TRANSITION TECHNOLOGIES

Mutanci są wśród nas – testy mutacyjne

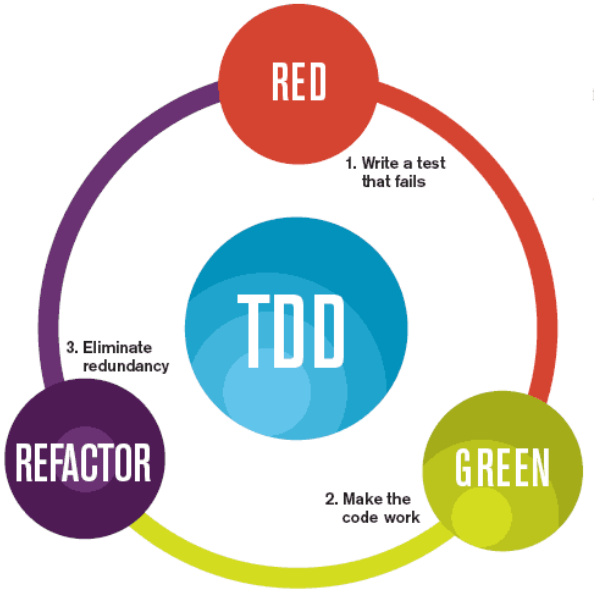
Michał Dubel

Instead of agenda… rise Your hand !

- Unit testing
- Integration testing
- TDD
- Continuous Integration
- Code coverage measures
- Mutation testing

Why testing ? Test Driven what ?

- Testing is cool
- Testing makes Your life easier
- Testing makes You work faster
- Testing makes Your code better
- Write test then code aka TDD
- Writing **good** tests is **hard** and takes **practice**



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

Code coverage

„In computer science, **code coverage** is a measure used to describe the degree to which the source code of a program is tested”

Wikipedia - https://en.wikipedia.org/wiki/Code_coverage

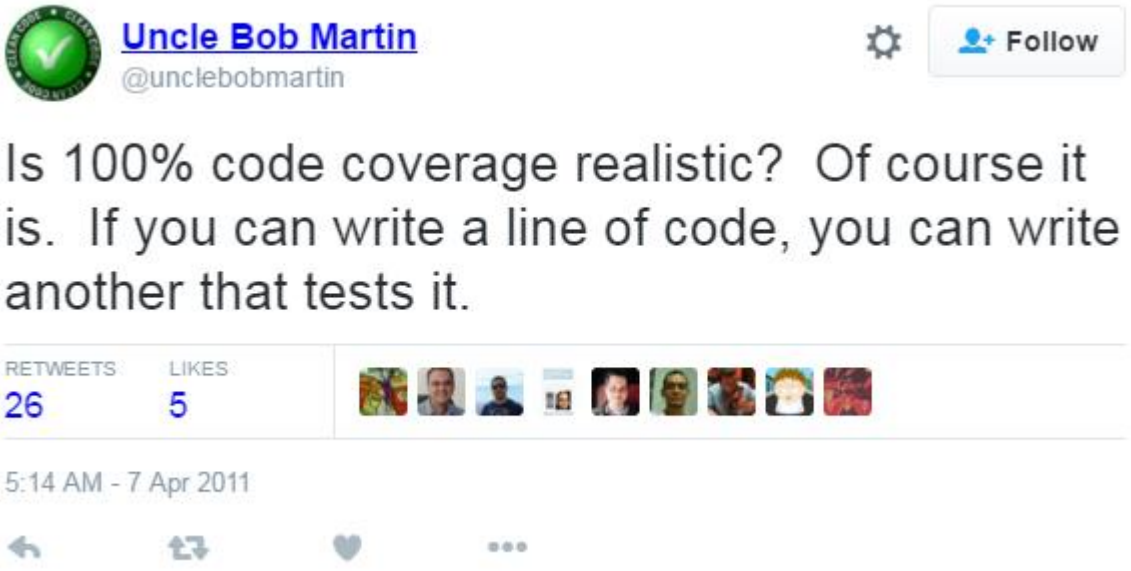
Package	# Classes	Line Coverage		Branch Coverage		Complexity
hudson.tools	6	89%	50/56	81%	13/16	0
hudson.views	15	75%	45/60	N/A	N/A	0
hudson.search	28	67%	197/295	60%	77/128	1.421
hudson.scheduler	7	67%	286/426	40%	95/240	2.294
hudson.node_monitors	28	61%	156/254	39%	29/74	0
hudson.slaves	50	52%	327/634	37%	74/202	1.6
hudson.model.listeners	5	52%	33/63	46%	12/26	1.667
hudson.triggers	18	51%	133/259	37%	28/76	1.5
hudson.widgets	5	50%	27/54	17%	3/18	1
hudson.matrix	27	50%	343/684	32%	103/318	1
hudson	125	49%	1393/2820	43%	485/1135	2.645
hudson.model	303	48%	3602/7534	39%	1234/3139	1.46
hudson.tasks	68	46%	623/1363	32%	178/552	1
hudson.security	84	45%	529/1172	28%	110/391	1.724
hudson.tasks.junit	17	43%	218/511	36%	68/190	3
hudson.maven	79	43%	894/2056	32%	252/790	1.406
hudson.util	167	35%	1030/2907	29%	297/1028	1.832
hudson.scm	83	35%	754/2150	25%	230/932	3
hudson.util.spring	10	34%	159/473	24%	49/208	1.238
hudson.diagnosis	3	31%	12/39	7%	1/14	0
hudson.maven.reporters	37	29%	167/579	19%	37/198	2.182
hudson.tasks.test	15	19%	54/288	6%	6/103	0
hudson.scm.browsers	20	8%	16/200	2%	1/66	2
hudson.logging	6	8%	13/172	4%	2/54	0
hudson.os.solaris	10	6%	13/226	5%	3/58	0
hudson.os.windows	4	3%	4/123	0%	0/14	0
hudson.lifecycle	11	2%	6/252	2%	1/66	0
hudson.util.jna	20	0%	0/157	0%	0/42	1
hudson.tasks.labelers	3	0%	0/39	0%	0/28	0
hudson.org.apache.tools.ant.taskdefs.cvslib	8	0%	0/402	0%	0/153	1.667
hudson.org.apache.tools.ant.taskdefs	1	0%	0/246	0%	0/118	0
hudson.fsp	5	0%	0/59	0%	0/12	0

