Kill All Mutants!

(Intro to Mutation Testing)

by Dave Aronson

Codosaur.us

@davearonson

(Blank slide so I can flip to a new one to start my timer, ignore this.)

CURRENT TIME: ~22, want ideally 20, up to 25

Kill All Mutants!

(Intro to Mutation Testing)

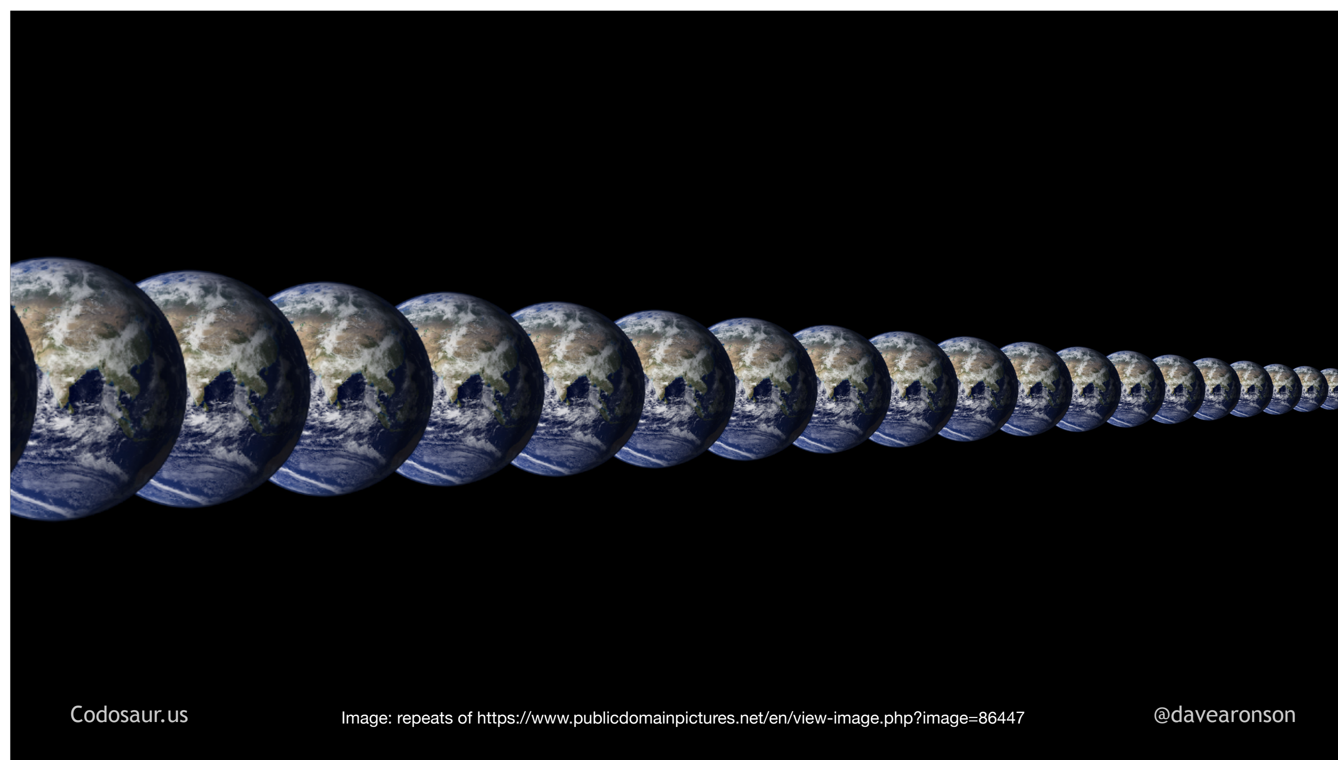
by Dave Aronson

Codosaur.us

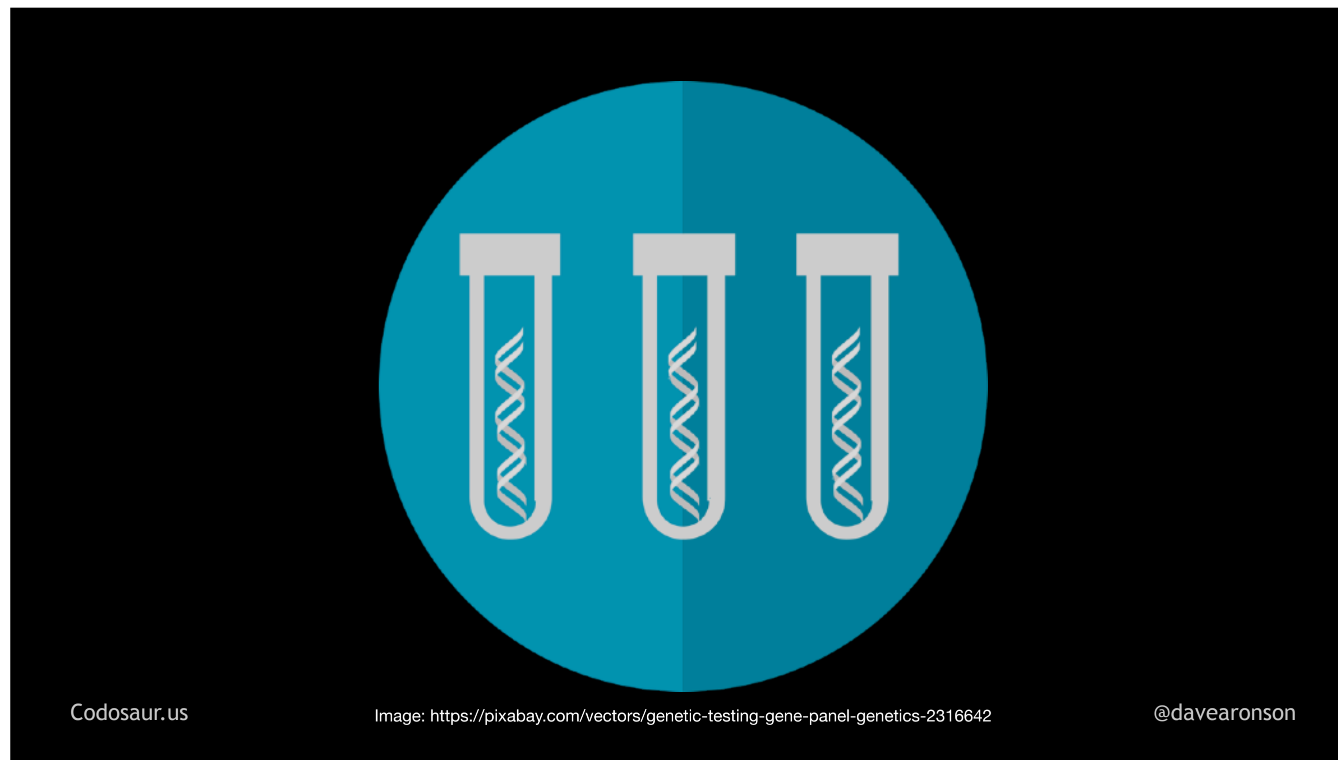
@davearonson

Hi everybody, I’m Dave Aronson, the T. Rex of Codosaurus, and I’m here to teach you to KILL MUTANTS!

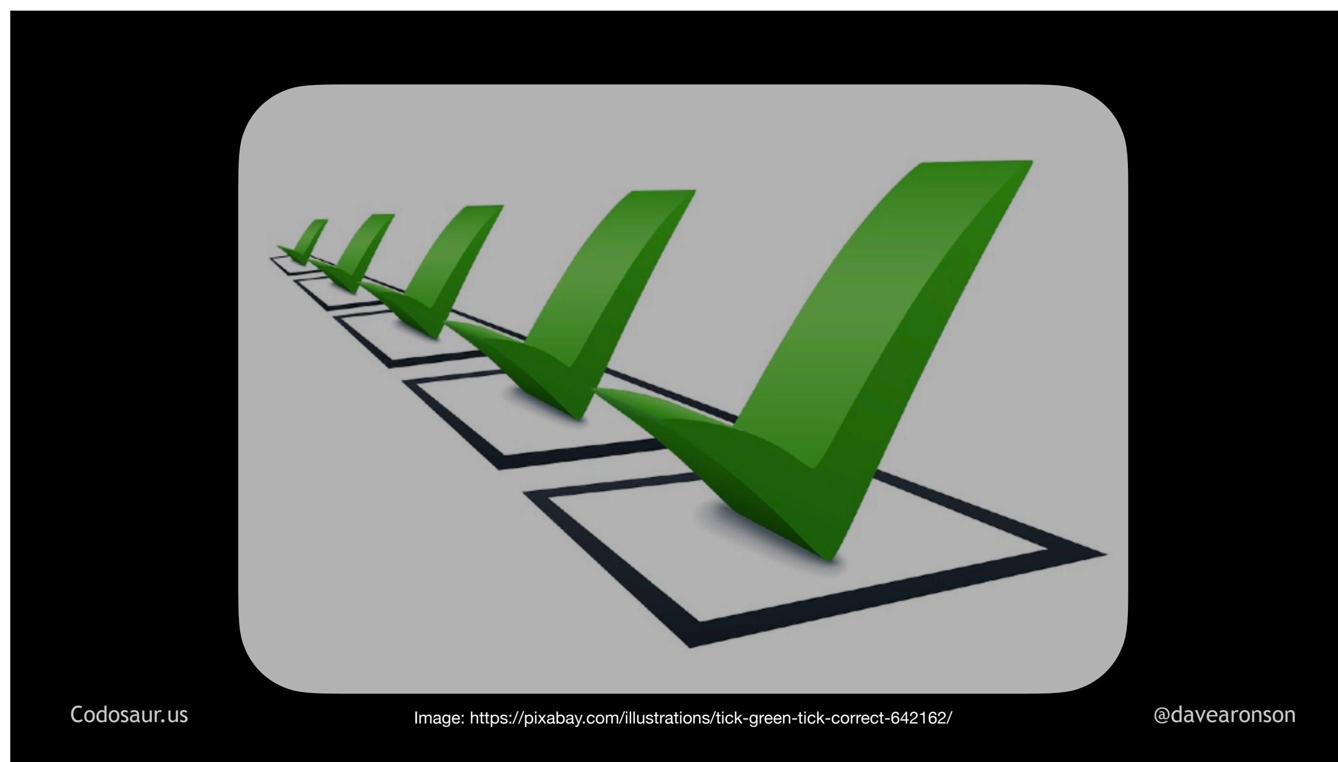
So what on . . .



