goto; copenhagen

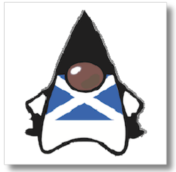
GOTO Copenhagen 2019
Conference Nov. 18 - 20

Making Mutants work for you

Henry Coles

Follow us @gotocph

Hello



pitest.org

Hello

?

Questions

How do I safely refactor my tests?

How do I know if I can trust a test suite I inherited?

How do I ensure the tests I'm writing are effective?

How do I know if my team is writing effective tests?

Really just one question

How do I assess the quality of a test suite?

Common developer answers

That's QA's problem

I’m a **Ninja Rockstar**, I know my tests are good

Better answers

I do TDD, I know my tests are good.

I do TDD, I know my tests are good.

- Are you sure?

I do TDD, I know my tests are good.

- Are you sure?
- What about the tests you didn't write?

I do TDD, I know my tests are good.

- Are you sure?
- What about the tests you didn't write?
- How do you test drive changes to your tests?

I do TDD, I know my tests are good.

- Are you sure?
- What about the tests you didn't write?
- How do you test drive changes to your tests?
- Do you write tests for your tests?

I do TDD, I know my tests are good.

- Are you sure?
- What about the tests you didn't write?
- How do you test drive changes to your tests?
- Do you write tests for your tests?
- Do you write tests for the tests for your tests??

Peer review

Peer review

Good but ...

Peer review

Good but ...

- Catches problems inconsistently

Peer review

Good but ...

- Catches problems inconsistently
- Labour intensive

Peer review

Good but ...

- Catches problems inconsistently
- Labour intensive
- Slow form of feedback

Code coverage

Code coverage

Most Commonly one of :-

Code coverage

Most Commonly one of :-

- Line

Code coverage

Most Commonly one of :-

- Line
- Branch

Code coverage

Most Commonly one of :-

- Line
- Branch
- Statement

But there are many others

But there are many others

- Data

But there are many others

- Data
- Path

But there are many others

- Data
- Path
- Modified condition / decision

But there are many others

- Data
- Path
- Modified condition / decision
- more ...

None of these of these coverage measures tell you which parts of your code have been tested

What code coverage does tell you

```java
public class AClass {
  private int count;

  public void count(int i) {
    if ( i >= 10 ) {
      count++;
    }
  }

  public void reset() {
    count = 0;
  }
}
```

What code coverage does tell you

```
public class AClass {
  private int count;

  public void count(int i) {
    if ( i >= 10 ) {  // This line has been executed
      count++;
    }
  }

  public void reset() {
    count = 0;
  }
}
```

What code coverage does tell you

```
public class AClass {
  private int count;

  public void count(int i) {
    if ( i >= 10 ) {  // This line has been executed
      count++;  // This line has been executed
    }
  }

  public void reset() {
    count = 0;
  }
}
```

What code coverage does tell you

```
public class AClass {
  private int count;

  public void count(int i) {
    if ( i >= 10 ) {  // This line has been executed
      count++;  // This line has been executed
    }
  }

  public void reset() {
    count = 0;  // This line has not been executed
  }
}
```

Executing code and testing code are not the same thing

```java
@Test
public void bossSaysMustHaveCodeCoverage() {
  AClass a = new AClass();
  a.count(0);
  a.count(9);
  a.count(11);
}
```

But most tests are written in good faith

```java
@Test
public void shouldFailWhenGivenFalse() {
  assertEquals("FAIL", foo(false));
}

@Test
public void shouldBeOkWhenGivenTrue() {
  assertEquals("OK", foo(true));
}

public static String foo(boolean b) {
  if (b) {
    performVitallyImportantBusinessFunction();
    return "OK";
  }
  return "FAIL";
}
```

Code coverage tells you only what has **not** been tested

So our answers aren't that great

Back in 1971 Richard Lipton provided a good answer to our questions

Back in 1971 Richard Lipton provided a good answer to our questions

decades before most people were writing unit tests.

He wrote a paper entitled "Fault diagnosis of computer programs"

If you want to know if a test suite has properly checked some code - introduce a bug

Then see if your test suite can find it

Here's a bug

```
public void count(int i) {
  if ( i > 10 ) {
    count++;
  }
}
```

Here's a bug

```
public void count(int i) {
  if ( i > 10 ) { // changed >= to  >
    count++;
  }
}
```

```java
@Test
public void shouldStartWithEmptyCount() {
    assertEquals(0,testee.currentCount());
}

@Test
public void shouldCountIntegersAboveTen() {
    testee.count(11);
    assertEquals(1,testee.currentCount());
}

@Test
public void shouldNotCountIntegersBelowTen() {
    testee.count(9);
    assertEquals(0,testee.currentCount());
}
```

Our tests still pass

Our test suite is deficient

A test case is missing

```
@Test
public void shouldCountIntegersOfExactlyTen() {
  testee.count(10);
  assertEquals(1,testee.currentCount());
}
```

Some terminology

Some terminology

A change such as >= to > is a **mutation operator**

Some terminology

A change such as >= to > is a mutation operator

Lots are possible

- >= to <=

Some terminology

A change such as >= to > is a **mutation operator**

Lots are possible

- >= to <=
- >= to >

Some terminology

A change such as >= to > is a **mutation operator**

Lots are possible

- >= to <=
- >= to >
- >= to =

Some terminology

A change such as >= to > is a mutation operator

Lots are possible

- >= to <=
- >= to >
- >= to =
- foo.aMethod(); to //foo.aMethod();

Some terminology

A change such as >= to > is a mutation operator

Lots are possible

- >= to <=
- >= to >
- >= to =
- `foo.aMethod();` to `//foo.aMethod();`
- `foo.aMethod();` to `foo.anotherMethod();`

Some terminology

A change such as >= to > is a mutation operator

Lots are possible

- >= to <=
- >= to >
- >= to =
- foo.aMethod(); to //foo.aMethod();
- foo.aMethod(); to foo.anotherMethod();
- 0 to 1

Some terminology

A change such as >= to > is a mutation operator

Lots are possible

- >= to <=
- >= to >
- >= to =
- foo.aMethod(); to //foo.aMethod();
- foo.aMethod(); to foo.anotherMethod();
- 0 to 1
- etc etc

Applying a mutation operator to some code creates a mutant

Applying a mutation operator to some code creates a mutant

We can create lots of mutants and we can do it automatically

If a mutant does not cause a test to fail it survived

If a mutant does cause a test to fail it was killed

If a mutant does cause a test to fail it was killed

So killing is good

But what about this?

```
class Foo {
  int min;
  public void bar(int i) {
    if (i < min) {
      min = i;
    }
    System.out.println("" + min);
  }
}
```

We can mutate it

```
class Foo {
  int min;
  public void bar(int i) {
    if (i <= min) { // changed < to <=
      min = i;
    }
    System.out.println("" + min);
  }
}
```

We can mutate it

```
class Foo {
  int min;
  public void bar(int i) {
    if (i <= min) { // changed < to <=
      min = i;
    }
    System.out.println("" + min);
  }
}
```

