Kill All Mutants!

(Intro to Mutation Testing)

by Dave Aronson

T.Rex-2022@Codosaur.us

www.Codosaur.us/reds/mutants-gambi-22-slides

NOTE TO SELF: Current time ~23:30, want ~25, can add some more, maybe ad-lib a bit

Olá, Lisboa!

(Hello, Lisbon!)

www.Codosaur.us

Image: standard emoji

@davearonson

O meu nome é Dave Aronson,

(I'm Dave Aronson,)

www.Codosaur.us

Image: me speaking at JSConf Hawai'i 2020

@davearonson

sou o T. Rex de Codosaurus,

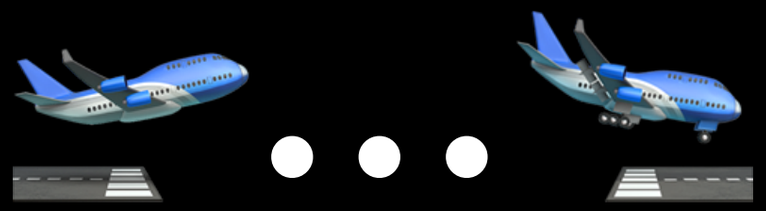
(the T. Rex of Codosaurus,)

www.Codosaur.us

Image: my company logo!

@davearonson

e voei aqui no meu



(and I flew here on my)

pterodátilo de estimação

(pet pterodactyl)

www.Codosaur.us

Images: https://pixabay.com/vectors/dinosaur-tyrannosaurus-t-rex-6273164/
and https://pixabay.com/vectors/bird-flying-wings-dinosaur-ancient-44859/

@davearonson

para vos ensinar

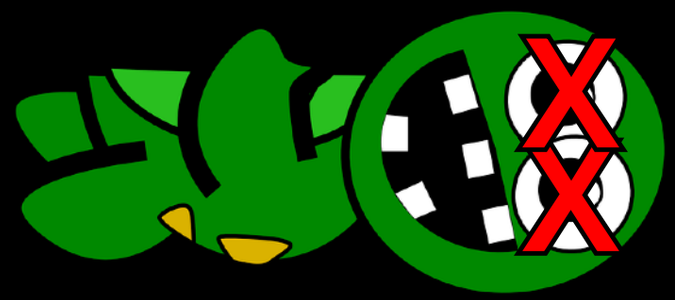
(to teach you)

www.Codosaur.us

Image: standard emoji

@davearonson

a *matar mutantes!*



(to *kill mutants!*)

www.Codosaur.us

Image: https://pixabay.com/vectors/turtle-tortoise-cartoon-animal-152079/

@davearonson

Mas . . .

(But . . .)

www.Codosaur.us

Image: standard emoji

@davearonson

vou fazê-lo

(I will do it)

www.Codosaur.us

Image: https://www.publicdomainpictures.net/en/view-image.php?image=76001

@davearonson

em inglês.

(in English.)

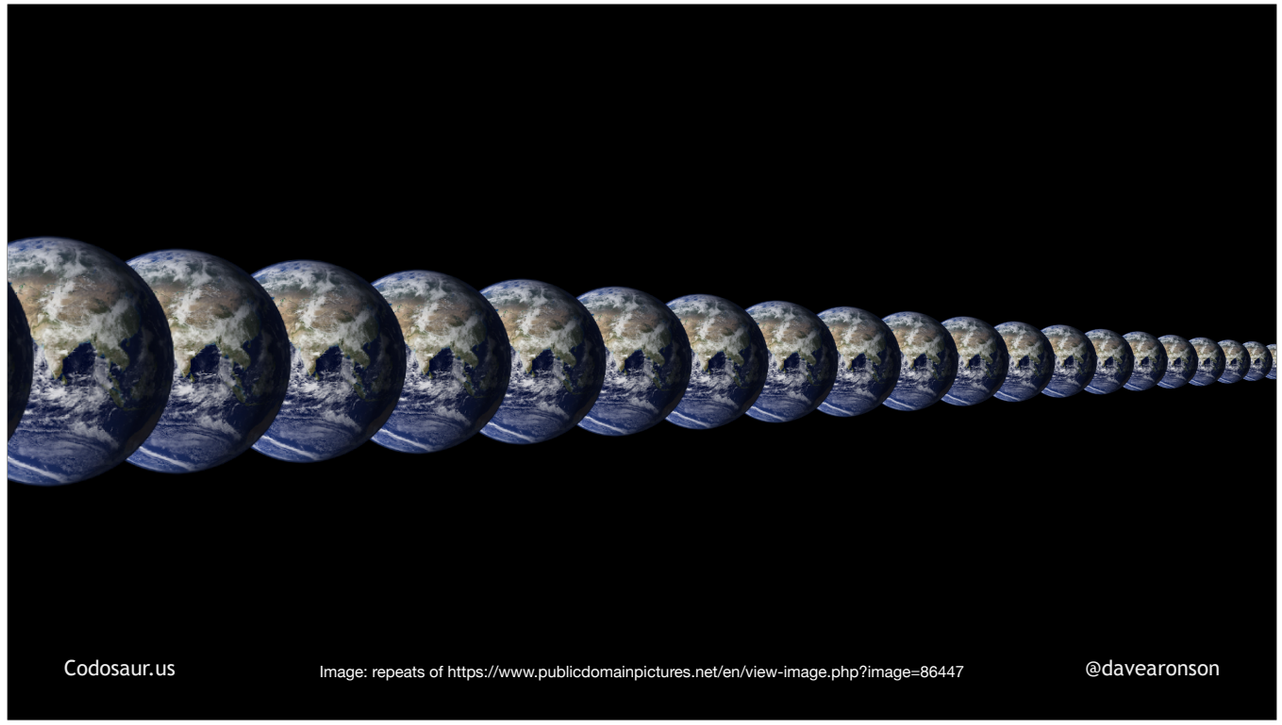
www.Codosaur.us

Image: standard emoji

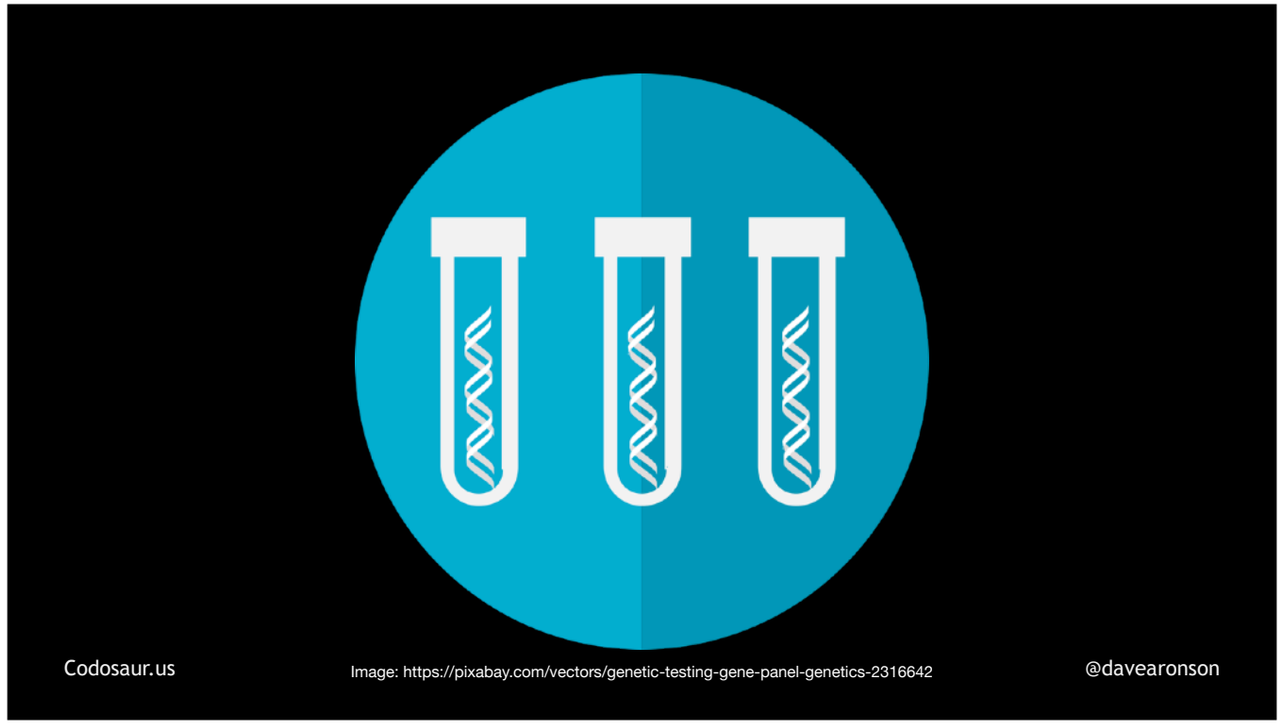
@davearonson

(PAUSE!)

So what on . . .



. . . Infinite Earths, is the big difference between . . .



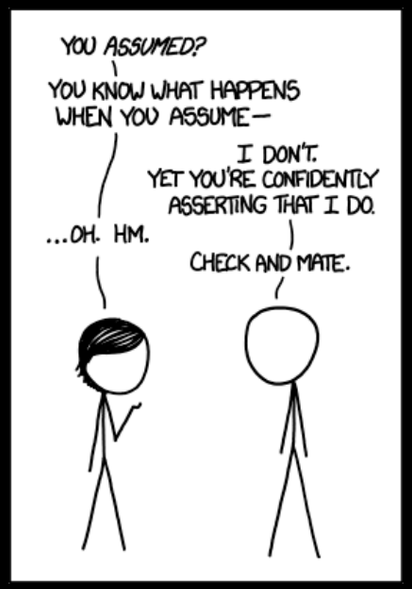
. . . mutation testing and all *other* software testing techniques? Most of the others are about . . .

Codosaur.us

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

@davearonson

. . . checking whether our code is correct, but mutation testing . . .

