Software Testing

Comp 47480: Contemporary Software Development

Why do we test software?

- We test to evaluate properties of software, e.g.
  - Correctness
  - Reliability
  - Performance
  - Memory Usage
  - Security
  - Usability
  - ...

- Our focus will be testing for **correctness**.
  - "Code does what it should"

Types of Testing

- User acceptance testing
- Integration testing
- Alpha testing, beta testing
- Black box testing, white box testing
- Unit testing

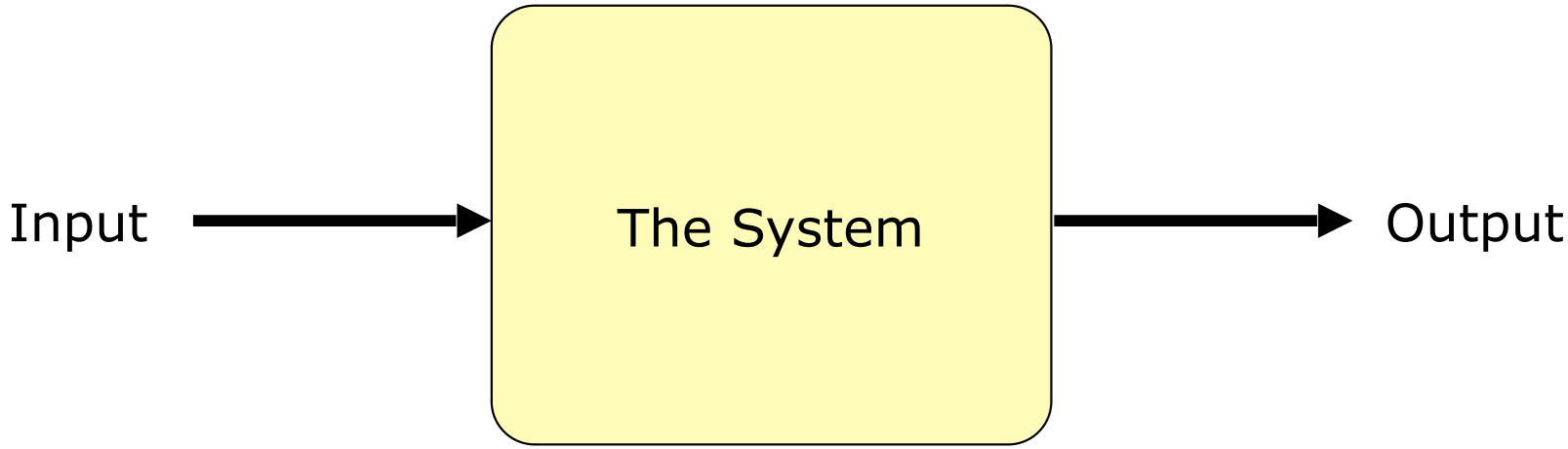
- Our focus will be on **unit testing**.
  - Testing classes/methods on their own
- We look at some theory first, then look at tool support for unit testing (JUnit) and at **Test-Driven Development** (TDD).

Roadmap of this Topic

- Introduction to Testing
- Black-box Testing
- White-box testing:
  - statement coverage
  - path coverage
  - branch testing
  - condition coverage
  - Cyclomatic complexity
- Summary

How do you test a system?

