From Examples to Exhaustive:
Intro to Property-Based Testing

by Dave Aronson
T.Rex-2026@Codosaur.us
linkedin.com/in/DaveAronson

Speaker notes

NOTE TO SELF: click on timer to reset it at the start

Guten Tag, Wien!

(Hello, Vienna!)

Image: standard emoji

Ich bin Dave Aronson, und hier

(I'm Dave Aronson, here)

Image: me speaking at JSConf Hawai'i 2020!

um Euch Eigenschaftsbasiertes Testen zu zeigen.

(to teach you about Property Based Testing.)

Image: https://www.publicdomainpictures.net/en/view-image.php?image=84072

Jedoch . . .

(However . . .)

Image: standard emoji

auf Englisch.

(in English.)

Image: standard emoji

Speaker notes

Property-Based Testing, often shortened to PBT, is a software testing technique (surprise!) that lets us stop writing . . .

```ruby
def test_this_1
  ...
end

def test_that_1
  ...
end

def test_something_1
  ...
end

def test_whatever_1
  ...
end

def test_something_else_1
  ...
end

def test_another_thing_1
  ...
end

def test_yet_another_thing_1
  ...
end

def test_one_more_thing_1
  ...
end

def test_random_thing_1
  ...
end
```

```ruby
def test_this_2
  ...
end

def test_that_2
  ...
end

def test_something_2
  ...
end

def test_whatever_2
  ...
end

def test_something_else_2
  ...
end

def test_another_thing_2
  ...
end

def test_yet_another_thing_2
  ...
end

def test_one_more_thing_2
  ...
end

def test_random_thing_2
  ...
end
```

```ruby
def test_this_3
  ...
end

def test_that_3
  ...
end

def test_something_3
  ...
end

def test_whatever_3
  ...
end

def test_something_else_3
  ...
end

def test_another_thing_3
  ...
end

def test_yet_another_thing_3
  ...
end

def test_one_more_thing_3
  ...
end

def test_random_thing_3
  ...
end
```

Speaker notes

. . . a huge number of tests that are identical except that they test different examples, and maybe expect different results or maybe not. That is what's usually called example-based testing, what you might recognize from typical unit tests, integration tests, feature tests, and so on. Property-based testing can also be done at all those levels; the big difference is, property-based testing lets us write . . .

```
def test_domain_1_aspect_1
  ...
end

def test_domain_1_aspect_2
  ...
end

def test_domain_2_aspect_1
  ...
end

def test_domain_2_aspect_2
  ...
end
```

Speaker notes

. . . a small number of tests, and describe the type of examples each one applies to, and some desired property of the output, perhaps in relation to the input. It may take multiple such tests to cover the whole domain of valid inputs if there are multiple kinds, or to test all the desired relationships and properties, but almost always many fewer tests than in an example-based approach. In fact, sometimes we may be able to write just one test. The property-based testing tool, generally a library or framework, will . . .

Image: https://pxhere.com/en/photo/615255

Speaker notes

. . . randomly generate lots of inputs, and examine the corresponding outputs. How many inputs, you may wonder? As many as we want to let it test, within constraints such as time, CPU cycles, etc. We might give it a length of clock time it can use, or some measure of CPU cycles or CPU time, or a number of iterations.

That . . .

Image: https://vectorportal.com/vector/price-reduction-sticker.ai/21963

Speaker notes

. . . reduction in the number of tests is the main advantage of Property-Based Testing, over the standard approach of writing a bunch of examples.

Now you might be reminded of Teamscale's technique of test suite reduction. Did any of you go to their talk Tuesday afternoon? I'm not going to recap their idea, you can talk to them at their booth. Both ideas are about reducing the test suite, but Teamscale's idea is aimed at reducing the clock time and CPU time for running a subset of an exisiting test suite, while Property-Based Testing is aimed at reducing the programmer time involved in writing the test suite in the first place.

Property-Based Testing also reduces implicit human . . .

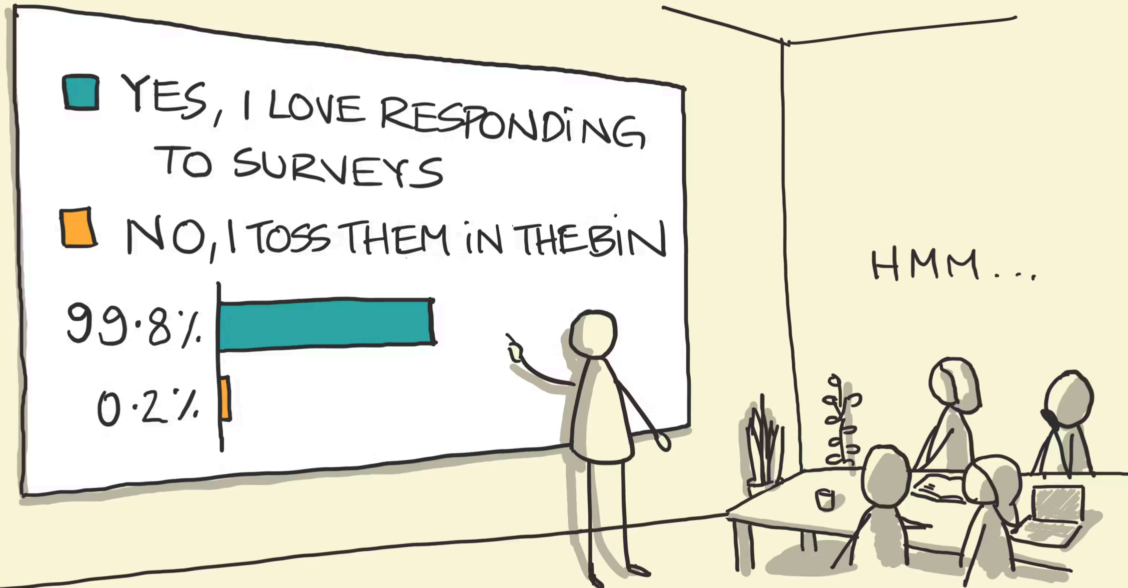


Image: PD, from https://www.thefemalelead.com/post/navigating-gender-bias-in-the-workplace-lessons-from-tackling-racial-bias

Speaker notes

. . . bias, like how some thing can pass tests with human-scale input numbers, but might blow up or or have errors with something huge.

