
2016
Mutation
testing in
practice

http://javabook.ncredinburgh.com/

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?

Or *"Who guards the guards?"*

Common developer answers

Don’t worry - it’ll be fine

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

```
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

Lets fix that the bad faith test

```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());
}
```

All ok now?

```
public class AClass {
  private int count;

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

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”

Here's Lipton's idea

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++;
  }
}
```

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?

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

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

We can mutate it

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

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

We can mutate it

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

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

But i can **never** be 100

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

The mutant is said to be **equivalent**

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

  doSomething();
}
```

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

  doSomething();
}
```

Logically equivalent to the original code - but there's **less** of it.

The equivalent mutation has revealed some **redundant** code

Although **sometimes** they tell us something useful

Equivalent mutations are also a **problem**

Equivalent mutations are also a **problem**

Needs a human to examine them

Mutation testing highlights code that definitely **is** tested

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

It can highlight **redundant code**

It can sometimes **find bugs**

It can sometimes **find bugs**

If your program was **almost** correct, a mutant version might **be** correct

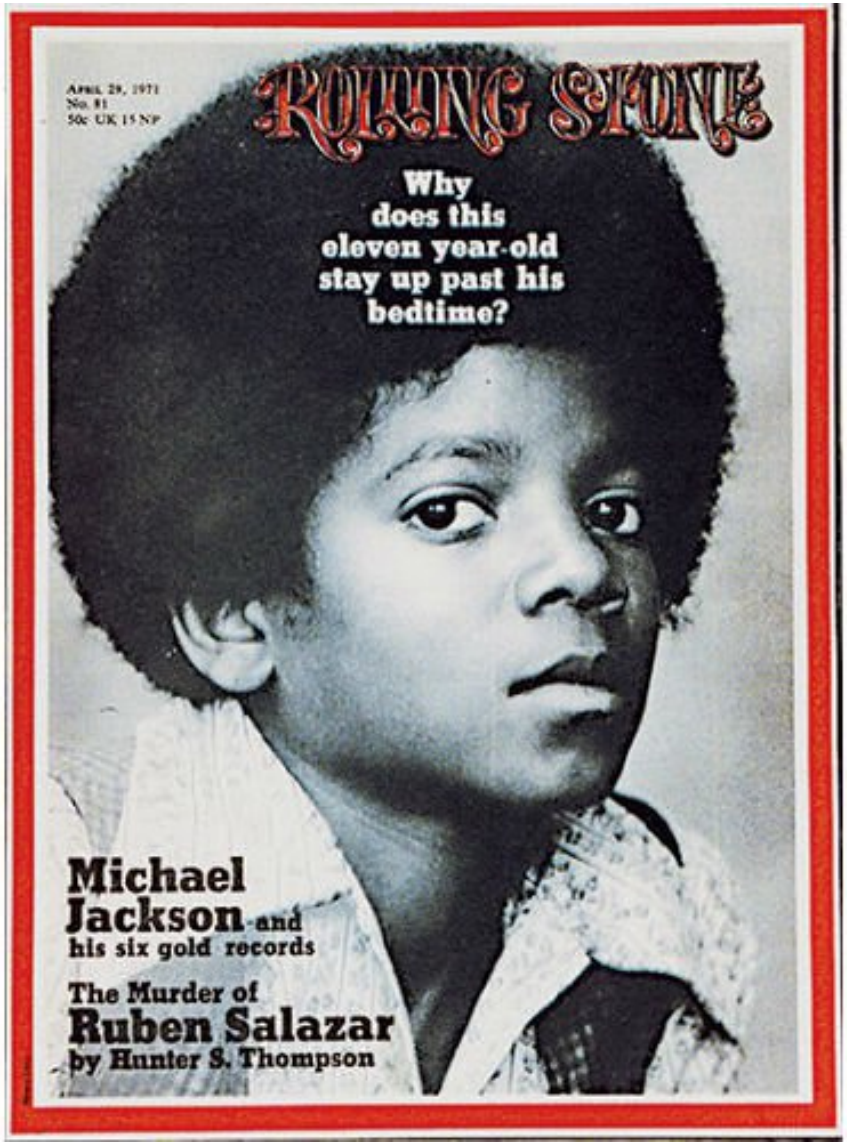
It effectively tests your tests

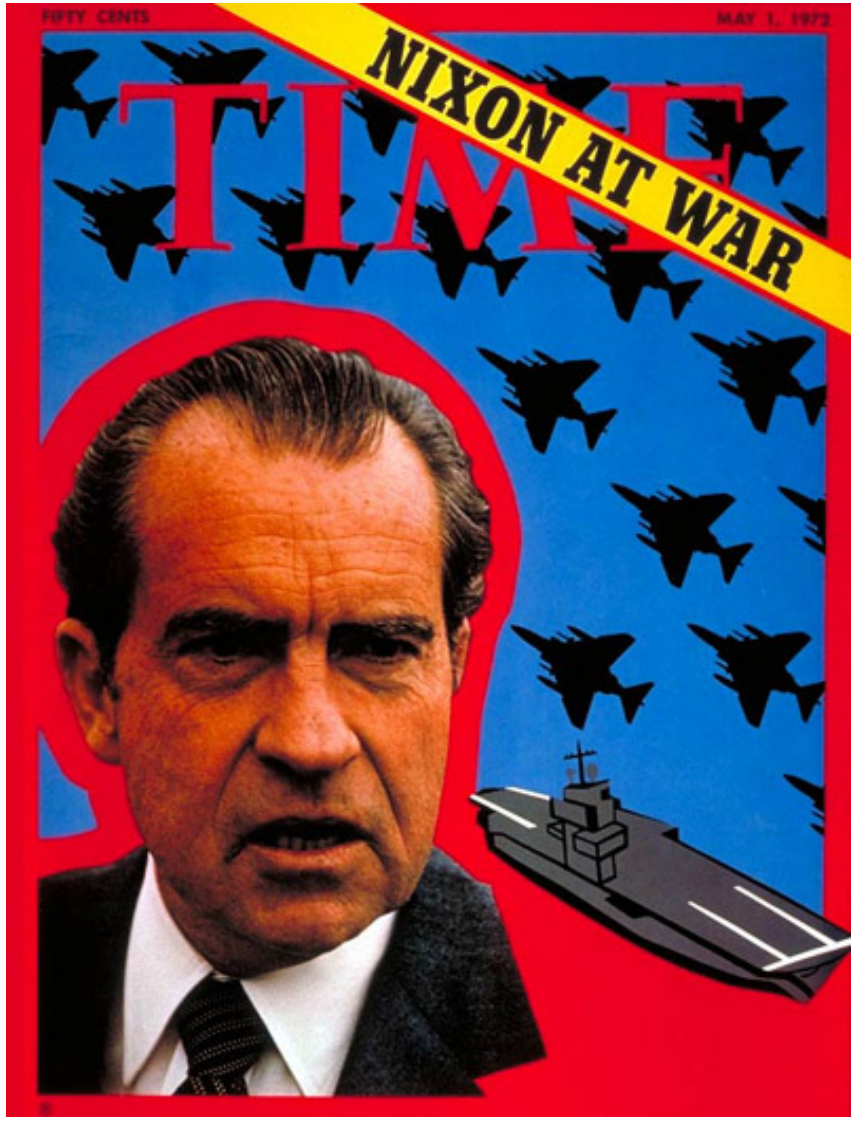
It effectively tests your tests

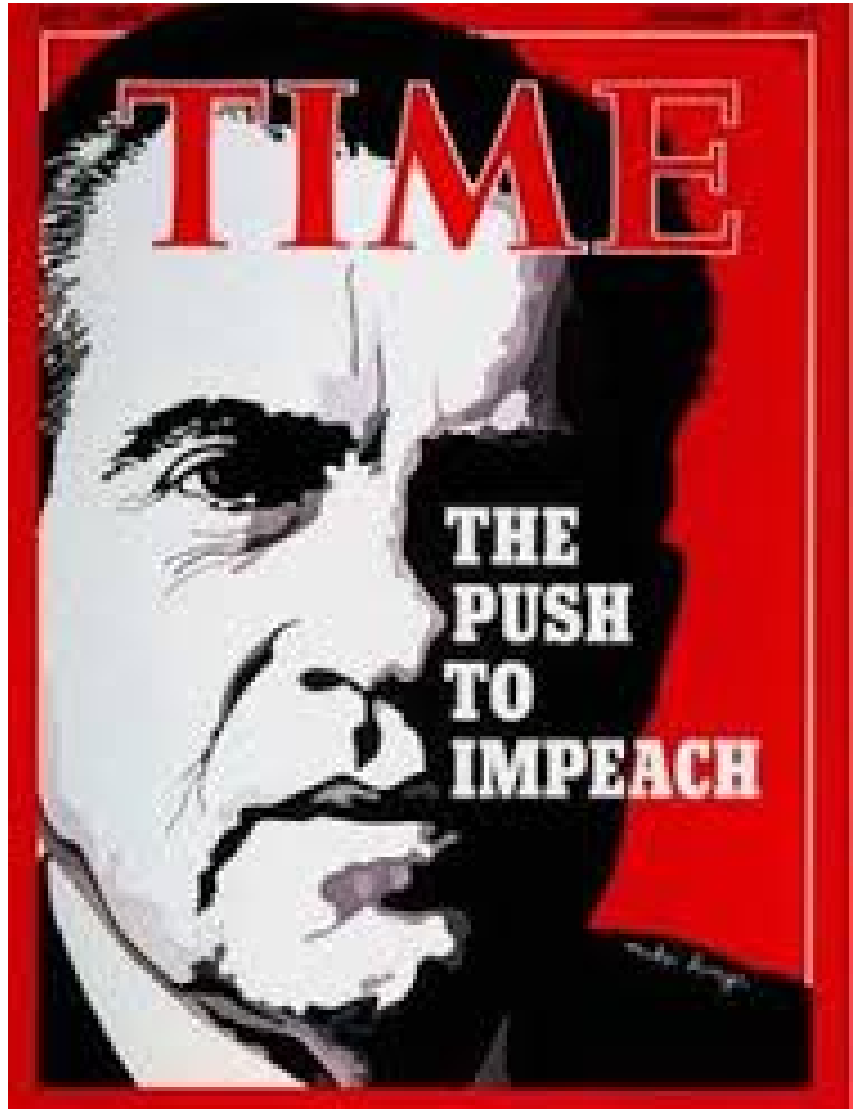
So you can refactor your tests without fear

So what happened to this idea?

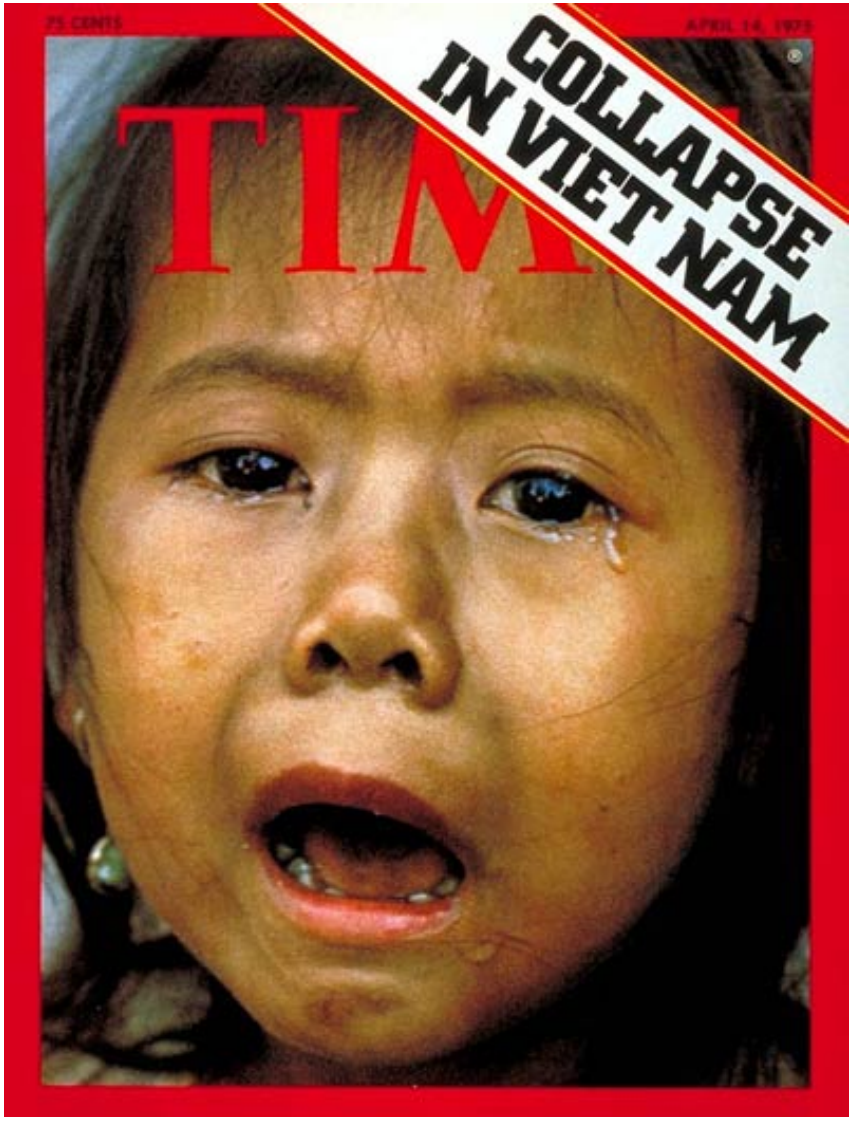
1971 - Lipton's paper

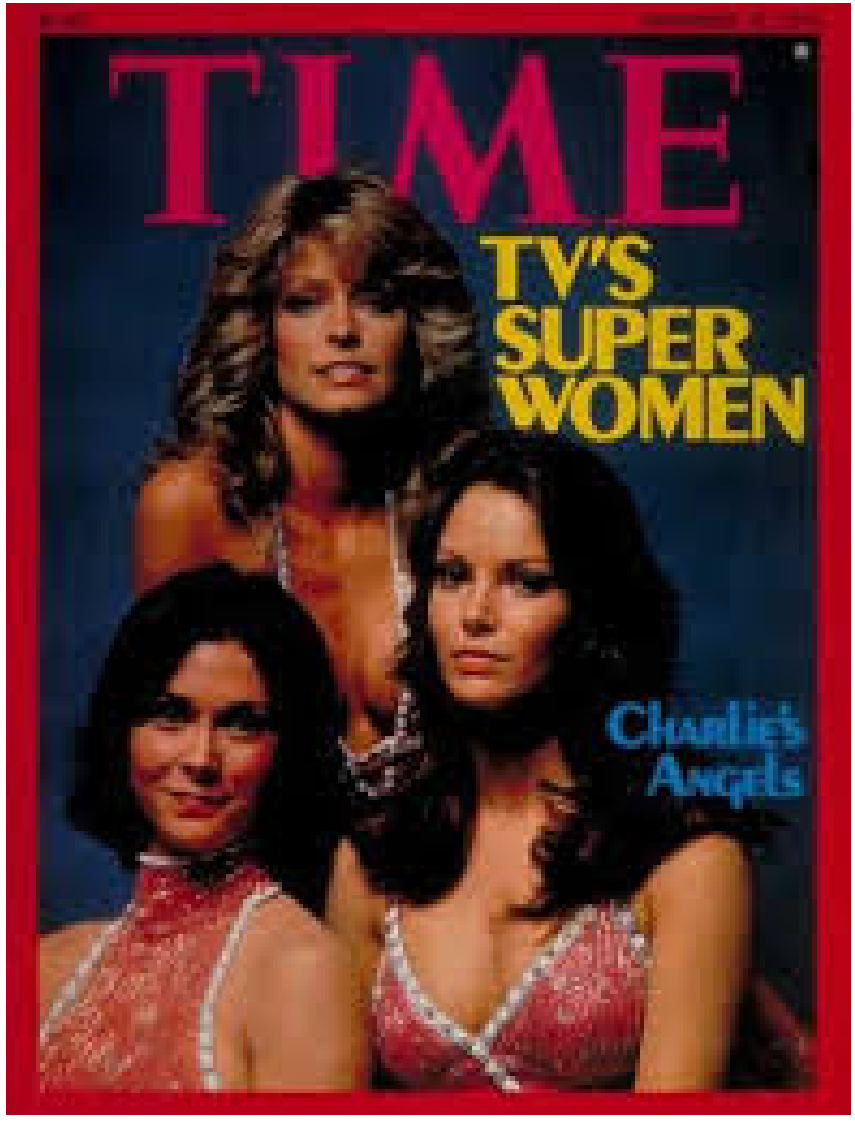