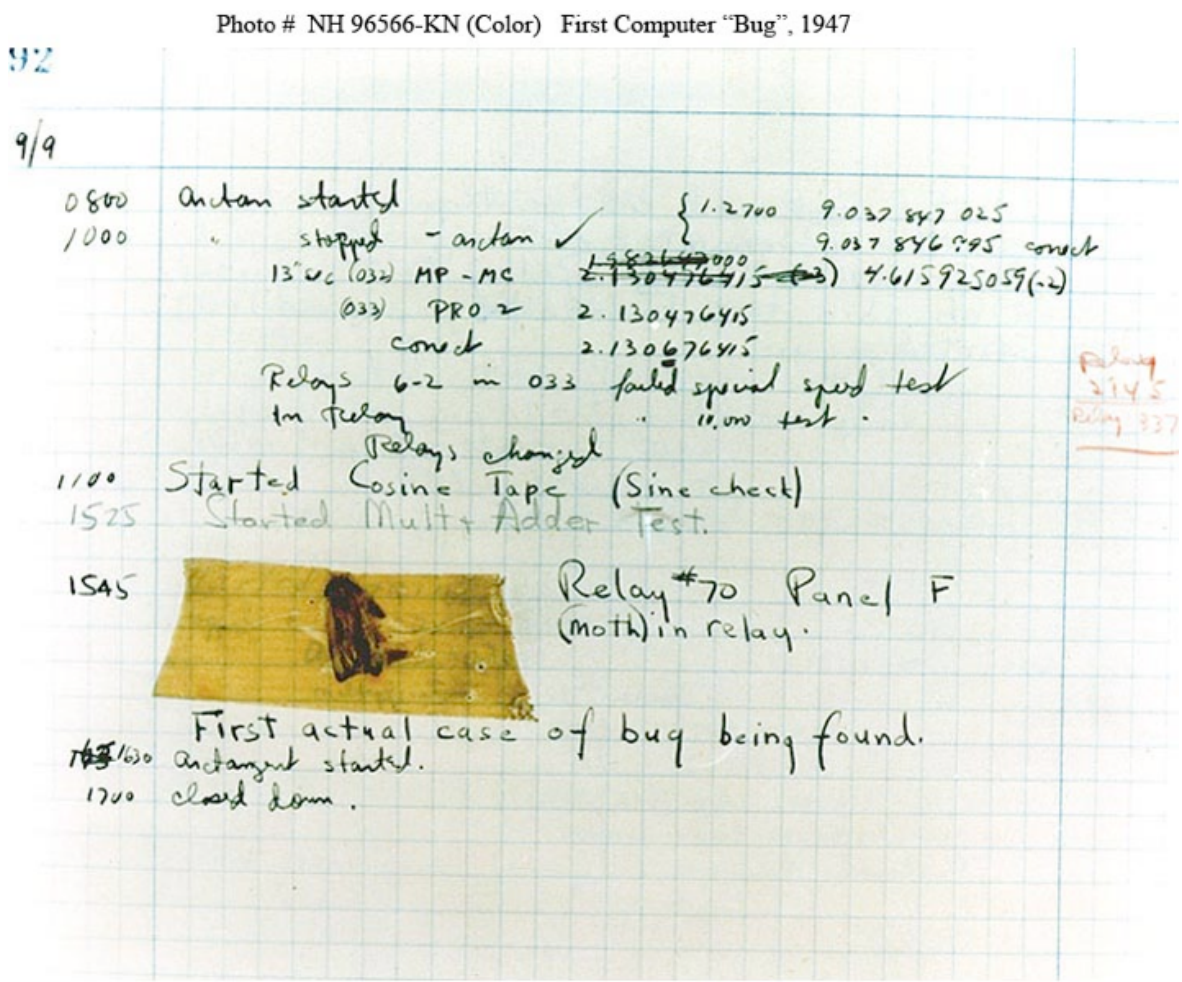
- Input test data to the system.
- Observe the output:
  - Check if the system behaved as expected.



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

Bugs and Failures

- A **bug** (aka **error**, **defect**) is a mistake in the code.
- A **failure** is a manifestation of an error.
- Presence of a bug may or may not lead to a failure.



Test cases and Test suite

- Test software using a set of carefully designed **test cases**:
  - the set of all test cases is called the **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 in which the data is input,
  - O is the expected output from the system.

It simplifies testing greatly if S can be ignored

- How do we know what the output should be?
  - We need an **oracle** that tells us what the output should be

- Does it ever make sense to test when we don't know what the outputs should be? Yes!

Regression Testing

- To **regress** is to 'return to a former or less developed state.'
  - a bad thing

- Regression in software is where updates to the software leads to the accidental introduction of new errors (or re-introduction of old bugs)
  - so the software gets worse in some sense

- **Regression testing** simply involves testing if the software has regressed
  - so creating test cases without knowing what the correct output can be very useful…
  - it enables regression testing to take place at a later stage

Exhaustive Testing

- Exhaustive testing of any non-trivial system is impractical
  - input data domain is extremely large.