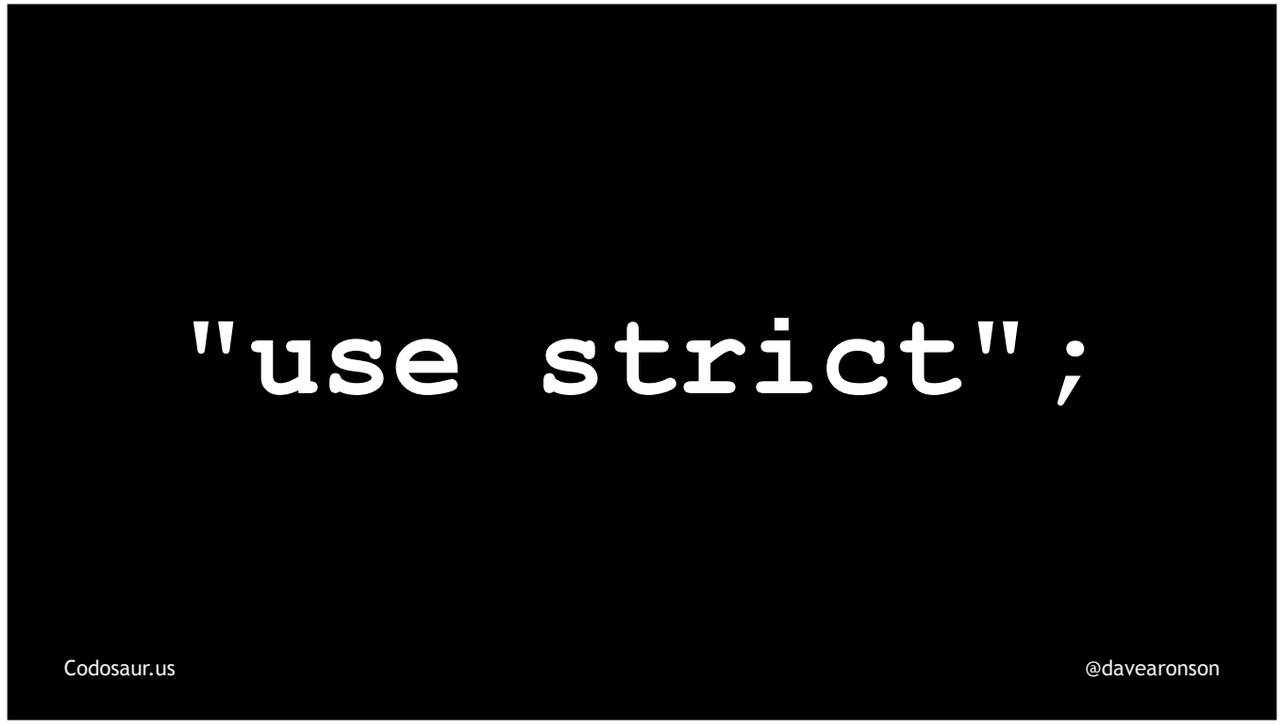


Codosaur.us

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

@davearonson

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



. . . *strict*. To check this, a mutation testing tool will . . .

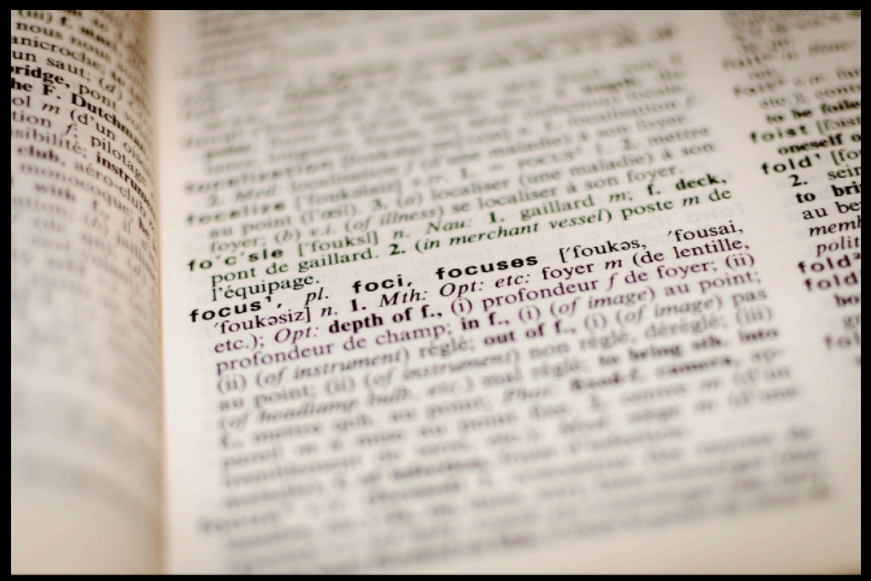
Codosaur.us

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

@davearonson

. . . find the gaps in our test suite. Once we find them, we can close them by adding tests or improving existing tests. Lack of strictness comes mainly from *lack* of tests, or poorly *written* tests.

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



Codosaur.us

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

@davearonson

. . . *meaningful*, meaning that any semantic change to the code, will produce a noticeable change in its behavior. Lack of *meaning* comes mainly from code being unreachable, redundant, or otherwise without real effect. Once we find "meaningless" code, the main fix is to just remove it.

Mutation testing . . .

Codosaur.us

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

@davearonson

. . . puts these two together, by checking that any change to the code, does change its behavior, *and* that at least one test notices that change, and fails.

That's the upside, 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!

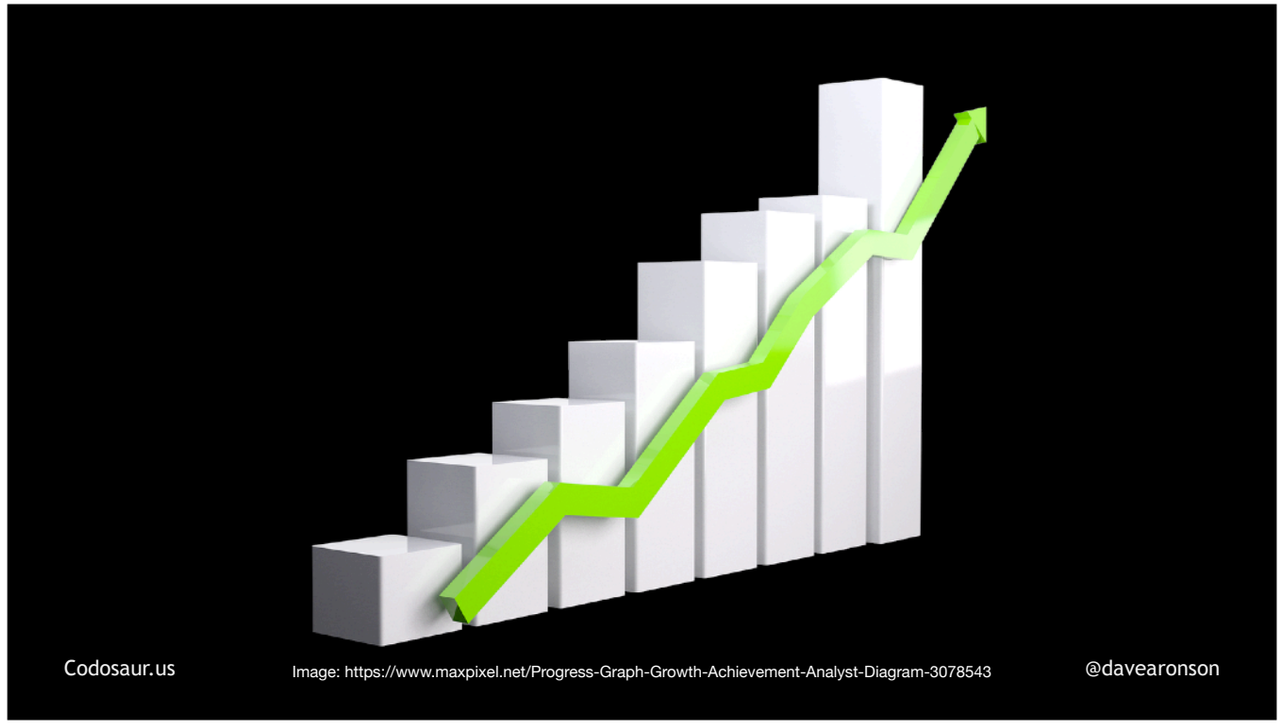
The first drawback is that it's rather . . .

Codosaur.us

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

@davearonson

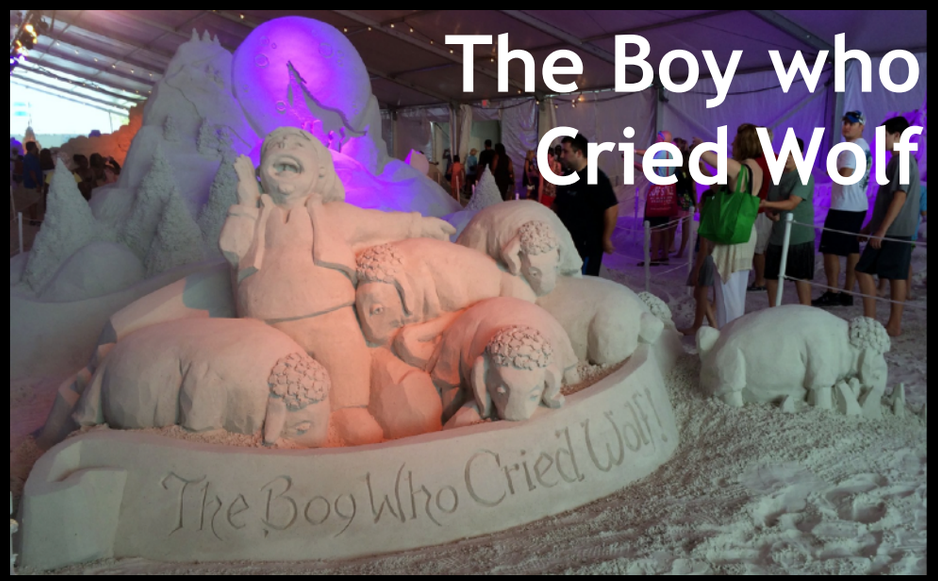
. . . hard labor for the CPU, and therefore usually ra-ther sloooow. We certainly won't want to mutation-test our whole codebase every time we save a file! Maybe over a lunch break for a small system, or a weekend for a large one. Fortunately, most tools let us just check specific functions, modules,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



. . . incremental mode, so that we can test just the changes since the last mutation test, or the last git commit, or the main branch, or some such delta. With such filters, maybe we can do it on each save, or at least over a much shorter break.

Another drawback is that it’s often . . .

. . . not clear what to do! It tells us that some change to the code made no difference to the test results, but what does that even *mean?* It *usually* means that our code is meaningless, or our tests are lax, or both, but it can be very hard to figure out exactly *why!* Even worse, sometimes it's a . . .

