CURRENT MAIN CONTENT TIME: ~40; time slot is 45 total, so aim for 35-40 of content

Hallå, Stockholm!

(Hello, Stockholm!)

www.Codosaur.us

@davearonson

hal-LO stockHOLM!

Jag heter Dave Aronson,

(I'm Dave Aronson,)

www.Codosaur.us

@davearonson

Yaw Heter Dave Aronson,

T. Rex på Codosaurus,

(the T. Rex of Codosaurus,)

www.Codosaur.us

@davearonson

T. Rex po Codosaurus,

och jag flög hit

(and I flew here)

www.Codosaur.us

@davearonson

oh YAW flug HEET

på min tama pterodactyl

(on my pet pterodactyl)

www.Codosaur.us

@davearonson

po meen taw-maw tero-Dac-Teel

för att lära er

(to teach you)

www.Codosaur.us

@davearonson

fur at laara eer

hur man dödar mutanter!

(how to kill mutants!)

www.Codosaur.us

@davearonson

hoor man deu-dar muTANter!

Men . . .

(But . . .)

www.Codosaur.us

@davearonson

Men . . .

jag kommer att göra det

(I will do it)

www.Codosaur.us

@davearonson

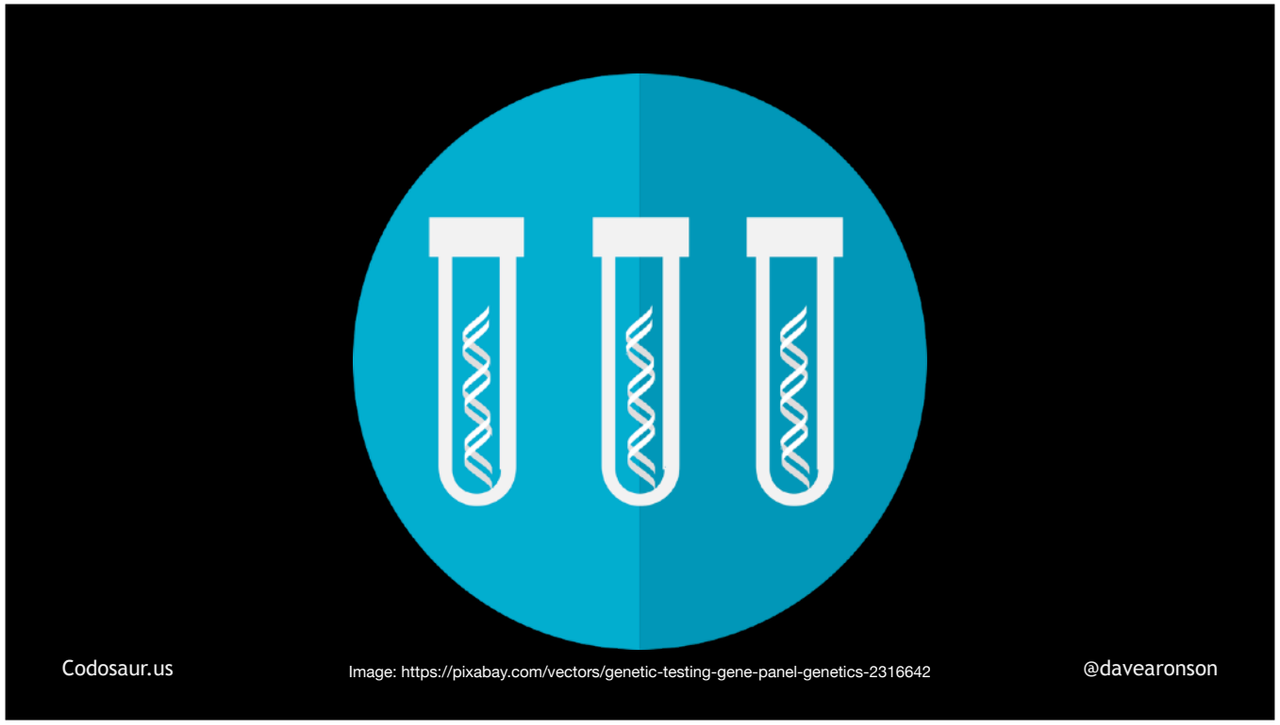
. . . yaw Kommer att Yora deuh . . .

. . . po ENyelska.

(PAUSE!)

Mainly because you've just heard almost all the Swedish I speak.

Let's start with the basics. What on Infinite Earths is . . .



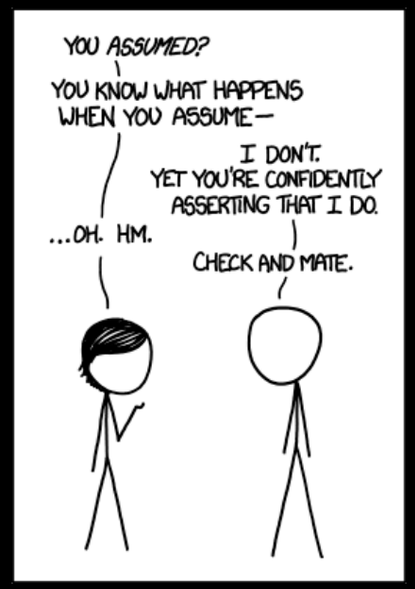
. . . mutation testing? In *our* universe, that of software development, not comic books, it's a software testing technique. (Surprise!) But why is *this* technique different from all *other* techniques? The big difference is that most others are about . . .

Codosaur.us

Image: https://pixabay.com/illustrations/tick-green-tick-correct-642162/

@davearonson

. . . checking whether our code is correct. Mutation testing . . .

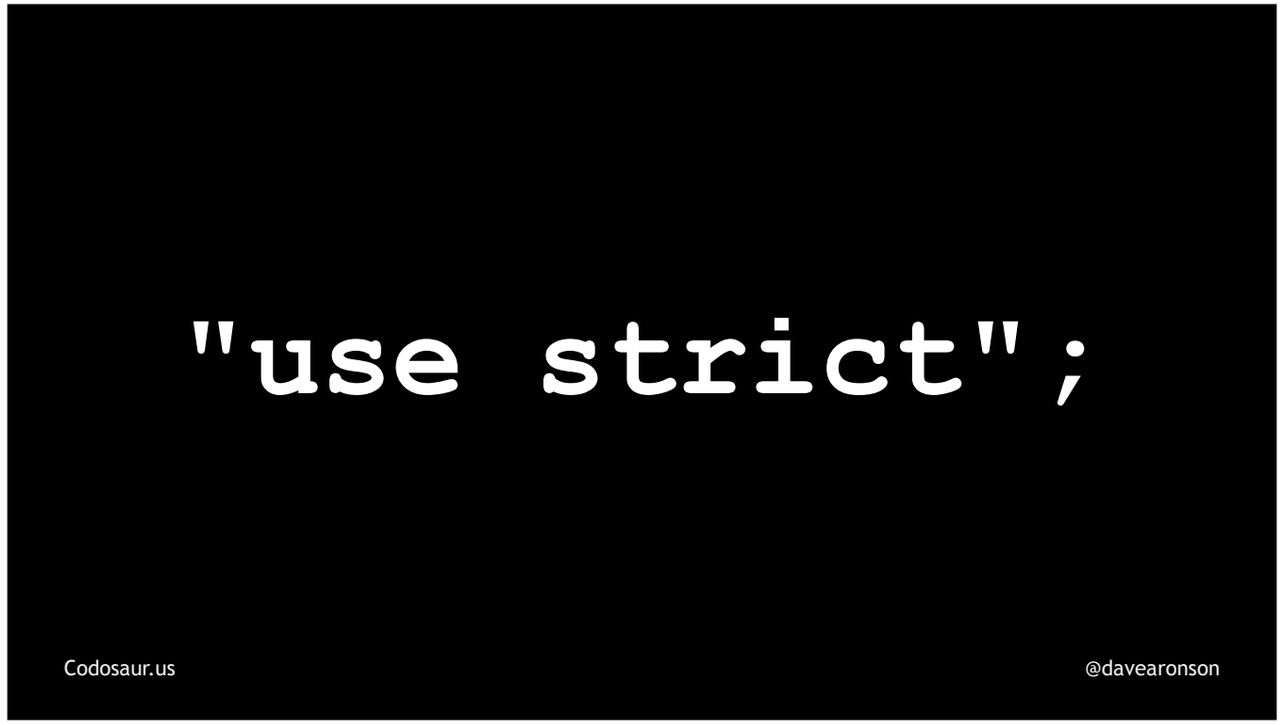


Codosaur.us

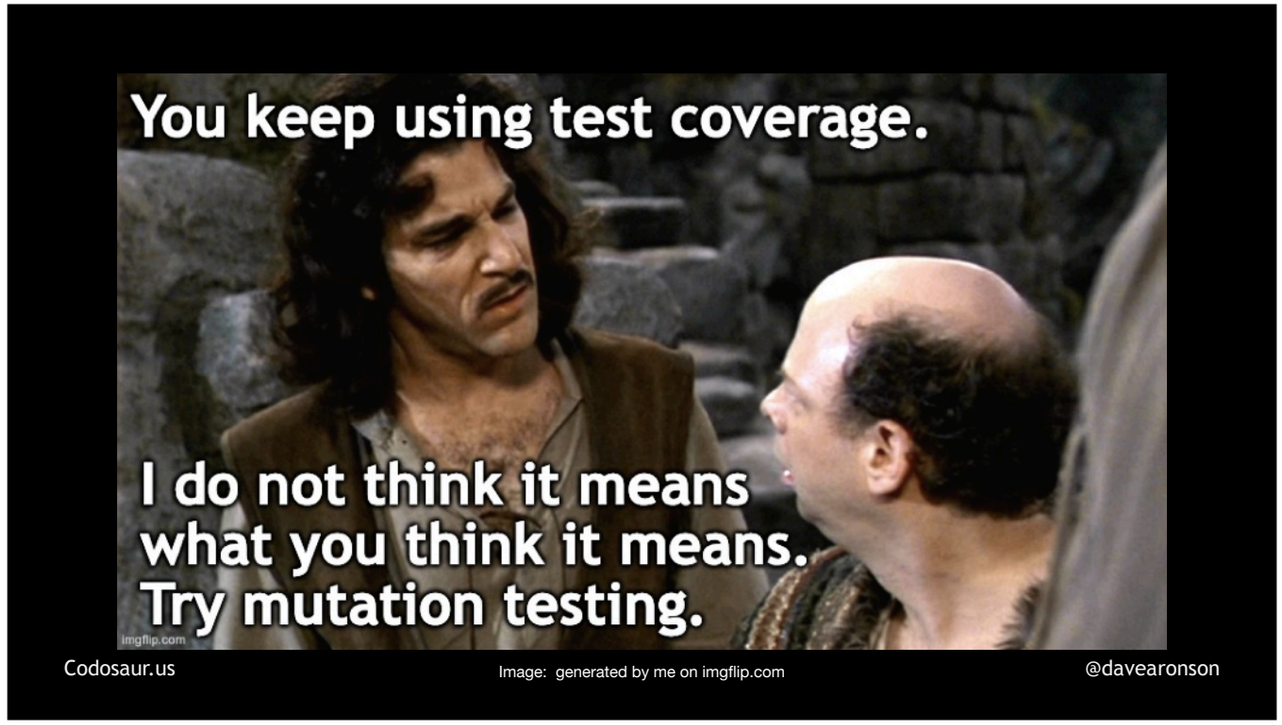
Image: https://xkcd.com/1339/

@davearonson

. . . *assumes* that our code is correct, at least in the sense of passing its tests. Instead, mutation testing is about checking for two *different* qualities. In my opinion, the more *important, interesting,* and *immediately useful* of these two qualities is that our test suite is . . .



. . . *strict*. Now you may be thinking, “Isn’t that what code coverage is for? If we have 100% coverage, doesn’t that mean our code is fully tested?”



No. (PAUSE!) The *only* thing that code coverage tells us . . .

```
defmodule Conway do
  @alive "*"
  @dead  " "

  def next_state(@alive, neighbors),
    do: if Enum.member?([3, 4], neighbors),
      do: @alive, else: @dead

  def next_state(@dead, neighbors),
    do: if neighbors == 3,
      do: @alive, else: @dead
end
```

Codosaur.us

@davearonson

. . . is whether code was *executed* by at least one test. It tells us NOTHING about whether the *correctness* of that code *made any difference* to whether the test passed or failed.

Let’s look at . . .

```
test "live with 3 survives" do
  expected = @alive
  actual = next_state(@alive, 3)
  assert actual == expected
end
```

Codosaur.us

@davearonson

. . . a test, and . . .

```
test "live with 3 survives" do
  expected = @alive
  actual = next_state(@alive, 3)
  # assert actual == expected
end
```

Codosaur.us @davearonson

. . . comment out the assertion. This might be done for any number of reasons, usually not very good, but it's often done anyway. Our test still *runs* the function, so the lines show as covered, but we're not asserting anything, so the code is *obviously* not tested. This is just *one* of *many* ways that coverage can be misleading.

So how *can* we tell if the code's correctness made any difference to whether the test passes? That . . . is where mutation testing comes in.

To check that our test suite is *strict*, a mutation testing tool will . . .

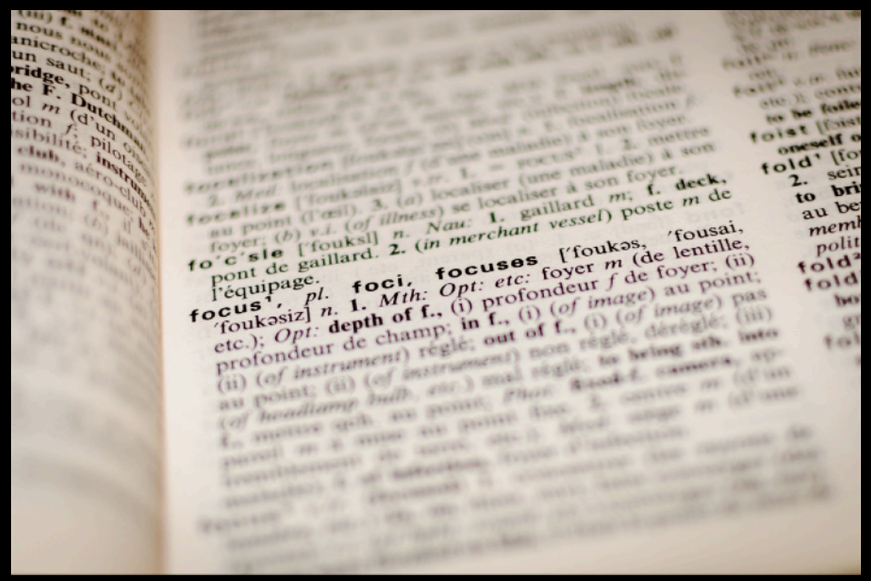
Codosaur.us

Image: https://commons.wikimedia.org/wiki/File:Mind_the_gap_2.JPG

@davearonson

. . . find the gaps in our test suite, that let our code get away with unwanted 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, poorly *written* tests, or poorly *maintained* tests, such as ones that didn't keep pace with changes in the code.

Speaking of which, the other thing mutation testing checks is that our code is what I call . . .



Codosaur.us

Image: https://pxhere.com/en/photo/825760

@davearonson

. . . *meaningful*, by which I mean that any change to the code, will produce a noticeable change in its behavior. Lack of *meaning* comes mainly from code being unreachable, redundant with other code, or otherwise just not having any real effect. Once we find "meaningless" code, we can figure out *why* it's meaningless, then either make it meaningful, if that fits our intent, or just remove it.

Mutation testing . . .