- How long would it take to test this simple method **on all possible inputs**? Assume 32-bit integers and that we can do 1,000,000 tests per second:

```
/**
* Return gcd of x and y
**/
int gcd(int x, int y) {
   ...
}
```

- About 50 thousand years…

Design of Test Cases

- If test cases are selected randomly
  - many test cases will not detect errors not already detected by other test cases in the suite => waste of time running and maintaining useless tests

- The number of test cases in a test suite is **not** a good indication of the effectiveness of the testing.

- Design an optimal test suite:
  - of reasonable size
  - to uncover as many errors as possible.
  - Bigger does not mean better!

Remember it may take days to run all the test cases

How long to run test cases?

- One scrum master noted this in his blog:

Scenario II = 24 hours

Commit at 06:00 PM
Automatic build ready to be tested at 00:00 AM
Results at 6:00 PM (It takes 18h to run all the test cases, developers are still at the office)

- So creating a effective but trim test suite is important.

https://luis-goncalves.com/regression-testing/

Test Suite Examples

- Consider this method and two possible test suites for it

```
/**
* Return maximum of arguments
**/
int maxInt(int x, int y) {
   ...
}
```

- Test suite I: `{(x=3,y=2);(x=2,y=3)}`
- Test suite II `{(x=3,y=2);(x=4,y=3); (x=5,y=1)}`

Design of Test Cases

- Now consider this buggy implementation and decide which test suite is better:

```
/**
* Return maximum of arguments
**/
int maxInt(int x, int y) {
   int max;
   if (y > x)
      max = y;
   max = x;
   return max;
}
```

- Test suite I: `{(x=3,y=2); (x=2,y=3)}`
- Test suite II `{(x=3,y=2); (x=4,y=3); (x=5,y=1)}`

Design of Test Cases

- Systematic approaches are required to design a "good" test suite:
  - each test case in the suite should detect different errors.

- Two main approaches to designing test cases:
  - **Black-box** approach
  - **White-box** approach

From here on, we assume we're testing a single, stateless **method**.

Black-box/White-box Testing

- In **black-box testing**, test cases are designed using only a functional specification of the software under test (**SUT**), without any knowledge of the internal structure of the software.
  - Also known as **functional testing**.

- **White-box** testing involves examining the code itself.
  - requires knowledge about the internal structure of the software under test.
  - Also known as **glass box** testing or **structural testing**.

Black-box Testing

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

- (Two fancy names for two very simple concepts)

Simple Income Tax Example

- Consider this specification of a simple income tax system:

Income Tax System

Salary	Tax
€0 to €20K	15% of total income
€20K to €50K	€3K + 25% of amount over €20K
Above €50K	€10.5K + 40% of amount over €50K

The input clearly forms three **equivalence classes**:

€0 to €20K
€20K to €50K
€50K+

and at least one case from each class should be tested. Testing many cases from the same equivalence class is probably a waste of time.

Boundary value analysis

- Boundaries are areas where errors frequently lie:
  - Iterating once too many or once too few
  - Overrunning an array, or not processing the last element
  - Not considering zero or negative values…
- Many of these involve **off-by-one** bugs

In the previous example, there are clearly three boundaries to be tested explicitly:

€0
€20,000
€50,000

White-Box Testing

- The distinguishing feature of white box testing is that it's based on examining the code itself.

- What are we trying to achieve in white box testing?
  - think about this

- **Coverage**. Making sure that the code is well **covered** by tests

- If code isn't covered by one or more test cases, who knows what bugs might lurk there?

Code Coverage

- A key feature of any testing approach is to what extent the testing **covers** the code. Several approaches to code coverage exist:
  - statement coverage
  - branch coverage
  - condition coverage
  - path coverage
  - mutation testing
  - data flow-based testing

Statement Coverage

- Statement coverage:
  - design test cases so that every statement in a method is executed at least once.

```
int foo(int x, int y) {
   int z=0;
   if ((x>0) && (y>0)) {
      z = x;
   }
   return z;
}
```

- Testing with e.g. {(x=1, y=1)} gives 100% statement coverage.

Statement coverage criterion

- Based on the observation that an error in a method cannot be discovered unless the part of the method containing the error is executed.

- However, observing that a statement behaves properly for one input value provides no guarantee that it will behave correctly for all input values.

Example

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

Euclid's Greatest Common Denominator algorithm.

