Kill All Mutants!

(Intro to Mutation Testing)

by Dave Aronson

T.Rex-2022@Codosaur.us

Codosaur.us

@davearonson

CURRENT TOTAL TIME: ~50

NOTES TO SELF:
- add example like codemanship's one about needing to handle invalid instruction

Kill All Mutants!

(Intro to Mutation Testing)

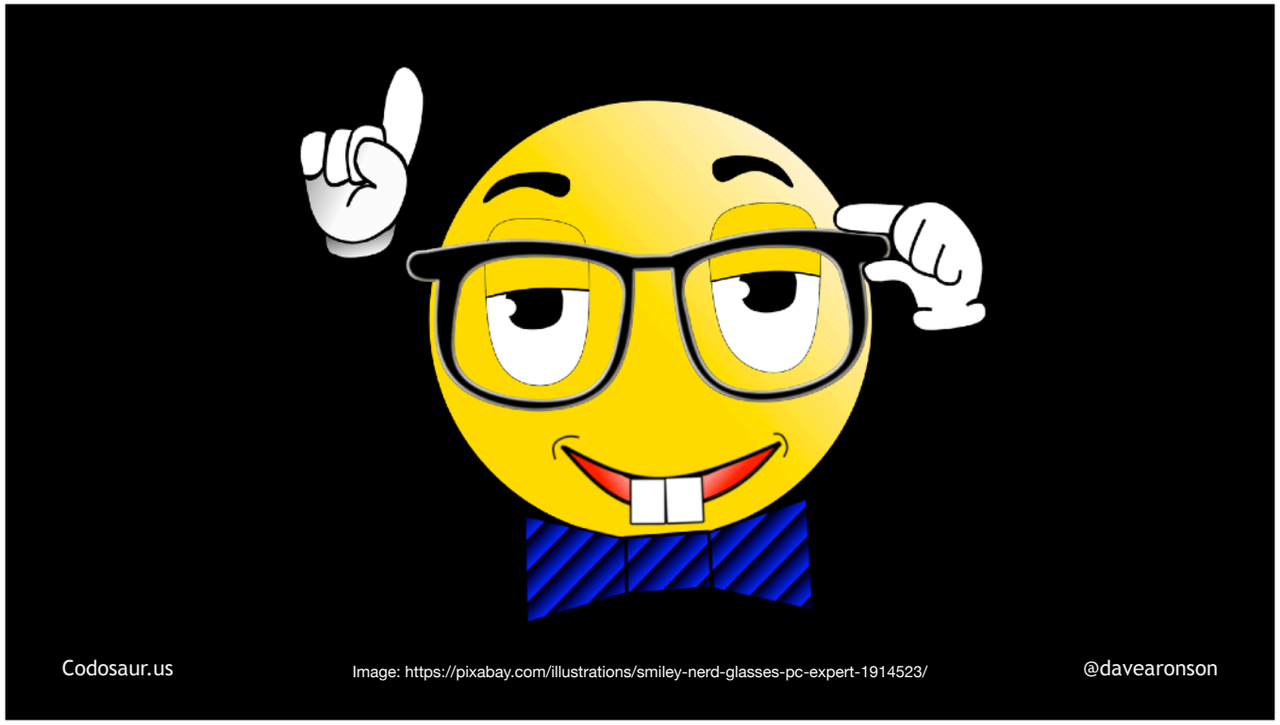
by Dave Aronson

T.Rex-2022@Codosaur.us

Codosaur.us

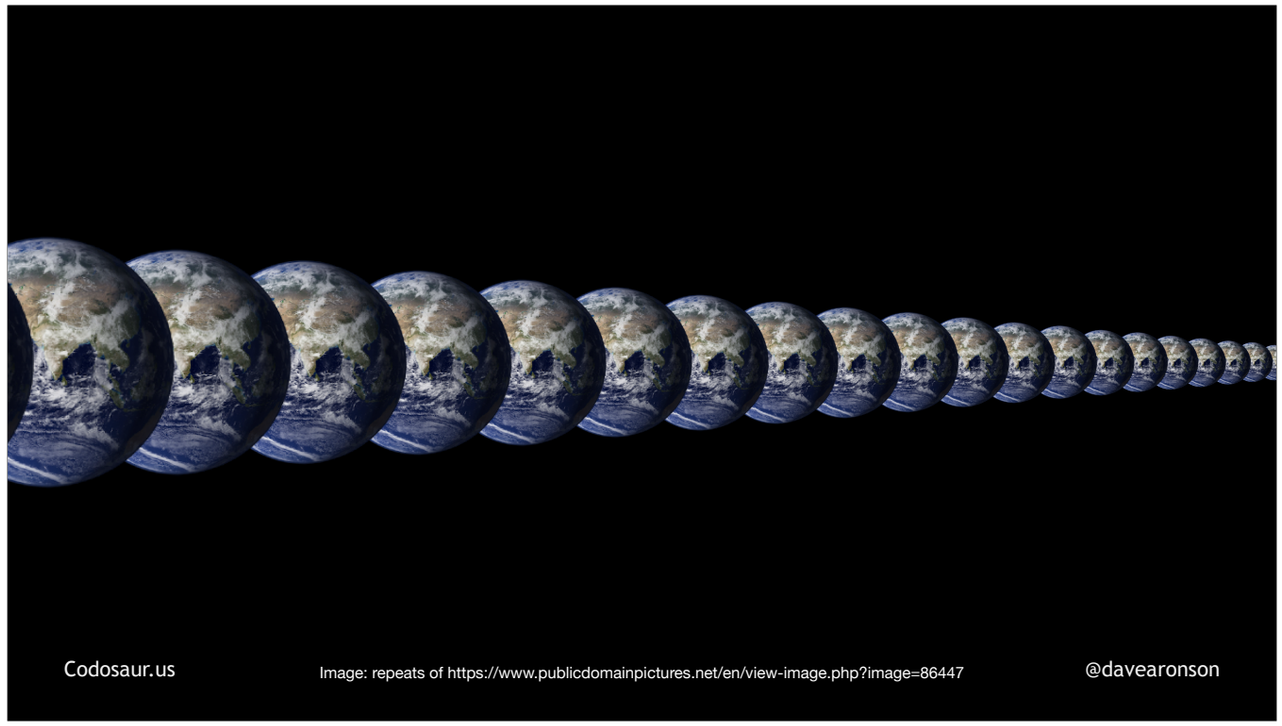
@davearonson

Good morning, Poconos! I'm Dave Aronson, the T. Rex of Codosaurus, and I flew here up here on my pet pterodactyl to teach you to *kill mutants!* (PAUSE!)
But first, a small disclaimer: I do not consider myself an . . .

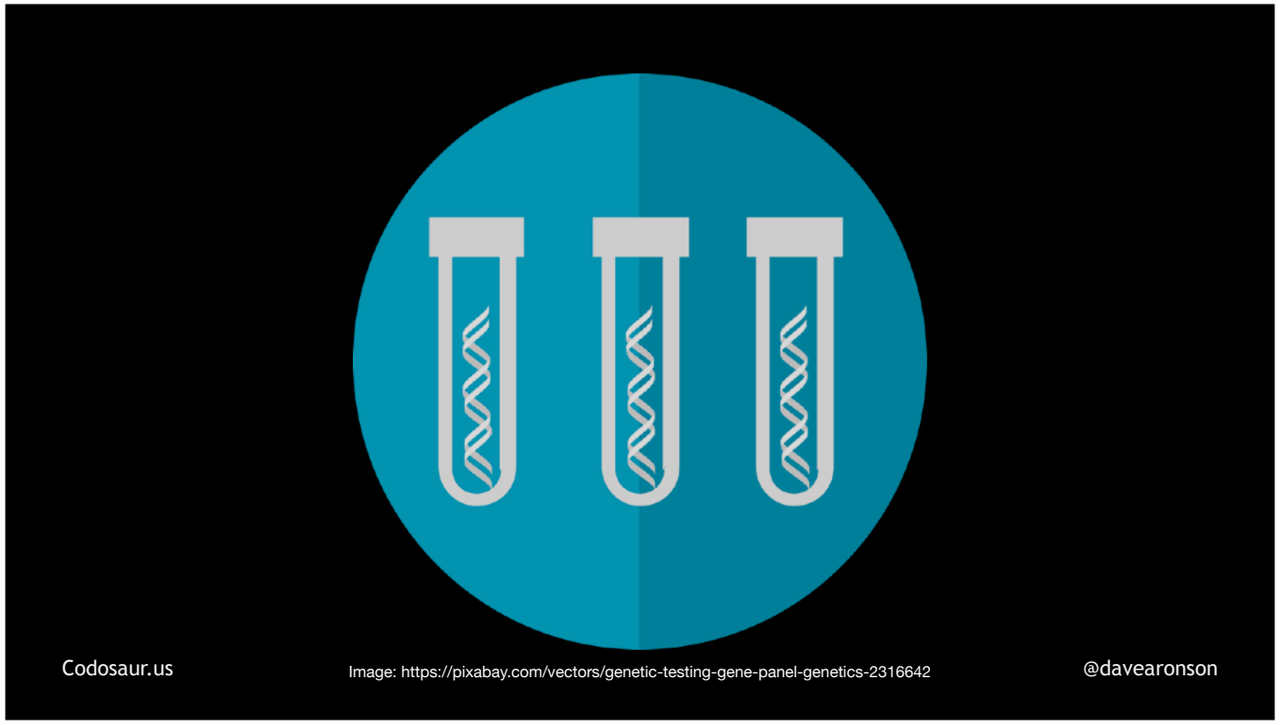


. . . expert on mutation testing. But, one of the dirty little secrets of public speaking is that you don't *have* to be an expert on your topic! You just have to know a *little bit more* about it than the audience does, enough to make it worth their time to listen to you, and be able to *convey* it to them. And mutation testing is still rare enough, that most developers have never even *heard* of it!

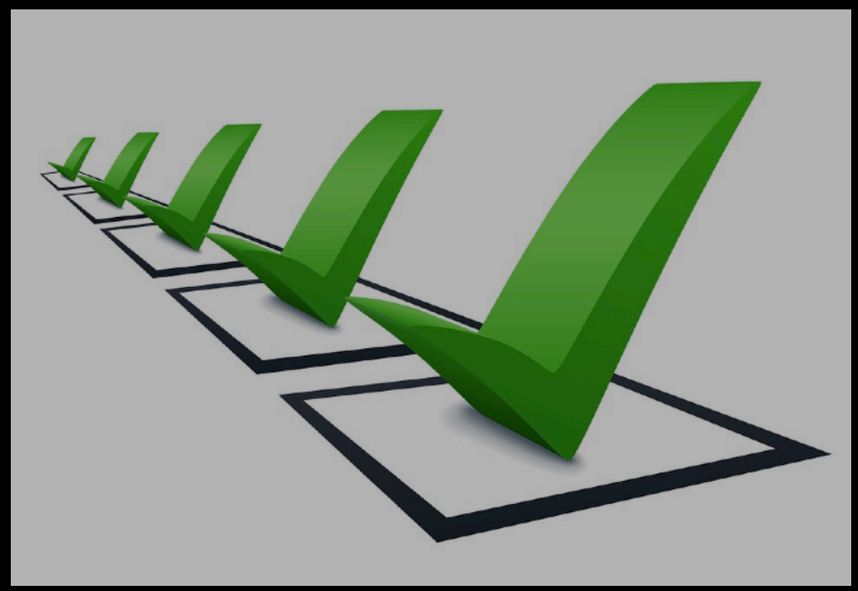
So let's start with the basics. What on . . .



. . . 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* night, er, I mean, technique different from all *other* techniques? The main difference is that most of the others are about . . .

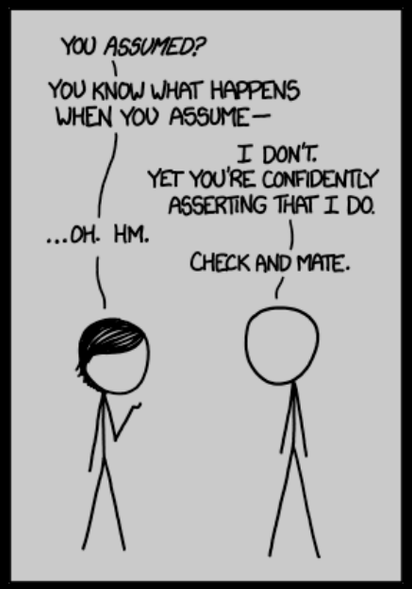


Codosaur.us

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

@davearonson

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

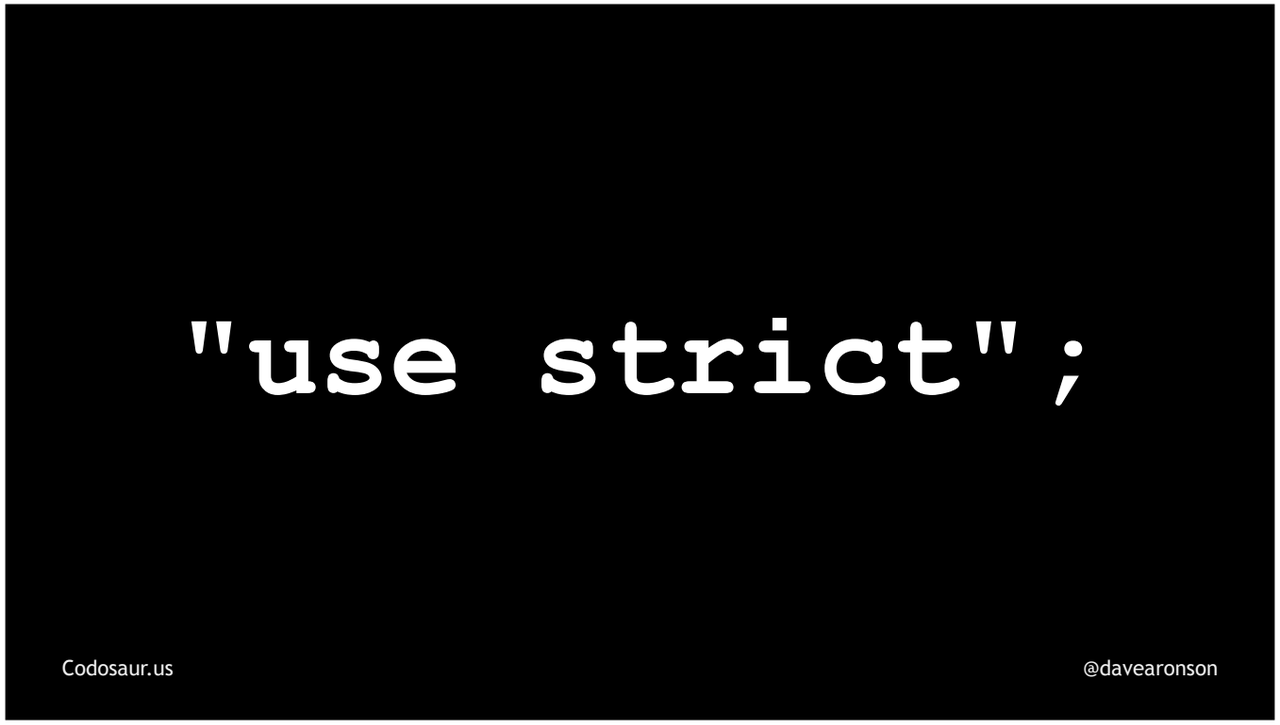


Codosaur.us

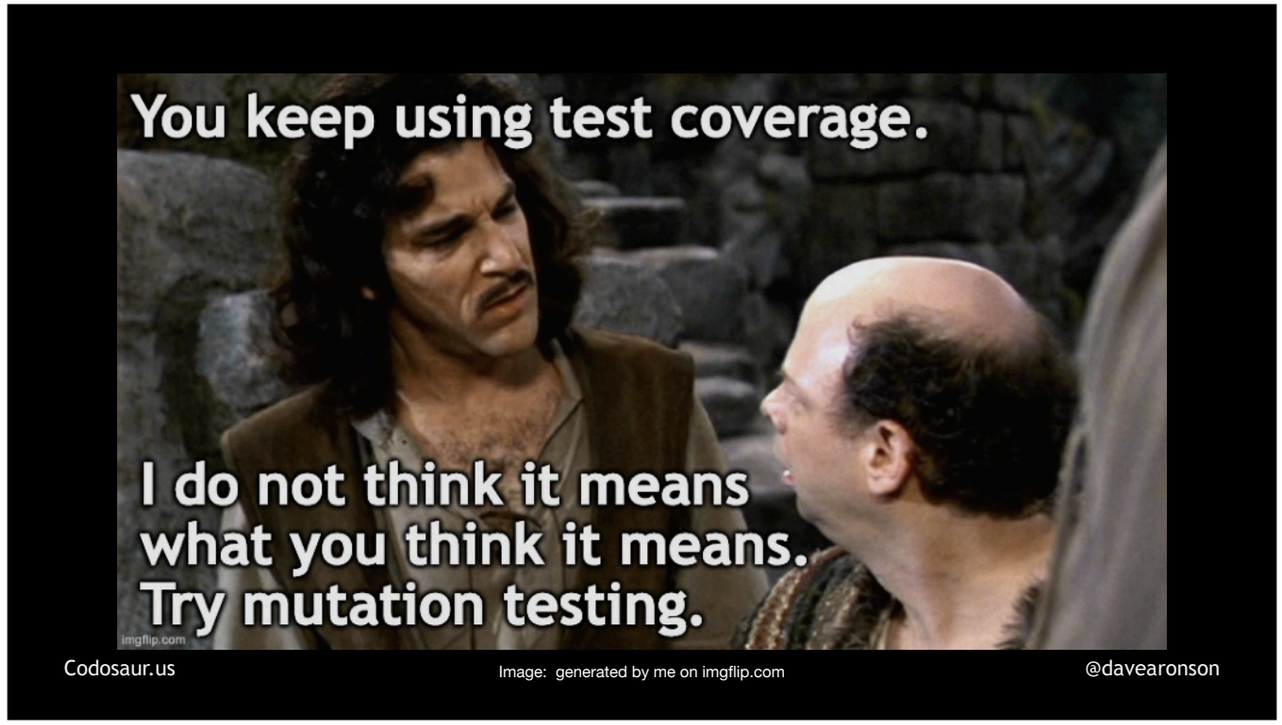
Image: https://xkcd.com/1339/

@davearonson

. . . *assumes* that our code is correct, at least in the sense of passing its tests. (And that means it assumes we *have* tests. More on that later.) Instead, mutation testing checks for not one but *two* other qualities. In a typical codebase, I think the more *important, interesting,* and *immediately useful* of these two qualities is that our test suite is . . .