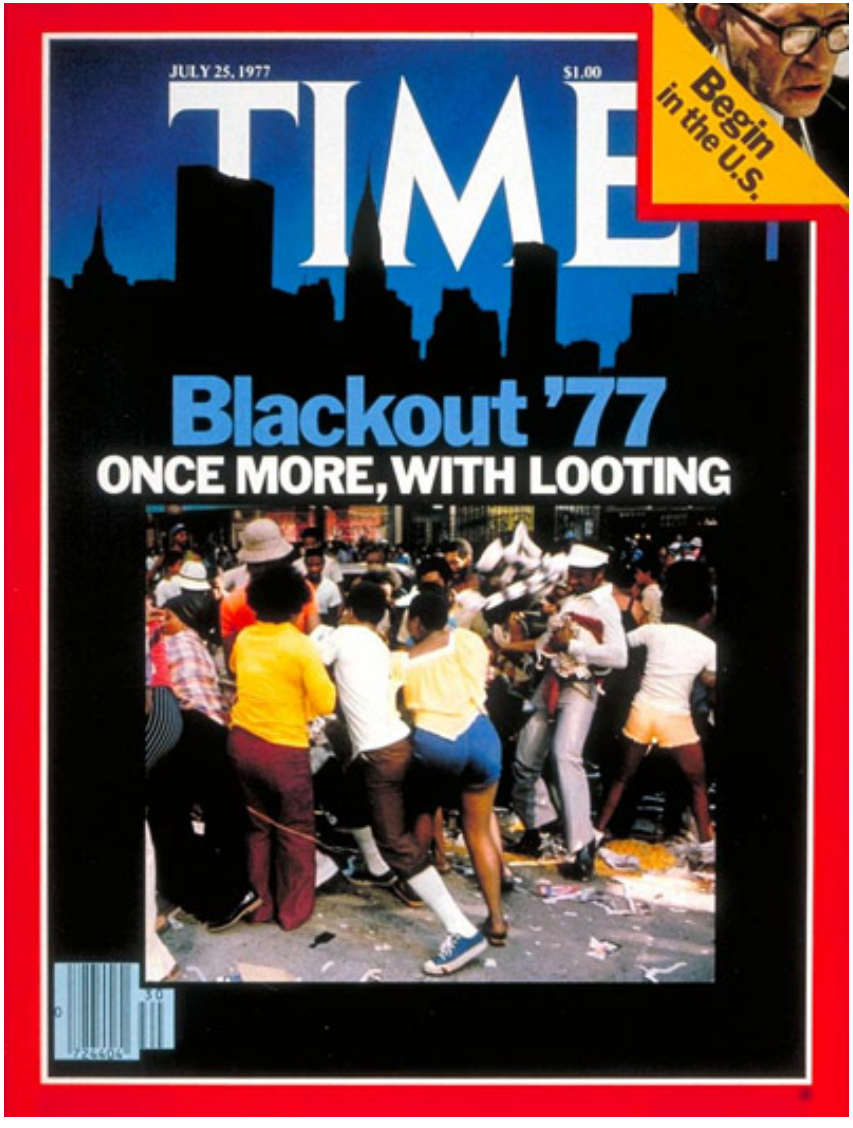

FIFTY CENTS
MAY 1, 1972
TIME
NIXON AT WAR

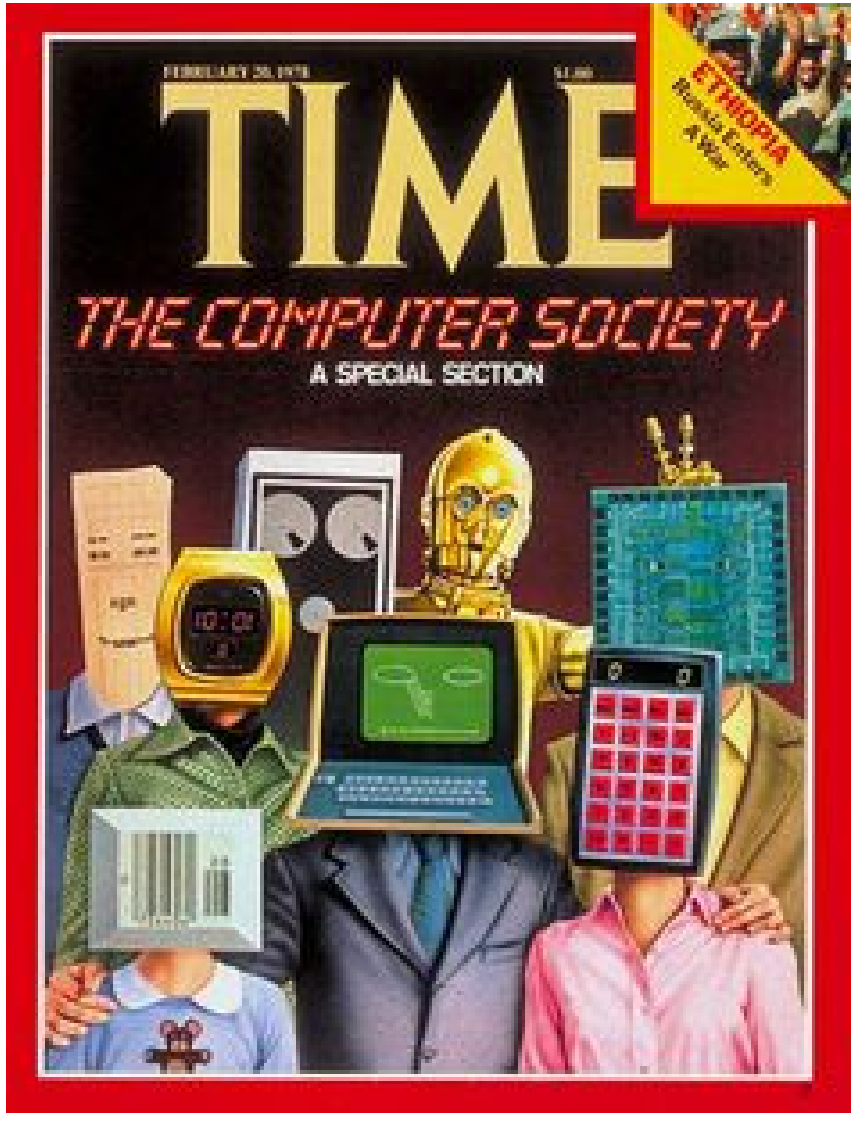

TIME
THE PUSH TO IMPEACH

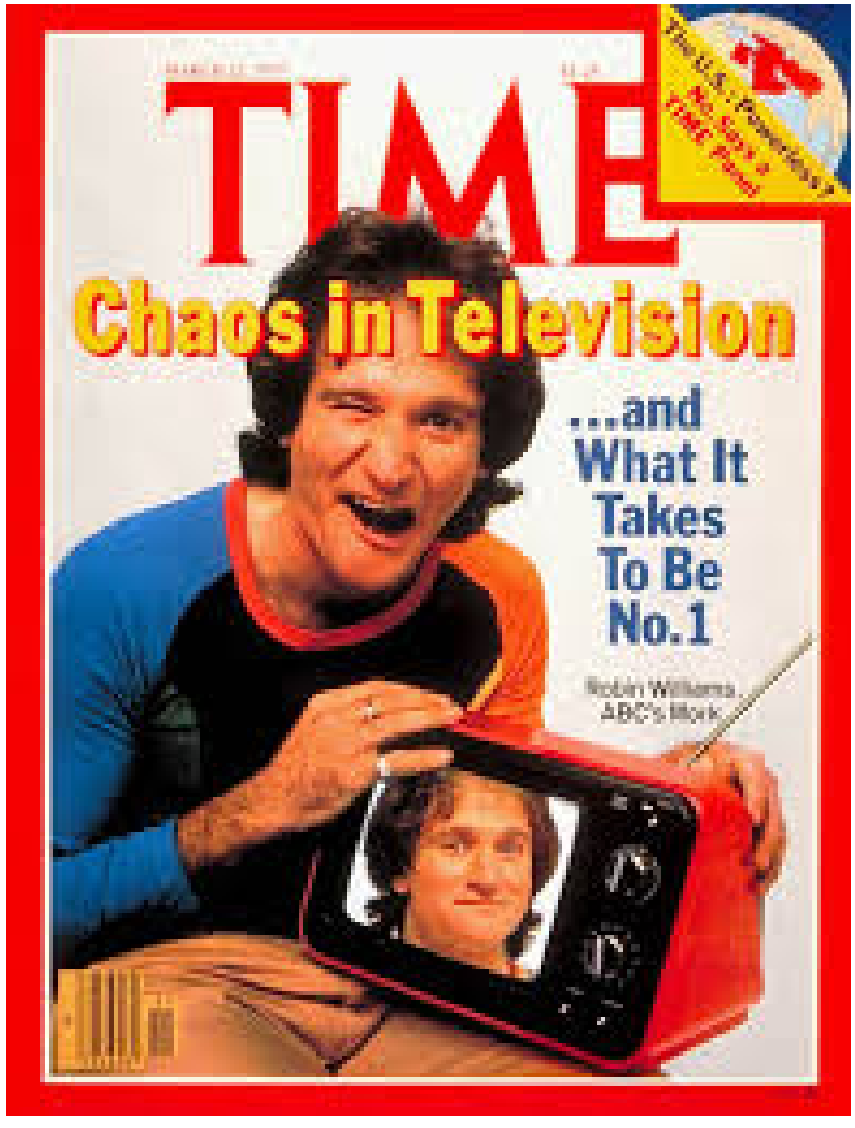

TIME
CHARGING NIXON'S MEN
ecial Prosecutor
Leon Jaworski

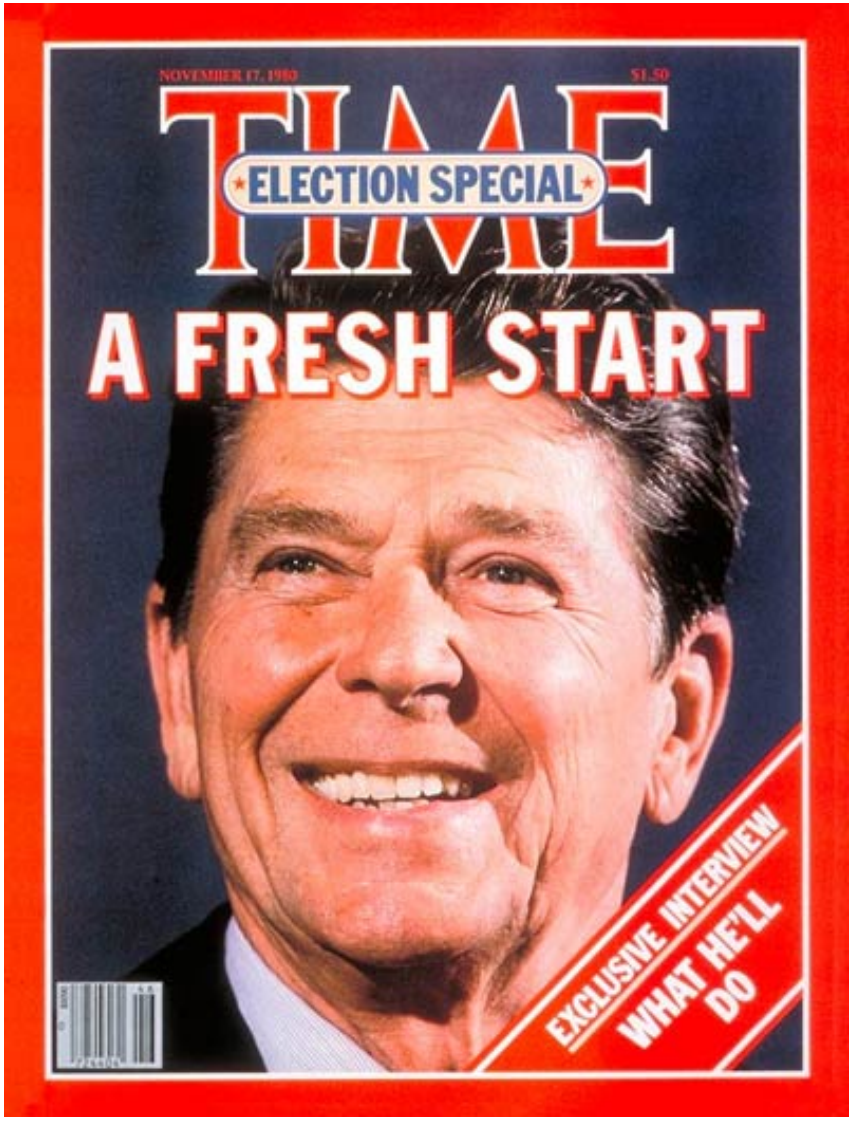

75 CENTS
APRIL 14, 1975
TIME
COLLAPSE
IN VIET NAM


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


TIME
THE COMPUTER SOCIETY
A SPECIAL SECTION
ETHIOPIA


TIME
Chaos in Television
...and What It Takes To Be No.1
Robin Williams, ABC's Mork

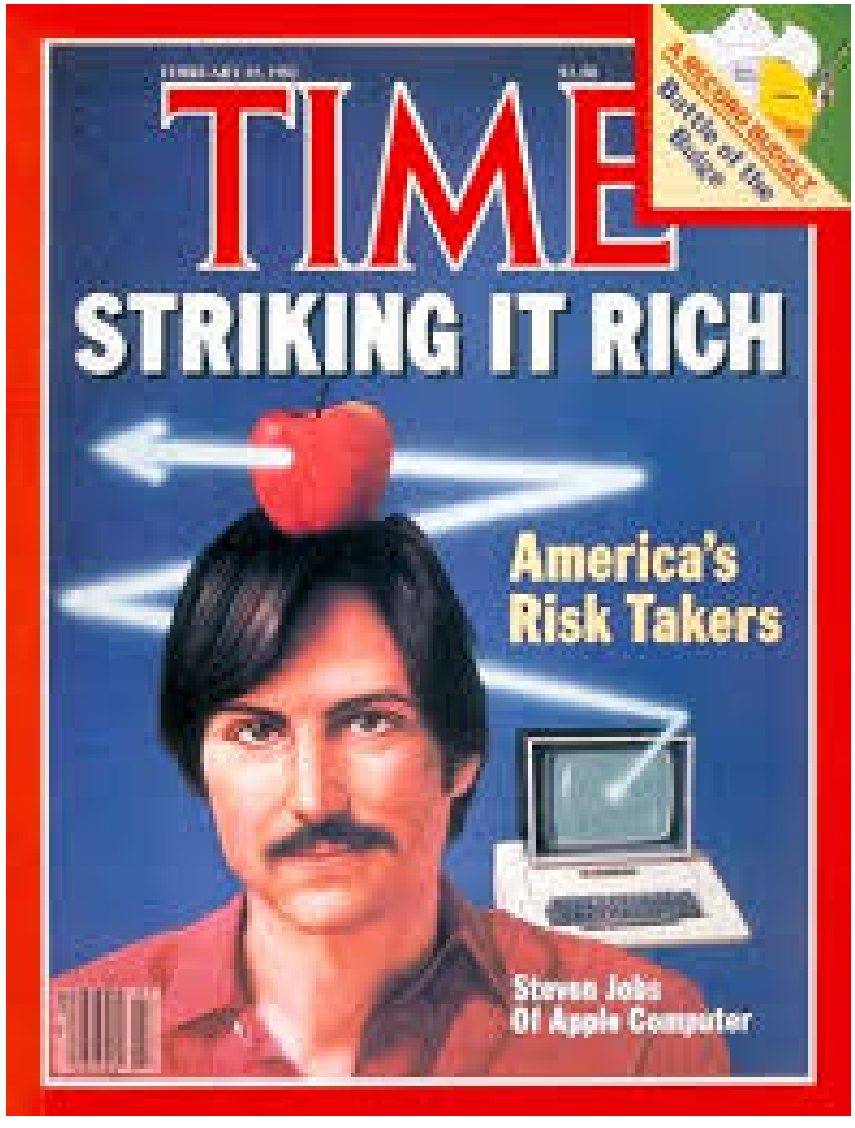

NOVEMBER 17, 1980
$1.50
TIME
ELECTION SPECIAL
A FRESH START
EXCLUSIVE INTERVIEW
WHAT HE'LL DO

Just 9 short years later

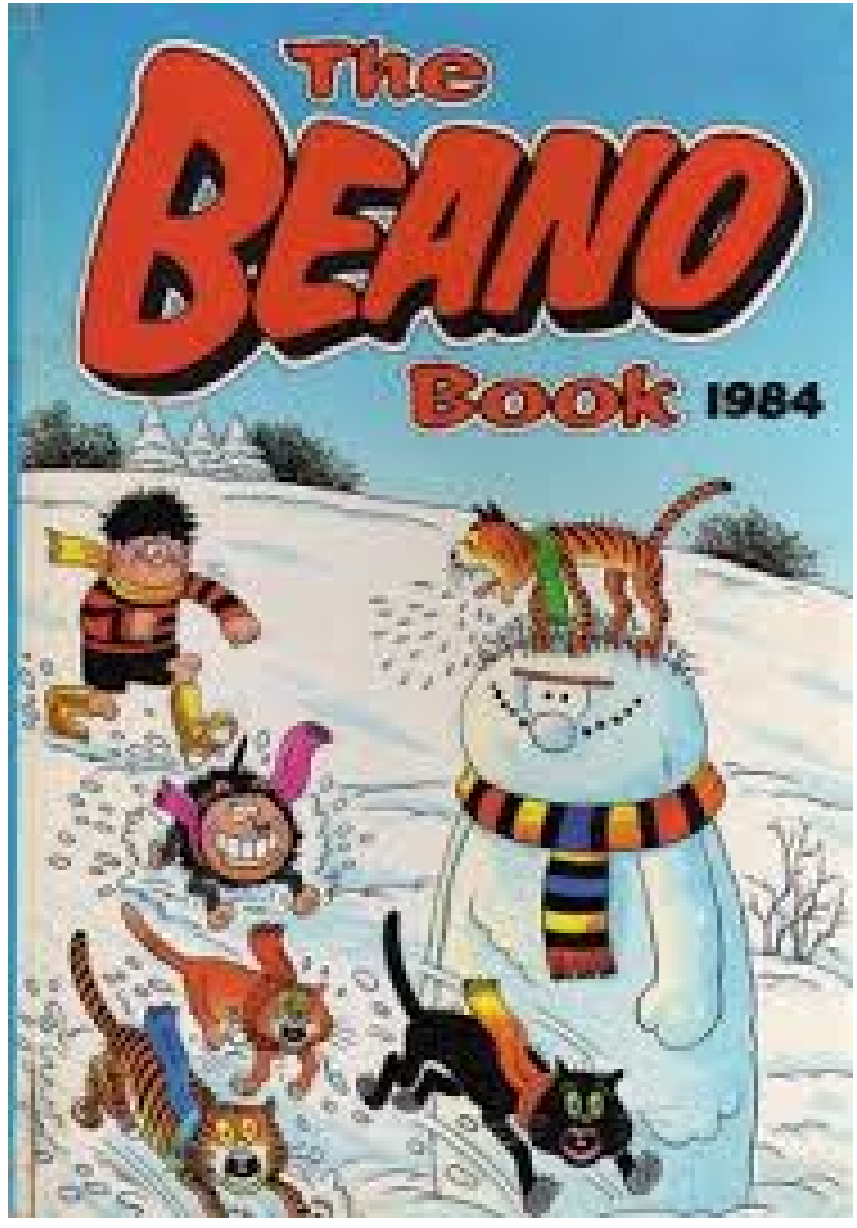
Just 9 short years later

The first automated tool!