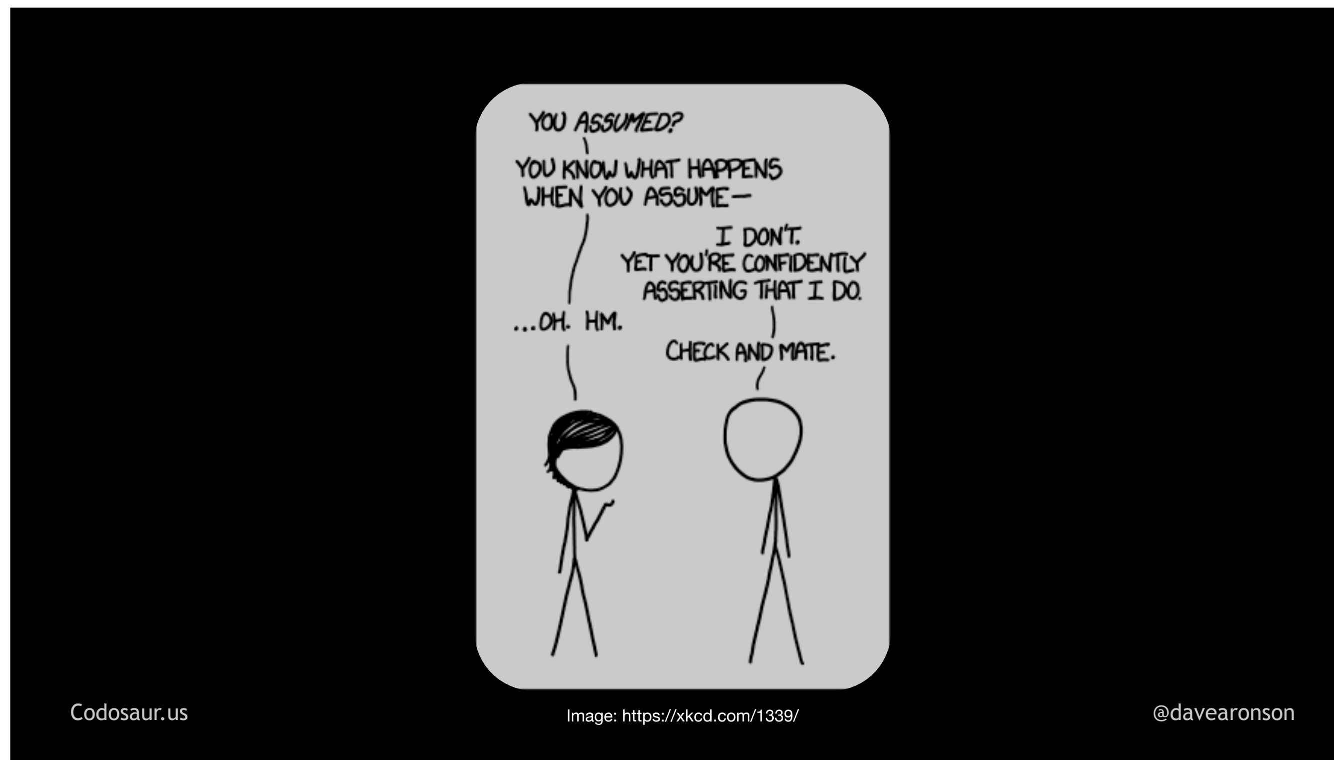
. . . Infinite Earths, makes . . .



. . . mutation testing different from all the *other* software testing techniques? The main difference is that most of the others are about . . .



. . . checking whether our code is correct. But mutation testing . . .



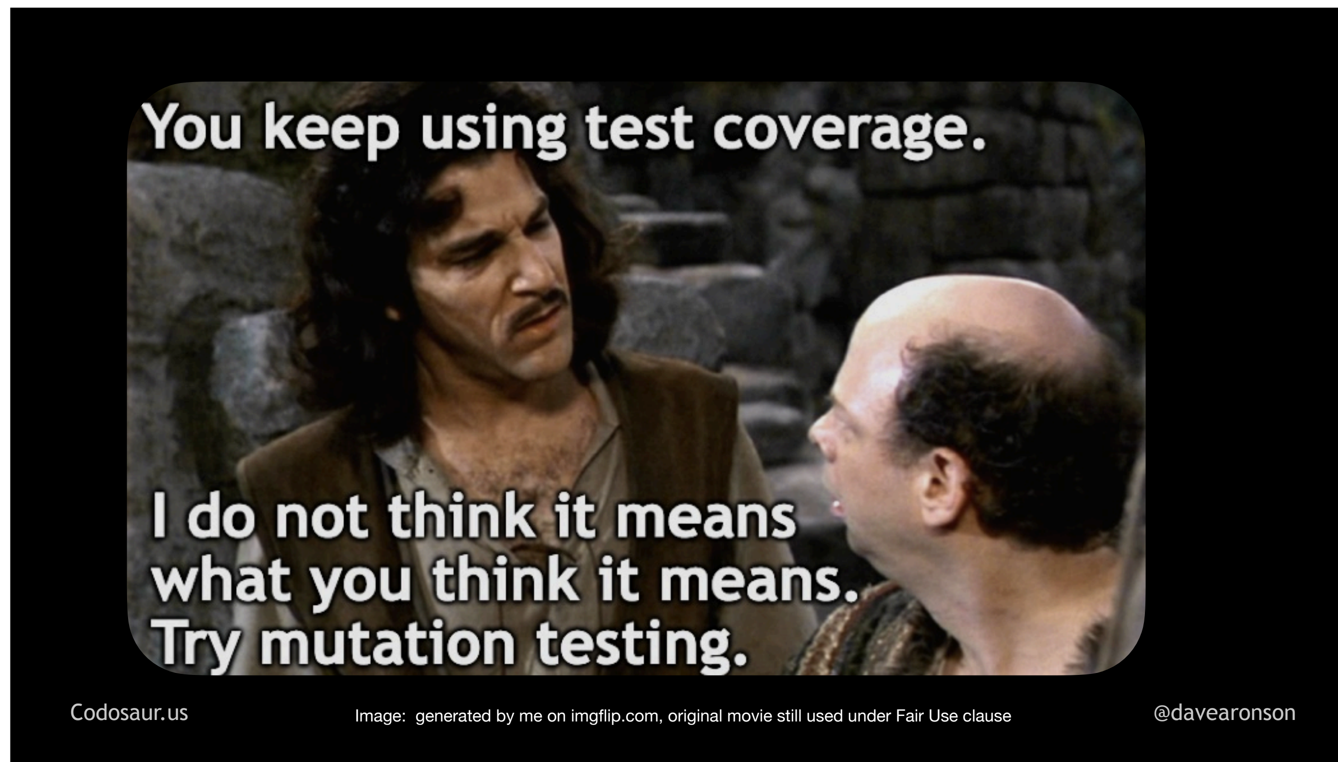
. . . *assumes* that our code is correct, at least in the sense of passing its tests. Instead, mutation testing checks two *other* qualities. In a typical codebase, I think the more *important* one is that our test suite is . . .

"use strict";

Codosaur.us

@davearonson

. . . *strict*. Now you might think that's what test coverage is for, but . . .



. . . no. The *only* thing that test coverage tells us is that at least one test *ran* . . .

```
class Conway:
  ALIVE = "*"
  DEAD  = " "


  @classmethod
  def next_state(cls, cur_state, neighbors):
    if cur_state == cls.ALIVE:
      r = cls.ALIVE if neighbors in [2,3] else cls.DEAD
    else:
      r = cls.ALIVE if neighbors == 3 else cls.DEAD
    return r

  def another_func:
    # whatever
```

Codosaur.us

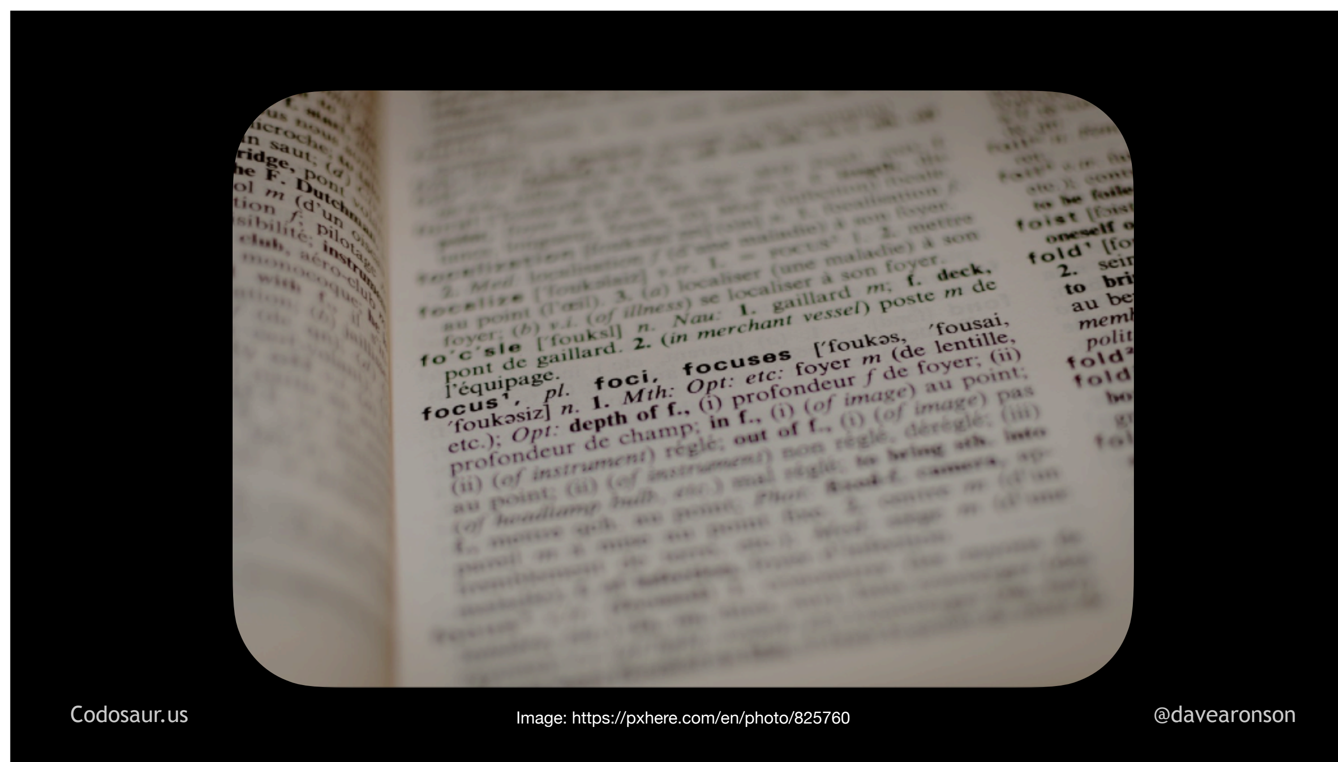
@davearonson

. . . the code it claims is “covered”. It tells us NOTHING about whether the *correctness* of the code *made any difference* to whether any test passed. And that’s what “tested” really *means*, right?

So how *can* we tell if our code really is tested? That’s where mutation testing comes in. To check that our test suite is *strict*, a mutation testing tool will try to . . .

. . . find the gaps in our test suite, that let our code get away with unintended behavior. Once we find gaps, we can close them by either adding tests or improving existing tests. Lack of strictness comes mainly from *lack* of tests, or poorly *written* tests.

The other thing mutation testing checks is that our code is . . .

