Software Testing

Comp 47480: Contemporary Software Development

Roadmap of this Topic

- Introduction to Testing
- Black-box Testing
- White-box testing:
  - Statement coverage
  - Path coverage
  - Branch testing
  - Condition coverage
  - Cyclomatic Complexity
- Mocking
- Mutation Testing
- Test-Driven Development
- Summary

Why do we test software?

We test to evaluate properties of software, e.g.

Correctness

Reliability

Performance

Memory Usage

Security

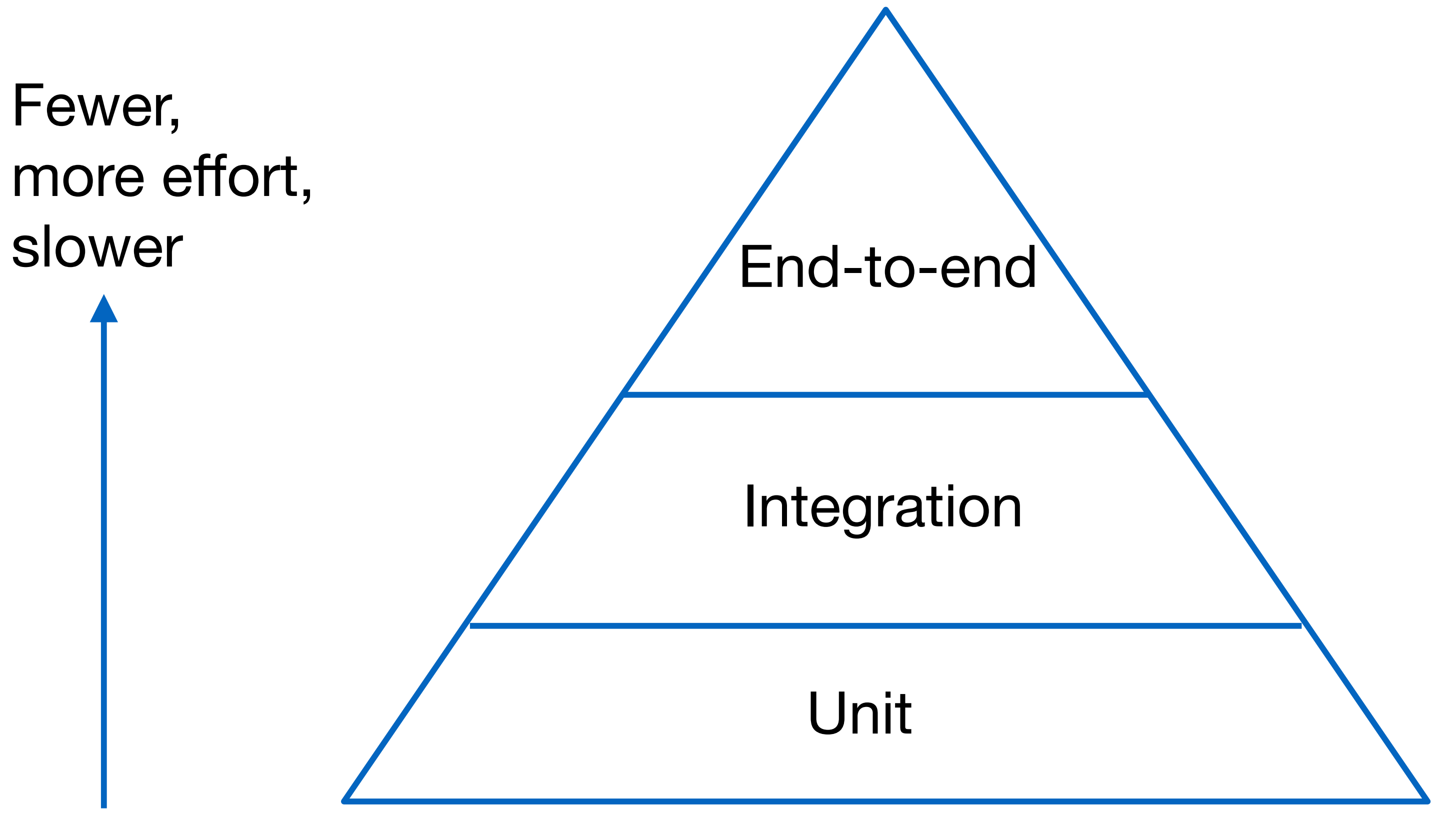
Usability

...

Our focus will be on testing for **correctness.**

“Code does what it should”

Testing Pyramid

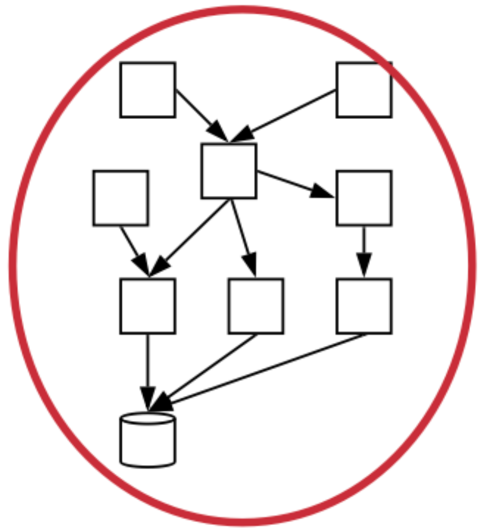
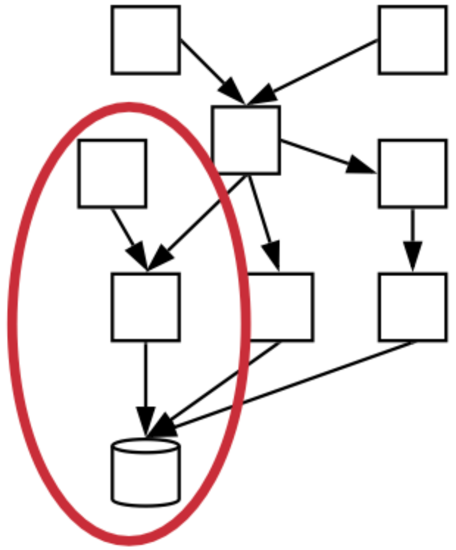
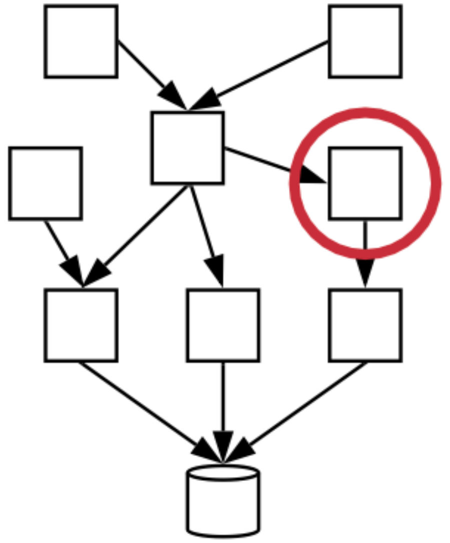


End-to-end testing: testing from UI to database.

