Integrazione e Test di Sistemi Software

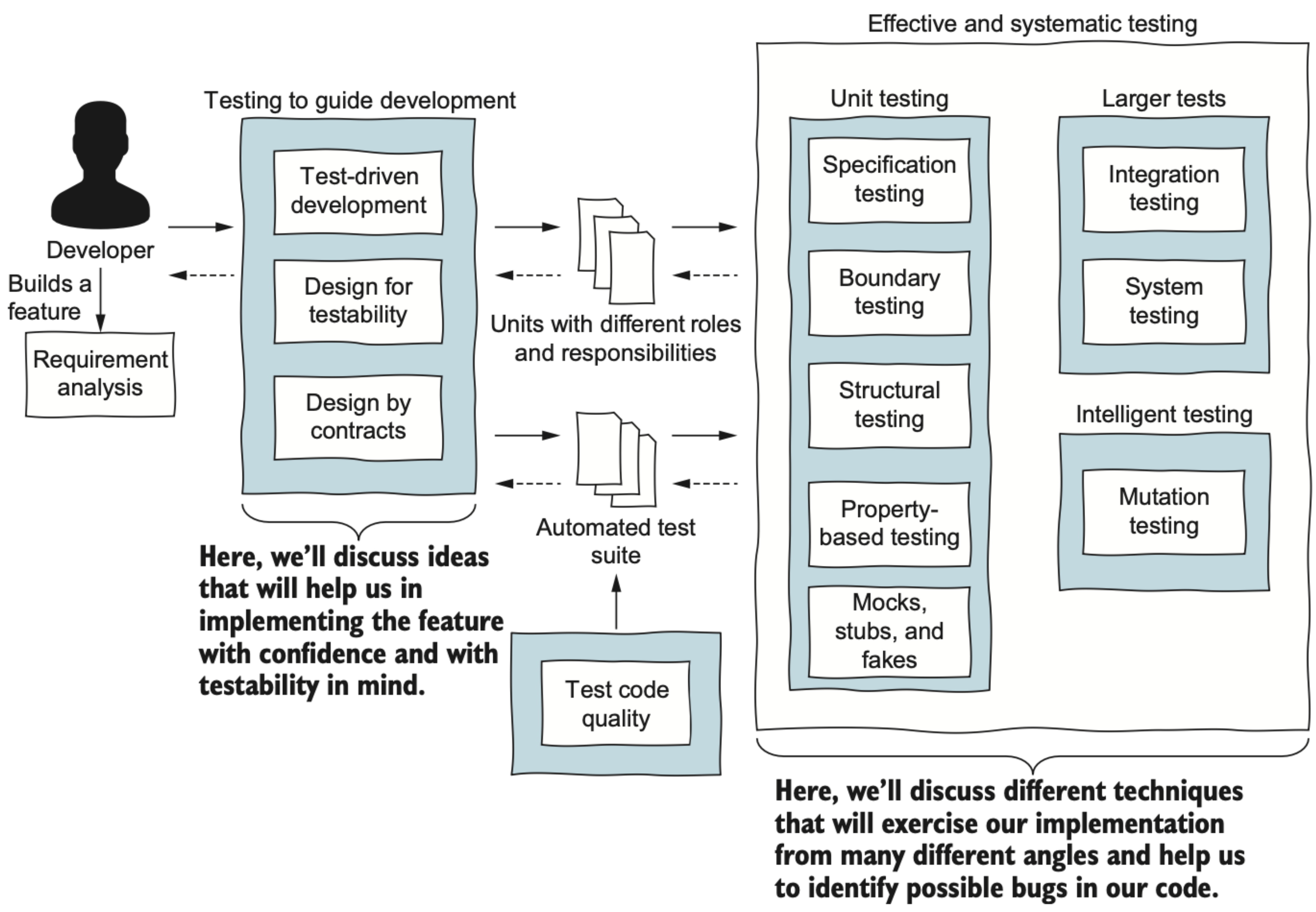
Structural testing and Code Coverage

Azzurra Ragone

Dipartimento di Informatica - Università degli Studi di Bari
Via Orabona, 4 - 70125 - Bari
Tel: +39.080.5443270 I Fax: +39.080.5442536
serlab.di.uniba.it

What is Structural Testing?

Effective and systematic testing (workflow)


Developer
Builds a feature
Requirement analysis
Testing to guide development
Test-driven development
Design for testability
Design by contracts
Here, we'll discuss ideas that will help us in implementing the feature with confidence and with testability in mind.
Units with different roles and responsibilities
Automated test suite
Test code quality
Effective and systematic testing
Unit testing
Specification testing
Boundary testing
Structural testing
Property-based testing
Mocks, stubs, and fakes
Larger tests
Integration testing
System testing
Intelligent testing
Mutation testing
Here, we'll discuss different techniques that will exercise our implementation from many different angles and help us to identify possible bugs in our code.

Strutural Testing: create test cases based on the code structure

Specification-based vs Structural testing

Specification-based testing:
Black-box test
The requirements guide the testing

vs

Structural testing:
White-box test
The source code guides the testing (and Code Coverage Criteria)

Code Coverage: an example

Spec: *Given a sentence, the program should count the number of words that end with either “s” or “r”. A word ends when a non-letter appears. The program returns the number of words.*

Example: *“Cats and dogs are in love”*

CountWords implementation

```
public class CountWords {
  public int count(String str) {
    int words = 0;
    char last = ' ';

    for (int i = 0; i < str.length(); i++) {

      if (!isLetter(str.charAt(i)) &&
       (last == 's' || last == 'r')) {
         words++;
      }

      last = str.charAt(i);
    }

    if (last == 'r' || last == 's') {
      words++;
    }

    return words;
  }
}
```

Loops through each character in the string

If the current character is a non-letter and the previous character was "s" or "r", we have a word!