Code coverage - tools

- Clover
- EclEmma
- Cobertura
- And more…

$$CC = \frac{L_{executed}}{L_{total}} * 100$$

100% Code Coverage = Holy Grail ?



Robert C. Martin (Uncle Bob)
https://twitter.com/unclebobmartin/status/559666205096673328

100% Code Coverage = Holy Grail ?

```
@Before
public void initialize(){
    sut = new MagicCalculator();
}


@Test
public void shouldAddTwoPositiveNumbers(){
    //given
    //when
    double result = sut.add(2, 3);
    //then
}
```

Element	Coverage	Covered Instru...	Missed Instruct...	Tota
mutants	14,9 %	7	40	
src/main/java	14,9 %	7	40	
pl.com.tt.mutants	14,9 %	7	40	
MagicCalculator.java	20,0 %	7	28	
MagicCalculator	20,0 %	7	28	
calculateSquareRoot(double)	0,0 %	0	12	
divide(double, double)	0,0 %	0	12	
subtract(double, double)	0,0 %	0	4	
add(double, double)	100,0 %	4	0	
CannotCalculateSquareRootOfNegativeNumber.java	0,0 %	0	9	
CannotDivideByZeroException.java	0,0 %	0	3	

Code Coverage = code quality metric ?

- Any metric can be fooled
- CC is a **metric**
  - CC would be fooled:
    - Accidentaly
    - On purpose

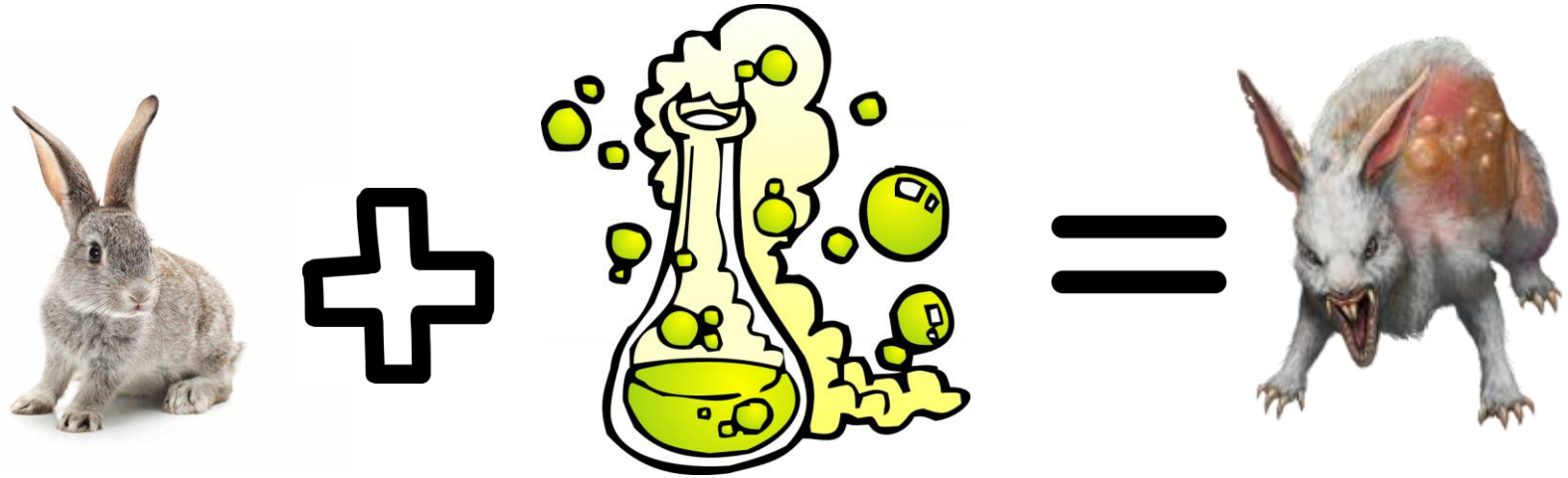
Did You see (or write ?) tests:

- without verifications or assertions?
- just to fake and boost coverage?

100% branch coverage proves (almost) nothing

Mutation testing

- Proposed by Richard J. Lipton in 1971 (winner of 2014 Knuth Prize)
- A way to measure the quality of your tests



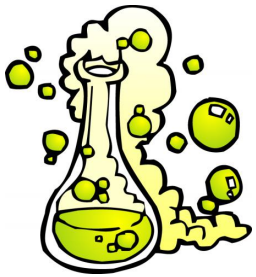
Original code | Mutation | Mutated codebase

Mutation

Small (yet significant) change in Your code, ex:

```
public double add(double number1, double number2) {

    return number1 + number2;
}
```



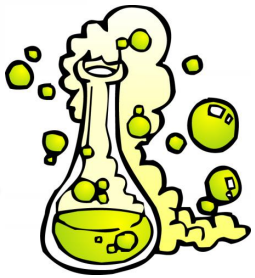
```
public double add(double number1, double number2) {

    return number1 - number2;
}
```

Mutation workflow – standard testing

Execute tests

Mutation workflow – mutation testing



Execute same tests

Mutation workflow – testing outcome

Execute same tests

Mutant survived

Mutant killed

Execute same tests

Killing mutants is **GOOD !**

So gooood… why not used widely since ages ?

- Lack of tools
- Production code modifications
- Time, time, time !
- Endless loops
- Overflow

Tools for Java

- μJava: http://cs.gmu.edu/~offutt/mujava/ **(inactive)**
- Jester: http://jester.sourceforge.net/ **(inactive)**
- Jumble: http://jumble.sourceforge.net/ **(inactive)**
- javaLanche: http://www.st.cs.uni-saarland.de/mutation/ **(inactive)**
- PIT: http://pitest.org/

PIT

- Bytecode manipulation
- Analyze lines with standard coverage
- Analyze only related tests
- Parallel execution (fast)
- Incremental analysis
- Available for: ant, maven, gradle…

Mutators

Mutators are patterns, that are applied to our codebase to produce mutants.

MUTATORS: CONDITION BOUNDARY

> into >=

< into <=

>= into >

<= into <

MUTATORS: NEGATE CONDITIONALS

== into !=

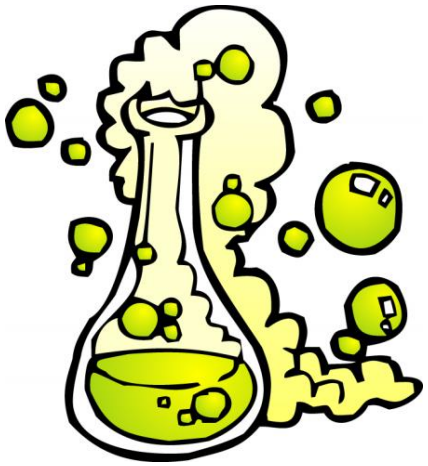
!= into ==

<= into >

>= into <

< into >=

> into <=



Mutators

MUTATORS: REMOVE CONDITIONALS

```
if(a == b) {
//something
}
```

into

```
if(true) {
//something
}
```

Mutators

MUTATORS: MATH

+ into -

- into +

* into /

/ into *

% into *

& into |

<< into >>

>> into <<

>>> into <<<

a++ into a--

a-- into a++

Mutators

MUTATORS: MANY MORE

- Replacing return values to ex. return constanc (return var; becomes return 0;)
- Removal of void invocations (callServiceXYZ(); is removed)
- Some enabled by default, others are optional or configurable

Demo

pitest.org

Time metrics…

Metrics (kind of)

On *joda-money*

```
mvn clean test-compile
```

Total time: 2.181 s

```
mvn surefire:test
```

```
mvn pit-test...
```

Total time: 48.634 s

@nicolas_frankel

40

geecon

Configure !