Integration testing: testing that components work together correctly.

Unit testing: testing that individual components work correctly.

Testing Scope

End-to-end testing

Integration testing

Unit testing

Our focus in this module will be **unit testing.**

Recall too that we're focussed on non-critical systems -- bugs are undesirable but not catastrophic.

Other Types of Testing

Beta testing: testing in the real world prior to full release.

User acceptance testing: testing with real users.

Smoke testing: superficial test to ensure that basic functionality is correct.

Canary testing: releasing a new version of the software to a small group of users.

Black-box testing: testing based on a functional specification (aka *functional testing*)

e.g. integration testing

White-box testing: testing based on the structure of the software.

e.g. unit testing

Bugs and Failures

A **bug** (aka **defect**) is a mistake in the code.

A **failure** is a manifestation of an error (e.g. incorrect output or a crash).

Presence of a bug may or may not lead to a failure.

Test cases and Test suite

Software is tested 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, aka the test **fixture**
- O is the expected output from the system.

It simplifies testing greatly if S can be ignored.

Test case is a generic term. For this module, our test cases will invariably be **unit tests.**

Regression Testing

Does it ever make sense to create unit tests when we don’t know a priori what the outputs should be?

Regression in software is where updating the software introduces bugs into code that was working.

Regression testing involves testing if the software has regressed.

Creating a test suite even without knowing the correct output creates a 'snapshot' of expected system behaviour.

This test suite can be used for regression testing at a later stage.

Exhaustive Testing

Exhaustive testing is usually impossible as the input domain is simply too large.

E.g. how long would it take to test this method **on all possible inputs**? Assume 32-bit integers and a billion tests per second:

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

About 585 years

Desirable Features of a Test Suite

Fast: unit tests are run frequently.

Independent: Order of execution should not matter.

Repeatable: Unit tests should always provide the same result (ignoring **flaky tests** for now).

Self-checking: Unit tests should be able to check if they have passed, no developer intervention required.

Timely: Created by the developer as the production code is being written, not in a later test phase.

Also:

Fully-automated: Can be run with a single keystroke.
Self-contained: No external dependencies.

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 1: `{(3,2), (2,3)}`

Test suite 2: `{(3,2), (4,3), (5,1)}`

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 1: `{(3,2), (2,3)}`

Test suite 2: `{(3,2), (4,3), (5,1)}`

Design of Test Cases

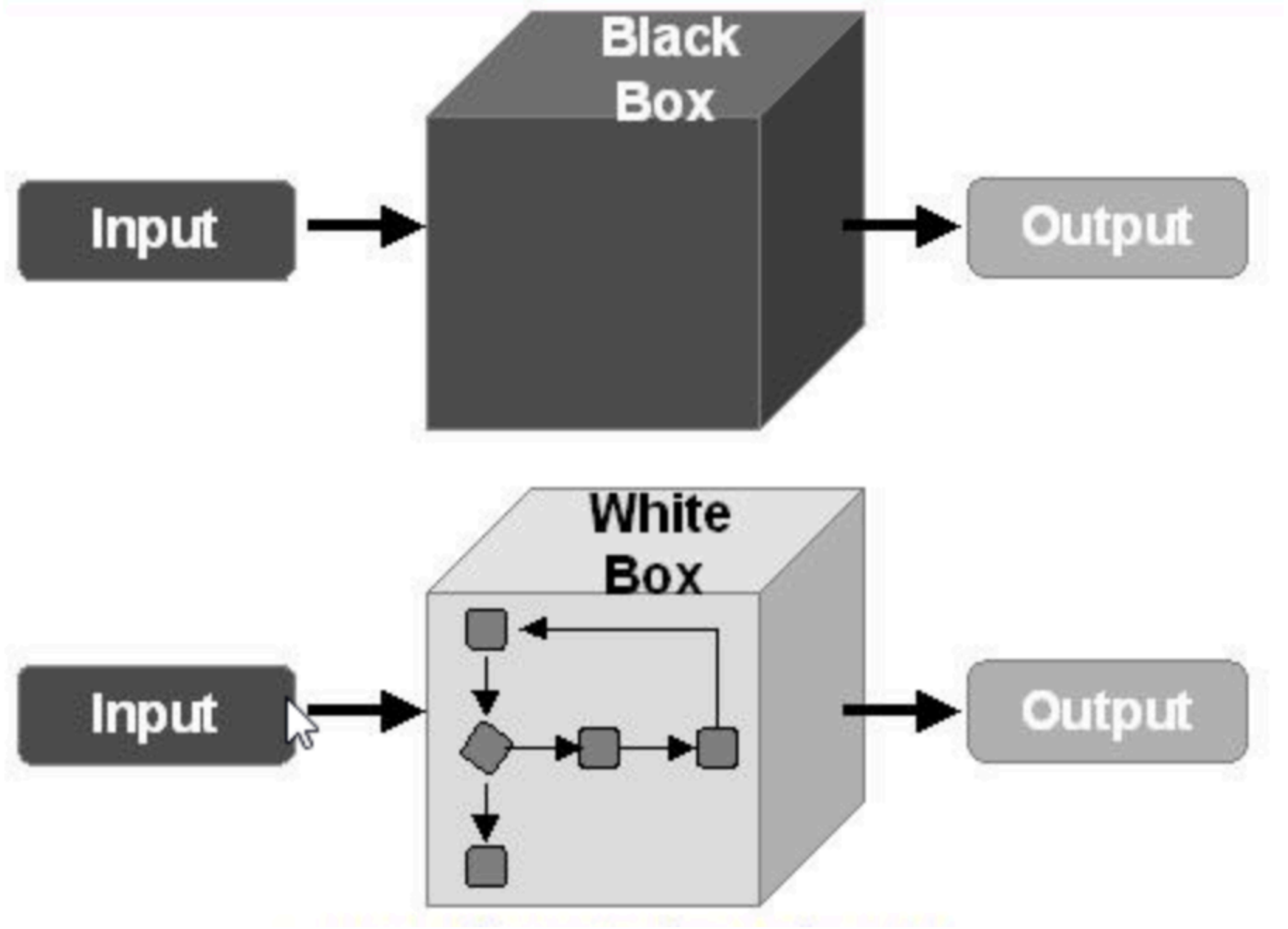
A systematic approach is required to design an effective test suite:

- each unit test should detect different errors.
- the tests should be "complete" in some way

- Two main approaches to designing unit tests:
  - **Black-box** approach
  - **White-box** approach

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

Black-box versus White-box Testing



Black-box testing relies solely on a functional description of the SUT.

White-box testing: based on the code structure of the SUT.

Black-box Testing

There are two basic approaches to designing black box test cases:

- **Equivalence class partitioning**
- **Boundary value analysis**

Equivalence class partitioning example

Consider a functional specification of a simple income tax system:

Salary	Tax
€0 to €20K	15% of total income
€20K to €50K	€3K + 25% of income over €20K
Above €50K	€5K + 40% of income over €50K

There are obviously three **equivalence classes.**

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 example

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

Aside: Hard Problems in Computer Science

There are only ***two hard problems*** *in Computer Science:*

1. *Naming things*
2. *Cache Invalidation*
3. *Off-by-one errors*

Note: Quotation partly attributed to Philip Karlton

The zero/one/many rule

Where the SUT takes an aggregation (list, array etc.) as argument, it should be tested where:

The aggregation has **zero** elements.

The aggregation has **one** element.

The aggregation has **many** (>1) elements.

Obviously these cases may be processed differently in the SUT, and so should be tested separately.

White-Box Testing

What are we trying to achieve in white box testing?

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

If some code isn't covered by one or more unit tests, no claim can be made about its correctness.

covered ~= executed by a unit test.

Code Coverage

In general, code that is “well covered” is better tested.

not covered => no correctness claim can be made

covered => improved confidence that code is correct

Many approaches to code coverage exist. We will look further at:

statement coverage
branch coverage
condition coverage
path coverage

We also look at a related technique: **mutation testing.**

Statement Coverage

We design test cases so that every statement in a method is executed at least once, e.g.

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

Testing with {(1, 1)} => 100% statement coverage.

Statement Coverage Example

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

Euclid's Greatest Common Denominator algorithm.

How can we achieve 100% statement coverage?

E.g. choosing the test set {(4, 3), (3, 4)} yields 100% statement coverage.

Rely on structure, not semantics!

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

Don't try to exploit the semantics of the code to reduce the number of tests required -- it won't work in general.

E.g. {(4,3)} covers all statements, only because of the semantics of this particular example.

Dead Code

100% statement coverage is always achievable, unless the system has **dead code**.

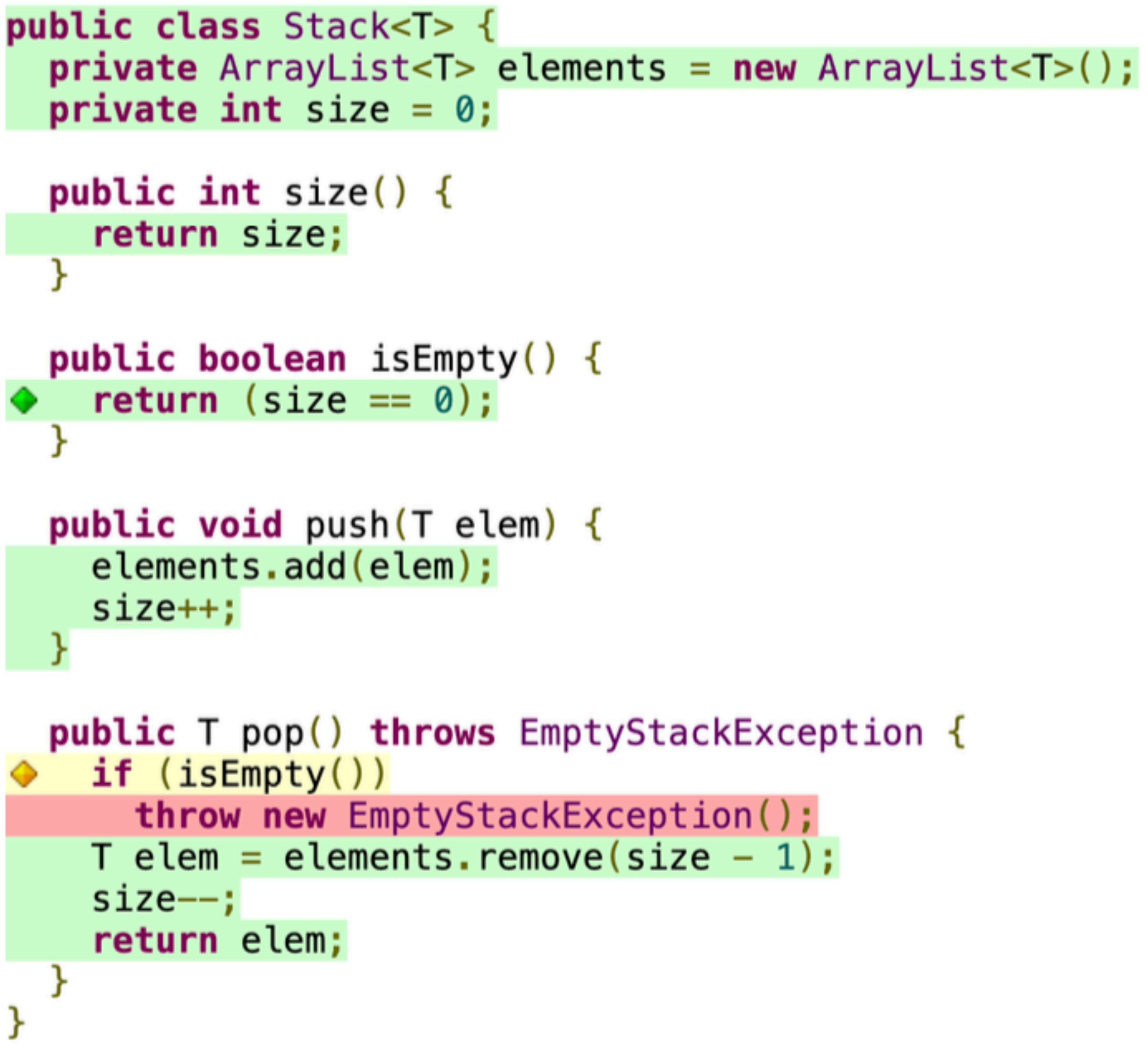
Dead code can never be executed and can be safely deleted from the system. It may, or may not, be testable

Aside: dead code is remarkably common!