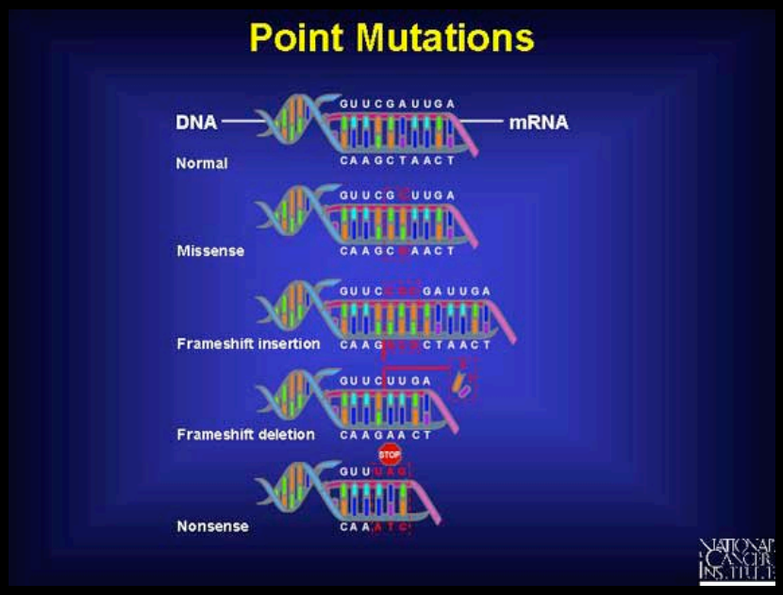


. . . false alarm, because the mutation didn't make a test fail, but it didn't make any real difference in the first place. It can still take quite a lot of time and effort to figure *that* out.

Even if a mutation *does* make a difference, most programs contain a lot of code that we . . .

. . . *shouldn't bother* to test, like a debugging log message. Fortunately, most tools have ways to say "don't bother mutating this line", or function or whatever, but that's usually with comments, which can clutter up the code, and make it less readable.

So how does mutation testing work — unlike this guy? It . . .

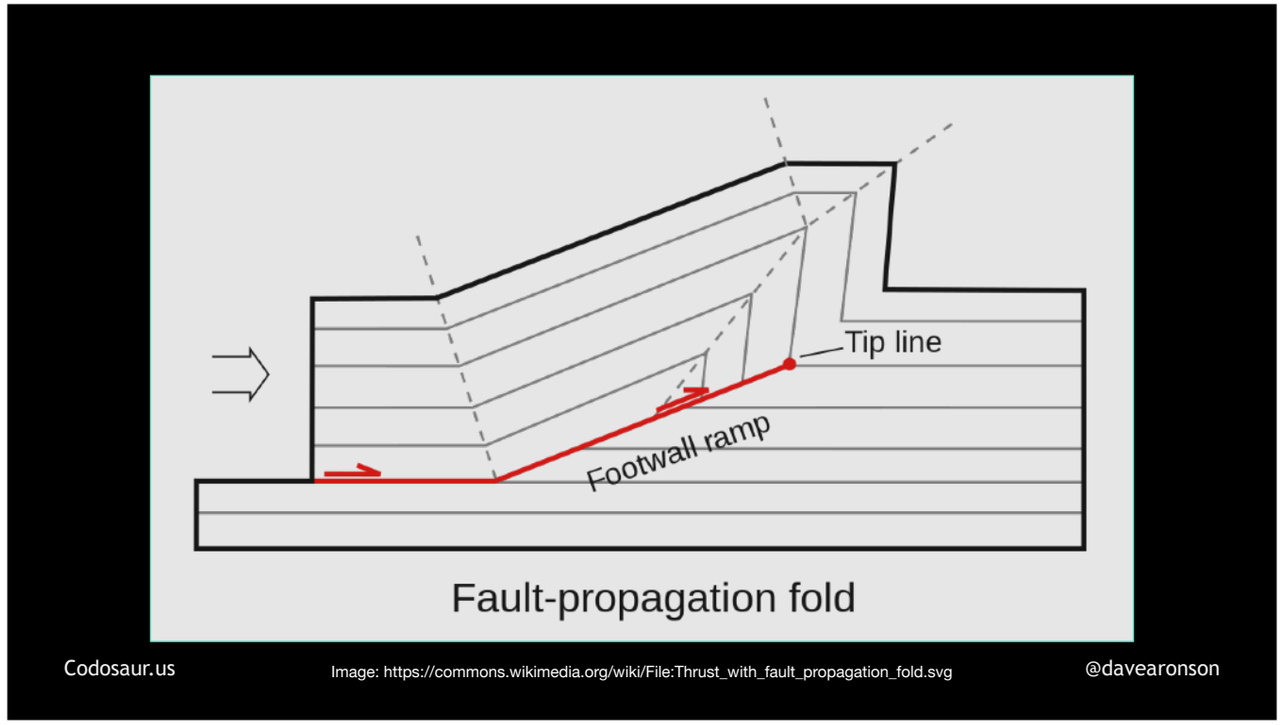


Codosaur.us

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

@davearonson

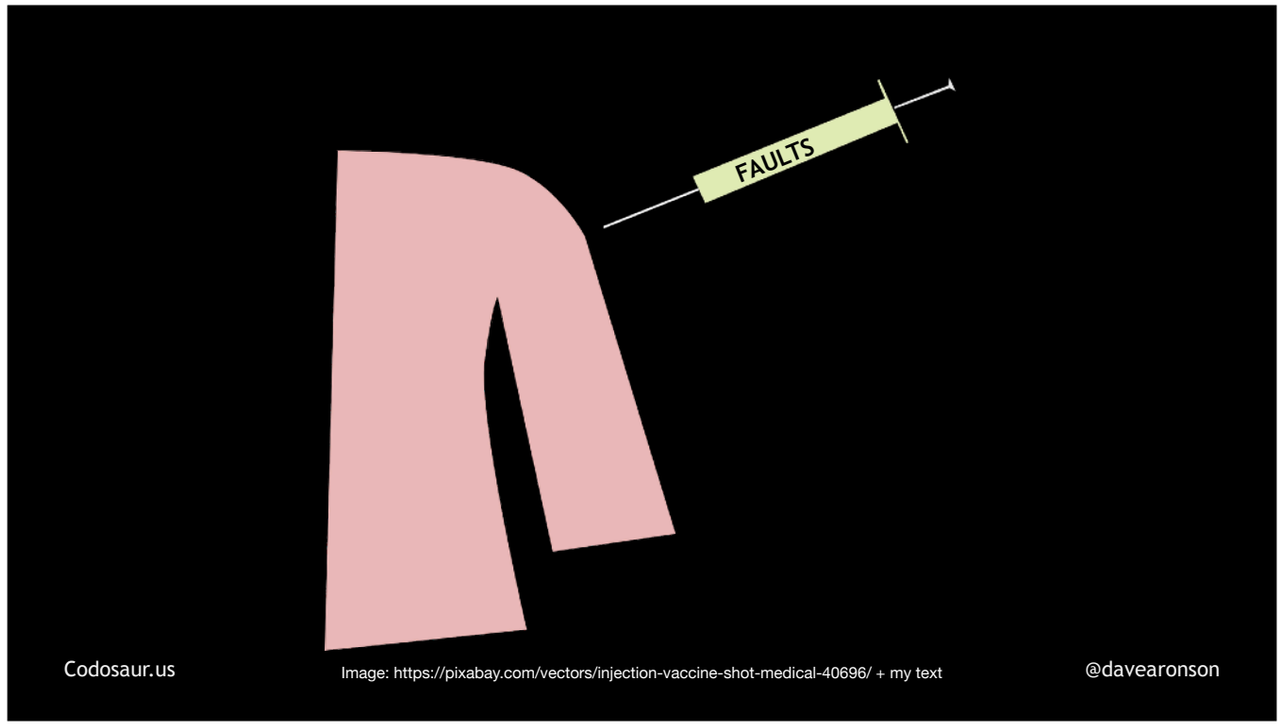
. . . *mutates* copies of our code, hence the name. It does this to create test failures, also known as . . .



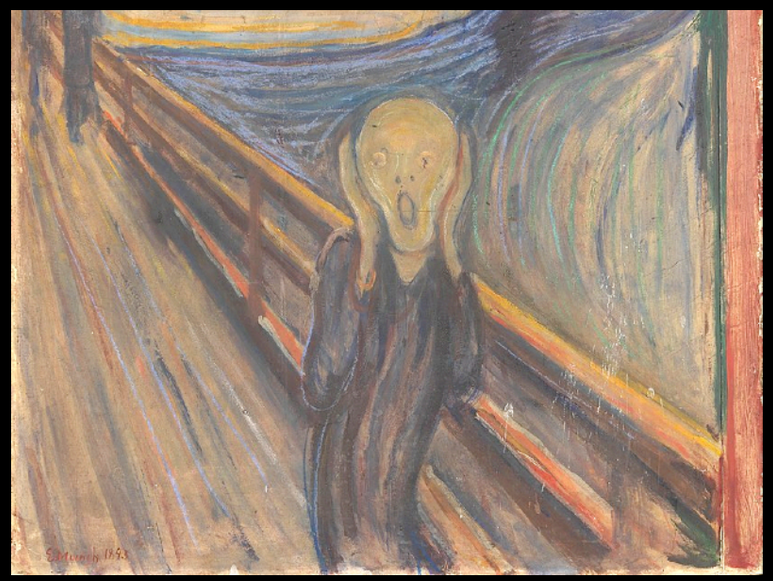
. . . faults. So, mutation testing is a *fault-based* testing technique, so it's related to something you might already know about:

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

. . . upside down from that. Chaos Monkey is best known for . . .



. . . injecting faults, like latency or jitter, 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.