Stores the current character as the "last" one

Counts one more word if the string ends in "r" or "s"

CountWords implementation

```
@Test
void twoWordsEndingWithS() {
  int words = new CountLetters().count("dogs cats");
  assertThat(words).isEqualTo(2);
}
```

Two words ending in "s" (dogs and cats): we expect the program to return 2.

```
@Test
void noWordsAtAll() {
  int words = new CountLetters().count("dog cat");
  assertThat(words).isEqualTo(0);
}
```

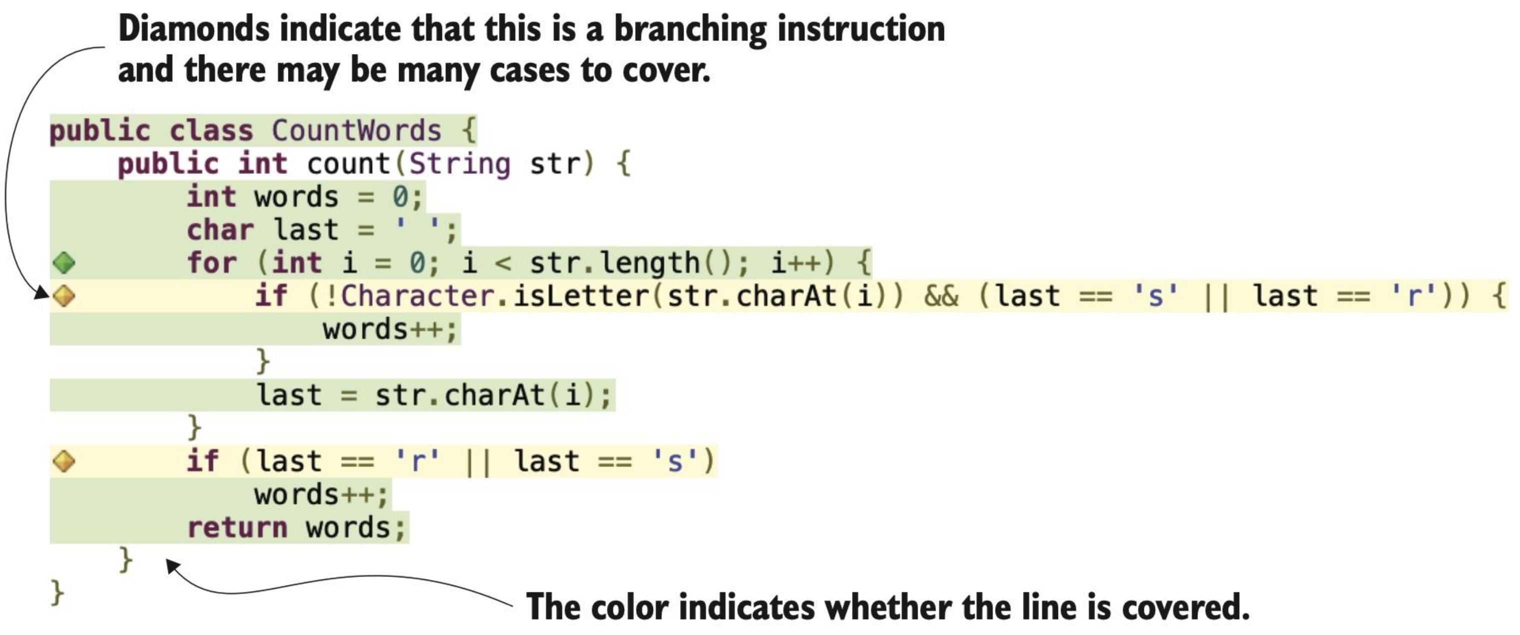
No words ending in "s" or "r" in the string: the program returns 0.

Do we miss something?

In Structural testing we can identify which part of the code our test suite do not exercise…and write new test cases using a Code Coverage tool

Code Coverage tool

Green: line *completely* covered by the test suite
Yellow: line is *partially* covered
Red: line is *not* covered



