Code coverage

Note to DevoXX UK reviewers:
this is a very rough prototype based on an 20 min internal presentation I prepared for my team.

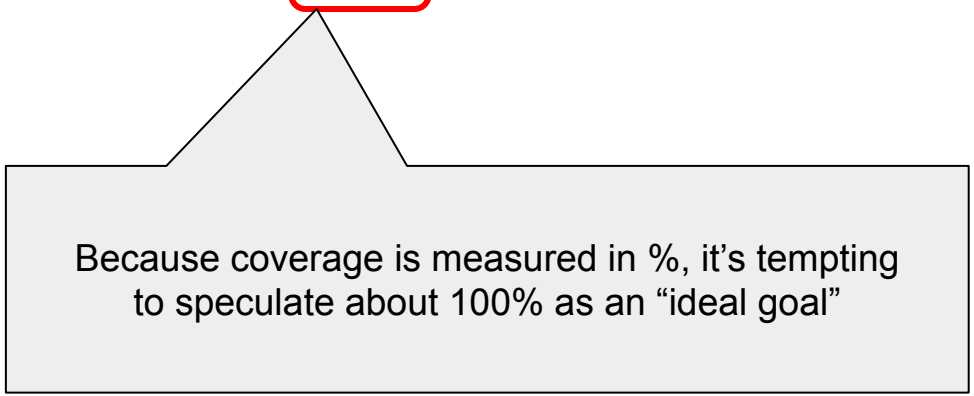
The content, structure and visual representation are going to change, also I want to make it more universal / less JVM/JaCoCo specific

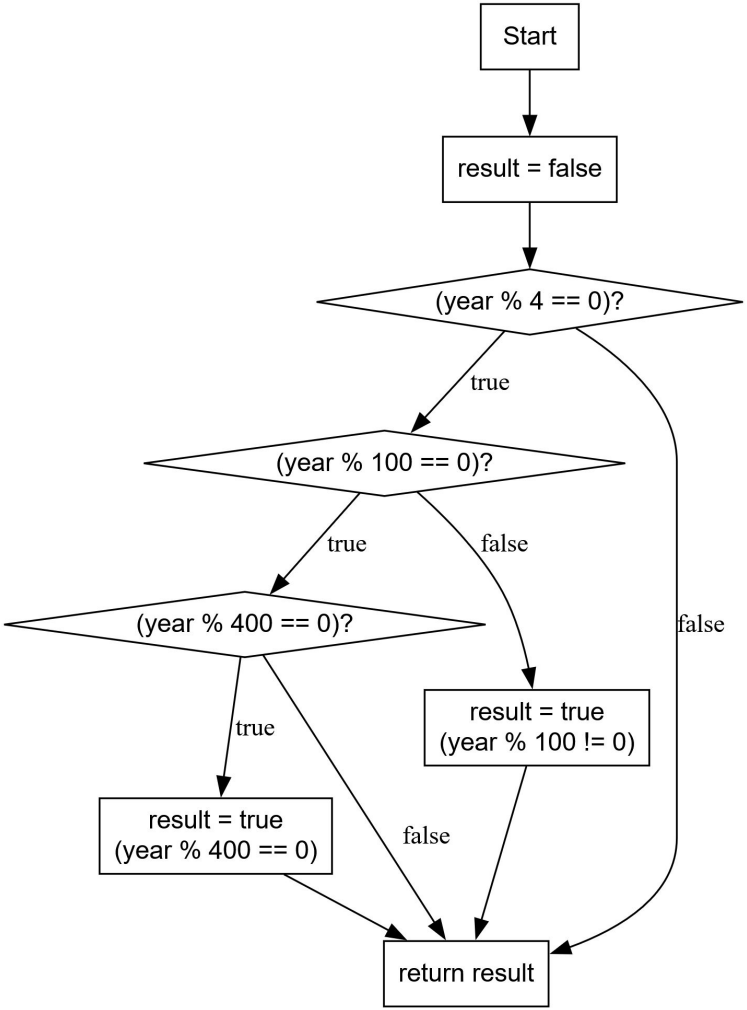
The magical “coverage” metric

- “What’s the test/code coverage in your project?”
- “We have very high, 80%..90% test coverage”
- “We need to improve coverage!”
- “Can you give us 100% test coverage?”

