Kill All Mutants!

(Intro to Mutation Testing)

by Dave Aronson

T.Rex-2020@Codosaur.us

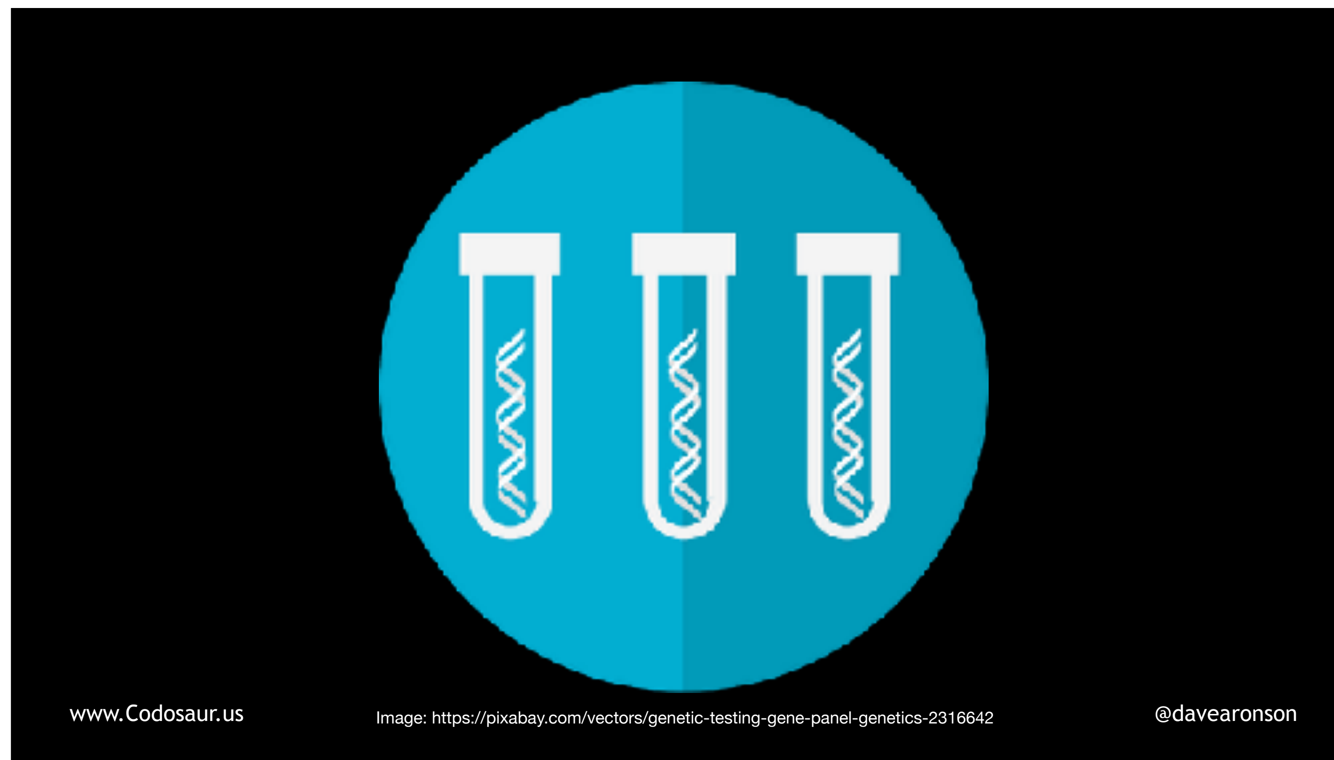
Slides: bit.ly/kill-mutants-JSConfHI-2020

www.Codosaur.us

@davearonson

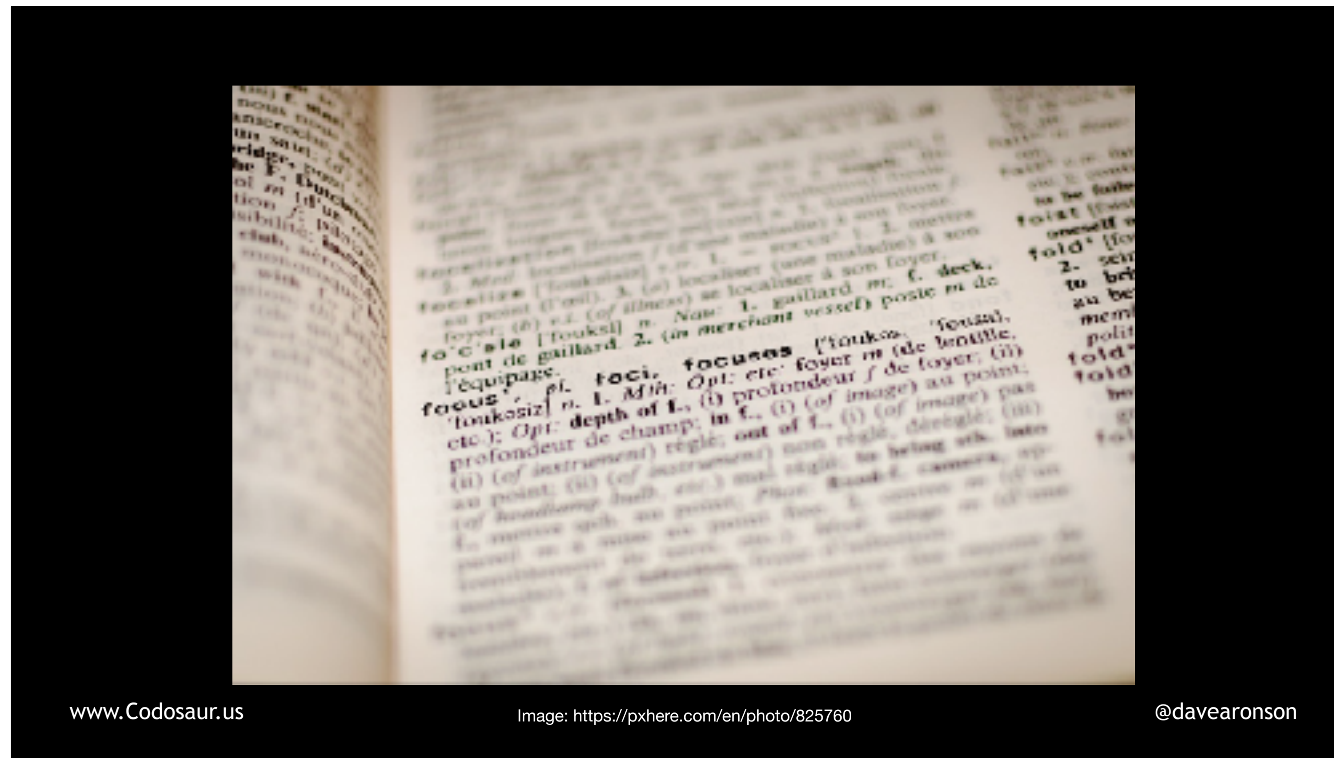
Aloha, Hawai'i! I'm Dave Aronson, the T. Rex of Codosaurus, LLC, and I'm here to teach you to KILL MUTANTS!

So what are those? In *our* universe, that of software development, not comic books, they're something used in Mutation Testing. So what on Infinite Earths is . . .



. . . mutation testing? You might think it's about testing the mutations used in genetic algorithms, but no. It's a way to test our code, *and our unit test suite*, by *using* mutations. Its most *unusual* benefit is to help ensure that our unit tests are *strict*, by . . .

. . . finding the gaps in our unit test suites, that let our code get away with unwanted behavior. Lack of strictness usually comes from lack of tests, poorly written tests, or poorly *maintained* tests, that didn't keep pace with changes in the code. It also helps ensure that our code is . . .



. . . *meaningful*, meaning that that any change to the code, will produce a noticeable change in its behavior. Lack of *meaning* usually comes from code being redundant, unreachable, or otherwise without any real effect.

Mutation testing . . .

. . . puts these two together, by checking that changing the code does indeed change its behavior, *and* that the unit test suite does indeed notice that change, and fail. Not all tests have to fail, but each change should make at least one test fail.