Codosaur.us

Image: https://www.flickr.com/photos/garryknight/2565937494

@davearonson

. . . puts these two together, by checking that every possible small change to the code does indeed make a noticeable change to its behavior, *and* that the test suite is indeed strict enough notice that change, and fail. Not all of the tests have to fail, but each change should make *at least one* test fail.

That's the positive side, but there are some drawbacks. As Fred Brooks told us back in 1986, there's no . . .

Codosaur.us

Image: https://www.flickr.com/photos/sdasmarchives/4590226412

@davearonson

. . . silver bullet! Besides, those are for killing . . .

. . . werewolves, not mutants!

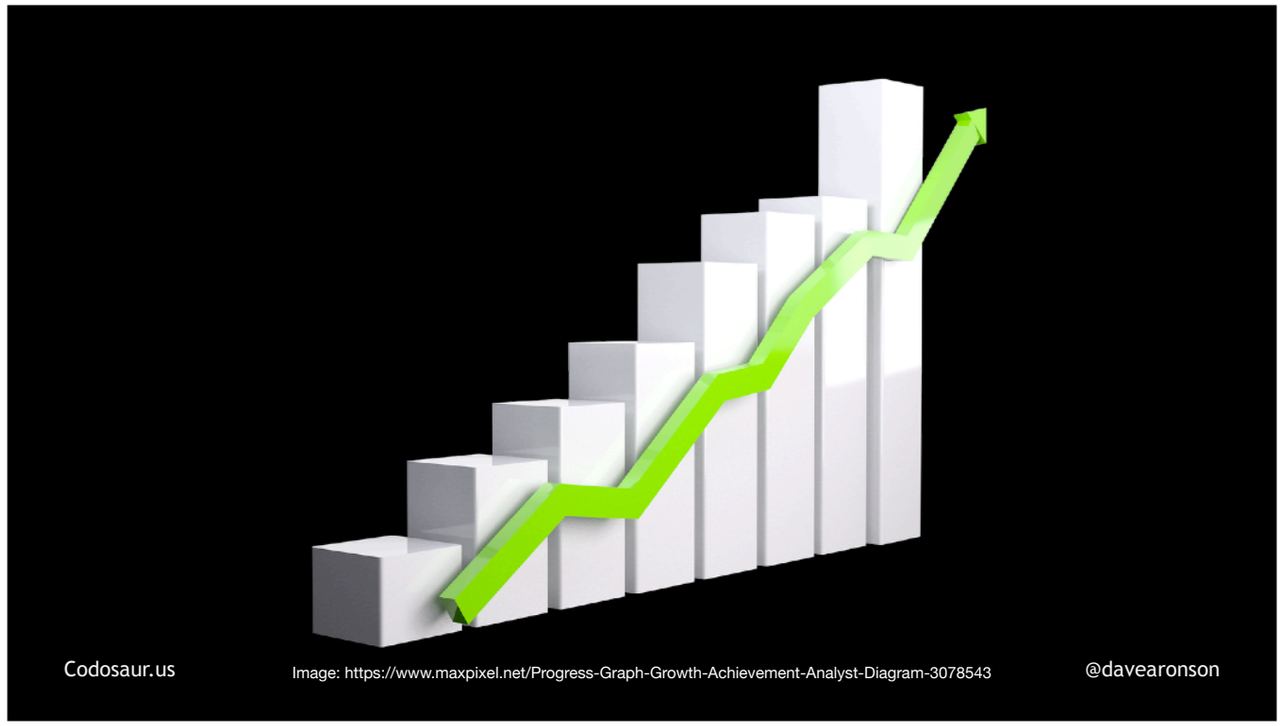
The first drawback is that it's rather . . .

Codosaur.us

Image: https://www.jtfb.southcom.mil/Media/Photos/igphoto/2000888525/

@davearonson

. . . hard labor for the CPU, and therefore usually ra-ther sloooow. We certainly won’t want to mutation-test our whole codebase on every save! 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, 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 main branch, or some such difference. That, maybe we can do on each save, if we save often, use short-lived branches, and so on.

Also, its CPU-intensive nature can really . . .

Codosaur.us

Image: https://pixabay.com/illustrations/clouds-dollar-symbol-characters-96588/

@davearonson

. . . run up our bills on cloud platforms such as AWS or Azure! (Or aZURE, however you PROnounce it.)

Another drawback is that mutation testing is . . .

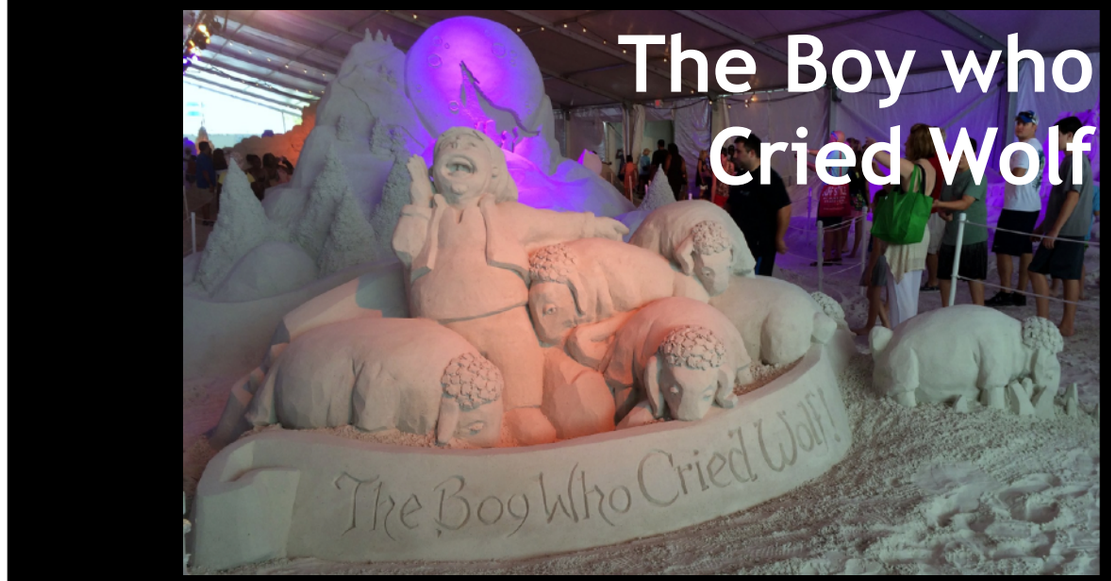
Codosaur.us

Image: https://www.flickr.com/photos/ell-r-brown/5866767106

@davearonson

. . . not at all a beginner-friendly technique! 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. Their accent is verrah straynge, and they're almost as incoherent as . . .

. . . zombies, but with a much bigger vocabulary, so they’re not always on about braaaaaaains. 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 . . .

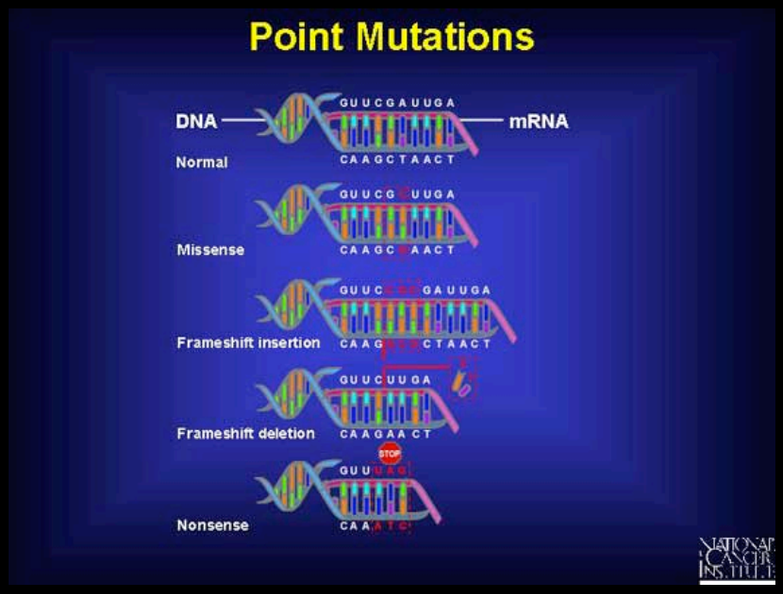


. . . false alarm, because the mutation didn't make a test fail, but it didn't make any real difference in the first place. It can still take quite a lot of time and effort to figure *that* out.

Even if a mutation *does* make a difference, there is normally quite a lot of code that we . . .

. . . *shouldn't bother* to test. For instance, if we have a debugging log message that says "The value of X is" and then the value of X, that constant part will get mutated, but we don't really care! Fortunately, most tools have ways to say "don't bother mutating this line", or even this whole function . . . but that's usually with comments, which can clutter up the code, and make it less readable.

Now that we've seen the pros and cons, how does mutation testing work, unlike the guy on this sign? It . . .

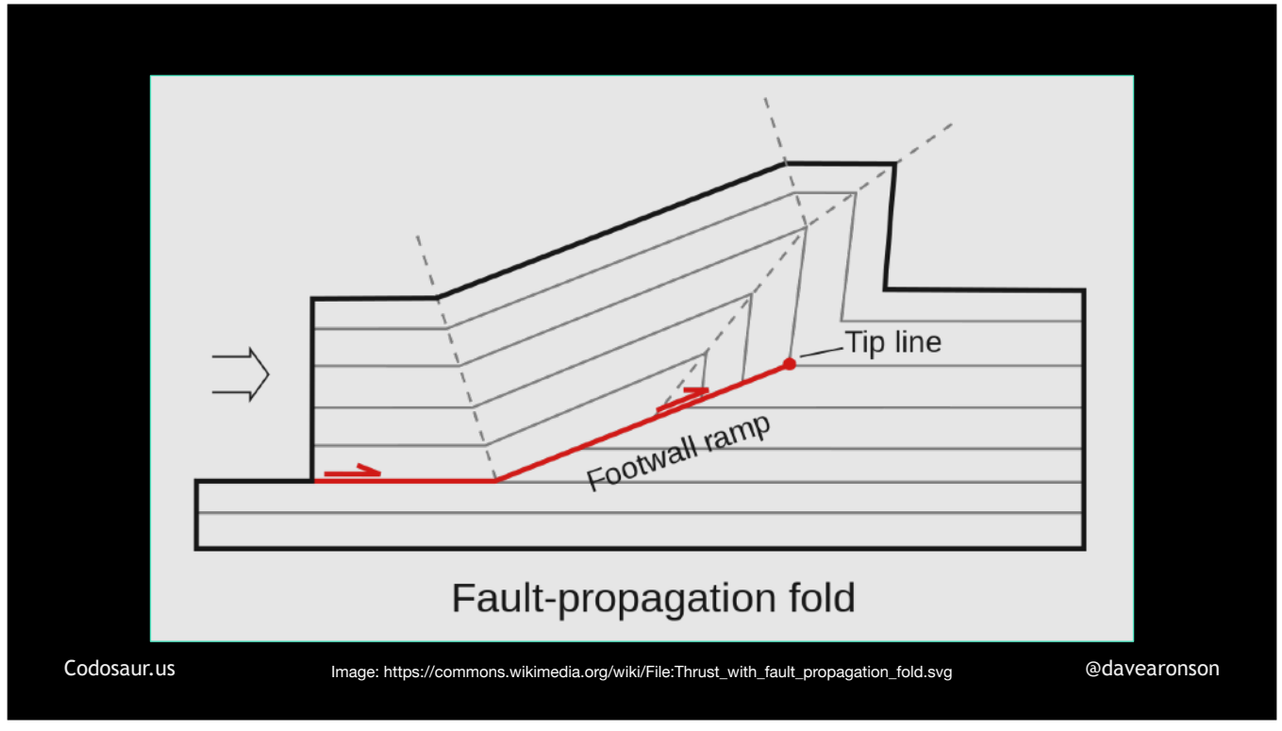


Codosaur.us

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

@davearonson

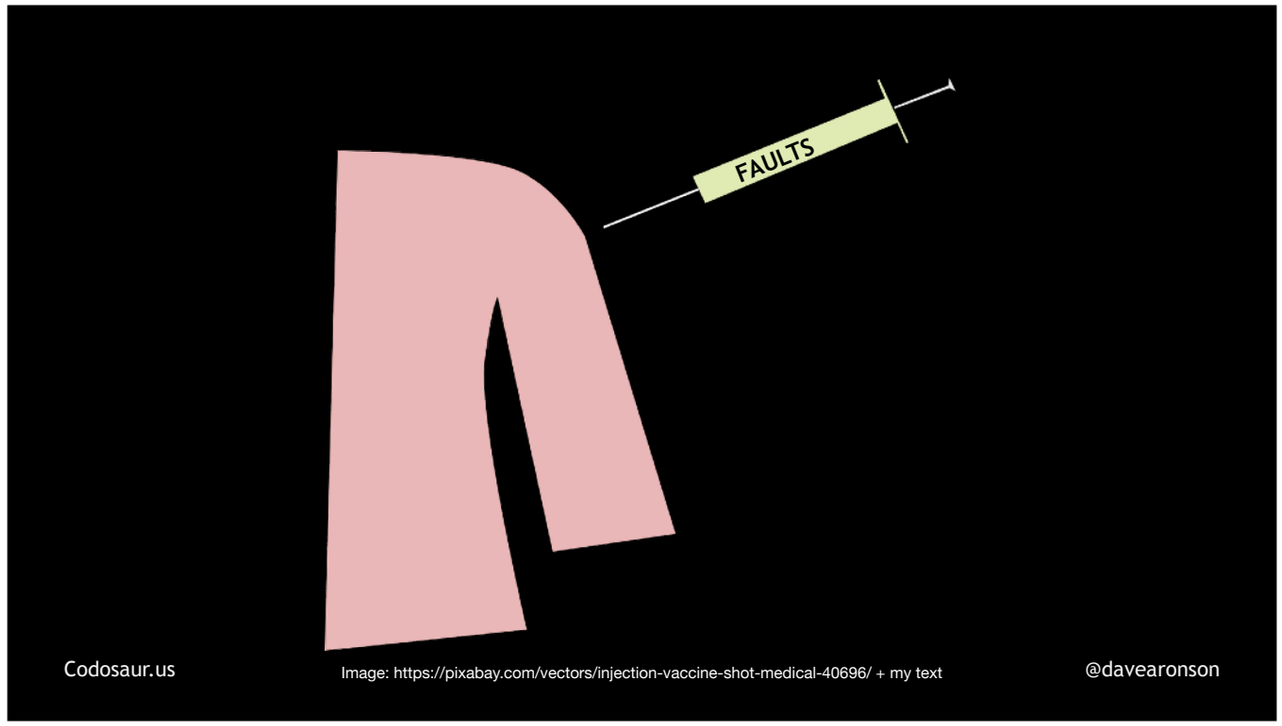
. . . *mutates* copies of our code, hence the name. It does this to create test failures, also known as . . .