But it still behaves the same

It is not possible to write a test that kills this mutant

The mutant is said to be
equivalent

Equivalent mutants are considered to be a problem

Equivalent mutants are considered to be a problem

They need a human to examine them

Equivalent mutants are considered to be a problem

They need a human to examine them
We'll talk more about them later

Mutation testing highlights code that definitely is tested

It gives a very high degree of confidence in a test suite

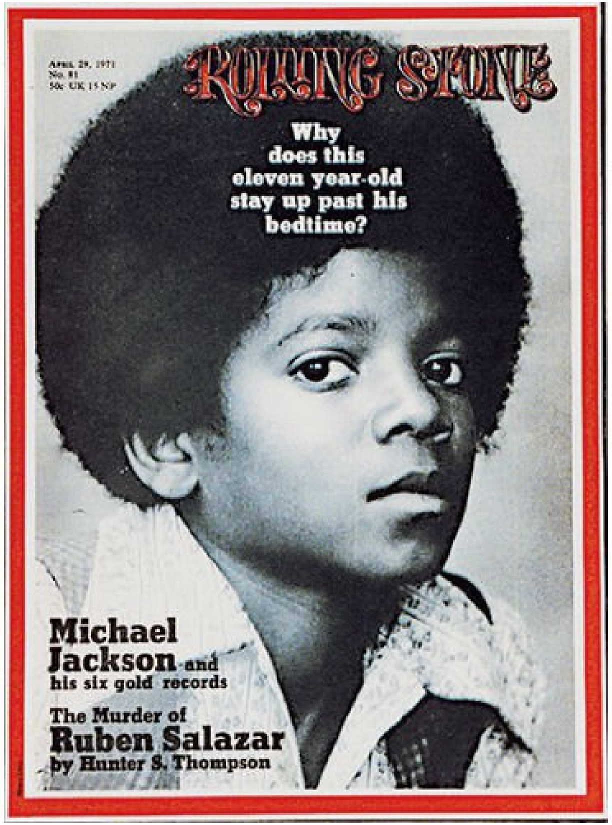
It effectively tests your tests

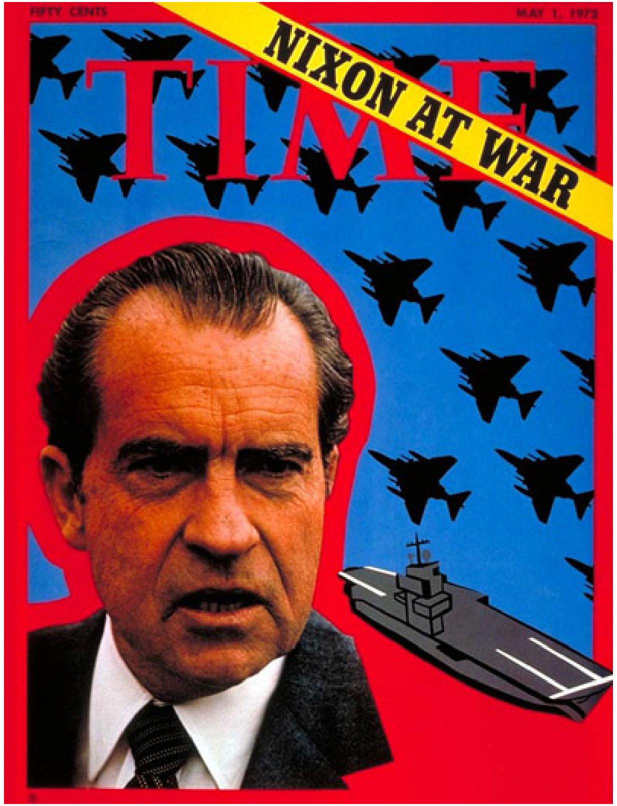
It effectively tests your tests

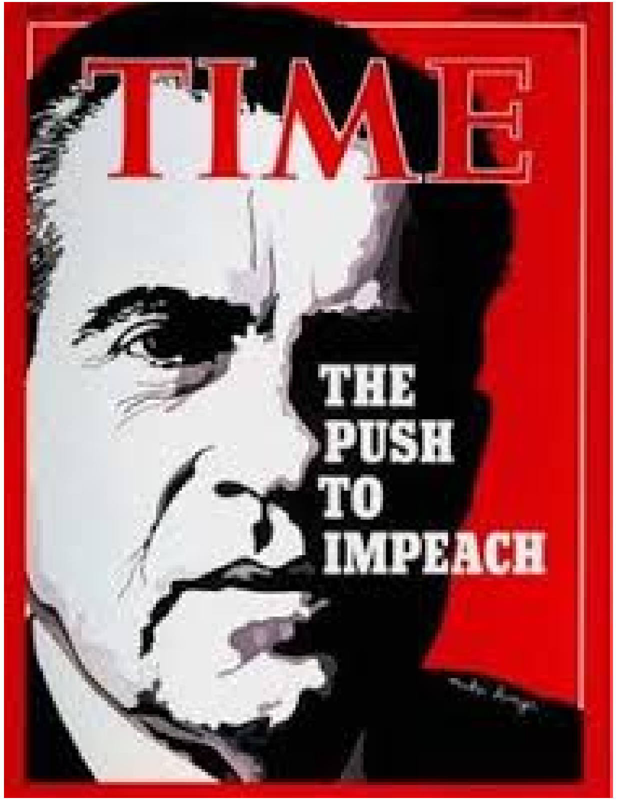
So you can refactor your tests without fear

So what happened to this idea?