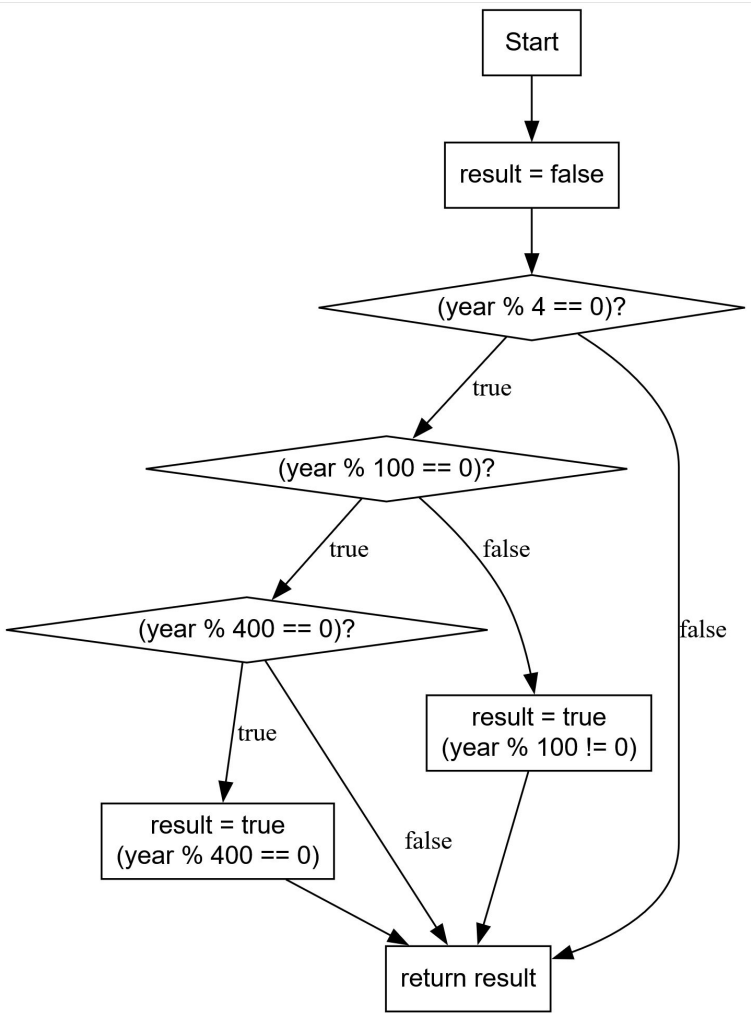
The magical metric

- “What’s the test/code coverage in your project?”
- “We have very high, 80%..90% test coverage”
- “We need to improve coverage!”
- “Can you give us 100% test coverage?”





boolean isLeapYear(int year)



Number of branches – number of execution paths needed to cover all the edges (4 in this case)

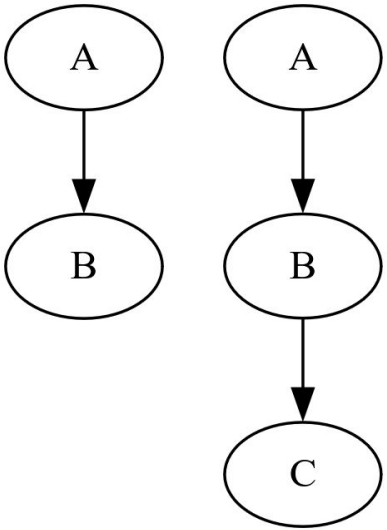
Number of paths – number of all possible execution paths (4 in this case)

Cyclomatic complexity –

CC = E - N + 2 = 10 - 8 + 2 = 4

Is it a coincidence? (unfortunately, **yes,** it is just a coincidence, in general it's more complicated)

Adding extra statements to a branch does not contribute to CC…



For a chain of statements without any decision points, CC = 1

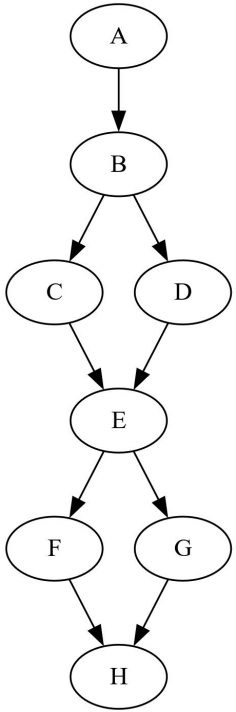
E - N + 2 =

= 1 - 2 + 2 =
= 2 - 3 + 2 =
= 3 - 4 + 2 = 1

(we add +1 edge and +1 node)

no matter how long is the chain!!