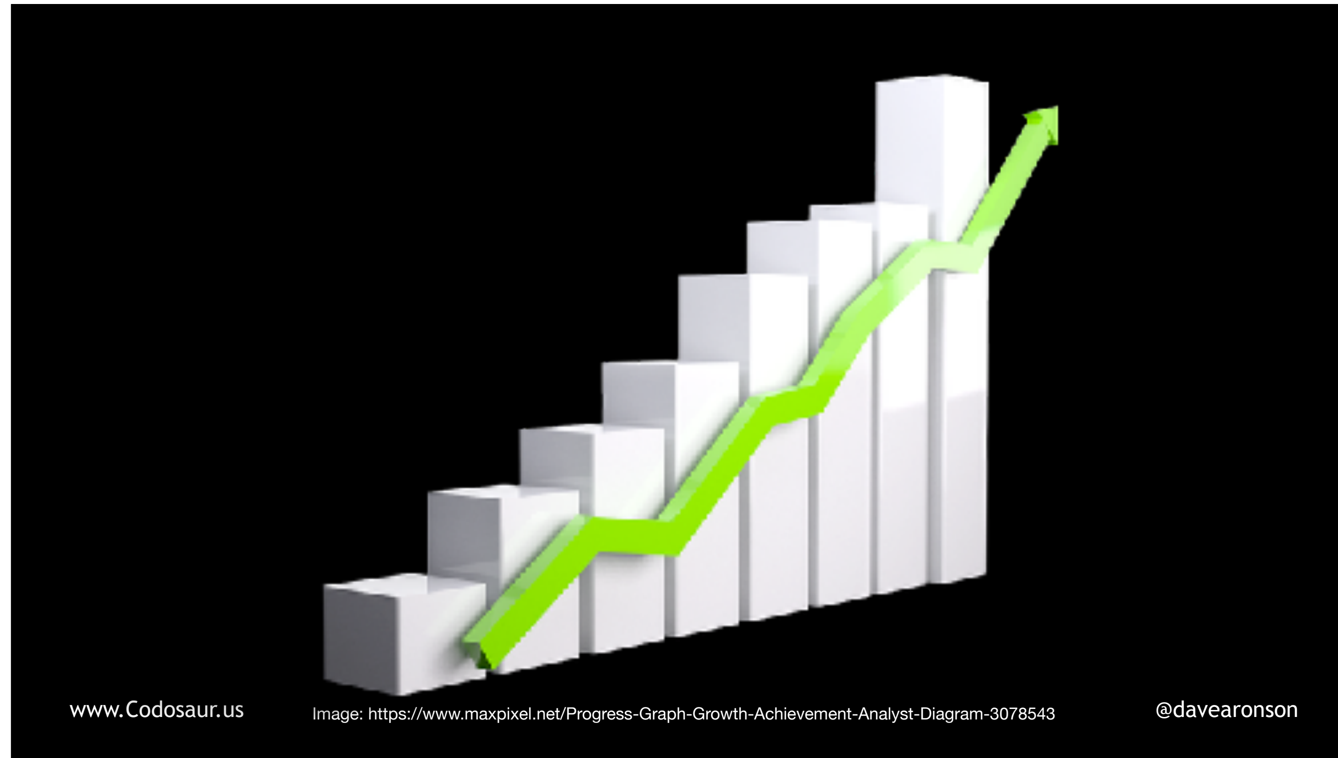
However, there are some drawbacks. As Fred Brooks told us back in '86, there's no . . .

. . . silver bullet! Besides, they're for killing . . .

. . . werewolves, not mutants!

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

. . . hard labor for the CPU, so it's slow. We surely won’t mutation-test our whole codebase on every save! Maybe over a lunch break for a small system, or over a night or weekend for a larger one. Fortunately, most tools include an . . .

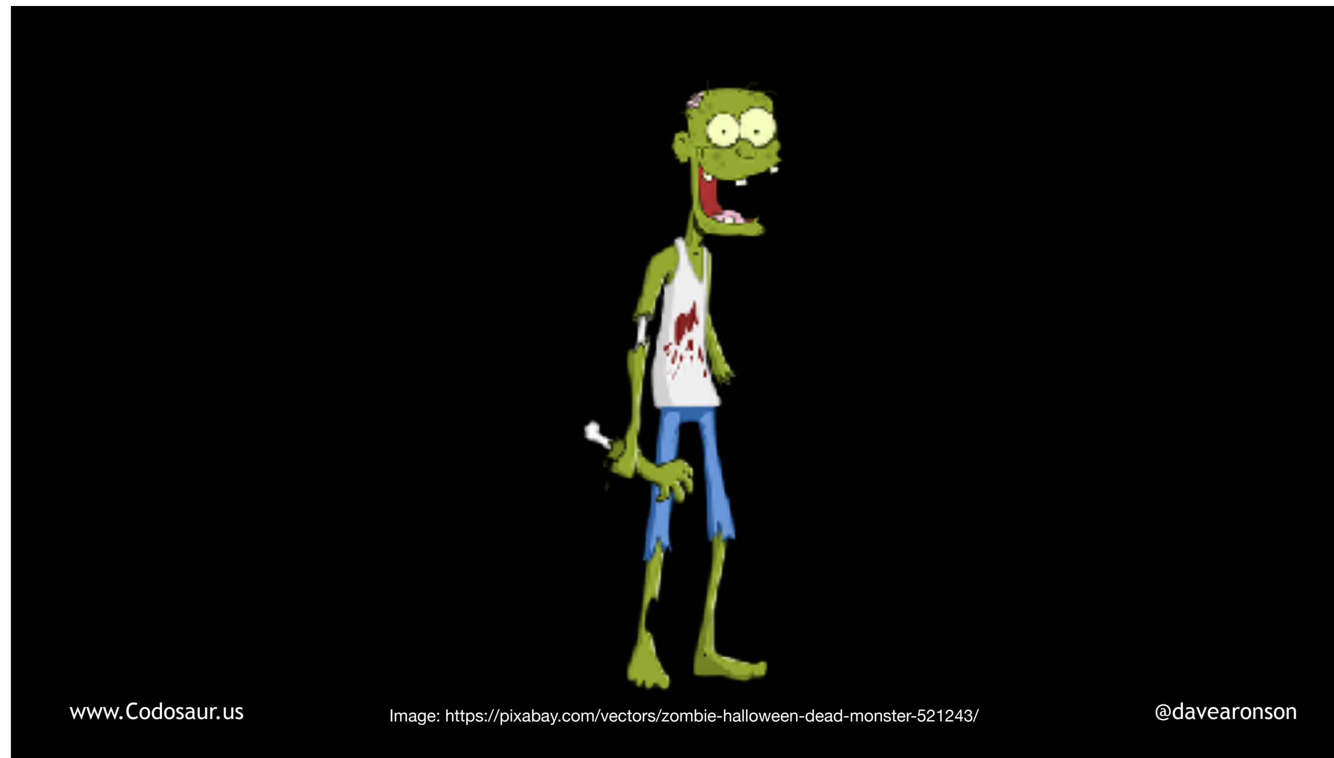


. . . incremental mode, so we can test only what has changed since last time. That, *maybe* we can do on each save, if we save often. Also, its CPU-intensive nature can really . . .

. . . run up our bills on cloud platforms such as AWS or Azure! (Or aZURE, or whatever.)

It's also . . .

. . . not a beginner-friendly technique! It tells us that some particular change to the code made no difference to the tests, but it takes a lot of interpretation to figure out what a mutant is trying to tell us. Their accent is vurrah strayinge, and they're almost as incoherent as . . .



. . . zombies, but with a much bigger vocabulary; 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 *how!* Even worse, sometimes it's a . . .

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

Now that we've heard the main pros and cons, what does mutation testing actually *do?* It . . .