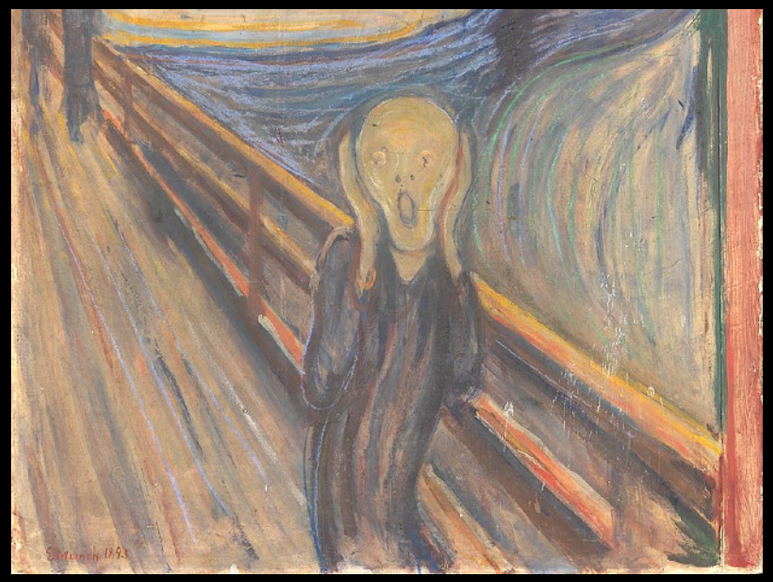
. . . faults. So, mutation testing can be categorized as a *fault-based* testing technique. This means it is related to something you might already be familiar with:

. . . Chaos Monkey, from Netflix. Just like Chaos Monkey helps Netflix discover flaws in their error recovery, mutation testing helps us discover flaws in our tests and our code. But the way mutation testing does it, is sort of . . .

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



. . . injecting faults, such as dropped connections, into Netflix's . . .



Codosaur.us

Image: https://commons.wikimedia.org/wiki/File:Edvard_Munch,_1893,_The_Scream,_oil,_tempera_and_pastel_on_cardboard,_91_x_73_cm,_National_Gallery_of_Norway.jpg

@davearonson

. . . production network.

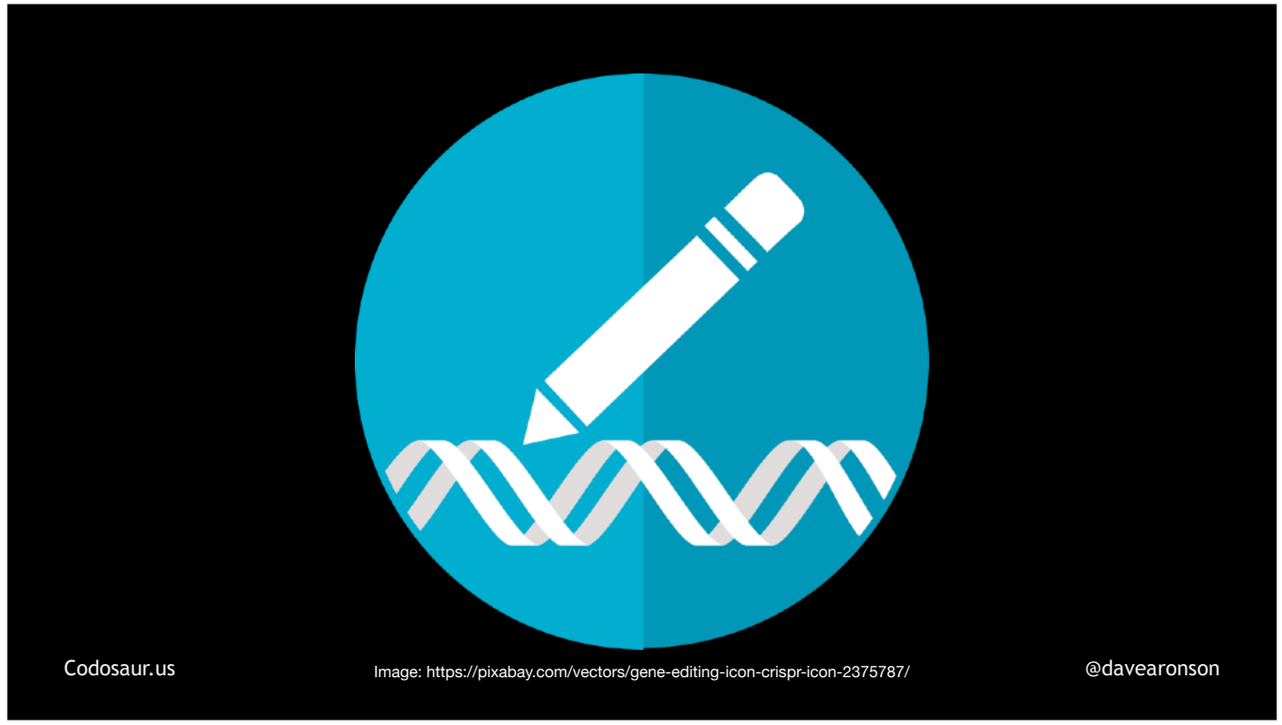
If all still goes well, in the sense that Netflix's customers don't notice, and their metrics are still good, then Netflix knows that their error recovery is working fine. Mutation testing, however, injects semantic . . .

Codosaur.us

Image: https://www.flickr.com/photos/funkblast/16937812322

@davearonson

. . . *changes*, not necessarily *problems*. It doesn't usually *know* whether these semantic changes will create *faults* or not. We certainly hope they all will, but that depends on the test suite. It injects them *into* . . .



. . . copies of our code, not our actual network. It does its work in our . . .

Codosaur.us

Image: https://sservi.nasa.gov/articles/ladee-vibration-testing-complete/

@davearonson

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

```
$ mutation_test
........................................
........................................
........................................
........................................
........................................
........................................
........................................
280 tests, 420 assertions,
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 . . .

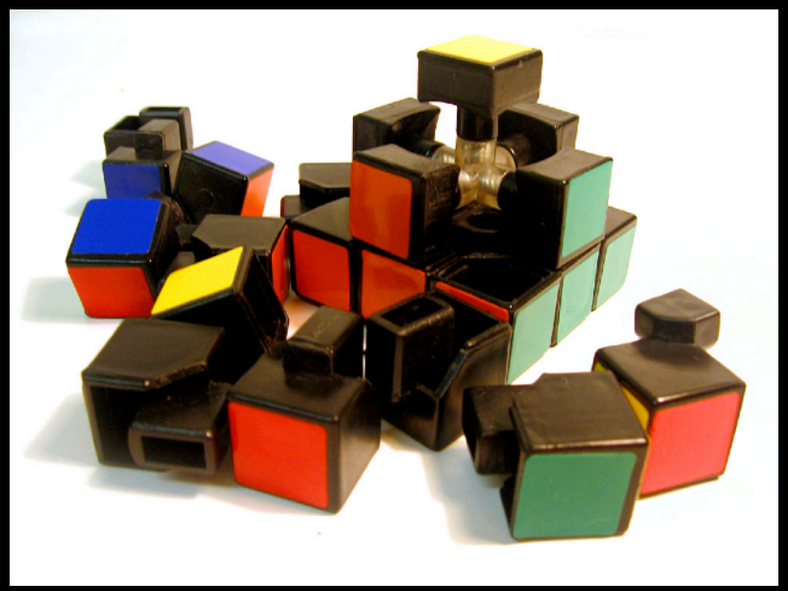
. . . there *is* a problem! Remember, each change to our code should make *at least* one test *fail*.

Mutation testing has also been compared to . . .

. . . fuzzing, a security penetration technique involving throwing random data at an application. Mutation testing is somewhat like fuzzing our *code* rather than fuzzing the *data*, but it's . . .

. . . not random. Most tools have a set of mutations they know how to do. The smart ones can use the results of simpler mutations, to know they don’t need to bother with more complex ones, but it’s still not random.

But enough about differences. What exactly does mutation testing *do*, and how? Let's start with a high-level view. First, our chosen tool . . .

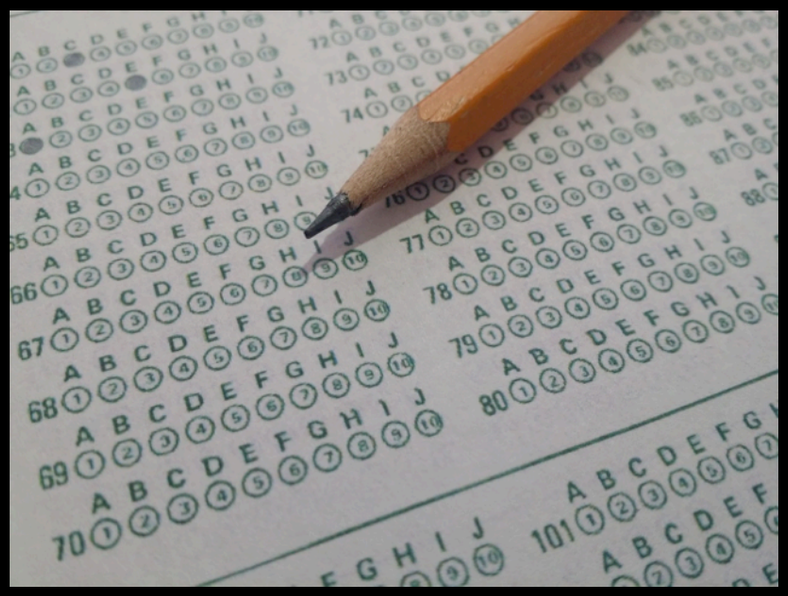


Codosaur.us

Image: https://commons.wikimedia.org/wiki/File:Disassembled-rubix-1.jpg

@davearonson

. . . breaks our code apart into pieces to test. Usually, these are our functions -- or methods if we're using an object-oriented language, but I'm just going to say functions. Then, for each function, it tries to find . . .



Codosaur.us

Image: https://pixabay.com/fi/photos/testi-testaus-kupla-muoto-986935/

@davearonson

. . . the *tests* that cover that function. If the tool can't find any applicable tests, most will simply skip this function. Some, though, will use the whole test suite, which is horribly inefficient, because it's running a lot of irrelevant tests.

Assuming we aren't skipping this function, next the tool . . .

. . . makes the mutants. To do that, it looks closely at this function to see how it can be changed. For each tiny little way the tool sees to change this function, the tool makes . . .

Codosaur.us

Image: https://pixabay.com/en/genetic-modification-mutant-mutation-549889/

@davearonson

. . . one mutant, with *that one tiny little change*.

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

Codosaur.us

Image: https://www.flickr.com/photos/39160147@N03/15074089655

@davearonson

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

Mutating function `whatever`, at `something.ex: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 represent the progress of our tool. The tools generally don't give us anything quite so well organized, or even all the info needed to produce such a chart, but it's a conceptual model I use to illustrate the point.

For each . . .

Mutating function `whatever`, at `something.ex: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

. . . mutant, derived from . . .

Mutating function `whatever`, at `something.ex: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

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

Mutating function whatever, at something.ex: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

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

Mutating function `whatever`, at `something.ex: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

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

(PAUSE) If any test . . .

Mutating function `whatever`, at `something.ex: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

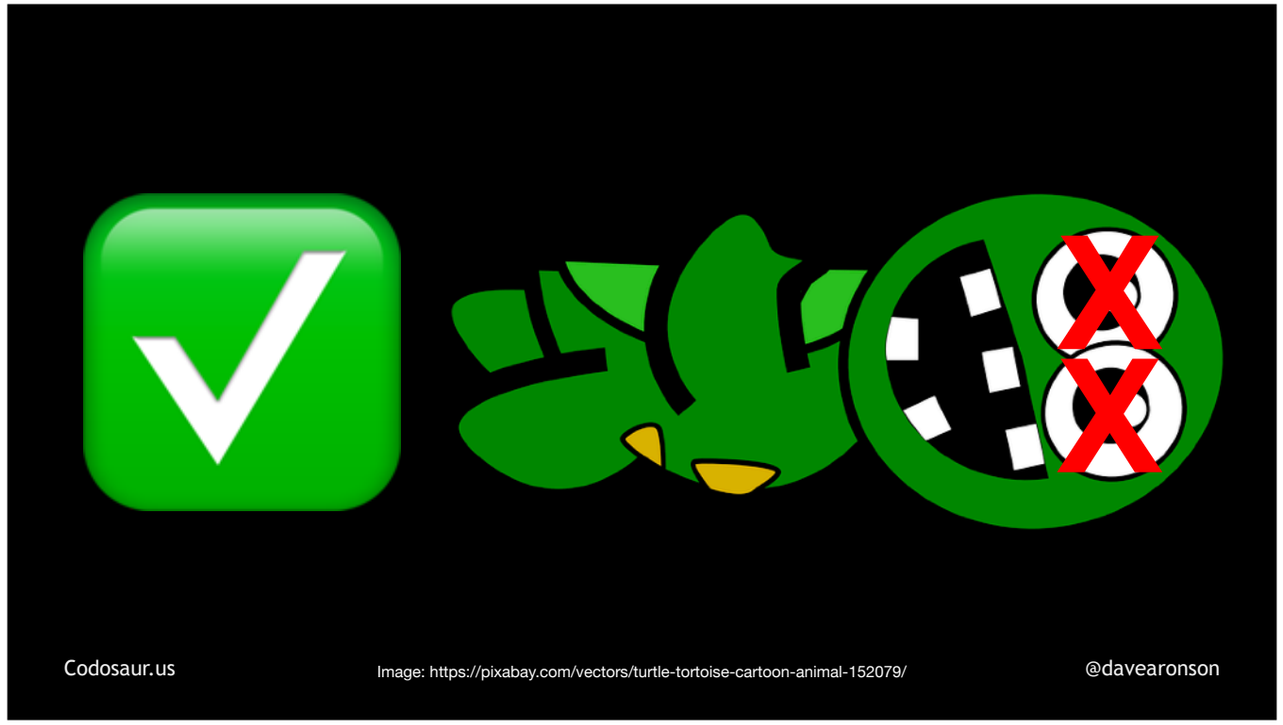
. . . *fails*, this is called . . .

Codosaur.us

Image: https://pixabay.com/id/illustrations/tengkorak-dan-tulang-bersilang-mawar-693484/

@davearonson

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



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

Mutating function `whatever`, at `something.ex: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, stop running any more tests against it, and . . .

Mutating function `whatever`, at `something.ex: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 perhaps some of . . .

Mutating function `whatever`, at `something.ex: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

. . . tests six through ten for Mutant #1. Like so much in computers, we only care about ones and zeroes.

On the other claw, if a mutant . . .

Mutating function `whatever`, at `something.ex: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

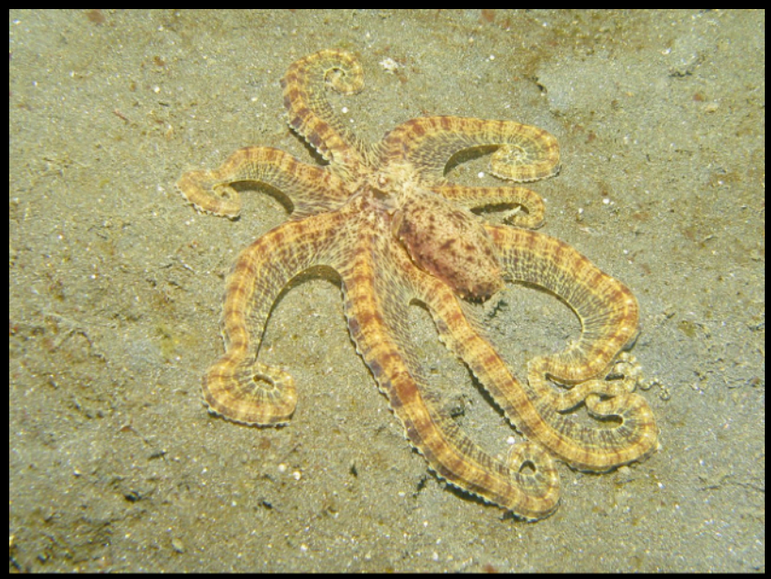
. . . lets all the tests pass, then the mutant is said to have . . .

Mutating function `whatever`, at `something.ex: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 . . .



Codosaur.us

Image: https://nl.wikipedia.org/wiki/Bestand:Mimic_Octopus2.jpg

@davearonson

. . . 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 . . .

Codosaur.us

Image: https://pixabay.com/fi/photos/avaruusolento-marsin-vihreä-hirviö-722415/

@davearonson

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

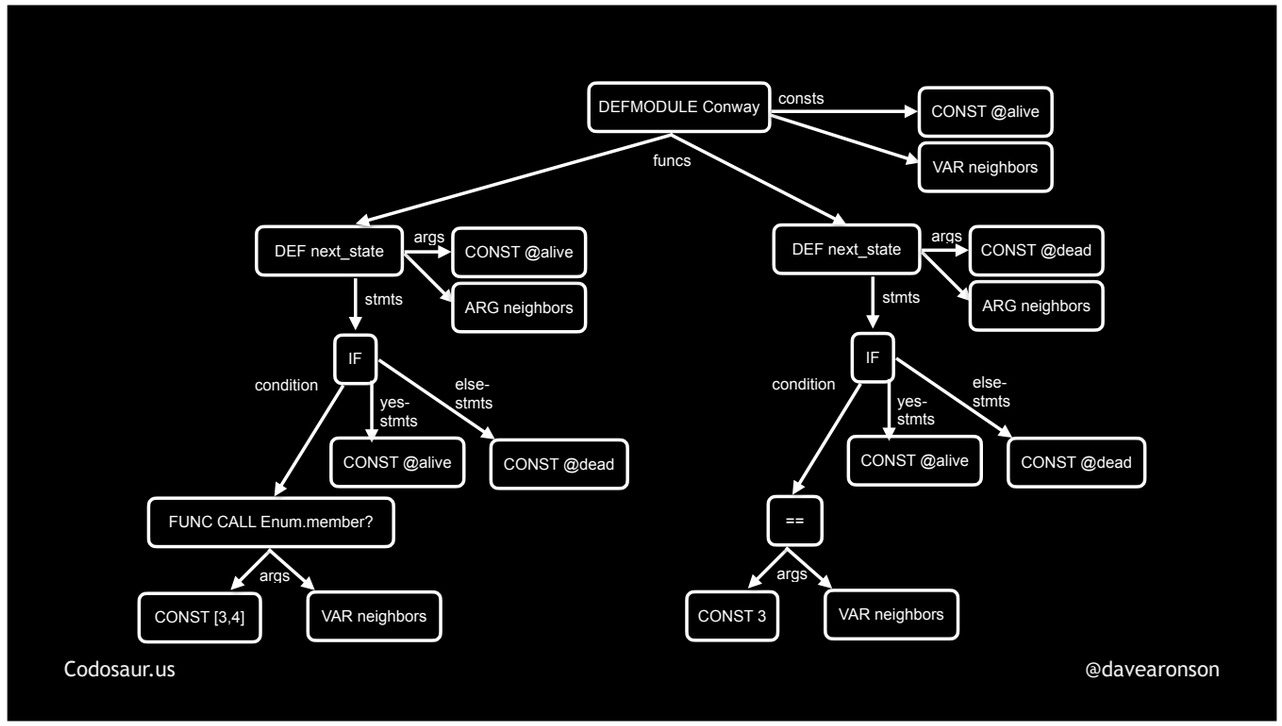
```
defmodule Conway do
  @alive "*"
  @dead  " "