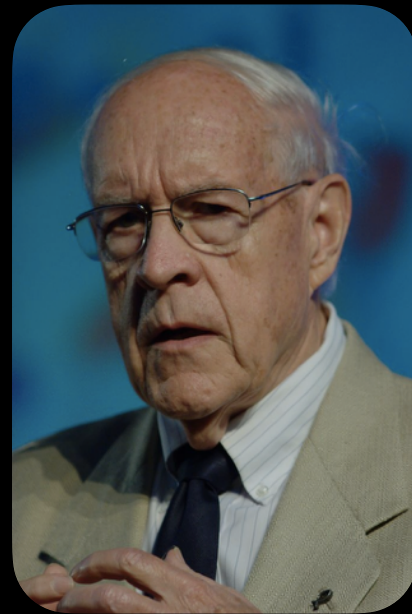


. . . *meaningful*, so that any semantic change to the code will produce a noticeable change in its behavior. Lack of *meaning* comes mainly from code being unreachable, redundant, or otherwise just not having any real effect. When we find "meaningless" code, the usual fix is just to remove it.

Mutation testing . . .

. . . puts these two together, by checking that every change to the code, that the tool knows how to do, does make a noticeable change to its behavior, *and* that the test suite is strict enough that at least one test notices that change, and fails.

That's the positive side, but there are some drawbacks. As . . .



Fred Brooks, author of "No Silver Bullet — Essence and Accident in Software Engineering" (1986 paper)

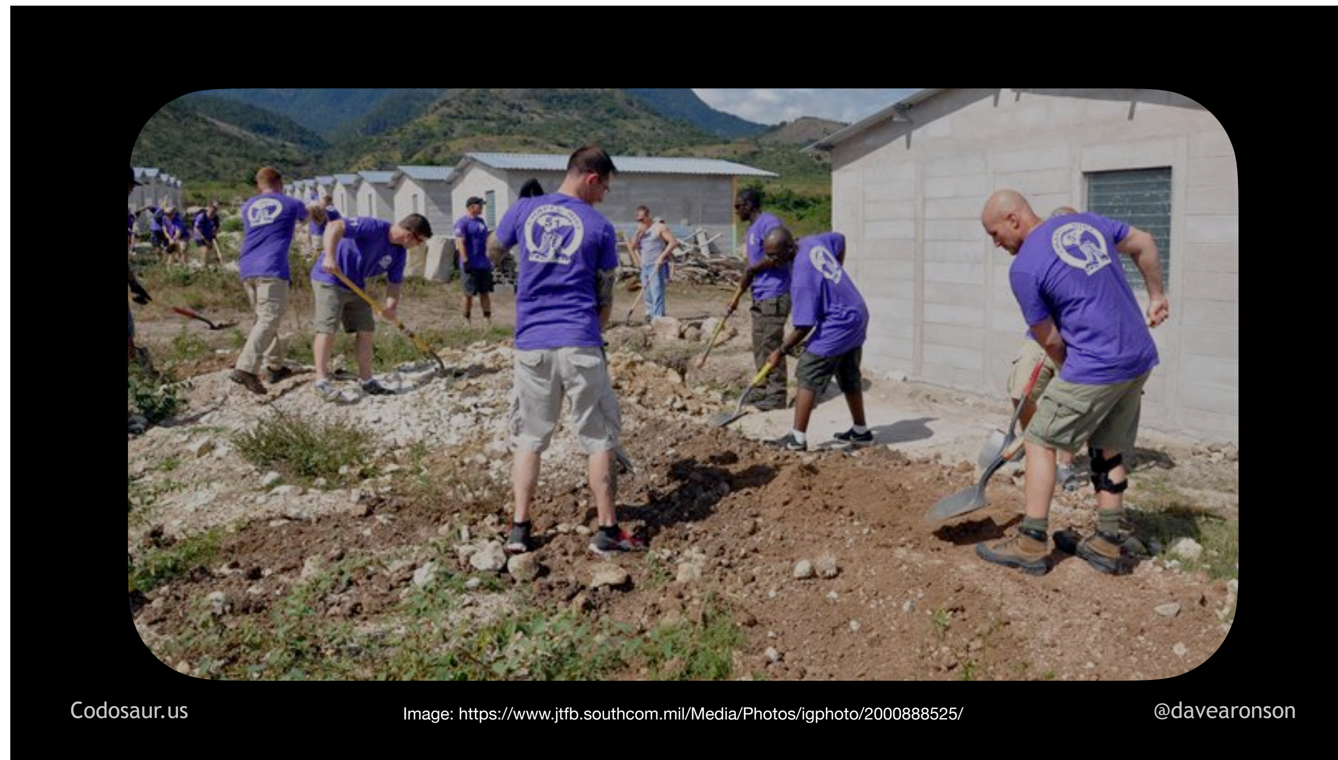
Codosaur.us

Image: https://commons.wikimedia.org/wiki/File:Frederick_Brooks_IMG_2279.jpg

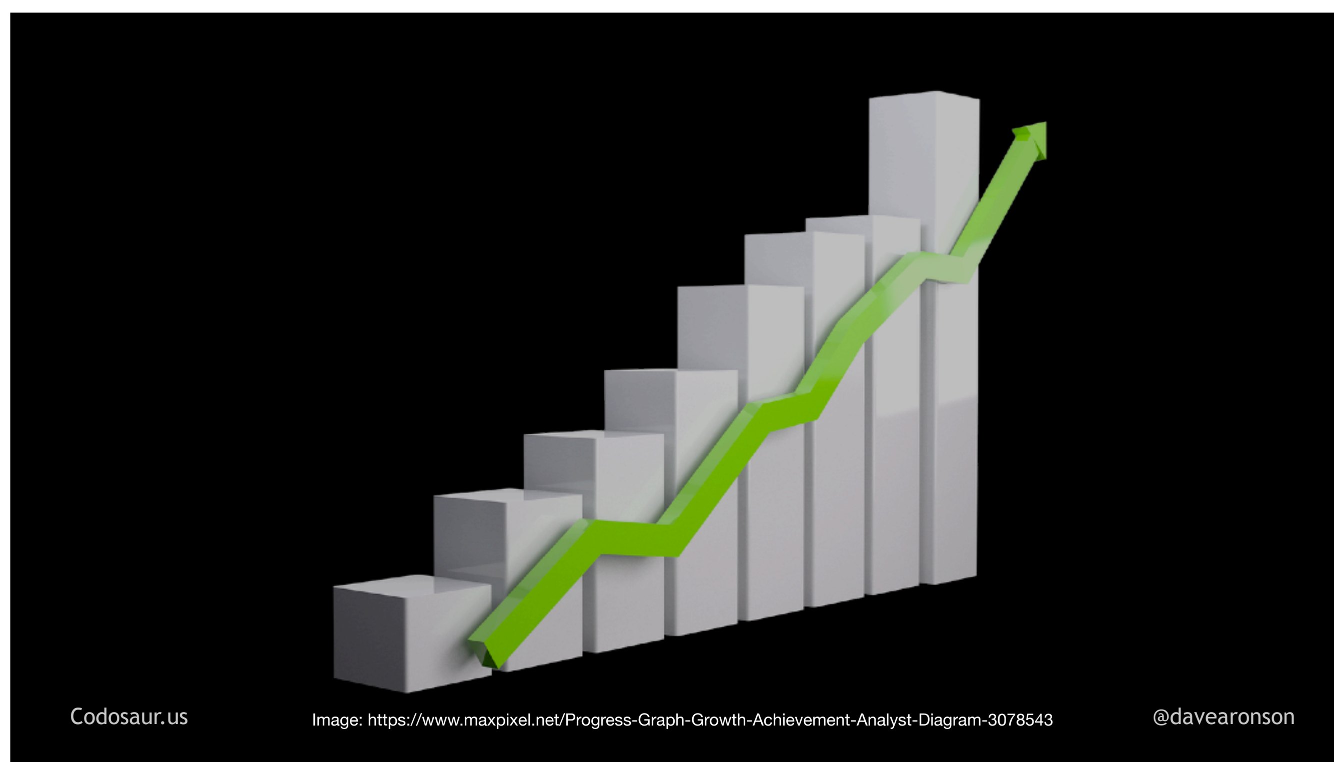
@davearonson

. . . Fred Brooks told us in 1986, there's no . . .

. . . silver bullet! The first drawback is that it's rather . . .



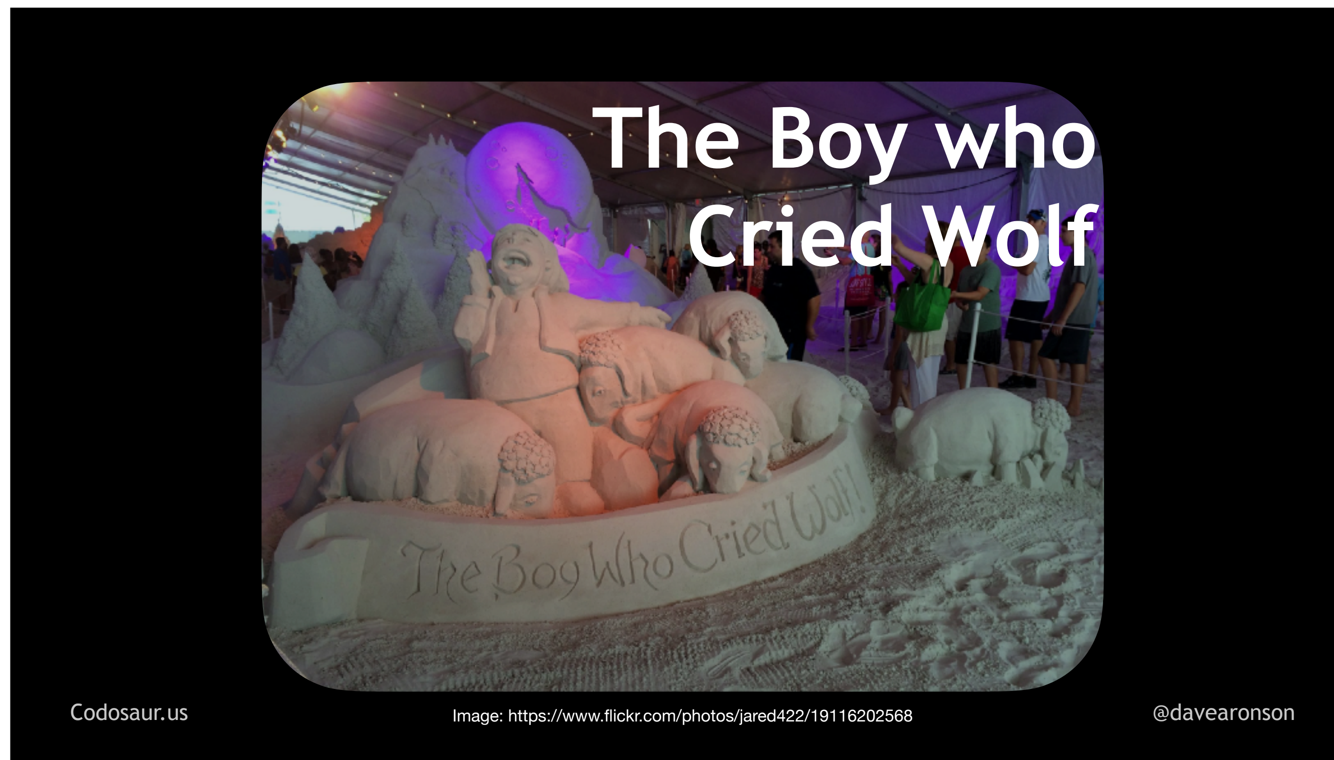
. . . hard labor for the CPU, and therefore usually ra-ther slllooooow. We certainly won't want to mutation-test our entire codebase every time we save a file! Maybe over a lunch break for a smallish system, or a weekend for a large one. Fortunately, most tools let us just check specific functions, classes, files, and so on. Plus, they usually include some kind of . . .



