

More Unit Testing

JaCoCo

- JaCoco is a coverage library for unit testing

JaCoCo

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
 org.jacoco.core		97%		89%	201	1,689	140	3,872	19	786	2	160
 org.jacoco.examples		57%		64%	24	53	98	194	19	38	6	12
 org.jacoco.agent.rt		75%		82%	33	134	80	357	21	83	7	23
 jacoco-maven-plugin		90%		82%	35	193	49	465	8	116	1	23
 org.jacoco.cli		97%		100%	4	109	10	275	4	74	0	20
 org.jacoco.report		99%		99%	4	572	2	1,345	1	371	0	64
 org.jacoco.ant		98%		99%	4	162	8	428	3	110	0	19
 org.jacoco.agent		86%		75%	2	10	3	27	0	6	0	1
Total	1,498 of 30,242	95%	242 of 2,612	90%	307	2,922	390	6,963	75	1,584	16	322

Image: <https://www.jacoco.org/jacoco/trunk/coverage/>

JaCoCo metrics

- Instructions: number of bytecode instructions covered (green) and missed (red)
- Branches: number of branches (if, switch, and loops) covered (green) and missed (red)
- Cyclomatic complexity: number of linearly independent paths through a program's source code, equal to $D + 1$, where D is the number of binary decision (results show missed and total)
- Lines, methods and classes: Missed and total
- Coverage: $\# \text{ covered} / \text{total} \times 100\%$

Cyclomatic complexity

Complexity = $D + 1$, where D = number of binary decisions

- minimum number of tests for basis path coverage (will represent all paths)
- maximum number of tests for full *branch coverage* (decision point outcome is tested at least once)

```
// Complexity 1: straight-line code, a single path.  int add(int a, int b) {  
    return a + b;  
}
```

Cyclomatic complexity

Complexity = $D + 1$, where D = number of binary decisions

- minimum number of tests for basis path coverage (will represent all paths)
- maximum number of tests for full *branch coverage* (decision point outcome is tested at least once)

// Complexity 4: three binary decisions

// (age < 18, age > 65, member).

```
int classify(int age, boolean member) {  
    if (age < 18) { // +1  
        return 0;  
    } else if (age > 65 && member) {  
        return 2; // else if: +1 for age > 65, +1 for member  
    }  
    return 1;  
}
```

Parameterized tests

- A parameterized test allows you to run test methods multiple times with different arguments.
- You must declare the test method as a `@ParameterizedTest` and specify the source values as
 - `@ValueSource(type = {value1, value2, ...})`
 - `@CsvSource({"a1, b1, ...", "a2, b2, ...", ...})`

For more details see:

<https://docs.junit.org/6.1.3/writing-tests/parameterized-classes-and-tests.html>