  def next_state(@alive, neighbors),
    do: if Enum.member?([3, 4], neighbors),
      do: @alive, else: @dead

  def next_state(@dead, neighbors),
    do: if neighbors == 3,
      do: @alive, else: @dead
end
```

Codosaur.us @davearonson

. . . parses our code, usually into an Abstract Syntax Tree, so that this code becomes . . .



. . . this AST. I'm going to assume that you're familiar with the concept of an AST, but don't worry about understanding this one in detail.

After the tool creates an AST out of our code, then it . . .

Codosaur.us Image: https://www.needpix.com/photo/download/667144/cat-tree-climb-young-cat-pet-nature-cat-in-the-tree-domestic-cat-in-the-free @davearonson

. . . traverses the tree, looking for sub-trees, or branches if you will, that represent each function. After finding *them*, it handles them as I described before, starting with looking for each one's *tests*, but how does it do *that?* That usually relies mainly on us developers, either . . .

```
# @mumu tests-for foo
test "#foo turns 3 into 6" do
  foo(3).must_equal 6
end

test "#foo turns 4 into 10" do
  foo(4).must_equal 10
end
```

Codosaur.us

@davearonson

. . . annotating our tests, as I hinted at earlier, or following some kind of . . .

```
test "#foo turns 3 into 6" do
  foo(3).must_equal 6
end

test "#foo turns 4 into 10" do
  foo(4).must_equal 10
end
```

Codosaur.us @davearonson

. . . convention in naming the tests, the files, or perhaps both. These manual techniques are often supplemented and sometimes even replaced by . . .

```
test "#foo turns 3 into 6" do
  foo(3).must_equal 6
end

test "#foo turns 4 into 10" do
  foo(4).must_equal 10
end
```

Codosaur.us

@davearonson

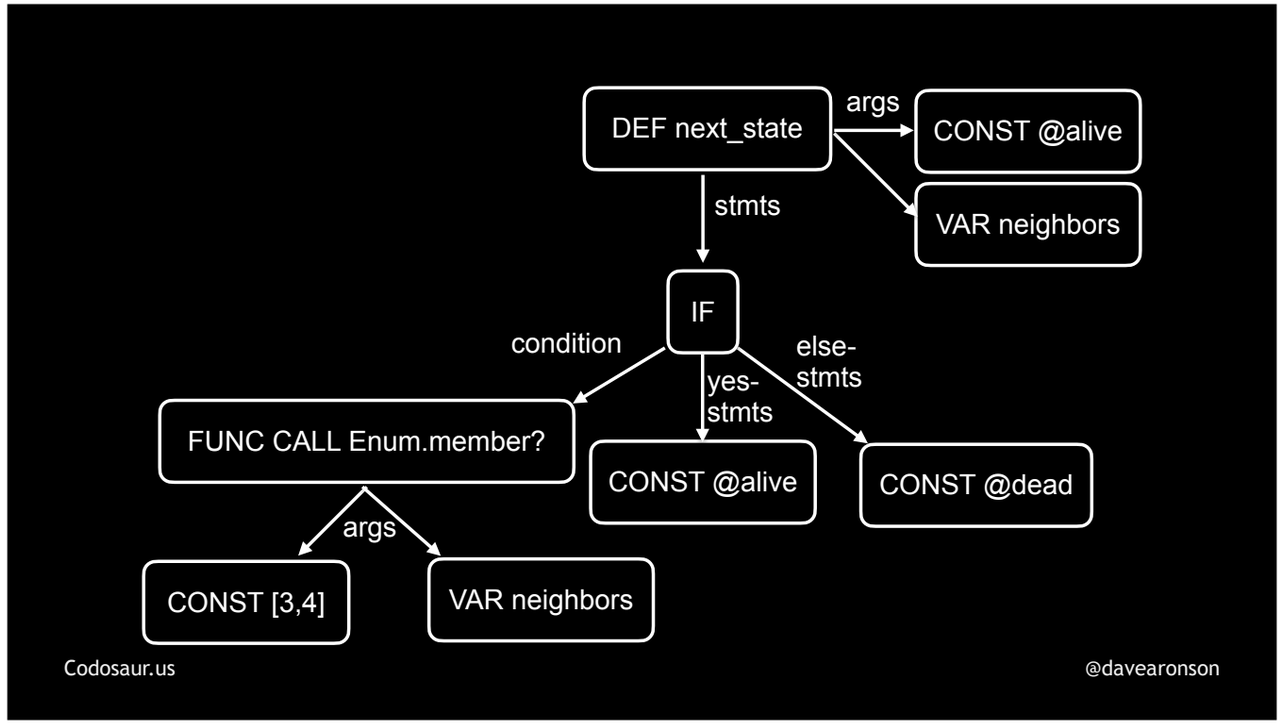
. . . the tool looking at what tests call what functions. After the tool has found the function's tests, then, assuming it won't skip this function because it *didn't* find any tests, it makes the mutants. To make mutants *from* an AST subtree, it . . .

Codosaur.us

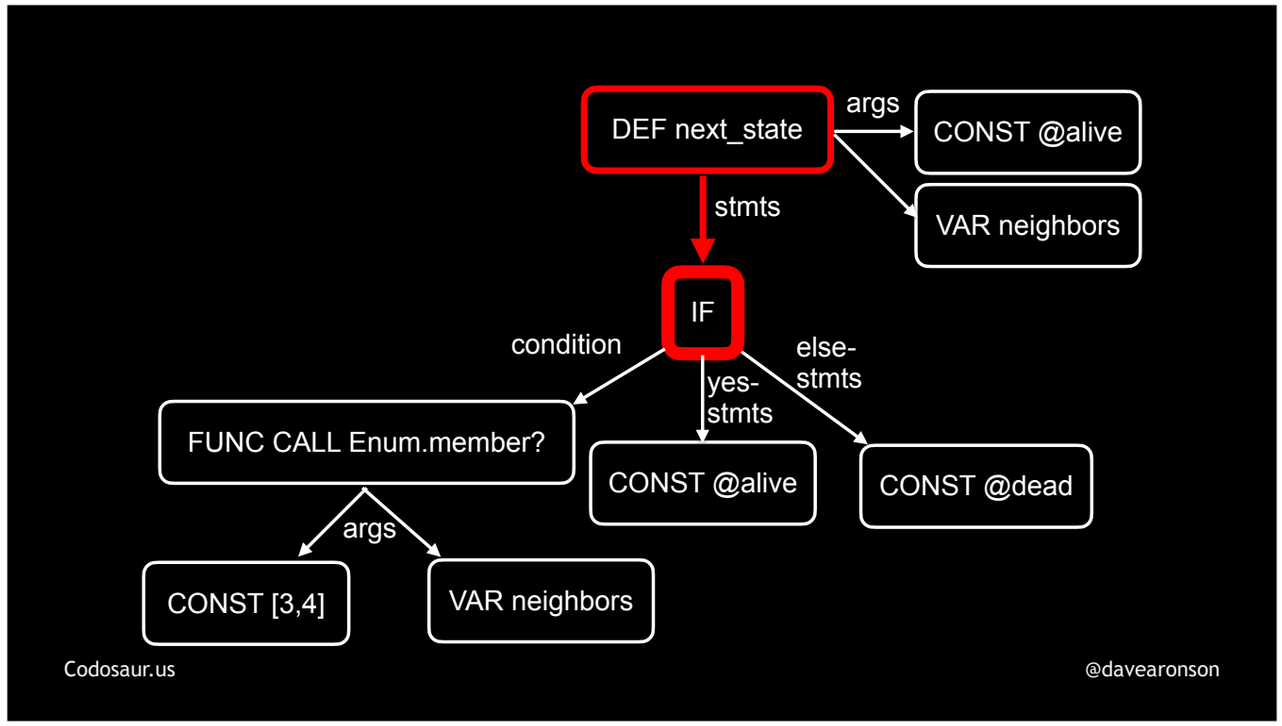
Image: https://pxhere.com/en/photo/1230969

@davearonson

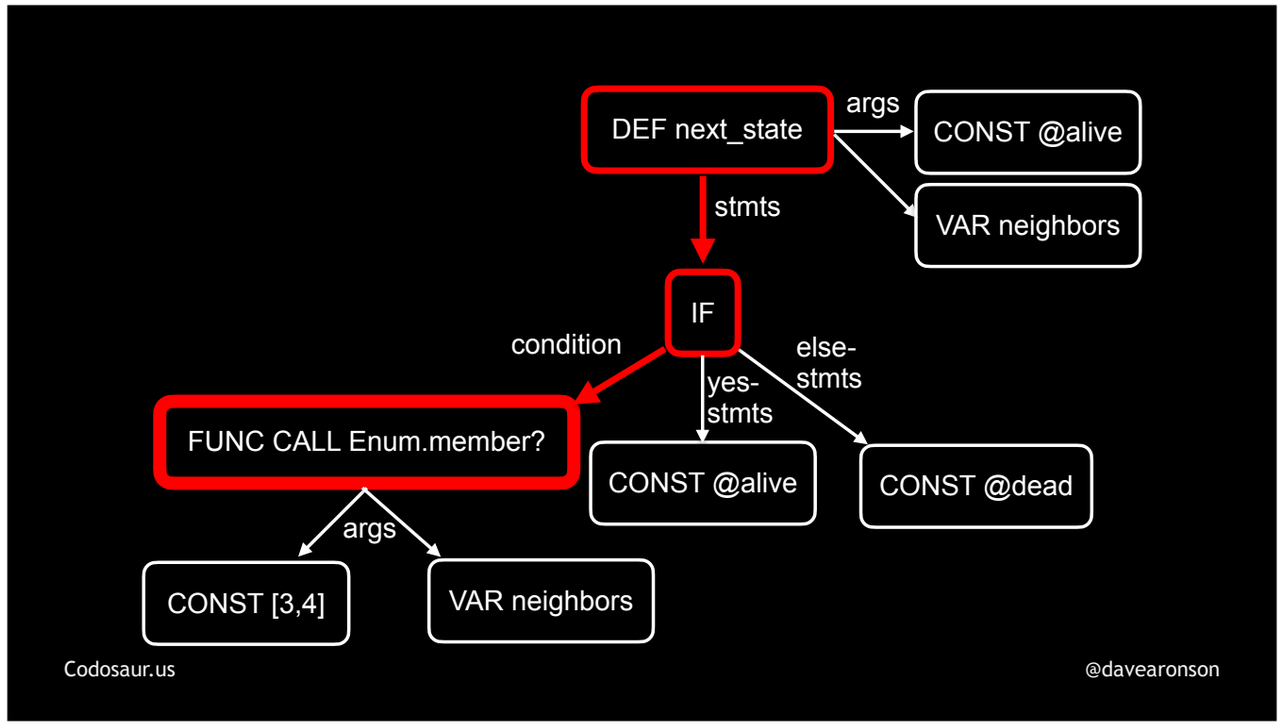
. . . traverses that subtree, just like it did to the whole thing. However, now, instead of looking for even *smaller* subtrees it can *extract*, like twigs or something, it looks for *nodes* where it can *change* something. Each time it finds one, then for each way it can change that node, it makes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



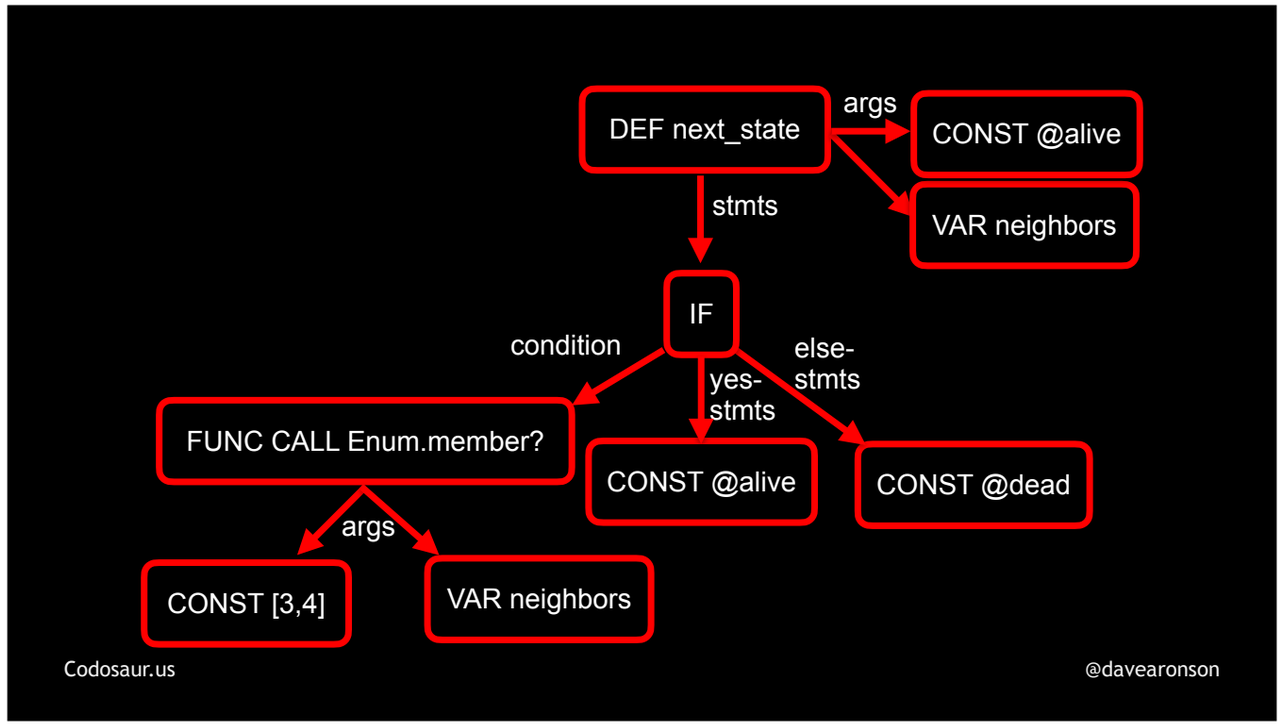
. . . one of the function subtrees from the AST I showed earlier, and has only gotten down to . . .



. . . this if statement. For each way the tool could change that node, it would make a fresh copy, of this whole subtree, with only that one node changed, in that one way. After it's done making as many mutants as it can from *that* node, it would continue traversing the subtree, down to . . .

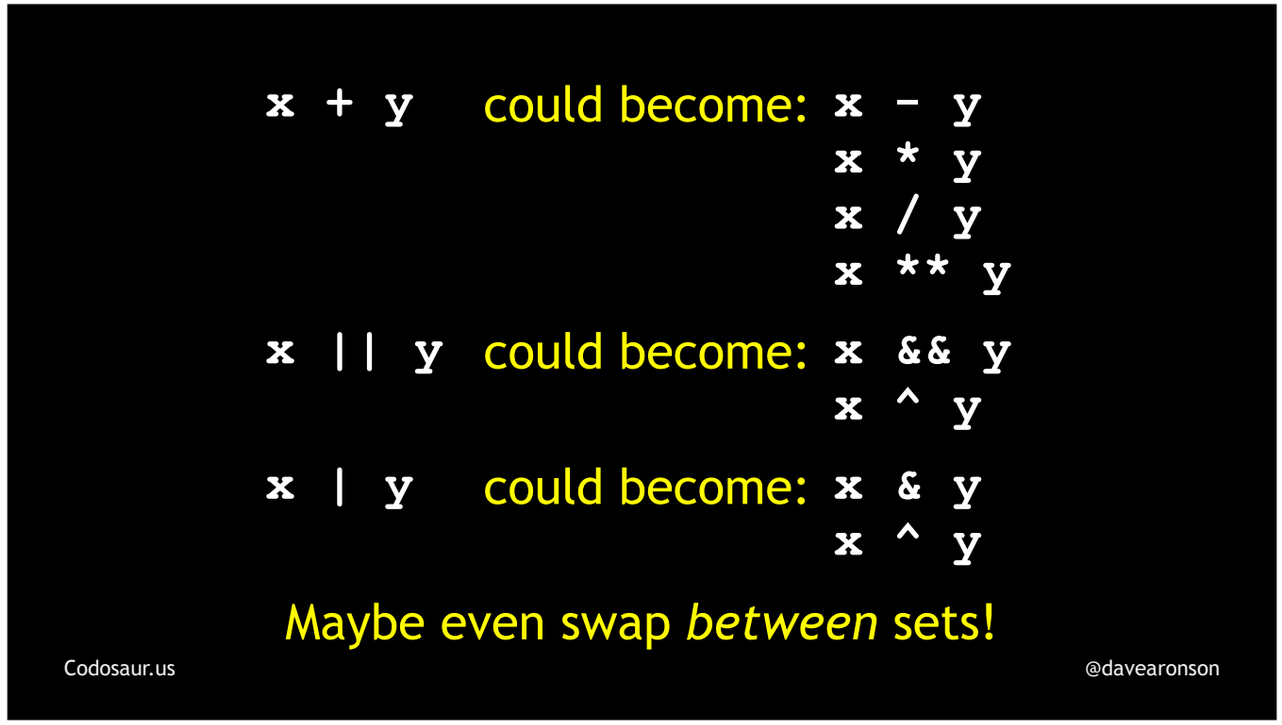


. . . another node. Again, for each way it could change *that* node, it would make a copy of this whole subtree, with only that mutation. And so on, until it has . . .



. . . traversed the entire subtree.

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



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

In languages and situations where we can do so, it could even substitute an operator from a different category. For instance, in many languages, we can treat *anything* as *booleans*, so x *times* y could become, for instance, x *and* y, or x *exclusive-or* y.

`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, such as in subtraction, division, exponentiation, or string concatenation, it could *swap* them.