. . . incremental mode, so that we can test only the changes since the last mutation test, or the last git commit, or the changes from the main branch, or some such milestone. With such filtering, maybe we can test just the relevant changes on each save, or at least over a much shorter break.

Another drawback is that it’s often . . .

. . . not at all clear what to do about the results! It tells us that some particular change to the code made no difference to the test results, but what does *that* even mean? It takes a lot of interpretation to figure out what a mutant is trying to tell us. They're *usually* trying to tell us that our code is meaningless, or our tests are lax, or both, but it can be very hard to figure out exactly *how!* Even worse, sometimes it's a . . .



. . . false alarm, because the mutation didn't make a test fail, but it didn't make any actual difference in the first place. It can still take quite a lot of time and effort to figure *that* out.

Now that we've seen some of the pros and cons, what does mutation testing do? It . . .

Point Mutations

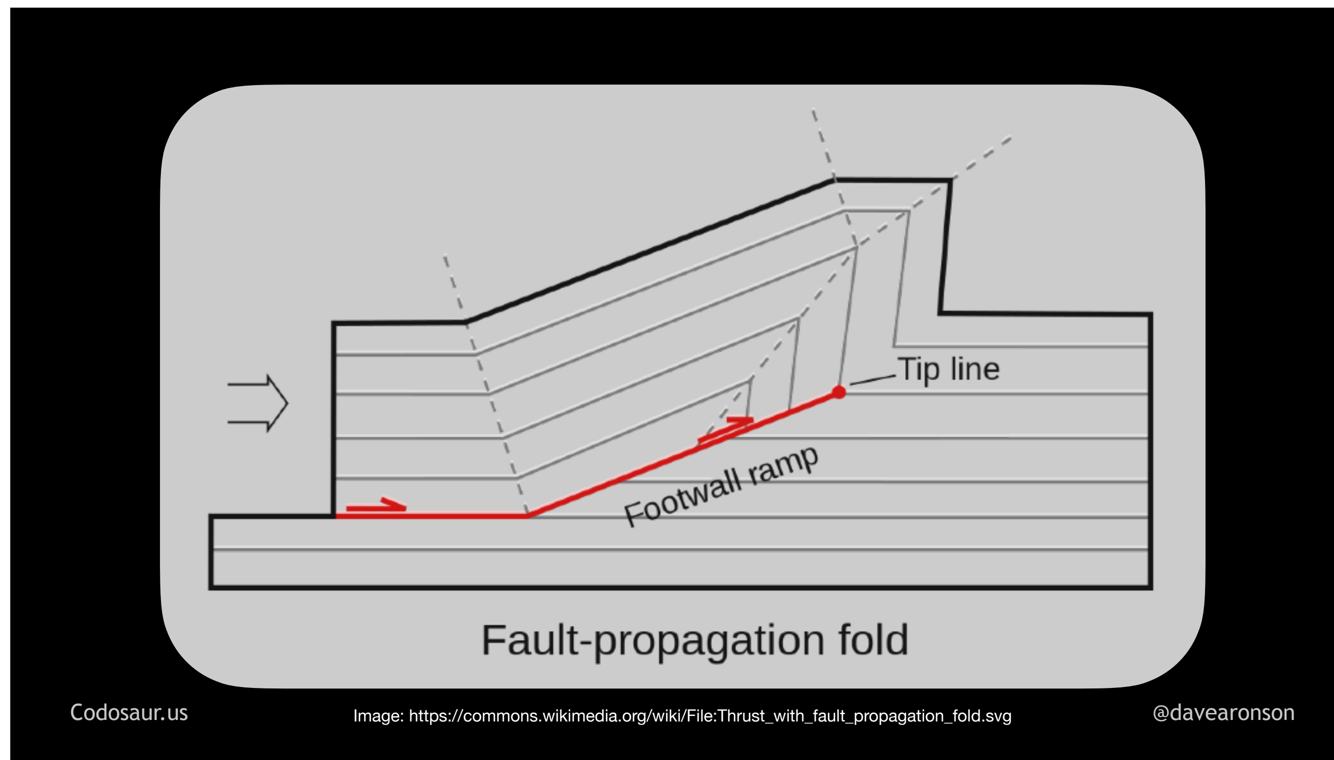
DNA
mRNA
Normal
Missense
Frameshift insertion
Frameshift deletion
Nonsense

Codosaur.us

Image: https://commons.wikimedia.org/wiki/File:Thrust_with_fault_propagation_fold.svg

@davearonson

. . . *mutates* copies of our code, hence the name. It does this to create test failures, also known as . . .



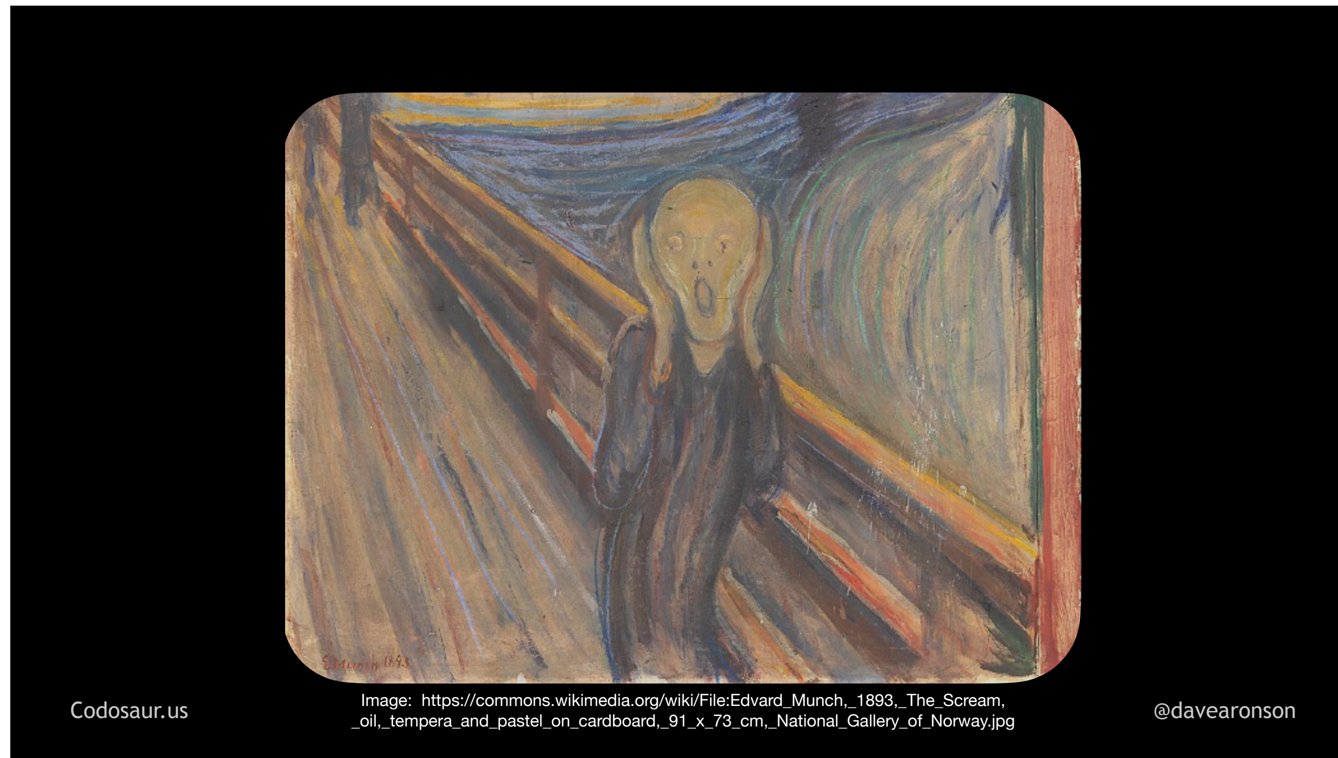
. . . faults. So, mutation testing can be categorized as a *"fault-based"* testing technique, which means that it's related to something you might already know:

. . . Chaos Monkey, from Netflix. But the way mutation testing does it, is sort of . . .

Image: https://github.com/Netflix/chaosmonkey/raw/master/docs/logo.png
(used for educational Fair Use purposes)

. . . upside down from what Chaos Monkey does. Chaos Monkey is best known for . . .

. . . injecting faults into Netflix's production network. (QUICK-CUT TO NEXT SLIDE!)



If all still goes well, in the sense that Netflix's customers don't notice, and their metrics still look good, then Netflix knows that their error recovery is working fine. Mutation testing, however, injects *semantic* . . .

. . . *changes*, not necessarily *problems*. We *hope* each of these changes will create faults, but that depends on the test suite. It injects them *into* . . .

. . . copies of our code, not our actual network, and does this in our . . .

. . . *test* environment, not production. (Whew!) And if everything still goes well, *in the sense that* . . .

```
$ mutation_test
........................................
........................................
........................................
........................................
........................................
........................................
........................................
280 tests, 420 assertions, 4,987 mutants,
0 failures, 0 errors, 0 excluded
```

Codosaur.us @davearonson

. . . our tests all still pass, that *doesn't* mean that all is well, that means that . . .

. . . there *is* a problem! Remember, each change to our code should make *at least* one test *fail*.

So how does mutation testing actually *work?* First, the tool . . .

. . . breaks our code apart into pieces to test, usually our functions. Then, for each one, it makes . . .

. . . mutants from that function. To do that, it looks at it to see how it can be changed, and for each way, the tool makes . . .

. . . one mutant, with *that one mutation*.

Once our tool is done creating all the mutants it can for a given function, it iterates over . . .

. . . that list. And now we get to the heart of the concept.

Mutating function `whatever`, at `something.py:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	⏳						In Progress
2											To Do
3											To Do
4											To Do
5											To Do

Codosaur.us @davearonson

This chart represents the progress of our tool. Most of them don't give us all this info, let alone so organized, but it's a useful *conceptual* model. For each . . .

Mutating function `whatever`, at `something.py:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	⏳						In Progress
2											To Do
3											To Do
4											To Do
5											To Do

. . . mutant, derived from . . .

Mutating function `whatever`, at `something.py:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	⏳						In Progress
2											To Do
3											To Do
4											To Do
5											To Do

. . . a given function, the tool runs the function's . . .

Mutating function `whatever`, at `something.py:42`

Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	⏳						In Progress
2											To Do
3											To Do
4											To Do
5											To Do

. . . tests, but it runs them . . .

Mutating function `whatever`, at `something.py:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	⏳						In Progress
2											To Do
3											To Do
4											To Do
5											To Do

. . . using the *current mutant* in place of the original function.

(PAUSE) If any test . . .

Mutating function `whatever`, at `something.py:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	✘						In Progress
2											To Do
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

. . . *fails*, this is called . . .

. . . “killing the mutant”, and it’s a . . .

. . . *good* thing. It means that our code is *meaningful* enough that the change that the tool made, to *create* this mutant, made a difference in the function's behavior, *and* that at least one test *noticed* that difference, and failed. Then, the tool will . . .

Mutating function whatever, at something.py:42											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	✘						Killed
2											To Do
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

. . . mark that mutant killed, . . .