…Adding “decision points” do contribute



`if`, `while`, `for`, `switch case`, also all the boolean expressions with short-cutting (`&&`, `||`) will add +1 to CC

CC = E-N+2=

= 9 - 8 + 2 =3

Mathematical facts: CC is

- **the upper bound** for number of tests for full “**branch coverage**” (2 in this case)
- **the lower bound** for number of tests for full “**path coverage**” (4 in this case)

Cyclomatic Complexity metric

- Introduced by Thomas J. McCabe in 1976.
- **It’s very easy to calculate** for every programming language (only very basic lexical/syntax analysis is needed). On the other hand, estimation of actual possible paths of execution involves analysis of “possible” and “impossible” paths.
- It’s additive (we can calculate CC for a whole project).
- It’s **“roughly the number of unit tests needed”**

What is an acceptable value for CC?

- MCabe’s categorization of CC of a single procedure (1976 paper->wikipedia):
  - 1 - 10: Simple procedure, little risk
  - 11 - 20: More complex, moderate risk
  - 21 - 50: Complex, high risk
  - > 50: Untestable code, very high risk

What is an acceptable value for CC?

- In my personal view, everything above **15** is a mess.
- Java Checkstyle default limit is **10**,
- Kotlin's detekt **15**.
- In TDK project, we reconfigured detekt to use **40** as a limit for CC (which is too high IMO).

    - Metrics & static analysis is a topic for a separate presentation

How do we measure coverage in JVM?

- **JaCoCo** (Evgeny Mandrikov)
- Works on bytecode level, injects instrumentation into every “decision point” of the code, setting bits in a huge bitmap of boolean values when a decision point is passed by
- NB: no need to instrument every instruction / statement, just “decision points” (JaCoCo calculates CC for each method as a side-effect of its work).
- NB: even without any source code, we can calculate bytecode instructions coverage & branch coverageNB: JaCoCo is a JVM “agent” (“plugin”) which can be run with every execution, so manual testers also can use it (e. g. one clicks on the button and it doesn’t work correctly: with JaCoCo we can figure out the code).

