: Design tests that have the highest likelihood of finding defects with the minimum amount of time and effort.

Why?

A test plan specifies:

1. A testing strategy: what tests to perform; how to perform them; required test and code coverage; percentage that should execute with a specific result.
2. A schedule for the testing: when to run which tests.
3. An estimate of resources required: human/system.

☞ **Key problem:** choosing a test suite (input set) that is:

- small enough to finish quickly;
- large enough to validate and verify the software.

INPUT SPACE PARTITIONING

A program's behaviour is the "same" for "equivalent" input sets.

- Ideal test suite
  - A partition of the inputs into sets with the same behaviour so that we can test with one value from each set.

- Two problems

1. The notion of same behaviour is subtle.
   - Naïve approach: execution equivalence
   - Better approach: revealing subdomains
2. Discovering the sets of inputs requires perfect knowledge.
   - Need to use heuristics to approximate the sets cheaply.

NAÏVE APPROACH: EXECUTION EQUIVALENCE

```
int abs (int x) {
// returns:   x < 0        => returns -x
//            otherwise    => returns x

if (x < 0)    return -x;
else          return x;
}
```

- All **x** < 0 are execution equivalent.
  - The program *takes the same sequence of steps* for any **x** < 0.

- All **x** ≥ 0 are execution equivalent.
  - The program *takes the same sequence of steps* for any **x** ≥ 0.

☞ **This suggests, for example, that {-3, 3} is a good test suite.**

NAÏVE APPROACH: EXECUTION EQUIVALENCE

- However, consider the following buggy code.

```
int abs (int x) {
// returns:   x < 0        => returns -x
//            otherwise    => returns x

if (x < -2)   return –x;
else          return x;
}
```

- Two executions:

  x < -2   **x** ≥ -2

- Three behaviours:

  x < -2 (**OK**)   **x** = -2 or -1 (**BAD**)   **x** ≥ 0 (**OK**)

☞ **Using {-3, 3} as the test suite does not reveal the error!**

BETTER APPROACH: REVEALING SUBDOMAINS

- We say a program has the "same behaviour" for two inputs if it
  1. gives a correct result on both, or
  2. gives an incorrect result on both.

- A subdomain is a subset of the set of possible inputs.

- A subdomain is revealing for an error, E,
  1. if each element in the subdomain has the same behaviour.
  2. if the program has error E, then it is revealed by the test.

REVEALING SUBDOMAINS EXAMPLE

- For buggy **`abs`**, what are the revealing subdomains?

```
int abs (int x) {
// returns:   x < 0          => returns -x
//            otherwise      => returns x
if (x < -2)   return –x;
else          return x;
}
```

- Consider these input sets:

{-1}   {-2}   {-3, -1}   {-3, -2, -1}   **Which is best to reveal the error?**

Why?

How to partition the inputs/outputs into *revealing subdomains* that have a high likelihood to uncover errors?

REVEALING SUBDOMAINS: HEURISTICS

- To partition the inputs/outputs into revealing subdomains we use heuristics based on
  - program-dependent information (i.e., the actual code).
  - program-independent information (e.g., requirements specification, algorithm used, input/output data structures).
- A good heuristic gives:
  - few subdomains
  - for all errors in some class of errors E, high probability that some subdomain is revealing for E.
- Different heuristics target different classes of errors.
  - In practice, we usually combine multiple heuristics.

DESIGN TESTS: TEST CASE

A *test case* is one way of testing the system.

Specifies: *what to test*; under *what conditions*; *how to test*; the *expected result*.

Basic steps of a test case

1. Choose input data/software configuration.
2. Define the expected outcome.
3. Run the program/method against the input and record the result.
4. Compare the results against the expected outcome.

DESIGN TESTS: TEST CASE TYPES

White Box: **"testing-in-the-small"**

Verify component logic based on data or control structures.

Test cases use *knowledge of the internal workings* of a component.