```
x < y
```

could become:

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

Codosaur.us

@davearonson

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

`x`

could become:

`-x`
`!x`
`~x`

. . . or vice-versa!

Codosaur.us

@davearonson

It could insert *or remove* a mathematical, logical, or bitwise *negation*.

```
a = foo(x)
b = bar(y)
```

could become:

```
a = foo(x)
```

or

```
b = bar(y)
```

Codosaur.us

@davearonson

It can remove entire lines of code, though usually because it's a *statement*, rather than looking at the *physical written lines* of the source code.

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

could become:

```
foo(z)
```

Codosaur.us

@davearonson

It can remove a condition, so that something that might be skipped or done, is always done.

```
def f(x, y), do: x * y
```

could become:

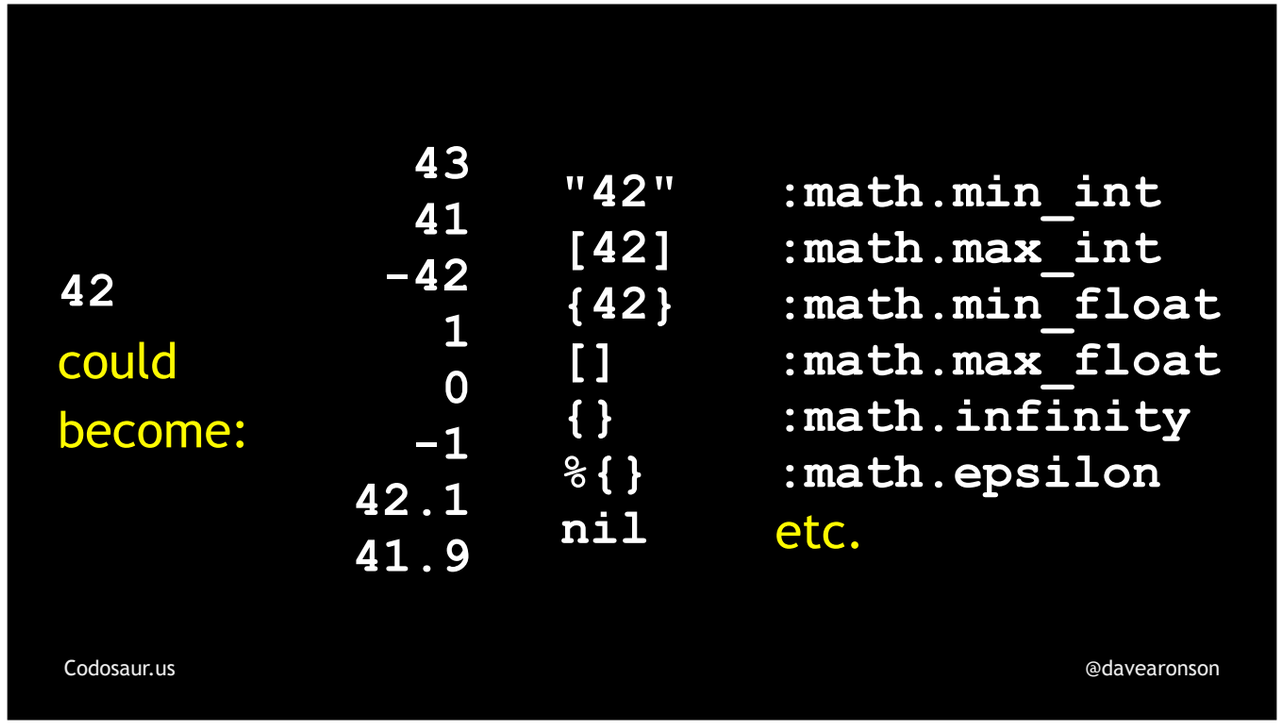
```
def f(x, y), do: 0
def f(x, y), do: :math.max_int
def f(x, y), do: "a string"
def f(x, y), do: nil
def f(x, y), do: x
def f(x, y), do: fail(“boom")
def f(x, y), do: # nothing
```

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, if the language permits, as many do.



It could change a value item to some other value, such as these, and many more but I had to stop somewhere. (Some of those math constants don't really exist on the BEAM, they're there just for illustration.) It could even change it to something of an entirely different and incompatible type, such as changing a number into a, if I may quote . . .

Codosaur.us

Image: https://www.flickr.com/photos/herval/50674160

@davearonson

. . . Smeagol, “string, or nothing!”

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

From here on, there are no more low-level details I want to add, so let’s peel back one last layer, and look at . . .

```
for function <- find_functions(Application) do
  tests = find_tests(function)
  if Enum.none?(tests) do
    warn_about_no_tests(function)
    next function
  end
  for mutant <- make_mutants(function) do
    for test <- tests do
      if (fails(test, with_code: mutant)) do
       next mutant
      end
    end
    report_as_surviving(mutant)
  end
end
```

Codosaur.us @davearonson

. . . some Elixir-flavored pseudocode illustrating how it works. I'll talk a bit so you have some time to take pictures. Never mind how it goes on to the next function or next mutant, I said it was pseudocode, not real Elixir, and doing it in proper recursive functional style, rather than for-loops, wouldn't fit well on one slide.

Everybody done taking pictures?

Now let's *finally* walk through some *examples!* We'll start with an easy one. Suppose we have a function . . .

```
def power(x, y) do
  x ** y
end
```

Codosaur.us @davearonson

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

Think about what a mutant made from this might *return*, since that's what our tests would probably be looking at.

Mainly it could return results such as . . .

```
x + y                   :math.min_int
x - y                   :math.max_int
x * y                   :math.max_float
x / y                   :math.min_float
y ** x                  :math.infinity
x                       :math.epsilon
y                       raise(DeliberateError)
0                       "some random string”
1                       []
-1                      {}
0.1                     %{}
-0.1                    nil
```

Codosaur.us @davearonson

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

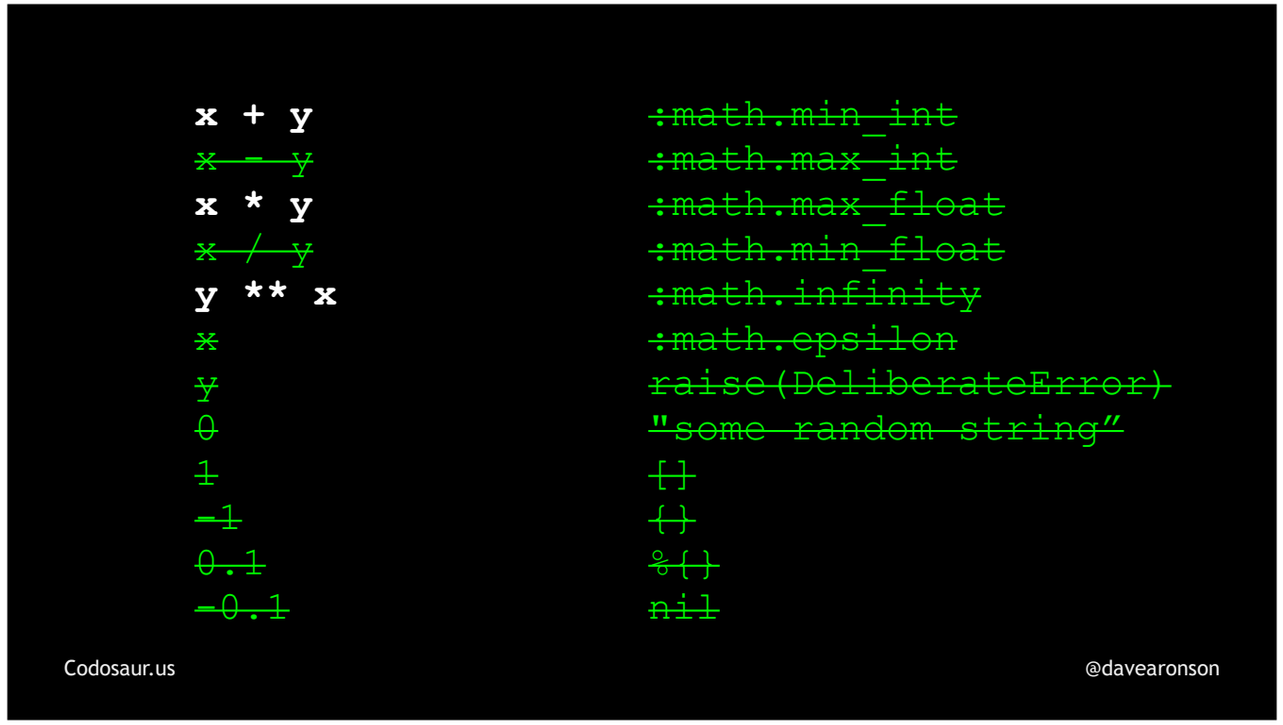
Now suppose we had only one test . . .

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

Codosaur.us

@davearonson

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



. . . 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, and other significant numbers. Subtracting gets us zero, dividing gets us one, returning either argument alone gets us two, and the error conditions will at *least* make the test not pass. But . . .

. . . addition, multiplication, and exponentiation in the reverse order, all get us the correct answer. Mutants based on *these* mutations will therefore *"surivive"* this test.

So how do we see that happening? When we run our tool, it gives us a report, that looks roughly like . . .

```
function "power" (demo.ex:42)
has 4 surviving mutants:

42 - def power(x, y) do
42 + def power(y, x) do

43 -   x ** y
43 +   x + y

43 -   x ** y
43 +   x * y

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

Codosaur.us

@davearonson

. . . this. The exact words, format, amount of context, etc., will depend on exactly which tool we use, but the information should be pretty much the same.

To fully unpack this, it's saying that if we changed . . .

```
function "power" (demo.ex:42)
has 4 surviving mutants:

42 - def power(x, y) do
42 + def power(y, x) do

43 -   x ** y
43 +   x + y

43 -   x ** y
43 +   x * y

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

Codosaur.us

@davearonson

. . . the function called power, which is in . . .

```
function "power" (demo.ex:42)
has 4 surviving mutants:

42 - def power(x, y) do
42 + def power(y, x) do

43 -   x ** y
43 +   x + y

43 -   x ** y
43 +   x * y

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

Codosaur.us

@davearonson

. . . file demo.ex, and starts at line 42 . . .

```
function "power" (demo.ex:42)
has 4 surviving mutants:

42 - def power(x, y) do
42 + def power(y, x) do

43 -   x ** y
43 +   x + y

43 -   x ** y
43 +   x * y

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

Codosaur.us

@davearonson

. . . in any of four different ways, then all its tests would still pass, and those four ways are: . . .