- How can we achieve 100% statement coverage?

- E.g. by choosing the test set {(x=4,y=3), (x=3,y=4)}, 100% statement coverage is achieved.

Branch Coverage

- Branch coverage means testing each branch of each control structure, e.g. in `if`, `case`, `while` statements

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

- Branch testing guarantees statement coverage, so it's a **stronger** testing criterion compared to statement coverage-based testing.

```
int foo(int x, int y) {
   int z=0;
   if ((x>0) && (y>0)) {
      z = x;
   }
   return z;
}
```

{(2,2)} gives statement coverage but not branch coverage

Branch Coverage Example

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

- How can we achieve 100% branch coverage?
- Possible test suite that gives **branch coverage** would be: {(x=3,y=3),(x=3,y=2), (x=3,y=4)}
- Remember we're doing **structural** testing. Don't try to be smart and use values that exploit the peculiarities of this example — they won't work in the general case.

Condition Coverage

- In **basic condition coverage** test cases are designed such that each boolean sub-expression of a composite conditional expression is given both true and false values.
  - Slightly more complex in the presence of "short circuit" evaluation of C++/Java, but we pass over that for simplicity

- In **complete condition coverage** test cases are designed such that all possible combinations of boolean values are tested,
  - requires 2^n test cases for n boolean variables
  - very demanding criterion

- **Modified Condition/Decision Coverage** is an "in between" technique, better than basic condition coverage but not as demanding as complete condition coverage.