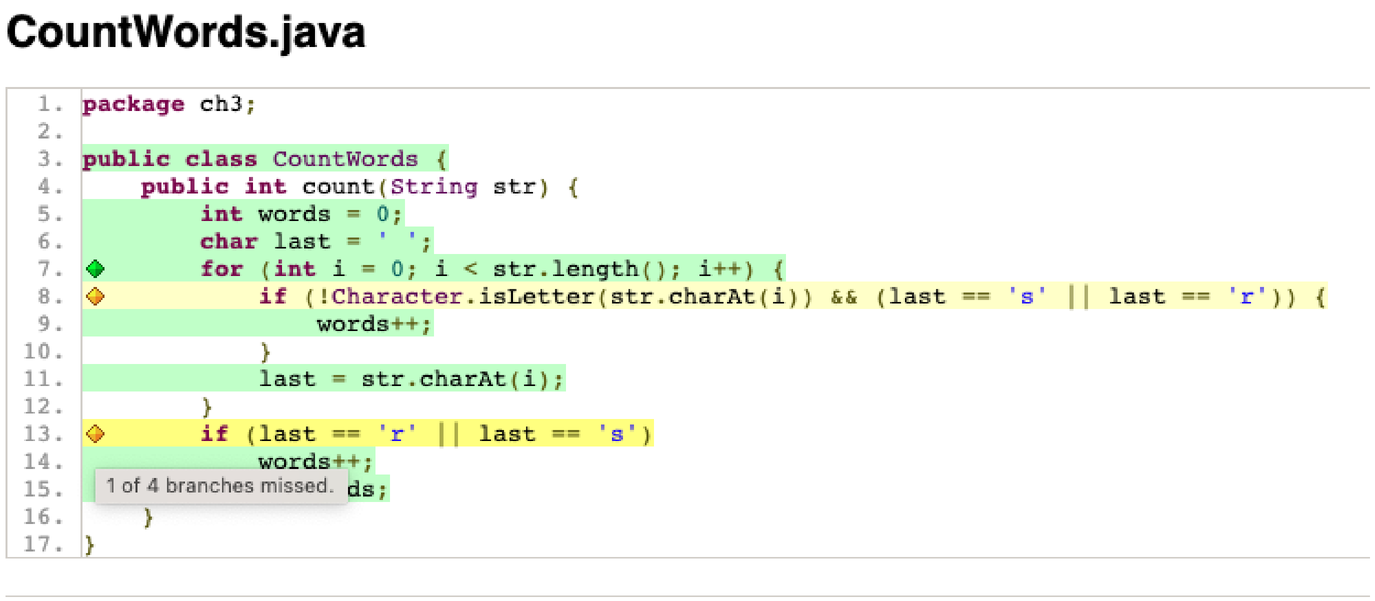
Diamonds indicate that this is a branching instruction and there may be many cases to cover.

```
public class CountWords {
    public int count(String str) {
        int words = 0;
        char last = ' ';
        for (int i = 0; i < str.length(); i++) {
            if (!Character.isLetter(str.charAt(i)) && (last == 's' || last == 'r')) {
                words++;
            }
            last = str.charAt(i);
        }
        if (last == 'r' || last == 's')
            words++;
        return words;
    }
}
```

The color indicates whether the line is covered.

Code Coverage tool

Row 8 (if): *1 of 6 branches missed*
Row 13 (if): *1 of 4 branches missed*



CountWords.java

```
 1. package ch3;
 2.
 3. public class CountWords {
 4.     public int count(String str) {
 5.         int words = 0;
 6.         char last = ' ';
 7.         for (int i = 0; i < str.length(); i++) {
 8.             if (!Character.isLetter(str.charAt(i)) && (last == 's' || last == 'r')) {
 9.                 words++;
10.             }
11.             last = str.charAt(i);
12.         }
13.         if (last == 'r' || last == 's')
14.             words++;
15.   1 of 4 branches missed. ds;
16.     }
17. }
```

Testing words that end with 'r'

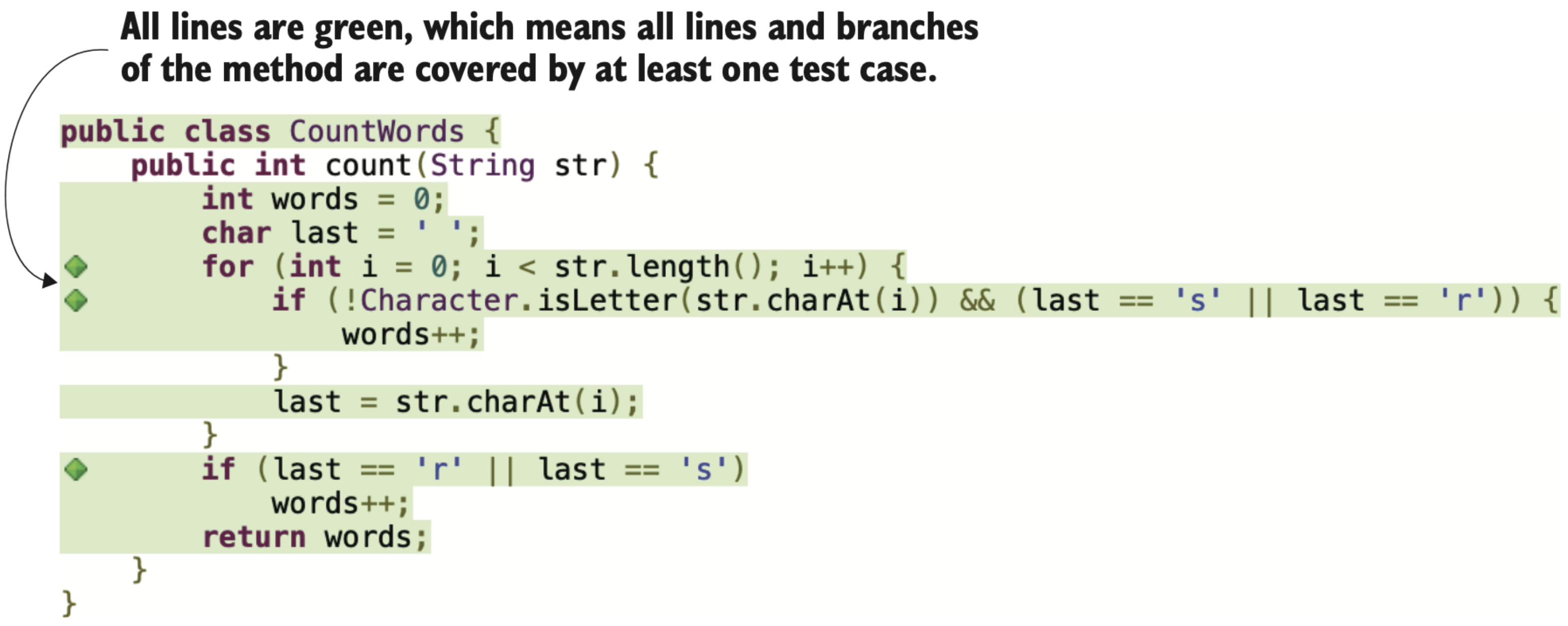
```
@Test
void wordsThatEndInR() {
  int words = new CountWords().count("car bar");
  assertThat(words).isEqualTo(2);
}
```

