UNIVERSITÀ
DEGLI STUDI DI BARI
ALDO MORO

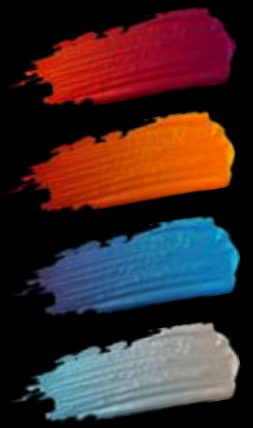
SERLAB
Software Engineering Research

Integrazione e Test di Sistemi Software

Structural testing and Code Coverage

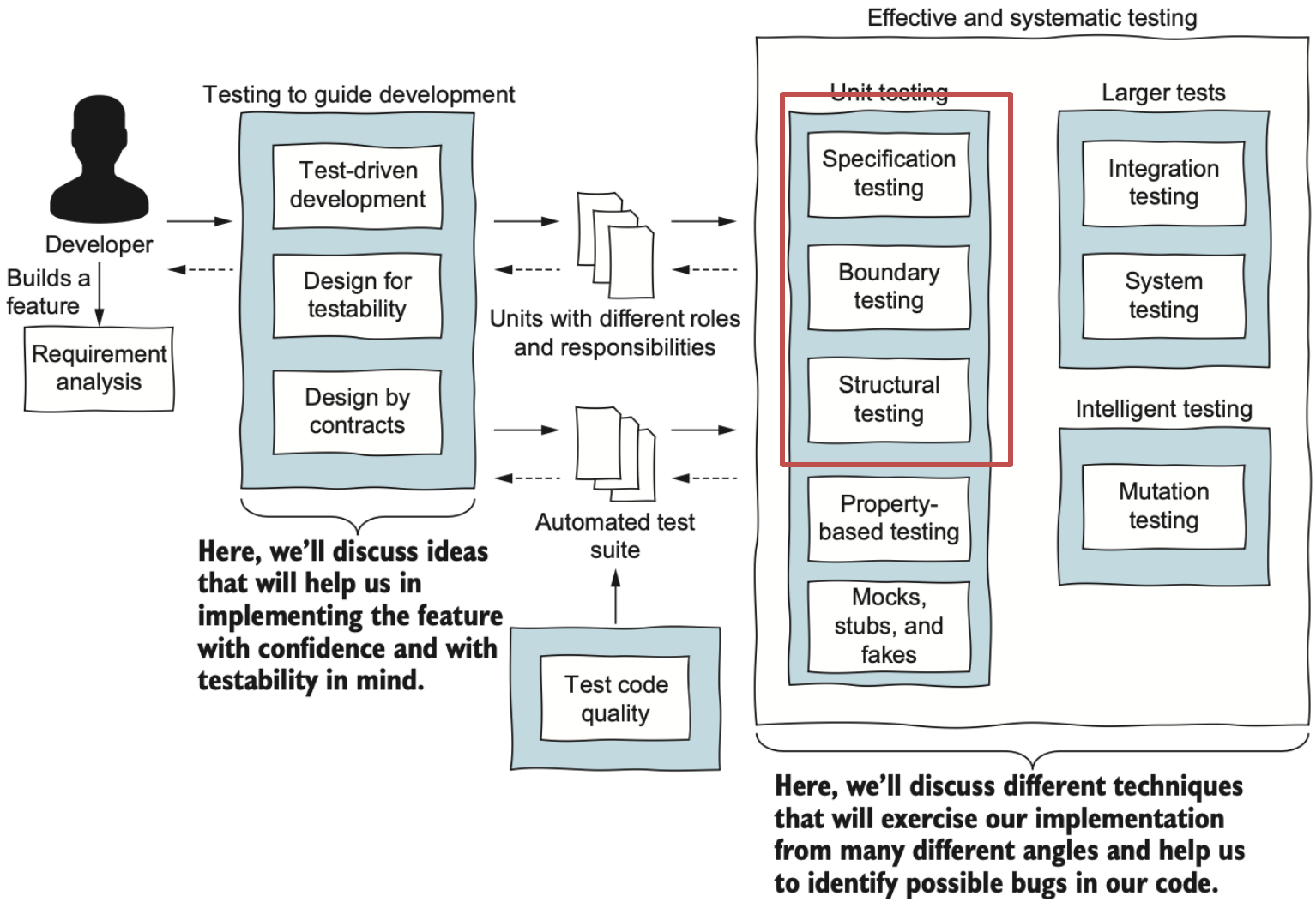
Azzurra Ragone

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



What is Structural Testing?