- e.g. a study of one PHP application found that 30% of the files contained **entirely dead code** (about 2,500 files)

Source: Boomsma et al., "Dead code elimination for web systems written in PHP: Lessons learned from an industry case", Proceedings of the 28th International Conference on Software Maintenance, pp. 511-515, 2012.

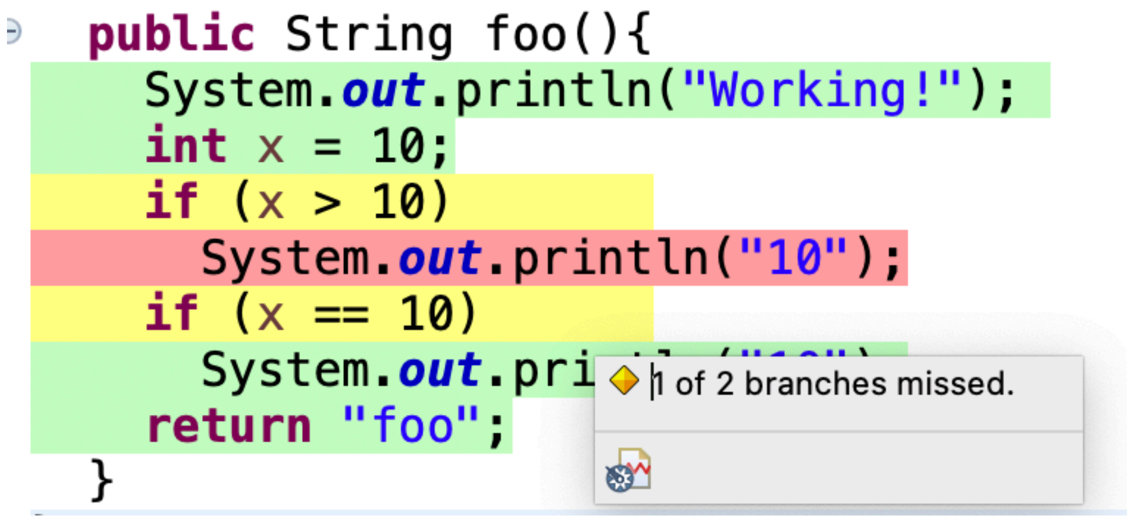
Tool Support -- JaCoCo



```
public class Stack<T> {
  private ArrayList<T> elements = new ArrayList<T>();
  private int size = 0;

  public int size() {
    return size;
  }

  public boolean isEmpty() {
    return (size == 0);
  }

  public void push(T elem) {
    elements.add(elem);
    size++;
  }

  public T pop() throws EmptyStackException {
    if (isEmpty())
      throw new EmptyStackException();
    T elem = elements.remove(size - 1);
    size--;
    return elem;
  }
}
```

All statements are covered (except) for the `throw` (red).

Tool Support -- JaCoCo



```
public String foo(){
    System.out.println("Working!");
    int x = 10;
    if (x > 10)
      System.out.println("10");
    if (x == 10)
      System.out.pri
    return "foo";
}
```

1 of 2 branches missed.

Yellow indicates partial coverage.

The first `if` statement is never executed where its condition is true, so its body is red.

The first `if` statement is only executed where its condition is true, so its body is green.

Neither `if` statement has **branch coverage**, so they are coloured yellow.

Recall the CI/CD Toolchain

JaCoCo is the sort of tool you might add to your CI/CD toolchain.

As part of preparing software for deployment, its test coverage is evaluated.

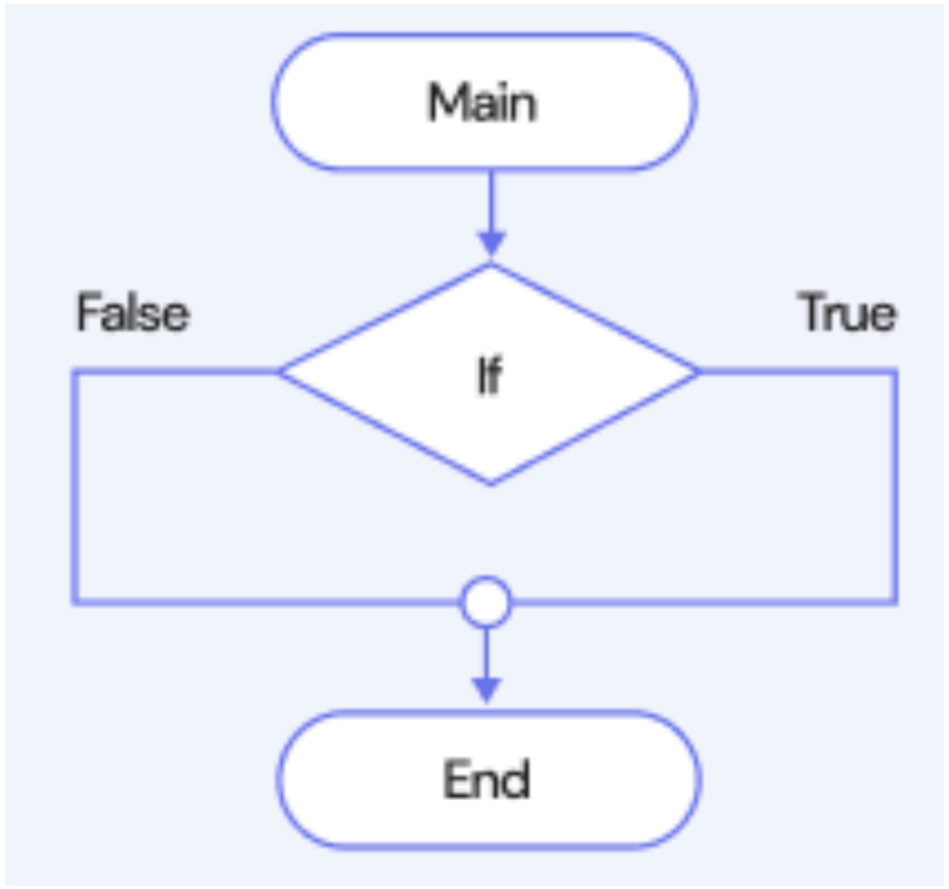
If coverage is poor, the developer can be alerted.