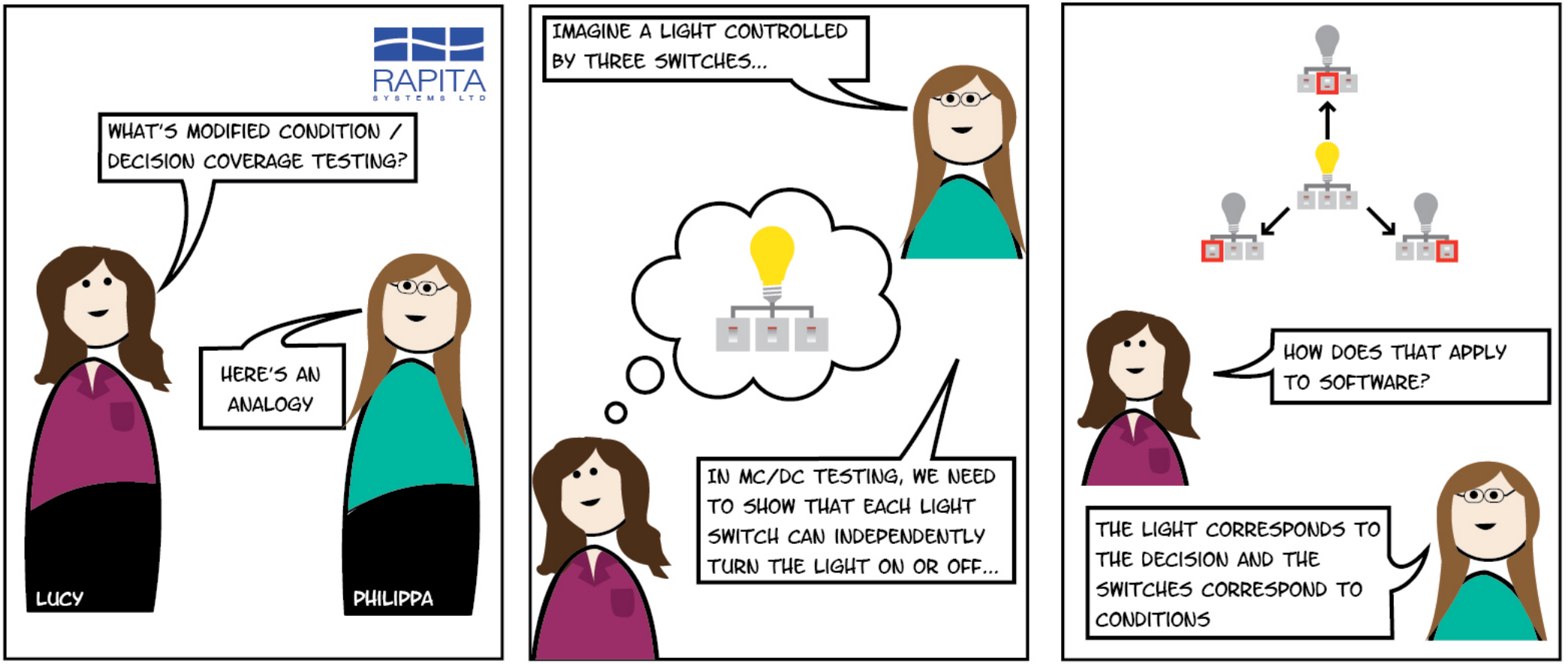
Modified condition/decision coverage

- Modified condition/decision coverage requires
  - basic condition coverage is satisfied, and
  - **each condition must affect the outcome independently.**

- More prosaically: each condition must be tested both being true and being false, in a context where its truth value determines the outcome of the overall decision.

- See example…

MC/DC Explained



www.rapitasystems.com

Modified condition/decision coverage

- Consider the expression: `if ((a || b) && c)`
  - {(a=true, b=true, c=true), (a=false, b=false c=false)} yields **basic condition coverage**
  - **complete condition coverage** requires 8 test cases

What testcases are sufficient to yield modified condition/decision coverage?

a	b	c	(a||b)&&c
FALSE	FALSE	TRUE	FALSE
TRUE	FALSE	TRUE	TRUE
FALSE	TRUE	TRUE	TRUE
TRUE	TRUE	FALSE	FALSE

All conditions take on both true and false in situations where they affect the decision outcome independently (the shaded boxes).

Path Coverage

- A **path** through a method is a route that control may flow from the method entry point to an exit point.
- If a method has a loop, testing **all paths** through the method is usually impossible.

- **Basis path coverage** means
  1. creating a maximum set of **linearly independent paths** for the method being tested
  2. designing test cases such that all **linearly independent paths** are executed at least once.

- **Linearly independent paths** are defined in terms of the **Control Flow Graph** (CFG) of a method.

- A CFG describes:
  - the possible sequences in which different statements of a method get executed ("flow of control").

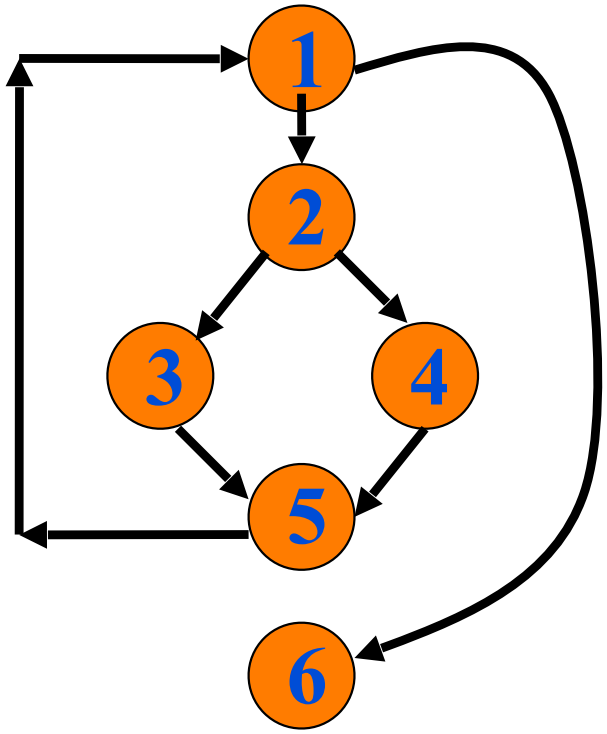
How to draw a Control Flow Graph?

- Number all the statements of a method.
- Numbered statements represent nodes of the CFG
- An edge from one node to another node exists if execution of the statement representing the first node can result in transfer of control to the other node.

What follows can all be expressed in formal graph theory. We take an informal approach.