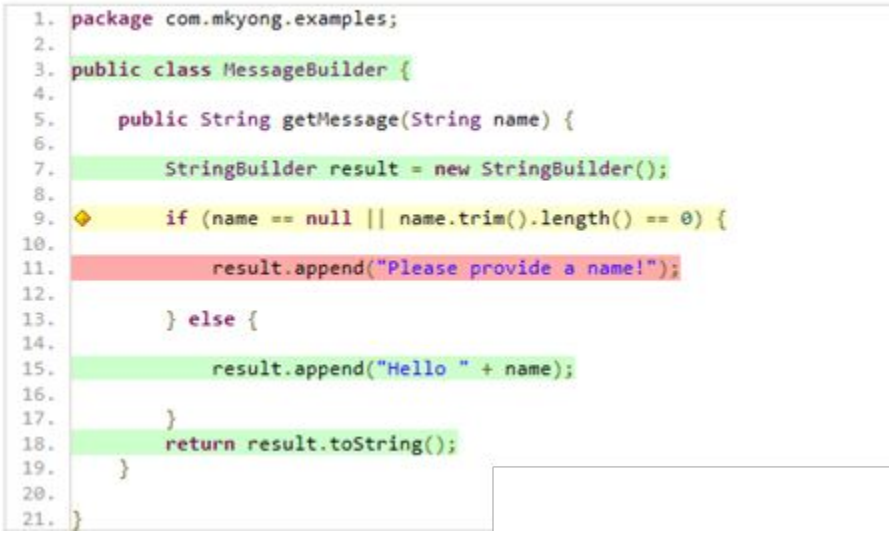
JaCoCo coverage report examples

JaCoCo

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.jacoco.examples		58%		64%	24	53	97	193	19	38	6	12
org.jacoco.core		97%		92%	137	1,507	125	3,555	19	740	2	147
org.jacoco.agent.rt		75%		83%	32	130	75	344	21	80	7	22
jacoco-maven-plugin		90%		82%	35	194	49	466	8	117	0	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	163	8	429	3	111	0	19
org.jacoco.agent		86%		75%	2	10	3	27	0	6	0	1
Total	1,429 of 28,544	94%	177 of 2,338	92%	242	2,738	369	6,634	75	1,537	15	308

JaCoCo report

MessageBuilder.java



```
1.  package com.mkyong.examples;
2.
3.  public class MessageBuilder {
4.
5.      public String getMessage(String name) {
6.
7.          StringBuilder result = new StringBuilder();
8.
9.          if (name == null || name.trim().length() == 0) {
10.
11.             result.append("Please provide a name!");
12.
13.         } else {
14.
15.             result.append("Hello " + name);
16.
17.         }
18.         return result.toString();
19.     }
20.
21. }
```

- 6 tracked lines of code
- 4 “fully covered”
- 1 “partially covered”
- 1 “not covered”

Oh, btw! what happens on line 3?

100%, but not 100% coverage

```
1. package org.example;
2.
3. class Example1 {
4.
5.   static int sum(int a, int b) {
6.     return a + b;
7.   }
8.
9. }
```

Example from E. Mandrikov's talk: in order to get 100% coverage, one should add a useless test.

Let’s calculate percentages on lines of code!

```
if (x > 0){
        blah.blah();
        blah.blah.blah();
} else {
        blah.blah();
        blah.blah.blah();
}
```

1. 5 tracked lines
2. 2 covered
3. 1 partially covered
4. 2 uncovered

LOC coverage: 60% or 40% (codecov report which is used by TDK tells its 40%).

Let’s calculate percentages! LOC coverage

```
if (x > 0){
        blah.blah();
        blah.blah
                .blah();
} else {
        blah.blah();
        blah.blah.blah();
}
```

1. 6 tracked lines
2. 3 covered
3. 1 partially covered
4. 2 uncovered

LOC coverage **increased**:
83% or 50% !

(the reason why JaCoCo measures bytecode coverage)

Let’s calculate percentages! LOC coverage

```
if (x > 0){
        blah.blah();
        blah.blah.blah();
        blah.blah();
        blah();
        blah();
        blah();
        blah.blah();
} else {
        blah.blah();
        blah.blah.blah();
}
```

1. 10 tracked lines
2. 7 covered
3. 1 partially covered
4. 2 uncovered

LOC coverage: 80% or 70% !

(though in fact, if it’s the same test, we’re at a higher risk)

Branch coverage

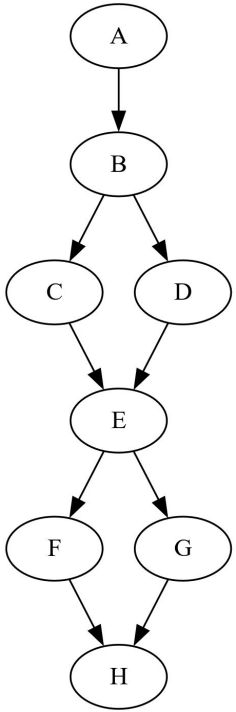
```
if (x > 0){
        blah.blah();
        blah.blah.blah();
        blah.blah();
        blah();
        blah();
        blah();
        blah.blah();
} else {
        blah.blah();
        blah.blah.blah();
}
```

Regardless the amount of code in each branch, branch coverage for this example is 50%.

JaCoCo calculates branch coverage, but the numbers are usually disappointing, people don’t wanna look at them…

CodeCov for Kotlin doesn’t show branch coverage.