Words that end in "r" should be counted.

Now we can re-run the Code Coverage tool, every line should now be covered, otherwise, we will repeat the process

Code Coverage tool

The test suite now achieves full coverage of branches and conditions



All lines are green, which means all lines and branches of the method are covered by at least one test case.

```
public class CountWords {
    public int count(String str) {
        int words = 0;
        char last = ' ';
        for (int i = 0; i < str.length(); i++) {
            if (!Character.isLetter(str.charAt(i)) && (last == 's' || last == 'r')) {
                words++;
            }
            last = str.charAt(i);
        }
        if (last == 'r' || last == 's')
            words++;
        return words;
    }
}
```

Testing workflow for Structural testing

1. Perform Specification-based testing (7-steps-approach)
2. Read the implementation: understand the code (if you did not code that)
3. Run the test suite with a *code coverage* tool (to identify in an automated way parts not covered)
4. For each piece of code "not covered":
   a. Why was it not covered?
   b. Decide if that piece of code needs a test (if yes go to c.)
   c. Implement an automated test case
5. Go back to point 3

Structural testing suite complements the test suite devised via specification-based testing

Code Coverage criteria

What does exactly mean “to cover a line of code”?

```
if (!Character.isLetter(str.charAt(i)) &&
(last == 's' || last == 'r'))
```

A developer may apply different criteria:
1 – Line coverage: the line is considered as “covered” even if a single test passes through the if line (1 test case)

2 – Branch coverage: The `if` statement can be evaluated as `true` or `false` (2 test cases)

3 – Condition + branch coverage: explore each condition in the `if` statement: here we have 3 conditions requiring each 2 tests (3*2=6 tests)

4 – Path coverage: Cover every possible execution path of this statement (2^3=8 test cases)

Line Coverage

A developer achieves this if at least one test case covers the line under test

If that line contains a complex `if statement` with multiple conditions it does not matter, a single test is enough to count that line as covered.

$$\text{line coverage} = \frac{\text{lines covered}}{\text{total number of lines}} \times 100\%$$

Branch Coverage

When we have **branching instructions** (`if`s, `for`s, `while`s, etc.) that make the program behave in different ways, depending how the instruction is evaluated

$$\text{branch coverage} = \frac{\text{branches covered}}{\text{total number of branches}} \times 100\%$$

Example: `if(a && (b || c))`

How many tests do we need to achieve branch coverage?

Condition + Branch Coverage

It considers *not only possible branches* but also **each condition** of each branch statement.

The test suite should exercise:
- each of those individual conditions being evaluated to `true` and `false` at least once
- the entire branch statement being `true` and `false` at least once.

$$\text{c+b coverage} = \frac{\text{branches covered} + \text{conditions covered}}{\text{number of branches} + \text{number of conditions}} \times 100\%$$

Condition + Branch Coverage

It considers *not only possible branches* but also **each condition** of each branch statement.

The test suite should exercise:

- each of those individual conditions being evaluated to `true` and `false` at least once
- the entire branch statement being `true` and `false` at least once.

Example: `if(A || B)`

```
T1 : A= true, B = false
T2 : A = false, B = true
```

Are these two tests enough?

Condition + Branch Coverage: examples

Example: `if(A || B)`

```
T1 : A= true, B = true
T2 : A = false, B = true
```

Compute the value of c+b coverage (2 mins)

$$\text{c+b coverage} = \frac{\text{branches covered} + \text{conditions covered}}{\text{number of branches} + \text{number of conditions}} \times 100\%$$

Condition + Branch Coverage: examples

Example: `if (A || B)`

```
T1 : A= true, B = true
T2 : A = false, B = true
```

Compute the value of c+b coverage (2 mins)

Branch = 1/2
Condition = 3/4

C+b coverage= (1+3)/(2+4) * 100= 66,6%

$$\text{c+b coverage} = \frac{\text{branches covered} + \text{conditions covered}}{\text{number of branches} + \text{number of conditions}} \times 100\%$$