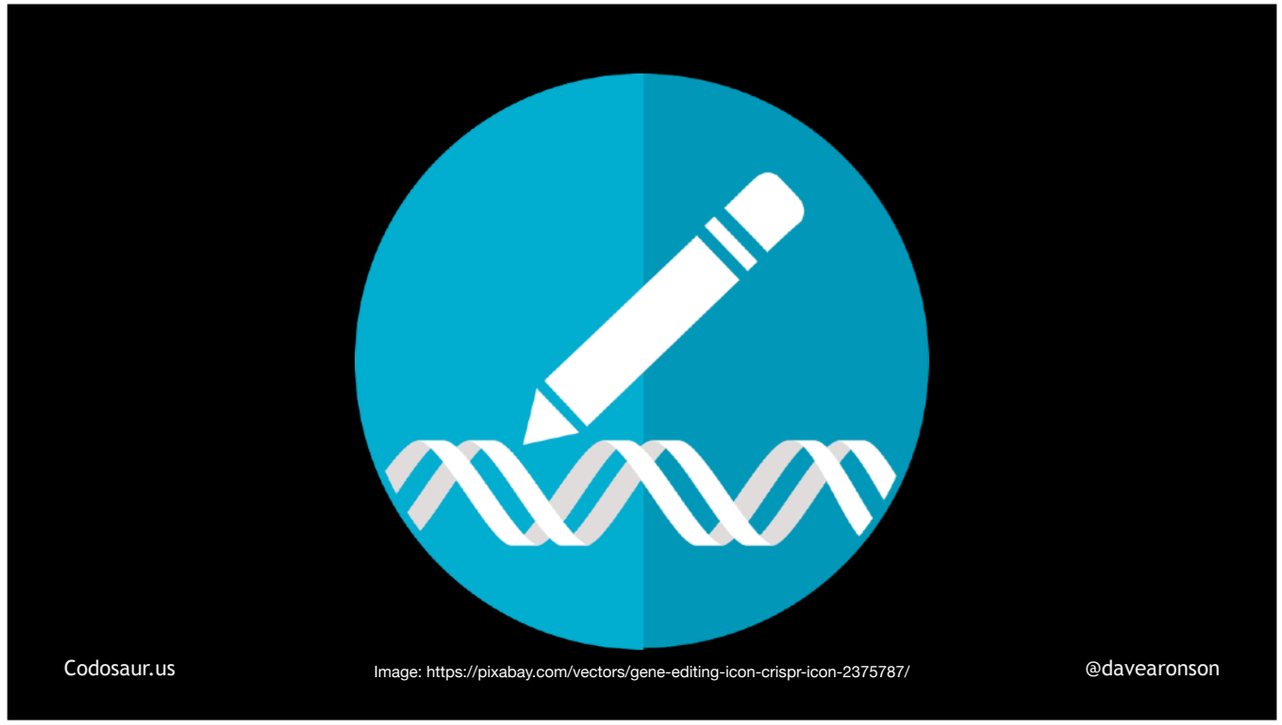
If all still goes well, in the sense that Netflix's customers don't notice, and their metrics still look good, then Netflix knows that their error recovery is working fine. Mutation testing, however, injects semantic . . .

Codosaur.us

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

@davearonson

. . . *changes*, or *mutations*. It doesn't *know* whether these mutations will cause faults or not. We 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 means that . . .

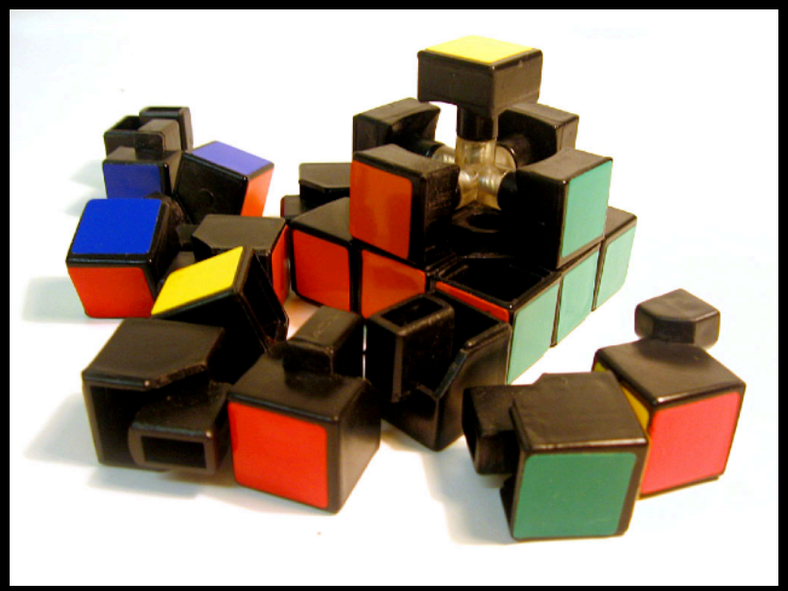
. . . there *is* a problem! Remember, each change should make *at least* one test *fail*.

Mutation testing has also been compared to . . .

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

. . . not random.

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

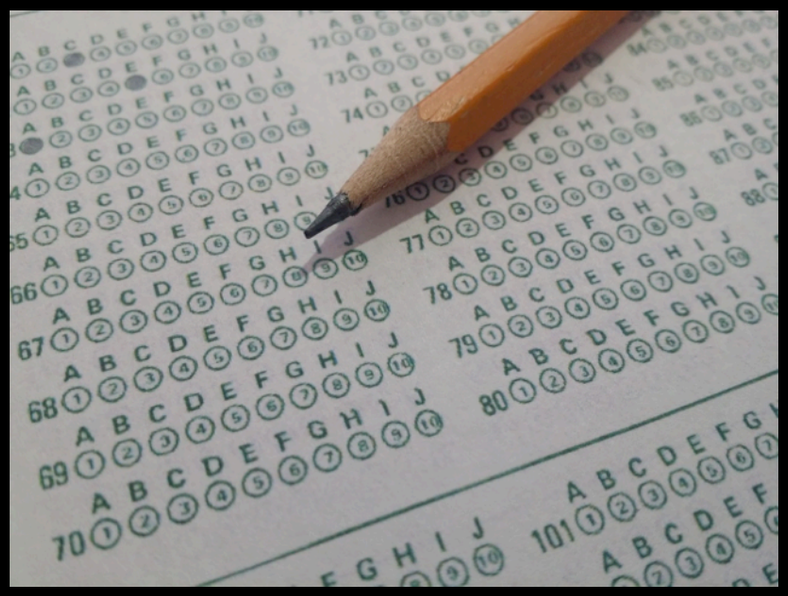


Codosaur.us

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

@davearonson

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



Codosaur.us

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

@davearonson

. . . the *tests* that *cover* that function, and then . . .

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

Codosaur.us

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

@davearonson

. . . one mutant, with *that one 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

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 . . .

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

. . . against the *current mutant* instead 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

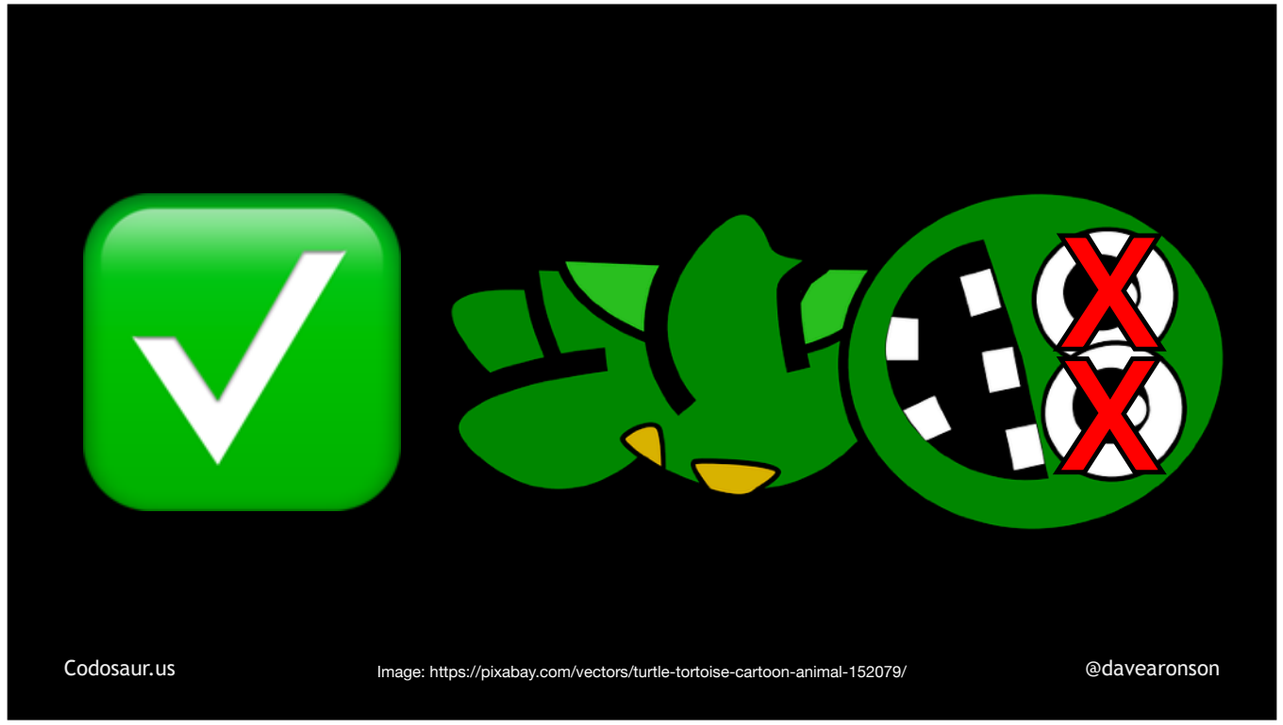
. . . *fails*, this is called . . .

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 mutation that the tool injected to *create* this mutant, made a noticeable difference in the function's behavior, *and* that our *test* suite is *strict* enough that at least one test *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. But 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

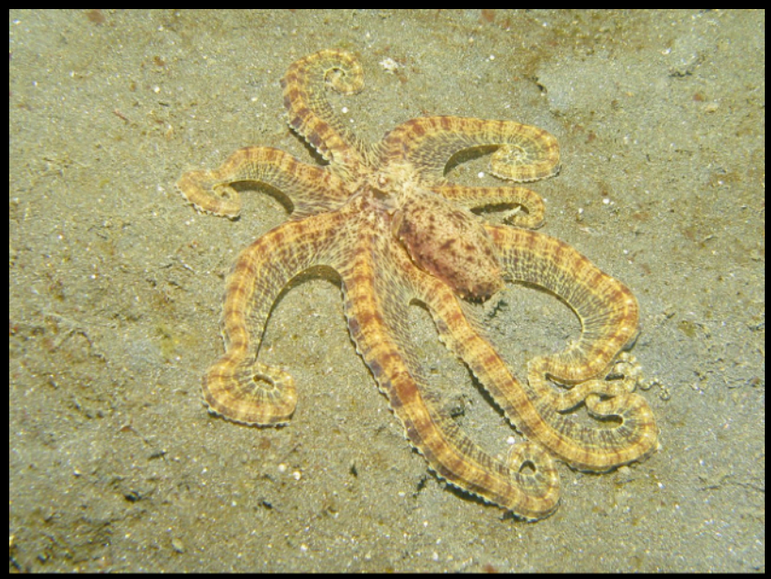
. . . lets all the tests pass, then the mutant is said to have . . .

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 it 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 . . .