1971 - Lipton's paper

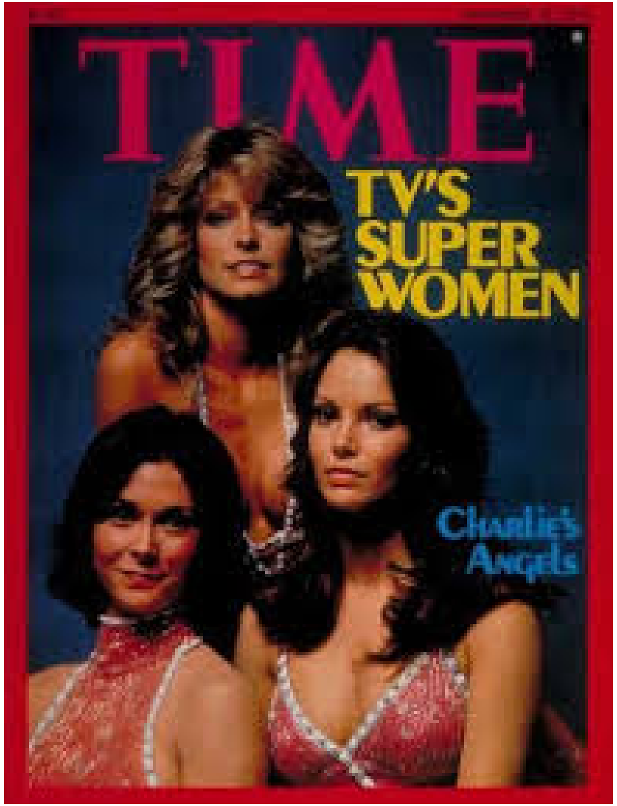


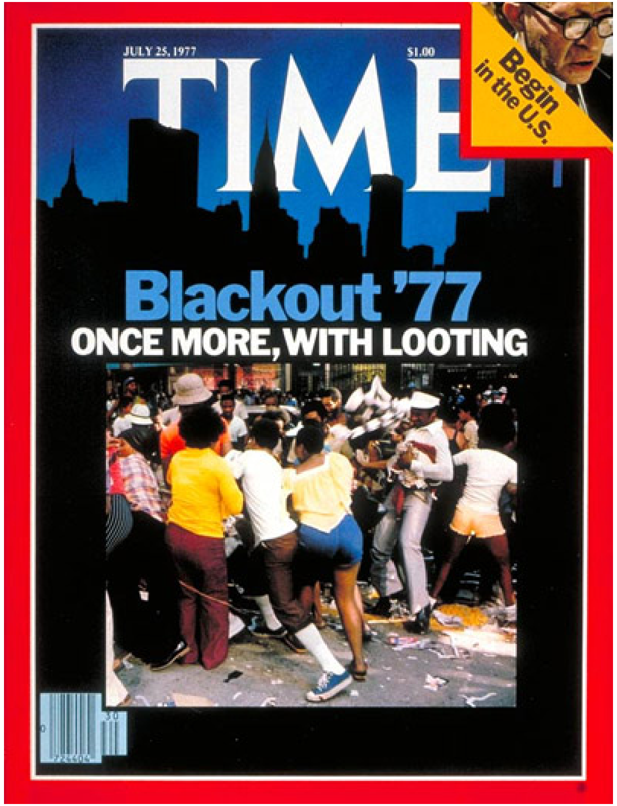

FIFTY CENTS
MAY 1, 1972
TIME
NIXON AT WAR

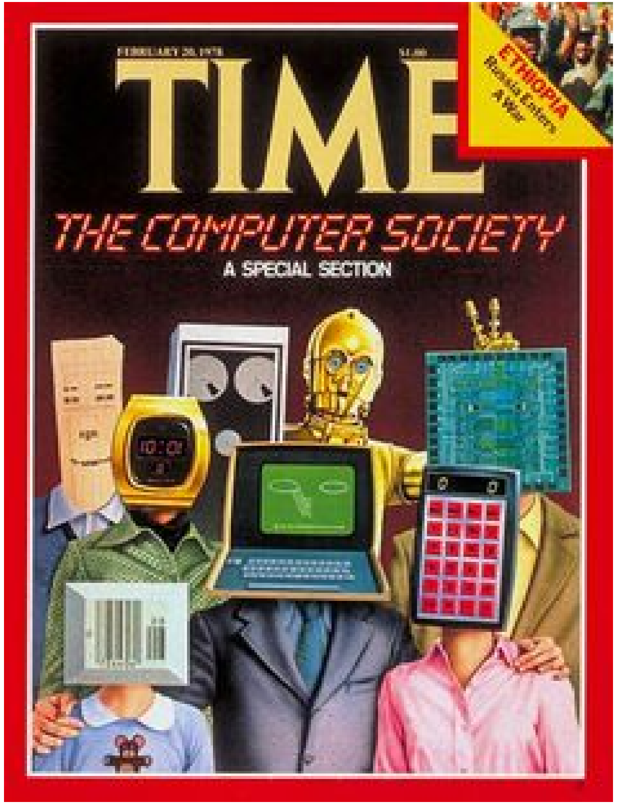

TIME
THE PUSH TO IMPEACH

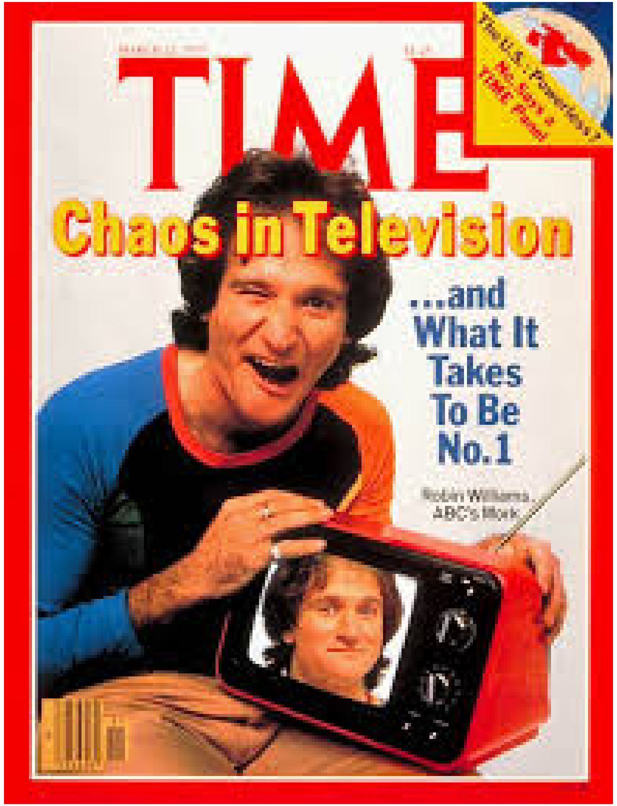

TIME
CHARGING NIXON'S MEN
ecial Prosecutor
Leon Jaworski

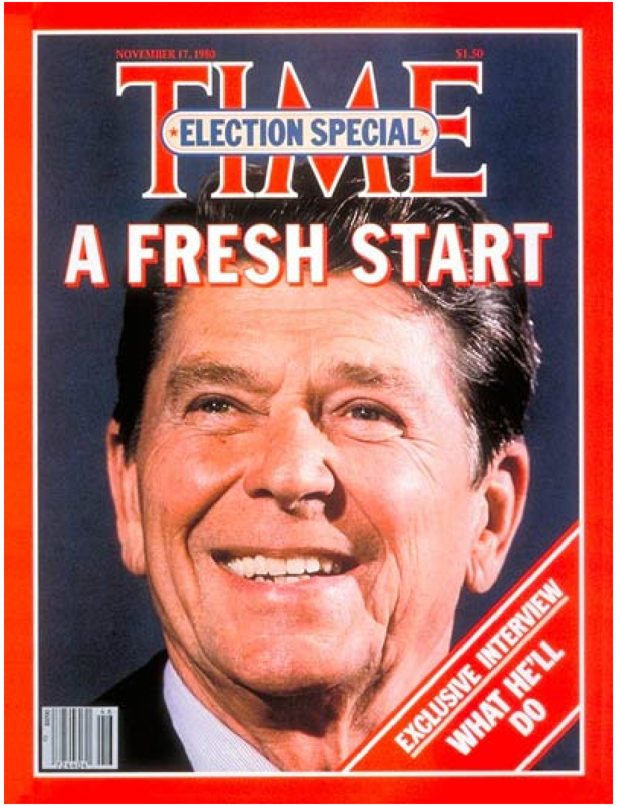

75 CENTS
APRIL 14, 1975
TIM
COLLAPSE
IN VIET NAM


TIME
TV'S SUPER WOMEN
Charlie's Angels


JULY 25, 1977
$1.00
TIME
Begin
in the U.S.
Blackout '77
ONCE MORE, WITH LOOTING


FEBRUARY 20, 1978
$1.00
TIME
ETHIOPIA
Russia Enters
A War
THE COMPUTER SOCIETY
A SPECIAL SECTION


TIME
The U.S.: Powerless?
No, Says a TIME Panel
Chaos in Television
...and What It Takes To Be No.1
Robin Williams, ABC's Mork