If coverage is very poor (e.g. <50%), perhaps the deployment should be rejected.

Suggests that not all the units test were found and executed.

Branch Coverage (aka Decision Coverage)

=> testing each branch of each control structure, e.g. `if`, `case`, `while` statements



Test all branch decisions with both true and false values.

Branch versus Statement Coverage

You can have statement coverage but not branch coverage, e.g.

```
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

However

branch coverage => statement coverage

Branch Coverage Example

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

How can we achieve 100% branch coverage?

Possible test suite that gives **branch coverage** would be:
{(3,3), (3,2), (3,4)}

Conditions and Decisions

A **condition** is a boolean expression that cannot be decomposed into a simpler boolean expression, e.g. (where **a** is boolean):

```
true
x == 10
b
```

A **decision** is a compound of conditions with zero or more Boolean operators, e.g.

```
a && b
(a || b) && (x == 35)
```

These are **decisions**, but not conditions

Basic Condition Coverage

In **basic condition coverage** each condition of a decision is tested with both true and false values.

So for this code fragment:

```
if (A && B) {
    ...
}
```

Basic condition coverage can be achieved with two unit tests.

Calculating Basic Condition Coverage

Basic condition coverage is usually expressed as the percentage of tested conditions, e.g.

```
if (A && B || C) {
   ...
}
while (X || Y && Z) {
    ...
}
```

6 conditions => 12 cases to be tested, so if e.g. the unit tests cover 4, we have 4/12 => 33% condition coverage.

(For simplicity, we ignore **lazy evaluation**, e.g. for **`A && B`**, **`B`** will not be evaluated if **`A`** is false.)

Does Basic Condition Coverage => Branch Coverage?

Unfortunately not.

Consider this simple case

```
if (A && B) {
    ...
 }
```

Testing with

```
A == true,  B == false
A == false, B == true
```

yields basic condition coverage, but not branch coverage.

Complete Condition Coverage

In **complete condition coverage** unit tests cover all possible combinations of boolean values

- requires 2^n test cases for n conditions
- very demanding criterion

So for this code fragment:

```
if (A && B) {
    ...
 }
```

Four test cases are required to achieve complete condition coverage.

Condition Coverage

Basic condition coverage is easy to achieve, but does not even guarantee branch coverage.

Complete condition coverage guarantees branch coverage, but requires many test cases.

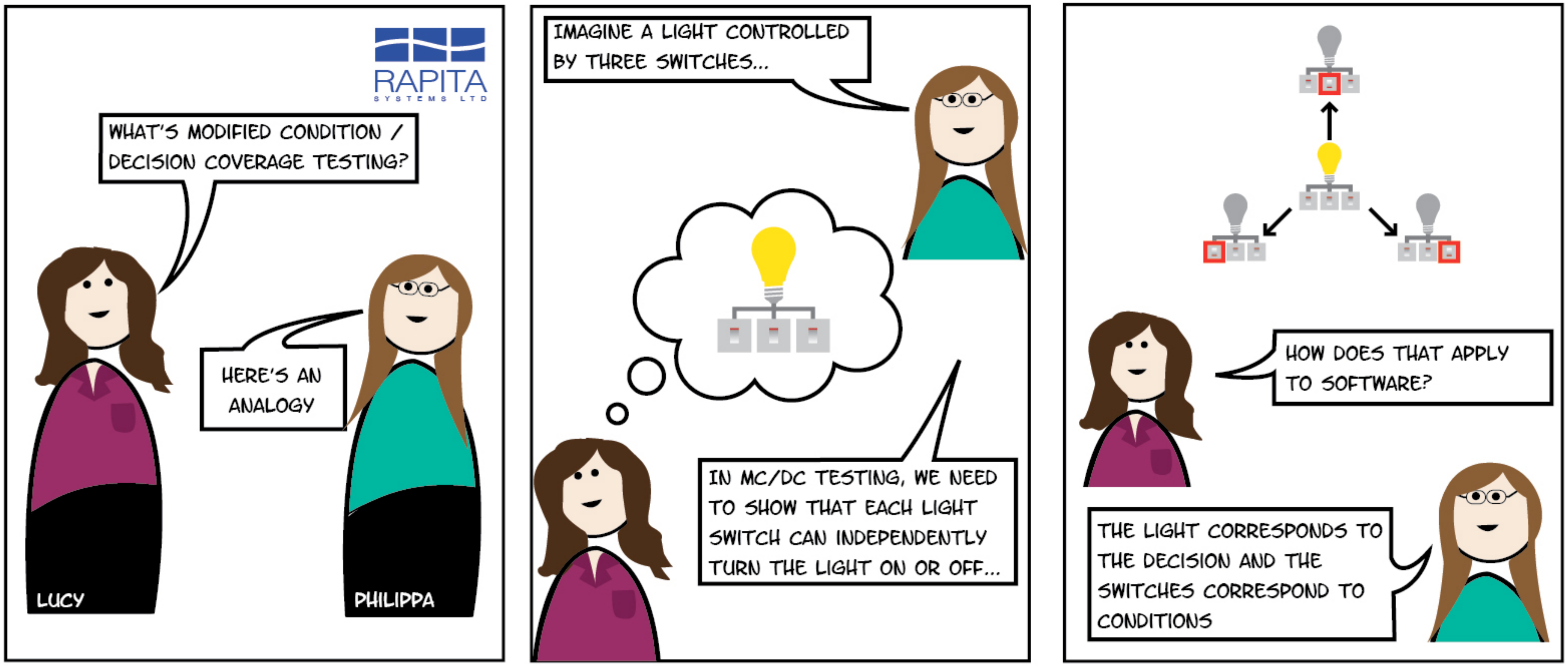
This led to the development of a condition coverage criterion that is less demanding than complete condition coverage, but guarantees branch coverage…

Modified Condition/Decision Coverage

- **MC/DC** is an “in between” technique, better than basic condition coverage but not as demanding as complete condition coverage.

- MC/DC coverage requires
  - basic condition coverage is satisfied, and
  - **each condition must affect the outcome independently, both when true and when false**

MC/DC Explained



RAPITA SYSTEMS LTD
WHAT'S MODIFIED CONDITION / DECISION COVERAGE TESTING?
HERE'S AN ANALOGY
LUCY
PHILIPPA
IMAGINE A LIGHT CONTROLLED BY THREE SWITCHES...
IN MC/DC TESTING, WE NEED TO SHOW THAT EACH LIGHT SWITCH CAN INDEPENDENTLY TURN THE LIGHT ON OR OFF...
HOW DOES THAT APPLY TO SOFTWARE?
THE LIGHT CORRESPONDS TO THE DECISION AND THE SWITCHES CORRESPOND TO CONDITIONS

www.rapitasystems.com

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.

MC/DC Example

- Consider the decision: `((a || b) && c)`
- **basic condition coverage** is achieved with two test cases, e.g.
  (a = true, b = true, c = true)
  (a = false, b = false c = false)
- **complete condition coverage** requires eight test cases

What testcases are sufficient
to yield MC/DC coverage?

MC/DC Example

What testcases are sufficient to yield MC/DC coverage?

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

MC/DC Example

Highlight the cases where toggling the condition (a, b or c) changes the decision ((a||b)&&c).

a	b	c	(a||b)&&c	use?
FALSE	FALSE	FALSE	FALSE	n
FALSE	FALSE	TRUE	FALSE	y
FALSE	TRUE	FALSE	FALSE	opt
FALSE	TRUE	TRUE	TRUE	y
TRUE	FALSE	FALSE	FALSE	n
TRUE	FALSE	TRUE	TRUE	y
TRUE	TRUE	FALSE	FALSE	opt
TRUE	TRUE	TRUE	TRUE	n

Here's one solution

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** is a route that control may flow from the method entry point to an exit point.

Testing **all paths** through a method is desirable, but is usually impossible in the presence of loops.

To explain this further, we first look at what a **Control Flow Graph** is.

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;
   }
```