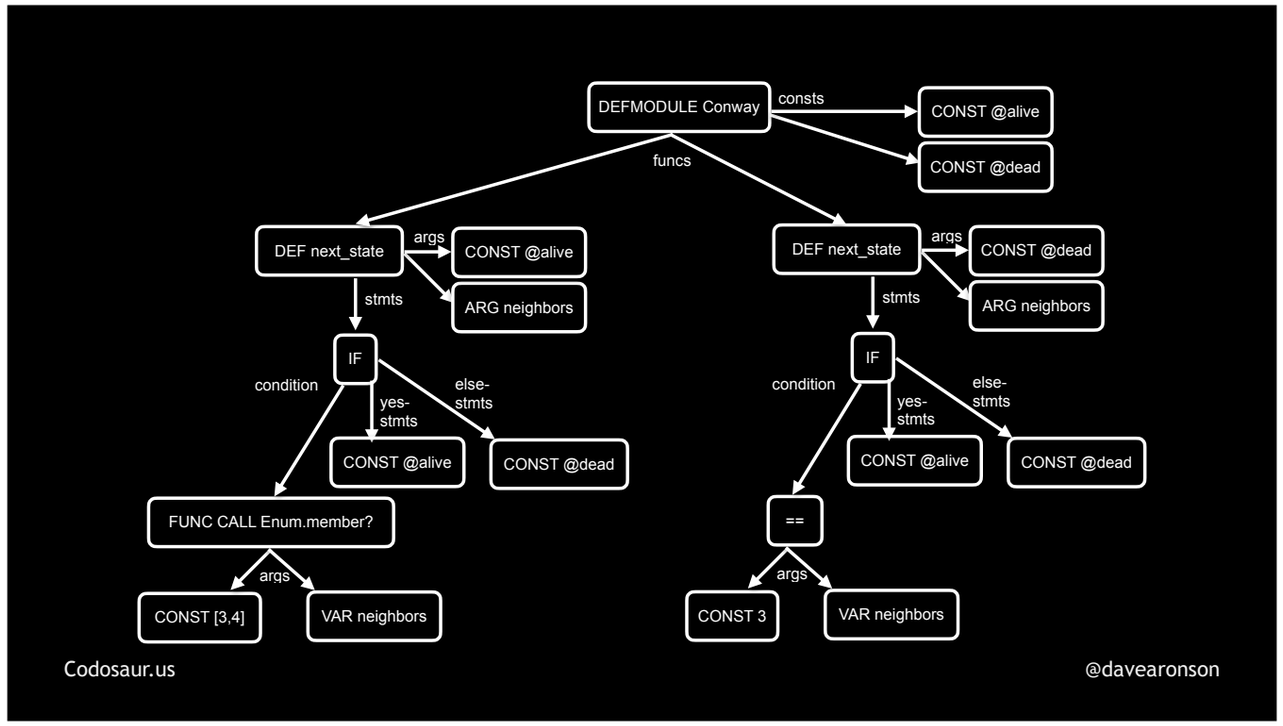
```
defmodule Conway do
  @alive "*"
  @dead  " "

  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, so that this code becomes . . .



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

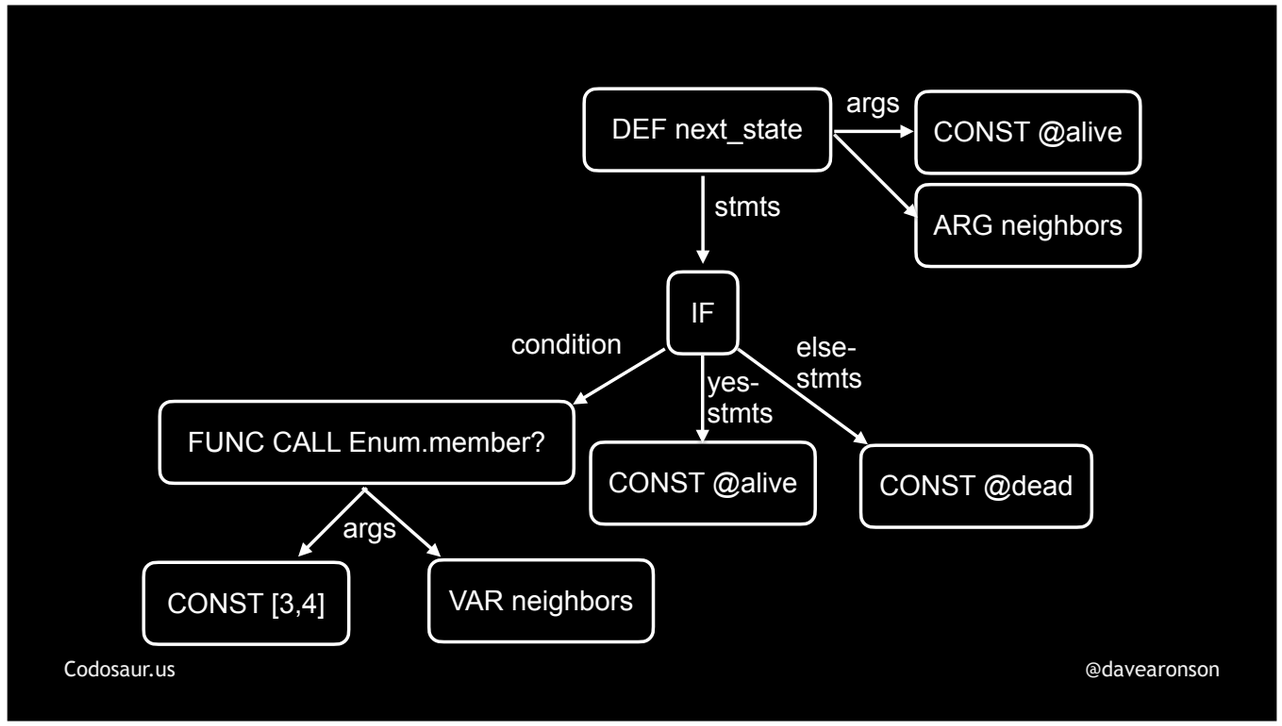
. . . traverses the tree, looking for sub-trees that represent each function. After finding one, it looks for the *tests* . . . and I'm going to completely gloss over how. Then it makes the mutants. To make them *from* an AST subtree, it . . .

Codosaur.us

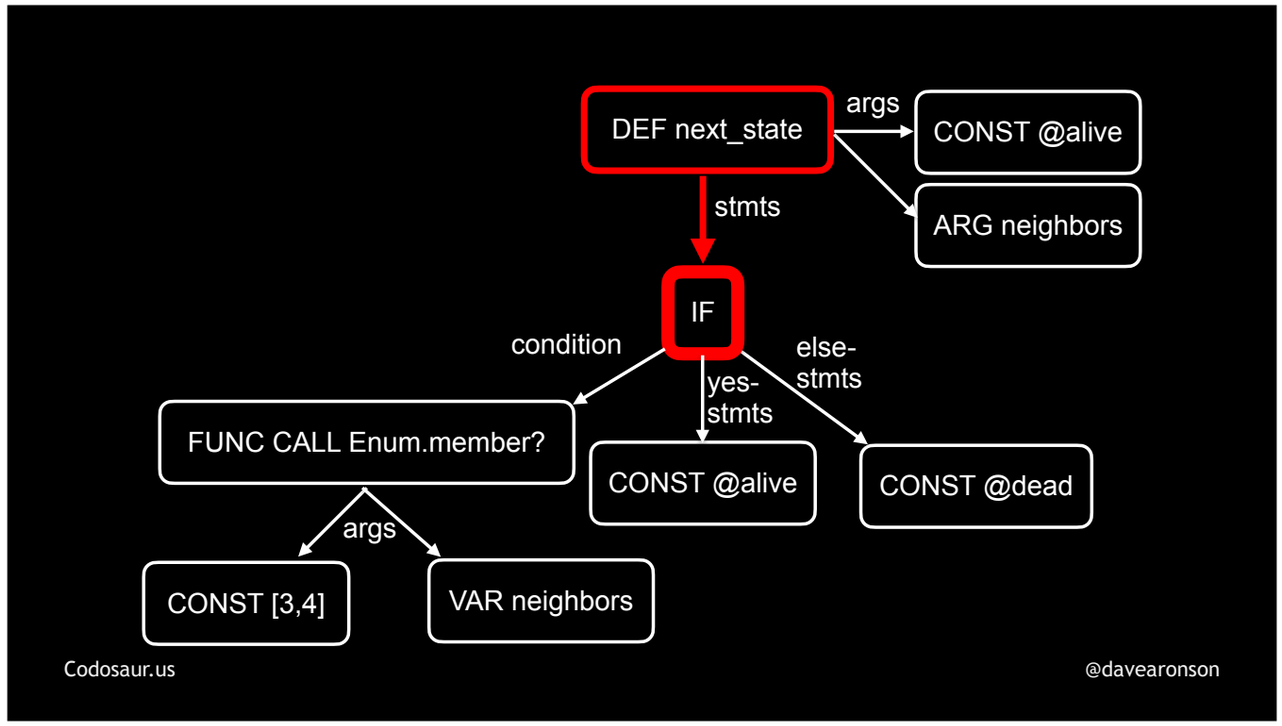
Image: https://pxhere.com/en/photo/1230969

@davearonson

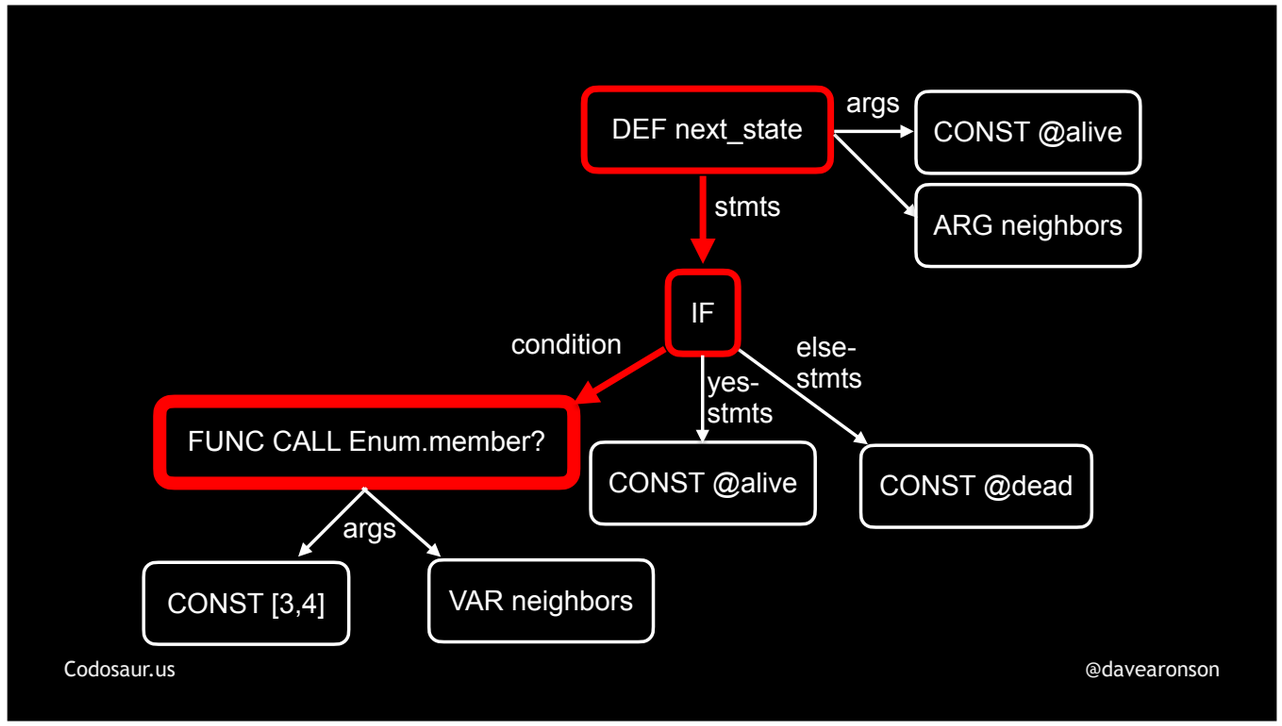
. . . traverses that subtree, just like it did to the whole thing, but instead of looking for even *smaller* subtrees to *extract*, it looks for *nodes* where it can *change* something. Each time it finds one, then for each way it can change that node, it makes one copy of the function's AST subtree,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



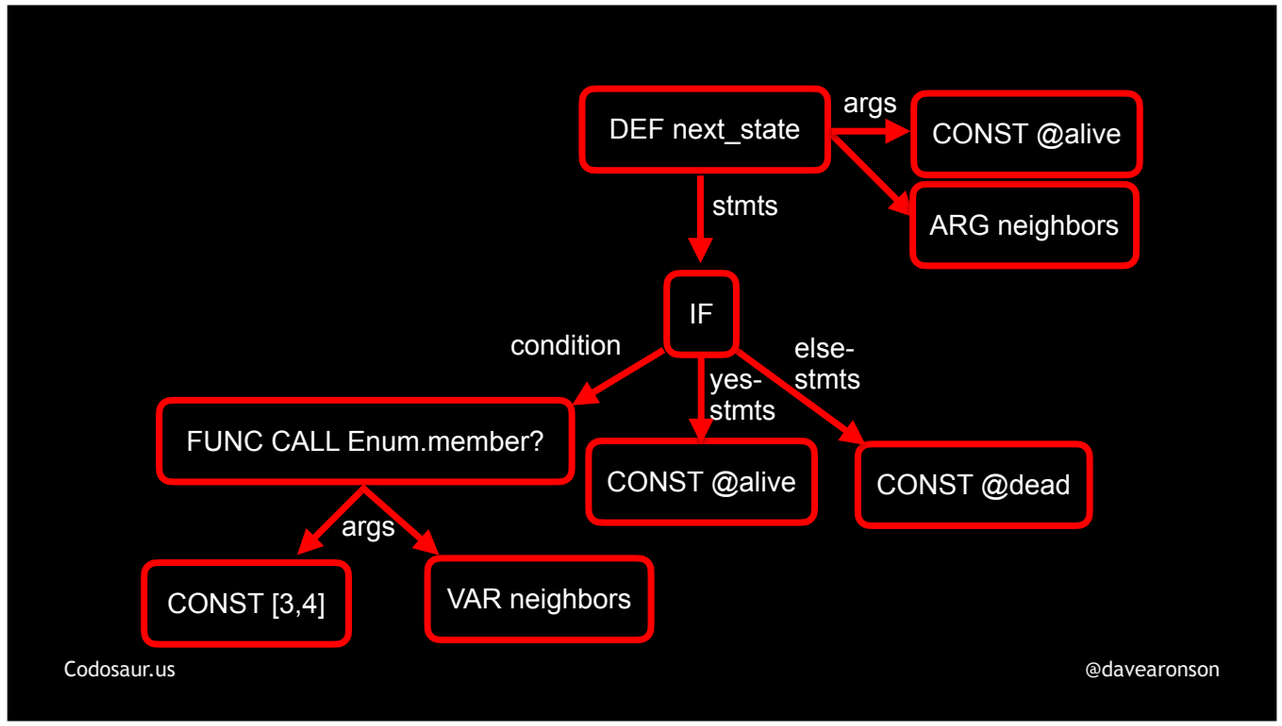
. . . this function subtree, and has gotten down to . . .



. . . this if statement. For each way the tool could change that node, it would make a fresh copy, of this whole subtree, with that one mutation. After it's done making as many mutants as it can by changing *that* node, it would continue . . .

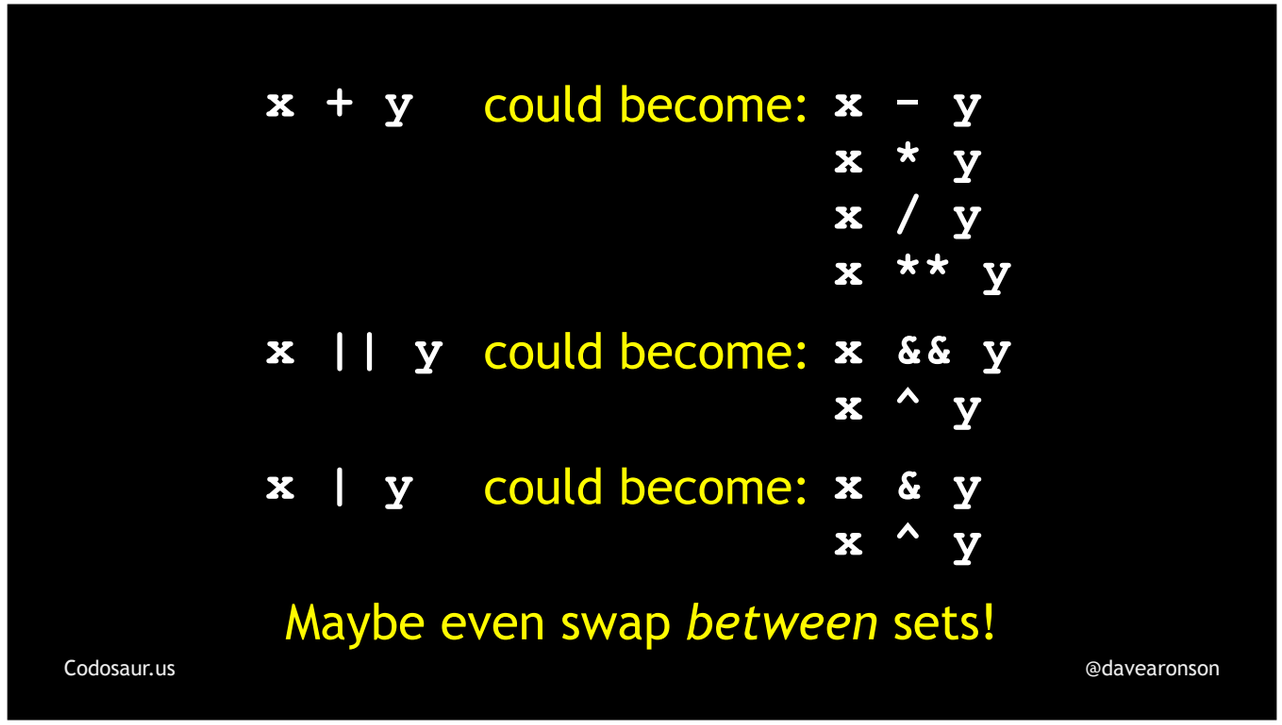


. . . on to the next node, do likewise, and so on, until it has . . .



. . . 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, even one from a different category if allowed, so x *times* y could become x *and* y or 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, 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*.

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

could become:

```
foo(z)
```


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

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

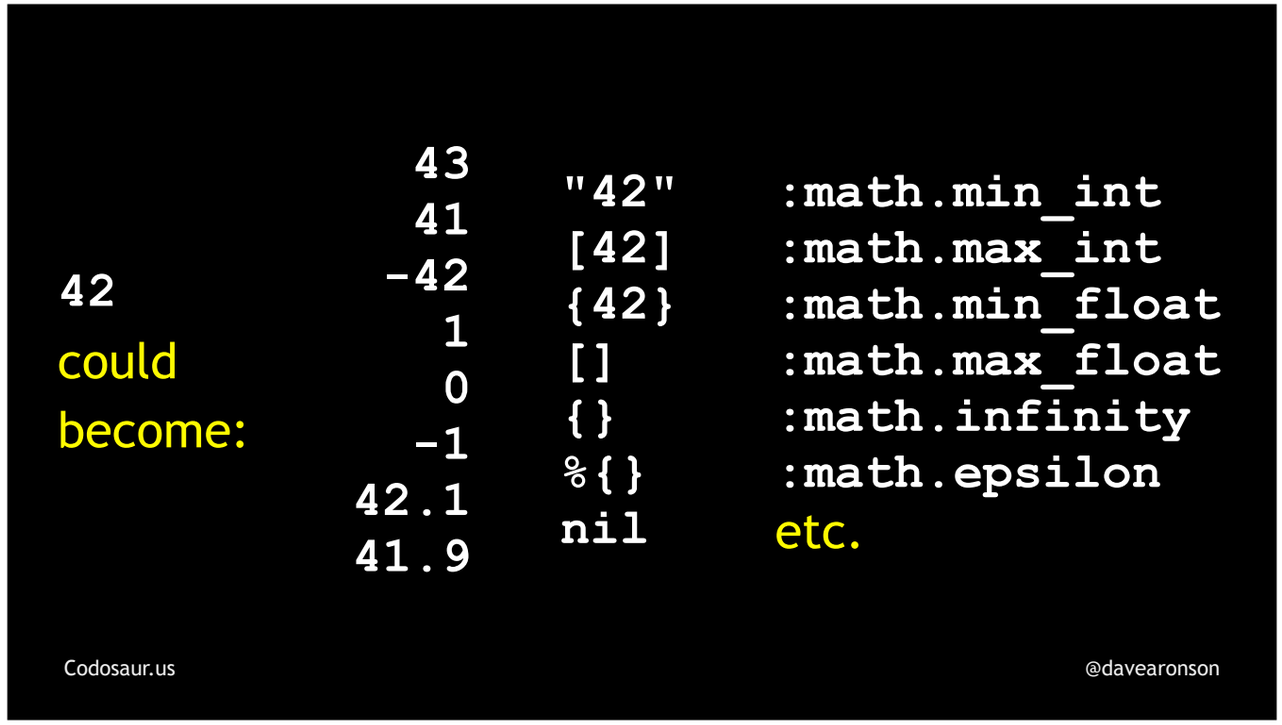
could become:

```
foo(z)
```

Codosaur.us

@davearonson

It can remove a looping condition, so that something that might be skipped, done once, or done *multiple* times, is always done once.



It could change a value to some other value, such as changing 42 to any of these. It could even change it to an 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!

Now let’s peel back one last layer, 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. We won't stop to inspect this, but the final slide has the URL for all the slides, so you can take a picture of that and ponder this at your leisure.

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 just makes a good simple example.

Think about what a mutant made from this might *return*.

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*.

Now suppose we had only one test . . .

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

Codosaur.us

@davearonson

. . . like so. This is a poor test, but even so, it would kill *most* of those mutants, the ones shown . . .