Even branch coverage does not give us the full picture



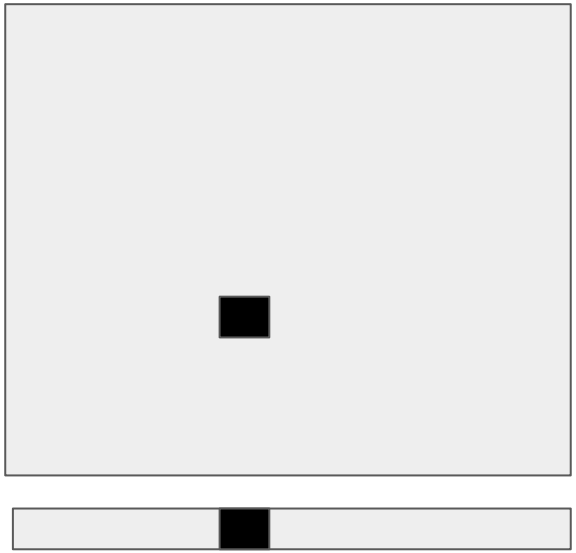
Branch coverage can be 100%, but execution paths coverage still may be less than 100%.

JaCoCo does not analyze that.

Intermediate conclusions

- Coverage reports are very informative. They help to find bugs, unused code, and missing tests
- Coverage metrics in % (especially over the whole project) is less informative when coverage is already good.

Overall code coverage is just a uni-dimensional projection of multi-dimensional picture

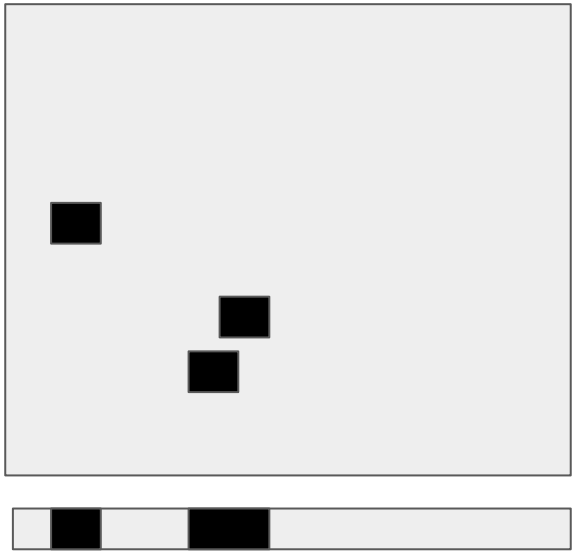


Full space of program execution paths

LOC coverage

Low code coverage:

increase of code coverage reflects the increase of tests quality



Full space of program execution paths

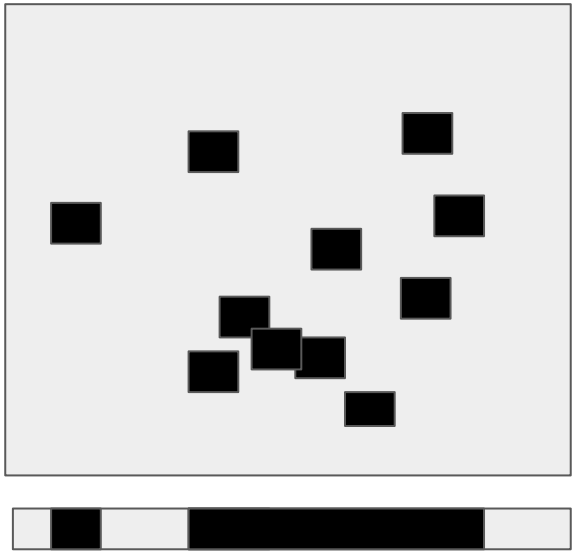
LOC coverage

Higher code coverage:
percentages are high, but vast areas are untested

Full space of program execution paths

LOC coverage

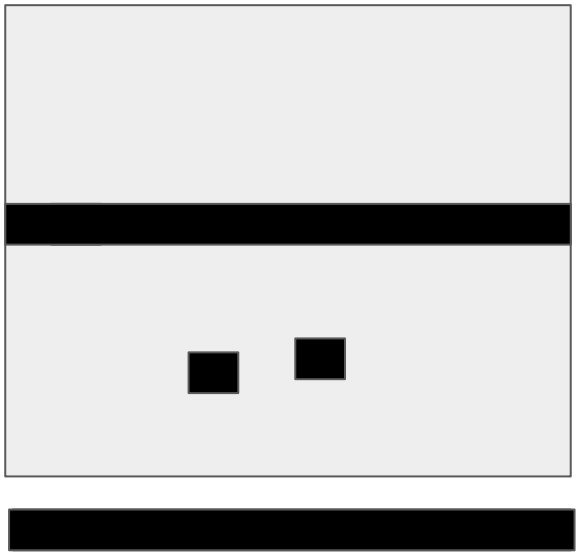
When code coverage is high, adding new valuable tests does not contribute to the metric