Point Mutations

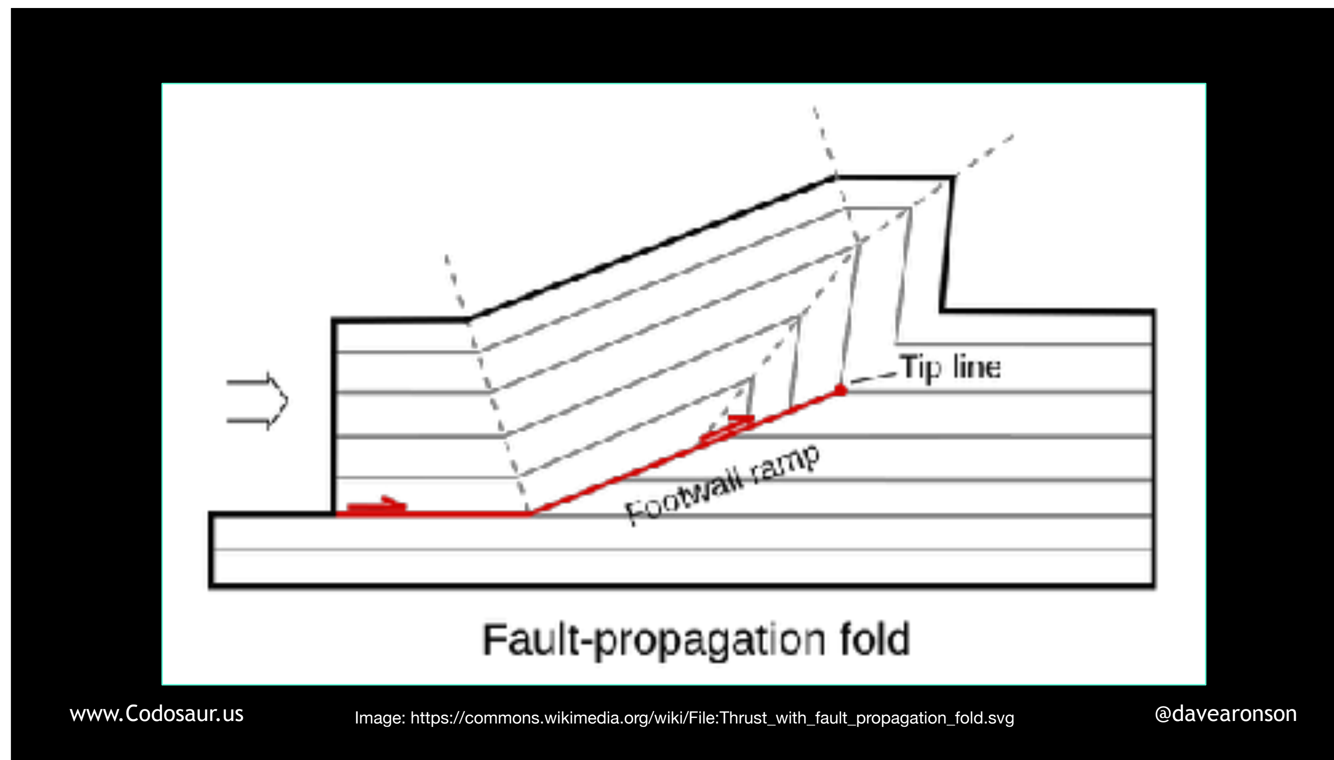
DNA
mRNA
Normal
Missense
Frameshift insertion
Frameshift deletion
Nonsense

www.Codosaur.us

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

@davearonson

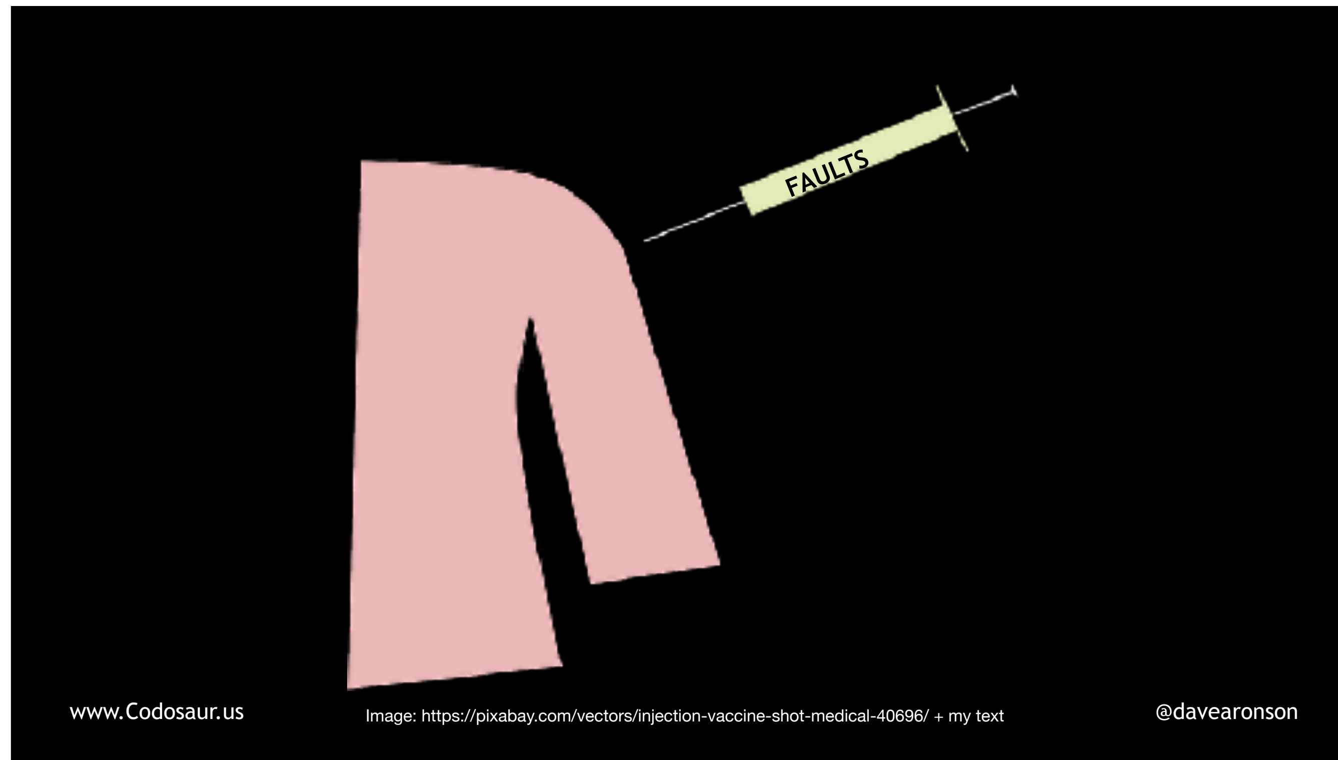
. . . *mutates* copies of our code, hence the name, trying to create *test failures*, also known as . . .