Effective and systematic testing (workflow)



Structural 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);
}

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

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

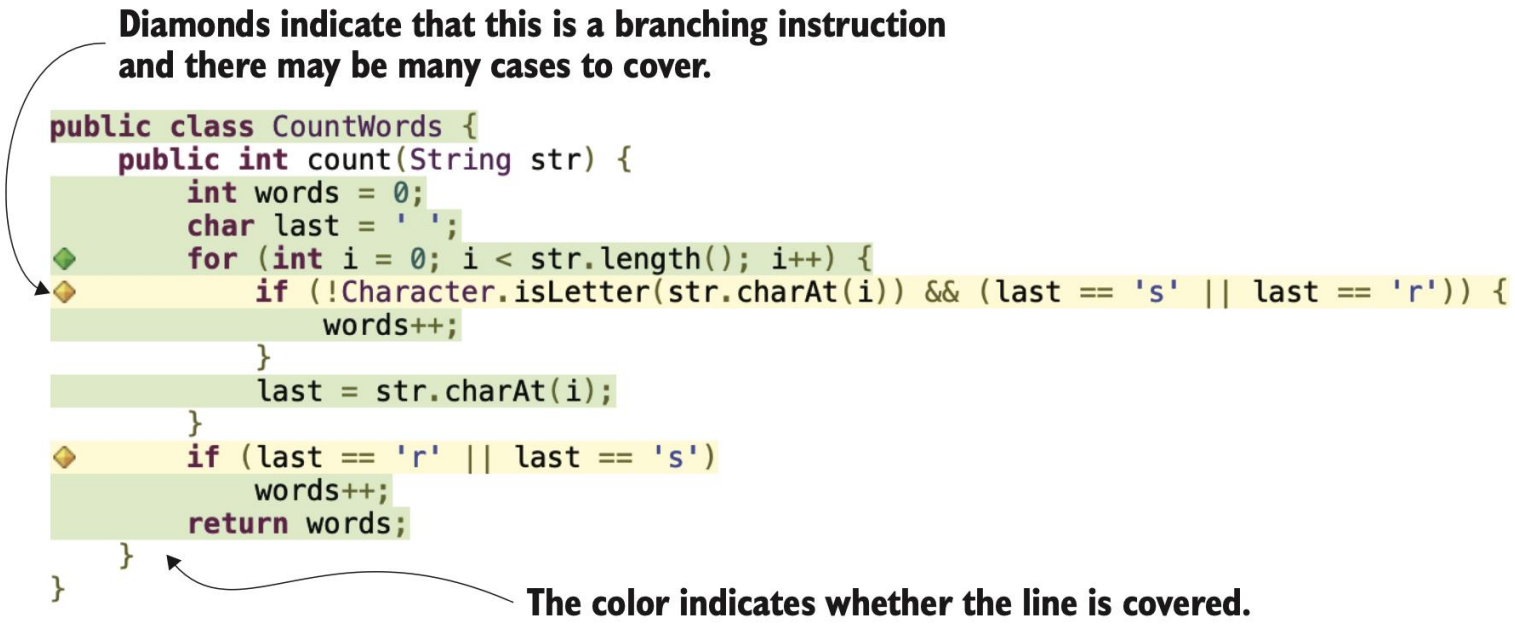
No words ending in "s" or "r" in the string: the program returns 0.