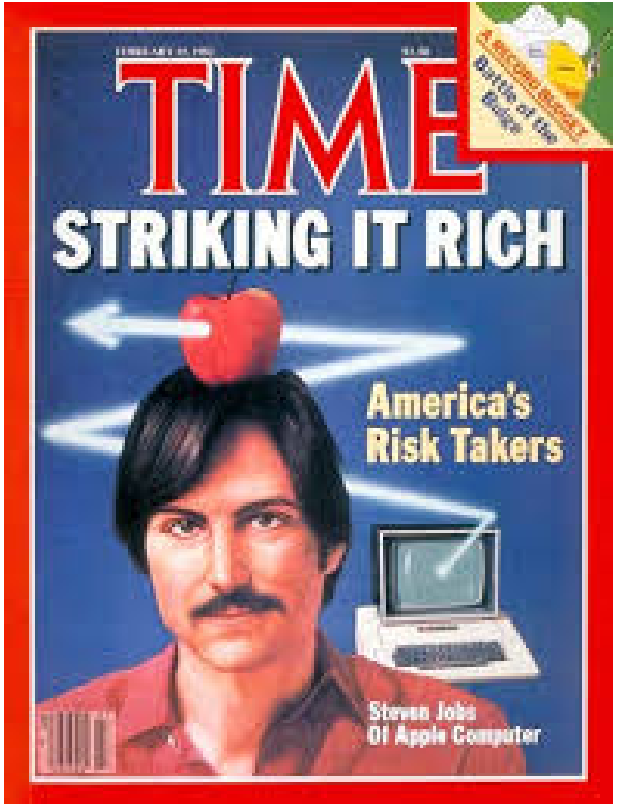
NOVEMBER 17, 1980
$1.50
TIME
ELECTION SPECIAL
A FRESH START
EXCLUSIVE INTERVIEW
WHAT HE'LL DO

Just 9 short years later

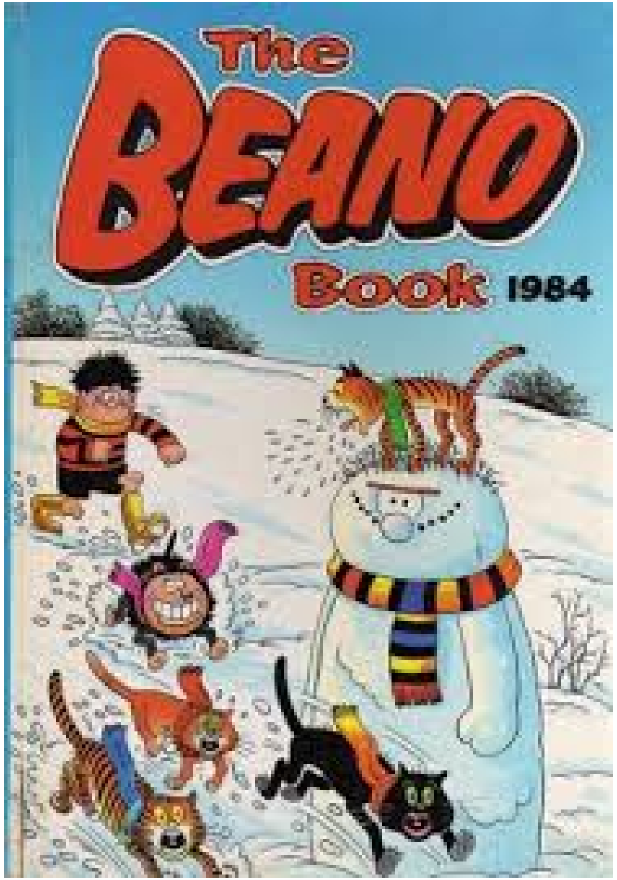
Just 9 short years later

The first automated tool!


TIME
The Prince's Charmer
Lady Diana
Is Wowing
Britain
Granting Poland
A Reprieve


TIME
STRIKING IT RICH
America's Risk Takers
Steven Jobs Of Apple Computer
A RECORD BUDGET
Battle of the Bulge


TIME
STAR WARS III
RETURN OF THE JEDI
George Lucas
& Friends
Wrap It All Up
LEBANON


The
BEANO
Book 1984

Lots of research papers

If your test suite can find artificial bugs, can it find real ones?

The competent programmer hypothesis

The competent programmer hypothesis

“Programmers are generally competent enough to produce code that is at least almost right”

The competent programmer hypothesis

“Programmers are generally competent enough to produce code that is at least almost right”

Mutation testing introduces small changes to the code

Mutation testing introduces small changes to the code

So the mutants look like bugs from our "competent" programmer

Some real bugs do look like this

Some real bugs do look like this

But others are more complex

The coupling effect

The coupling effect

"Tests that can distinguish a program differing from a correct one by only simple errors can also implicitly distinguish more complex errors"

There is strong empirical support

[1] A. Offutt. 1989. The coupling effect: fact or fiction. In Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification

There is strong empirical support

"The major conclusion from this investigation is that by explicitly testing for simple faults, we are also implicitly testing for more complicated faults" [1]

[1] A. Offutt. 1989. The coupling effect: fact or fiction. In Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification

But this is just a probabilistic statement

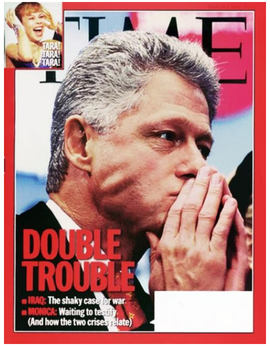
But this is just a probabilistic statement

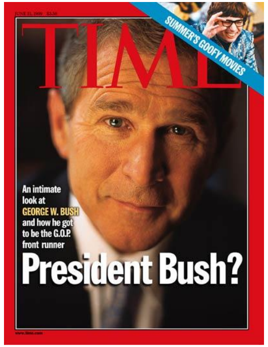
You will find counter examples

So if your tests find mutants,
they will probably find real bugs


Gorbachev's Darkest Hour (So Far)
TIME
THE BEST OF '90
Yes, Bart, even you made the list

A few more academic tools






AFGHANISTAN: DEADLY HUNT
INDIA & PAKISTAN: WAR DANCE
TIME
FLAT-OUT
COOL!
Steve Jobs thinks he has seen the future—again. Apple's new iMac is an all-in-one hub for music, pictures and movies. It's elegant and affordable. But will millions of PC users get it?


JANUARY 1, 2000
TIME
YELTSIN GOES
January 1, 2000
Welcome to a New Century
The Eiffel Tower, Paris

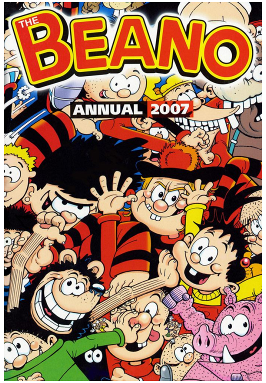
Jester

“Why just think your tests are good when you can know for sure? Sometimes Jester tells me my tests are airtight, but sometimes the changes it finds come as a bolt out of the blue. Highly recommended.”