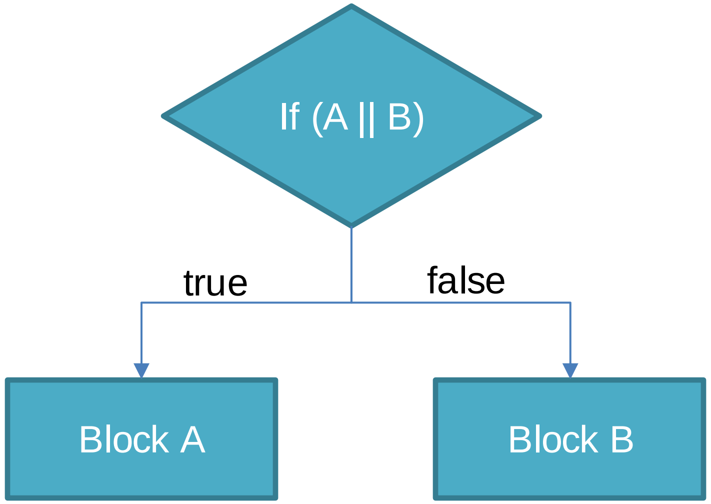
Condition + Branch Coverage: examples

Example:

```
      if(A || B)
              {Block A}
      else
              {Block B}
```

```
T1 : A= true, B = false
```

Compute the value of line, branch and c+b coverage (3 mins)



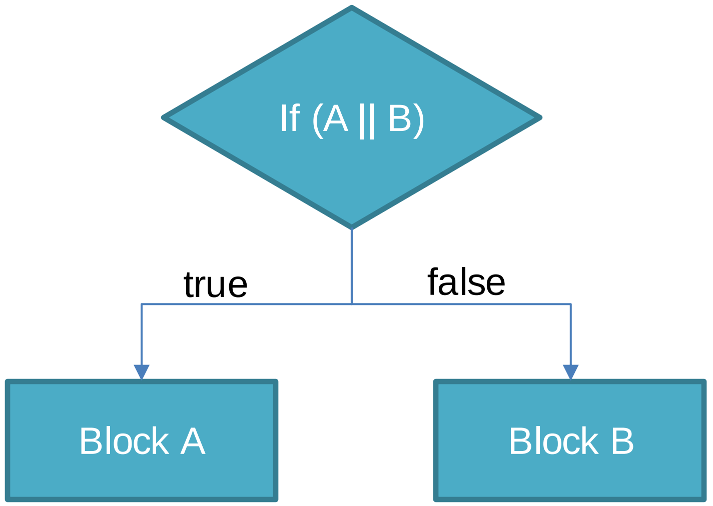
Condition + Branch Coverage: examples

Example:

```
        if(A || B)
                {Block A}
        else
                {Block B}
```

```
T1 : A= true, B = false
```

Compute the value of line, branch and c+b coverage (3 mins)



Line = 2/3 = 66,6%
Branch = 1/2 = 50%
Condition= 2/4

C+b coverage= (1+2)/(2+4) * 100= 50%

Path Coverage

When you cover all possible paths of execution of the program.

This is the **strongest** criterion, but often **impossible** or too **expensive** to achieve

In a program with `n` conditions, where each one could be evaluated `true` or `false`, we have 2^n paths to cover

Example: If a program has 10 conditions, the total number of combinations would be $2^{10} = 1024$. This means more than a thousand tests!

What Coverage criteria to choose

It depends…

Trade-off between:

- maximize the number of bugs found
- minimizing the effort/cost of building the test suite

Is there a good comprise between path coverage (too expensive) and condition+branch coverage?

MC/DC coverage criterion

Modified Condition/Decision Coverage (MC/DC) is a good answer.

The MC/DC criterion looks at combinations of conditions, as path coverage does.

MC/DC instead of testing *all* possible combinations identify the *important* combinations that need to be tested.

If conditions have only **binary** outcomes (that is, *true* or *false*), the number of tests required to achieve 100% MC/DC coverage is N + 1, where N is the number of conditions in the decision.

Note that $(N+1) << 2^N$

MC/DC coverage criterion: an example

Conditions a, b, c are evaluated to Boolean values:

```
if(a && (b || c))
```

If we apply the MC/DC criterion only 3+1 = 4 tests cases will be enough, instead of 8 (path coverage criterion)

For each condition (e.g. ‘a’) we should look for:
- A case where the condition ‘a’ is `true` (T1)
- A case where the condition ‘a’ is `false` (T2)
- T1 and T2 must have different outcome (one true and one false)
- Variable ’b’ and ‘c’ in T1 must have the same value in T2

T1 and T2 are called independence pairs, because the variable ‘a’ independently influences the outcome (decision).

MC/DC coverage criterion: an example

For each condition (e.g. ‘a’) we should look for:
- A case where the condition ‘a’ is `true` (T1)
- A case where the condition ‘a’ is `false` (T2)
- T1 and T2 must have different outcome (one true and one false)
- Variable ’b’ and ‘c’ in T1 must have the same value in T2

Test case	`isLetter`	`last == s`	`last == r`	`decision`
T1	true	true	true	true
T2	true	true	false	true
T3	true	false	true	true
T4	true	false	false	false
T5	false	true	true	false
T6	false	true	false	false
[illegible]	[illegible]	[illegible]	true	false
[illegible]	[illegible]	[illegible]	false	false

T1 and T5 are a pair of test (an independence pair) where `isLetter` is the only parameter that is different and the outcome (decision) changes.

MC/DC coverage criterion: an example

For each condition (e.g. ‘a’) we should look for:
- A case where the condition ‘a’ is `true` (T1)
- A case where the condition ‘a’ is `false` (T2)
- T1 and T2 must have different outcome (one true and one false)
- Variable ’b’ and ‘c’ in T1 must have the same value in T2

Test case	isLetter	last == s	last == r	decision
T1	true	true	true	true
T2	true	true	false	true
T3	true	false	true	true
T4	true	false	false	false
T5	false	true	true	false
T6	false	true	false	false
T7	false	false	true	false
T8	false	false	false	false

MC/DC coverage criterion: an example

We found the following pairs:

- `isLetter`: {1, 5}, {2, 6}, {3, 7}
- `last` == s: {2,4}
- `last` == r: {3,4}

What pair to choose for each variable?