Mutating function `whatever`, at `something.py:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	✕						Killed
2											To Do
3											To Do
4											To Do
5											To Do

Codosaur.us @davearonson

. . . stop running any more tests against it, and . . .

Mutating function `whatever`, at `something.py:42`

Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	✘						Killed
2	⌛										In Progress
3											To Do
4											To Do
5											To Do

Codosaur.us @davearonson

. . . move on to the next one. Once a mutant has made *one* test fail, we don't care how many more it *could* make fail. Like so much in computers, we only care about ones and zeroes.

On the other claw, if a mutant . . .

Mutating function `whatever`, at `something.py:42`											
Test # Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	✘						Killed
2	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	In Progress
3											To Do
4											To Do
5											To Do

. . . lets all the tests pass, then the mutant is said to have . . .

Mutating function `whatever`, at `something.py:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	✘						Killed
2	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	Survived!
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

. . . *survived*. That means that the mutant has the . . .

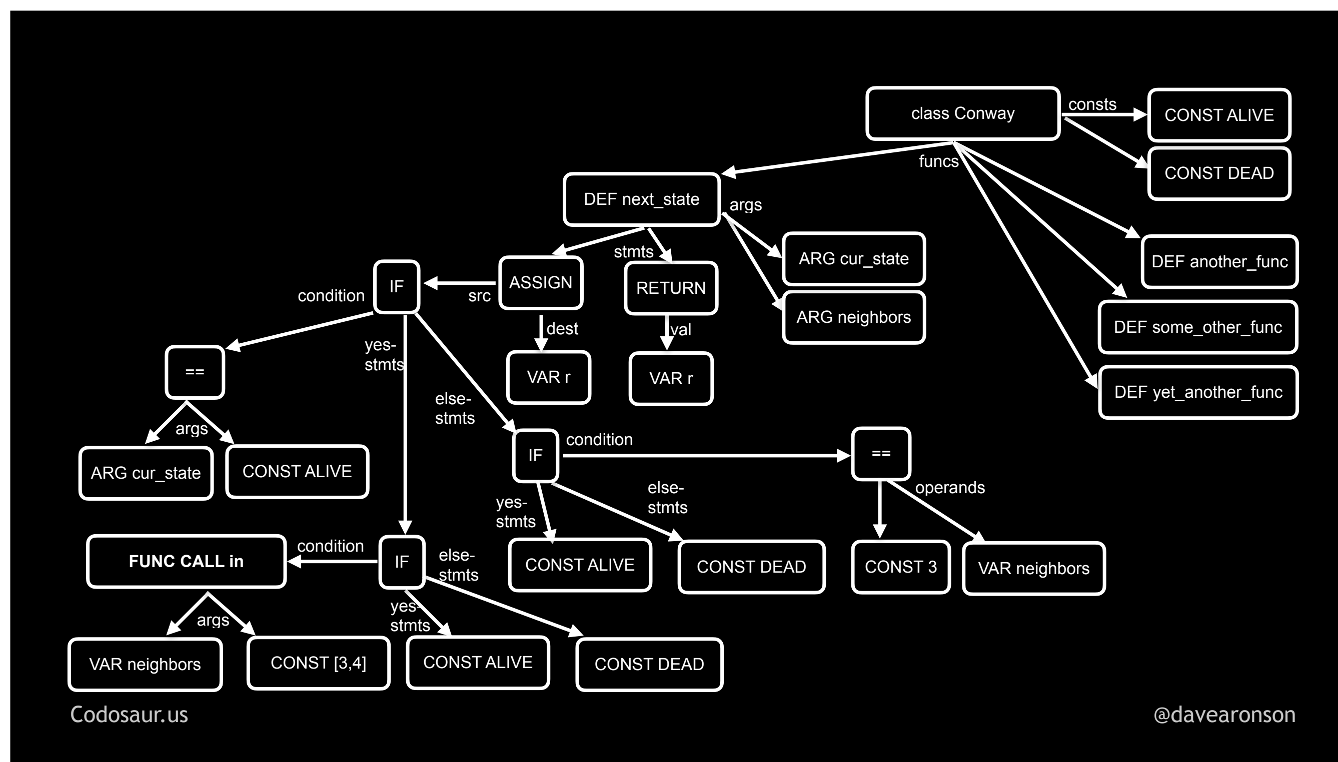
. . . superpower of mimicry, skilled enough to *fool our tests!* This usually means that our code is meaningless, or our tests are lax, or both — and now it's up to us to figure out how.

Now let's peel back one . . .

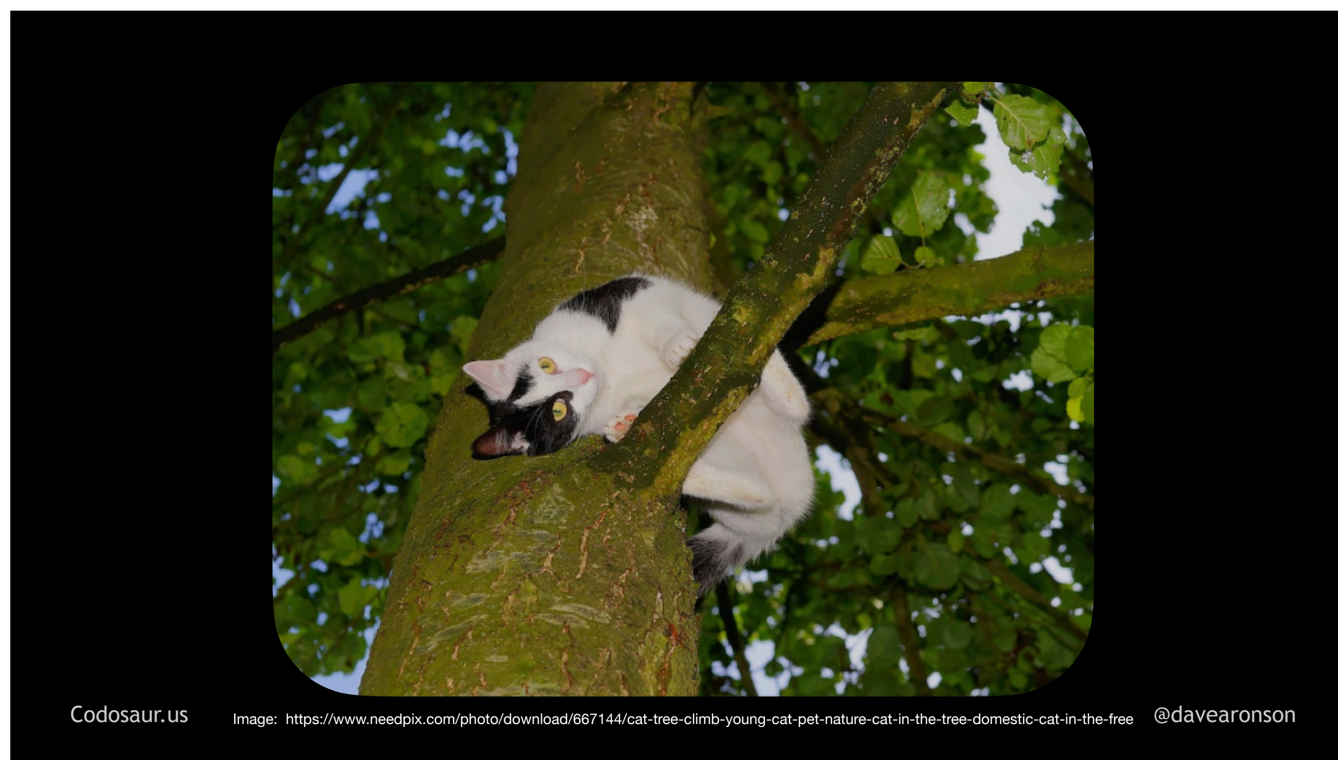
. . . layer of the onion, and look at some *technical details* of how this works. First, our tool parses . . .

```
class Conway:
  ALIVE = "*"
  DEAD  = " "

  @classmethod
  def next_state(cls, cur_state, neighbors):
    if cur_state == cls.ALIVE:
      result = cls.ALIVE if neighbors in [2,3] else cls.DEAD
    else:
      result = cls.ALIVE if neighbors == 3 else cls.DEAD
    return result

  def another_func:
    # whatever
  def some_other_func:
    # whatever
  def yet_another_func:
    # whatever
```

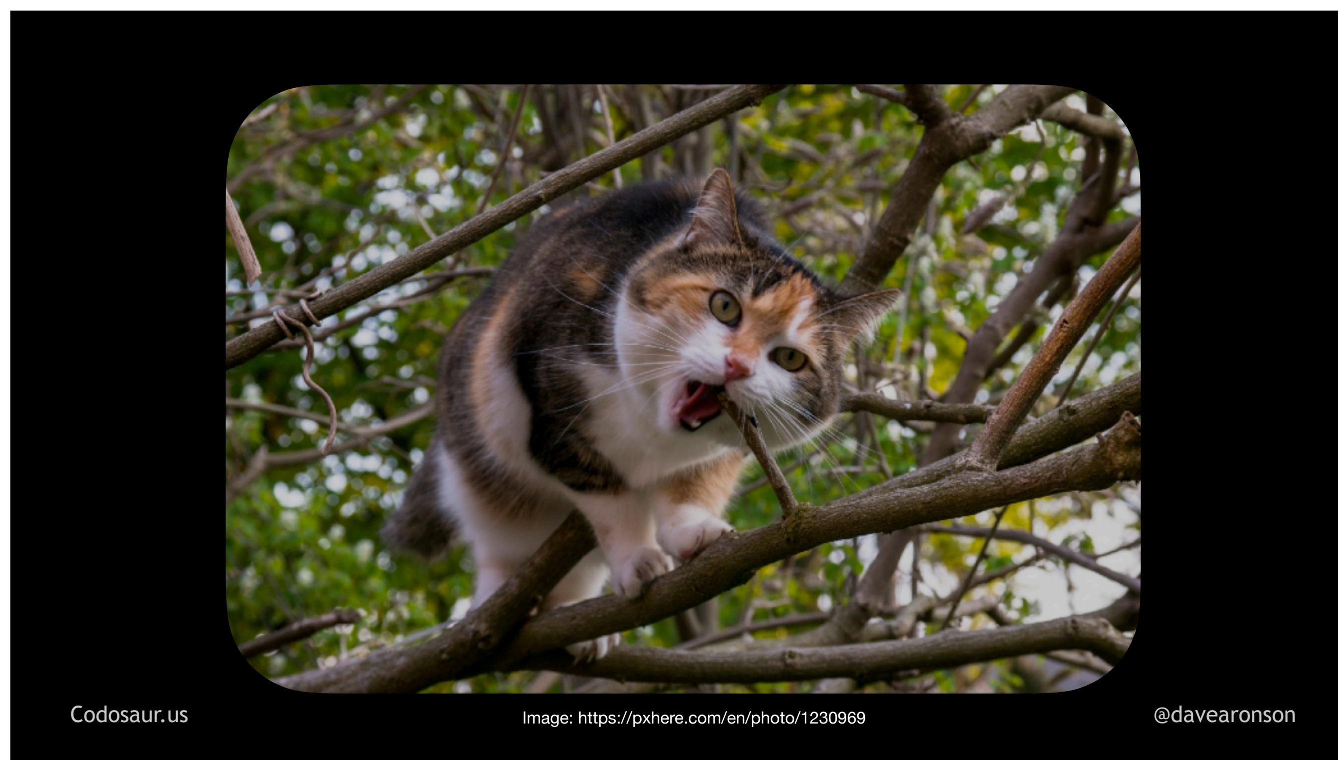

. . . our code, so this code (which you don’t really need to read and understand) becomes . . .