TIME
The Prince's Charmer
Lady Diana
Is Wowing
Britain
Greenhouse Poland
A Rainbow


TIME
STRIKING IT RICH
America's
Risk Takers
Steven Jobs
Of Apple Computer


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


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

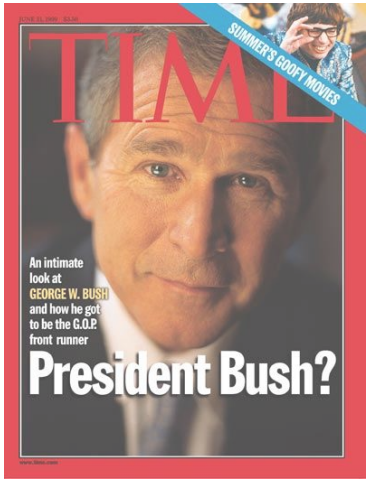
But this is just a **probabilistic** statement

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

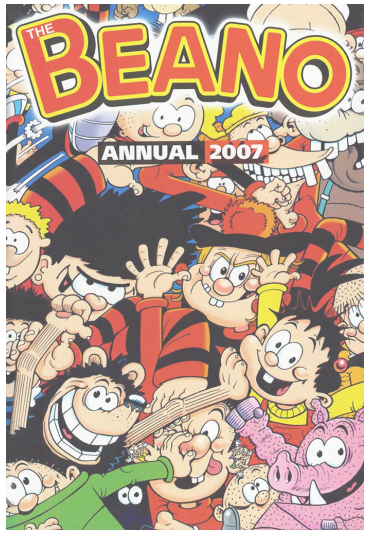
Jester

"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."

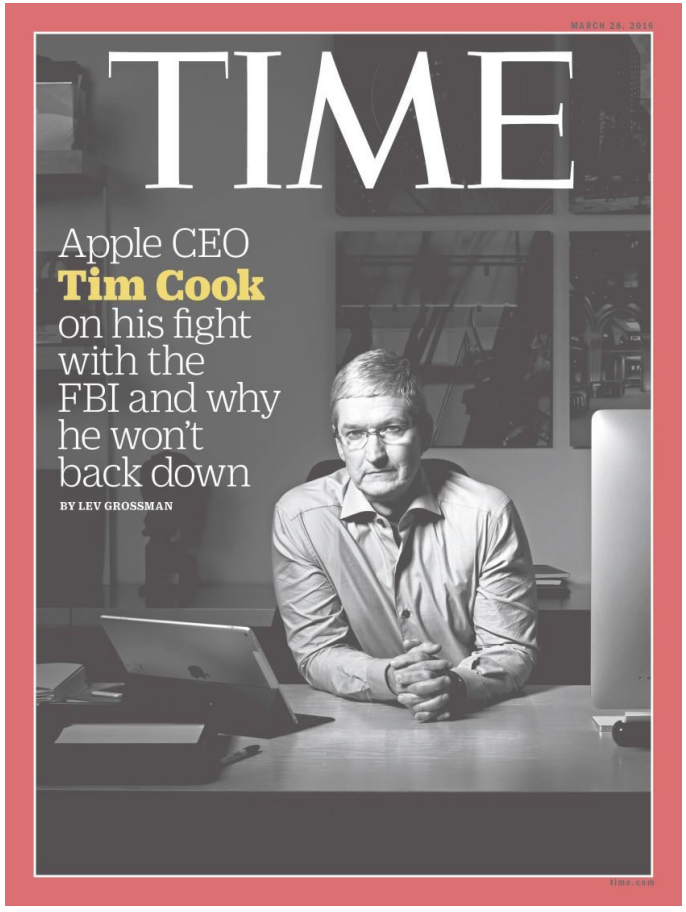
Kent Beck

No-body used it

Lots more research papers

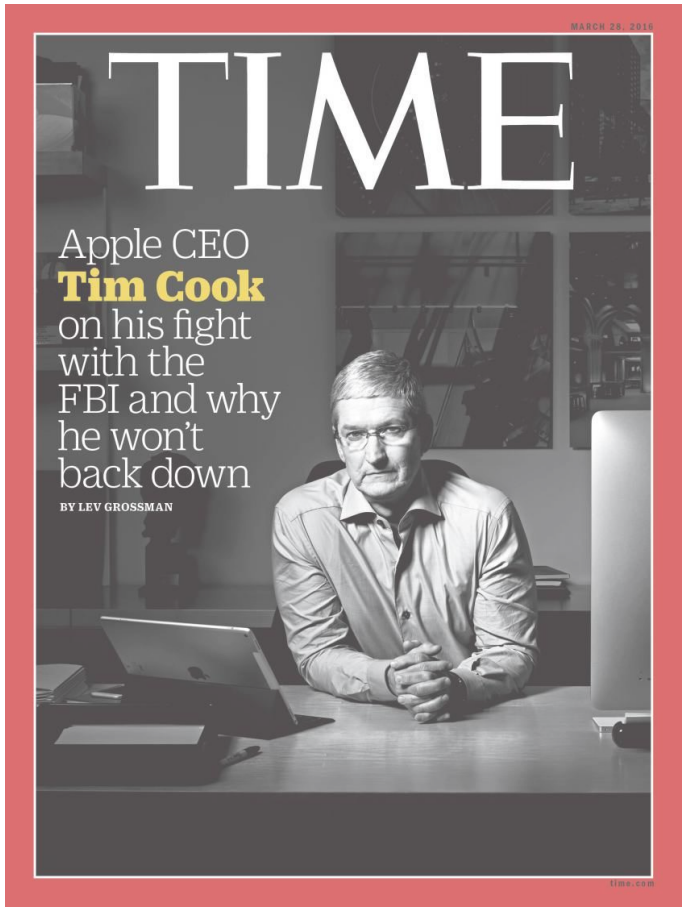


2016

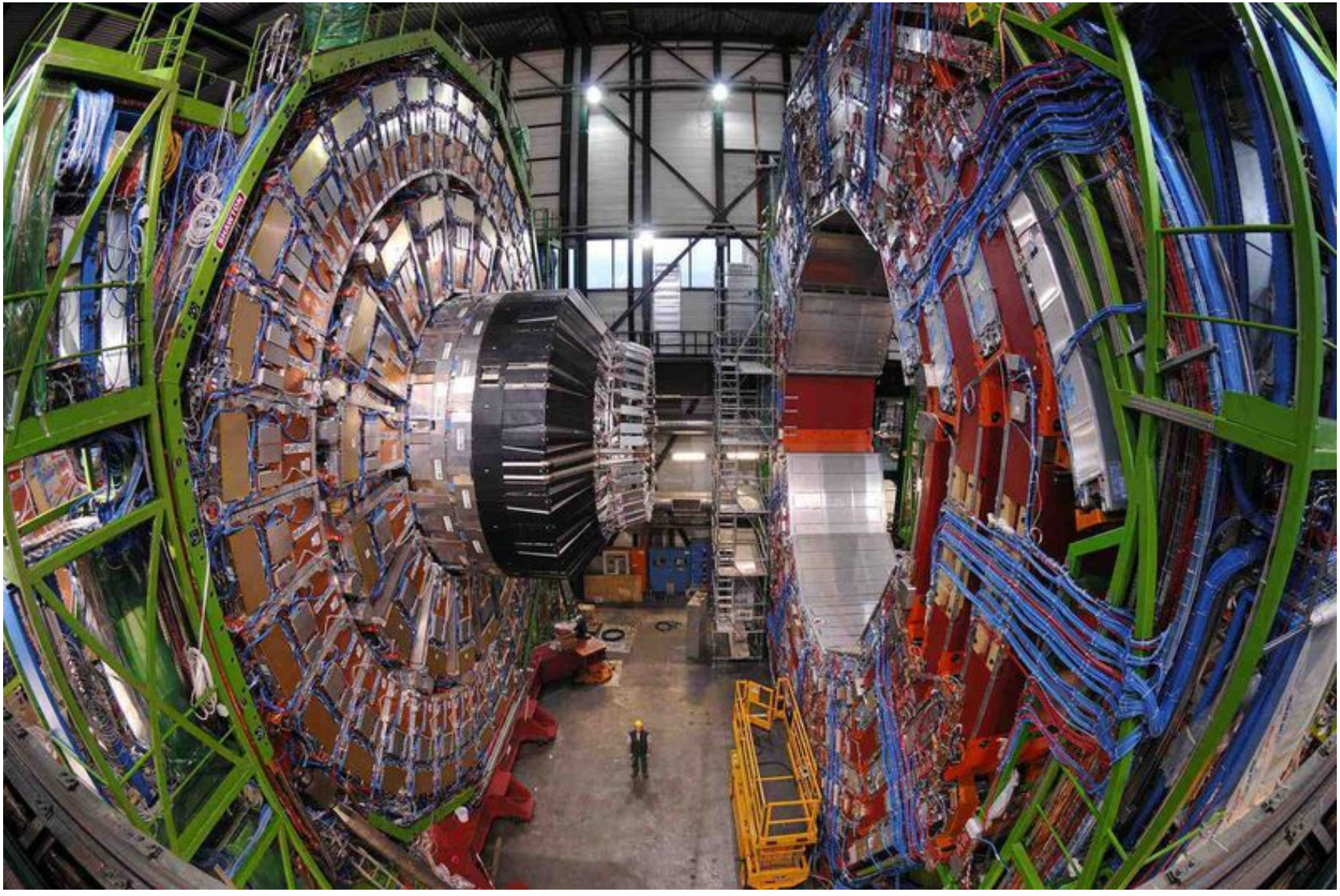

MARCH 28, 2016
TIME
Apple CEO
Tim Cook
on his fight
with the
FBI and why
he won't