☞ **Availability of source code is *required*.**

Black Box: **"testing-in-the-large"**

Verify component functionality based on the inputs and outputs.

Test cases use *knowledge of specified functionality* of a component.

☞ **Availability of source code is *not required*!**

Regression: **"re-testing"**

Verify no new defects are introduced after making a change.

Uses *selective* White Box and Black Box *test cases*.

DESIGN TESTS: TEST CASE DOCUMENTATION

Test case name → name of the test case

Description of the test → what the test is designed to do

Target class/component/subsystem name → name of the thing to be tested

Target class/component/subsystem operation → name of the operation to be tested

Test type → black box/white box; valid/invalid input or output

Test value(s) → inputs to be used for the test

Verification → expected result

It requires a lot of time and effort to generate all this information!

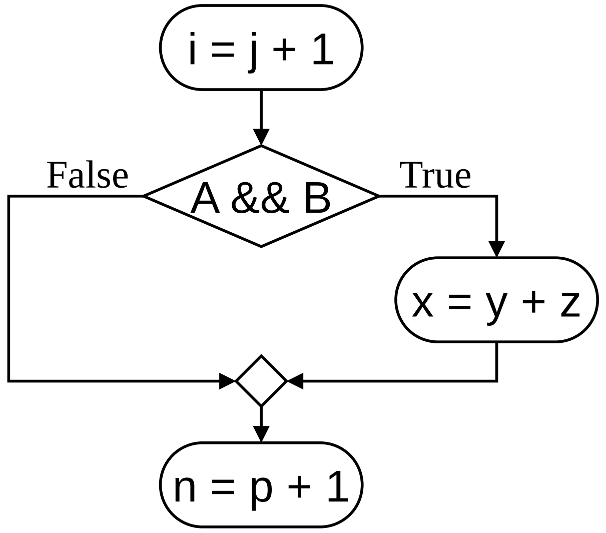
DESIGN TESTS: WHITE/GLASS BOX TESTING

Goal: Choose *subdomains* to ensure that we have executed all:

1. independent paths in the code at least once.
   - ☞ **Basis Path Testing**
2. logical decisions on their true and false sides.
   - ☞ **Condition Testing**
3. loops at their boundaries and within their bounds.
   - ☞ **Loop Testing**
4. internal data structures to ensure their validity.
   - ☞ **Data Flow Testing**

IS STATEMENT TESTING SUFFICIENT?

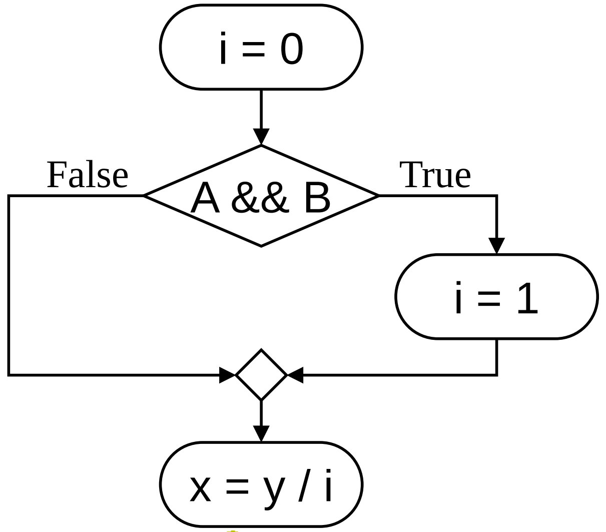
```
i = j + 1
if (A && B) {
   x = y + z;
   }
n = p + 1;
```



If we execute *every statement* in this program, is that sufficient to find errors?

Suppose we modify the program to:

```
i = 0
if (A && B) {
   i = 1;
}
x = y / i;
```



Do you see any problems with the modified program?

WHITE BOX TESTING: BASIS PATH TESTING

Goal: To execute each independent path at least once.