Kent Beck

No-body used it

Lots more research papers



2019

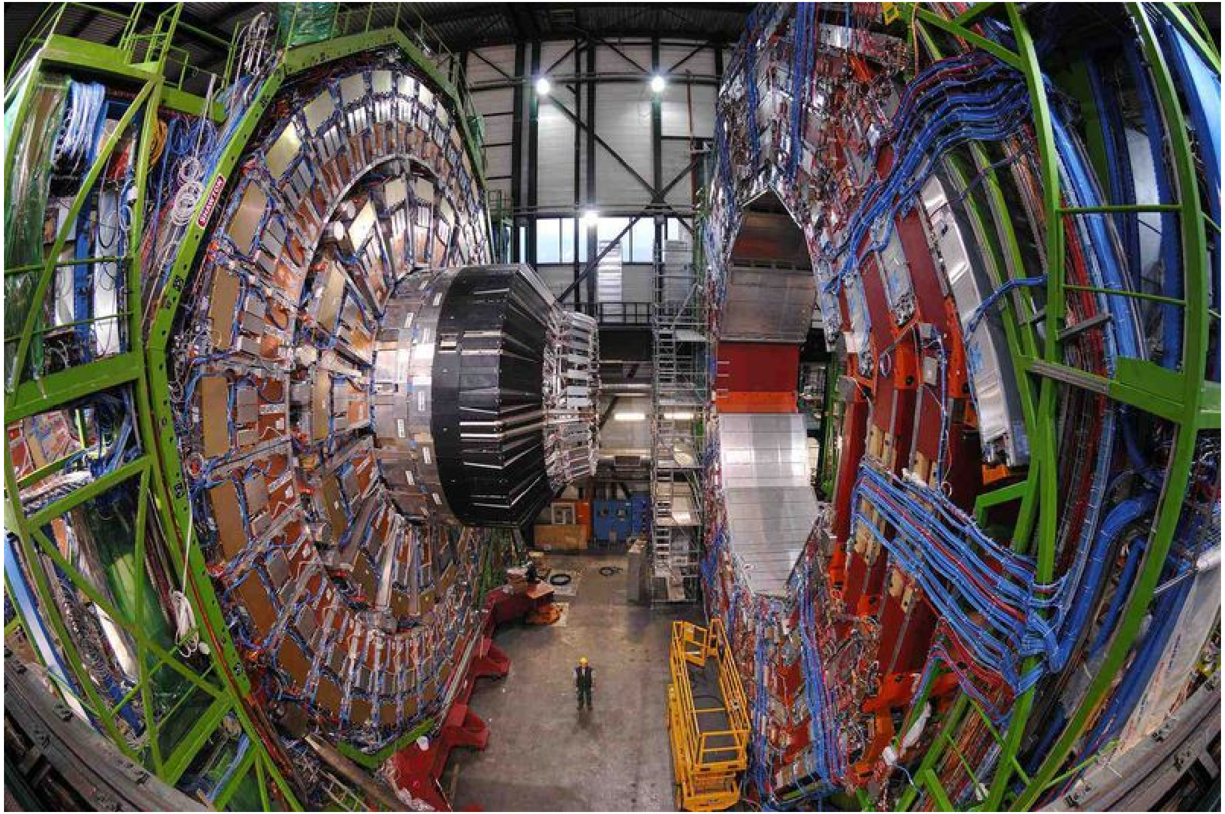
2019

In daily use all over the world

2019

In daily use all over the world
Folk keep talking about it at conferences

High profile projects



But mainly “normal” code

But mainly “normal” code

- Recruitment websites
- Tractor sales
- Insurance
- Banking
- Biotech
- Media

So what happened?

40 years of research suggested there were two fundamental problems

1. Too slow

2. Equivalent mutants

Too slow

Too slow

- Need to compile the code thousands of times

Too slow

- Need to compile the code thousands of times
- Need to run the test suite thousands of times

Joda Time

Joda Time

A small library for dealing with dates and times.

Joda Time

A small library for dealing with dates and times.

- 68k lines of code

Joda Time

A small library for dealing with dates and times.

- 68k lines of code
- 70k lines of test code

Joda Time

A small library for dealing with dates and times.

- 68k lines of code
- 70k lines of test code
- Takes about 10 seconds to compile

Joda Time

A small library for dealing with dates and times.

- 68k lines of code
- 70k lines of test code
- Takes about 10 seconds to compile
- Takes about 16 seconds to run the unit tests

Lets say we have 10k mutants

(100000)
compile

+

(160000)
test

260000
seconds

72 hours

3 days!

So in theory mutation testing is wildly impractical

I didn't understand this

I didn't understand this

So I tried to build a mutation testing tool pitest.org

Live demo time

Truth

A tiny assertion library from Google

About 3000 lines of code

Takes about 3 seconds to compile

Takes about 7 seconds to run the tests

Has about 90% line coverage

If we generated a modest 700 mutants

Would take about 2 hours

Lets try it

```
mvn -Ppitest test
```

So why didn't that take 2 hours?

Lots of reasons

It runs in parallel

It runs in parallel

Mutation testing is embarrassingly parallelisable

It runs in parallel

Mutation testing is embarrassingly parallelisable
Most machines these days have at least 2 cores

It runs in parallel

Mutation testing is embarrassingly parallelisable

Most machines these days have at least 2 cores

2 cores = half the time

No compilation cycles

No compilation cycles

Mutants created by bytecode manipulation

No compilation cycles

Mutants created by bytecode manipulation

Can generate hundreds of thousands in <1 second

Test prioritisation

Test prioritisation

Run the cheap tests first, the expensive ones later

Test prioritisation

Run the cheap tests first, the expensive ones later
stop when one fails

Test selection

Pitest gathers per test line coverage data

Test selection

Pitest gathers per test line coverage data

Tests are only run against a mutant if they exercise the mutated line of code

This makes a huge difference

```
public class AClass {
  private int count;

  public void count(int i) {
    if ( i >= 10 ) {
      count++;
    }
  }

  public void reset() {
    count = 0;
  }
}
```

```java
public class AClass {
  private int count;

  public void count(int i) {
    if ( i >= 10 ) {
      count++;
    }
  }

  public void reset() {
    count = 0;
  }
}
```

shouldNotCountIntegersBelowTen

```java
public class AClass {
  private int count;

  public void count(int i) {
    if ( i >= 10 ) {
      count++;
    }
  }

  public void reset() {
    count = 0;
  }
}
```

shouldNotCountIntegersBelowTen

shouldCountIntegersAboveTen

```java
public class AClass {
  private int count;

  public void count(int i) {
    if ( i >= 10 ) {
      count++;
    }
  }

  public void reset() {
    count = 0;
  }
}
```

shouldNotCountIntegersBelowTen

shouldCountIntegersAboveTen

shouldStartWithEmptyCount

```java
public class AClass {
  private int count;

  public void count(int i) {
    if ( i > 10 ) {
      count++;
    }
  }

  public void reset() {
    count = 0;
  }
}
```

```
1  public class AClass {
2    private int count;
3
4    public void count(int i) {
5      if ( i > 10 ) {
6        count++;
7      }
8    }
9
10   public void reset() {
11     count = 0;
12   }
13 }
```