back down
BY LEV GROSSMAN
time.com

2016

In daily use all over the world



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. Equivalent mutants

2. Too slow

2. Too slow

- Need to compile the code thousands of times

2. 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) + (160000)

compile test

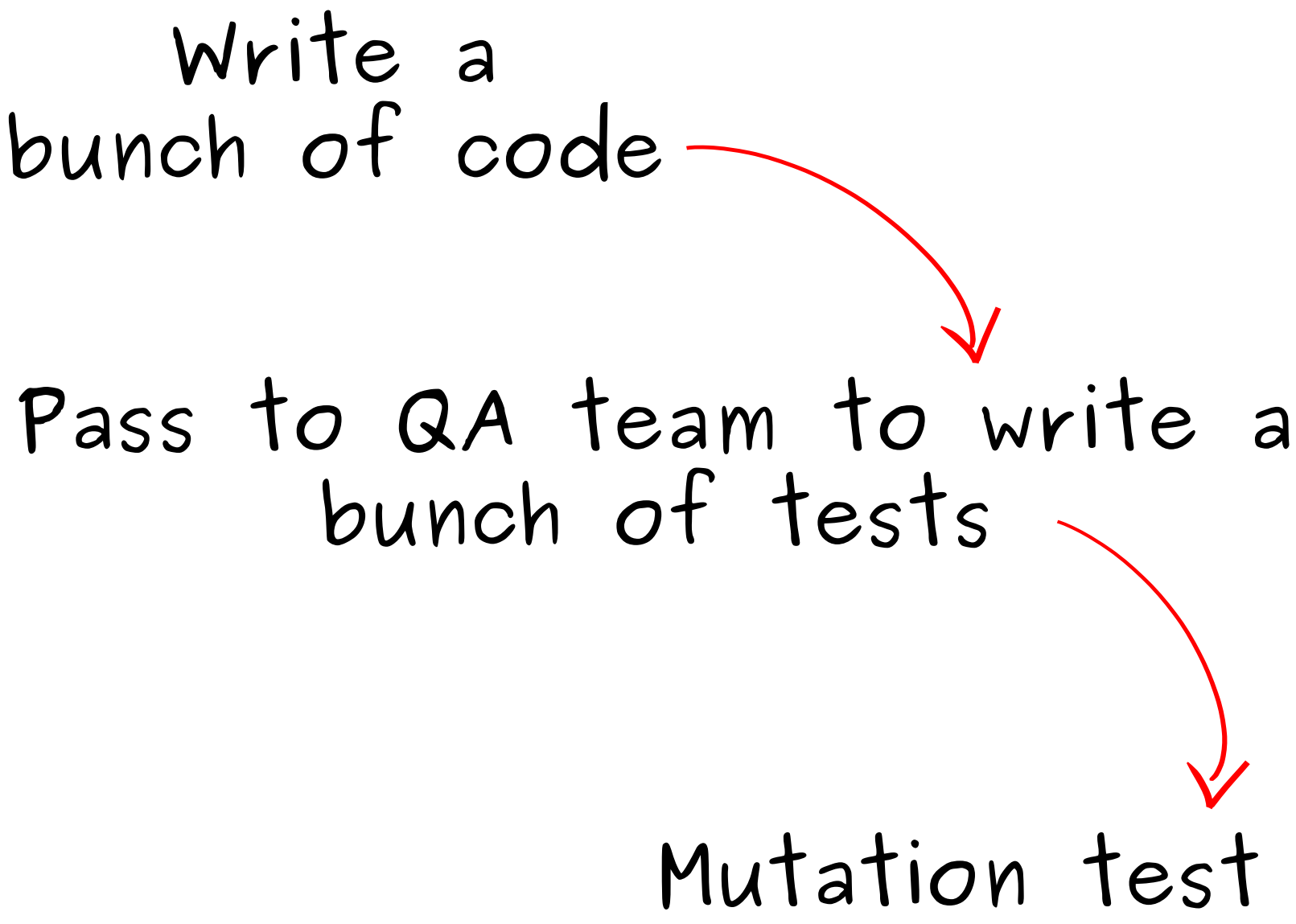
260000 seconds

72 hours

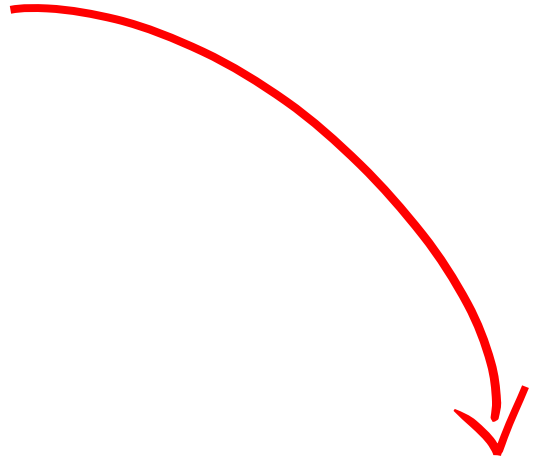
3 days!

So in theory mutation testing is wildly impractical

What happens when we do it in practice?

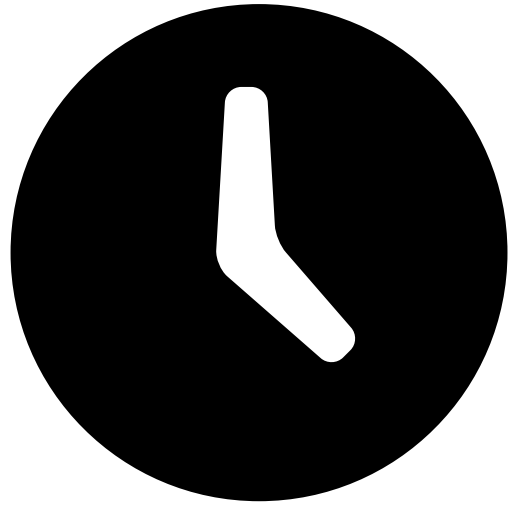

Write a
bunch of code
Pass to QA team to write a
bunch of tests
Mutation test

Write a
bunch of code
and tests

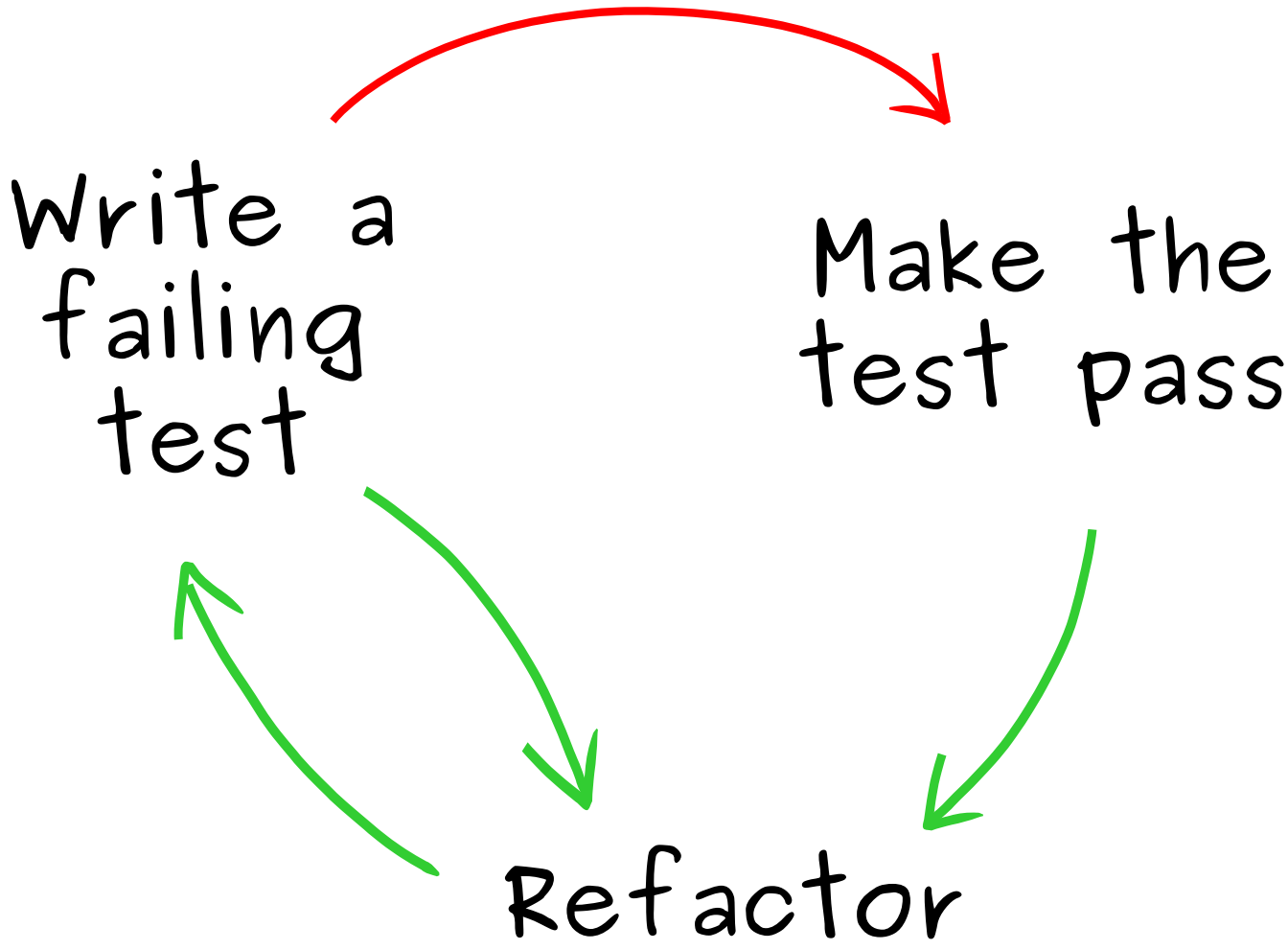


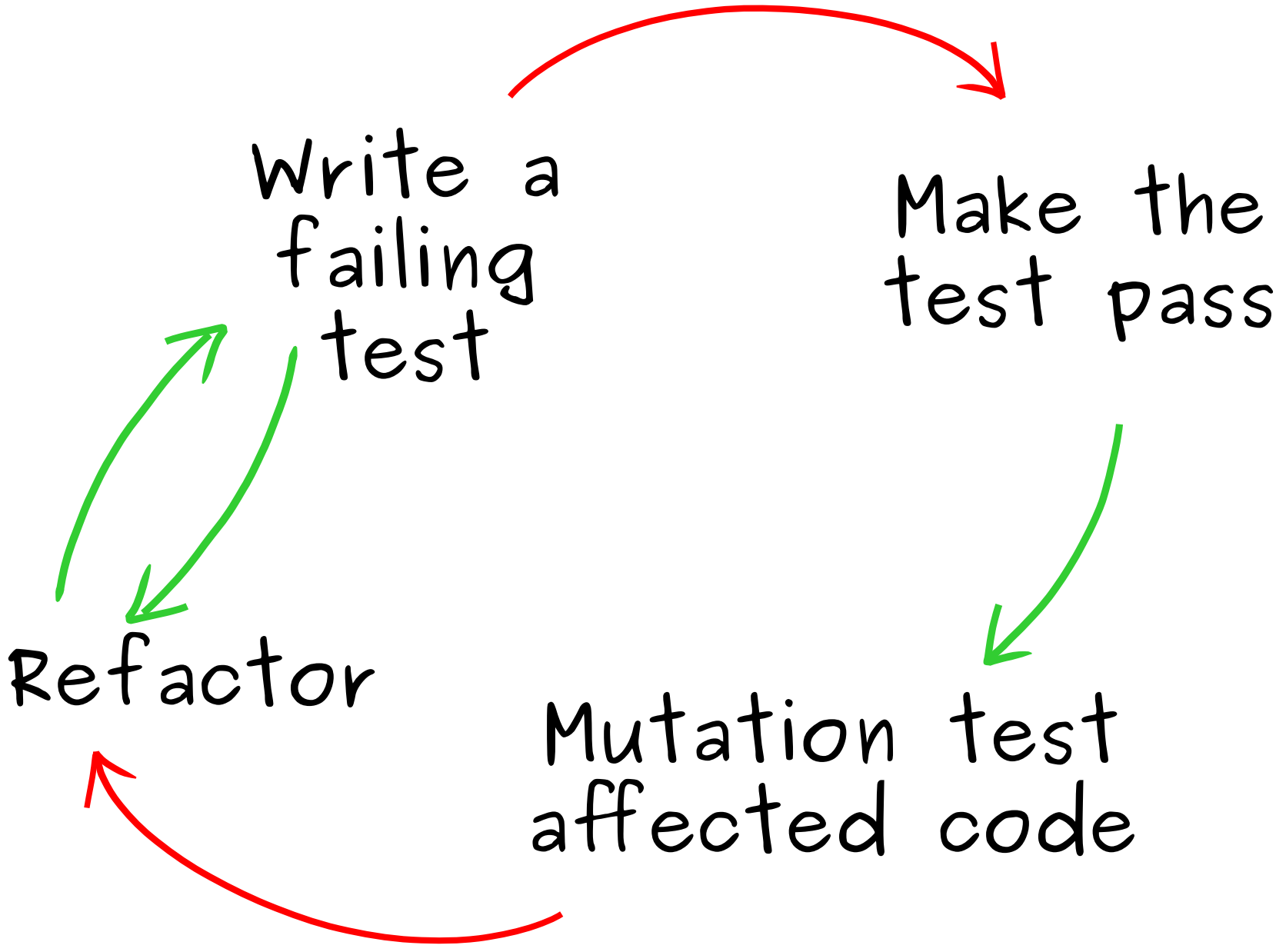
Pass it to a QA
team to mutation test