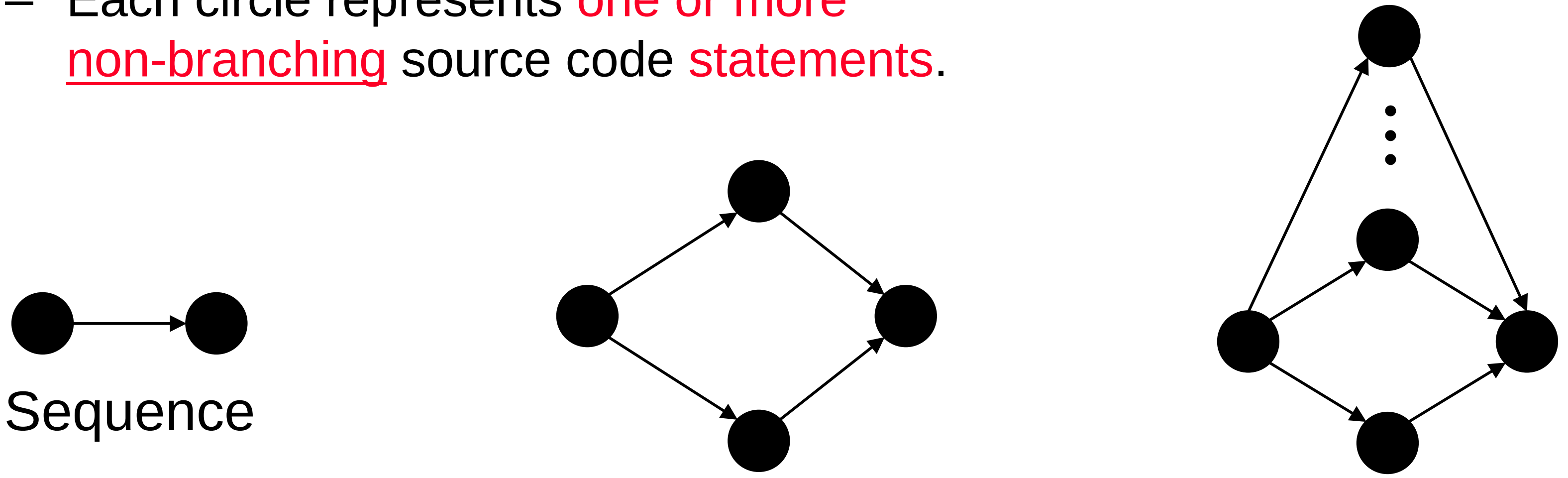
We derive a logical complexity measure for the code and then use this measure to define a basis set of execution paths.

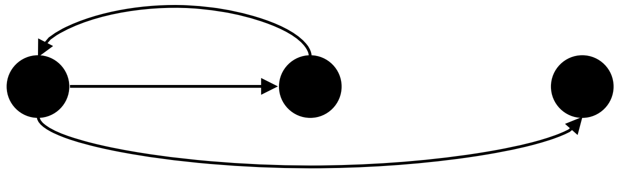
Overview

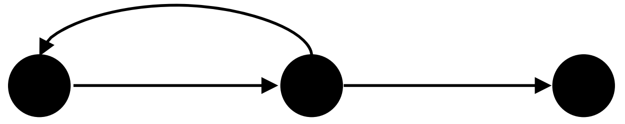
1. **From the code, draw a corresponding flow graph.**
2. **Determine the cyclomatic complexity of the flow graph.**
3. **Determine a basis set of linearly independent paths based on the cyclomatic complexity.**
4. **Prepare test cases that force the execution of each path in the basis set.**

WHITE BOX TESTING: BASIS PATH TESTING (CONT'd)

1. **From the code, draw a corresponding flow graph.**
   – Each circle represents one or more non-branching source code statements.