. . . *strict*. Now you may be thinking, “But Dave, isn’t that what test coverage is for? If we have 100% coverage, doesn’t that mean our code is fully tested?” Long story short . . .



. . . no. (PAUSE!) The *only* thing that test coverage tells us is that at least one test *executed* . . .

```
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

. . . the code it claims is “covered”, and *no* tests ran the rest of the code. It tells us NOTHING about whether the *correctness* of any particular piece of code *made any difference* to whether any test, let alone any *particular* test, passed or failed.

By way of illustration, 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. Assertions might be commented out, removed, or not even written in the first place, for any number of reasons — usually not very *good* reasons, but it often happens anyway. Let's have a show of hands, who's seen that happen? I'm not asking if you've done it yourself, so no shame, just who's *seen* "tests" with no assertions.

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 some test passes? As you may have guess, that . . . is where mutation testing comes in.

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

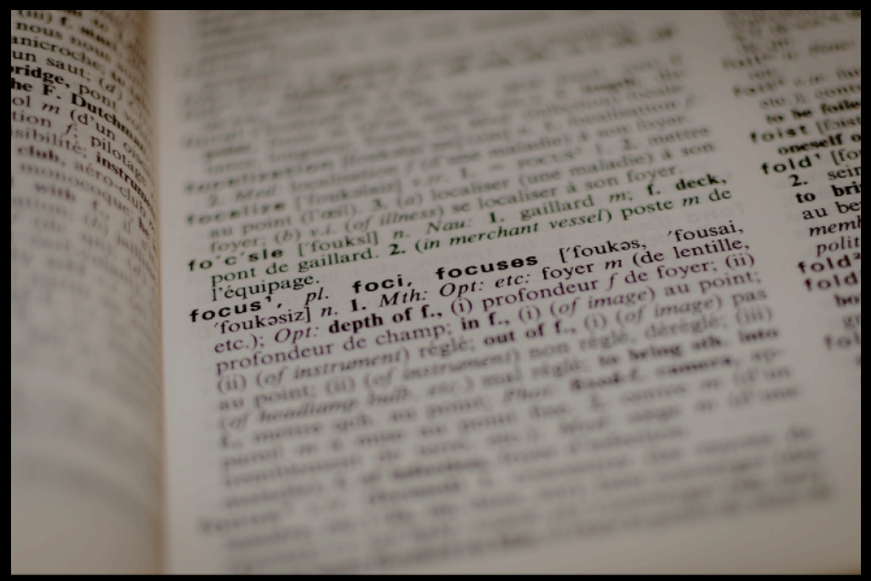
Codosaur.us

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 . . .



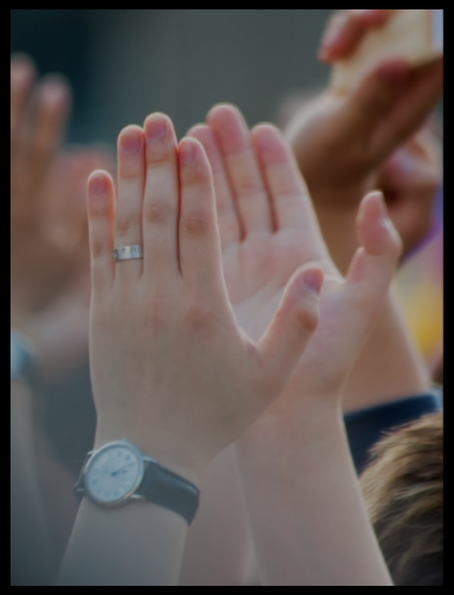
Codosaur.us

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

@davearonson

. . . *meaningful*, by which I mean that any tiny little semantic change to the code (versus just structural or syntactic changes), 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, at least that we care about. Once we find "meaningless" code, we can figure out *why* it's meaningless, then make it meaningful, if that fits our intent, but the usual fix is just to remove it.

Mutation testing . . .



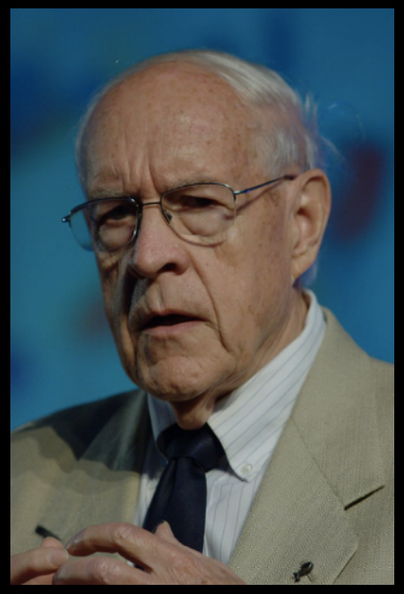
Codosaur.us

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

@davearonson

. . . puts these two together, by checking that every single small semantic change to the code, that the tool knows how to do, 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, author of "The Mythical Man Month" (1975 book) and "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 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!

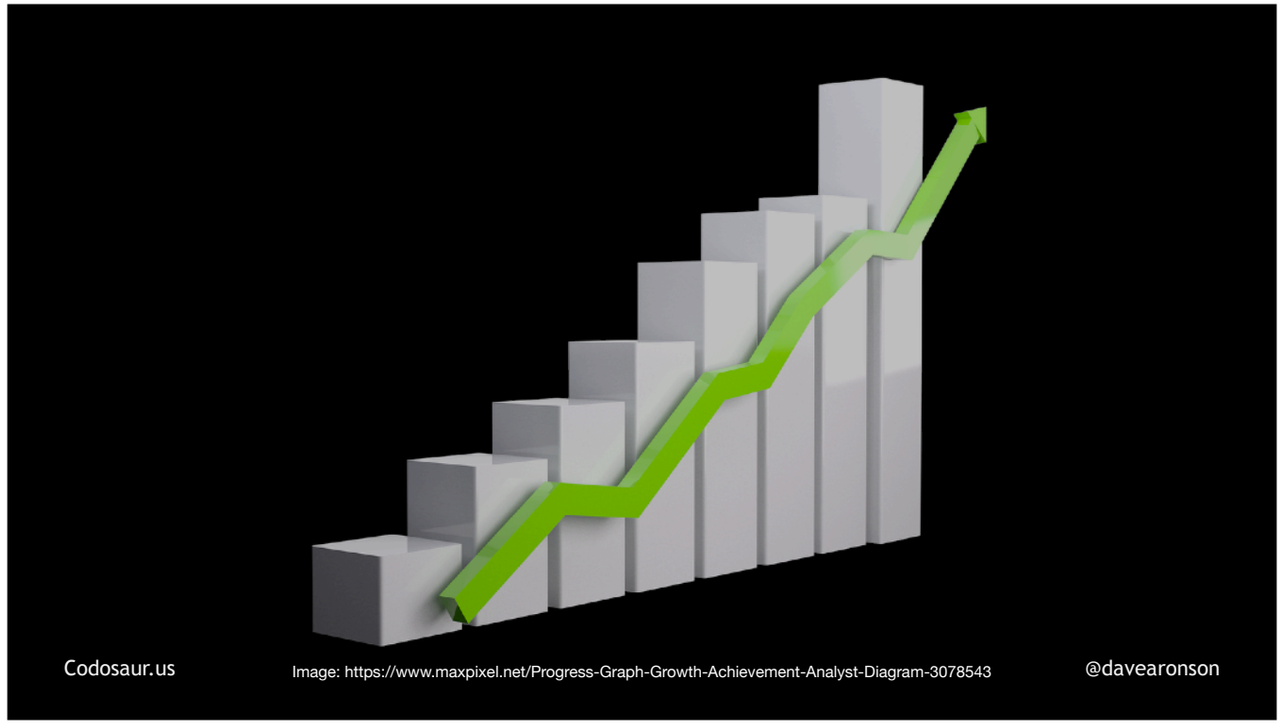
The first drawback is that it's rather . . .

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, modules, files, and so on. Also, they usually include some kind of . . .

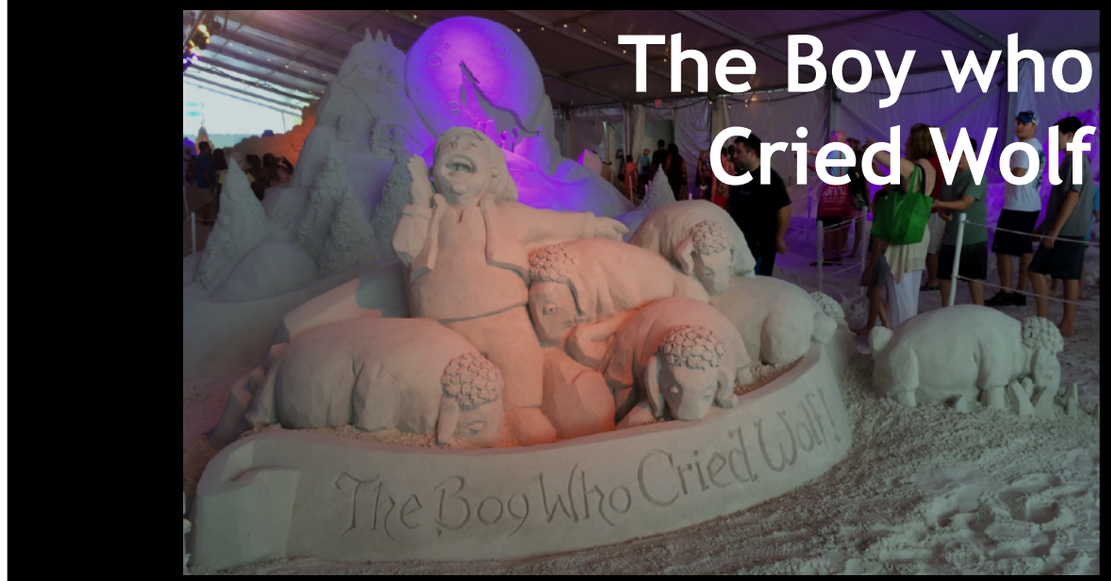


. . . incremental mode, so that we can test only the changes since the last mutation test, or the last git commit, or the main branch, or some such difference. With such filtering, maybe we can test the changes on each save, if we save often, use short-lived branches, and so on, like we know we're supposed to do but don't always.

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. Their accent is verrah strayinge, 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 . . .

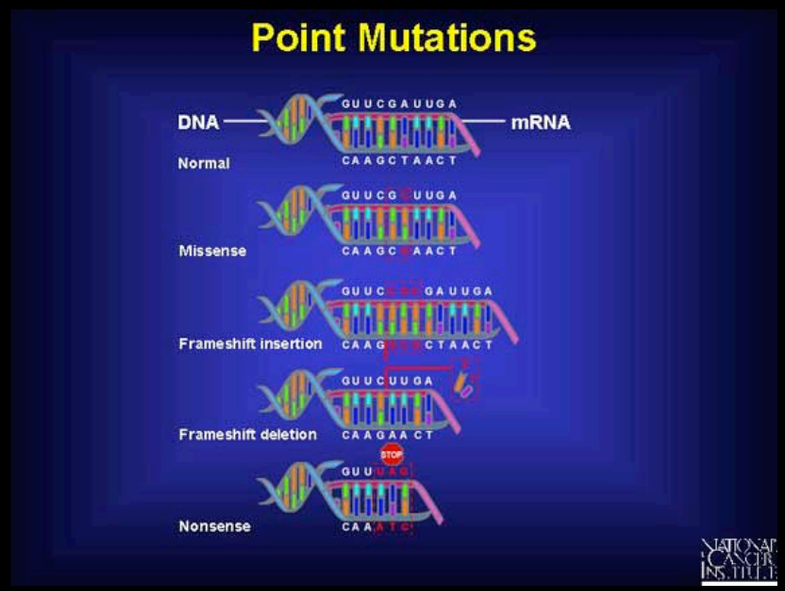


. . . false alarm, because the mutation didn't make a test fail, but it didn't make any behavioral difference in the first place. It can still take quite a lot of time and effort to figure *that* out.

And even if a mutation *does* make a difference, most programs have quite a lot of code that we just . . .

. . . *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 tell them "don't bother mutating this line", or even this whole function, module, file, or whatever . . . 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 . . .

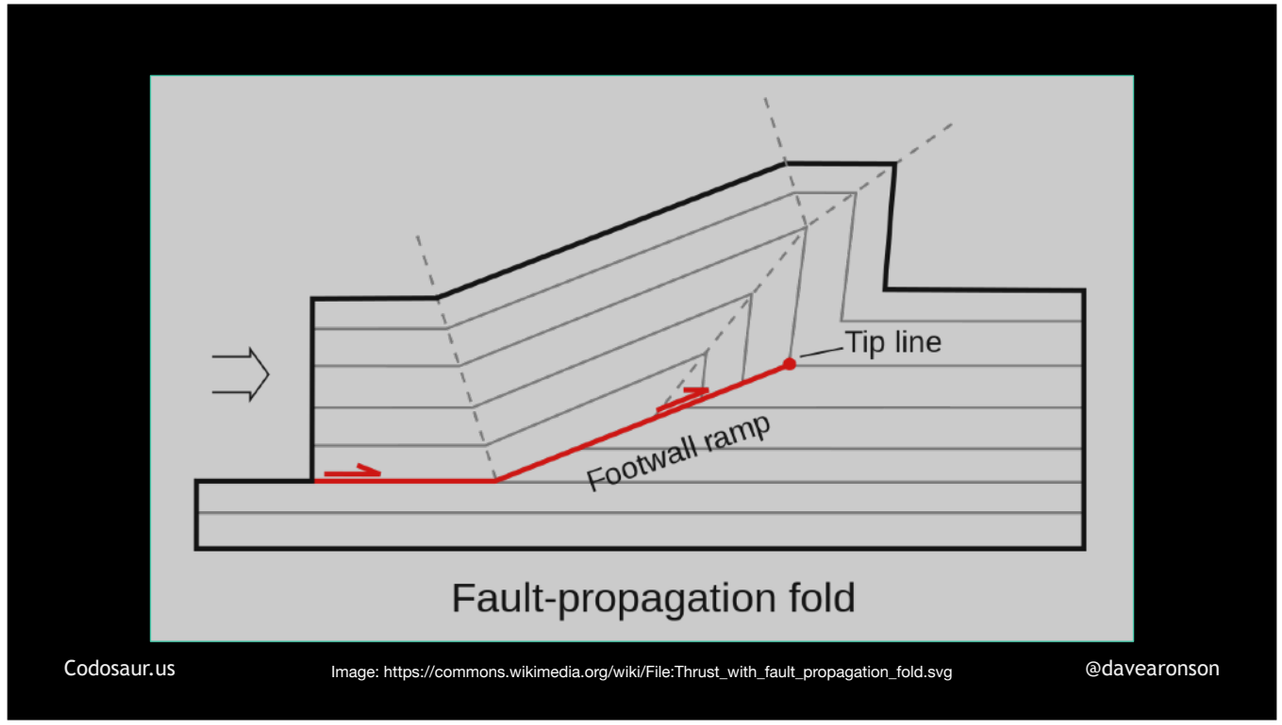


Codosaur.us

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

@davearonson

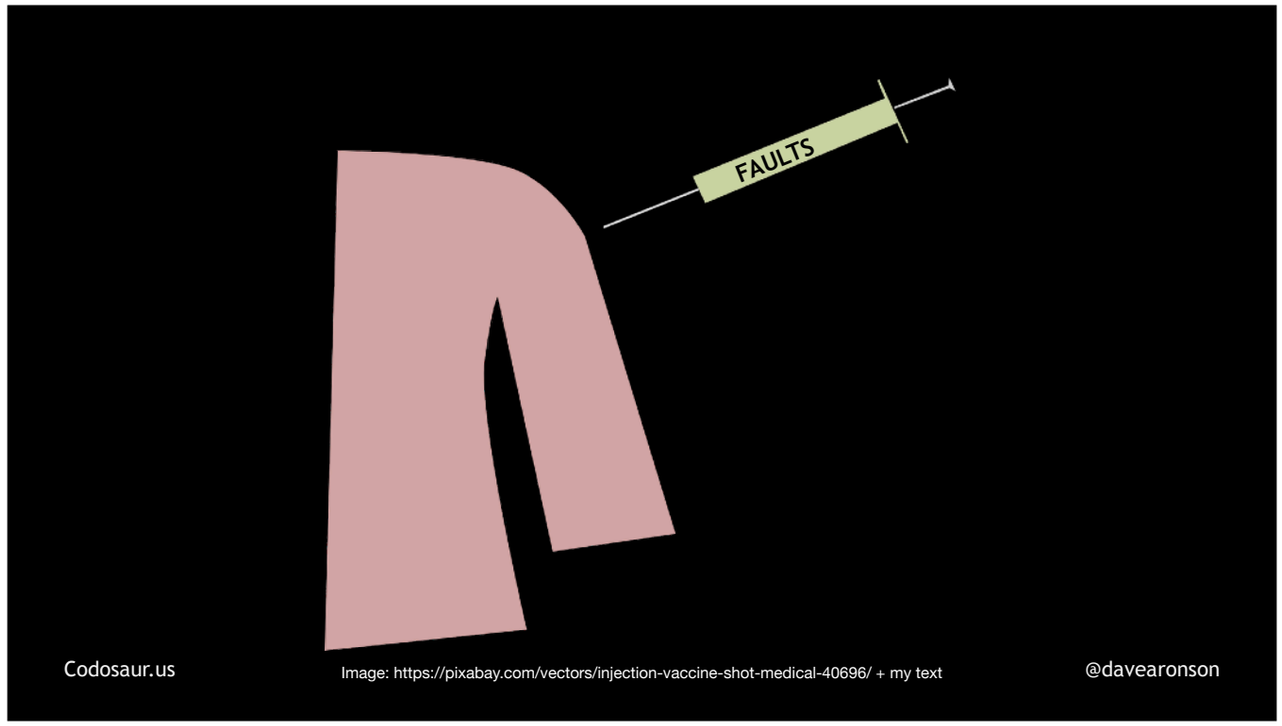
. . . *mutates* copies of our code, hence the name. It does this with the intent to create test failures, also known as . . .



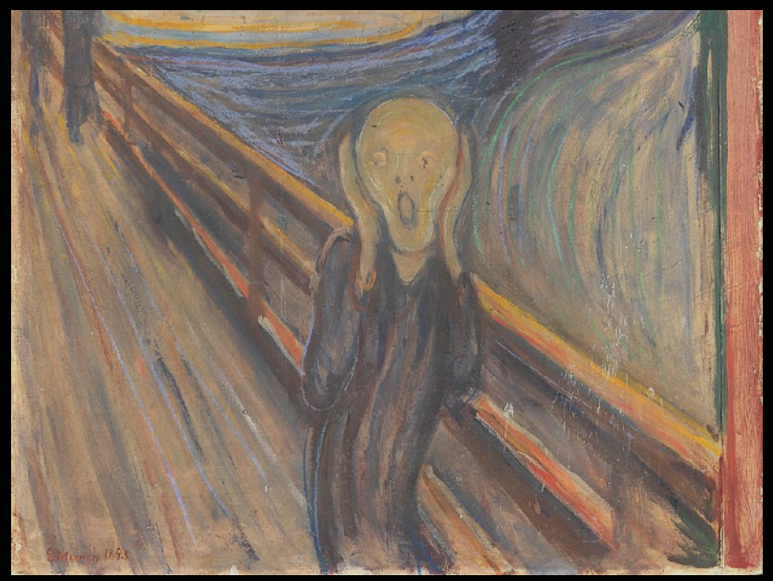
. . . faults. So, mutation testing can be categorized as a *fault-based* testing technique. This means it is related to something you might already be familiar with:

. . . Chaos Monkey, from Netflix. Just like Chaos Monkey uses faults to help Netflix discover flaws in their error recovery, mutation testing uses faults to help us discover certain 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 latency, jitter, and dropped connections, into Netflix's production network.



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

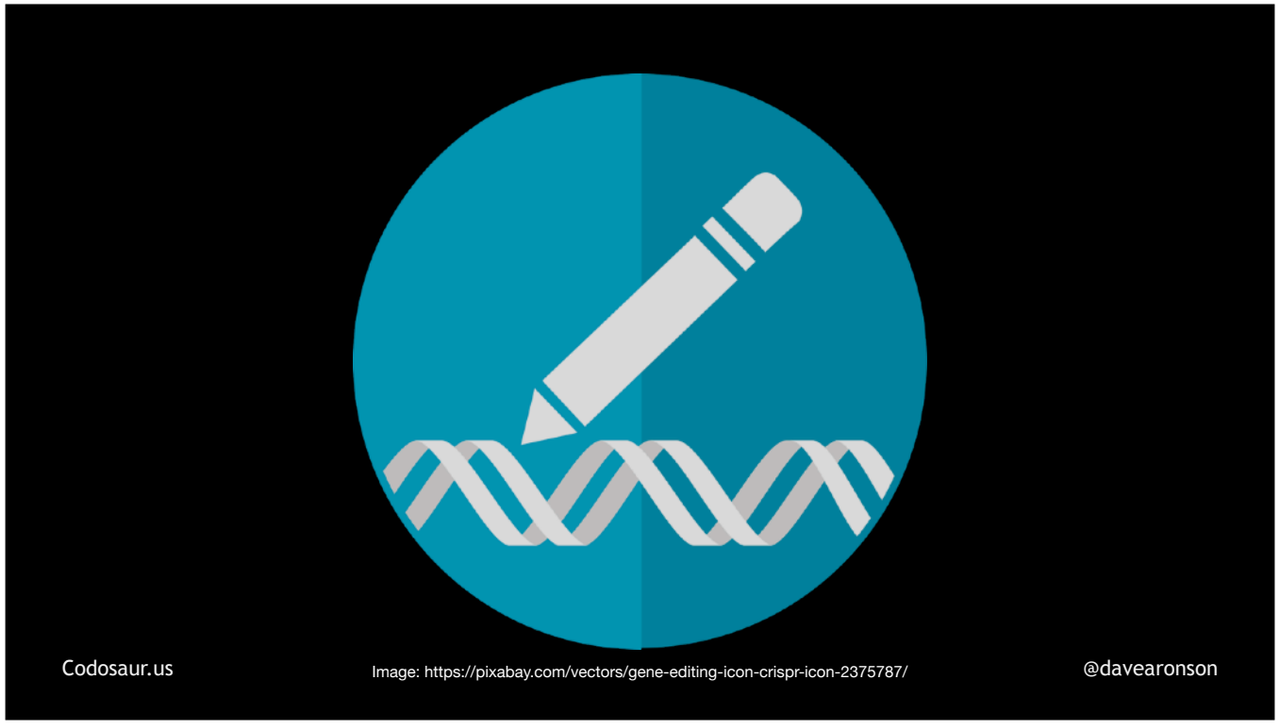
If all still goes well, in the sense that Netflix's customers don't notice, and their metrics still look good, then Netflix knows that their error recovery is working fine. Mutation testing, however, injects semantic . . .

Codosaur.us

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

@davearonson

. . . *changes*, not necessarily *problems*. It doesn't *know* whether these 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* . . .

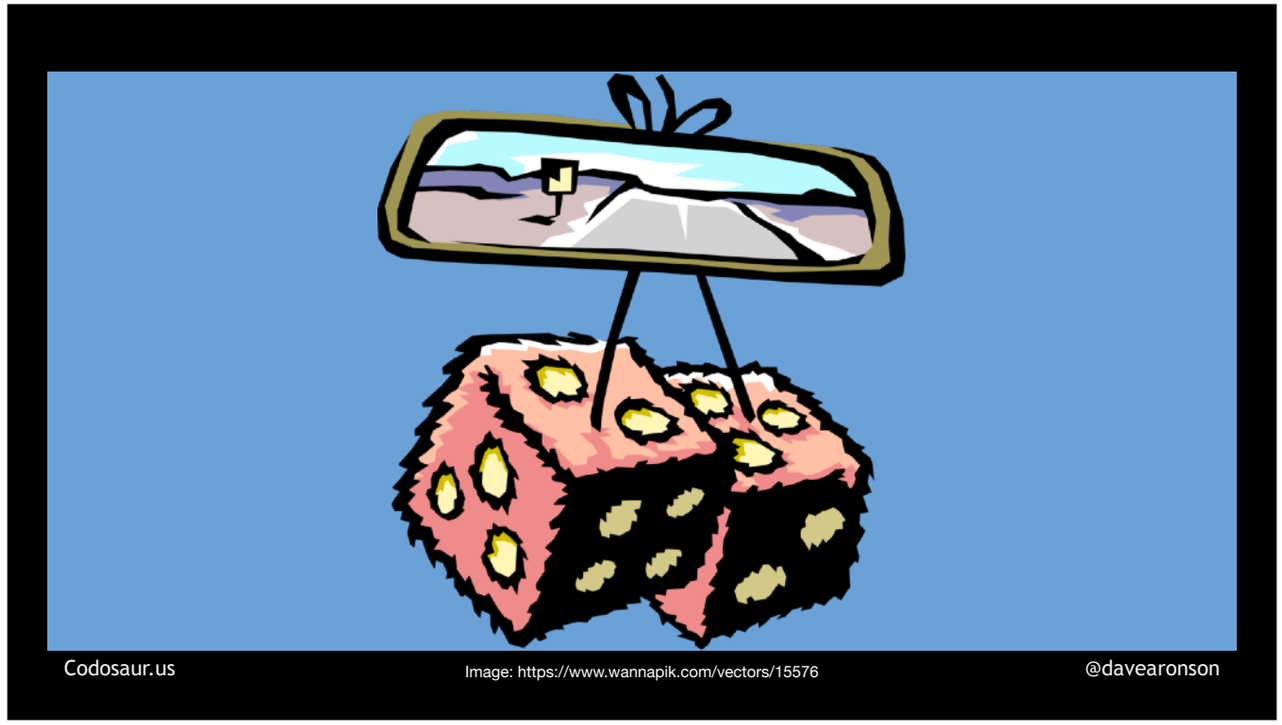
```
$ mutation_test
........................................
........................................
........................................
........................................
........................................
........................................
........................................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