. . . this Abstract Syntax Tree (which you also don't really need to understand, except to notice that it includes some function definitions, one of which I've fleshed out in full). Then it . . .



. . . traverses the tree, looking for sub-trees, or branches if you will, that represent each function. From each one, it makes mutants. To make mutants *from* an AST subtree, it . . .



. . . traverses that subtree, just like it did to the whole thing. However, now, instead of looking for even *smaller* subtrees it can *extract*, like twigs or something, it's looking for *nodes* where it can *change* something. Each time it finds one, then for each way it can change that node, it makes one copy of the function's AST subtree, with that one node changed, in that one way.

Now, I've been talking a lot about changing things, so what kind of changes are we talking about? There are quite a lot!

```
x + y    could become: x - y
                       x * y
                       x / y
                       x ** y
x || y   could become: x && y
                       x ^ y
x | y    could become: x & y
                       x ^ y
```

Maybe even swap *between* sets!

Codosaur.us

@davearonson

It could change a mathematical, logical, or bitwise operator from one to another.

`x - y` could *also* become `y - x`

`x / y` could *also* become `y / x`

`x ** y` could *also* become `y ** x`

`"x" + "y"` could *also* become `"y" + "x"`

Codosaur.us

@davearonson

When the *order* of operands matters, it could *swap* them.

```
x < y
```

could become:

```
x <= y
x == y
x != y
x >= y
x >  y
```


It could change a *comparison* from one to another.

```
if x == y:
  foo(z)
```

could become:

```
foo(z)
```

Codosaur.us @davearonson

It can remove an if-condition . . .

```
while x == y:
  foo(z)
```

could become:

```
foo(z)
```

Codosaur.us

@davearonson

. . . or a looping condition.

```
def f(x, y): # lots of code here
could become:
def f(x, y): return 0
def f(x, y): return :math.max_int
def f(x, y): return "a string"
def f(x, y): return nil
def f(x, y): return x  # or y
def f(x, y): fail("kaboom")
def f(x, y): # nothing
etc.
```

Codosaur.us

@davearonson

It could replace a function's *entire contents* with returning a constant, or any of the arguments, or raising an error, or nothing at all.

42 could become:

```
43
41
-42
1
0
-1
42.1
41.9
```

```
"42"
[42]
{42}
[]
()
{}
None
```

```
math.min_int
math.max_int
math.min_float
math.max_float
math.infinity
math.epsilon
```

and many more

Codosaur.us

@davearonson

It could change a value to some other value, even changing it to an incompatible type, such as changing a number into a, if I may quote . . .

. . . Smeagol, “string, or nothing!”

There are *many* more types of changes, but I trust you get the idea!

There are no more low-level details I want to add, so let’s *finally* walk through some *examples!* We’ll start with an easy one. Suppose we have a function . . .

```
def power(x, y):
  x ** y
```

Codosaur.us @davearonson

. . . like so.   Never mind *why*, it just makes a good simple example, so let’s roll with it.

Think about what a mutant made from this might *return*. Mainly, it could return results such as . . .

```
x + y
x - y
x * y
x / y
y ** x
x
y
0
1
-1
0.1
-0.1
```

```
math.min_int
math.max_int
math.max_float
math.min_float
math.infinity
math.epsilon
raise(DeliberateError)
"some random string"
[]
()
{}
None
```

and many more

Codosaur.us

@davearonson

. . . any of *these* expressions or constants, and many more but I had to stop somewhere.

Now suppose we had only one test . . .

```
assert power(2, 2) == 4
```

Codosaur.us @davearonson

. . . like so. This is a rather poor test, and I think at least one reason why is clear to most of us, but even so, *most* of those mutants on the previous slide *would get killed* by this test, the ones shown . . .

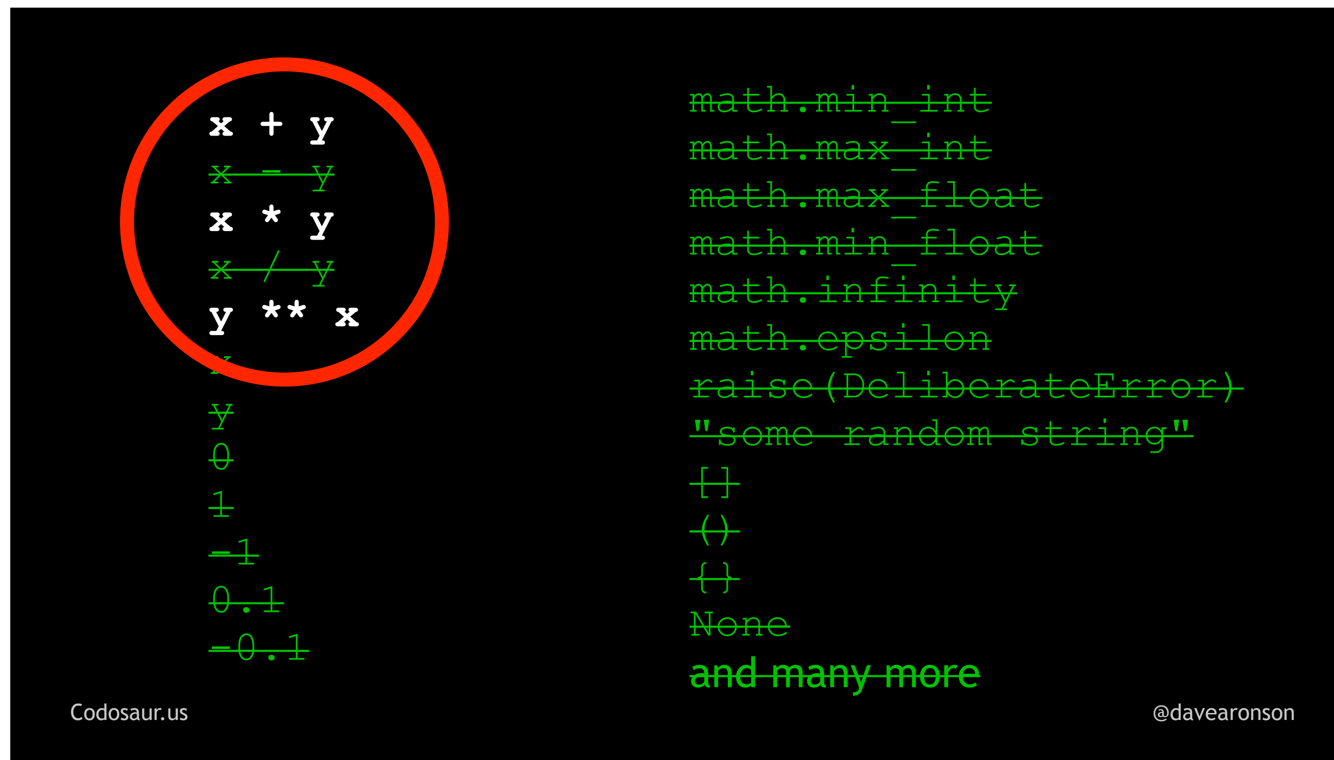
x + y
~~x - y~~
x * y
~~x / y~~
y ** x
~~x~~
~~y~~
~~0~~
~~1~~
~~-1~~
~~0.1~~
~~-0.1~~

~~math.min_int~~
~~math.max_int~~
~~math.max_float~~
~~math.min_float~~
~~math.infinity~~
~~math.epsilon~~
~~raise(DeliberateError)~~
~~"some random string"~~
~~[]~~
~~()~~
~~{}~~
~~None~~
~~and many more~~

Codosaur.us

@davearonson

. . . here in crossed-out green. The ones returning constants, are very unlikely to match. There's no particular reason a tool would put a 4 there, as opposed to zero, 1, -1, and other such significant numbers. Changing the exponentiation into subtracting one argument from the other gets us zero, dividing them gets us one, returning either one alone gets us two, and the mismatched types and deliberate errors will at *least* make the test not pass. But . . .



. . . addition, multiplication, and exponentiation in the reverse order, all get us the correct answer. So, mutants based on *these* mutations will *"survive"* our test. This is the usual way mutants survive, by . . .

```
mutant_power(x, y)
==
original_power(x, y)
```

Codosaur.us @davearonson

. . . returning the same result as the original function. Or they have the same side effect — whatever our tests are looking at. To determine how that *happens*, it helps to take a closer look at it *along with* a test it passes. Let's start with . . .

the change:

```
43 -   x ** y
43 +   x + y
```

our test:

```
assert power(2, 2) == 4
```

Codosaur.us @davearonson

. . . the "plus" mutant. Looking at the change, together with our test, makes it clear that this one survives because . . .

. . . two *plus* two equals two *to* the two. (And so does two *times* two, but he's in the background, we can save him for later.)

So how can we *kill* . . .

the change:

```
43 -   x ** y
43 +   x + y
```

our test:

```
assert power(2, 2) == 4
```

Codosaur.us

@davearonson

. . . this mutant, in other words, make at least one test fail when run against it, that would pass when run against the original code? To do that, we need to make at least one test use inputs such that x *plus* y is different from x *to the* y. For instance, we could add a test or change our existing test to . . .

```
assert power(2, 4) == 16
```


. . . assert that two to the *fourth* power is *sixteen*. All the mutants that our original test killed, *this would still kill*. But also, two *plus* four is six, not six*teen*, so **this kills the plus mutant.** See how that works?

Better yet, two *times* four is eight, which is *also* not sixteen! So, this kills the "times" mutant as well. Killing one mutant often kills many other mutants of the same function.

But . . .