For the last two conditions is straightforward because we have just one pair, so we surely need tests: T2, T3, T4

What pair to pick for `isLetter`?

MC/DC coverage criterion: an example

We found the following pairs:

- `isLetter`: {1, 5}, {2, 6}, {3, 7}
- `last` == s: {2,4}
- `last` == r: {3,4}

For the last two conditions is straightforward because we have just one pair, so we surely need tests: T2, T3, T4

What pair to pick for `isLetter`?

If we pick {1, 5} we will have at the end 5 tests (T1, T2, T3, T4, T5)!

While with {2, 6}, {3, 7} makes no difference, in both cases we have only 4 tests (as expected) which is better than 8 (path coverage)!:

- {2, 6} -> T2, T3, T4, T6
- {3, 7} -> T2, T3, T4, T7

MC/DC coverage criterion: exercises

Practice yourself with MC/DC!

Exercise 1 (10 mins):

- Consider the expression (A & B) | C
  What the test suite to achieve 100% MC/DC coverage?

Test case	A	B	C	(A & B) | C
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	T
8	F	F	F	F

MC/DC coverage criterion: exercises

Solution of Exercise 1:

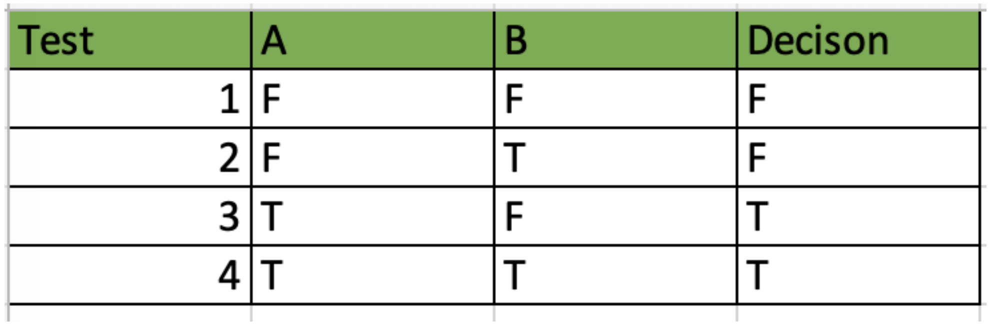
- Consider the expression (A & B) | C
  What the test suite to achieve 100% MC/DC coverage?

{2, 3, 4, 6}
{2, 4, 5, 6}

MC/DC coverage criterion: exercises

Exercise 2 (10 mins):

- Consider the expression A && (A || B)
  What the test suite to achieve 100% MC/DC coverage?



Test	A	B	Decison
1	F	F	F
2	F	T	F
3	T	F	T
4	T	T	T

MC/DC coverage criterion: exercises

Solution of Exercise 2:

- Consider the expression A && (A || B)
  What the test suite to achieve 100% MC/DC coverage?

A: {(1, 3), (2, 4)}
B: { (empty) }

1. There is no independence pair for B. Thus, it is not possible to achieve MC/DC coverage for this expression.
2. Since there is no independence pair for B, this parameter does not affect the result. You should recommend that the developer restructure the expression without using B, making the code easier to maintain.
3. This example shows that software testers can contribute to code quality not only by spotting bugs but also by suggesting changes that result in better maintainability.

MC/DC coverage criterion: exercises

Question 3:

- Consider the expression `(A && B)||( A && C)`

What do you notice?

MC/DC coverage criterion: exercises

Question 3:

- Consider the expression `(A && B)||( A && C)`
What do you notice?

It is impossible to change the first A without changing the second A!
In such cases, we allow A to vary, but we fix all other variables (this is called masked MC/DC).

MC/DC coverage criterion: exercises

Question 4:

- Consider the expression `(A && B)||( A && not B)`

What do you notice?

MC/DC coverage criterion: exercises

Question 4:

- Consider the expression `(A && B)||( A && not B)`
What do you notice?

It is impossible to achieve MC/DC in such expression
There are no pairs that show the independence of B
Revisit the expression to simply A

Loops boundary adequacy criterion

Given that exhaustive testing is impossible, how to handle:

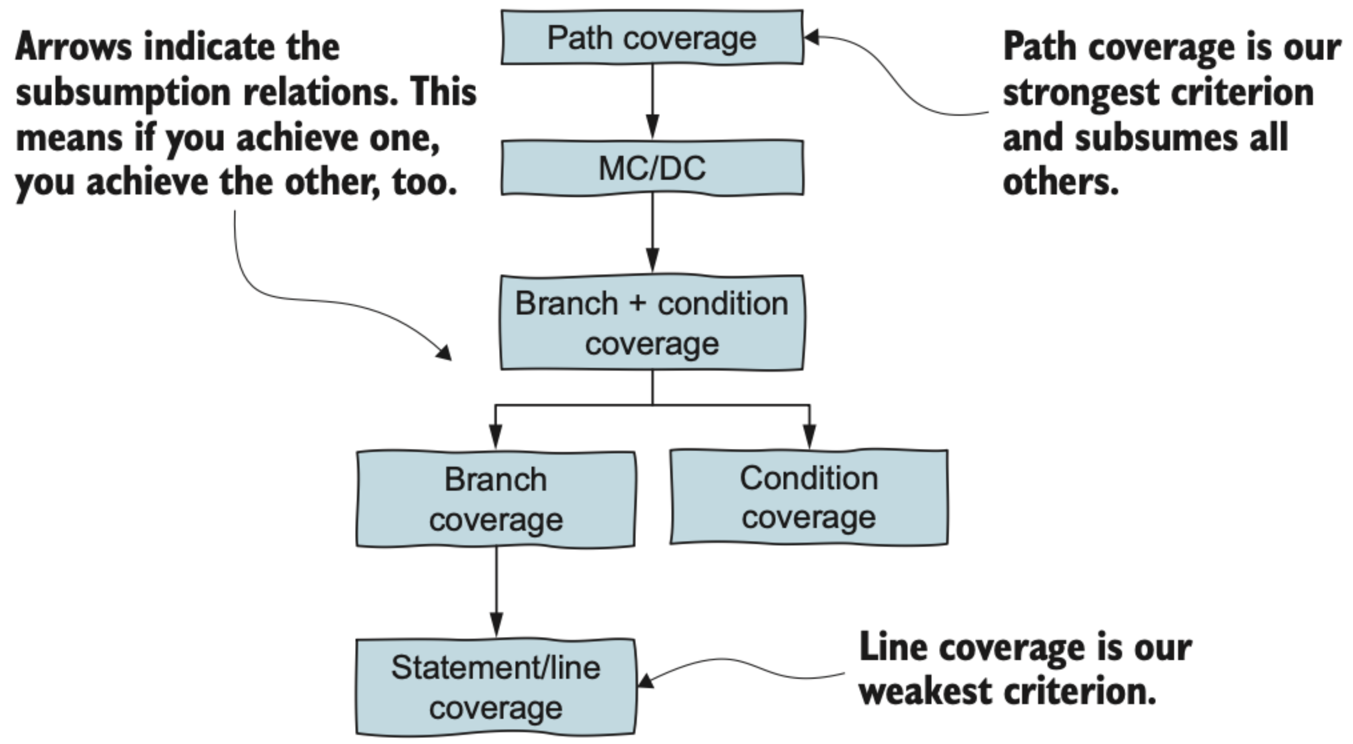
- A long-lasting loop (that runs for many iterations)
- An unbounded loop (that is executed an unknown number of times)

Loop boundary adequacy criterion:
a test suite satisfies this criterion if and only if for every loop

- There is a test case that exercises the loop zero times.
- There is a test case that exercises the loop once.
- There is a test case that exercises the loop multiple times.

NB: You can create also more than one test for the "multiple time" case.

Criteria subsumption


Arrows indicate the subsumption relations. This means if you achieve one, you achieve the other, too.
Path coverage
MC/DC
Branch + condition coverage
Branch coverage
Condition coverage
Statement/line coverage
Path coverage is our strongest criterion and subsumes all others.
Line coverage is our weakest criterion.

Weaker criterion: cheaper & faster, leave the code uncovered

Stronger criterion: higher cost, cover the code better

Branch vs Condition coverage

Example:

```
If(A||B)
```

T1= {true, false}
T2= {false, true}

What criterion are we satisfying?

Branch vs Condition coverage

Example:

```
If(A||B)
```

T1= {true, false}
T2= {false, true}

What criterion are we satisfying?

Condition, but not branch coverage!

Code Coverage (CC): summing up

- Not look at the coverage number blindly (do not take 100% as a goal, but try to understand the numbers)
- Not gaming the metric
- Structural testing & CC augment/complement specification-based testing:
  - Identify part of the code not covered yet by the test suite
  - Identify missed partitions
- Sometimes it is ok to leave some part of the code uncovered
- CC is not a *quality target* as it does not indicate a good test suite (100% coverage is not an indicator of how good a test suite is)
- If you have *low coverage* your system is NOT properly tested (but the vice versa is not true)

Code Coverage criteria

Rule of thumb:

- Start with branch coverage;
- If there are more complex expressions: evaluate if b+c or MC/DC is needed.

If expressions are too complicated, try to break them in smaller bits.

Specification based + structural testing: an example

REQS:

Left-pad a string with a specified string. Pad to a size of size.

- `str`—The string to pad out; may be null.
- `size`—The size to pad to.
- `padStr`—The string to pad with. Null or empty is treated as a single space.

The method returns a **left-padded string**, the original string if no padding is necessary, or null if a null string is input.

EX.

- `str= 'abc'`
- `size = 5`
- `padStr = 'X'`

RESULT: 'XXabc'

LeftPad(): implementation

```
public static String leftPad(final String str, final int size,
  String padStr) {

  if (str == null) {
    return null;
  }

  if (padStr==null || padStr.isEmpty()) {
    padStr = SPACE;
  }

  final int padLen = padStr.length();
  final int strLen = str.length();
  final int pads = size - strLen;

  if (pads <= 0) {
    // returns original String when possible
    return str;
  }

  if (pads == padLen) {
    return padStr.concat(str);
  } else if (pads < padLen) {
    return padStr.substring(0, pads).concat(str);
```

If the string to pad is null, we return null right away.

There is no need to pad this string.

If the number of characters to pad matches the size of the pad string, we concatenate it.

If we cannot fit the entire pad string, we add only the part that fits.