Full space of program execution paths

LOC coverage

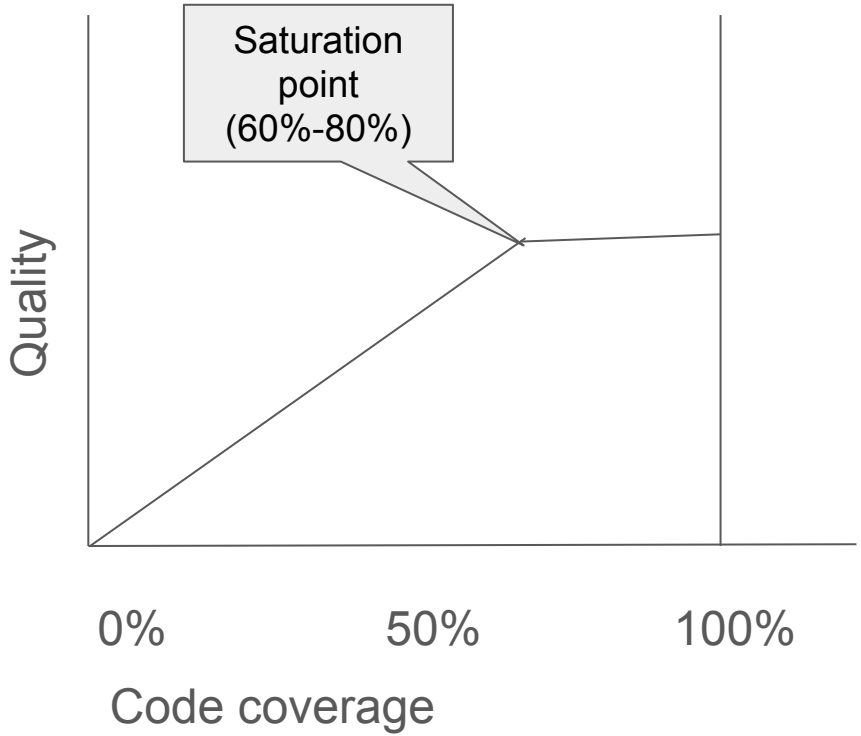
Artificial 100% coverage:
difficult to achieve, but not related to quality