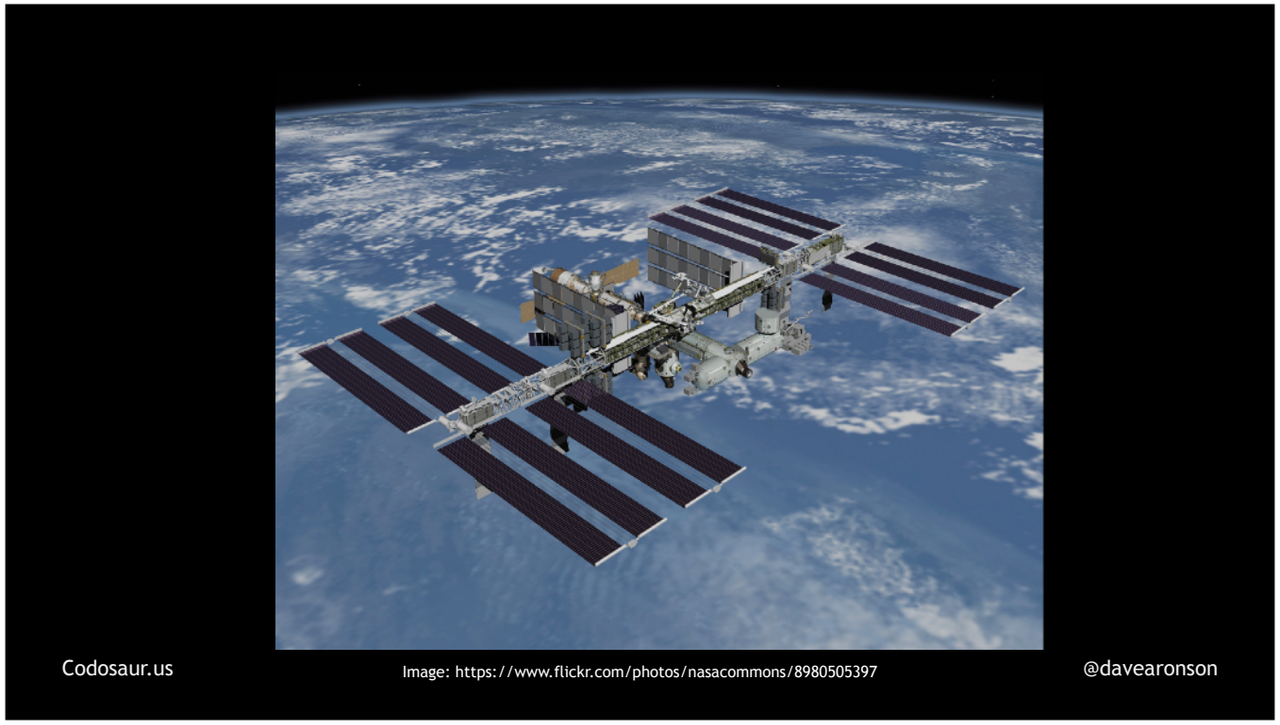
Mutation testing has also been compared to . . .



. . . 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. These tools have a set of mutations they know how to do. The smarter ones can use the results of simpler mutations, to know they don't need to bother with more complex ones, so it may sometimes do different things in different cases and therefore *look* random, but it's not.

But enough about differences. What exactly does mutation testing *do*, and how? Let's start with . . .

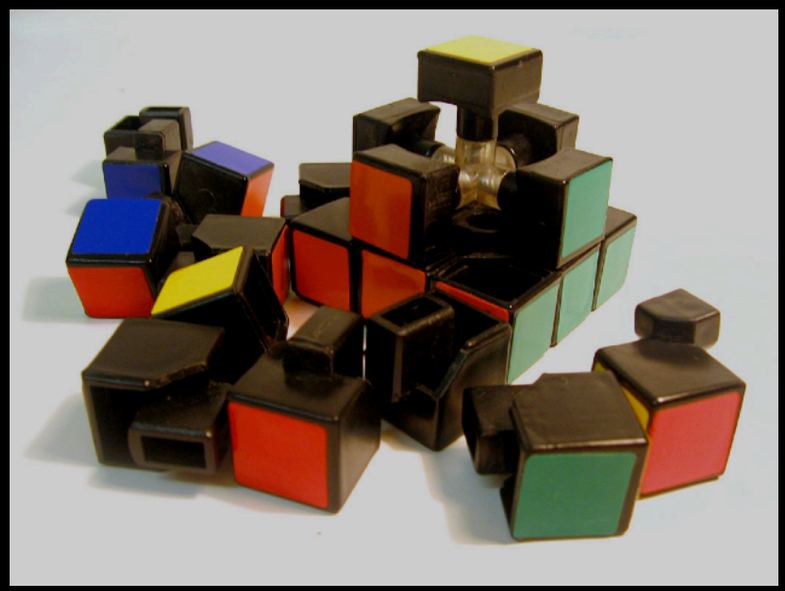


Codosaur.us

Image: https://www.flickr.com/photos/nasacommons/8980505397

@davearonson

. . . a high-level view. First, our chosen tool . . .

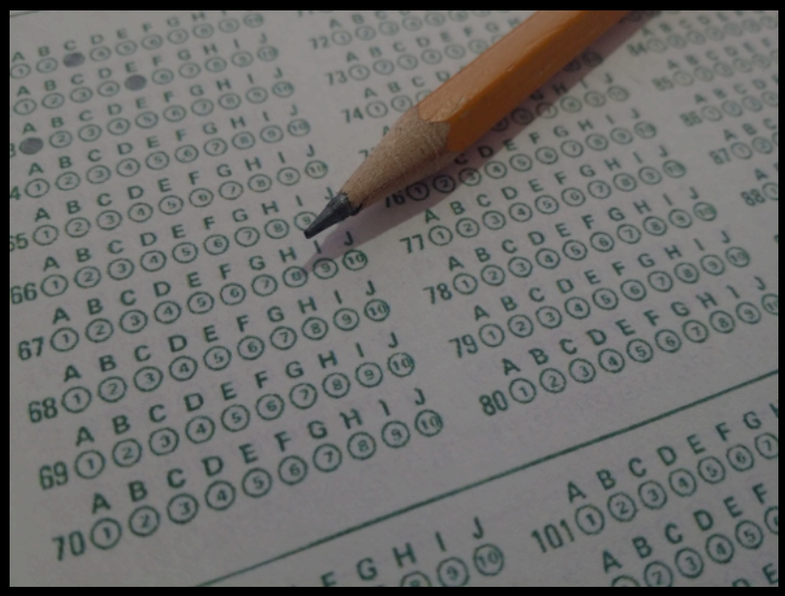


Codosaur.us

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. Better yet, most those of them will warn us, so we know we need to add or maybe annotate some tests. (More on that later.) Some, though, will use the whole test suite, which is horribly inefficient, because the vast majority of the tests are almost certainly not relevant to this particular function.

