Basler
Versicherungen

bluesky
CONSIDER IT DONE

KARAKUN

sympany
versicherungen

magnolia®

JAVA
USER
GROUP
CH

ELCA
We make it work.

Gradle

OPTRAVIS
Transfer Pricing Solutions

#BaselOne21

baselone.ch

Alle Tests grün? Oh no!!!

Warum es manchmal gut ist, wenn ein Test rot wird.

#BaselOne21, 21.10.2021, Birgit Kratz

About me

Birgit Kratz

- Freelancing IT Consultant
- Java-Backend
- More than 20 years experience
- Co-Organizer of Softwerkskammer in Düsseldorf and Köln (Cologne)
- Email: mail@birgitkratz.de
- Twitter: @bikratz
- Github: https://github.com/bkratz

Agenda

What is Mutation testing

Why use Mutation testing

How to do Mutation testing

How to use Mutation testing results

Demo

First some questions

Who of you is writing (unit) test?

Who of you is writing (unit) tests first?

Do you have a target number for code coverage?

Do you sometimes fake code coverage?

Even with 100% code coverage…

… can you tell how good and reliable your tests are?

Traditional test coverage (i.e line, statement, branch, etc.) measures only which code is **executed** by your tests. It does **not** check that your tests are actually able to **detect faults** in the executed code. It is therefore only able to identify code that is definitely **not tested**.
(https://pitest.org)

What is Mutation testing

Mutation testing is conceptually quite simple.

Faults (or mutations) are automatically seeded into your code, then your tests are run. If your tests fail then the mutation is killed, if your tests pass then the mutation lived/survived.

The quality of your tests can be gauged from the percentage of mutations killed.

(https://pitest.org)

Mutation testing … is used to **design new software tests** and **evaluate the quality of existing software tests**. Mutation testing involves modifying a program in small ways. Each mutated version is called a ***mutant*** and tests detect and reject mutants by causing the behaviour of the original version to differ from the mutant. This is called ***killing*** the mutant. … Mutants are based on well-defined ***mutation operators*** that either mimic typical programming errors … The purpose is to help the tester **develop effective tests** or **locate weaknesses** in the test data used for the program… Mutation testing is a form of white-box testing.

(https://en.wikipedia.org/wiki/Mutation_testing)

In simple words

1. To do Mutation testing you need a tested code base
2. All tests must be green
3. Introduce **one** small change to your codebase (create a **mutant**)
4. Run all the tests again
5. If there is at least one red test result, the mutant was killed (which is what you want), otherwise the mutant survived (which indicates a problem in either your codebase or your tests)

Mutation Operators

Conditional Boundary Mutator

Original conditional	Mutated conditional
<	<=
<=	<
>	>=
>=	>

Increment Mutator

Original	Mutated
i++	i—
i—	i++

Invert Negatives Mutator

inverts negation of integer and floating point numbers

Original	Mutated
return -i	return i

Math Mutator

Original	Mutated
+	-
*	/
&	|
>>	<<
...	...

Negate Conditionals Mutator

Original conditional	Mutated conditional
==	!=
!=	==
>	<=
>=	<
<=	>
<	>=

Many More

Void Method Call Mutator - removes calls to void methods

Empty Returns Mutator - replaces return values with an ‘empty’ value

False Returns Mutator - always returns false for a primitive boolean return value

True Returns Mutator - always returns true for a primitive boolean return value

Null Returns Mutator - replaces return values with null

Primitive Returns Mutator - replaces int, short, long, char, float and double return values with 0

Constructor Call Mutator - replaces constructor calls with null values

And more…

What kind of problems can be detected?

Poorly chosen or missing test data

Ambiguities in code base (logical errors)

Missing test coverage

Equivalent Mutation

The mutants in this set cannot be killed because they are equivalent to the original program. No possible test input exists that can distinguish their behaviour from that of the original program.

Mutation Score

The mutation score is defined as the percentage of killed mutants with the total number of mutants.

Mutation Score = (Killed Mutants / Total number of Mutants) * 100

Mutation Test Tools

https://github.com/theofidry/awesome-mutation-testing

Demo
with Java and PIT

(https://github.com/hcoles/pitest)

Disadvantages of Mutation testing

- Can be **very** time consuming
- Cannot detect/avoid equivalent mutations, since the resulting mutant behaves in exactly the same way as the original
- Not usable for Black Box Testing

Cost of Mutation Testing

Let’s assume we have:

- a code base with 300 Java classes
- 10 test cases for each class
- on average, each test case requires 0.2 seconds for its execution
- the total test suite execution costs 300 * 10 * 0,2 = **600 seconds** (10 minutes)

Let’s assume we have, on average, 20 mutants per each class.

The total cost of mutation analysis is 300 * 10 * 0,2 * 20 = **12000 seconds** (3h 20 min)

But there are ways to reduce theses costs

Questions?

Thank you

Sample code:

https://github.com/bkratz/MutationTestingSimpleMath

https://github.com/bkratz/MutationTestingWithConwayCubes

- Email: mail@birgitkratz.de
- Twitter: @bikratz
- Github: https://github.com/bkratz