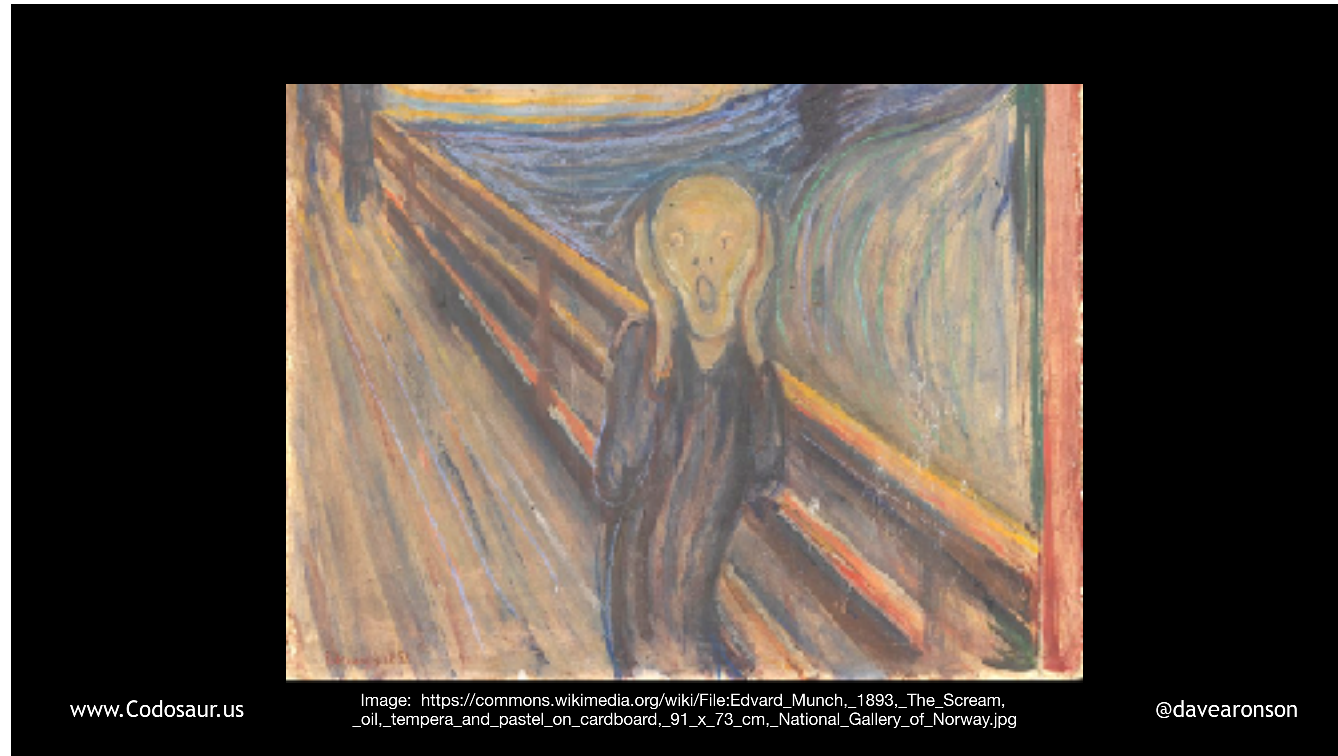
. . . faults. So, mutation testing is a *"fault-based"* testing technique. This means it is related to something you might already know about:

. . . Chaos Monkey, from Netflix. Just like Chaos Monkey helps Netflix discover error recovery flaws, mutation testing helps us discover test flaws and code flaws. But the way mutation testing does it, is sort of (CLICK!) upside down from what Chaos Monkey does. Chaos Monkey is best known for . . .

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



. . . injecting faults, into Netflix's production network. (CHANGE SLIDE IMMEDIATELY!)



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

. . . *changes*, not necessarily *problems*. It doesn't *know* if these changes will create *faults*. We hope they all will, but that's up to the test suite. It injects them *into* . . .

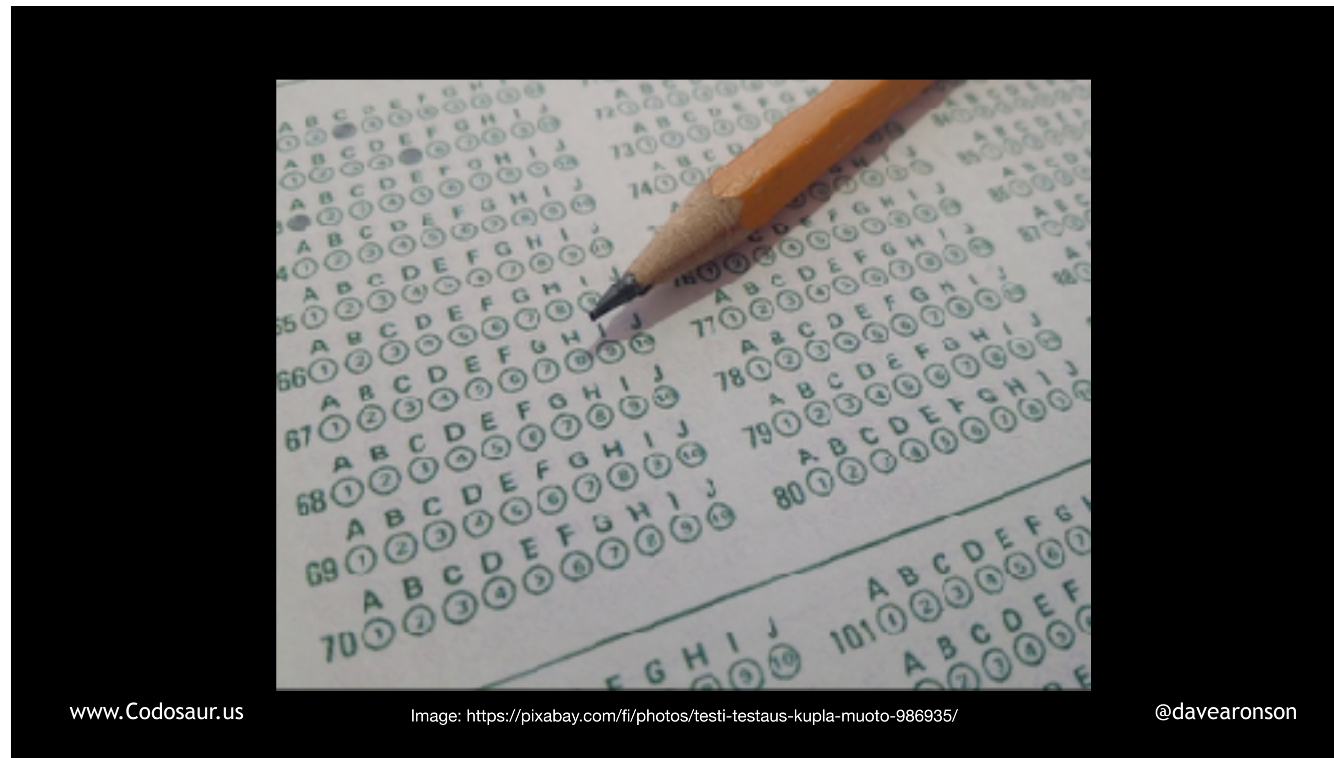
. . . copies of our code, not our actual network. It does its work in our . . .

. . . *test* environment, not production. (Whew!) And if our tests 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*.

But *how* does it do all that? Let's peel back . . .

. . . one layer of the onion, and take a high-level look. First, our chosen tool . . .

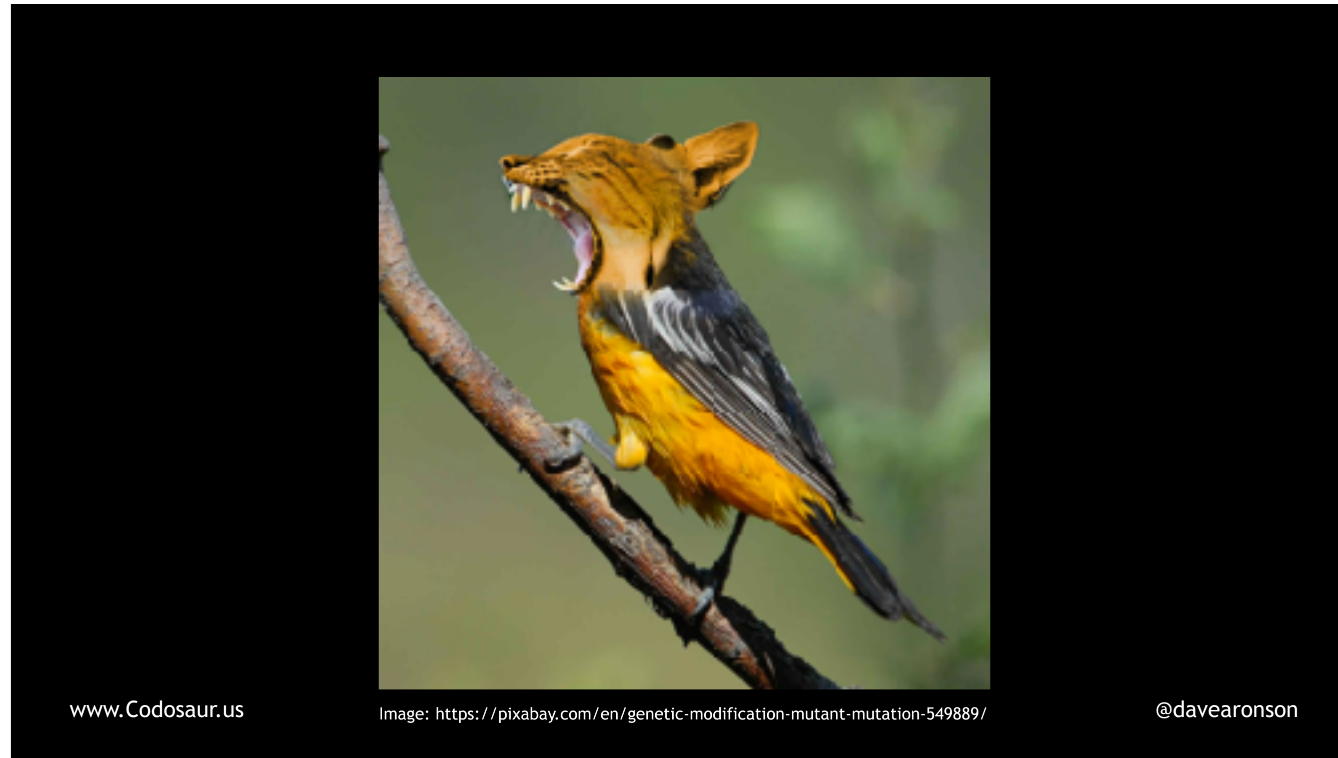
. . . breaks our code apart into pieces to test, usually our functions. Then, for each function, it tries to find . . .