. . . the pair of argument-swapping mutants survive, because . . .

```
4 ** 2 == 16

2 ** 4 == 16
```

Codosaur.us

@davearonson

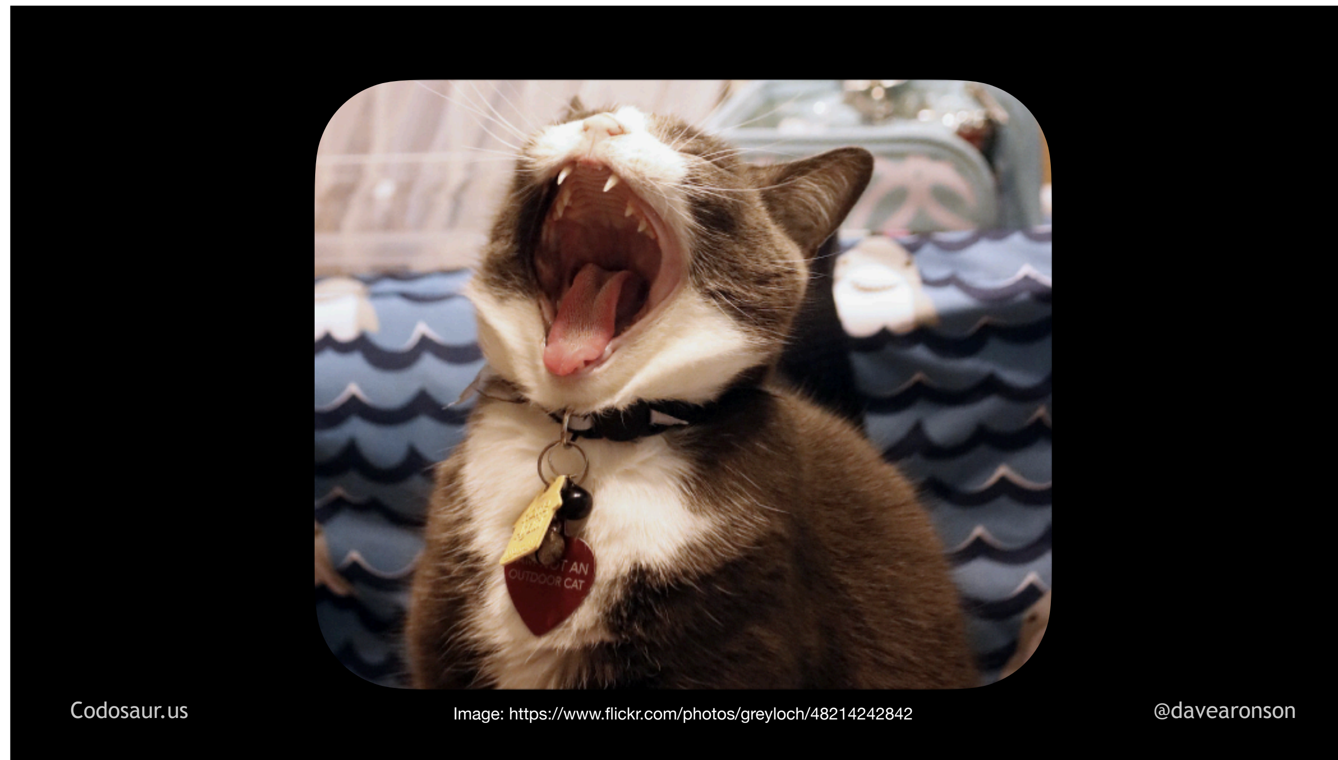
. . . four squared and two to the fourth, are both sixteen. But we can . . .

. . . attack these mutants separately, no need to kill them all in one shot and be some kind of superhero about it. To kill *them*, again, we can either add a test, or adjust an existing test, to . . .

```
assert power(2, 3) == 8
```

Codosaur.us @davearonson

. . . assert that two to the *third* power is *eight*. Three squared is nine, not eight, so **this kills the argument-swapping mutants**. Better yet, two *plus* three is five, two *times* three is six, and both of those are not eight, so the "plus" and "times" mutants *stay* dead, even if this were still our one and only test. (PAUSE!) With these inputs, the correct operation is the only simple common one that yields the correct answer. This isn't the *only* solution, though; there are *lots* of ways to skin . . .



. . . *that* flerken!

This may make mutation testing sound simple, but this was a downright trivial example. So let's look at a more *complex* one!

Suppose we have a function to send a message, . . .

```
def send_message(buf, len):
  sent = 0
  while sent < len:
    sent += send_bytes(buf + sent,
                       len - sent)
  return sent
```

Codosaur.us @davearonson

. . . like so. send_*message*, uses send_*bytes* to send as many bytes as send_bytes *could* send, like a woodchuck, looping to pick up where it left off, until the message is all sent.

A mutation testing tool could make lots of mutants from this, but one of particular interest, would be . . .

```
def send_message(buf, len):
  sent = 0
  while sent < len:
    sent += send_bytes(buf + sent,
                       len - sent)
  return sent
```

Codosaur.us @davearonson

. . . this, an example of removing a looping control.

Now suppose that this mutant does indeed survive our test suite, which consists mainly of . . .

```
assert send_message(msg, size) == size
```

Codosaur.us @davearonson

. . . *this*. (PAUSE!) There's a bit more that I'm not going to show you *quite* yet, dealing with setting the size and actually creating the message. But even without seeing that test code, what does the survival of that non-looping mutant tell us? (PAUSE!)

If a mutant that only goes through . . .

```
def send_message(buf, len):
  sent = 0
  while sent < len:
    sent += send_bytes(buf + sent,
                       len - sent)
  return sent
```

Codosaur.us

@davearonson

. . . that loop once, acts the same as our normal code, as far as our tests can tell, that means that our *tests* are only making our *normal* code go through that loop once. So, what does *that* mean? (PAUSE!) You'll find that interpreting mutants often involves a lot of asking yourself *"so, what does that mean"*, often deeply recursively!

In this case, it means that we're not testing sending a message larger than send_*bytes* can handle in one chunk! There are many ways that can happen, but we're only going to look at two possibilities. The most *likely* is that we simply *forgot*, or didn't *bother,* to test with a big enough message. For instance, . . .

in module Network:

```
max_chunk_size = 10_000
```

in test_send_message:

```
msg = "foo"
size = length(msg)
# other setup, like stubbing send_bytes
assert send_message(msg, size) == size
```


. . . suppose our maximum chunk size, what send_*bytes* can handle in one chunk, is 10,000 bytes. But . . .

in module Network:

```
max_chunk_size = 10_000
```

in test_send_message:

```
msg = "foo"
size = length(msg)
# other setup, like stubbing send_bytes
assert send_message(msg, size) == size
```

Codosaur.us @davearonson

. . . we’re only testing with a *three* byte message. (PAUSE!)

The obvious fix is to deliberately use a message larger than our maximum chunk size. We can easily construct one . . .

in module Network:

```
max_chunk_size = 10_000
```

in test_send_message:

```
size = socket.max_chunk_size + 1
msg = "x" * size
# other setup, like stubbing send_bytes
assert send_message(msg, size) == size
```

Codosaur.us

@davearonson

. . . like so.

But now let’s look at another possible cause and solution. Maybe we *did* test with the *largest* permissible message, out of a set of predefined messages, or message *sizes*. For instance, . . .

in module Message:

```
SmallMsgSize = 1_000
LargeMsgSize = 5_000  # the largest
```

in test_send_message:

```
size = Message.LargeMsgSize
msg = Message.make_msg("a" * size)
# other setup, like stubbing send_bytes
assert send_message(msg, size) == size
```

Codosaur.us

@davearonson

. . . here we have Small and Large message sizes, we test with a Large, and yet, this mutant survives! In other words, we're still sending the whole message in one chunk. What could possibly be wrong with that? It sounds like a *good* thing to me! What is this mutant trying to tell us in this case? (PAUSE!)

In *this* scenario, it’s trying to tell us that a version of send_message with the looping removed will do the job just fine. If we remove the looping, we wind up with . . .

```
def send_message(buf, len):
  sent = 0
  sent += send_bytes(buf + sent,
                     len - sent)
  return sent
```

Codosaur.us @davearonson

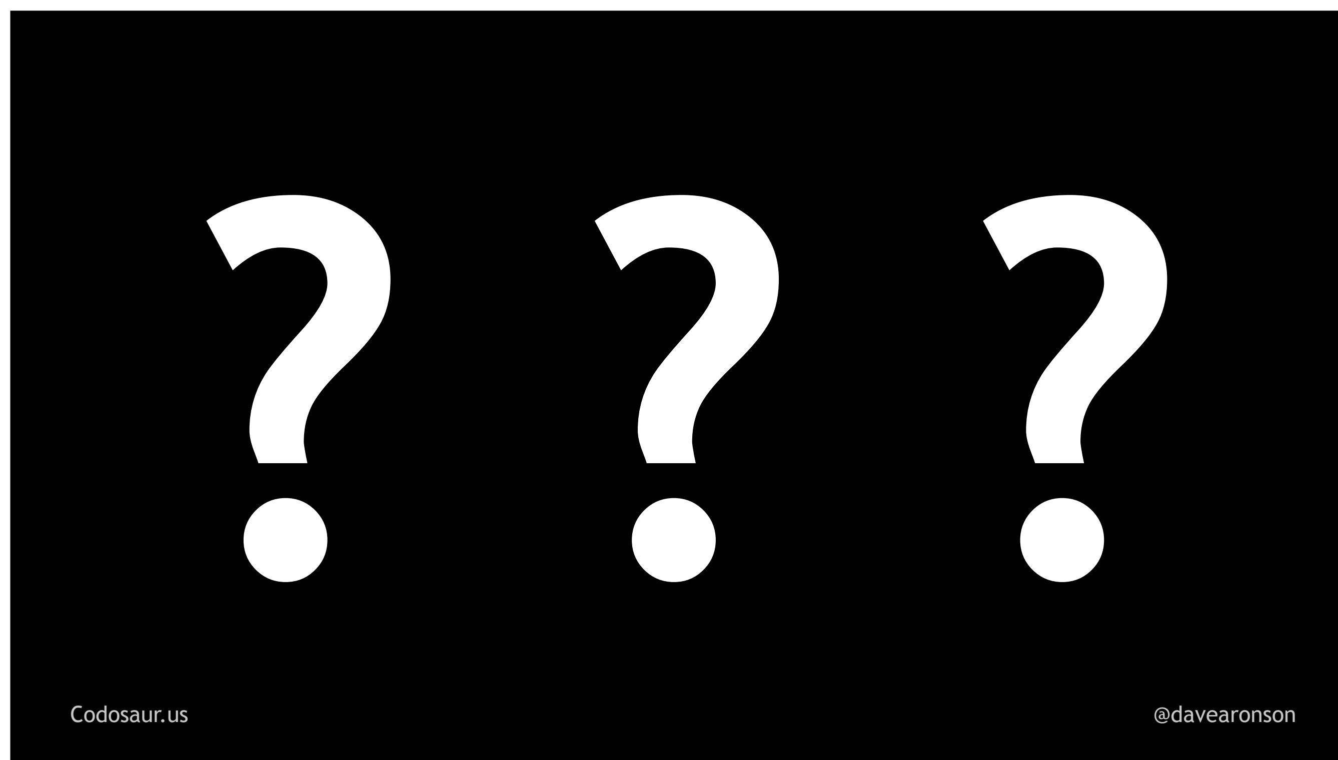
. . . this, and if we run our mutation testing tool on *this*, it will show some other stuff as now being redundant, because we only needed it to support the looping. If we *also* remove that, then it boils down to . . .

```
def send_message(buf, len):
  return send_bytes(buf, len)
```