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*, in other words, that *mutation*.

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 quite all this information, let alone so neatly organized, but it's a conceptual model I use to help 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/

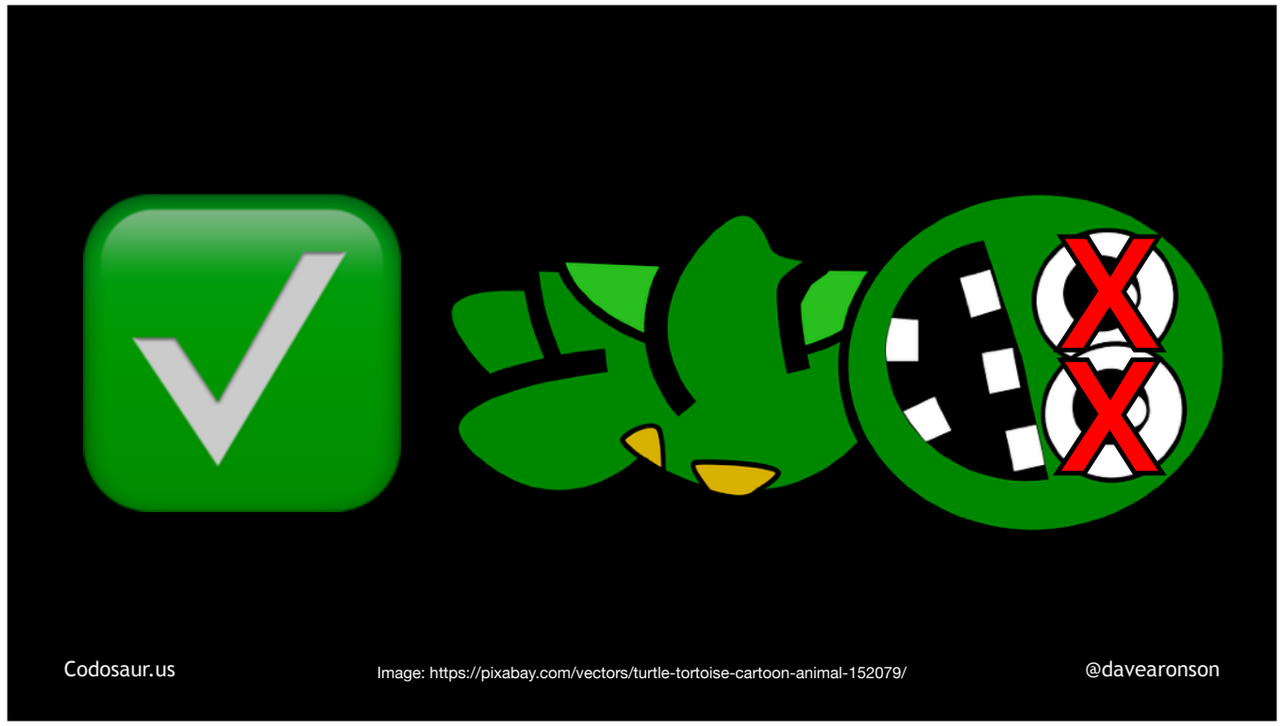
@davearonson

. . . “killing the mutant”. And now for a brief . . .

. . . detour. Some people object to this . . .

. . . “violent communication”, especially since, in the comic books, mutants are often metaphors for marginalized groups of people, and the tech industry is *finally* starting to become more sensitive to such issues. So, I’ve been trying to come up with something nicer, like covering or rescuing the poor little lost mutant. However, the terminology is pretty well established, so it’s hard to change, so for this presentation, I’m going to stick to the industry-standard term of "killing" the mutant.

Anyway, whatever we *call* it, 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, . . .

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

. . . 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

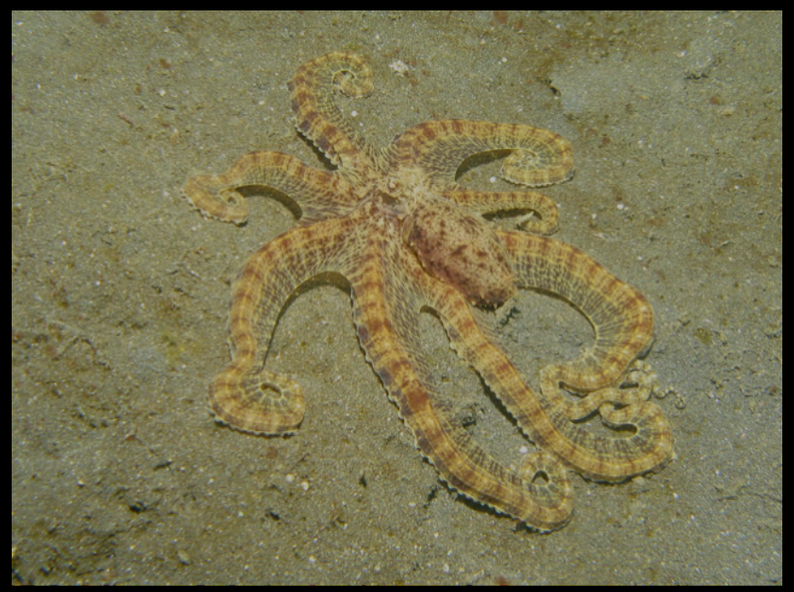
. . . lets all the tests pass, then the mutant is said to have . . .

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/

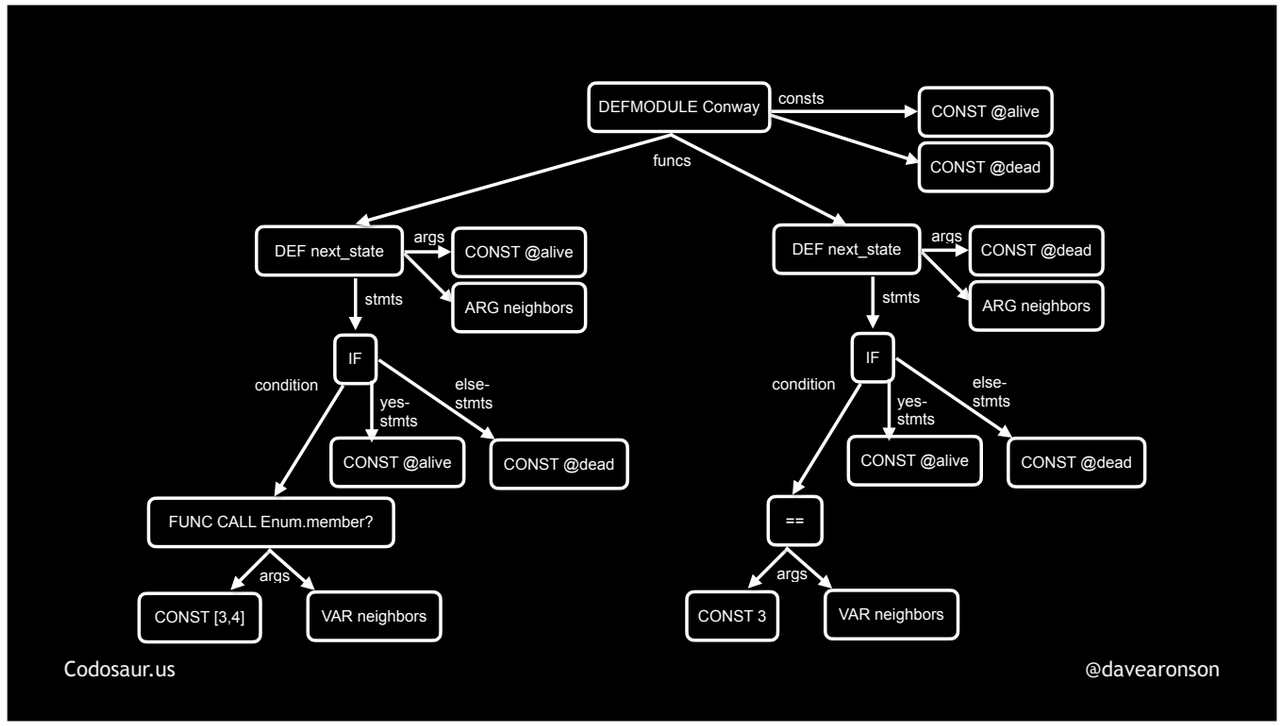
@davearonson

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

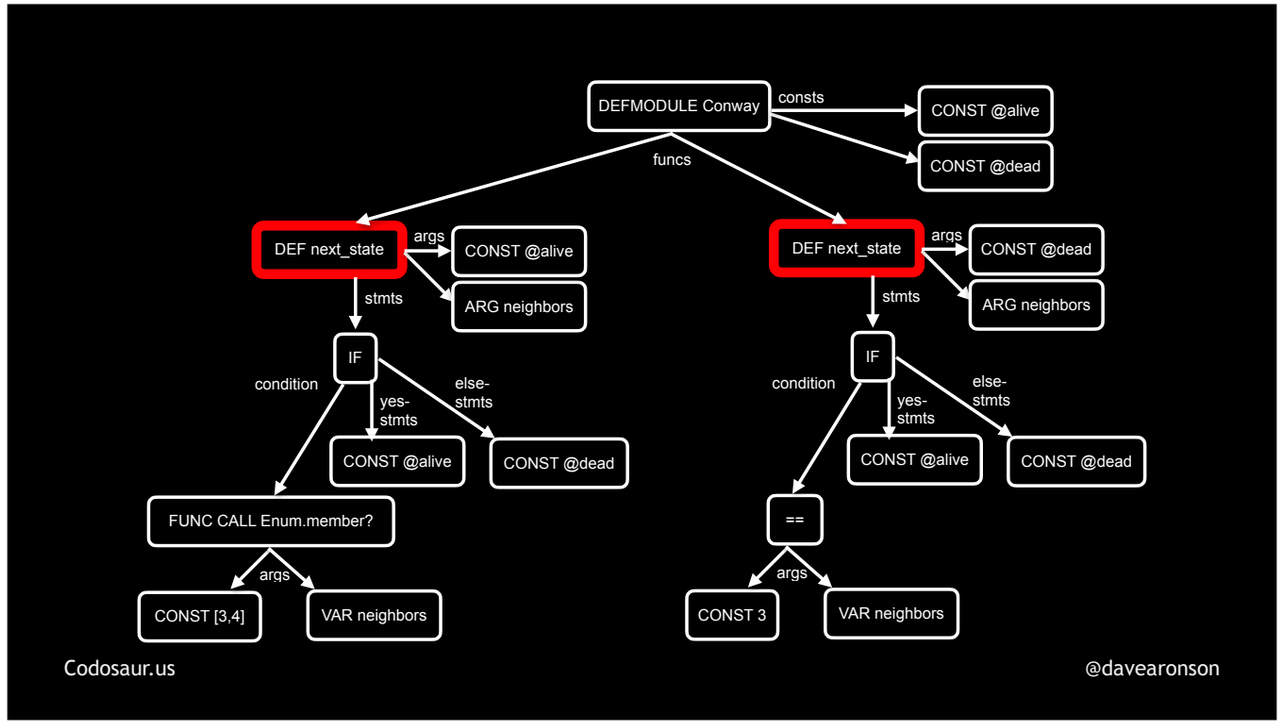
```
defmodule Conway do
  @alive "*"
  @dead  " "

  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

. . . our code, usually into an Abstract Syntax Tree. Some work with bytecode instead, or even raw source code, but most of the serious tools that are “ready for prime time” work with an AST, so let’s roll with that. So, this code becomes . . .



. . . this AST. I'm going to assume that you're all familiar with the concept of an AST, or at least can figure it out from context, but don't worry about understanding this one in detail. Just notice that there are two subtrees in there that represent functions, the ones rooted at . . .



. . . DEF nodes. They have the same name, but that's okay in some languages; this particular chunk of code was in one of those, called Elixir.

Anyway, 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 each one as I described before, starting with looking for each one's *tests* . . . so 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, though that can get tricky and unreliable, especially when . . .