```
function "power" (demo.ex:42)
has 4 surviving mutants:

42 - def power(x, y) do
42 + def power(y, x) do

43 -   x ** y
43 +   x + y

43 -   x ** y
43 +   x * y

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

Codosaur.us

@davearonson

. . . to change line 42 to swap the arguments, or . . .

```
function "power" (demo.ex:42)
has 4 surviving mutants:

42 - def power(x, y) do
42 + def power(y, x) do

43 -   x ** y
43 +   x + y

43 -   x ** y
43 +   x * y

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

Codosaur.us

@davearonson

. . . change line 43 to change the exponentiation into addition or multiplication, or . . .

```
function "power" (demo.ex:42)
has 4 surviving mutants:

42 - def power(x, y) do
42 + def power(y, x) do

43 -   x ** y
43 +   x + y

43 -   x ** y
43 +   x * y

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

Codosaur.us

@davearonson

. . . to change line 43 to to swap the operands.

So what is . . .

```
function "power" (demo.ex:42)
has 4 surviving mutants:

42 - def power(x, y) do
42 + def power(y, x) do

43 -   x ** y
43 +   x + y

43 -   x ** y
43 +   x * y

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

Codosaur.us @davearonson

. . . this set of surviving mutants trying to tell us? The very high level message is that our test suite is not sufficient, either because there aren't enough tests, or the ones we have just aren't very good, or both. But we knew that! We know it's probably not that there is redundant or unreachable code, as we can look at the code and see that those are unlikely.

The question boils down to, how are these mutants surviving? The usual answer is that . . .

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

Codosaur.us @davearonson

. . . they return 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, we can take a closer look, at one mutant, and a test it passes. Let's start with . . .

the change:

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

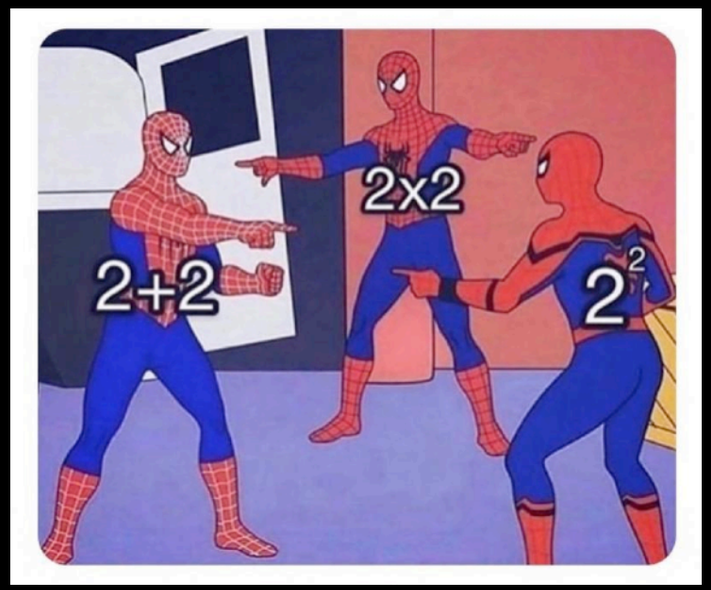
our test:

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

Codosaur.us

@davearonson

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



Codosaur.us

Image: meme going around, original source unfindable, sorry

@davearonson

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

So how can we *kill* . . .

the change:

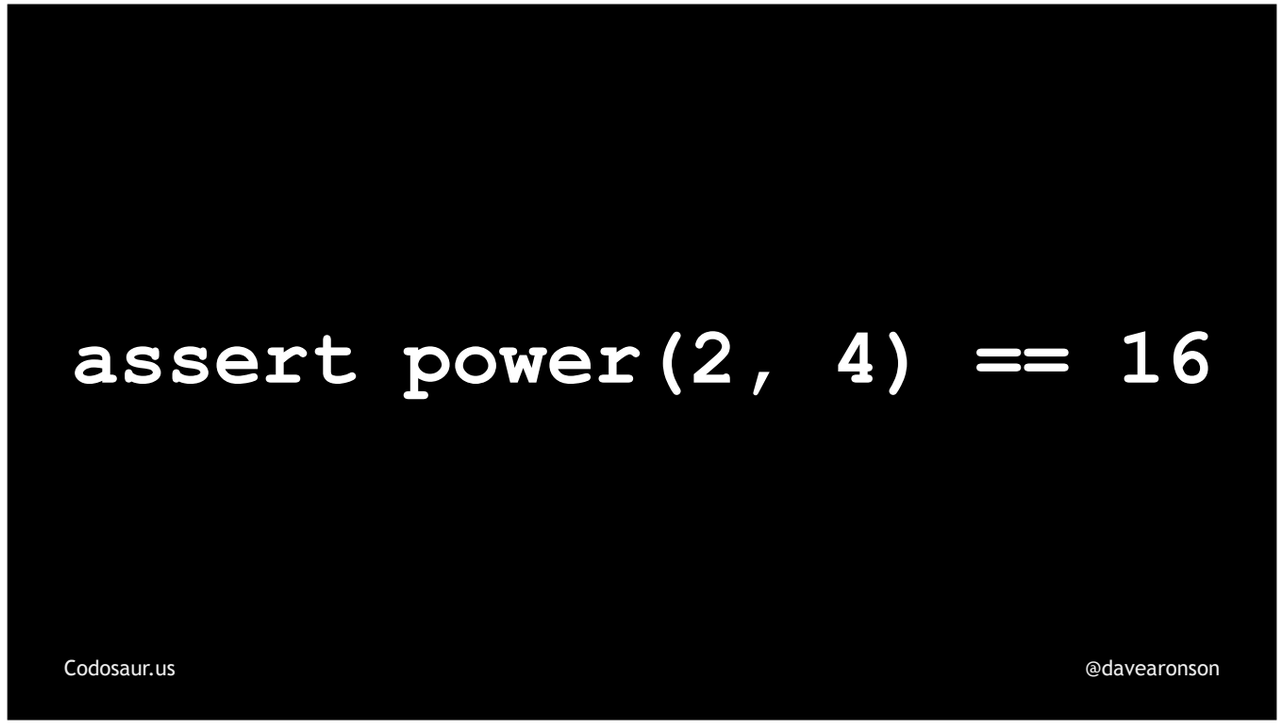
```
43 -    x ** y
43 +    x + y
```

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? It's quite simple in this case. We need to make at least one test use inputs such that x *to the* y is different from x *plus* y. For instance, we could add a test or change our test to . . .



. . . assert that two to the *fourth* power is *sixteen*. All the mutants that our original test killed, this would still kill. Two *plus* four is six, not six*teen*, so this should kill the plus mutant just fine. For that matter, two *times* four is eight, which is *also* not sixteen, so this should kill the "times" mutant as well.

However, . . .

. . . the (ahem) pair of argument-swapping mutants survive! That’s okay, we can . . .

. . . attack them separately, no need to kill all the mutants at once 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, guess what, not eight, so the "plus" and "times" mutants *stay* dead, and we don't get any . . .

. . . zombie mutants wandering around, even if we still had only one test. (PAUSE!) With . . .

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

Codosaur.us

@davearonson

. . . these inputs, the correct operation is the only simple common one that yields the correct answer. This isn't the *only* solution, though; we could have used two to the *fifth*, *three* squared, three to the fifth, vice-versa, and infinitely more. There are *lots* of ways to skin . . .

Codosaur.us

Image: https://www.flickr.com/photos/greyloch/48214242842

@davearonson

. . . *that* flerken!

This may make mutation testing sound . . .

Codosaur.us

Image: https://commons.wikimedia.org/wiki/File:Simple_Simon_LCCN2003677693.jpg

@davearonson

. . . simple, but this is a downright trivial example, so we could easily think up arguments to make *all* mutants, within reason, behave differently from the original code.

So let’s look at a more *complex* example!

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

```
def send_message(_, 0, acc), do: acc
def send_message(buf, len, acc) do
  sent = send_bytes(buf, len)
  send_message(Enum.drop(buf, sent),
               len - sent,
               acc + sent)
end
# if < 0 we get no-matching-clause err
```


. . . like so. This function, send_*message*, sends as much data as send_*bytes* can handle in one chunk, recursing until the message is all sent. This is a very common pattern in communication software.

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

```
  def send_message(_, 0, acc), do: acc
  def send_message(buf, len, acc) do
    sent = send_bytes(buf, len)
-   send_message(Enum.drop(buf, sent),
-                len - sent,
-                acc + sent)
  end
  # if < 0 we get no-matching-clause err
```

Codosaur.us @davearonson

. . . this, removing those lines with the minus signs, an example of removing a statement.

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

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

If a mutant that only goes through . . .

```
def send_message(_, 0, acc), do: acc
def send_message(buf, len, acc) do
  sent = send_bytes(buf, len)
  send_message(Enum.drop(buf, sent),
               len - sent,
               acc + sent)
end
# if < 0 we get no-matching-clause err
```

Codosaur.us @davearonson

. . . that function once, acts the same as our normal code, as far as our tests can tell, that means that our *tests* are only making our code go through that function once. So what does that mean? (PAUSE!) By the way, you'll find that interpreting mutants involves a lot of asking *"so what does that mean"*, often 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 the most *likely* is that we simply didn't 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
```

Codosaur.us @davearonson

. . . suppose our maximum chunk size, what send_*bytes* can handle in one chunk, is 10,000 bytes. However, for whatever reason, . . .

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 tiny little *three* byte message. Or maybe four if our protocol includes a null terminator, maybe a bit more depending just how it's serialized, but almost certainly not 10_000. (PAUSE!)

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

in module Network:

```
@max_chunk_size 10_000
```

in test_send_message:

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

Codosaur.us @davearonson

. . . here. (PAUSE!) We just take the maximum size, add some, and construct that big a message.

But maybe we tested with the *largest* permissible message, out of a set of predefined messages, or at least message *sizes*. For instance, . . .

in module Message:

```
@SmallMsgSize 1_000
@LargeMsgSize 5_000
```

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? What is this mutant trying to tell us in this case? (PAUSE!)

It’s trying to tell us that a version of send_message with the recursion removed will do the job just fine. If we remove the recursion, we wind up with . . .

```
def send_message(_, 0, acc), do: acc
def send_message(buf, len, acc) do
  sent = send_bytes(buf, len)
end
# if < 0 we get no-matching-clause err
```


. . . this. You've probably figured out the ultimate message at a glance, but play along. We'll probably get some compiler warnings out of this, so let's do some trivial improvements, by removing the unused parameter and assignment. That gets us to . . .

```
def send_message(_, 0, acc), do: acc
def send_message(buf, len, _acc) do
  send_bytes(buf, len)
end
# if < 0 we get no-matching-clause err
```

Codosaur.us @davearonson

. . . this, which makes it even clearer: the *entire* send_message *function* may well be *redundant*, so we can just use send_*bytes directly!* It might not be, though, because, in real-world code, there may be some logging, error handling, and so on, needed in send_message, but at least the recursion was redundant. Fortunately, when it's this kind of problem, with unreachable or redundant code, the 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.

Now that we've seen a few different examples, of spotting bad tests and redundant code, I'd like to address some . . .

. . . occasionally asked questions. (Mutation testing is still rare enough that I don't think there *are* any *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 anyway? Mutation testing has a surprisingly . . .

Codosaur.us

Image: https://pixabay.com/photos/egypt-education-history-egyptian-1826822/

@davearonson

. . . long history -- at least in the context of computers. It was first proposed in 1971, in Richard Lipton's term paper titled “Fault Diagnosis of Computer Programs”, at Carnegie-Mellon University. The first *tool* didn't appear until nine years *later*, in 1980, as part of Timothy Budd's PhD work at Yale. Even so, it was not *practical* for most people, with consumer-grade computers, until recently, maybe the past couple decades, with advances in CPU *speed*, multi-*core* CPUs, larger and cheaper memory, and so on.

That leads us to the next question: *why* is it so CPU-intensive? To answer that, we need do some math, but don't worry, it's pretty basic. Suppose our functions have, on average, . . .

10 lines

Codosaur.us

@davearonson

. . . about ten lines each. And each line has about . . .

10 lines

x 5 mutation points

Codosaur.us

@davearonson

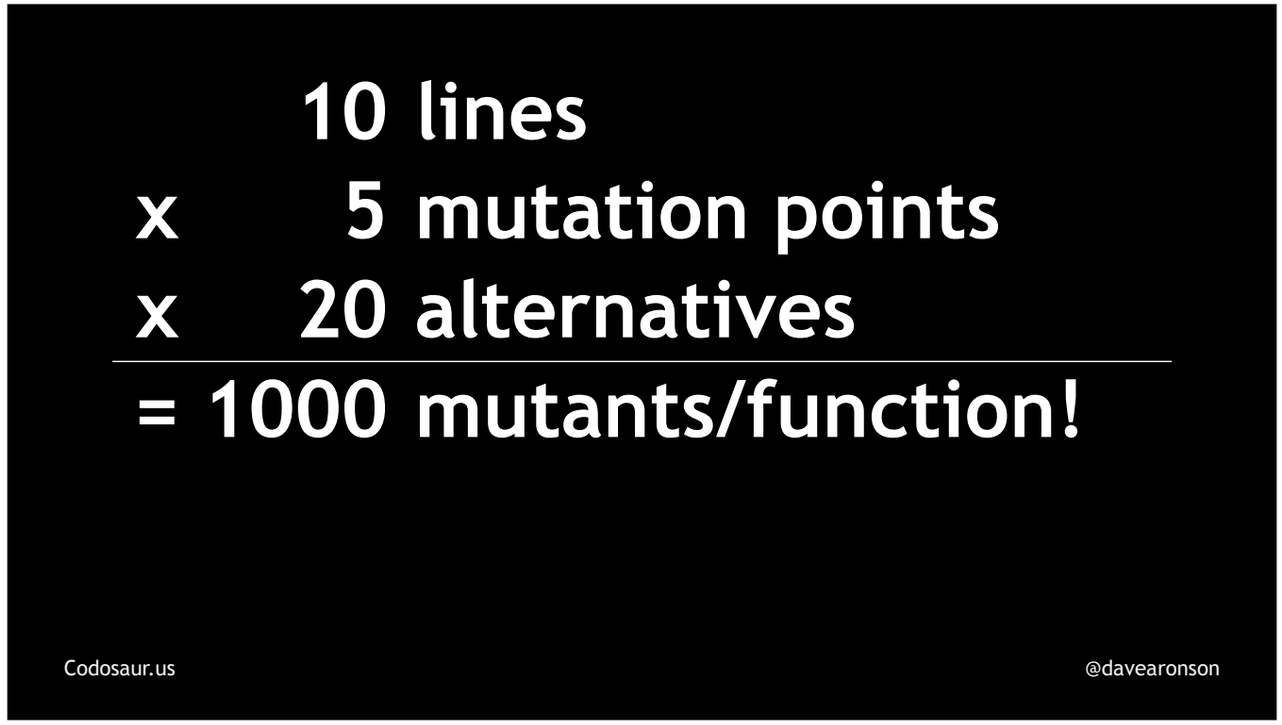
. . . five places where it can be mutated, to any of about . . .

```
      10 lines
x      5 mutation points
x     20 alternatives
```