. . . that function's *tests*. If the tool can't find any tests, most will skip this function and probably warn us. Some will use the whole unit test suite, but that's inefficient and leads to even more false alarms.

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

. . . makes the mutants. To do that, it inspects this function to see how it can be changed. For each way, the tool makes . . .

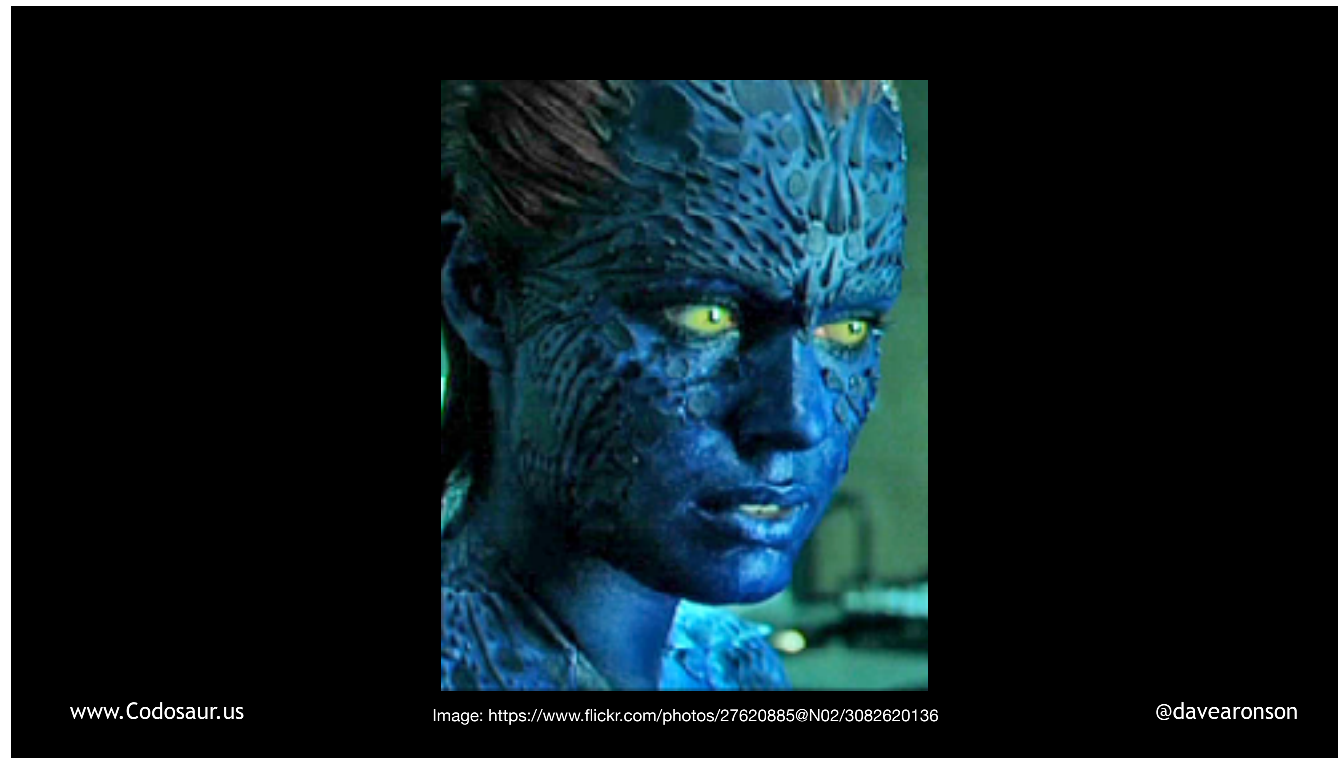


. . . one mutant, with that *one change*. Once our tool is done creating all the mutants it can for a given function, it iterates over . . .

. . . the list. And now we get to the heart of the concept. For each mutant, (PAUSE; SLOW!) derived from a given function, the tool runs the function's unit tests, but it runs them using the *current mutant* in place of the original function. (PAUSE!) If a test *fails*, this is called . . .

. . . “killing the mutant”, and it’s a *good* thing. It means that our code is *meaningful* enough that the tiny change that the tool made, to create the mutant, actually made a difference in the function's behavior, *and* that our *test suite* is *strict* enough to *notice* that difference, and fail. Then the tool will stop running tests against that mutant, and move on to the next one. Once this mutant has made *one* test fail, we don't care how many more it *could* make fail. Like so much in computers, we only care about ones and zeroes.

But if a mutant lets all those unit tests *pass*, that means it has the . . .



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

Now let's peel back another layer, and look at some *technical details* of how this works. First, our tool . . .

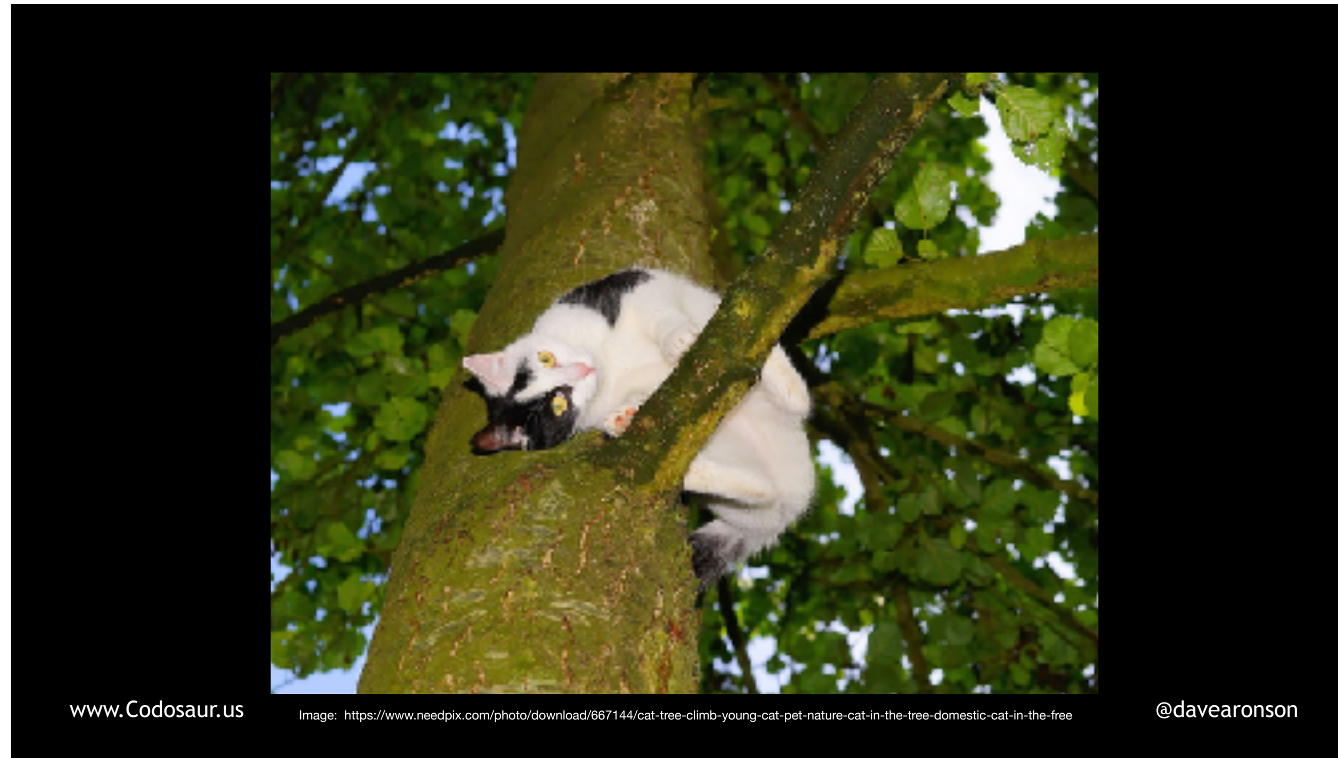
```
function euclid(a, b)
{
  while(b != 0) {
    if(a > b) a -= b;
    else      b -= a;
  }
  return a;
}
```