```
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

. . . here in crossed-out green. But . . .

. . . addition, multiplication, and exponentiation in the reverse order, all get us the correct answer. So, mutants based on *these* mutations would *"survive"* this test. A report might present them to us like this:

```
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

. . . though the format will vary widely depending which tool we're using. So what is this set of surviving mutants trying to tell us? Our function is just one line so redundant or unreachable code is very unlikely, so it's probably a test gap. The usual cause of those is that . . .

```
mutant_func(x, y)
==
original_func(y, x)
```

Codosaur.us

@davearonson

. . . the mutant returns the same result as the original function — or has the same side effect, if our tests are looking at that. 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
```

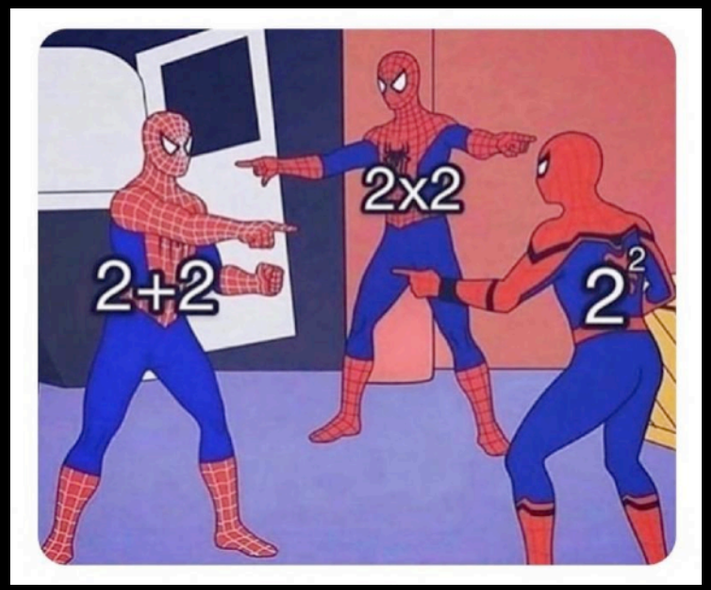
our test:

```
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. (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:

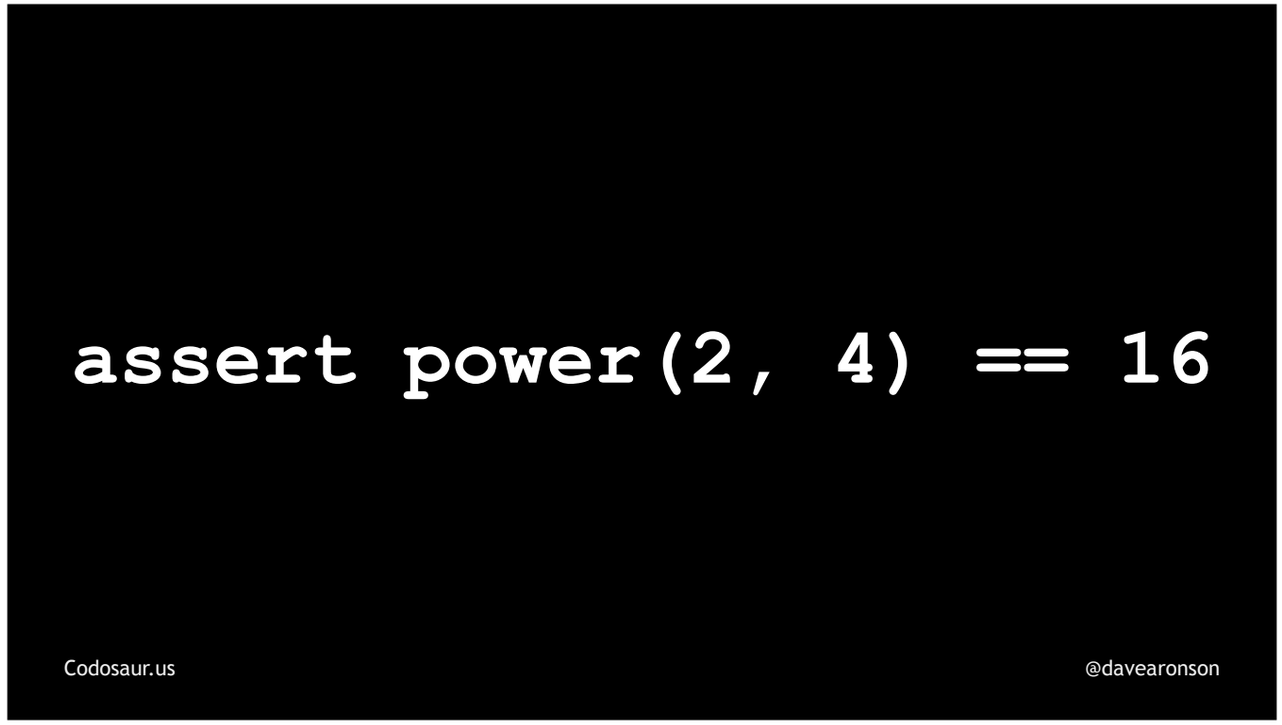
```
43 -    x ** y
43 +    x + y
```

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? 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 existing test to something like . . .

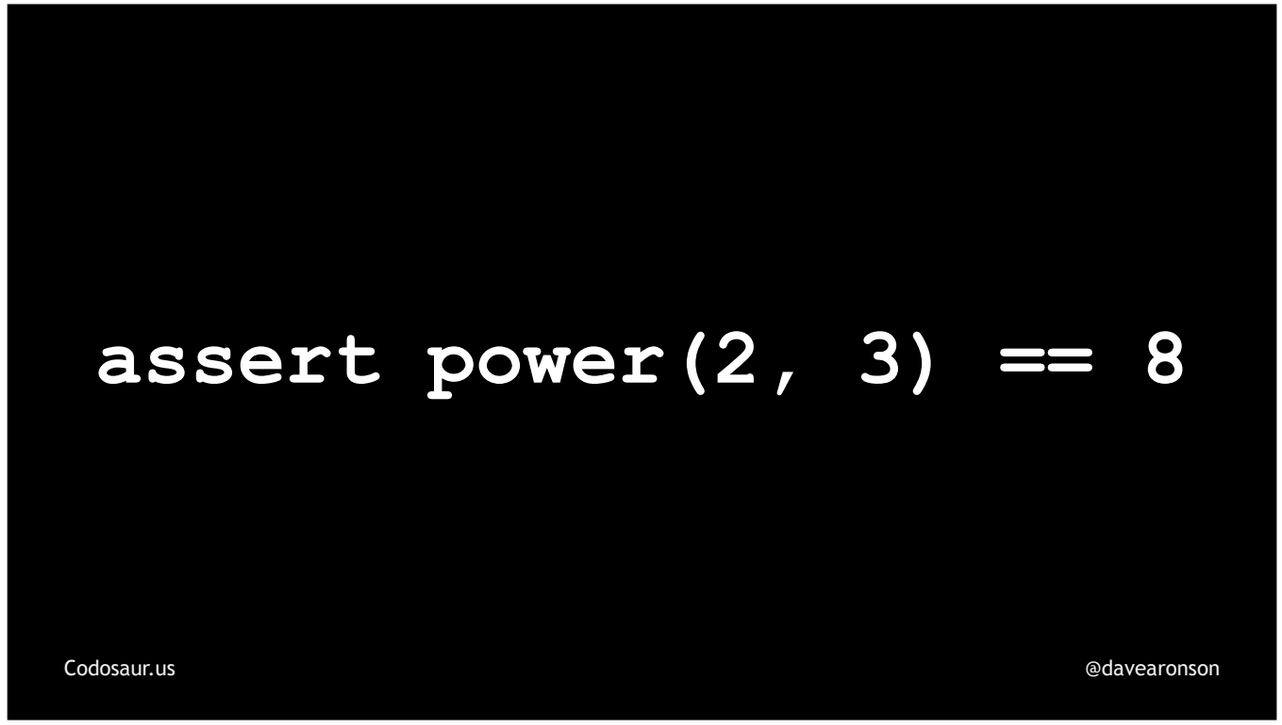


. . . asserting that two to the *fourth* power is *sixteen*. All the mutants that our original test killed, *this would still kill*. Also, two *plus* four is six, not six*teen*, **so this kills the plus mutant**. Even better, two *times* four is eight, *also* not sixteen, so this would kill the "times" mutant as well. You'll often find that killing one mutant, kills others as well.

But, . . .

. . . the pair of exponentiation operand swapping mutants survive! But that’s okay, we can . . .

. . . attack them separately, no need to kill all mutants in one shot and be some kind of superhero about it. We can add or adjust a test like . . .



. . . asserting that two to the *third* power is *eight*. Three squared is nine, not eight, so this kills the swapping mutants. Better yet, two *plus* three is five, two *times* three is six, and both of those are not eight, so the "plus" and "times" mutants *stay* dead, 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; 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, 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_now = send_bytes(buf + sent,
                          len - sent)
    sent += sent_now
  end
  sent
end
```

Codosaur.us @davearonson

. . . like so. This function, send_*message*, sends as much data as send_*bytes* can handle in one chunk, looping to pick up where it left off, until the message is all sent.

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

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

Codosaur.us

@davearonson

. . . this, removing a looping condition.

Now suppose that this mutant does indeed survive our test suite. Even without seeing the tests, what does that tell us? (PAUSE!)

If a mutant that only goes through . . .

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

Codosaur.us @davearonson

. . . that loop once, lets all tests pass, that means that our *tests* are only making our code go through that loop once. So what does that mean? (PAUSE!) You'll find that interpreting mutants involves a lot of asking *"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, starting with the most *likely* one, which is that we just didn't bother. 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 shot, is 10,000 bytes, but . . .

in module Network:

```
max_chunk_size = 10_000
```

in test_send_message:

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


. . . we're only testing with *three* bytes. (PAUSE!)

The obvious fix is to use a message larger than our maximum chunk size. We can make 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.

But . . . let's look at another possible cause. Maybe we *did* test with the *largest* permissible message size. 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, 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!)