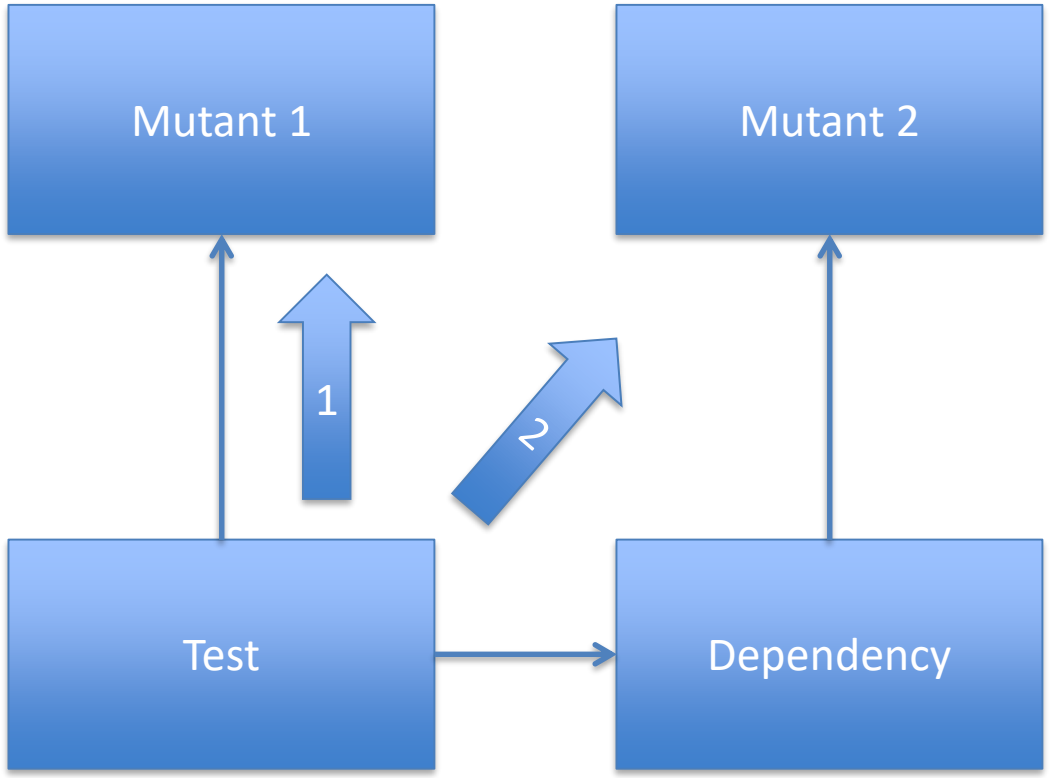
```
<configuration>
  <mutators>
    <mutator>
      CONSTRUCTOR_CALLS
    </mutator>
    <mutator>
      NON_VOID_METHOD_CALLS
    </mutator>
  </mutators>
</configuration>
```

Configure – setting target classes

```
<configuration>
  <targetClasses>
    <param>pl.com.tt.mutants*</param>
  </targetClasses>
</configuration>
```

Configure – limit dependency distance

```
<configuration>
  <maxDependencyDistance>
    4
  </maxDependencyDistance>
</configuration>
```



Configure – limit number of mutations

```
<configuration>
  <maxMutationsPerClass>
    10
  </maxMutationsPerClass>
</configuration>
```

Configure !

Don’t be tempted to use mutation testing on test phase (for ex. Unit test etc.)

```
<plugin>
  <groupId>org.p    est</groupId>
  <artifactId>pite    -maven</artifactI
  <executions>
    <execution>
      <goals>
        <goal>mutationCover    /goal>
      </goals>
      <phase>test</pha
    </execution>
  </executions>
</plugin>
```

Configure !

scmMutationCoverage goal

The scm mutation coverage goal analyses only classes that match the filters and the source file has a given status within the project source control system (by default **ADDED** or **MODIFIED**). This provides a quick way to check the coverage of changes prior to checking code in / pushing code to a repository.

How to help Yourself ?

- Write fast unit tests
- Good separation or concerns
- Separate unit and integration tests
- Use small classes

Summary

- Pretty interesting technique – the best is yet to come
- Not a silver bullet for code development !
- Use on green field projects when aiming at high quality of a tests
- …or when You have high CC and still poor quality of tests
- Keep code quality when developing tests !
- Be aware of limitations and corner cases
- Don’t focus on 100% CC !
- Stay open-minded ☺

Questions ?

That’s all Folks !