www.Codosaur.us

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

@davearonson

. . . parses our code, usually into an Abstract Syntax Tree. (I know those boxes are too small to read well, but we don't need to understand this one in detail.) After our tool makes an AST from our code, then it . . .



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

```
// @mumu tests-for foo
describe("#foo", function() {
  it("turns 3 into 6", function() {
    expect(foo(3)).toEqual(6)
  });

  it("turns 4 into 10", function() {
    expect(foo(4)).toEqual(10)
  });
});
```

www.Codosaur.us @davearonson

. . . annotating our tests or following some . . .

```
describe("#foo", function() {
  it("turns 3 into 6", function() {
    expect(foo(3)).toEqual(6)
  });

  it("turns 4 into 10", function() {
    expect(foo(4)).toEqual(10)
  });
});
```

www.Codosaur.us

@davearonson

. . . naming convention. This is often supplemented and sometimes even replaced by . . .

```
describe("#foo", function() {
  it("turns 3 into 6", function() {
    expect(foo(3)).toEqual(6)
  });

  it("turns 4 into 10", function() {
    expect(foo(4)).toEqual(10)
  });
});
```

www.Codosaur.us

@davearonson

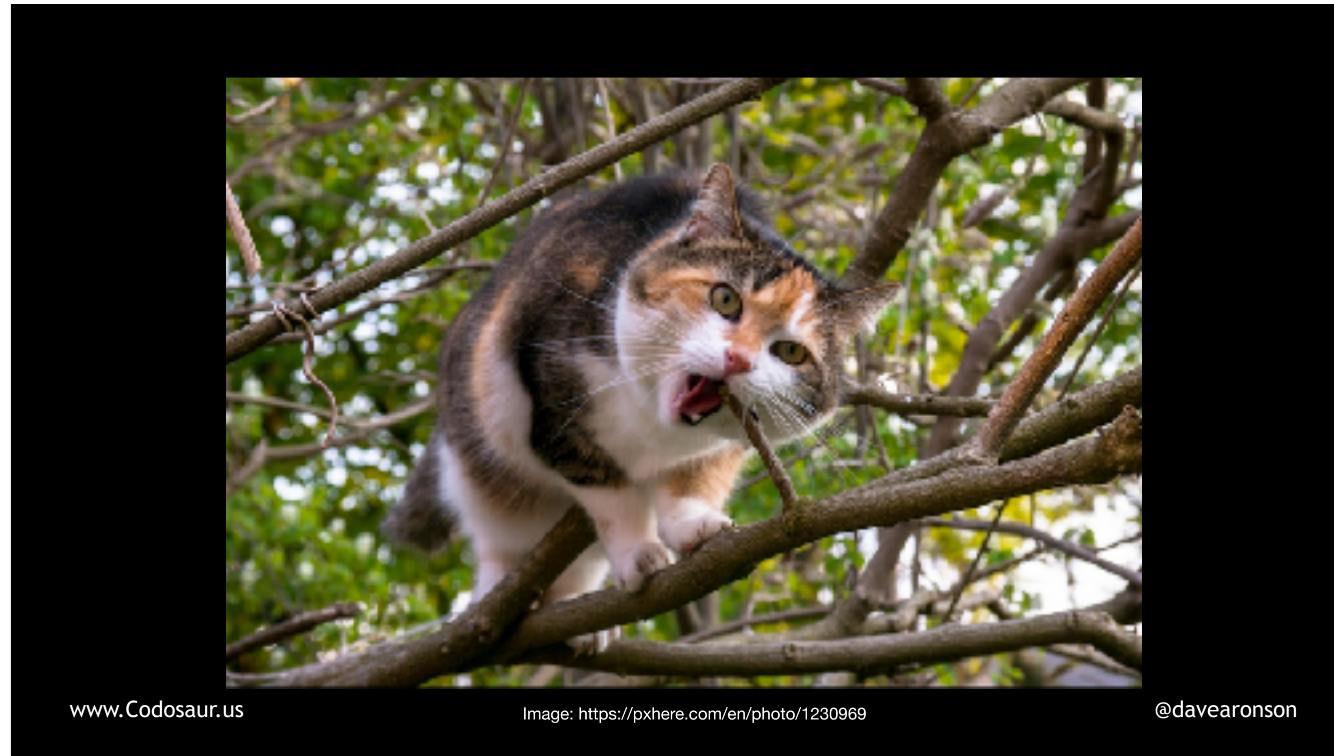
. . . the tool looking at what unit tests call what functions, though that can get tricky if the function . . .

```
describe("#foo", function() {
  it("turns 3 into 6", function() {
    foo_test_helper(3, 6)
  });

  it("turns 4 into 10", function() {
    foo_test_helper(4, 10)
  });
});
```

www.Codosaur.us @davearonson

. . . isn't called *directly* from the test. (PAUSE!) Next the tool makes the mutants. To make them *from an AST subtree*, it . . .



. . . 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 a copy of the function's AST subtree, with that node changed, in that way. For instance, suppose our tool has started traversing the AST I showed earlier, and has gotten down to . . .

. . . this not-equal comparison, following those arrows. For each way it could change that node, it would make a fresh copy, of this subtree, with only that 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 function subtree, and do likewise to all further nodes.

So, what kind of changes can it make? There are quite a lot!

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

Maybe even swap *between* sets!

www.Codosaur.us

@davearonson

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

If possible, it could substitute one from a different category. For instance, in JavaScript, we can treat *anything* as *booleans*, so x *times* y could become x *and* y.

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

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

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

www.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
x >  y
```


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

`x`

could become:

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

. . . or vice-versa!

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

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

or

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

could become:

```
foo(z)
```

www.Codosaur.us

@davearonson

It can remove a condition, or a loop control.

```
f(x, y)
```

could also become:

```
f(y, x)
f(x)
f(y)
f()
```

etc.

www.Codosaur.us @davearonson

It could *scramble* or *truncate* argument lists of function calls, or . . .

```
function f(x, y) { return x * y; }
```

could become:

```
function f(y, x) { return x * y; }
function f(x)    { return x * y; }
function f(y)    { return x * y; }
function f()     { return x * y; }
```

www.Codosaur.us @davearonson

. . . function *declarations*.

```
function f(x, y) {  /* many lines */   }
```

could become:

```
function f(x, y) { return x          ; }
function f(x, y) { return y          ; }
function f(x, y) { return 0          ; }
function f(x, y) { return MAX_INTEGER; }
function f(x, y) { return "a string" ; }
function f(x, y) { return undefined  ; }
function f(x, y) { throw("an error") ; }
function f(x, y) { /* no code here */  }
```

etc.

It could replace a function's *entire contents* with any of the arguments, or a constant, or raising an error, or nothing at all, if the language permits, and JavaScript does.