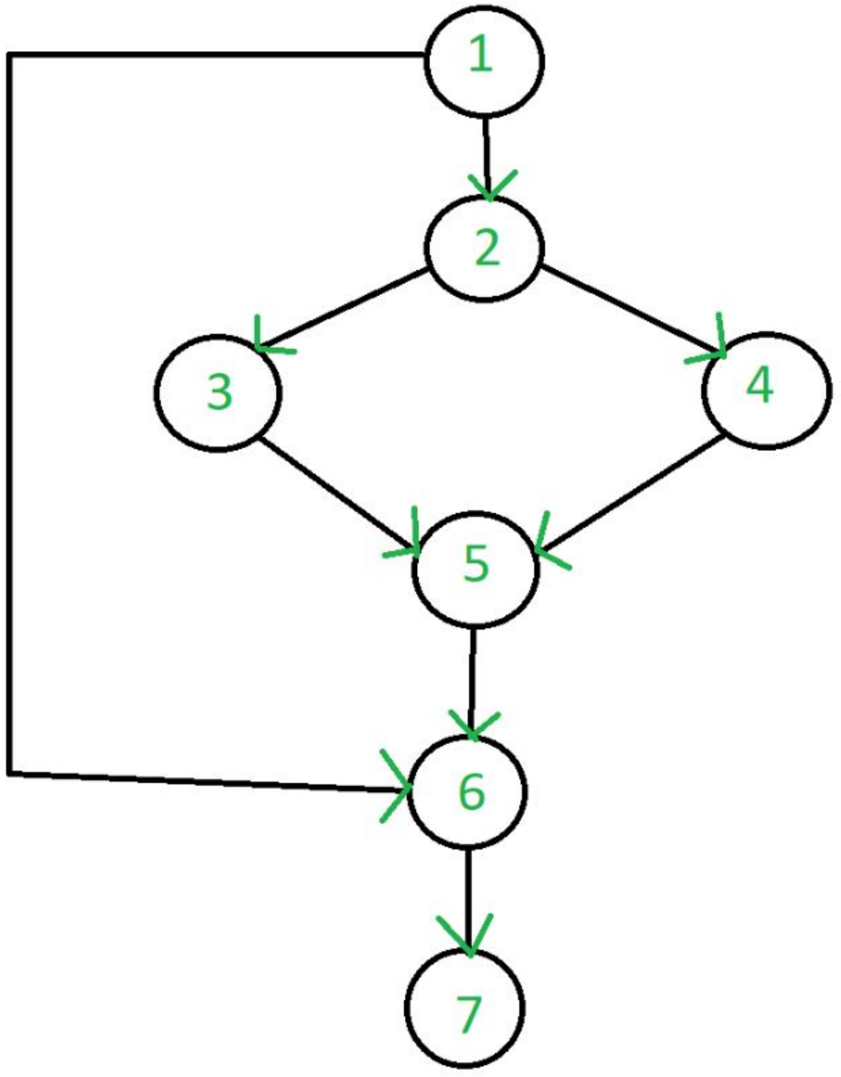
$$\begin{tikzcd}
 & 1 \arrow[d] \arrow[dddd, bend left=70] & \\
 & 2 \arrow[dl] \arrow[dr] & \\
3 \arrow[dr] & & 4 \arrow[dl] \\
 & 5 \arrow[uuu, bend left=70] & \\
 & 6 &
\end{tikzcd}$$

Note the CFG structure for decisions and loops.

Paths

- A **path** through a program/method is a node and edge sequence from the starting node to a terminal node of the CFG.

- There may be several terminal nodes (`return` statements), but for simplicity we'll assume there's only one



Control Flow Graph

Basis Path Testing

Testing all paths is impractical/impossible but is there a useful subset of paths we can test?

In 1976, Thomas McCabe proposed a solution, something he termed **basis path testing.**

This idea still has considerable traction in the software industry.