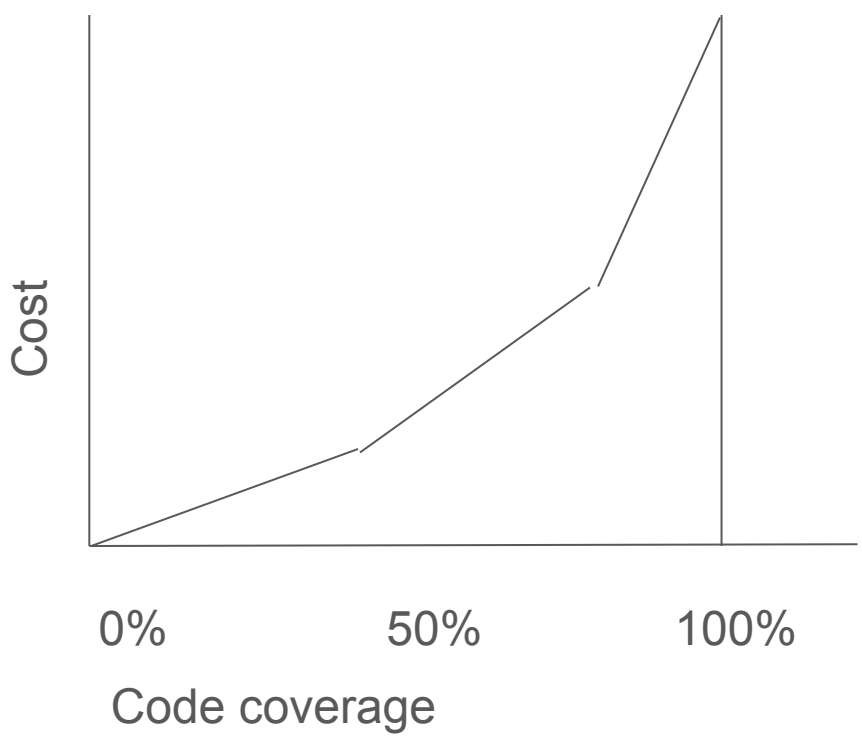


Full space of program execution paths

LOC coverage

Code coverage vs code quality vs cost





Code is covered, but nothing is tested!!!

```
public void doALotOfStuff(){
    doSomething();
    doSomethingElse();
    doAnotherThing();
    doYetAnotherThing();
}
```

We have 100% coverage, but removal of any of these lines (or adding more of them) will not affect the test result.

```
@Test
void test() {
    doALotOfStuff();
}
```

Measuring code coverage does not measure the ability of tests to actually detect faults

Mutation testing (e.g. PITest for Java)

- Takes unit tests and runs them against automatically modified (mutated) versions of code
- Mutated versions should fail the tests! If test indeed fails, it means that the mutation is **killed** (which is good), otherwise it means that the mutation is **survived** (which means that the test has a flaw!)
- Mutations include:
  - Replacement of Boolean subexpressions with *true* and *false*
  - Replacement of some arithmetic operations with others, e.g. `+` with `*`, `-` with `/`
  - Replacement of some Boolean relations with others, e.g. `>` with `>=`, `==` and `<=`
  - Replacement of variables with others from the same scope (variable types must be compatible)

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
21	99%	1505/1519	81%	459/564	82%	459/557

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
Classes' names omitted for privacy	100%	33/33	38%	5/13	38%	5/13
	94%	32/34	85%	17/20	89%	17/19
	100%	19/19	56%	5/9	56%	5/9
	100%	67/67	100%	33/33	100%	33/33
	100%	34/34	93%	14/15	93%	14/15
	99%	111/112	98%	40/41	98%	40/41
	99%	110/111	79%	30/38	79%	30/38
	100%	99/99	58%	7/12	58%	7/12
	100%	37/37	90%	26/29	90%	26/29
	100%	148/148	85%	46/54	85%	46/54
	100%	91/91	93%	41/44	93%	41/44
	98%	308/313	86%	94/109	88%	94/107
	100%	45/45	71%	12/17	71%	12/17
	100%	12/12	100%	10/10	100%	10/10
	99%	175/177	57%	30/53	59%	30/51
	100%	16/16	25%	1/4	25%	1/4
	100%	13/13	100%	1/1	100%	1/1
	60%	3/5	0%	0/1	0%	0/0

Mutation coverage out-takes

- For mission-critical pieces of code, mutation testing provides **high level of confidence**, as it measures the quality of tests themselves.
- Mutation coverage value <= code coverage value. If it’s hard to achieve high code coverage, it’s even harder to achieve high mutation coverage.
- Mutation testing is **slow**.

  In my practice, a set of unit tests for mission-critical code which normally executed in **3-5 seconds**, under PITest run for **30 minutes**.

In real life, mutation testing is considered to be impractical and is used very rarely.

General conclusions

- “Code coverage” is not a single term. There is byte code coverage, lines of code coverage, branch coverage, execution paths coverage, mutation coverage (for various sets of mutators), and probably others. All of them depend on the implementation in the tools used.
- These “coverages” build a hierarchy with traditional “lines of code” coverage being the most “optimistic” metric, while execution paths and mutation coverage being more precise and “pessimistic” (everybody wants to boast bigger figures, so so we all use the LOC coverage).
- When any coverage metric is high, its increase does not contribute to the increase of quality.

General conclusions

- **Code coverage is one of the most powerful & important tools in QA, but one must know how to properly use it.**
- **The value of coverage heavily depends on what we measure and how, and “100% coverage” is meaningless.**