```
42
num
num + 42
f(num, 42)
```

etc.

could become:

```
-num
1
0
-1
num + 1
num - 1
num / 2
num * 2
num ** 2
sqrt(num)
```

```
MAX_INTEGER
MIN_INTEGER
MAX_VALUE
MIN_VALUE
Infinity
'a string'
undefined
null
NaN
```

etc.

It could change a constant or variable or expression or function call to some other value, even one of a different type, such as changing a number into a, if I may quote . . .

. . . Smeagol, “string, or nothing!”

There are many more, but I trust you get the idea!

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

```
function power(x, y) {
  return x ** y
}
```

www.Codosaur.us

@davearonson

. . . like so. (PAUSE!) Think about what a mutant made from this might *return*, since that's what our unit tests would probably be looking at. (PAUSE!) Mainly it could return results such as . . .

```
x + y
x - y
x * y
x / y
y ** x
(x ** y) / 0
x
y
0
1
-1
```

```
0.1
-0.1
MIN_INTEGER
MAX_INTEGER
MAX_VALUE
MIN_VALUE
Infinity
throw(DeliberateError)
"some random string"
nil
```

www.Codosaur.us

@davearonson

. . . any of *these* expressions or constants, and many more.

Now suppose we had only one test . . .

```
expect(power(2, 2)).
  toEqual(4)
```

www.Codosaur.us

@davearonson

. . . like so. This is a rather poor test, but even so, *most* of those mutants on the previous slide *would get killed* by this test, the ones shown . . .

x + y
~~x - y~~
x * y
~~x / y~~
y ** x
~~(x ** y) / 0~~
~~x~~
~~y~~
~~0~~
~~1~~
~~-1~~

~~0.1~~
~~-0.1~~
~~MIN_INTEGER~~
~~MAX_INTEGER~~
~~MAX_VALUE~~
~~MIN_VALUE~~
~~Infinity~~
~~throw(DeliberateError)~~
~~"some random string"~~
~~nil~~

. . . here in crossed-out green. The ones returning constants, are very unlikely to match. Subtracting gets us zero, dividing gets us one, returning either argument alone gets us two, and the error conditions will at *least* make the test not pass. But . . .

x + y
~~x - y~~
x * y
~~x / y~~
y ** x
~~(x ** y) / 0~~
~~x~~
~~y~~
~~0~~
~~1~~
~~-1~~

~~0.1~~
~~-0.1~~
~~MIN_INTEGER~~
~~MAX_INTEGER~~
~~MAX_VALUE~~
~~MIN_VALUE~~
~~Infinity~~
~~throw(DeliberateError)~~
~~"some random string"~~
~~nil~~

www.Codosaur.us

@davearonson

. . . addition, multiplication, and exponentiation in the reverse order, all get us the correct answer, and will therefore survive this test.

We know this because when we run our tool, it gives us a report, that looks roughly like . . .

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

42 - function power(x, y) {
42 + function power(y, x) {

43 -   return x ** y
43 +   return x + y

43 -   return x ** y
43 +   return x * y

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

www.Codosaur.us

@davearonson

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

And that is, that if we changed . . .

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

```
42 - function power(x, y) {
42 + function power(y, x) {

43 -   return x ** y
43 +   return x + y

43 -   return x ** y
43 +   return x * y

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

www.Codosaur.us

@davearonson

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

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

42 - function power(x, y) {
42 + function power(y, x) {

43 -   return x ** y
43 +   return x + y

43 -   return x ** y
43 +   return x * y

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

www.Codosaur.us

@davearonson

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

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

42 - function power(x, y) {
42 + function power(y, x) {

43 -   return x ** y
43 +   return x + y

43 -   return x ** y
43 +   return x * y

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

www.Codosaur.us

@davearonson

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

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

42 - function power(x, y) {
42 + function power(y, x) {

43 -   return x ** y
43 +   return x + y

43 -   return x ** y
43 +   return x * y

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

www.Codosaur.us

@davearonson

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

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

42 - function power(x, y) {
42 + function power(y, x) {

43 -   return x ** y
43 +   return x + y

43 -   return x ** y
43 +   return x * y

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


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

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

42 - function power(x, y) {
42 + function power(y, x) {

43 -   return x ** y
43 +   return x + y

43 -   return x ** y
43 +   return x * y

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


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

So what is . . .

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

42 - function power(x, y) {
42 + function power(y, x) {

43 -   return x ** y
43 +   return x + y

43 -   return x ** y
43 +   return x * y

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

www.Codosaur.us @davearonson

. . . this set of surviving mutants trying to tell us? (PAUSE!) A good start to figuring that out, is to ask, how are these mutants surviving? The usual answer is that they give the same result as the original function. To determine how *that* happens, we can take a closer look at one mutant, and a test it passes. Let's start with . . .

the change:

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

our test:

```
expect(power(2, 2)).
  toEqual(4)
```

www.Codosaur.us

@davearonson

. . . the "plus" mutant. Looking at the change, together with our test, makes it pretty clear that this one survives because two plus two equals two to the second power.

To kill this mutant, we need to make at least one test use arguments such that x *to the* y is different from x *plus* y. For instance, we could add a test or change our test to something like . . .

```
expect(power(2, 4)).
  toEqual(16)
```

www.Codosaur.us @davearonson

. . . this. All the mutants that our original test killed, this would still kill. Two *plus* four is six, so this kills the plus mutant just fine. For that matter, two *times* four is eight, so this kills the "times" mutant as well. The argument-swapping mutants, however, survive, but we can attack them separately, no need to be a superhero about it. To do that, we can again either add a test, or slightly tweak our test . . .

```
expect(power(2, 3)).
  toEqual(8)
```

www.Codosaur.us

@davearonson

. . . like so. Three squared is nine, so this kills the argument-swapping mutants. Two *plus* three is five, and two *times* three is six, so the "plus" and "times" mutants *stay* dead, we don't get any . . .

. . . zombie mutants. With . . .

```
expect(power(2, 3)).
  toEqual(8)
```

www.Codosaur.us

@davearonson

. . . these inputs, the correct operation is the only simple common one that yields the correct answer.

This may make mutation testing sound . . .

www.Codosaur.us Image: https://commons.wikimedia.org/wiki/File:Simple_Simon_LCCN2003677693.jpg @davearonson

. . . simple, but this is a trivial example, so we could easily think up arguments to make all the mutants behave differently from the original. There were *lots* of ways to skin *that* flerken! So let's look at a more *complex* example! Suppose we have a function to send a message, . . .

```
function send_message(buf, len) {
  var sent = 0;
  while(sent < len) {
    var now = send_bytes(buf + sent,
                         len - sent);
    sent += now;
  }
  return sent;
}
```

www.Codosaur.us

@davearonson

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

A mutation testing tool could make lots and lots of mutants from this, but the one I want to show you is . . .

```
  function send_message(buf, len) {
    var sent = 0;
-   while(sent < len) {
      var now = send_bytes(buf + sent,
                           len - sent);
      sent += now;
-   }
    return sent;
  }
```


. . . this, an example of removing a loop control. Now suppose that this mutant survives our test suite, consisting mainly of . . .

```
expect(send_message(msg, size)).
  toEqual(size)
```

www.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 creating the message. Even without seeing that code, what does the survival of that non-looping mutant tell us? (PAUSE!)

If a mutant that only goes through . . .

```
function send_message(buf, len) {
  var sent = 0;
  while(sent < len) {
    var now = send_bytes(buf + sent,
                         len - sent);
    sent += now;
  }
  return sent;
}
```

www.Codosaur.us @davearonson

. . . that while-loop once, acts the same as our normal code, as far as our tests can tell, that implies that our *tests* are only making our code go through that while-loop once. So what does that mean? (You'll find that interpreting mutants involves a lot of asking yourself what something means! Recursively!)

It means that we're not testing sending a message larger than send_*bytes* can handle in one chunk! The most likely cause is that we simply didn't test with a big enough message. For instance, . . .

in module Network:

```
MaxChunkSize = 10_000;
```

in test_send_message:

```
msg = 'foo';
size = msg.length;
# other setup, eg, stub send_bytes
expect(send_message(msg, size)).
  toEqual(size);
```

www.Codosaur.us @davearonson

. . . suppose send_*bytes* can handle 10,000 bytes in one chunk, but, we're only testing with a *three* byte message! (PAUSE!)