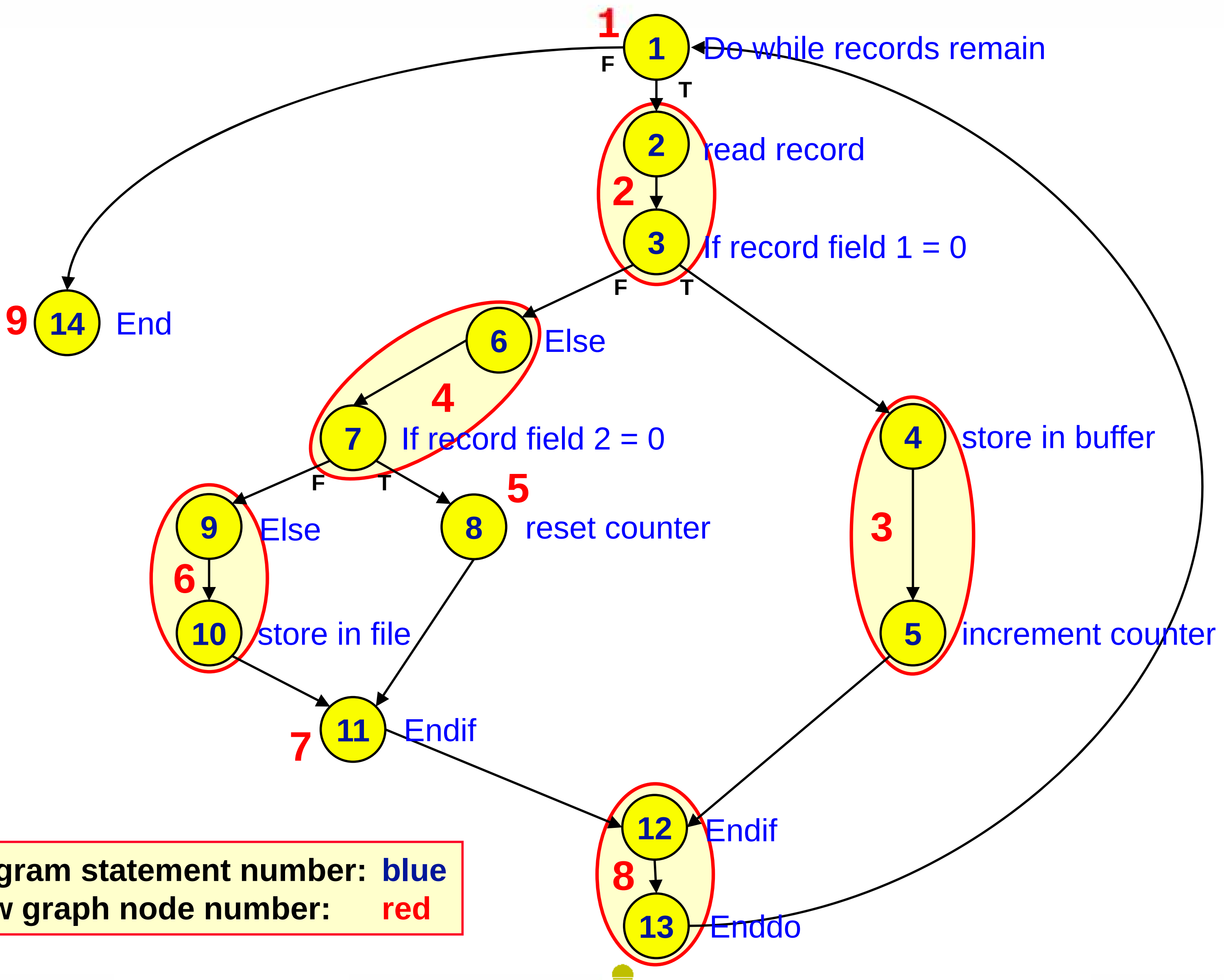

Do-while loop


Do-until loop

EXAMPLE BASIS PATH TESTING: FLOW GRAPH

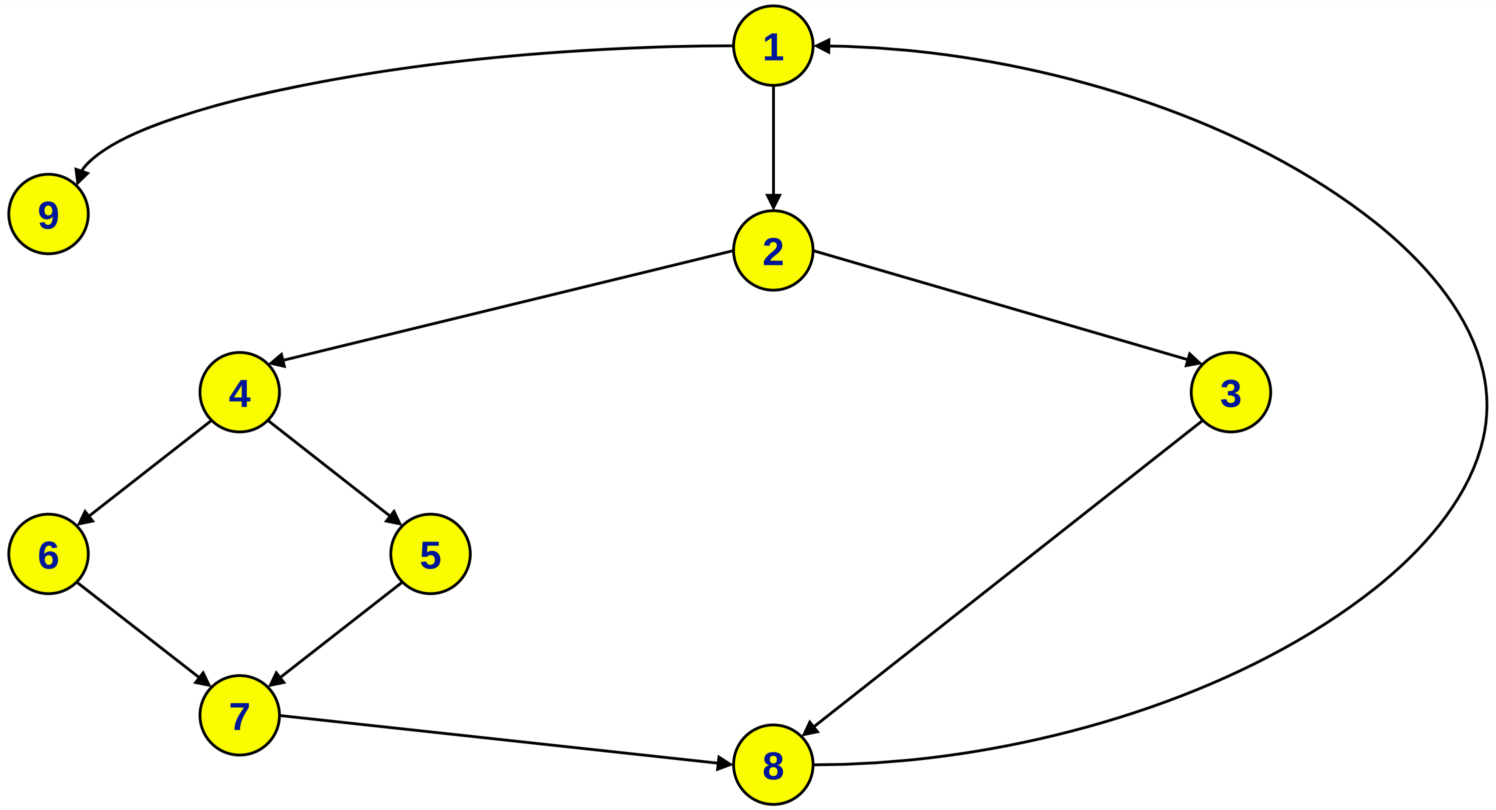
```
    Procedure: process records
1.      Do while records remain
2.          read record
3.          If record field 1 = 0
4.              store in buffer
5.              increment counter
6.          Else
7.              If record field 2 = 0
8.                  reset counter
9.              Else
10.                 store in file
11.             Endif
12.         Endif
13.     Enddo
14. End
```