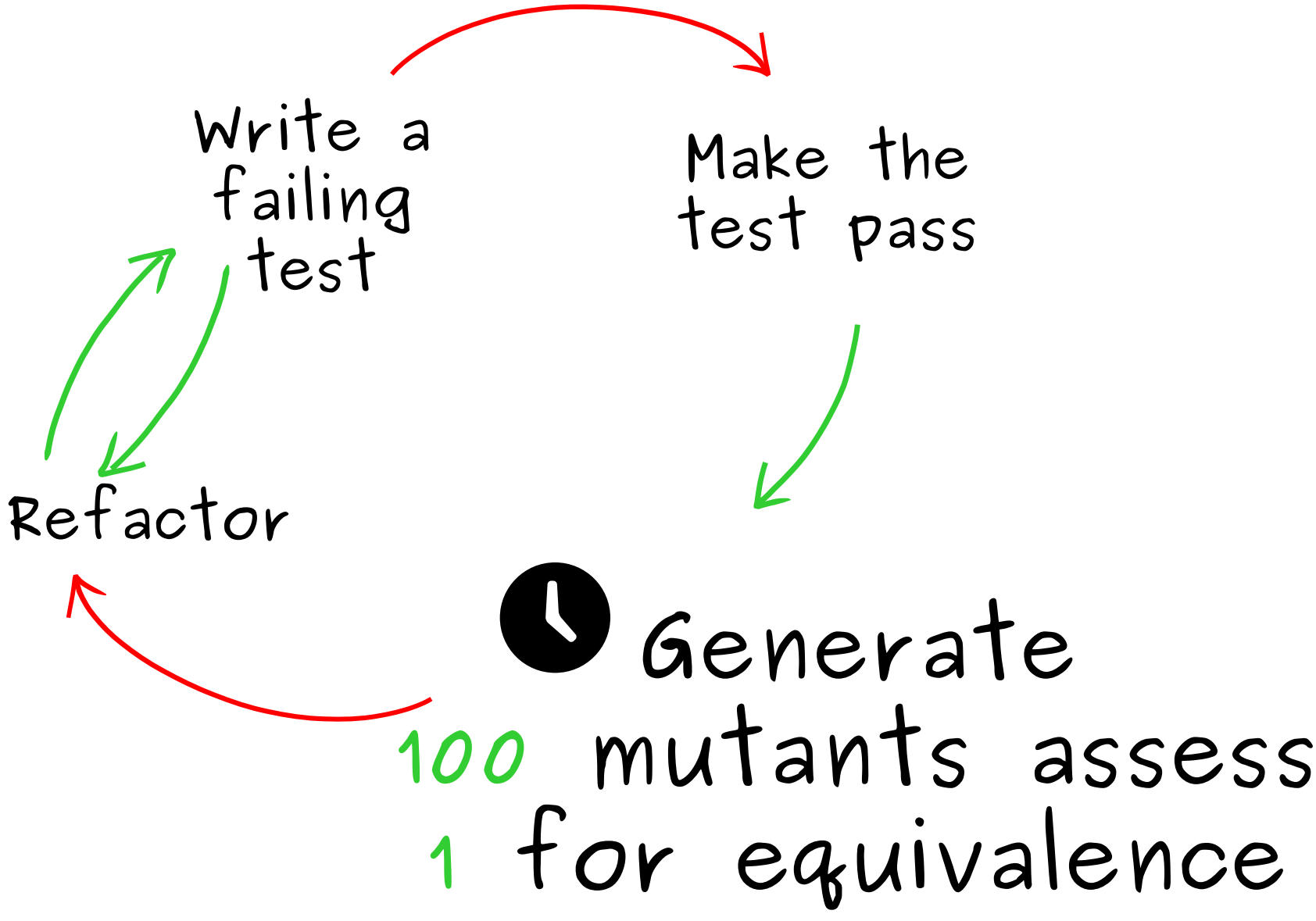
Write a
bunch of code
and tests



Generate 100,000 mutants
assess 10,000 survivors
for equivalence


Write a failing test
Make the test pass
Refactor


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


Write a failing test
Make the test pass
Refactor
Generate 100 mutants assess 1 for equivalence

Configuring the tool

Workarounds

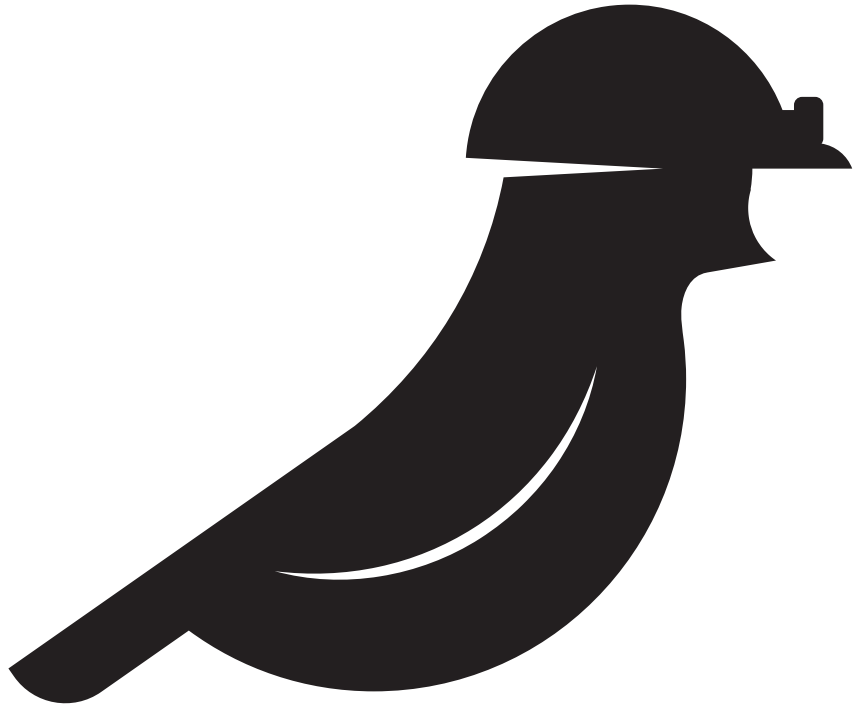
Running mutation analysis

Assessing equivalent mutants

Lots of the time burnt on mundane tooling problems

Around 2010 a new wave of open source projects began to emerge

Pitest (aka PIT)



pitest.org

Mutant

~~Mutagenisis~~ Humbug

They're easy to use

So they were used

Over time became **much** faster than earlier tools

They use coverage data to target tests