The obvious fix is to use a message larger than our maximum chunk size. To construct one, we can just . . .

in module Network:

```
MaxChunkSize = 10_000;
```

in test_send_message:

```
size = Network.MaxChunkSize + 1;
msg = "x" * size;
# other setup, eg, stub send_bytes
expect(send_message(msg, size)).
  toEqual(size);
```


. . . take the size, add one, and construct that big a message.

But perhaps 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:

```
SmallMsg = msg_class(SmallMsgSize);
LargeMsg = msg_class(LargeMsgSize);
```

in test_send_message:

```
size = Message.LargeMsgSize;
msg = new LargeMsg("a" * size);
# other setup, eg, stub send_bytes
expect(send_message(msg, size)).
  toEqual(size);
```

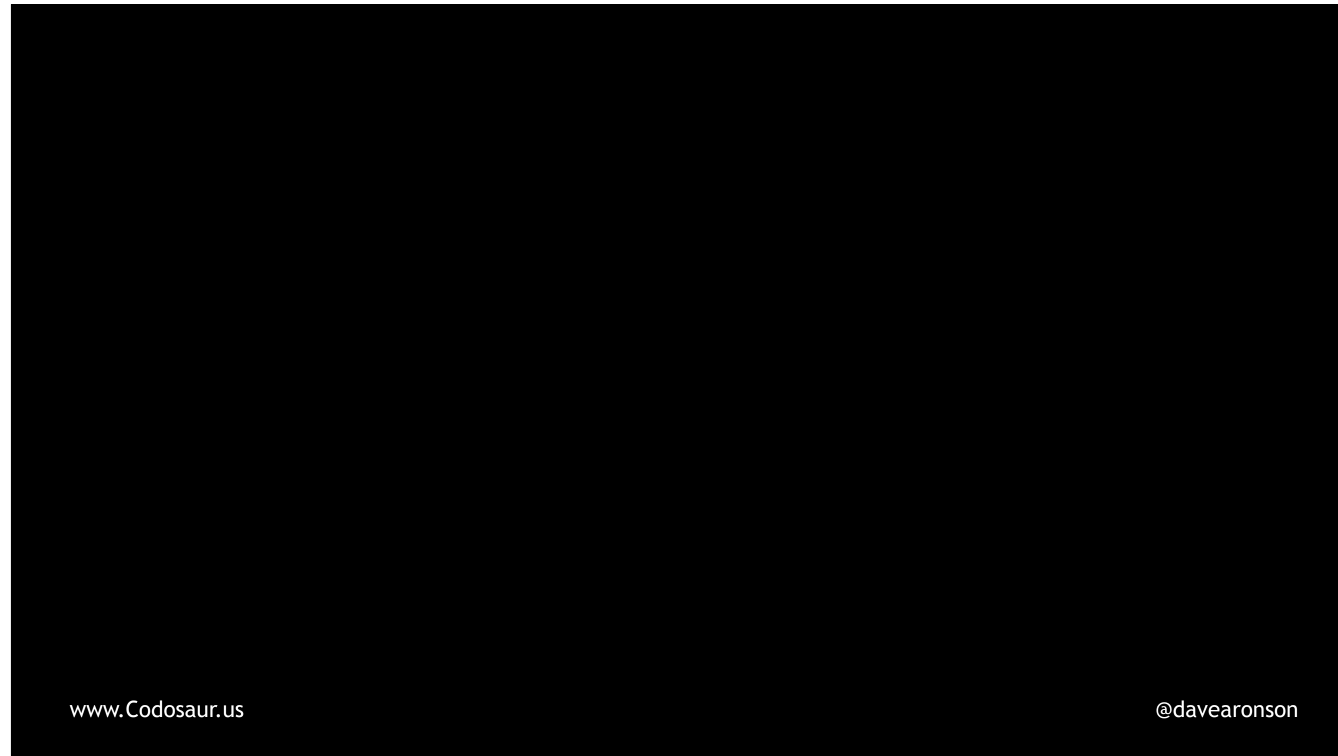

. . . here we have Small and Large message sizes. We tested with a Large, and yet, the mutant survived! In other words, we're still sending the whole message in one chunk. What is the non-looping mutant trying to tell us now? (PAUSE!)

It's trying to tell us that a version of send_message with the looping removed will do the job just fine. If we *also* remove all the other stuff we need only to support the looping, then it boils down to . . .

```
function send_message(buf, len)
{
  return send_bytes(buf, len);
}
```


. . . this. (PAUSE!) Now it's pretty clear: the *entire* send_message *function* may well be *redundant*, so we can just use send_*bytes directly!* It might not be, because, in real-world code, there may be some logging, error handling, and so on, needed in send_message, but at least the looping was redundant. Fortunately, when it's this kind of problem, with unreachable or redundant code, the solution is clear and easy, just chop out the extra junk that the mutant doesn't have, and anything else that makes redundant.

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



. . . ensure that our code is meaningful and (CLICK!) our tests are strict. It's (CLICK!) easy to get started with, in terms of setting up most of the tools and annotating our tests if needed (which may be *tedious* but at least it's *easy*), but it's (CLICK!) not so easy to interpret the results, nor is it (CLICK!) 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 (CLICK!) a really cool concept . . . in a geeky kind of way.

If I've EDUCATED you enough to INSPIRE you to CONNECT with your inner mutant, so you'd like to try mutation testing for yourself . . .

😀 Ensures our code is meaningful

😀 Ensures our code is meaningful

😀 Ensures our tests are strict

- 😀 Ensures our code is meaningful
- 😀 Ensures our tests are strict
- 😀 Easy to get started with

- 😀 Ensures our code is meaningful
- 😀 Ensures our tests are strict
- 😀 Easy to get started with
- 😩 Difficult to interpret results

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

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

Android:	mdroid+
C:	mutate.py, SRCIROR
C/C++:	accmut, dextool, MuCPP, Mutate++, mutate_cpp
C#/.NET/Mono:	nester, NinjaTurtles, Stryker.NET, VisualMutator
Clojure:	mutant
Elixir:	exmen, mutation, exavier
Erlang:	mu2
FORTRAN-77:	Mothra (written in mid 1980s!)
Go:	go-mutesting
Haskell:	fitspec, muCheck
Java:	jumble, major, muJava, pit, and many more
JavaScript:	**stryker**, ~~grunt-mutation-testing~~
PHP:	humbug, infection
Python:	cosmic-ray, mutmut, xmutant
Ruby:	mutant, mutest, heckle
Rust:	mutagen
Scala:	scalamu, stryker4s
Swift:	muter
Anything on LLVM:	llvm-mutate, mull

www.Codosaur.us

@davearonson

. . . here's 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 have time for pictures. Just be aware that some of these tools are outdated; I don't know or follow quite *all* of these languages and platforms. As for JavaScript, the only one I'm aware of is Stryker; there *used* to be one that was a plugin for the Grunt task runner, but that project has shut down and its code has been migrated into Stryker. Anybody need more time for pictures?

Lastly, a couple shoutouts, first to . . .

Thanks to Toptal and their Speakers Network!

https://toptal.com/#accept-only-candid-coders

www.Codosaur.us

Images: Toptal logo, used by permission

@davearonson

. . . Toptal, a consulting network I'm in, whose Speakers Network helped me prepare and practice this presentation. (Please use that referral link if you want to hire us or join us.)

And second, many thanks to . . .

Thank you Markus Schirp!

https://github.com/mbj

www.Codosaur.us

Images: Markus, from his Github profile

@davearonson

. . . Markus Schirp, who created mutant, a Ruby mutation testing tool, and has been very willing to answer my questions and critique this presentation.

And now, if you have any questions, . . .

https://www.Codosaur.us

T.Rex-2020@Codosaur.us

@davearonson (Twitter)

Slides: bit.ly/kill-mutants-JSConfHI-2020

. . . we're not supposed to do Q&A, but we can talk over lunch, and if you think of anything later, I'll be around for the rest of the conference, or if it's too late, there's my contact information up there, and of course I have cards. Now let's go get some lunch!

```
var killed;
for(meth of our_functions) {
  for(mutant of make_mutants(meth)) {
    killed = false;
    for(test of meth.tests) {
      killed = test.fails_with(mutant);
      if(killed) break;
    }
    if(!killed) report(mutant);
  }
}
```