- We will only run 2 tests for the mutation on line 5

```java
1  public class AClass {
2    private int count;
3
4    public void count(int i) {
5      if ( i > 10 ) {
6        count++;
7      }
8    }
9
10   public void reset() {
11     count = 0;
12   }
13 }
```

- We will only run 2 tests for the mutation on line 5
- The mutation will survive as we're missing an effective test case

```java
public class AClass {
  private int count;

  public void count(int i) {
    if ( i >= 10 ) {
     //count++;
    }
  }

  public void reset() {
    count = 0;
  }
}
```

```
1  public class AClass {
2    private int count;
3
4    public void count(int i) {
5      if ( i >= 10 ) {
6       //count++;
7      }
8    }
9
10   public void reset() {
11     count = 0;
12   }
13 }
```

- **We will run only 1 test for the mutation on line 6**

```java
1  public class AClass {
2    private int count;
3
4    public void count(int i) {
5      if ( i >= 10 ) {
6       //count++;
7      }
8    }
9
10   public void reset() {
11     count = 0;
12   }
13 }
```

- We will run only 1 test for the mutation on line 6
- The mutation will be killed

```java
public class AClass {
  private int count;

  public void count(int i) {
    if ( i >= 10 ) {
      count++;
    }
  }

  public void reset() {
    count = 1;
  }
}
```

```java
1  public class AClass {
2    private int count;
3
4    public void count(int i) {
5      if ( i >= 10 ) {
6        count++;
7      }
8    }
9
10   public void reset() {
11     count = 1;
12   }
13 }
```

- We will run **no tests** for the mutation on line 11

```java
1  public class AClass {
2    private int count;
3
4    public void count(int i) {
5      if ( i >= 10 ) {
6        count++;
7      }
8    }
9
10   public void reset() {
11     count = 1;
12   }
13 }
```

- We will run no tests for the mutation on line 11
- The mutation will be instantly marked as survived

```
1  public class AClass {
2    private int count;
3
4    public void count(int i) {
5      if ( i >= 10 ) {
6        count++;
7      }
8    }
9
10   public void reset() {
11     count = 1;
12   }
13 }
```

- We will run no tests for the mutation on line 11
- The mutation will be instantly marked as survived
- This is makes a huge different

To analyse JodaTime

To analyse JodaTime

- 8 minutes on cheap dual core laptop

To analyse JodaTime

- 8 minutes on cheap dual core laptop
- 3 minutes on a quad core

To analyse JodaTime

- 8 minutes on cheap dual core laptop
- 3 minutes on a quad core

To analyse JodaTime

- 8 minutes on cheap dual core laptop
- 3 minutes on a quad core

That's over 1000 times faster than early systems

But what about big codebases?

Turns out size doesn't matter

To understand why we need to talk about what mutation testing is useful for

To understand **why** we need to talk about what mutation testing is **useful** for