CFG Example

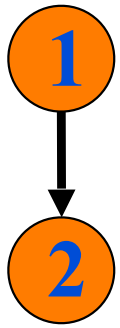
```
   int gcd(int x, int y) {
1.   while (x != y) {
2.      if (x>y)
3.        x = x-y;
4.      else y = y-x;
5.   }
6.   return x;
   }
```



Representing **Sequence** in a Control Flow Graph

Sequence:

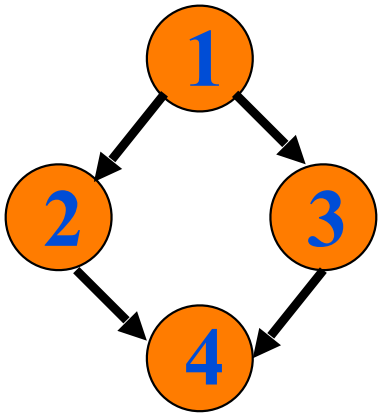
```
1   a=5;
2   b=a*b-1;
```



Representing **Selection** in a Control Flow Graph

Selection:

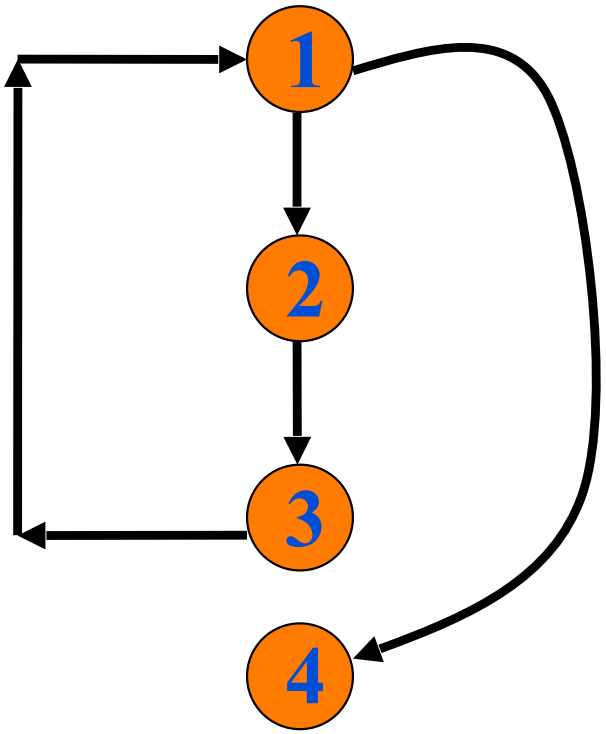
```
1   if(a>b) then
2     c=3;
3   else c=5;
4   c = c/2;
```



Representing **Iteration** in a Control Flow Graph

Iteration:

```
1   while(a>b) {
2      b=b*a;
3   }
4   x=10;
```

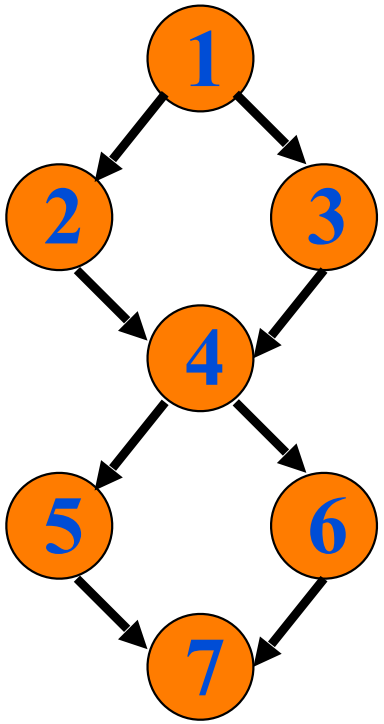


Linearly Independent Paths

- A set of paths is said to be **linearly independent** if no path through the CFG can be expressed as a linear combination (plus, minus) of the other paths in the set.
- We're interested in a **maximal** linearly independent set, where all possible paths can be expressed as a linear combination of the paths in the set.
- A set of test cases that exercise a maximal linearly independent set of paths, fully covers the method is some sense.