Codosaur.us

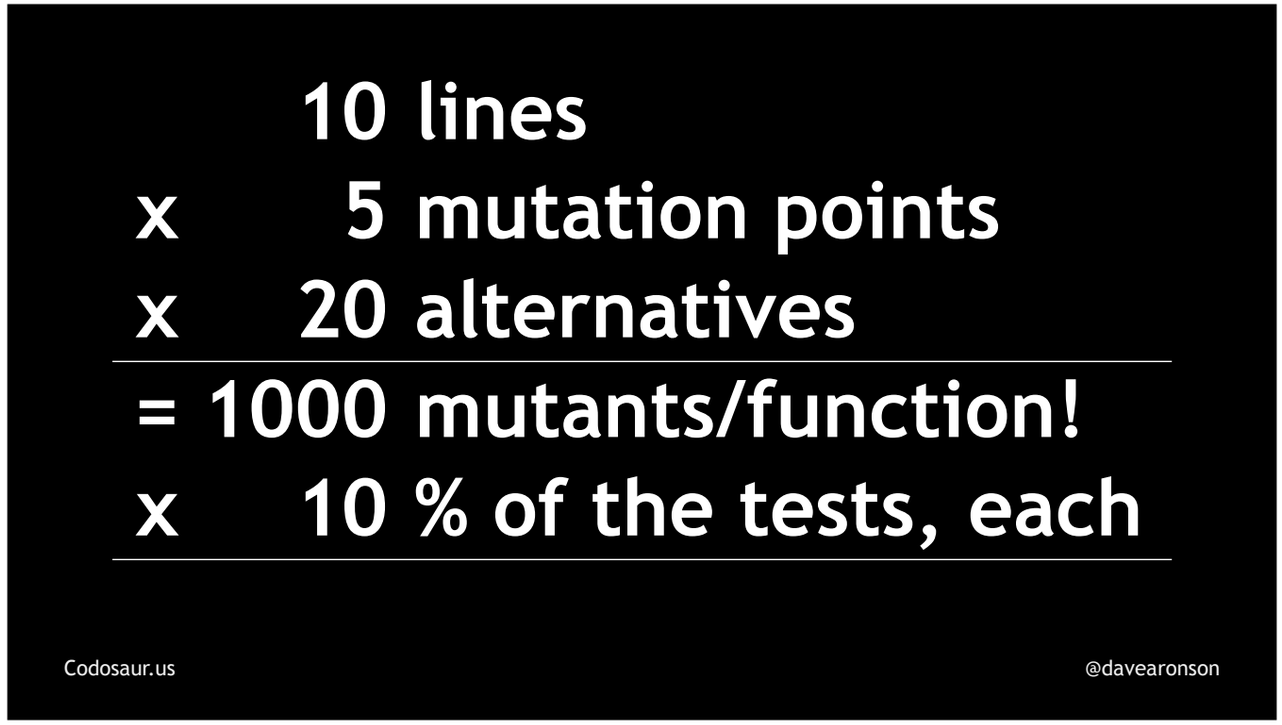
@davearonson

. . . twenty alternatives. That works out to about . . .

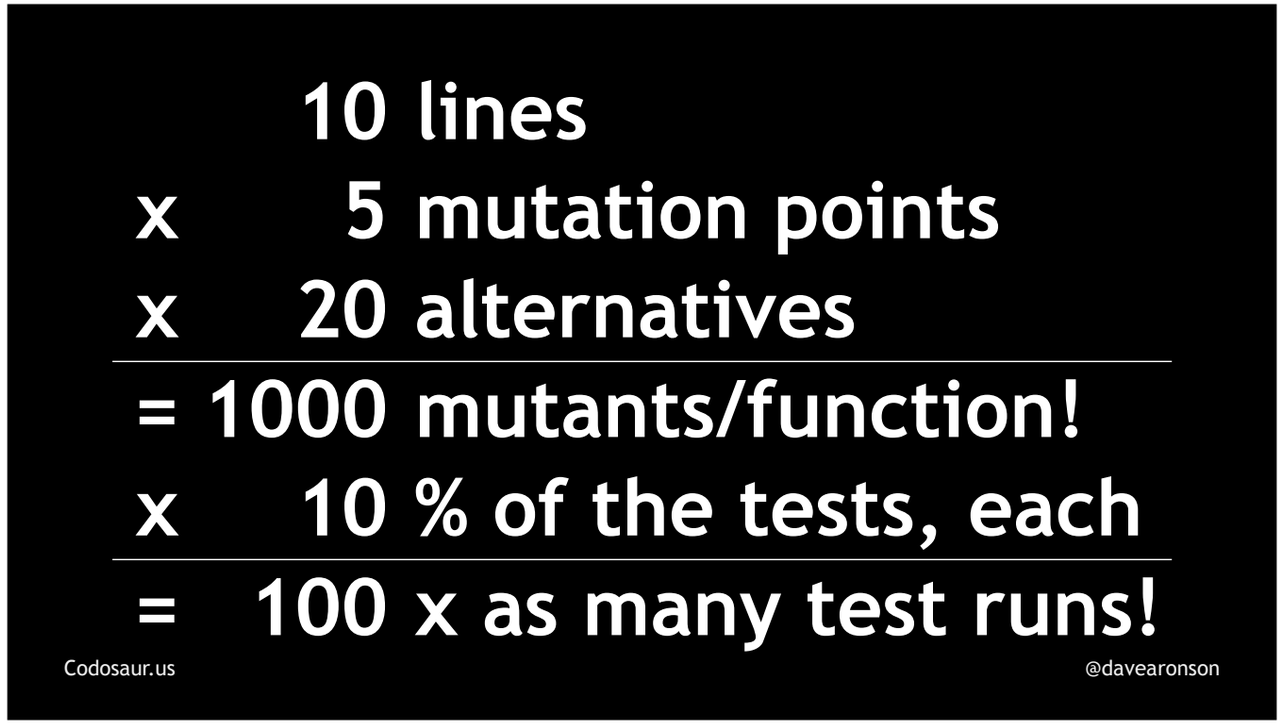


. . . a thousand mutants for each function! And for each one, we'll have to run somewhere between one test, if we're lucky and kill it on the first try, and *all* of that function's tests, if we kill it on the last try, or worse yet, it survives.

Suppose we wind up running just . . .



. . . one *tenth* of the tests for each mutant. Since we start with a thousand mutants, that's still . . .

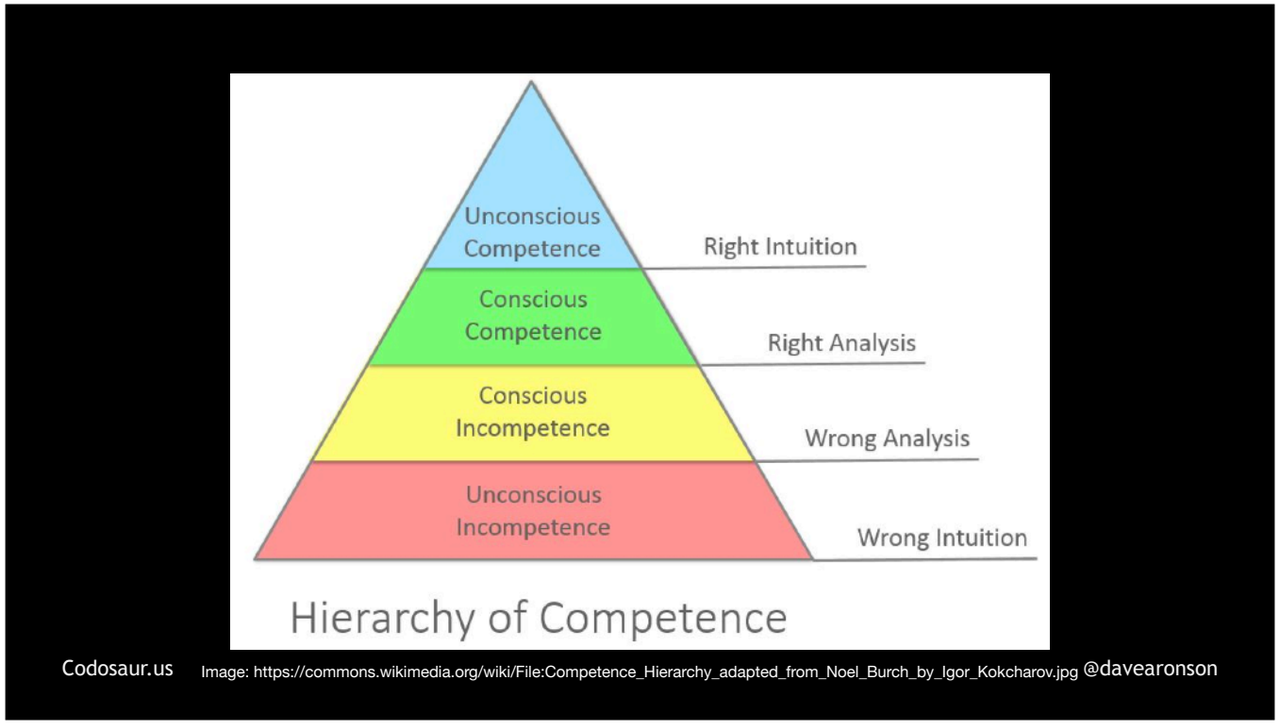


. . . a *hundred times* the test runs for that function. If our test *suite* normally takes a zippy ten seconds, mutation testing will take about a *thousand* seconds. That might not sound like much, because I'm saying "seconds", but do the math and it's *almost 17 minutes!*

The last question is, when making each mutant, why change it in only . . .

. . . one way?

There are multiple reasons. First off, the main theoretical underpinning is . . .



Codosaur.us Image: https://commons.wikimedia.org/wiki/File:Competence_Hierarchy_adapted_from_Noel_Burch_by_Igor_Kokcharov.jpg @davearonson

. . . the Competent Programmer hypothesis. Let's give that a quick check. Raise your hand if you're competent! (PAUSE!) Okay, looks like most of us. The rest of you, you probably really are competent, so you might want to read up on Impostor Syndrome.

But anyway, what is the Competent Programmer Hypothesis? Long story short, it's the idea that we generally have a pretty good clue what we're doing, and when we make a mistake, it's usually a single small mistake, like adding when we should subtract, or saying "less than or equal" when we mean "strictly less than", or greater than, or whatever. Does this kind of simple substitution sound familiar? It's exactly what a mutation testing tool does. We can think of mutation testing as sort of a "did you mean" function, like how Google suggests something else if our search didn't have many hits.

There are also practical considerations. For one, it helps us poor humans . . .

Codosaur.us

Image: https://pixabay.com/vectors/arrow-one-way-right-sign-road-759223/

@davearonson

. . . FOCUS! It's much easier to tell what a surviving mutant is trying to say, if we're only talking about one thing in the first place. Another reason is that multiple changes may . . .

. . . balance each other out, leading to more false alarms. Allowing multiple mutations would also create a combinatorial . . .

. . . explosion of mutants, with the tool making *orders of magnitude* more mutants per function, which would make it even *more* CPU-intensive.

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

😀 Ensures our code is meaningful

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

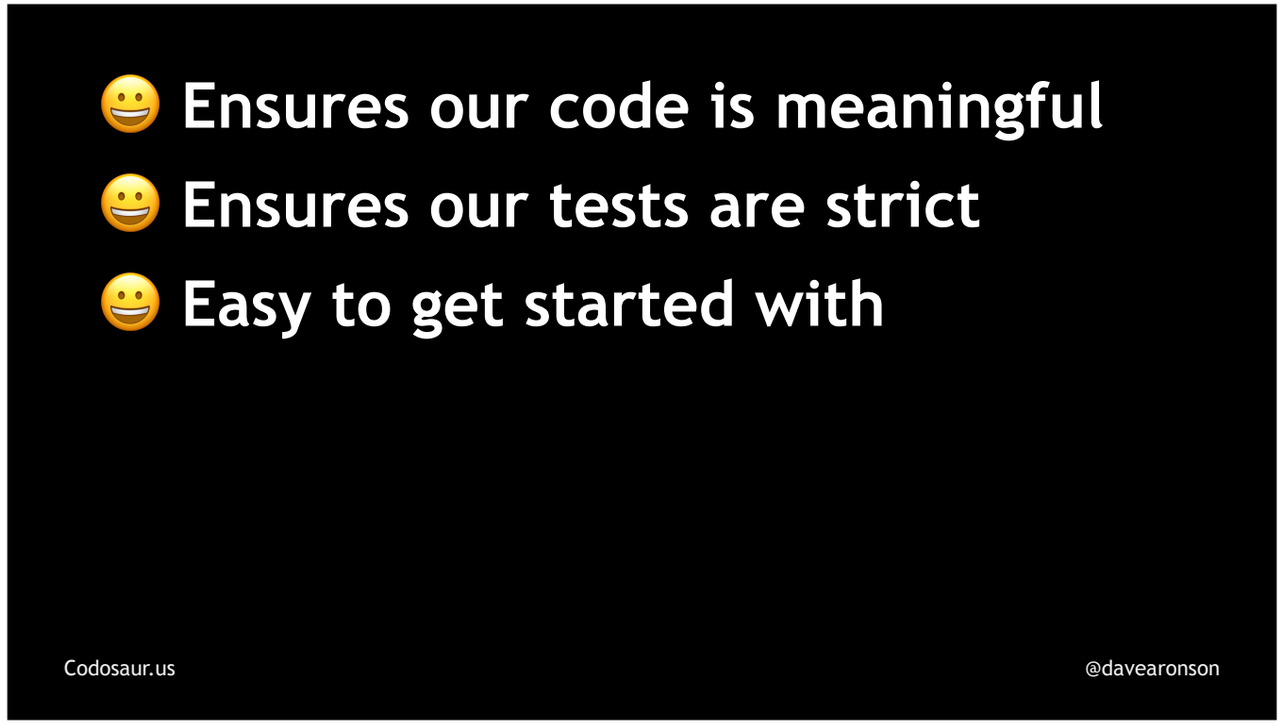
😀 Ensures our code is meaningful

😀 Ensures our tests are strict

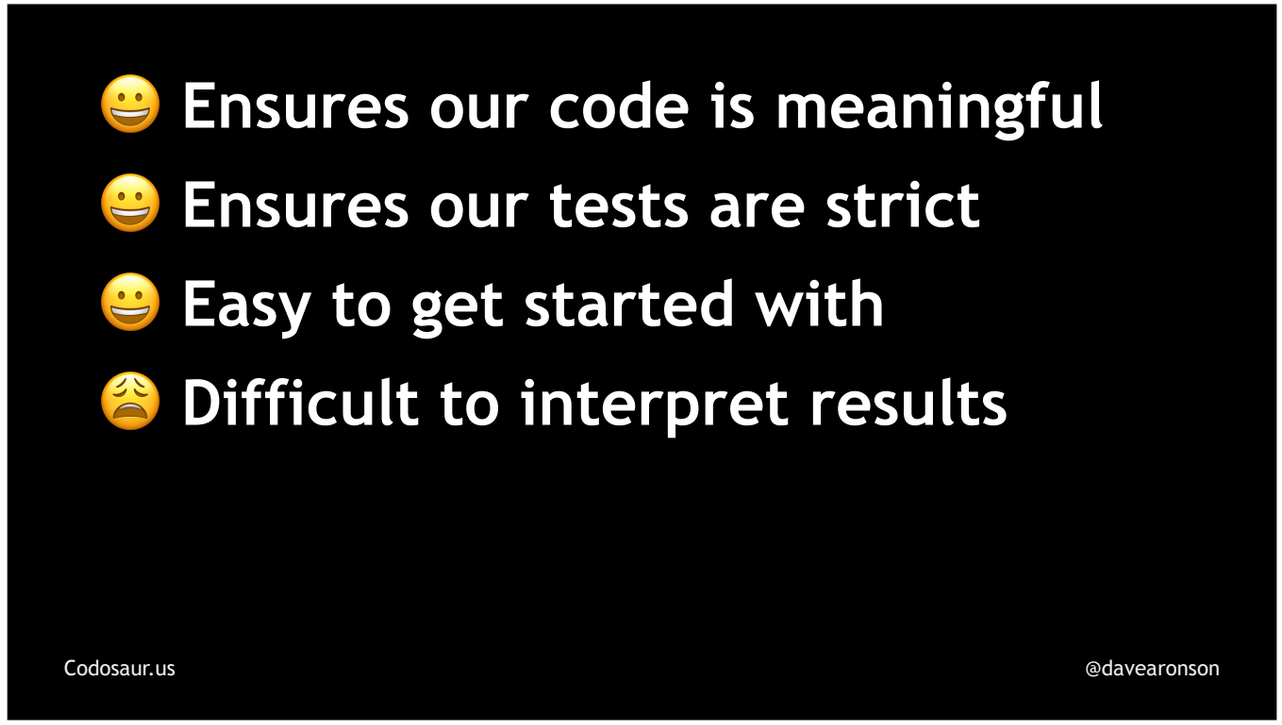
Codosaur.us

@davearonson

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



easy to get started with, in terms of setting up most of the tools and annotating our tests if needed
(which may be *tedious* but at least it's *easy*),
but it's . . .