EXAMPLE BASIS PATH TESTING: FLOW GRAPH



Program statement number: blue
Flow graph node number: red

EXAMPLE BASIS PATH TESTING: FLOW GRAPH



Flow graph node to program statement number mapping:

Node	Statement	Node	Statement	Node	Statement
1.	1	4.	6, 7	7.	11
2.	2, 3	5.	8	8.	12, 13
3.	4, 5	6.	9, 10	9.	14

WHITE BOX TESTING: BASIS PATH TESTING (CONT'd)

2. **Determine the cyclomatic complexity of the flow graph.**

cyclomatic complexity V(G): A quantitative measure of the logical complexity of the code.

> **Cyclomatic complexity provides an upper bound on the number of paths that need to be tested in the code.**

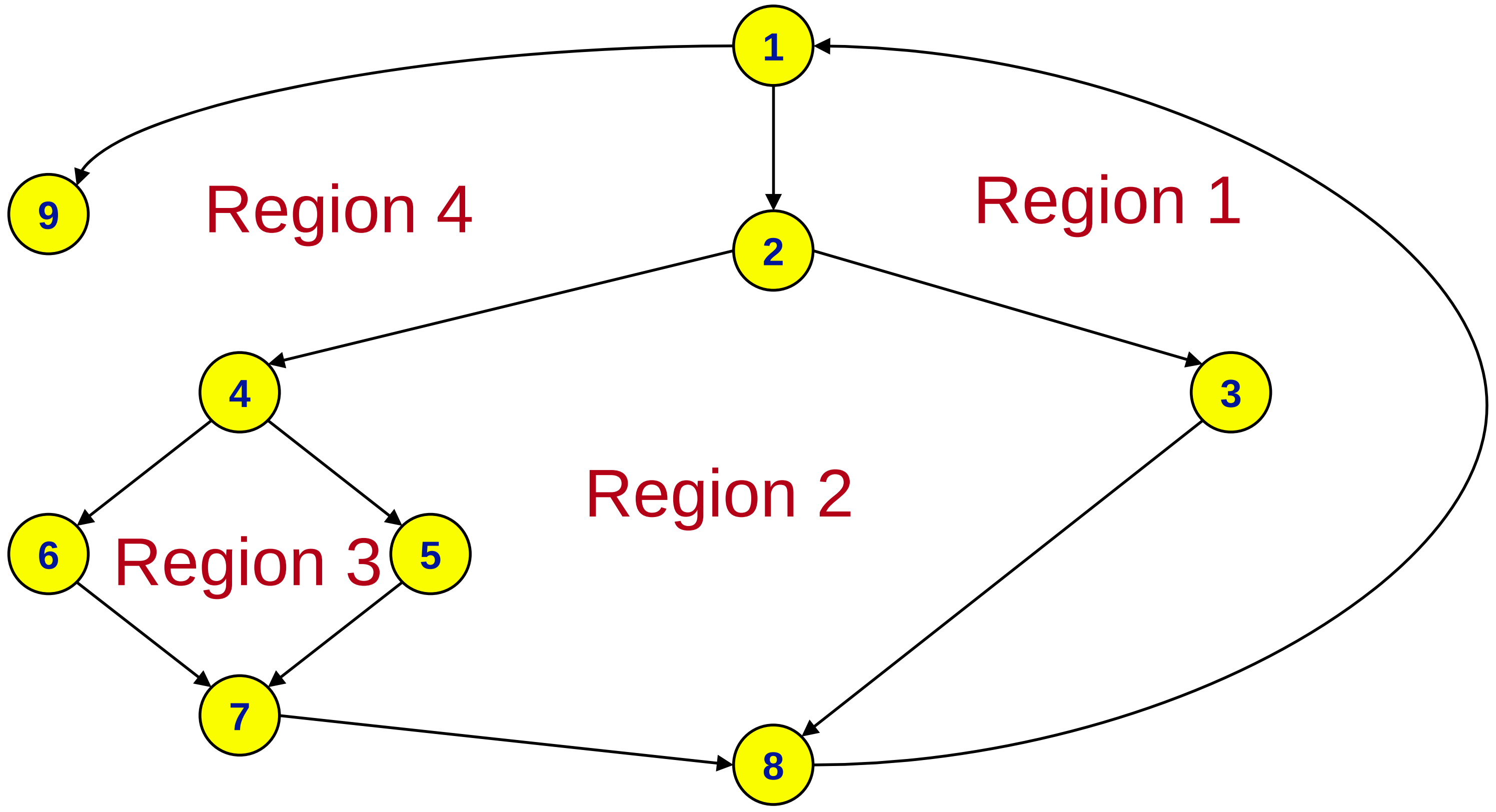
Ways to compute cyclomatic complexity V(G):