```
test "#foo turns 3 into 6" do
  foo_test_helper(3, 6)
end

test "#foo turns 4 into 10" do
  foo_test_helper(4, 10)
end
```

Codosaur.us

@davearonson

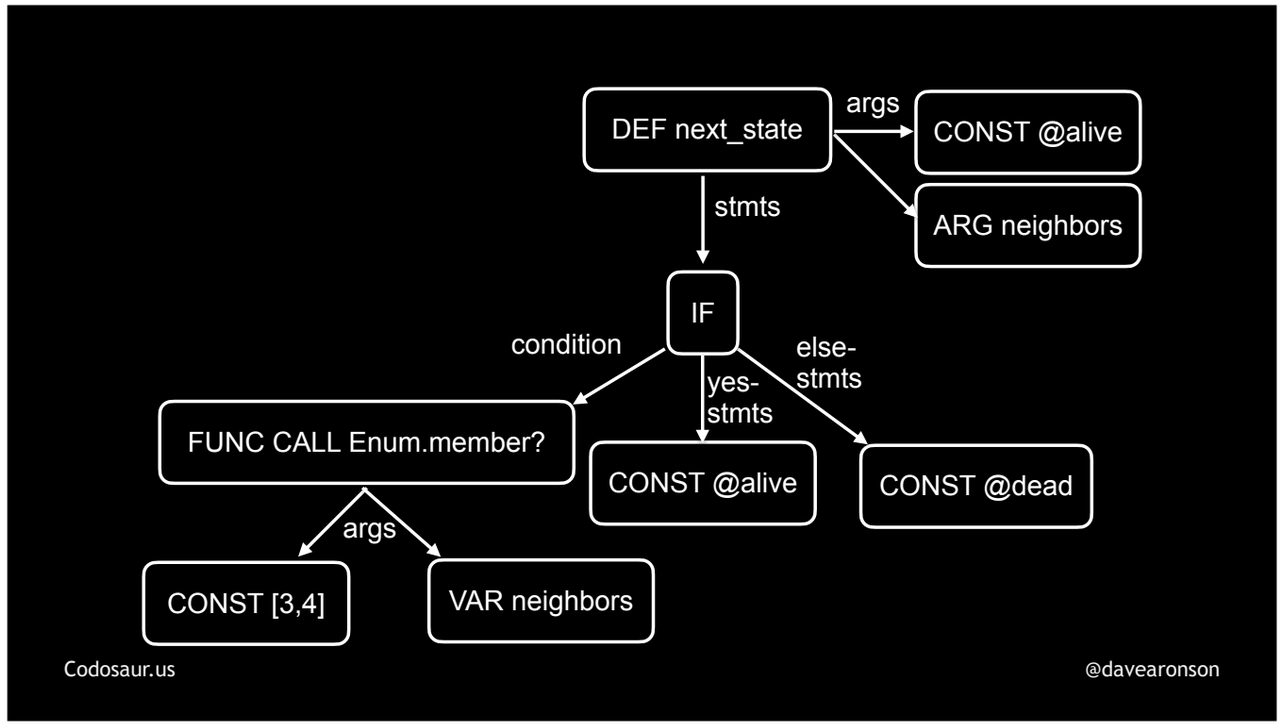
. . . the function isn’t called directly. Anyway, after the tool has found the function's tests, then, assuming it won't skip this function because it *didn't* find any tests, it makes the mutants. To make mutants *from* an AST subtree, it . . .

Codosaur.us

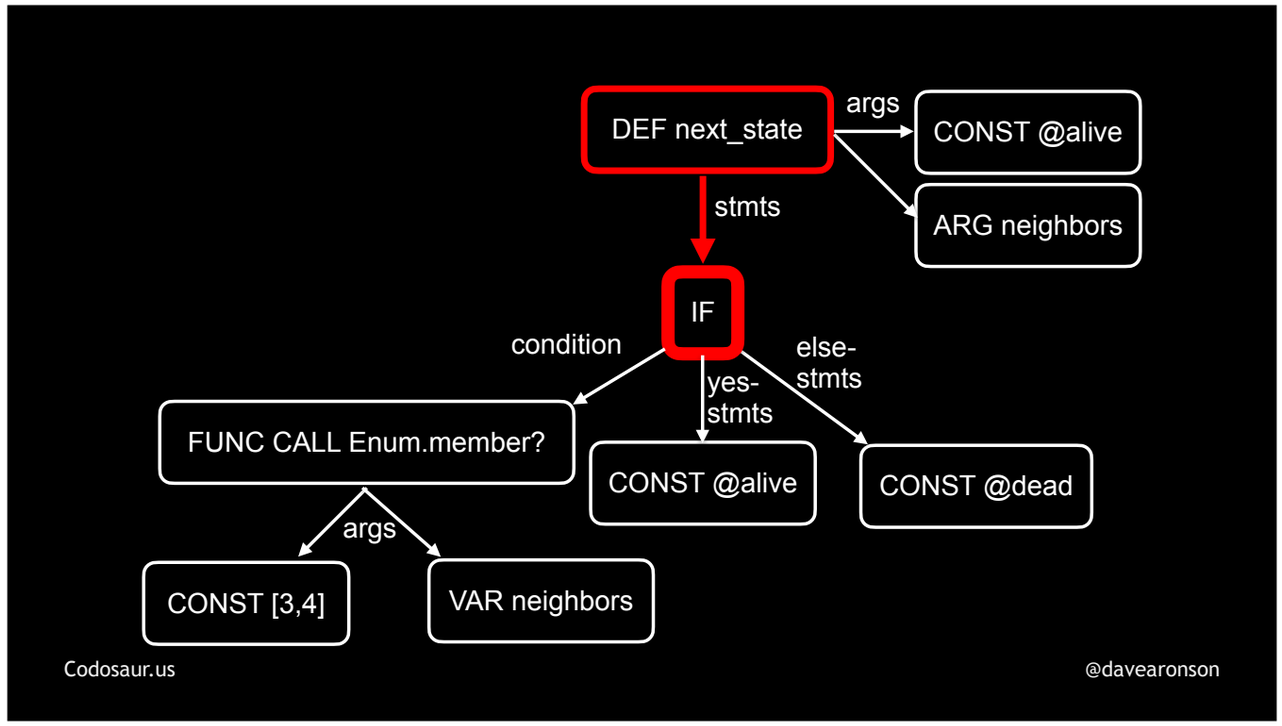
Image: https://pxhere.com/en/photo/1230969

@davearonson

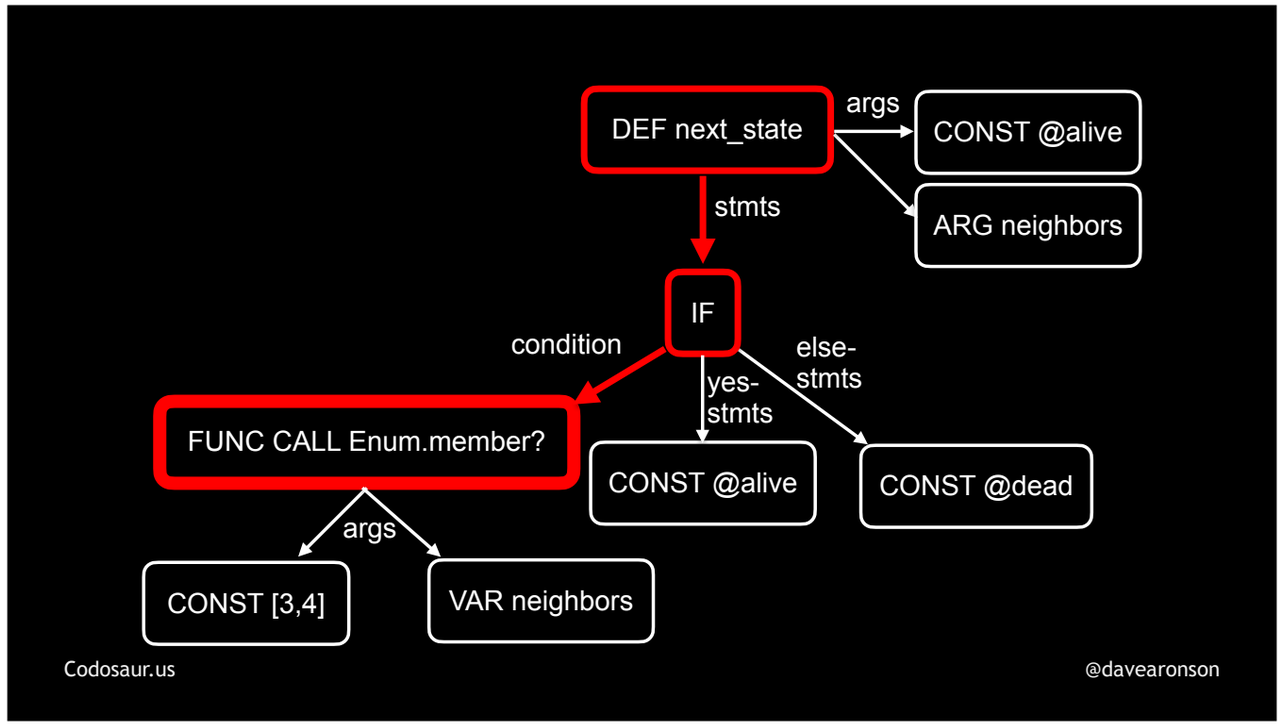
. . . traverses that subtree, just like it did to the whole thing. However, now, instead of looking for even *smaller* subtrees it can *extract*, like twigs or something, it's looking for *nodes* where it can *change* something. Each time it finds one, then for each way it can change that node, it makes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



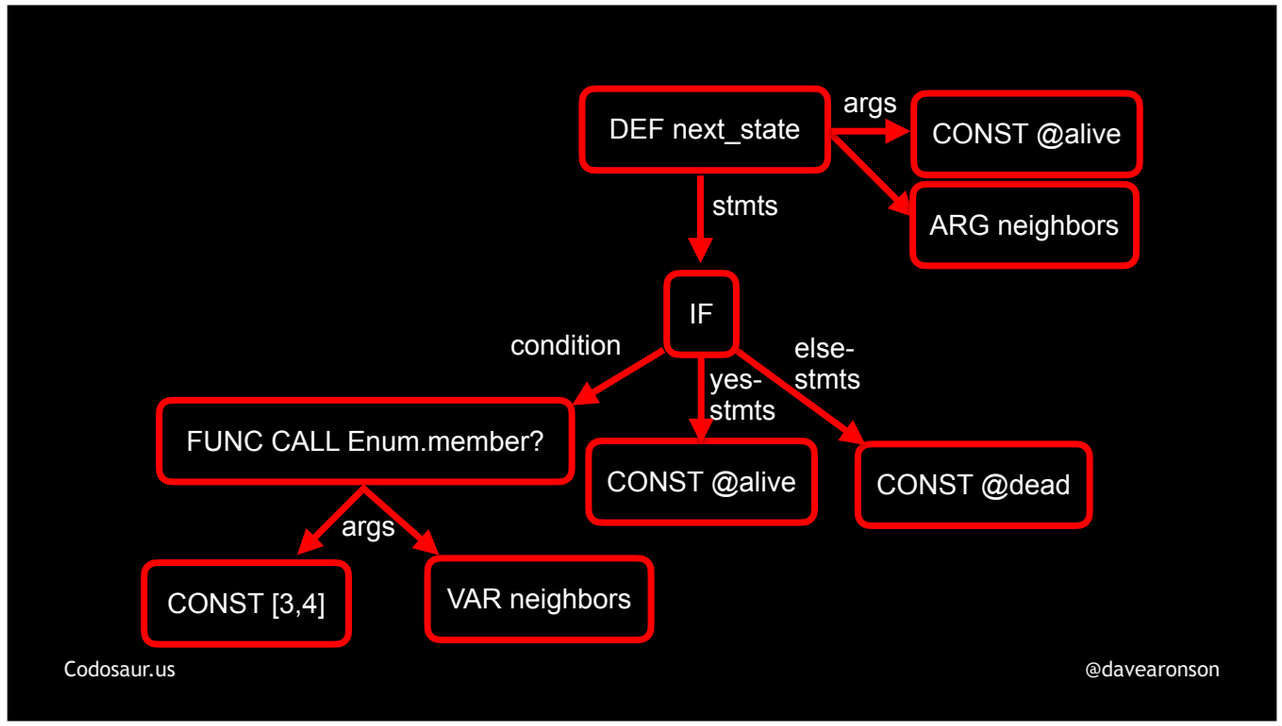
. . . one of the function subtrees from the AST I showed earlier, and has only gotten down to . . .



. . . 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 by mutating *that* node, it would continue traversing the subtree, to . . .

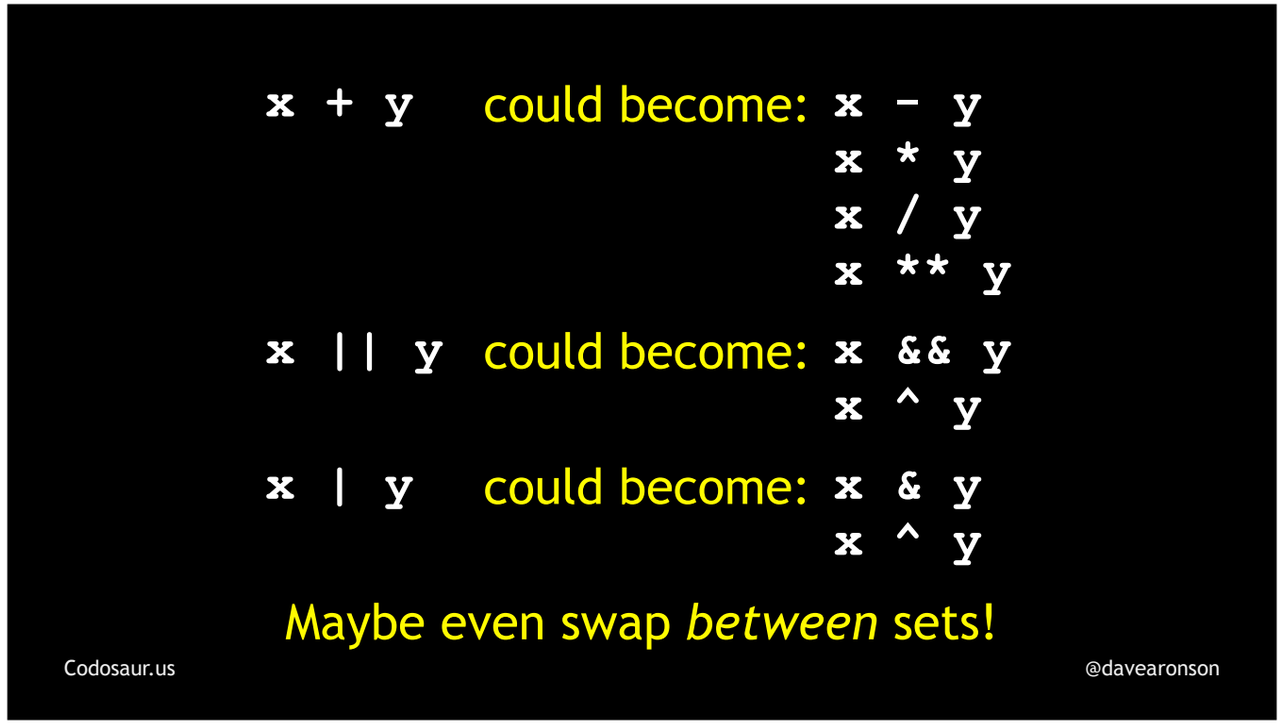


. . . the *next* node. Again, for each way it could change *that* node, it would make a copy of this whole subtree, with only that mutation. And so on, until it has . . .



. . . 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* and *most* things as *bitfields*, so x *times* y could become, for instance, x *and* y, and maybe even x *bitwise-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 an entire *statement*.

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

could become:

```
foo(z)
```


It can remove an if-condition, so that something that might be skipped over or done, is always done.

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

could become:

```
foo(z)
```

Codosaur.us

@davearonson

Taking that idea a step further, it can remove a looping condition, so that something that might be skipped over, done once, or done *multiple* times, is always done exactly once.

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

could become:

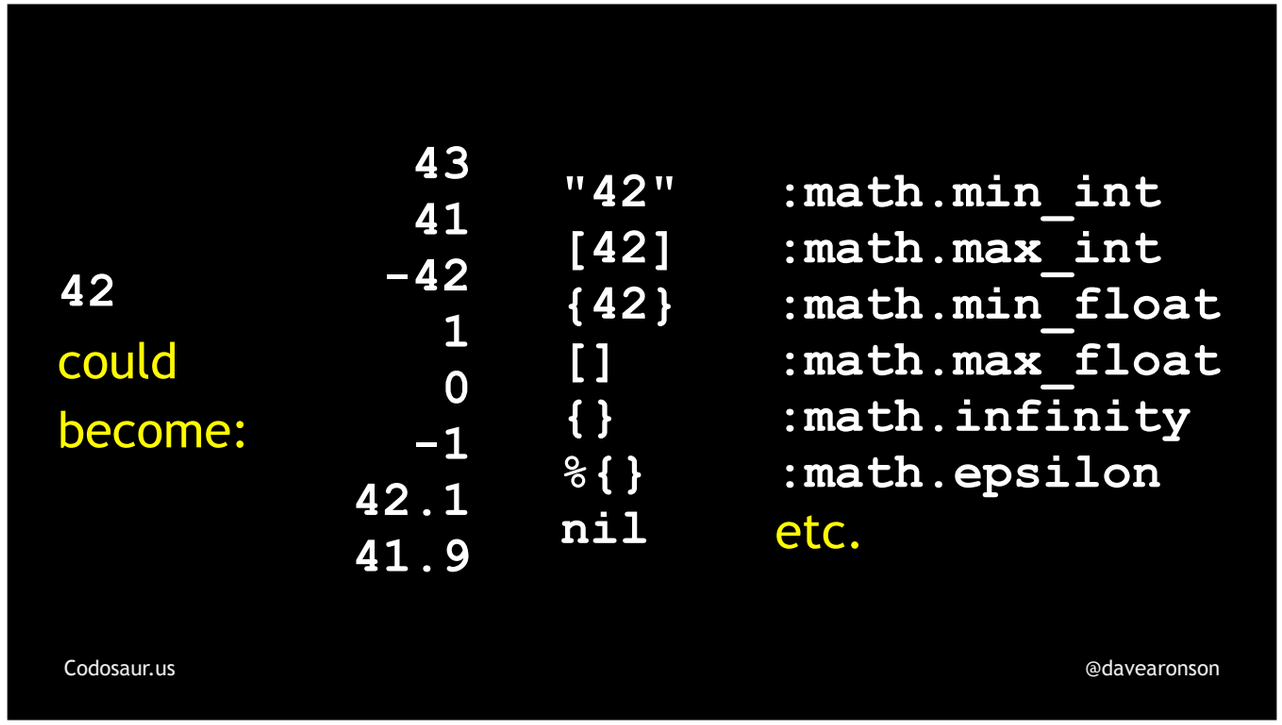
```
def f(x, y), do: 0
def f(x, y), do: :math.max_int
def f(x, y), do: "a string"
def f(x, y), do: nil
def f(x, y), do: x
def f(x, y), do: fail("boom")
def f(x, y), do: # nothing
```

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 changing 42 to any of these, and many more but I had to stop somewhere. It could even change it to something of a different and possibly 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 of technical detail, and look at . . .

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

Codosaur.us @davearonson

. . . some pseudocode illustrating how it works. I'll talk a bit so you have some time to take pictures.

For each function, first we find the tests. If we can't find any, warn about it and go on to the next function. Otherwise, make mutants from this function, and for each mutant, start running the function's tests against it. If any test fails, immediately go on to the next mutant, skipping any further tests against this one. If they all passed, report the mutant as surviving.

Sounds pretty simple once we see it in code, right? The hard part isn't so much understanding what mutation testing does, or how, or even why, but rather, figuring out *what we should do* about each reported surviving mutant.

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. It sure doesn’t look like it has side effects.

Mainly, such a mutant 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.

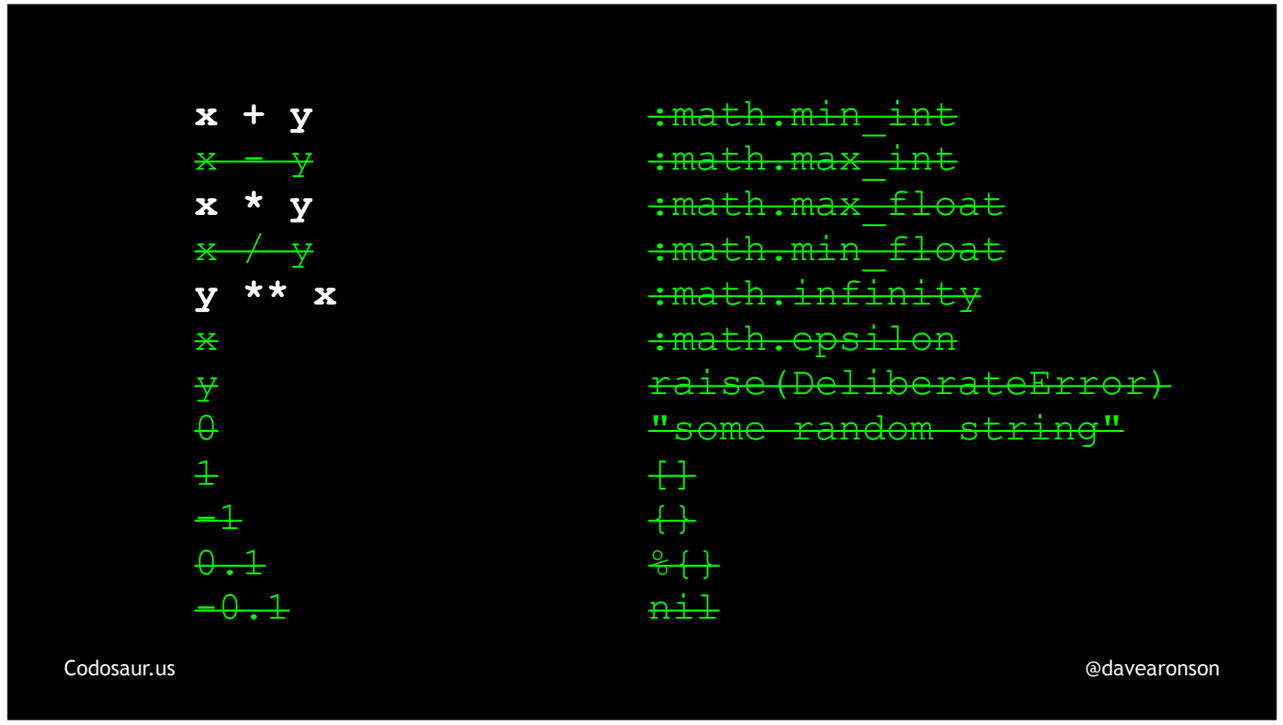
Now suppose we had only one test . . .

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

Codosaur.us

@davearonson

. . . like so. This is a rather poor test, and I think at least one reason 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, -1, minimum and maximum signed and unsigned integers and floats, infinity, minus infinity, and other such significant numbers. Subtracting one argument from the other gets us zero, dividing them gets us one, returning either argument 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. Mutants based on *these* mutations will therefore *"surivive"* our 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, and so on, will vary greatly depending on exactly which tool we use, but *semantically*, 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.

It then goes on to tell us that 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 the function declaration on 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 the function body on 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 of the exponentiation.

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? We can tell from a glance at . . .

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

Codosaur.us

@davearonson

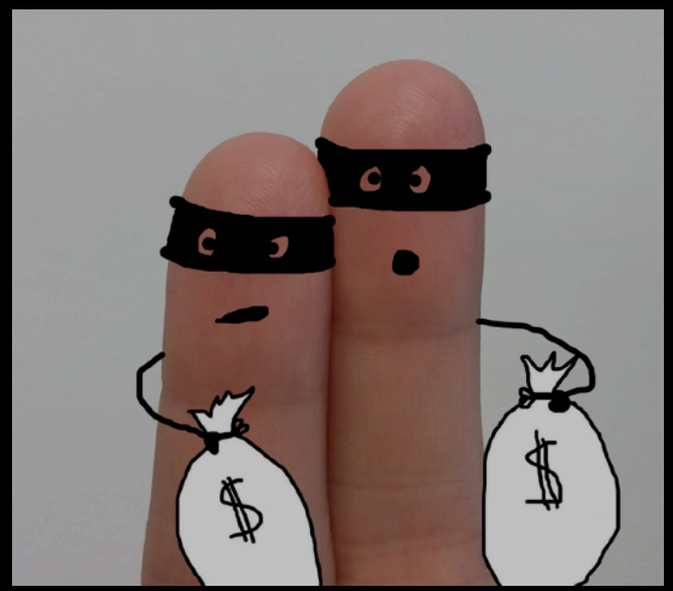
. . . our code, that it's probably not trying to tell us about redundant or unreachable code. The body is just one line so that sort of problem is extremely unlikely. So it's almost certainly a test gap, meaning that the very high level message is that our test suite is not sufficient, either because there aren't enough tests, or the ones we have just aren't very good, or both. But we knew that! The question now boils down to, how 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

. . . these mutants surviving? Are they . . .



Codosaur.us

Image: https://pixabay.com/fi/photos/varkaat-varkaus-ryöstö-nyytti-2012532/

@davearonson

. . . pulling heists? Are they getting free room and board at the . . .

Codosaur.us

Image: https://pixabay.com/fi/photos/varkaat-varkaus-ryöstö-nyytti-2012532/

@davearonson

. . . Xavier Institute? Or what?

The usual answer is that . . .