and **how** to use it

A lot of the research **assumed** it had to work like this

Write a
bunch of code
and tests

Pass it to a QA
team to mutation test

Many developers assume you use it like this

Write a
bunch of code
and tests

Write a
bunch of code
and tests

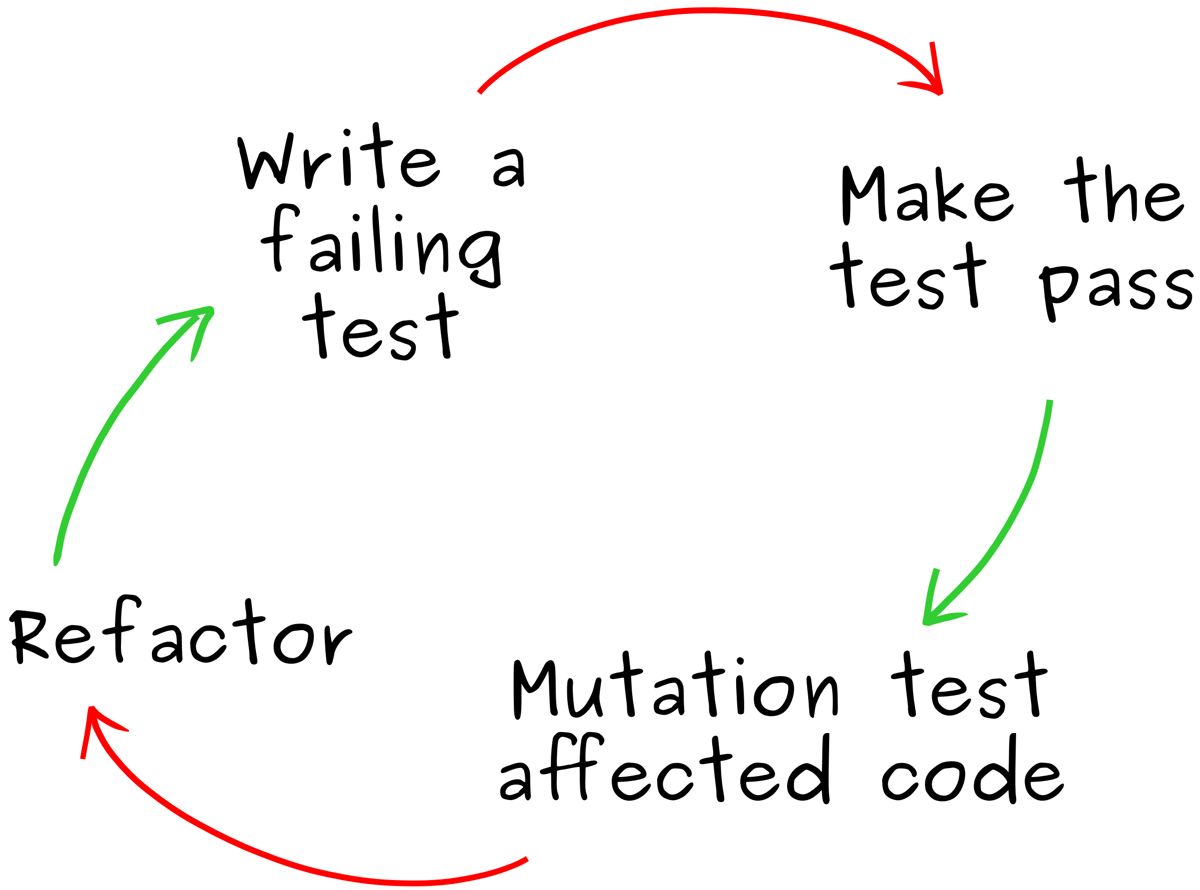
Mutation test
overnight on a server

Write a
bunch of code
and tests

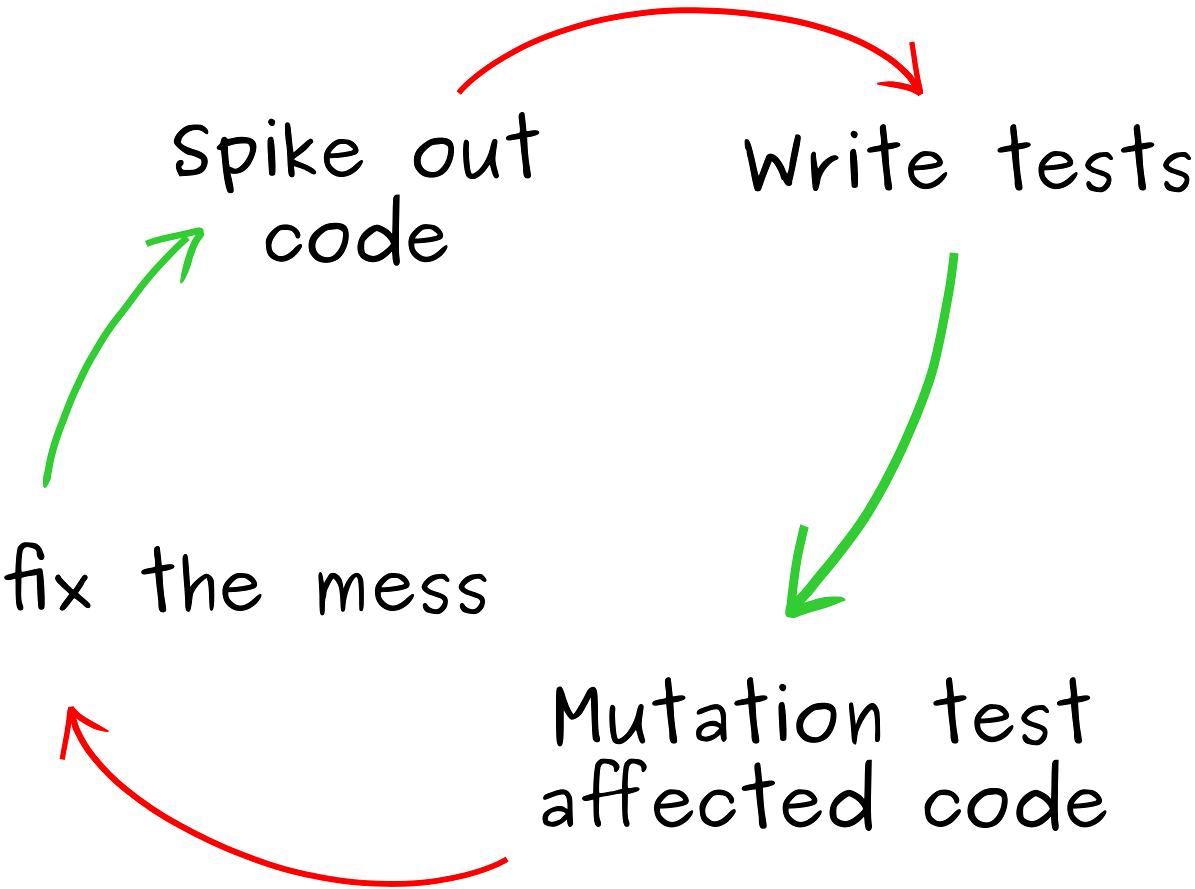
Mutation test
overnight on a server

Brag about the
size of your metric

But I only ever use it like this


Write a failing test
Make the test pass
Mutation test affected code
Refactor

Or sometimes


Spike out code
Write tests
Mutation test affected code
fix the mess

I’d be too scared to do this without mutation testing

I only have to analyse a small slice of the code

It doesn't matter how big the codebase is

The slice is always small

Pitest integrates with version control

Pitest integrates with version control

But often I just target it at a certain package

Mutation testing is a powerful tool

I don't use it as a metric

I **don't care** what the code coverage of a project is

I **don't care** what the mutation score for a project is

I care about the feedback it gives me

And the actions it prompts me to take

Equivalent mutants

Research suggests it takes 15 minutes to assess if a mutant is equivalent

But this assumes that the person assessing hasn’t just written the code

But this assumes that the person assessing **hasn't** just written the code

It's **much** less effort as part of a development feedback loop

But they can provide useful information for a developer

But they can provide useful information for a developer

They're a side benefit

If a mutant **survives** I do one of three things

Add a test

Add a test

Or sometimes fix a buggy test

Delete some code

Re-express some code

Some examples

```
public void someLogic(int i) {
  if (i <= 100) {
    throw new IllegalArgumentException();
  }

  if (i > 100) {
    doSomething();
  }
}
```

```java
public void someLogic(int i) {
  if (i <= 100) {
    throw new IllegalArgumentException();
  }

  if (i >= 100) { // mutated > to >=
    doSomething();
  }
}
```

```java
public void someLogic(int i) {
  if (i <= 100) {
    throw new IllegalArgumentException();
  }

  if (i >= 100) { // mutated > to >=
    doSomething();
  }
}
```

i can never be 100 at this point

```java
public void someLogic(int i) {
  if (i <= 100) {
    throw new IllegalArgumentException();
  }

  if (i >= 100) { // mutated > to >=
    doSomething();
  }
}
```

i can never be 100 at this point

So the mutant is equivalent

The code is redundant

```java
public void someLogic(int i) {
  if (i <= 100) {
    throw new IllegalArgumentException();
  }

  doSomething();
}
```

```java
public void someLogic(int i) {
  if (i <= 100) {
    throw new IllegalArgumentException();
  }

  doSomething();
}
```

Functionally equivalent but there's less of it

Mutation testing is really good at highlighting redundant code

```java
class Foo {
  int min;
  public void bar(int i) {
    if (i < min) {
      min = i;
    }
    System.out.println("" + min);
  }
}
```

```java
class Foo {
  int min;
  public void bar(int i) {
    if (i <= min) { // mutate < to <=
      min = i;
    }
    System.out.println("" + min);
  }
}
```

A classic logically equivalent mutant

A classic logically equivalent mutant

We can make it go away

```java
class Foo {
  int min;
  public void bar(int i) {
    min = Math.min(i, min);
    System.out.println("" + min);
  }
}
```

```
class Foo {
  int min;
  public void bar(int i) {
    min = Math.min(i, min);
    System.out.println("" + min);
  }
}
```

The code now expresses its intent

So sometimes equivalent mutants prompt us to improve the code

Mutation testing creates pressure to

Mutation testing creates **pressure** to

- Reduce the amount of code

Mutation testing creates pressure to

- Reduce the amount of code
- Reduce the amount of duplication

Many equivalent mutants affect only performance

Many equivalent mutants affect only performance

(performance is not a concern of unit testing)

Is it a premature optimisation?

Is it a premature optimisation?

- Yes - delete the code

Is it a premature optimisation?

- Yes - delete the code
- No - ignore the mutant

What did pitest find in Google Truth?