☞ V(G) = the number of regions (areas bounded by nodes and edges—area outside the graph is also a region)

☞ V(G) = the number of edges - the number of nodes + 2

☞ V(G) = the number of (simple) predicate nodes + 1

EXAMPLE BASIS PATH TESTING: CYCLOMATIC COMPLEXITY



V(G)=4

WHITE BOX TESTING: BASIS PATH TESTING (CONT'd)

3. **Determine a basis set of linearly independent paths based on the cyclomatic complexity.**

Independent path: a path that introduces at least one new set of processing statements or a new condition.

☞ An independent path must traverse at least one edge in the flow graph that has not been traversed before the path is defined.

Basis set: the set of linearly independent paths through the code.

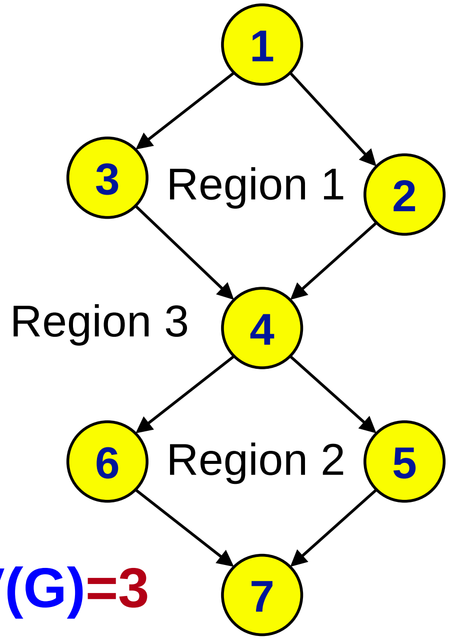
☞ A basis set in not unique.

Test cases derived from a basis set are guaranteed to execute every statement at least one time during testing.

☞ This is only the minimum number of test cases required.

BASIS PATH TESTING: INDEPENDENT PATHS

```
    Procedure: example()
1.  If c1
2.      f1()
3.  Else
4.      f2()
5.  Endif
6.  If c2
7.      f3()
8.  Else
9.      f4()
10. Endif
11. End
```