For example, until ~2020 JaCoCo reported if the number of unit tests was sufficient for basis path testing.

Basis path testing in a nutshell

View the set of paths through a method as a set of vectors.

The elements of the vectors are the nodes in the CFG. 1 indicates the node is on the path, 0 means it is not.

Find a linearly independent set of these vectors (using e.g. Gaussian Elimination).

Map these linearly independent vectors back to paths through the CFG.

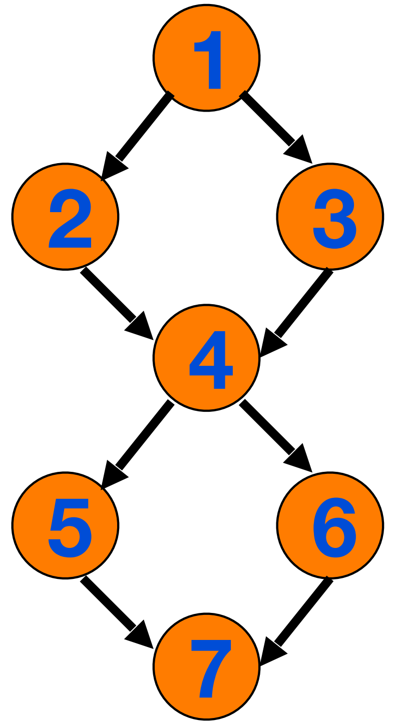
Create unit tests to traverse these paths.

The number of paths required is a popular metric, employed to measure the complexity of a method.

Linearly Independent Paths Example

	1	2	3	4	5	6	7
P1	1	1	0	1	1	0	1
P2	1	0	1	1	0	1	1
P3	1	1	0	1	0	1	1
P4	1	0	1	1	1	0	1

<=



The paths {P1, P2, P3, P4} are **not** linearly independent as e.g. P4 = P1 + P2 - P3

The paths {P1, P2, P3} **are** linearly independent.

McCabe's *Cyclomatic Complexity* Metric

The number of paths through a method required for basis path testing has become a popular way of measuring the complexity of the method.

For a CFG G, **Cyclomatic Complexity** is defined:

V(G) = E - N + 2,

where N is the number of nodes in G and E is the number of edges in G.

This is, very simply, the number of decisions in a method (`if`, `while` statements etc.) plus 1.

Cyclomatic Complexity as a metric

Cyclomatic Complexity (CC) is a commonly-used **software quality metric.**

Proponents claim code with a high CC score is:
- more error-prone
- harder to comprehend

However, in spite of its popularity it's of dubious merit.

Theoretical Issues with CC

Nesting is not considered, even though e.g. nested loop are clearly a greater cognitive burden than sequential loops.

`else` clauses do not contribute to CC

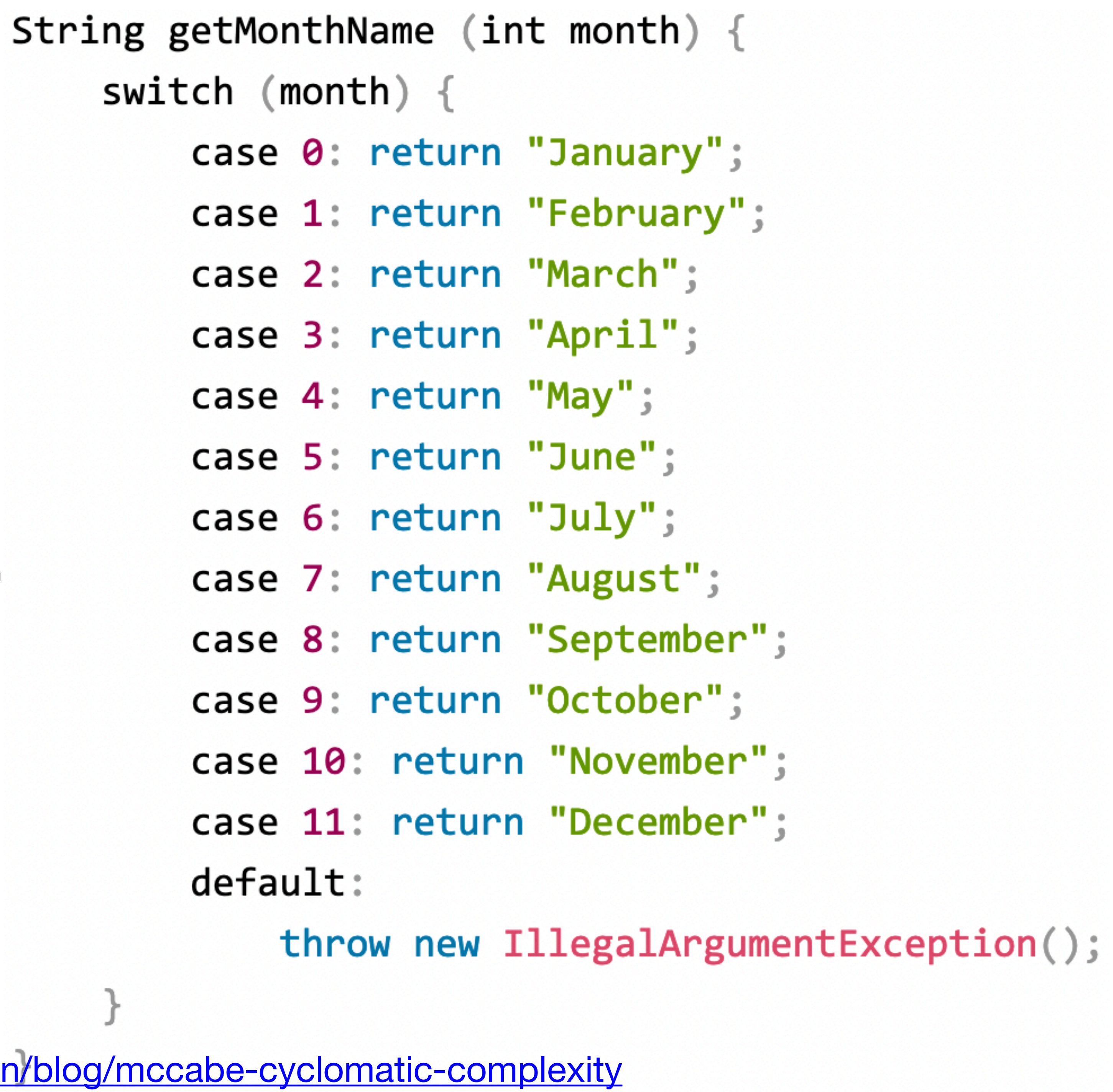
`switch` statements contribute disproportionately to CC (example next slide)

statement sequences do not contribute to CC, regardless of length

Source: Martin Shepperd, *A Critique of Cyclomatic Complexity as a Software Metric*, Software Engineering Journal, 1988.