Some classic test errors

PrimitiveIntSubjectArray

```java
public void isNotEqualTo(Object expected) {
  int[] actual = getSubject();
  try {
    int[] expectedArray = (int[]) expected;
    if (actual == expected || Arrays.equals(actual, expectedArray)) {
      failWithRawMessage("%s unexpectedly equal to %s."
                        , getDisplaySubject()
                        , Ints.asList(expectedArray));
    }
  } catch (ClassCastException ignored) {
  }
}
```

PrimitiveIntSubjectArray

```java
public void isNotEqualTo(Object expected) {
  int[] actual = getSubject();
  try {
    int[] expectedArray = (int[]) expected;
    if (actual == expected || Arrays.equals(actual, expectedArray)) {
      //failWithRawMessage("%s unexpectedly equal to %s."
      //                  , getDisplaySubject()
      //                  , Ints.asList(expectedArray));
    }
  } catch (ClassCastException ignored) {
  }
}
```

Mutant is covered by at least one test

```java
@Test
public void isNotEqualTo_FailEquals() {
  try {
    assertThat(array(2, 3)).isNotEqualTo(array(2, 3));
  } catch (AssertionError e) {
    assertThat(e).hasMessage(
      "<(int[]) [2, 3]> unexpectedly equal to [2, 3].");
  }
}
```

Mutant is covered by at least one test

```
@Test
public void isNotEqualTo_FailEquals() {
  try {
    assertThat(array(2, 3)).isNotEqualTo(array(2, 3));
    throw new Error("Expected to throw"); // <--- missing
  } catch (AssertionError e) {
    assertThat(e).hasMessage(
      "<(int[]) [2, 3]> unexpectedly equal to [2, 3].");
  }
}
```

An equivalent mutation

PrimitiveDoubleArraySubject

```java
public void isNotEqualTo(Object expectedArray, double tolerance) {
  double[] actual = getSubject();
  try {
    double[] expected = (double[]) expectedArray;
    if (actual == expected) {
      failWithRawMessage(
          "%s unexpectedly equal to %s.", getDisplaySubject(), Doubles.asList(expected));
    }
    if (expected.length != actual.length) {
      return; // Unequal-lengthed arrays are not equal.
    }
    List<Integer> unequalIndices = new ArrayList<Integer>();
    for (int i = 0; i < expected.length; i++) {
      if (!MathUtil.equals(actual[i], expected[i], tolerance)) {
        unequalIndices.add(i);
      }
    }
    if (unequalIndices.isEmpty()) {
      failWithRawMessage(
          "%s unexpectedly equal to %s.", getDisplaySubject(), Doubles.asList(expected));
    }
  } catch (ClassCastException ignored) {
    // Unequal since they are of different types.
  }
}
```

PrimitiveDoubleArraySubject

```java
public void isNotEqualTo(Object expectedArray, double tolerance) {
  double[] actual = getSubject();
  try {
    double[] expected = (double[]) expectedArray;
    if (actual == expected) {
     // failWithRawMessage(
     //   "%s unexpectedly equal to %s.", getDisplaySubject(), Doubles.asList(expected));
    }
    if (expected.length != actual.length) {
      return; // Unequal-lengthed arrays are not equal.
    }
    List<Integer> unequalIndices = new ArrayList<Integer>();
    for (int i = 0; i < expected.length; i++) {
      if (!MathUtil.equals(actual[i], expected[i], tolerance)) {
        unequalIndices.add(i);
      }
    }
    if (unequalIndices.isEmpty()) {
      failWithRawMessage(
          "%s unexpectedly equal to %s.", getDisplaySubject(), Doubles.asList(expected));
    }
  } catch (ClassCastException ignored) {
    // Unequal since they are of different types.
  }
}
```

PrimitiveDoubleArraySubject

```java
public void isNotEqualTo(Object expectedArray, double tolerance) {
  double[] actual = getSubject();
  try {
    double[] expected = (double[]) expectedArray;
    if (actual == expected) {
      failWithRawMessage(
          "%s unexpectedly equal to %s.", getDisplaySubject(), Doubles.asList(expected));
    }
    if (expected.length != actual.length) {
      return; // Unequal-lengthed arrays are not equal.
    }
    List<Integer> unequalIndices = new ArrayList<Integer>();
    for (int i = 0; i < expected.length; i++) {
      if (!MathUtil.equals(actual[i], expected[i], tolerance)) {
        unequalIndices.add(i);
      }
    }
    if (unequalIndices.isEmpty()) {
      failWithRawMessage(
          "%s unexpectedly equal to %s.", getDisplaySubject(), Doubles.asList(expected));
    }
  } catch (ClassCastException ignored) {
    // Unequal since they are of different types.
  }
}
```

Performance isn't unit testable

Performance isn't unit testable

Optimisation makes sense in this case

```java
public void isNotEqualTo(Object expectedArray, double tolerance) {
  double[] actual = getSubject();
  try {
    double[] expected = (double[]) expectedArray;
    if (actual == expected) {
      failWithRawMessage(
          "%s unexpectedly equal to %s.", getDisplaySubject(), Doubles.asList(expected));
    }
    if (expected.length != actual.length) {
      return; // Unequal-lengthed arrays are not equal.
    }
    List<Integer> unequalIndices = new ArrayList<Integer>();
    for (int i = 0; i < expected.length; i++) {
      if (!MathUtil.equals(actual[i], expected[i], tolerance)) {
        unequalIndices.add(i);
      }
    }
    if (unequalIndices.isEmpty()) {
      failWithRawMessage(
          "%s unexpectedly equal to %s.", getDisplaySubject(), Doubles.asList(expected));
    }
  } catch (ClassCastException ignored) {
    // Unequal since they are of different types.
  }
}
```

```java
public void isNotEqualTo(Object expectedArray, double tolerance) {
  double[] actual = getSubject();
  try {
    double[] expected = (double[]) expectedArray;
    if (areEqual(actual,expected,tolerance)) {
      failWithRawMessage(
          "%s unexpectedly equal to %s.", getDisplaySubject(), Doubles.asList(expected));
    }
  } catch (ClassCastException ignored) {
    // Unequal since they are of different types.
  }
}
private boolean areEqual(double[] actual, double[] expected, double tolerance) {
  if (actual == expected) return true;

  if (expected.length != actual.length) return false;

  return compareArrayContents(actual, expected, tolerance);
}
private boolean compareArrayContents(double[] actual, double[] expected,
    double tolerance) {
  List<Integer> unequalIndices = new ArrayList<Integer>();
  for (int i = 0; i < expected.length; i++) {
    if (!MathUtil.equals(actual[i], expected[i], tolerance)) {
      unequalIndices.add(i);
    }
  }
  return unequalIndices.isEmpty();
}
```

In many code bases you will encounter no equivalent mutants

Seems to depend on the domain and code style

Mutation testing is a powerful technique

It finds missing test cases

It finds buggy tests

It provides a safety net while refactoring your tests

It highlights redundant code

It can highlight code smells

Run it as you develop

Run it as you develop

Not some time later

Remember it's a tool

Remember it's a tool

Not a number you need to make go up

Remember it's a tool

Not a number you need to make go up

Or a stick to beat people with

Other languages

- Ruby - Mutant
- PHP - Humbug (now Infection)
- Java - Pitest (could also try Major)
- Kotlin - Pitest (with caveats)
- Python - Cosmic Ray
- LLVM (C, C++, Swift) - Mull
- Javascript - Stryker
- C# - Fettle

@_pitest

http://pitest.org

turns out
it's me that guards the guards

goto; copenhagen

GOTO Copenhagen 2019

Conference Nov. 18 - 20

Please

Remember to rate this session

Thank you!