Ex:

```
str= 'abc'
size = 2
padStr = 'X'
pads = 2-3<= 0
RESULT = 'abc'
```

LeftPad(): implementation

```
public static String leftPad(final String str, final int size,
  String padStr) {

  if (str == null) {
    return null;
  }

  if (padStr==null || padStr.isEmpty(
    padStr = SPACE;
  }

  final int padLen = padStr.length();
  final int strLen = str.length();
  final int pads = size - strLen;

  if (pads <= 0) {
    // returns original String when po
    return str;
  }

  if (pads == padLen) {
    return padStr.concat(str);
  } else if (pads < padLen) {
    return padStr.substring(0, pads).concat(str);
```

If the string to pad is null, we return null right away.

Ex:

```
str = 'abc'
size = 4
padStr = 'X'
pads = 4-3 = 1
padLen = 1
RESULT = 'Xabc'
```

If the number of characters to pad matches the size of the pad string, we concatenate it.

If we cannot fit the entire pad string, we add only the part that fits.

LeftPad(): implementation

```
public static String leftPad(final String str, final int size,
  String padStr) {

  if (str == null) {
    return null;
  }

  if (padStr==null || padStr.isEmpty()) {
    padStr = SPACE;
  }

  final int padLen = padStr.length();
  final int strLen = str.length();
  final int pads = size - strLen;

  if (pads <= 0) {
    // returns original String when p
    return str;
  }

  if (pads == padLen) {
    return padStr.concat(str);
  } else if (pads < padLen) {
    return padStr.substring(0, pads).concat(str);
```

If the string to pad is null, we return null right away.

If the pad string is null or empty, we make it a space.

Ex:

```
str = 'abc'
size = 4
padStr = 'XXX'
pads = 4-3 = 1
padLen = 3
RESULT = 'Xabc'
```

rs to
he
te it.

If we cannot fit the entire pad string, we add only the part that fits.

LeftPad(): implementation

```
    } else {
      final char[] padding = new char[pads];
      final char[] padChars = padStr.toCharArray();

      for (int i = 0; i < pads; i++) {
        padding[i] = padChars[i % padLen];
      }

      return new String(padding).concat(str);
    }
  }
```

We have to add the pad string more than once. We go character by character until the string is fully padded.

Ex: pads>padLen

```
str= 'abc'
size = 7
padStr = 'XY'
pads = 7-3 = 4
padLen = 2
RESULT = 'XYXYabc'
```

Specification based + structural testing: an example

REQS:

Left-pad a string with a specified string. Pad to a size of size.

- `str`—The string to pad out; may be null.
- `size`—The size to pad to.
- `padStr`—The string to pad with. Null or empty is treated as a single space.

The method returns a left-padded string, the original string if no padding is necessary, or null if a null string is input.

For this example, write specification + structural tests

Let’s start with Specification-based testing

leftPad()

Three inputs:

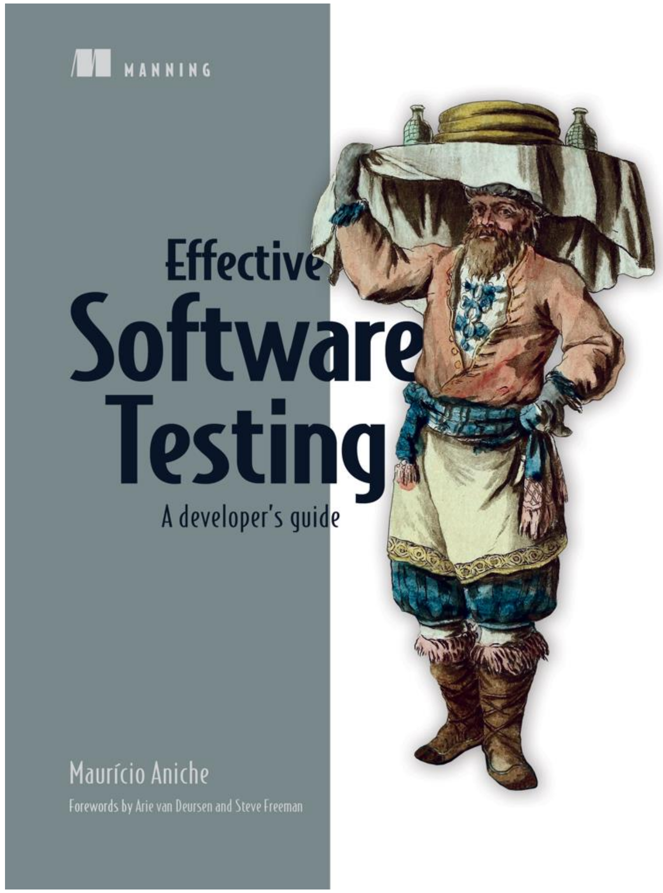
```
str= ‘abc’
size = 5
padStr = ‘X’
```

One output: `String`

Identify the partitions and the test suite (20 mins)

Reference book:

Effective Software Testing. A developer’s guide. Mauricio Aniche. Ed. Manning. (**Chapter 3**)


MANNING
Effective Software Testing
A developer's guide
Maurício Aniche
Forewords by Arie van Deursen and Steve Freeman

References

- JetBrains IntelliJ IDEA Code Coverage documentation: https://www.jetbrains.com/help/idea/code-coverage.html

SERLAB
Software Engineering Research

Azzurra Ragone

Department of Computer Science- Floor VI – Room 616
Email: azzurra.ragone@uniba.it