CC and Case Analysis

This method is simple to understand, in spite of CC of **14**.



```
String getMonthName (int month) {
    switch (month) {
        case 0: return "January";
        case 1: return "February";
        case 2: return "March";
        case 3: return "April";
        case 4: return "May";
        case 5: return "June";
        case 6: return "July";
        case 7: return "August";
        case 8: return "September";
        case 9: return "October";
        case 10: return "November";
        case 11: return "December";
        default:
            throw new IllegalArgumentException();
    }
}
```

Source: https://www.cqse.eu/en/blog/mccabe-cyclomatic-complexity

Empirical Issues with CC

Theoretical issues can be ignored if a technique is found to be of practical value.

Several studies have investigated if CC correlates with bug rate, with inconclusive results:

- Some correlation with bug rate is apparent
- Correlation between CC and LOCs very strong
- Correlation with bug *density* not apparent

So the only claim supported by empirical studies is "larger programs have more bugs" which is unremarkable of course.

How is CC used?

Some companies set a threshold for method complexity, e.g. VisualStudio documentation promotes a maximum value of 10.

More commonly (and sensibly), CC is used only as a *guide*.

E.g. some teams in Microsoft, Dublin use CC to detect outliers.

The industry-standard software quality tool, SonarQube, reports on CC values.

(For some explanation of the popularity of this maligned metric, see *Cyclomatic Complexity* (invited content), IEEE Software, Nov/Dec 2016.)

Back to Testing and Coverage...

We want our test suite to detect bugs.

Coverage (statement, branch, path) is only a *proxy measure*.

Statement coverage is essential to make any claim about correctness.

However it is simply not clear what stronger forms of coverage really mean in terms of bug detection.

How can we we better evaluate if our tests are good at bug detection??

Mutation Testing

Mutation Testing is a technique to make a test suite better at catching bugs.

In a nutshell, mutation testing involves:

1. Create a copy of the method you are testing
2. *Mutate* this new method (e.g. invert a decision)
3. Run the unit tests for the method under test
4. If they fail, all is good! Your suite found the bug.
5. If they pass, further exploration is needed.

Multiple mutants are created for each method.

Original GCD Method

```java
public class GCD {
  public static int gcd(int x, int y) {
    while (x != y) {
      if (x > y)
        x = x - y;
      else
        y = y - x;
    }
    return x;
  }
}
```

... and one possible Mutant

```
public class GCD {
  public static int gcd(int x, int y) {
    while (x != y) {
      if (x >= y)
        x = x - y;
      else
        y = y - x;
    }
    return x;
  }
}
```

This is the mutation. ">" has been changed to ">=".

Mutation Testing -- terminology

Each mutated program is called a **mutant.**

If a mutant causes the tests to fail, is it said to be **killed.**

A mutant that causes the tests to pass, is termed a **surviving mutant.**

A surviving mutant that has the same behaviour as the original is termed an **equivalent mutant.**

A collection of mutants is termed a **horde.**

"Useless" Mutants

Many mutants won't help improve test cases. Some examples:

Replacing division with subtraction, but in a logging statement.

```
log.Infof("Found %d (%.2f %%)!", e,
           float64(e)*100.0 / total)
```

```
log.Infof("Found %d (%.2f %%)!", e,
           float64(e)*100.0 - total)
```

Replacing *greater than with less than* when comparing length of a collection to zero.

```
showCart := len(cart.GetItems()) > 0
```

```
showCart := len(cart.GetItems()) < 0
```

Changing a tuning parameter.

```
slo = (20 * time.Second)
```

```
slo = (20 * time.Second) + 1
```

Note: Thanks to Goran Petrović for these examples.

Equivalent Mutants

Equivalent mutants can be ignored, as they are simply different implementations of the original method, e.g.

```
int compareTo(Object o) {
  ...
  if (...) return +1;
  else return -1;
  }
  ...
}
```

original program

=>

```
int compareTo(Object o) {
  ...
  if (...) return +2;
  else return -1;
  }
  ...
}
```

mutant

Deciding that a mutant is equivalent is theoretically impossible in general. In practice, tool support can help.

One study found 40% of surviving mutants to be equivalent mutants! It's a serious challenge.

Surviving, Non-equivalent Mutants

Surviving, non-equivalent mutants are the interesting ones!

We've introduced a bug (the mutant), but the test suite failed to detect it. i.e. all unit tests passed.

Solution: Add a new unit test that kills this mutant.

The central thesis of mutation testing is that this new unit test makes the test suite better at catching real bugs.

Is this a valid assumption?

Does mutation testing improve testing?

A 2021 study performed in Google found:

1. Developers using mutation testing write more tests, and improve their test suites so that fewer mutants remain.

so what?

2. Mutants tend to be coupled with real faults. interesting...

In (2), ~1500 real high-priority bugs were analysed to determine if mutation testing would have led to the creation of a unit test that would have caught the bug.

In 70% of cases it would have. wow!

Source: Goran Petrović et al., "Does Mutation Testing Improve Testing Practices?," 43rd International Conference on Software Engineering (ICSE), 2021.

Mutation Testing Challenges

Even with tool support, this is a labour-intensive technique (surviving mutants have to be inspected).

Creating and running the mutants is automated, but time-consuming nevertheless.

Do the mutants represent the type of bugs that occur in practice? This is a key question; the Google study on the previous slides suggests yes.

Applying multiple mutations (higher-order mutation testing) aims to be more realistic.

Testing Summary

Unit testing is central in the development of reliable software.

Coverage is an easy way to evaluate a test suite, but the worth of this evaluation is unclear.

Cyclomatic Complexity has some use in determining how many unit tests are required, and is a weak measure of method complexity.

Mutation testing is more focussed on making the test suite better at fixing bugs.