```
mutant_power(x, y)

==

original_power(x, y)
```

Codosaur.us @davearonson

. . . they return the same result as the original function. Or they have the same side effect — whatever it is that our tests are looking at. To determine how *that* happens, it helps to 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 two (singing) tu tu, tu tu tu-tu tu . . . . (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

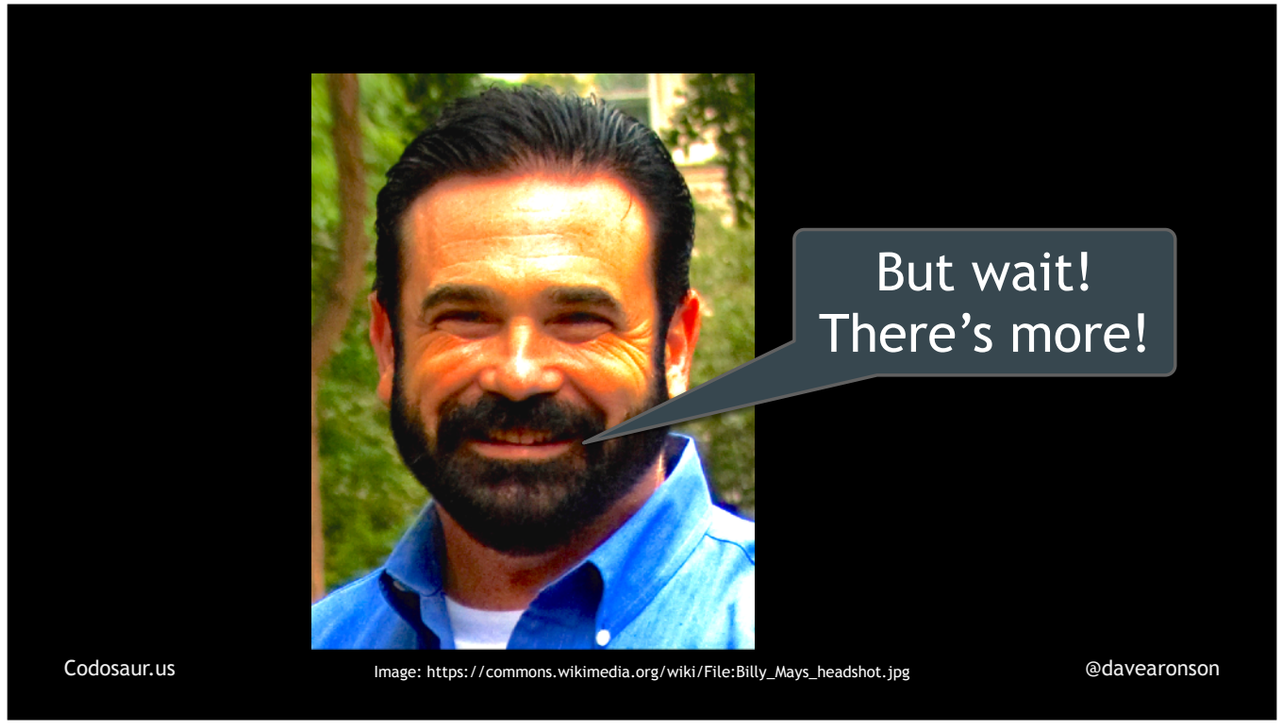
@davearonson

. . . this mutant, in other words, make at least one test fail when run against it, that would pass when run against the original code? It's quite simple in this case. 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 . . .

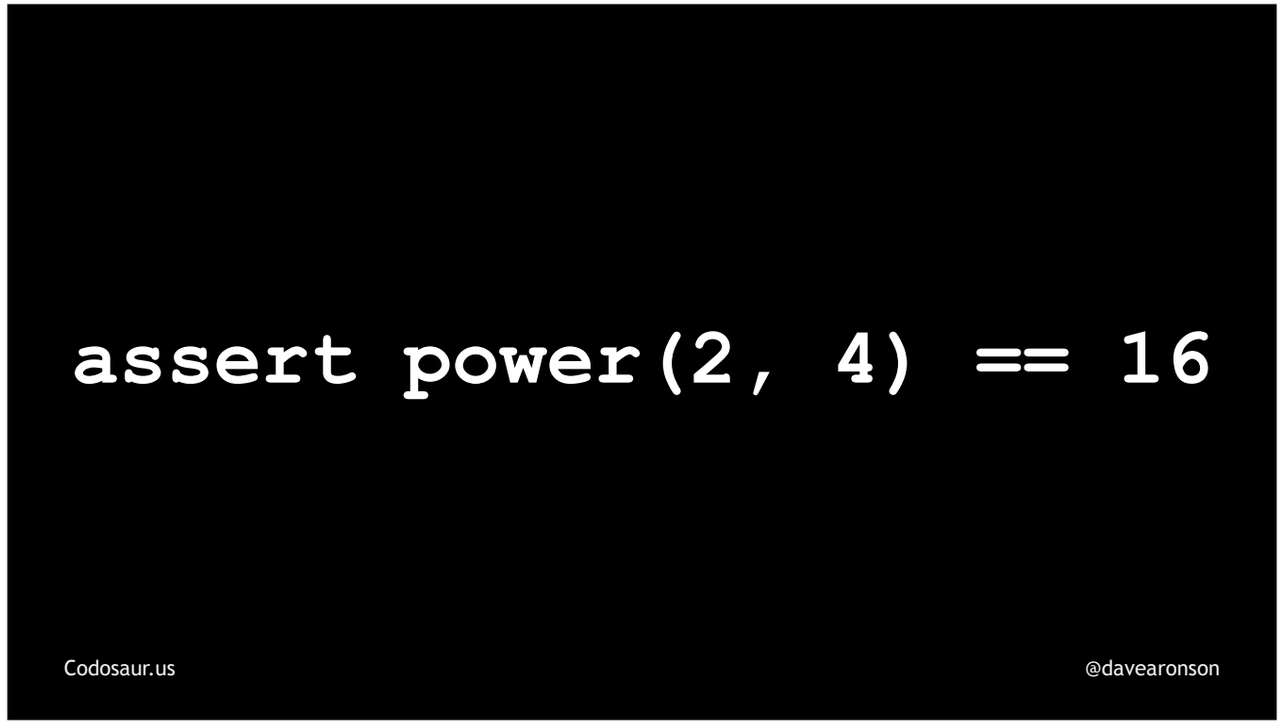
```
assert power(2, 4) == 16
```

Codosaur.us @davearonson

. . . something like this, asserting that two to the *fourth* power is *sixteen*. All the mutants that our original test killed, *this would still kill*. But in addition, two *plus* four is six, not six*teen*, so **this kills the plus mutant.** (See how that works?)



But wait! There’s more!



Two *times* four is eight, which is *also* not sixteen! We devs should certainly know our powers of two at least *that* well! So, this kills the "times" mutant as well. Killing one mutant often kills other mutants of the same function, often a large fraction of them.

However, in this case, . . .

. . . the (ahem) pair of argument-swapping mutants survive! What, how can that be? It’s because even with our new test inputs . . .

```
mutant_power(x, y)

==

original_power(x, y)
```

Codosaur.us

@davearonson

. . . these mutants return the same result as the original function, because . . .

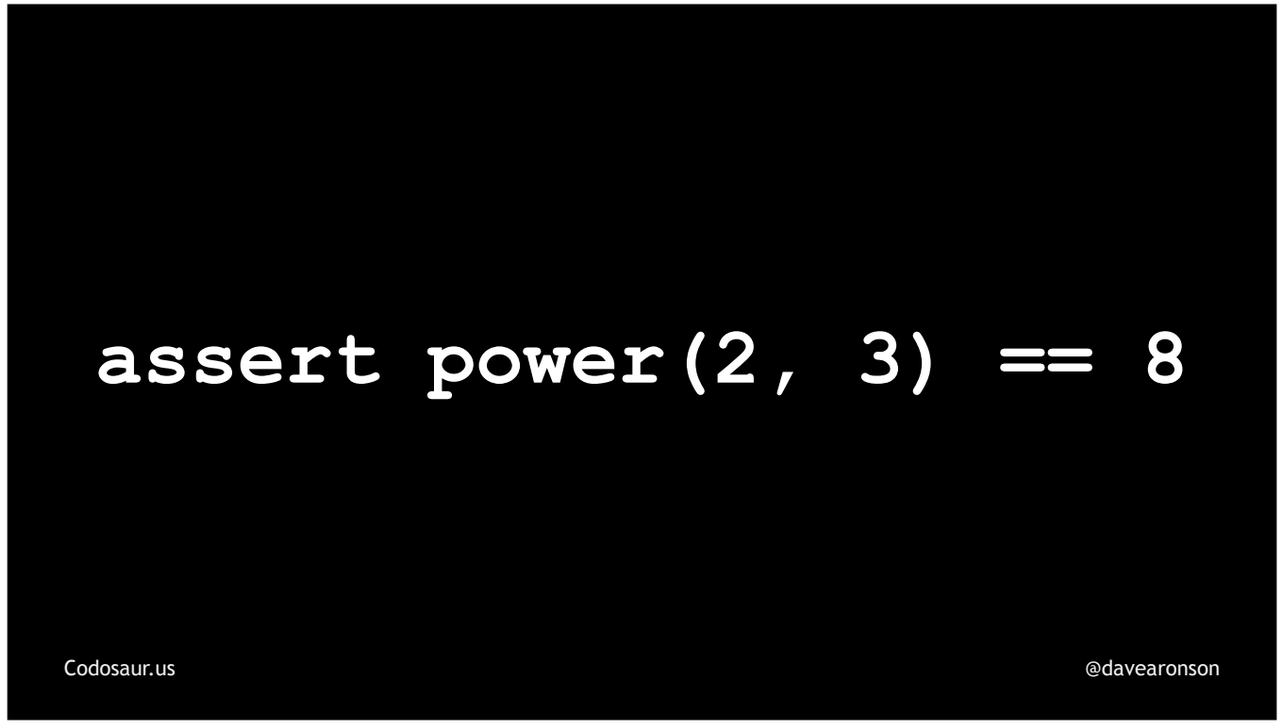
```
4 ** 2 == 16

2 ** 4 == 16
```

Codosaur.us

@davearonson

. . . four squared is the same as two to the fourth, they’re both sixteen. But that’s not a big deal, we can . . .

. . . attack these mutants separately, no need to kill all the mutants 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 something like . . .



. . . this, asserting 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 . . .

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

Codosaur.us @davearonson

. . . 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; even if we stuck to single digits, 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 was 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(buf, len)
  sent = 0
  while sent < len
    sent += send_bytes(buf + sent,
                       len - sent)
  end
  sent
end
```

Codosaur.us @davearonson

. . . like so. This function, send_*message*, uses send_*bytes* to send as many bytes as send_bytes *could* send, kind of like a woodchuck, looping to pick up where it left off, 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(buf, len)
    sent = 0
-   while sent < len
      sent += send_bytes(buf + sent,
                         len - sent)
-   end
    sent
  end
```

Codosaur.us @davearonson

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

That would make the code read effectively like . . .

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

Codosaur.us

@davearonson

. . . this.

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)
  end
  sent
end
```

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!) By the way, you'll find that interpreting mutants 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 should have, but 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 a phenomenal 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 an itty-bitty *three* byte message. Or maybe four if we have a null terminator, maybe a bit more depending just how it's serialized, but almost certainly nowhere near 10_000. (PAUSE!)

The obvious fix is to deliberately use a message larger than our maximum chunk size. With this kind of message, 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 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 at least 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)
  sent
end
```

Codosaur.us @davearonson

. . . this code I showed you earlier. 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, lather rinse repeat, then it boils down to . . .

```
def send_message(buf, len)
  send_bytes(buf, len)
end
```

Codosaur.us @davearonson

. . . this. (PAUSE!) Now the ultimate message is crystal 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, with unreachable, redundant, or otherwise meaningless code, 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 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 then, it was not *practical* on typical developer-grade computers, until the early 2000s, 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- and memory-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

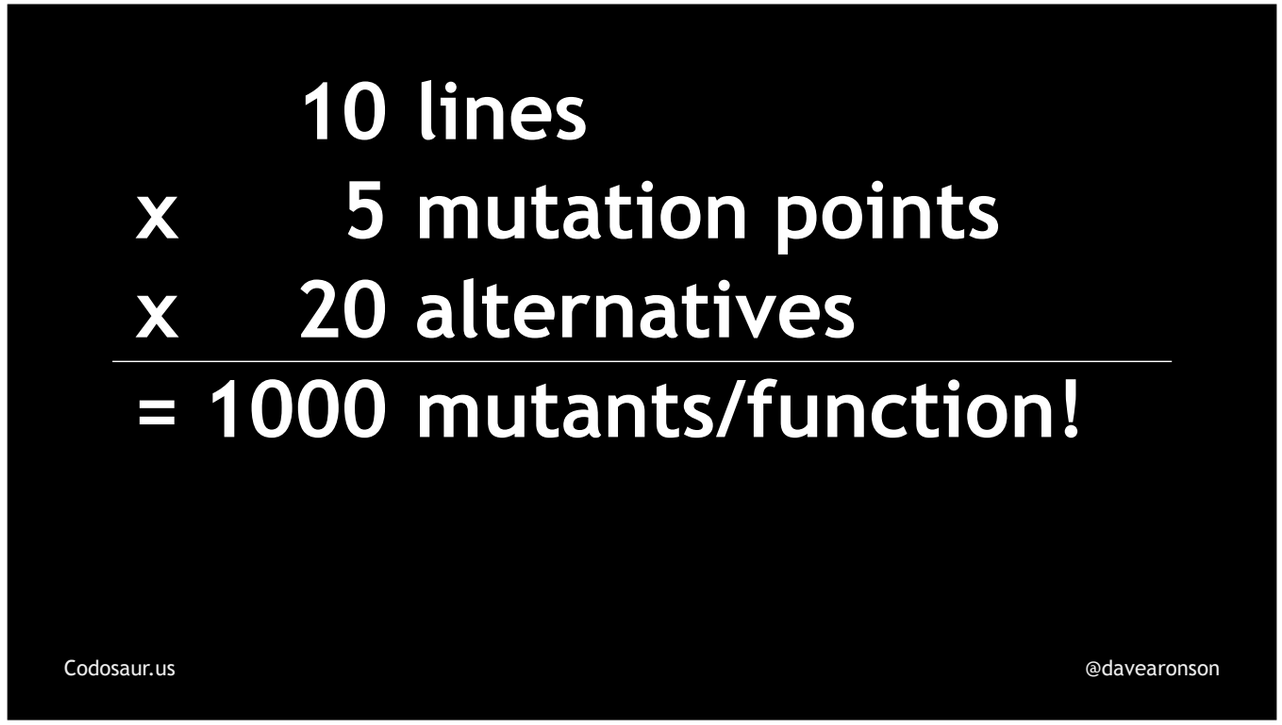
. . . five places where it can be mutated, to any of about . . .