Linearly Independent Paths Example 1

```
1   if (cond1) then
2     body1
    else
3     body2;
4   if (cond2) then
5     body3
    else
6     body4;
7   ...
```

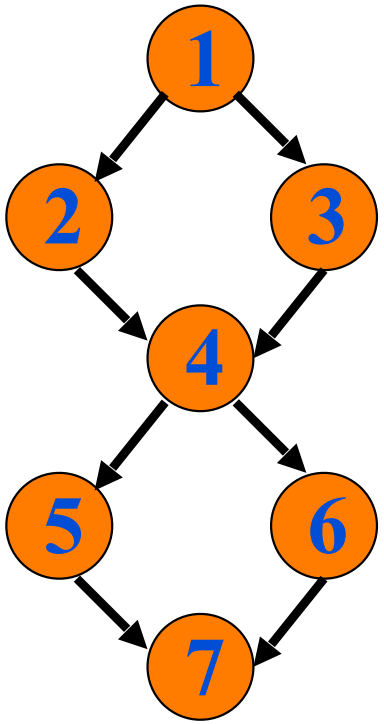


- There are 4 possible paths through this method: P1: (1,2,4,5,7), P2: (1,3,4,6,7), P3(1,2,4,6,7) and P4:(1,3,4,5,7).
- However they are **not linearly independent** as e.g.

  P4 = (P2 - P3) + (P1 - P3)

Linearly Independent Paths Example II

```
1   if (cond1) then
2     body1
    else
3     body2;
4   if (cond2) then
5     body3
    else
6     body4;
7   ...
```



- {P1, P2} is also not **maximal** linearly independent set, as neither P3 nor P4 can be expressed as a linear combination of P1 and P2.
- However e.g. {P1, P2, P3} is maximal linearly independent set as we've seen:

  P4 = (P2 - P3) + (P1 - P3)

McCabe's Cyclomatic Metric

- Provides an upper bound for the number of linearly independent paths to be tested in a method
  - it's an upper bound because some paths may in fact be impossible to traverse, e.g.

```
   int foo() {
1.   if (...)
2.     y = 10;
3.   if (y != 10)
4.     foobar();
5. }
```

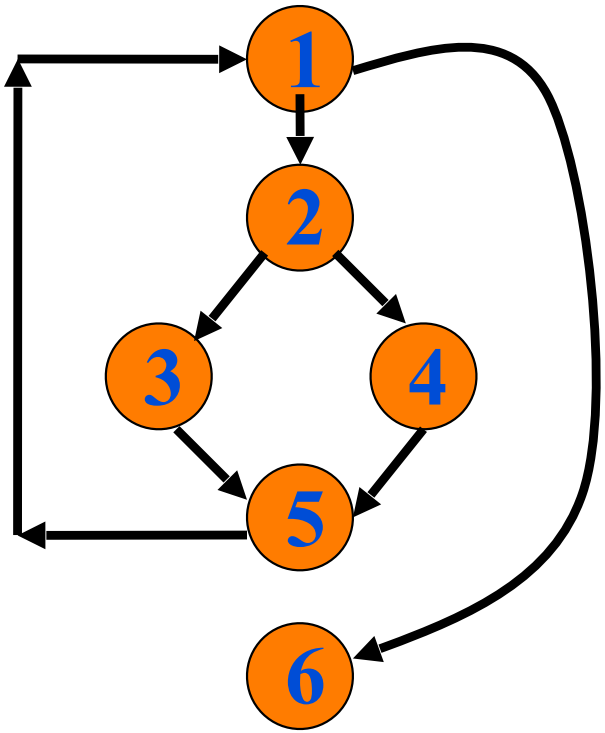
Structurally there seems to be four paths through `foo`:
1, 2, 3, 4, 5;
1, 3, 4, 5
1, 2, 3, 5
1, 3, 5
However the first is impossible to traverse.

- Cyclomatic complexity provides a practical way of determining the maximum number of linearly independent paths in a method.

Computing McCabe's Cyclomatic Metric

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

- A very simple way of calculating this is that it's the number of decisions in a method (if, while statements etc.) plus 1.