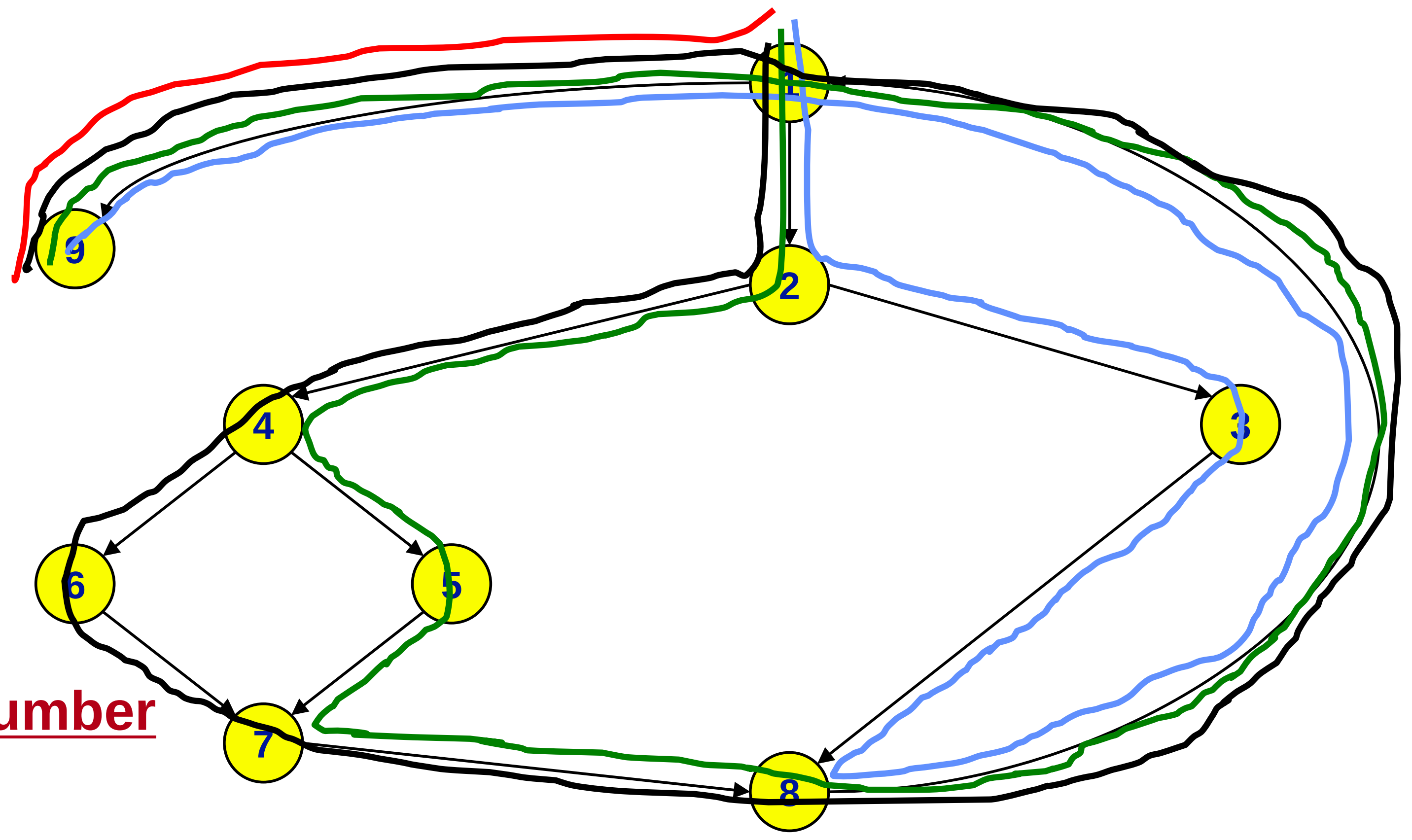
V(G)=3

How many independent paths are there in the basis set?

Recall: An independent path introduces at least one new set of statements or a new condition (i.e., it traverses at least one new edge).

☞ V(G) is just an **upper bound** on the number of independent paths.

EXAMPLE BASIS PATH TESTING: INDEPENDENT PATHS



Statement number

1-14

1-2-3-4-5-12-13-14

1-2-3-6-7-8-11-12-13-14

1-2-3-6-7-9-10-11-12-13-14

WHITE BOX TESTING: BASIS PATH TESTING (CONT'd)

4. **Prepare test cases that force the execution of each path in the basis set.**

Notes:

- The basis set refers to the statement numbers in the program.
- Count each logical test—compound tests count as the number of Boolean operators + 1 (i.e., count each simple predicate).
- Basis path testing should be applied to all components, if possible, and to critical components **always.**

Basis path testing *does not* test all possible combinations of all paths through the code; *it just tests every path at least once*.