```
      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, all the way up to *all* of that function's tests, if we kill it on the last try, or worse yet, it survives.

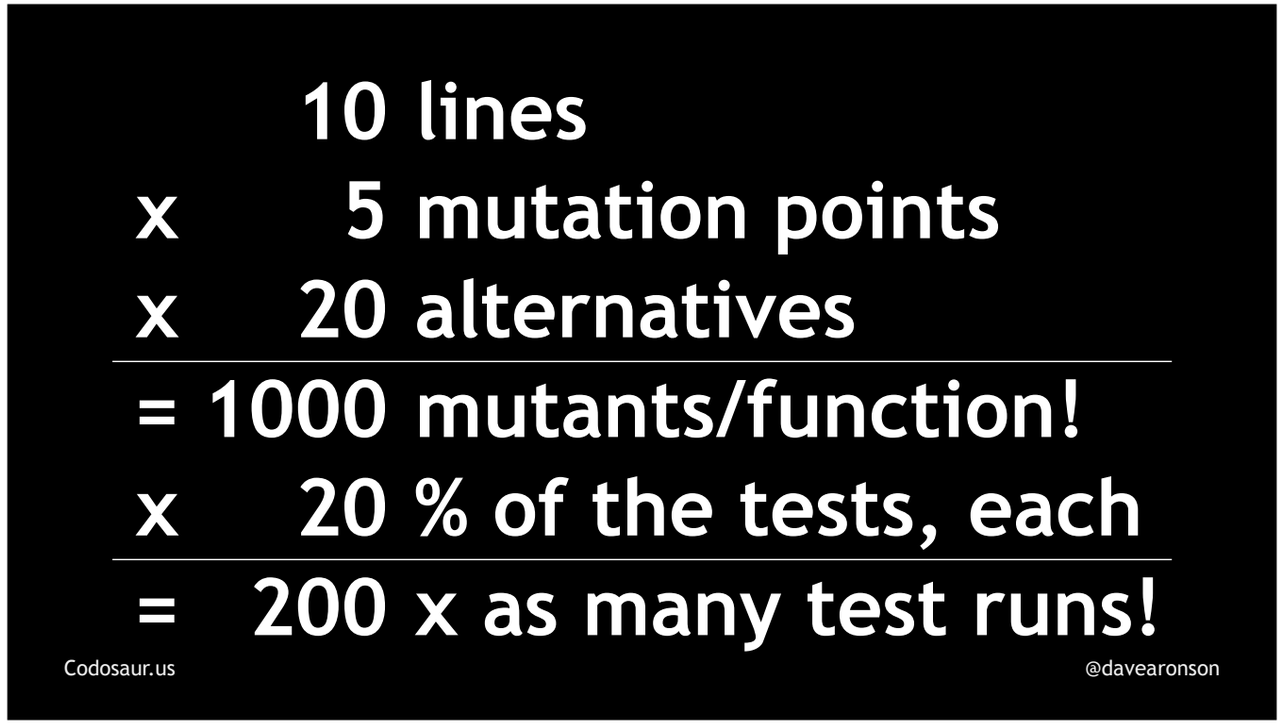
Suppose we wind up running just . . .

```
      10 lines
x      5 mutation points
x     20 alternatives
-----------------------
= 1000 mutants/function!
x     20 % of the tests, each
-----------------------
```

Codosaur.us

@davearonson

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



. . . *two hundred times* the test runs for that function, compared to regular testing. If our test suite normally takes a zippy ten seconds, then with these assumptions, mutation testing will take about *two thousand* seconds. That might not sound like much, because I'm saying "seconds", but do the math and it's *over 33 minutes*, in other words, a bit over *half an hour!* I don’t know about you, but I don’t want to sit and wait for that.

But there is some . . .

Codosaur.us

Image: https://www.flickr.com/photos/akuchling/50310316

@davearonson

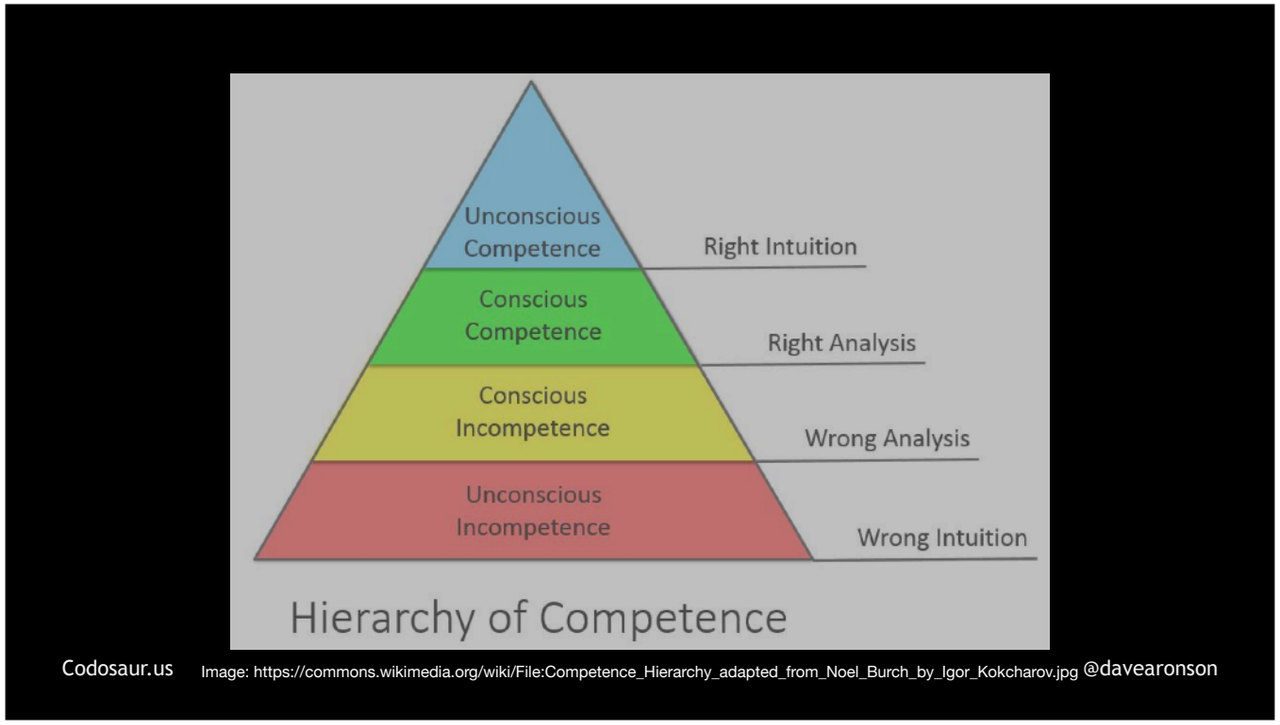
. . . good news! Over the past decade or so, there has been a lot of research on trimming down the number of mutants, mainly by weeding out ones that are semantically equivalent to the original code, redundant with other mutants, or trivial in various ways such as creating an obvious error condition. Such things have reduced the mutant horde down to about one third! But even with that rare level of success, it's still . . .

. . . no silver bullet, as this takes lots of CPU time itself -- and there are still quite a lot of mutants left to deal with.

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

. . . one way?

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



. . . 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.

So 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 comparing using "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! So we can think of mutation testing as sort of a "did you mean" function, like how Google suggests a different search if ours 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. Lastly, allowing multiple mutations would create a combinatorial . . .

. . . explosion of mutants, with the tool making many *orders of magnitude* more mutants per function, which would make it even *more* CPU- and memory-intensive. I'll spare you the math, but with our earlier code size assumptions, even if we manage to weed the mutants down to one third at each step, with . . .

Mutations/Mutant vs Mutants/Function

1:

Codosaur.us

@davearonson

. . . one mutation per mutant, we'd have . . .

Mutations/Mutant vs Mutants/Function

1: 333

Codosaur.us

@davearonson

. . . 333 mutants per function (and a third, but I'm rounding). With . . .

Mutations/Mutant vs Mutants/Function

1: 333
2:

Codosaur.us

@davearonson

. . . two, we'd already have . . .

Mutations/Mutant vs Mutants/Function

1: 333
2: almost 110_000

Codosaur.us

@davearonson

. . . almost 110,000, and with . . .

Mutations/Mutant vs Mutants/Function

1: 333
2: almost 110_000
3:

. . . three we'd already have . . .

Mutations/Mutant vs Mutants/Function

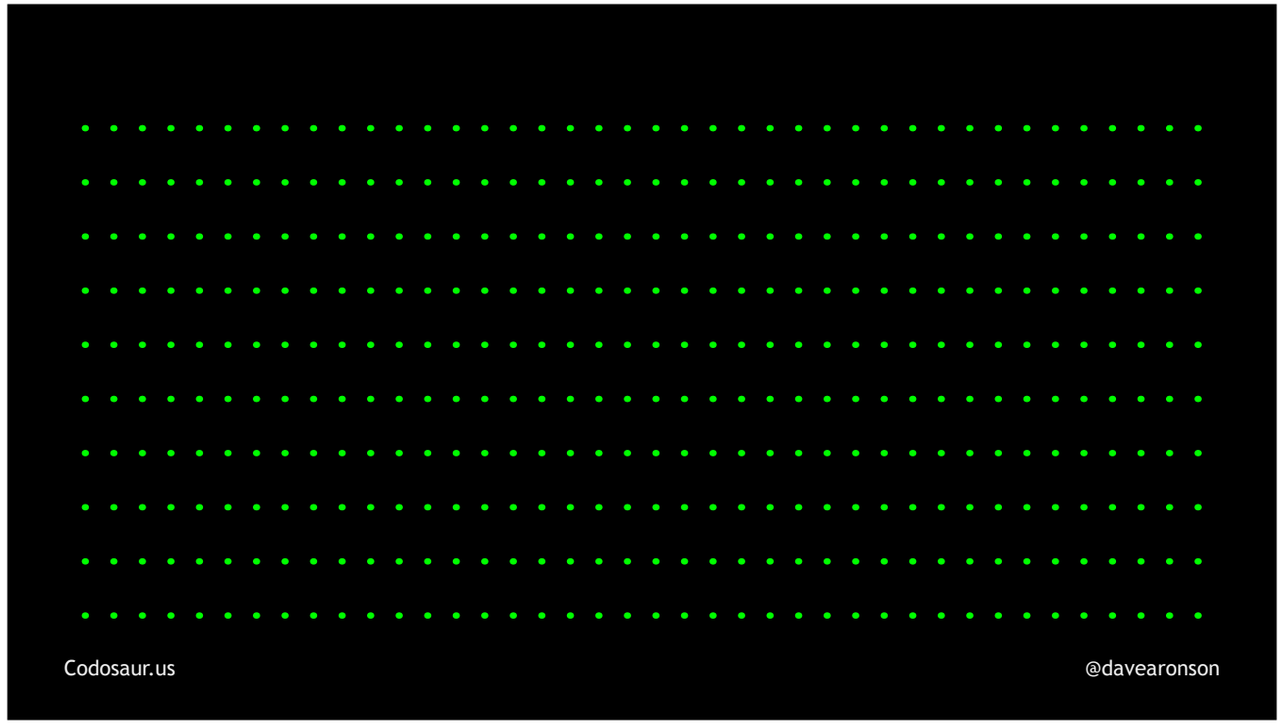
1: 333
2: almost 110_000
3: over 35_000_000

Codosaur.us @davearonson

. . . over 35 million! Never mind actually *running* the *tests*, just *creating* the *mutants* would get to be quite a heavy workload! But we can avoid this huge workload, *and* the increased false alarms, *and* the lack of focus, if we just . . .

. . . limit it to one mutation per mutant.

Now since we have time, let’s delve into some more unusual questions. First, 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 of the current tools give us all the necessary information. You may remember how I said early on, that when . . .

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 mutant makes a test fail, 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, . . .

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

. . . *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. So when we're done with a given function, we wind up with a chart like . . .

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

. . . this. If we were to run the rest of the combinations, the blank cells in this table, that would take a lot longer, 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!

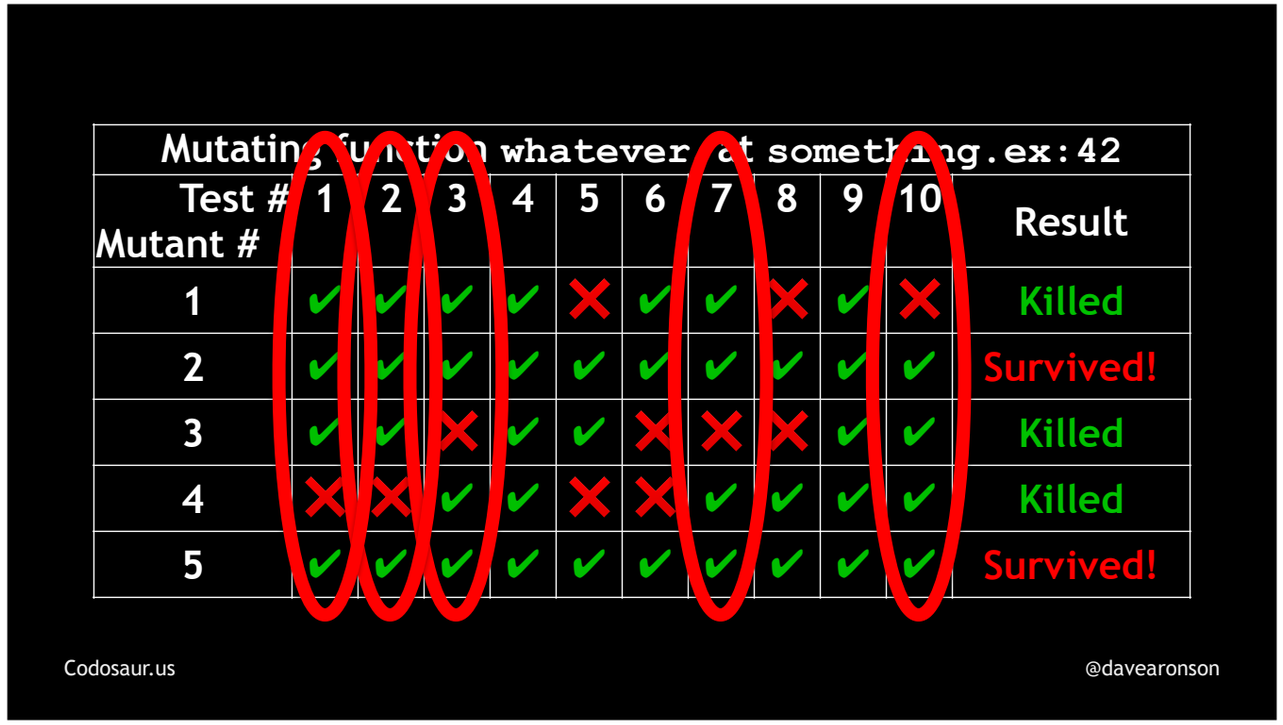
. . . more information, that we can use to assess the quality of *some* individual tests. Look at . . .

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!