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



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.         return words;
16.     }
17. }
```

1 of 4 branches missed.

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%

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

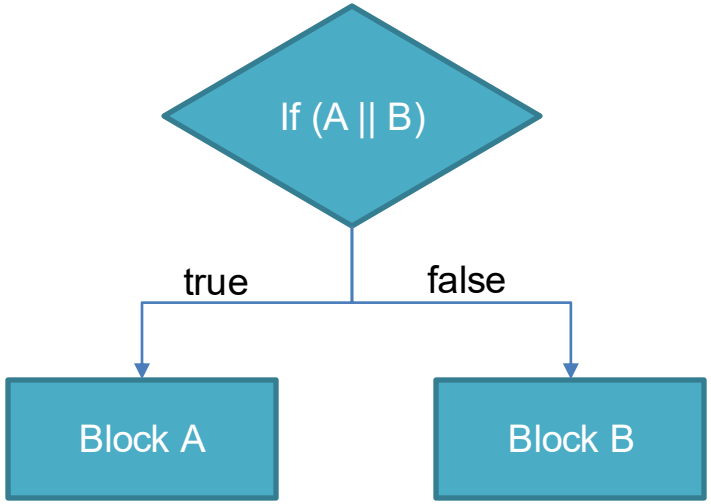
Condition + Branch Coverage: examples

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)



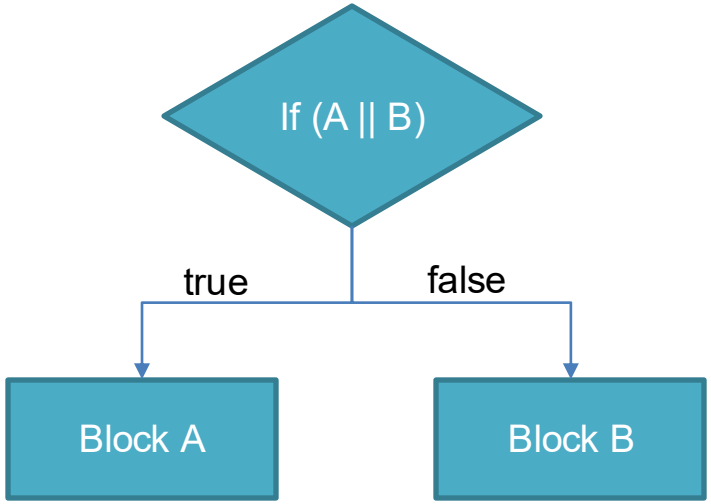
Condition + Branch Coverage: examples

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.

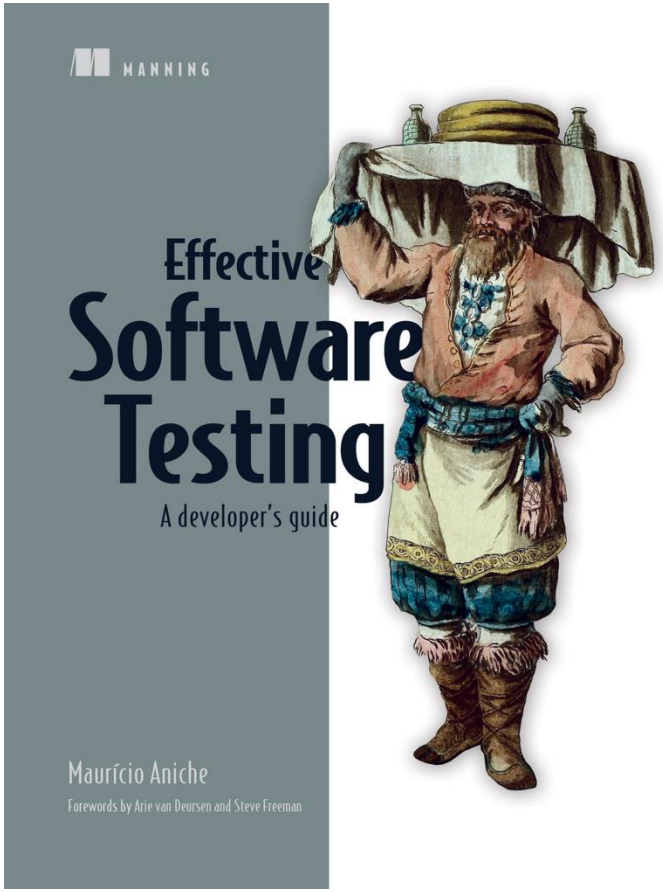
MC/DC instead of testing *all* possible combinations, identifies the *important* combinations that need to be tested.

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

Reference book:

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



References

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

Azzurra Ragone

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