```
public class AClass {
  private int count;

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

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

They use coverage data to target tests

```
public class AClass {
  private int count;

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

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

shouldNotCountIntegersBelowTen

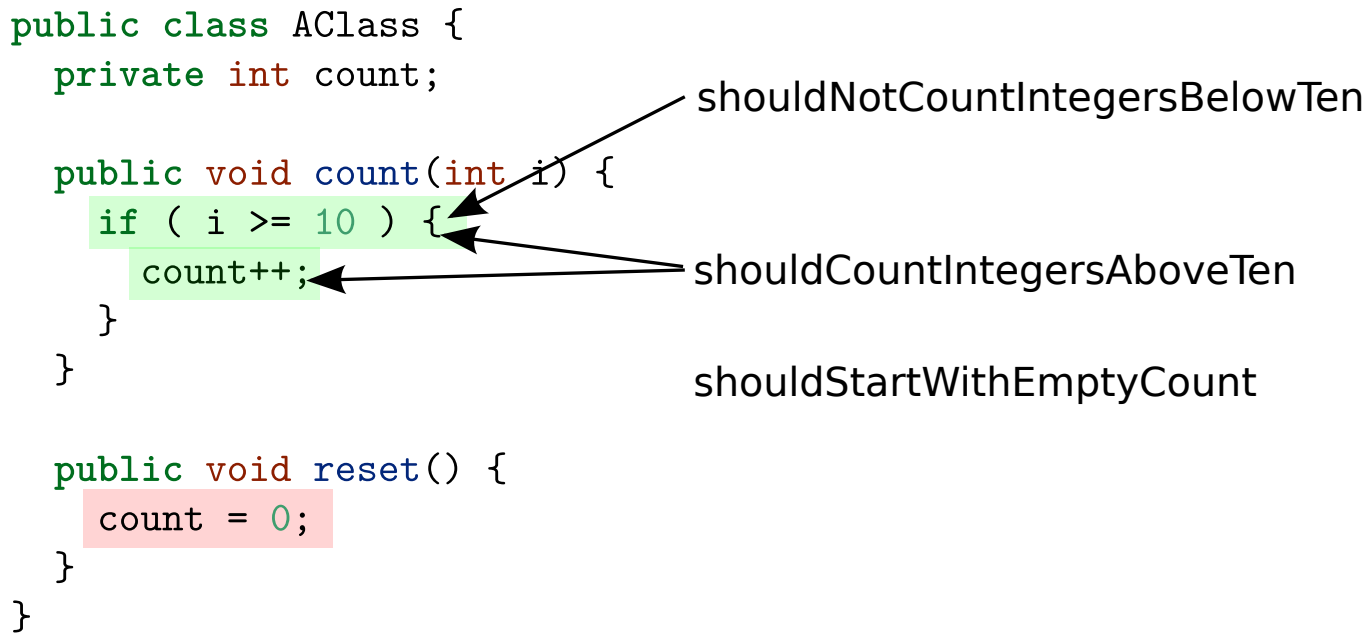
They use coverage data to target tests

```
public class AClass {
  private int count;

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

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

shouldNotCountIntegersBelowTen

shouldCountIntegersAboveTen

They use coverage data to target tests



```
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 }
```

```
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

```
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

```
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 }
```

```
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

```
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**

```
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  }
```

```
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

```
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**

Pitest uses coverage targeting and a few other tricks

Pitest uses coverage targeting and a few other tricks

e.g bytecode manipulation to avoid compile cycles

Can analyse 10k mutants in Joda time in **3 minutes**

Can analyse 10k mutants in Joda time in **3 minutes**

Almost 1000 times faster than earlier tools

Equivalent mutants turned out not to be a **big** problem

In fact they can be helpful

In fact they can be helpful

- Highlight redundant code

In fact they can be helpful

- Highlight redundant code
- Often a cleaner implementation without an equivalent mutant

Mutant introduces them on purpose

Pitest is designed to make them unlikely

How to start mutation testing?

Find a **good** tool

Make it a **development** activity

Make it a **development** activity

Not an after the fact QA step

Mutation test **locally** as you develop

If your project is **small** mutate the whole thing

If it's larger just mutate a slice

If it's larger just mutate a slice

E.g just the code you've **changed**

If it's larger just mutate a slice

E.g just the code you've **changed**

Pitest integrates with version control to make this easy

Mutate on CI server to track trends

Don't have to mutate everything on CI server if project is too big

Maybe just a sample of 1000 [1]

[1] "How hard does mutation analysis have to be anyway?"
http://rahul.gopinath.org/resources/issre2015/gopinath2015howhard.pdf

Maybe just a sample of 1000 [1]

Won't tell you where the gaps are, but will alert you if overall test strength is falling

[1] "How hard does mutation analysis have to be anyway?" http://rahul.gopinath.org/resources/issre2015/gopinath2015howhard.pdf

But whatever happened to . . .

But whatever happened to . . .

But whatever happened to . . .

- Went on to found DNA computing

But whatever happened to . . .

- Went on to found DNA computing
- Contributed (half) of the Karp-Lipton theorem

But whatever happened to . . .

- Went on to found DNA computing
- Contributed (half) of the Karp-Lipton theorem
- Won the Knuth Prize in 2014

@0hjc

http://pitest.org

turns out
it's me that guards the guards