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

😀 Ensures our code is meaningful

😀 Ensures our 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's not a good fit for our particular current projects, though,
I still think it's just . . .

😀 Ensures our code is meaningful

😀 Ensures our tests are strict

😀 Easy to get started with

😩 Difficult to interpret results

😩 Hard labor on the CPU

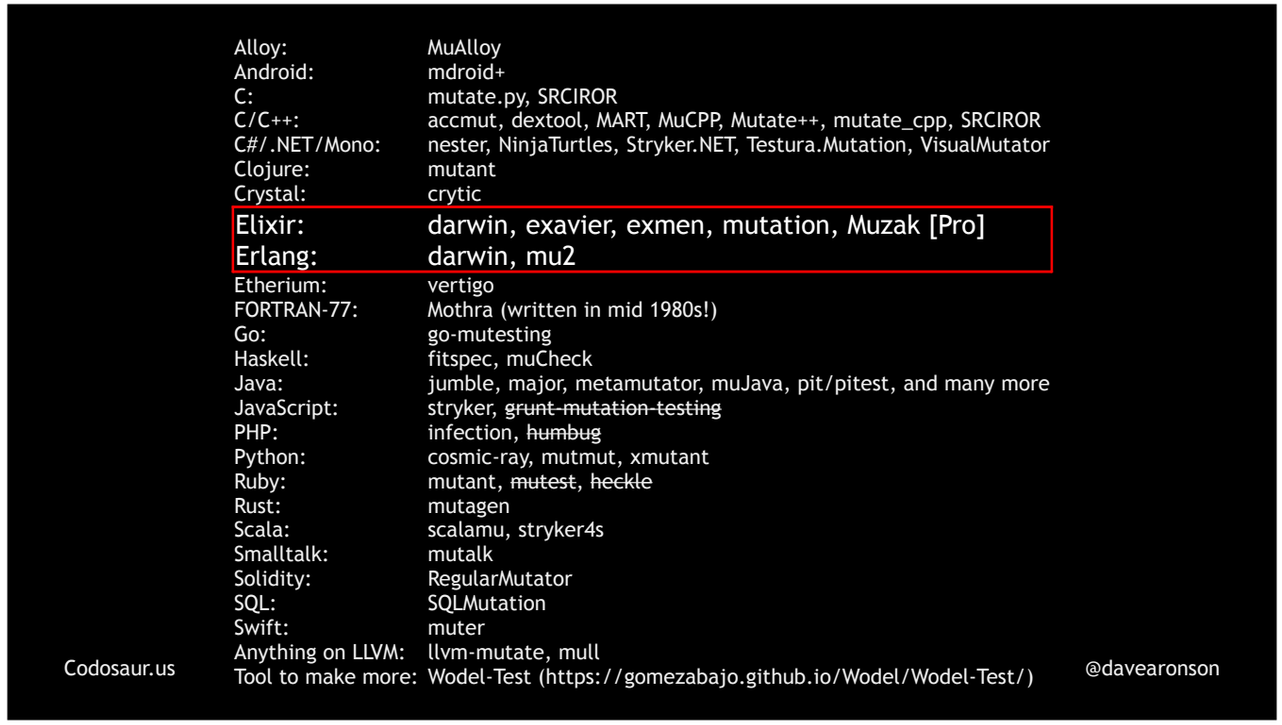
😎 Fascinating concept! 🤓

Codosaur.us

@davearonson

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

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



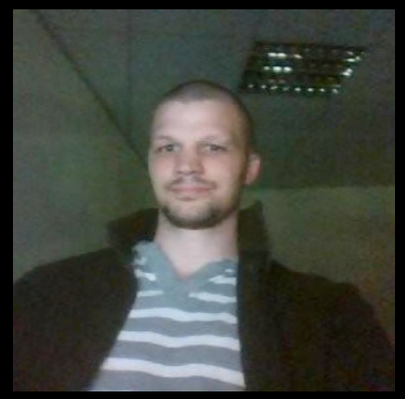
. . . here is a list of tools for some popular languages and platforms . . . and some others; I doubt many of you are doing FORTRAN-77 these days. I'll talk a bit so you can take pictures, especially since it’s much too small to read easily. Just be aware that many of these are outdated; I don't know or follow quite *all* of these languages and platforms. The ones I *know* are outdated, are crossed out. There's also a promising tool called Wodel-Test, a *language-independent* mutation engine, with which you can make language-specific mutation testing tools. (Sorry, I haven't looked into it much myself.)

For Elixir, there are exavier, exmen, mutation, and a fairly new one called Muzak, in both Pro and free versions; for Erlang, there is mu2; and another fairly new one called darwin does both.

Is everybody done taking pictures? Before we get to Q&A, I'd like to give a shoutout to . . .

. . . Toptal, a consulting network I'm in, whose Speakers Network helped me prepare and practice previous versions of this presentation. (Please use that referral link if you want to hire us or join us, and that's also where that QR code goes.)

Also, many thanks to . . .



Thank you Markus Schirp!

https://github.com/mbj

Codosaur.us

Images: Markus, from his Github profile

@davearonson

. . . Markus Schirp, who created mutant, the main mutation testing tool I've actually used, for Ruby. He has also been very willing to answer my ignorant questions and critique the original version of this presentation.

And now, . . .

. . . it's almost your turn! If you have any questions, I'll take them in just a moment. If you think of anything later, I'll be around for the rest of the conference. If it's too late by then, there's my contact information up there, plus the URL where you can get the slides, complete with script *and* some bonus material. And of course I have cards. Any questions?

BONUS MATERIAL

Codosaur.us

@davearonson

```
$ mix test
..........................................
..........................................
..........................................
..........................................
..........................................
..........................................
..........................................
280 tests, 420 assertions,
0 failures, 0 errors, 0 excluded
```

Codosaur.us @davearonson

this sounds like mutation testing only makes sure that our . . .

. . . test *suite* as a *whole* is strict. Is there any way it can help us assess the quality of . . .

. . . *individual* tests?

Yes there is, but it would take a lot longer, and I don't think any current tools give us all the necessary information. If we could run all the applicable tests against each mutant, that would take a lot longer (with our earlier assumptions, ten times as long), but it would give us . . .

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

Codosaur.us

@davearonson

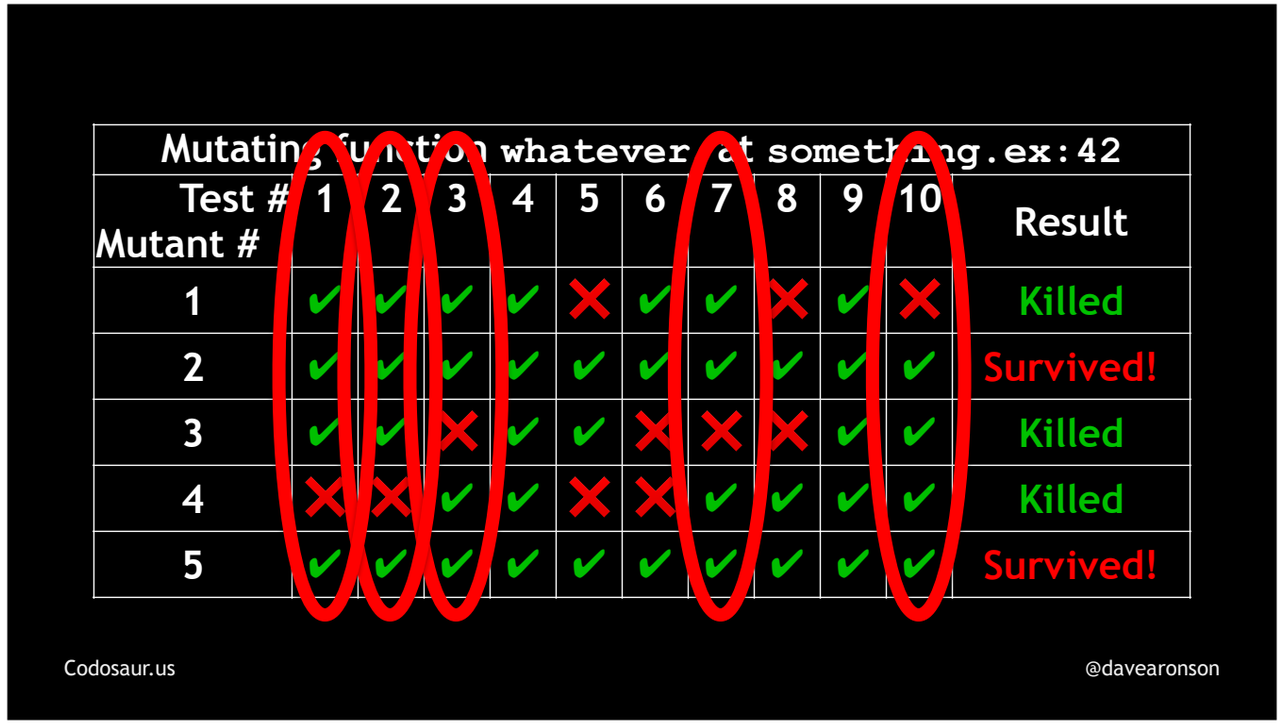
. . . some useful information, that we can use to assess the quality of *some* individual tests. Look at . . .

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

Codosaur.us

@davearonson

. . . tests four and nine. They don’t kill *any* of these mutants! This isn't an absolute indication that they're no good, but it does mean that they may merit a closer look, somewhat like a code smell. We could look *next* at . . .



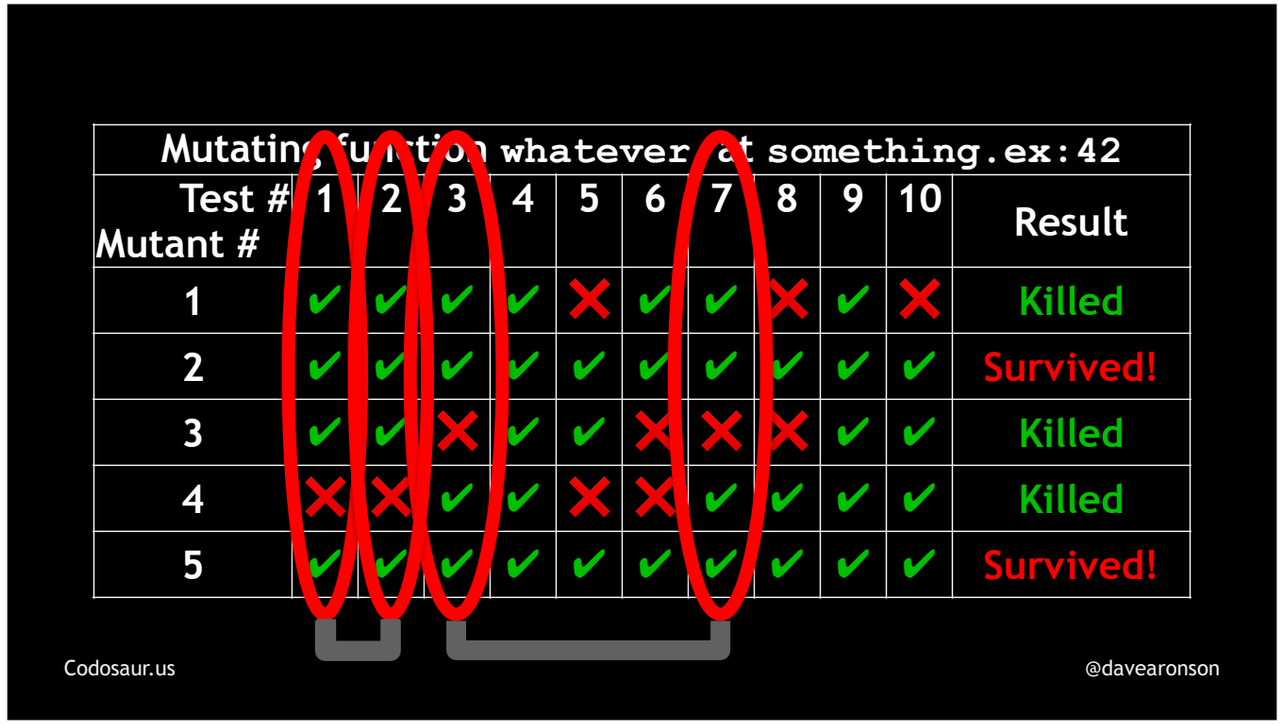
. . . those that only stop *one* mutant, then those that . . .

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

Codosaur.us @davearonson

. . . only stop *two* mutants, and so on, but I think it would rapidly reach a point of diminishing returns, probably at one.

We could also use such a full report to improve our test *suite* again, by looking at *pairs* or larger *sets* of tests, that kill exactly the same sets of mutants, such as . . .



. . . tests one and two, or tests three and seven. We can take a closer look at each set, and decide if we need to keep all the tests. In this case, maybe tests one and two are testing different important aspects, but three and seven are essentially testing the same thing, so we can get rid of one of those, not due to low mutant-stopping power of one test, but due to redundancy between them.

```
$ mix test
.....................................
.....................................
.....................................
.....................................
.....................................
.....................................
.....................................
280 tests, 420 assertions,
0 failures, 0 errors, 0 excluded
```

Codosaur.us @davearonson

as mentioned earlier, mutation testing *assumes* that the code . . .

. . . passes its tests. What if . . .

```
$ mix test
0 tests, 0 assertions,
0 failures, 0 errors, 0 excluded
```

Codosaur.us @davearonson

. . . we don’t have any yet? Can mutation testing be of any help in *that* case?

Well, first of all, whoever wrote a substantial production codebase with no tests needs some educating about the value of tests. Maybe with a two-by-four. But yes, mutation testing can help us . . .

Codosaur.us

Image: https://www.pxfuel.com/en/free-photo-qzzxl

@davearonson

. . . *build* our test suite in the first place! We can start with a . . .

```
test "nothing" do
  assert true
end
```

Codosaur.us @davearonson

. . . meaningless test, and run our mutation testing tool. We'll probably get a . . .

Codosaur.us

Image: https://commons.wikimedia.org/wiki/File:The_mutants!_(9363719694).jpg

@davearonson

. . . lot of mutants, including many that are essentially duplicates. Out of the mutants for each function, . . .

Codosaur.us

Image: https://pixabay.com/photos/mutant-daisy-flower-bloom-macro-593712/

@davearonson

. . . pick one at random. Try to kill it, by adding a test. This will probably kill many others as well. Then lather, rinse, repeat, though on later iterations maybe *adjust* a test rather than *adding* one. Now, this won't . . .

. . . *guarantee* that we wind up with a great test suite, but it will get us off to a decent start. Then we can look at what code is untested, and write more tests to fill in the holes.