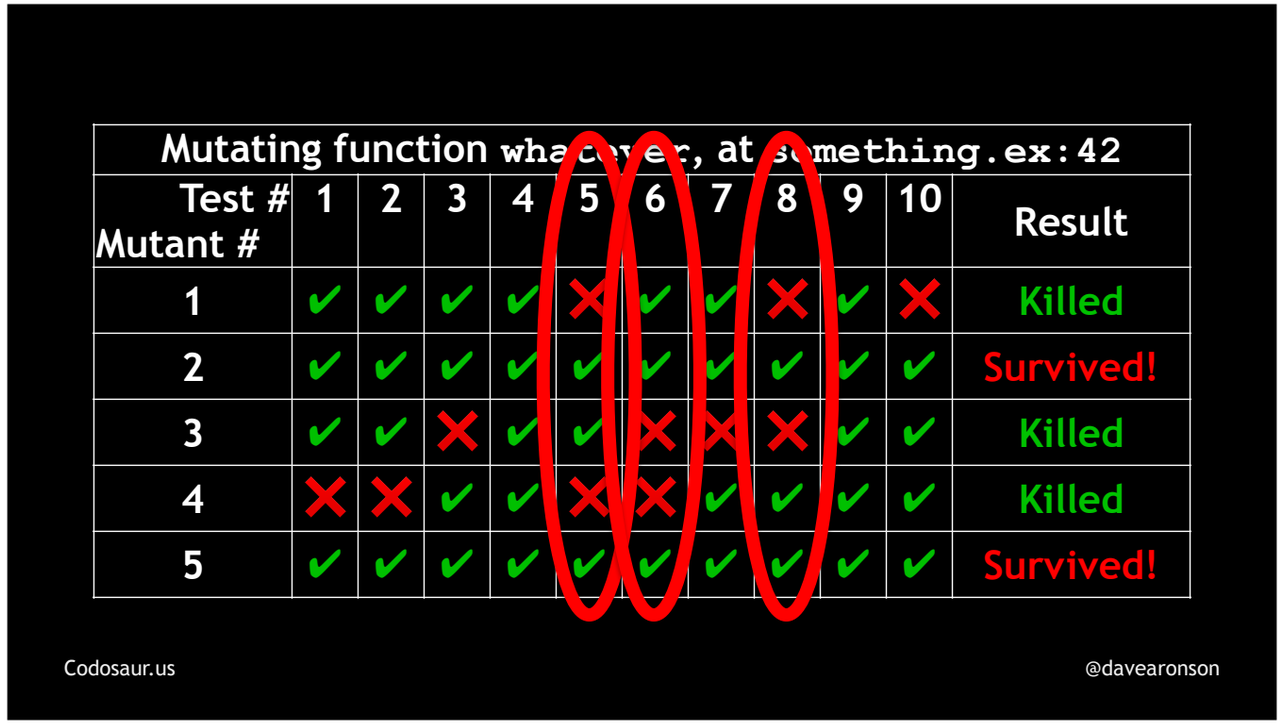
Codosaur.us

@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 . . .



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

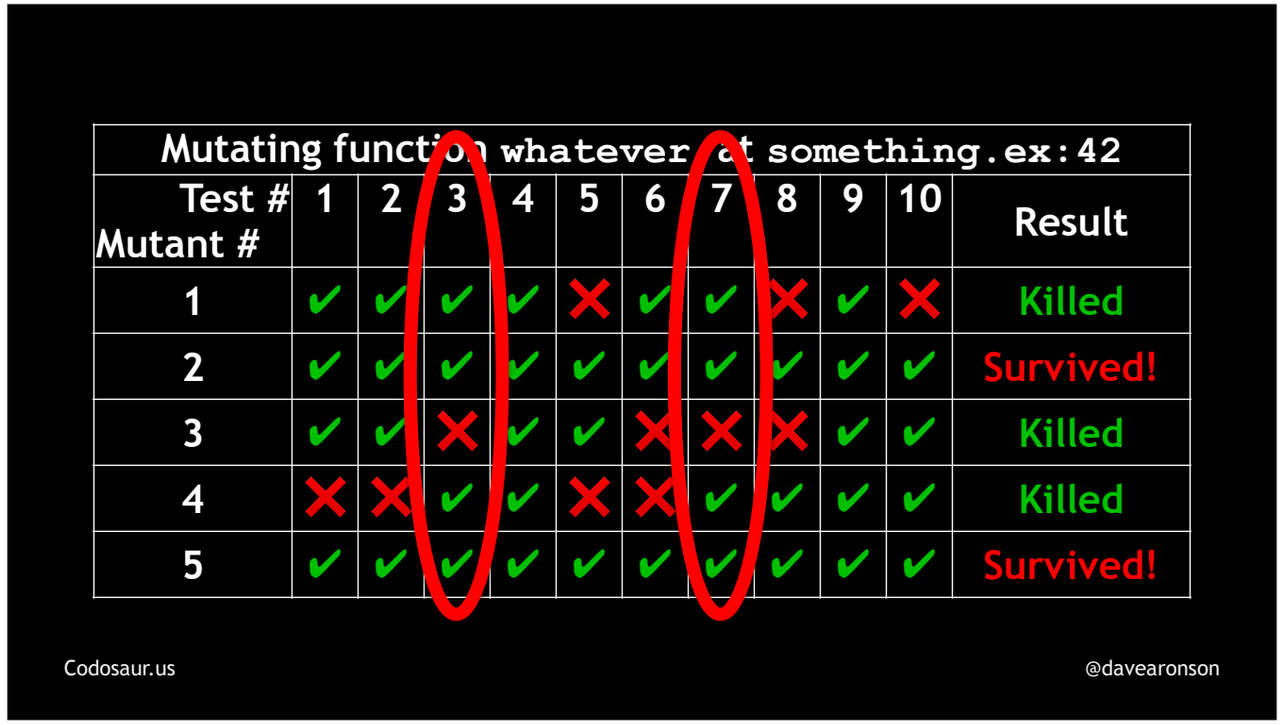
This raises the question of whether we could also use such a full report to improve our test *suite* as a *whole* again. Yes we could, by looking at *pairs* or larger *sets* of tests, that kill exactly the same sets of mutants, such as . . .

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 one and two, or . . .



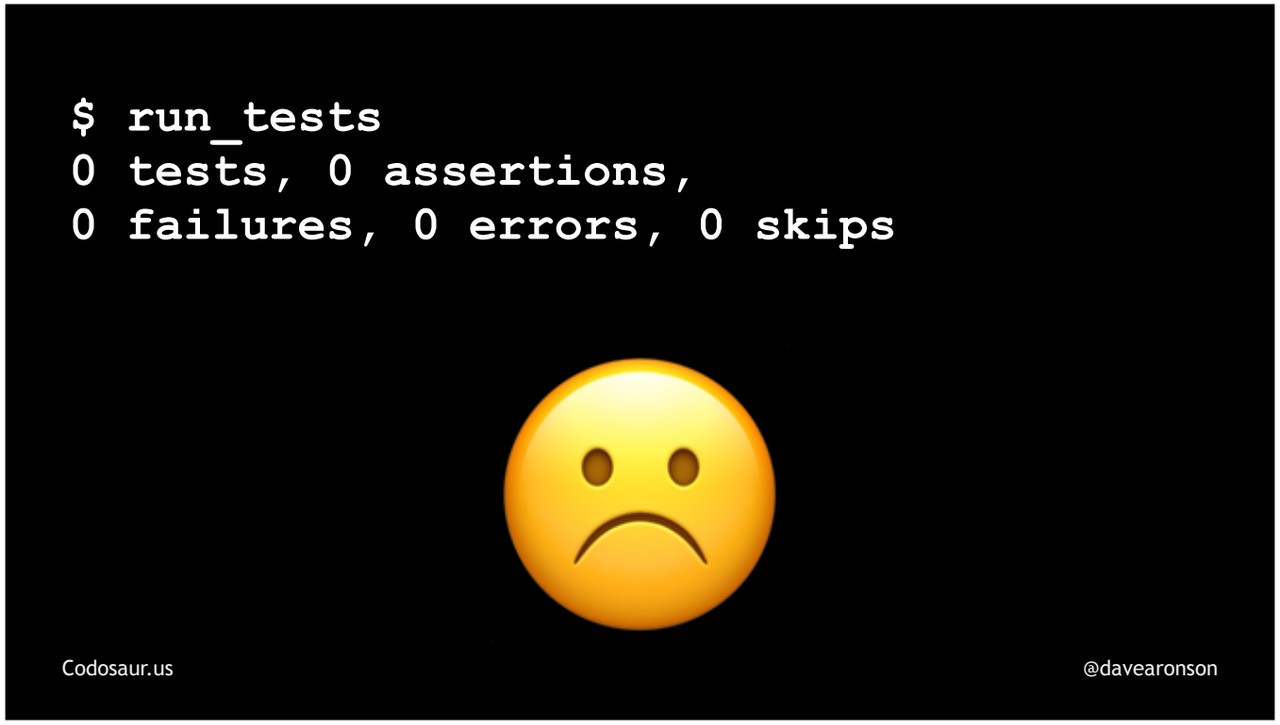
. . . tests three and seven. We can take a closer look at those sets of tests, and decide if we need to keep the whole set. Maybe tests one and two are testing different important aspects, but three and seven are essentially testing the same thing, so we can get rid of one of them, not due to poor quality of either test, but *redundancy between* them. This will make our test suite slightly faster, but just as thorough.

The last question is: as mentioned earlier, mutation testing *assumes* that we have . . .

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

Codosaur.us @davearonson

. . . tests already. What if . . .



. . . we don't? 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, like with a clue-by-four. (Uh-oh, there's that "violent communication" again!) But yes, mutation testing can help us . . .

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

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

Codosaur.us @davearonson

. . . meaningless test. When we run it, we'll probably get . . .

Codosaur.us

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

@davearonson

. . . lots and lots of mutants, including many *duplicates* telling us about the same problem, especially since any basically *viable* mutant will survive our meaningless test. Now, for each function with surviving mutants, we can . . .

. . . pick one mutant. We can just do it randomly, no need to overthink it. Add one test that we think would kill it. As mentioned before, this will probably kill many other mutants of the same function, so move on to the next function, until we've killed one mutant from each function that had any survivors. Then we can rerun our tool, and lather, rinse, repeat, though on further iterations we might *improve* a test rather than *add* any. Now, this won't . . .

. . . *guarantee* that we wind up with a great test suite. Some code might remain . . .

```
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

. . . untested. However, this idea will get us off to a decent start. Then we can look at what code is untested, and write more tests to cover that, subjecting them to mutation testing of course. At this point, never mind computer science, at least psychologically it will be a much less daunting task than writing our whole test suite from scratch.

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

😀 Checks that code is meaningful

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

😀 Checks that code is meaningful

😀 Checks that tests are strict

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

😀 Checks that code is meaningful

😀 Checks that tests are strict

😀 Easy to get started with

Codosaur.us

@davearonson

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

😀 Checks that code is meaningful

😀 Checks that tests are strict

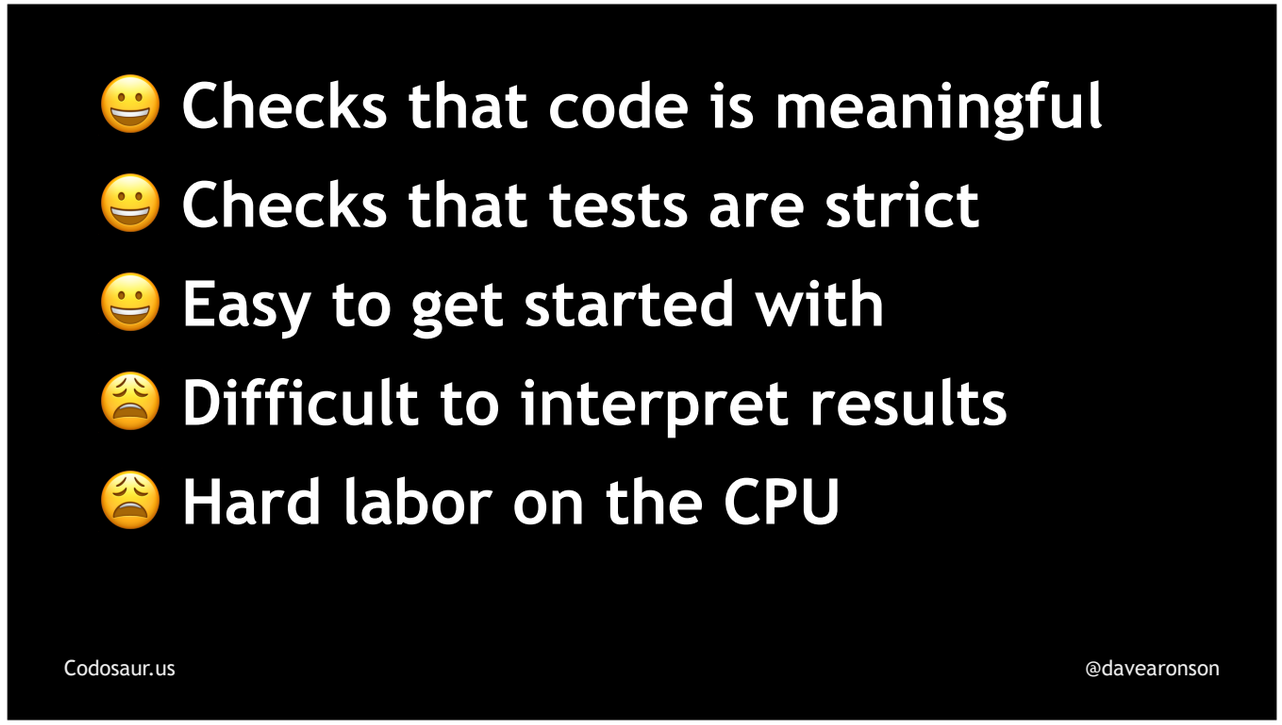
😀 Easy to get started with

😩 Difficult to interpret results

Codosaur.us

@davearonson

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



. . . easy on the CPU.
Even if these drawbacks mean it might not be a good fit for our particular current projects right now, 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! 🤓

Codosaur.us

@davearonson

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

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

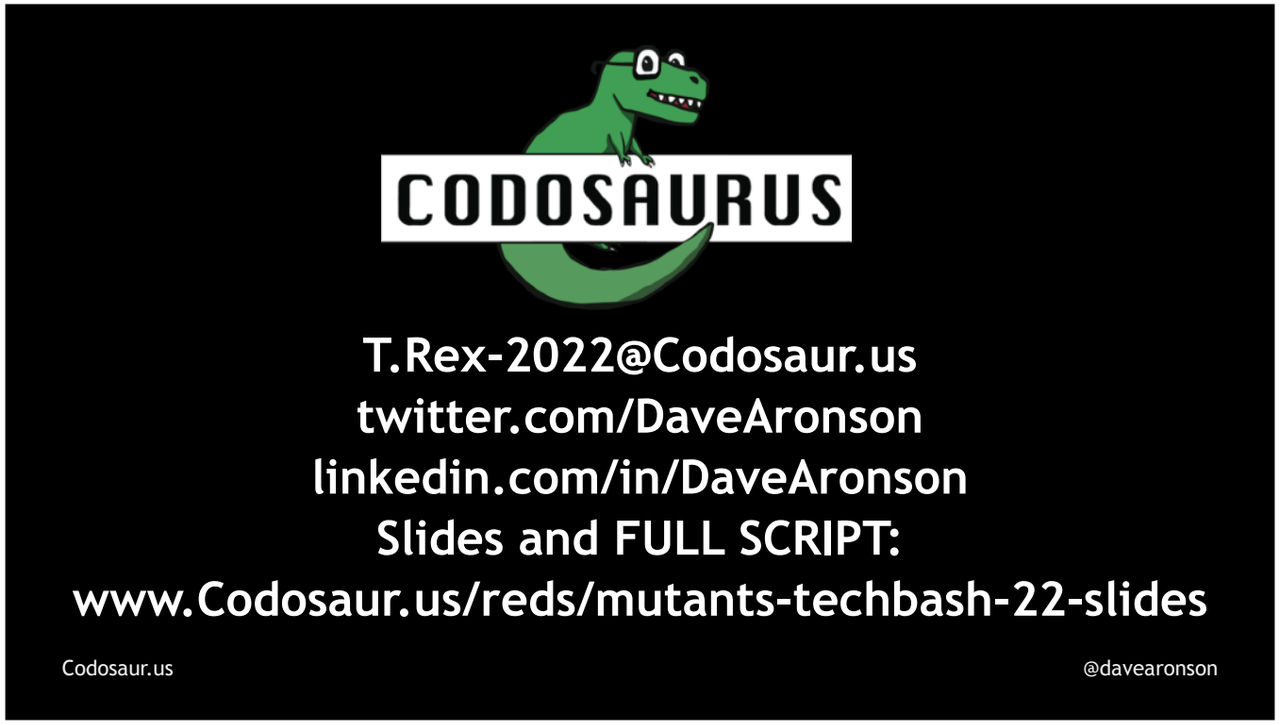
Alloy:	MuAlloy
Android:	mdroid+
C:	mutate.py, SRCIROR
C/C++:	accmut, dextool, MART, MuCPP, Mutate++, mutate_cpp, SRCIROR
C#/.NET/Mono:	nester, NinjaTurtles, Stryker.NET, Testura.Mutation, VisualMutator
Clojure:	mutant
Crystal:	crytic
Dart:	mutation_test
Elixir:	darwin, exavier, exmen, mutation, Muzak [Pro]
Erlang:	mu2
Etherium:	vertigo
FORTRAN-77:	Mothra (written in mid 1980s!)
Go:	go-mutesting, gremlins
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, 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/)

. . . 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. Just be aware that many of these are outdated; I don't know or even follow quite *all* of these languages and platforms, let alone the tools. 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.)

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 productions of this presentation. (The preceding sentence brought to you by the American Association for the Advancement of the Alliterative and Assonantal Arts — AAAAAA!) Please use that referral link if you want to hire us or join us, and that's also where that QR code goes. The anchor part tells them it’s me, and if you hire us or work a project through us, we’ll both get some bounty.

And now . . .



. . . it's your turn! Any questions?