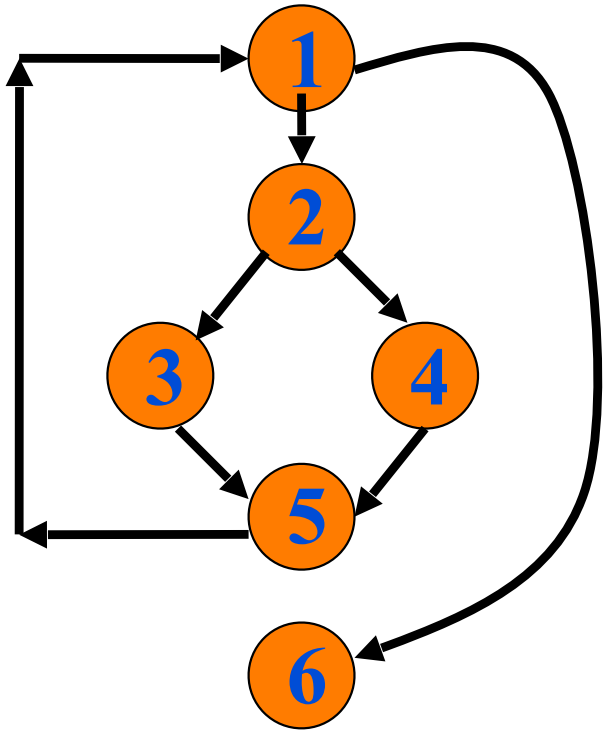
Cyclomatic complexity

- The cyclomatic complexity of a method provides a lower bound on the number of test cases to be designed to achieve **basis path coverage** (coverage of all linearly independent paths).

- 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.

Example

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



- Number of independent paths = E-N+2 = 7-6+2 = 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)`

Another Example

E.g.
{(x=0, y=0), (x=1, y=1)}
yields branch coverage.

In general, what bugs might this test suite miss?

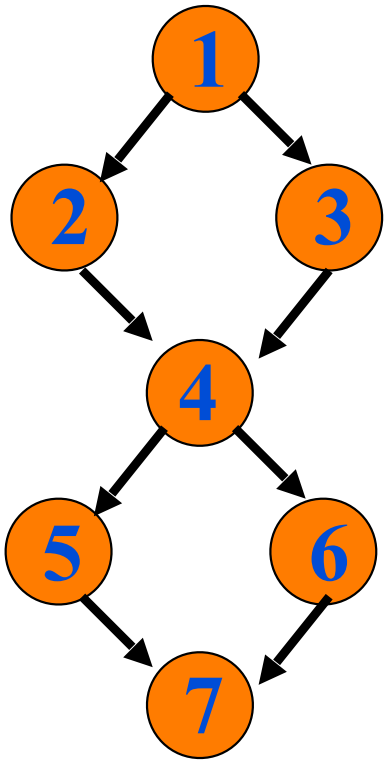
```
if (x==0)
  body1;
else
  body2;

if (y==0)
  body3;
else
  body4;
return
```

Applying Basis Path Coverage…

```
1. if (x==0)
2.    body1;
   else
3.    body2;

4. if (y==0)
5.    body3;
   else
6.    body4;
7. return
```



E = 8
N = 7
E-N+2 = **3**

Possible test cases for basis path coverage:
(x=0, y=0),
(x=0, y=1)
(x=1, y=0)

Application of cyclomatic complexity

- Relationship exists between cyclomatic complexity and
  - the number of errors existing in the code,
  - the time required to find and correct the errors.

- Cyclomatic complexity of a method also claimed to indicate the psychological **complexity** of a method and the difficulty level of **comprehension** of the method.

- Some companies demand method complexity to be below a certain threshold (~10)
  - E.g., some teams in Microsoft, Dublin *use* this metric as a guide, but does not set a threshold for it.

Criticism of Cyclomatic Complexity

- CC suffers from theoretical problems
  - e.g., level of nesting is ignored
- In empirical evaluation, several studies have found that Lines of Code better correlates with complexity than the CC metric.
- For more detail, see "A Critique of Cyclomatic Complexity as a Software Metric" http://csserver.ucd.ie/~meloc/47480/resources/critique_of_cyclomatic_complexity.pdf
- This practitioner's blog gives a good idea of some on the problems with CC as a metric: https://www.cqse.eu/en/blog/mccabe-cyclomatic-complexity

Summary

- Exhaustive testing of non-trivial systems is impractical.
- Two possible approaches to testing are:
  - **black-box testing**: Code is hidden, based on specification only
  - **white-box testing**: based on the structure of the code

- White-box testing strategies include: statement coverage, branch coverage, condition coverage, modified condition/decision coverage, path coverage.

- McCabe's Cyclomatic complexity metric:
  - provides an upper bound for number of linearly independent paths