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, and everything that's only there to support the looping, we wind up with . . .

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

Codosaur.us @davearonson

. . . this. (PAUSE!) Now it’s pretty clear: the *entire* send_message *function* may well be *redundant*, so we can just use send_*bytes directly!* Fortunately, when it's this kind of problem, with unreachable or redundant code, the solution is clear and easy, just rip it out! This will also make our code more *maintainable*, by getting rid of useless cruft.

Now I'd like to address some . . .

. . . common 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 . . .

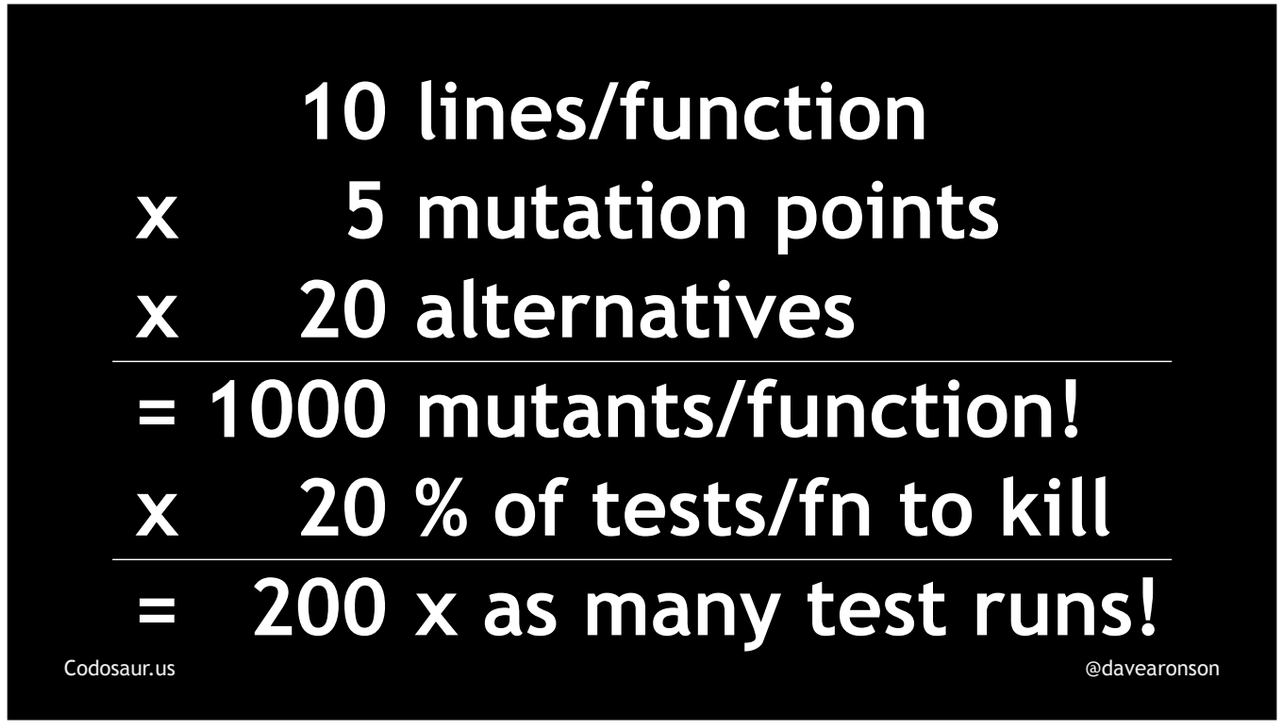
Codosaur.us

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

@davearonson

. . . long history. It was first proposed in 1971, in a term paper by Richard Lipton, titled “Fault Diagnosis of Computer Programs”. The first *tool* appeared in 1980, as part of Timothy Budd's PhD work. But it wasn’t *practical* on normal developer-grade computers, until about 2000 or so, due to faster CPU cores, multi-core CPUs, faster, larger, cheaper memory, and so on.

But *why* does it need so much resources? With some reasonable assumptions, we wind up with mutation testing needing about . . .



. . . *two hundred times* the test runs, as just running the test suite. If our test suite normally takes a zippy ten seconds, with these assumptions, mutation testing will take *over 33 minutes!*

To sum up, 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, once you understand the concept, but it's . . .

😀 Checks that code is meaningful

😀 Checks that tests are strict

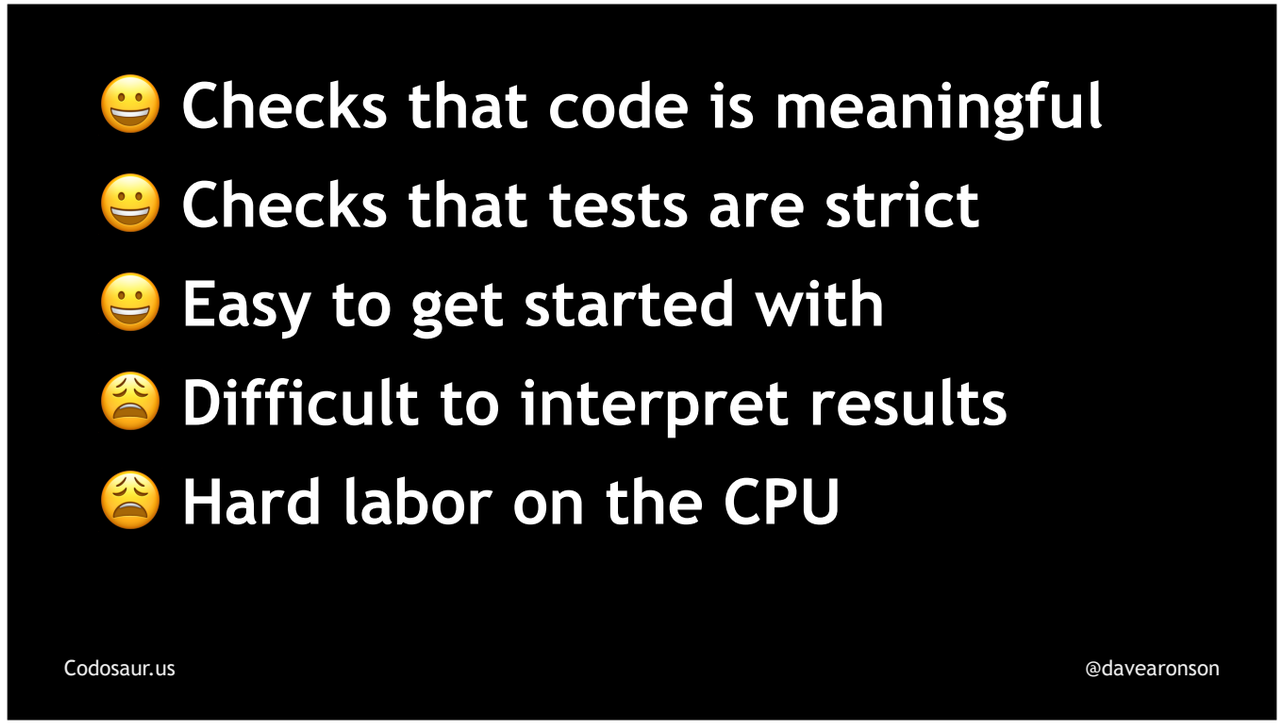
😀 Easy to get started with

😩 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’s not a good fit for our current projects, though, 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.

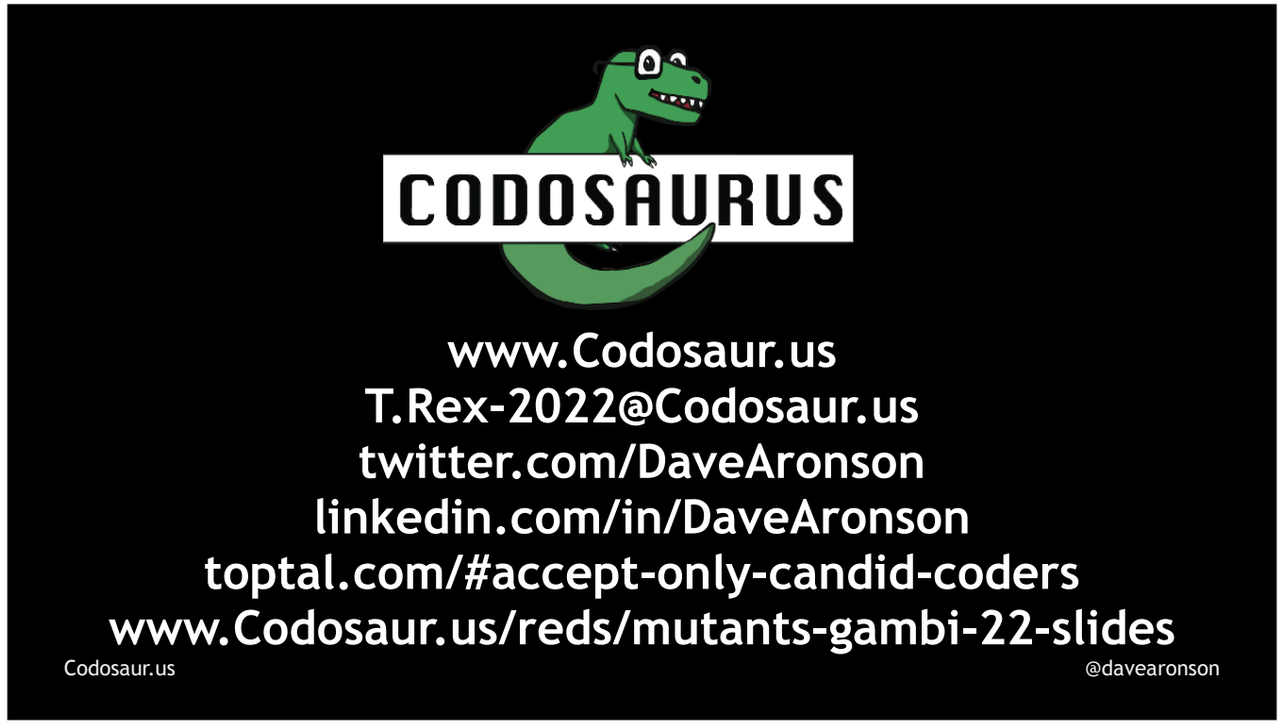
If you'd like to try it for yourself . . .

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:	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~~
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’s a list of tools for many languages and platforms. I know this is too small to read, but once again, the final slide contains the URL for all the slides. But before we get to that, 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. (Please use that referral link if you want to hire us or join us. The anchor part tells them it’s me, and we’ll both get some bounty.)

And now . . .



. . . here’s that promised final slide with a bunch of URLs, and it’s your turn! Any questions?