Codosaur.us @davearonson

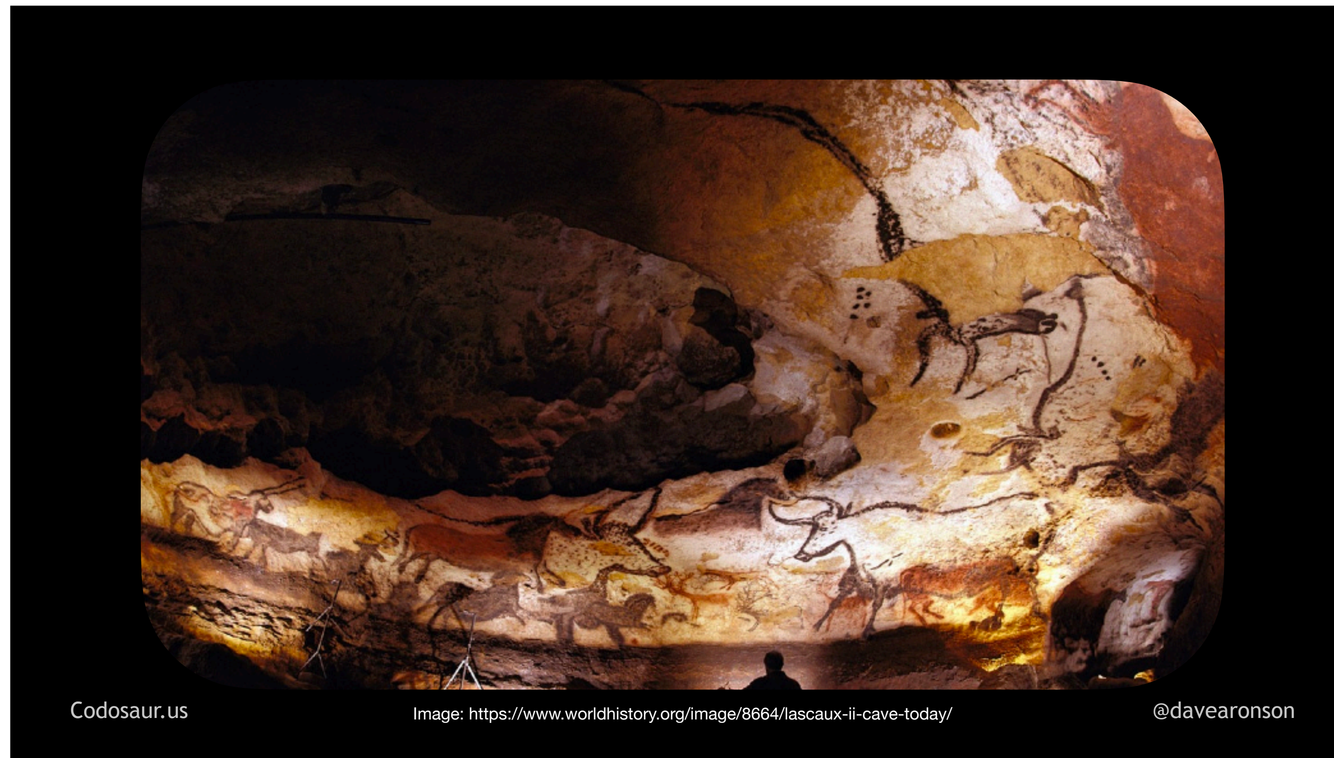
. . . this. (PAUSE!) Now the message is clear: the *entire* send_message *function* may well be *redundant*, so we can just use send_*bytes directly!* In real-world code, though, it might not be, because there may be some logging, error handling, and so on, needed in send_message, but at the very least, the *looping* was redundant. Fortunately, when it's this kind of problem, the usual solution is clear and easy, just rip out the extra junk that the mutant doesn't have. This will also make our code more *maintainable*, by getting rid of useless cruft that just gets in the way of understanding it.

Now that we've seen examples of finding both bad tests and redundant code, I'll address a couple of . . .



. . . Frequently Asked Questions. First, this all sounds pretty weird, deliberately making tests fail, to prove that the code succeeds! Where did this whole . . .

. . . bizarro idea come from? Mutation testing has a surprisingly . . .



. . . long history -- at least in the context of computers. It was first proposed in 1971, in Richard Lipton's term paper “Fault Diagnosis of Computer Programs”, at Carnegie-Mellon University. The first *tool* didn't appear until 1980, in Timothy Budd's PhD work at Yale. But it was not *practical* on typical computers, until the early 2000s, with significant advances in CPU *speed*, multi-*core* CPUs, and so on. But now, it’s practical even on fairly low-end modern systems, like this 2020 MacBook Air that I’m presenting on.

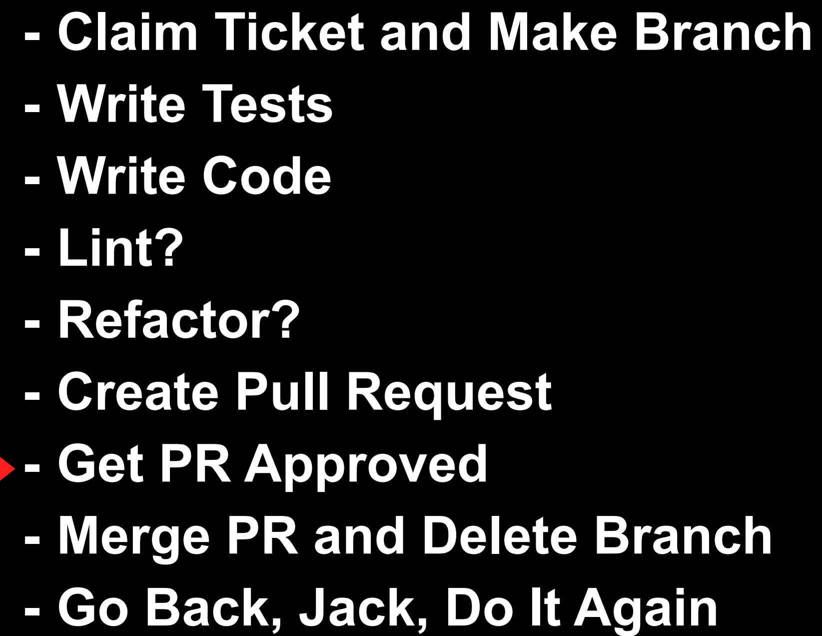
Another common question is: where should we fit this into . . .

- Claim Ticket and Make Branch
- Write Tests
- Write Code
- Lint?
- Refactor?
- Create Pull Request
- Get PR Approved
- Merge PR and Delete Branch
- Go Back, Jack, Do It Again

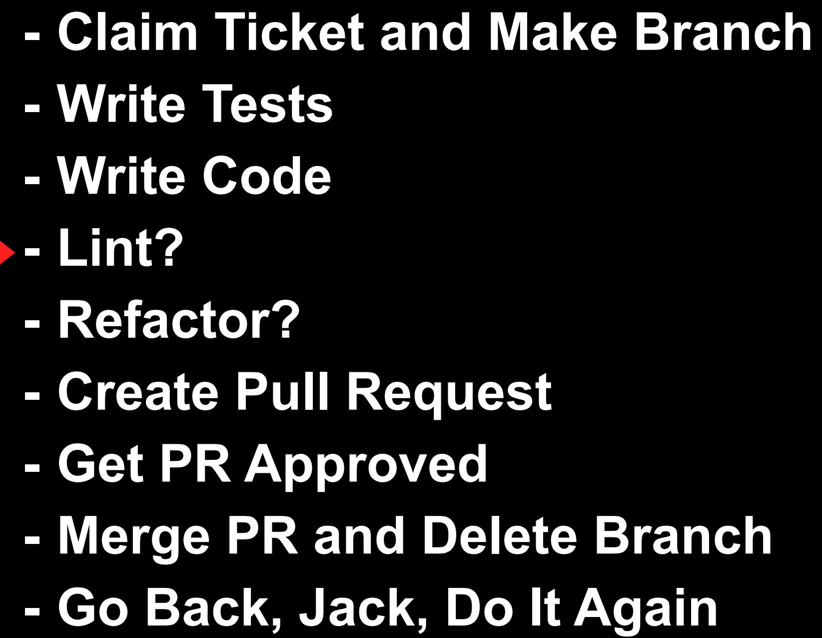
Codosaur.us

@davearonson

. . . our development process? Mainly, I think it belongs at *least* . . .



. . . here, as part of the requirements for a Pull Request (or whatever your process uses) to be approved. You can set some standards for what you're willing to tolerate, on the changed code, or the whole codebase, or both, such as a number of surviving mutants or a percent increase in them, ideally zero or less. Ideally this would be automated, as part of a CI pipeline. That said, I personally would also do it in my *own* work as part of . . .



. . . the Linting step, where I apply all sorts of quality checking tools.

If you'd like to try mutation testing for yourself . . .

Alloy:	MuAlloy
Android:	mdroid+
C:	mutate.py, SRCIROR
C/C++:	accmut, dextool, MART, MuCPP, Mutate++, mutate_cpp, SRCIROR
C#/.NET/Mono:	nester, NinjaTurtles, Stryker.NET, Testura.Mutation, VisualMutator
Clojure:	mutant
Crystal:	crytic
Dart:	mutation_test
Elixir:	darwin, exavier, exmen, mutation, Muzak [Pro]
Erlang:	mu2
Etherium:	vertigo
FORTRAN-77:	Mothra (written in mid 1980s!)
Go:	go-mutesting, gremlins, ooze
Haskell:	fitspec, muCheck
Java:	jumble, major, metamutator, muJava, pit/pitest, and many more
JavaScript:	stryker, ~~grunt-mutation-testing~~
Pharo:	MUTALK
PHP:	infection, ~~humbug~~
PL/SQL:	MuPLSQL
Python:	**cosmic-ray, mutmut, mutpy, pester, xmutant**
Ruby:	mutant, ~~mutest~~, ~~heckle~~
Rust:	mutagen
Scala:	scalamu, stryker4s
Smalltalk:	mutalk
Solidity:	RegularMutator
SQL:	SQLMutation
Swift:	muter
Anything on LLVM:	llvm-mutate, mull
Tool to make more:	Wodel-Test (https://gomezabajo.github.io/Wodel/Wodel-Test/)

Codosaur.us @davearonson

. . . here is a list of tools for some popular languages and platforms, including of course Python. The tools I know are outdated, are crossed out.

To summarize at last, mutation testing is a powerful technique to . . .

😀 Checks that code is meaningful

Codosaur.us @davearonson

. . . help ensure that our code is meaningful and . . .

😀 Checks that code is meaningful

😀 Checks that tests are strict

Codosaur.us

@davearonson

. . . our tests are strict. It's . . .

😀 Checks that code is meaningful

😀 Checks that tests are strict

😀 Easy to get started with

easy to get started with, but it's . . .

😀 Checks that code is meaningful

😀 Checks that tests are strict

😀 Easy to get started with

😩 Difficult to interpret results

Codosaur.us

@davearonson

. . . not so easy to interpret the results, nor is it . . .

- 😀 Checks that code is meaningful
- 😀 Checks that tests are strict
- 😀 Easy to get started with
- 😩 Difficult to interpret results
- 😩 Hard labor on the CPU

Codosaur.us @davearonson

. . . easy on the CPU.

Even if these drawbacks mean it might not be a good fit for our current projects, I still think it's just . . .

- 😀 Checks that code is meaningful
- 😀 Checks that tests are strict
- 😀 Easy to get started with
- 😩 Difficult to interpret results
- 😩 Hard labor on the CPU
- 😎 Fascinating concept! 🤓

. . . a really cool idea . . . in a geeky kind of way.

If you have any questions, . . .

T.Rex-2024@Codosaur.us
twitter.com/DaveAronson
linkedin.com/in/DaveAronson

Slides and FULL SCRIPT:
Codosaur.us/reds/mutants-xtremepy-24-slides

. . . I'll take them now, and if you think of anything later, there's my contact info, plus the URL for the slides, complete with a full script, which I've *mostly* stuck to.