For instance, suppose some step in our algorithm involves multiplying two input integers, that could have any 64-bit value, with the result put into another 64-bit integer. It should work fine with numbers that humans are likely to use, up to maybe hundreds or thousands. However, it’s quite easy to create a bug in that sort of situation, mainly integer overflow. Multipling six times seven (DO HAND GESTURE), no problem. Multipling that many thousand or million, still no problem. But multiplying six billion by seven billion, exceeds the 64 bit range.

By sampling the rest of the range, beyond the tiny slice we humans would probably think of, Property-Based Testing can help point out how the term "valid input" . . .

Image: all over the web!

Speaker notes

. . . doesn't mean what we think it means. This will help us think about our valid input ranges. Then we can restrict the inputs by other means, such as catching invalid values programmatically, or declaring a more restrictive type, if our language of choice has such a feature.

To clarify the idea, let’s try an example. Let’s suppose we’re writing tests for everybody's favorite interview whiteboard-coding classic . . .

```
Is x a:            multiple of 3?
                     Y          N
              _______Y__________N____
multiple  Y| FizzBuzz | Buzz |
of 5?     N| Fizz     | x    |
```

Speaker notes

. . . FizzBuzz. I don't know how common this is over here, especially its use as a developer hiring filter, but if you're not familiar with it, the basic idea is that it's a children's counting game, used in schools to drill the idea of multiples. For each number, if the number is

Is x a:		multiple of 3?	
		Y	N
multiple of 5?	Y	FizzBuzz	Buzz
	N	**Fizz**	x

Speaker notes

a multiple of three but not five, they say "Fizz" instead of the number. If it's the other way around,

Is x a:		multiple of 3?	
		Y	N
multiple of 5?	Y	FizzBuzz	**Buzz**
	N	Fizz	x

Speaker notes

they say "Buzz".

Is x a:		multiple of 3? Y	N
multiple	Y	**FizzBuzz**	Buzz
of 5?	N	Fizz	x

Speaker notes

If it's a multiple of both they say "FizzBuzz". And

```
Is x a:          multiple of 3?
                    Y          N
              _____________________
multiple  Y| FizzBuzz  | Buzz |
of 5?     N| Fizz      | x    |
```

Speaker notes

if it's neither, then they finally get to say the actual number. So it should start

```
1: 1      |  6: Fizz
2: 2      |  7: 7
3: Fizz   |  8: 8
4: 4      |  9: Fizz
5: Buzz   | 10: Buzz
```

Speaker notes

1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, and so on.

So let's look at how we would usually test an implementation of this, by writing example-based tests. Each test would look for the correct response for a specific input number. The typical path would be to test 3, 5, 15, and some number that’s not a multiple of 3 nor 5, let's . . .

```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```


Image: https://xkcd.com/221/

Speaker notes

. . . randomly pick 4. After writing tests for those cases, most would call it a day. But I’m sure we can all think of easy implementations that would give the correct answer for those numbers, but not necessarily for any others, like . . .

```
case number
  when  3 then "Fizz"
  when  4 then "4"
  when  5 then "Buzz"
  when 15 then "FizzBuzz"
  else         "Garbage!"
end
```

Speaker notes

. . . so. For our carefully chosen examples, sure it returns the correct response . . . but for anything else, literally garbage. Maybe we’d be a bit more thorough, and also test another set, like 6, 10, 30, and 8.

```
case number
  when  3,  6 then "Fizz"
  when  4,  8 then "4"
  when  5, 10 then "Buzz"
  when 15, 30 then "FizzBuzz"
  else             "Garbage!"
end
```

Speaker notes

The same idea applies, no matter what small set of numbers we pick! What we really want to do is test that

$\forall x \in$ valid range, $f(x)$ is correct

Speaker notes

it gives correct answers for all valid inputs. We could skip to writing just one test like

```
def test_any_fizzbuzz
  n = rand(MAXINT)
  is_3 = n % 3 == 0
  is_5 = n % 5 == 0
  cond
    is_3 && is_5: expected = "FizzBuzz"
    is_3:         expected = "Fizz"
    is_5:         expected = "Buzz"
    else:         expected = n.to_string()
  end
  assert(fizzbuzz(n) == expected)
```

Speaker notes

this, but that's a serious test smell, of duplicating the system under test, what a friend of mine compares to double entry bookkeeping. We're not sure it's implemented quite like this, but that's a perfectly reasonable implementation. There are a number of things we can do instead, like look for invariants, or an inverse operation, or divide the input space into easily described cases. Aha! That cond statement up there shows us a set of four cases we can use. It's basically just an encoding of

```
Is x a:           multiple of 3?
                      Y          N
               _____________________
multiple   Y| FizzBuzz  | Buzz  |
of 5?      N| Fizz      | x     |
```

Speaker notes

the table from several previous slides.

Then we can write just one test for each of these four cases, let our property-based testing tool create random inputs, let it run for some reasonable amount of time or iterations or whatever, and have very strong confidence that we've tested a significant sampling of the input space, enough to have strong confidence in the correctness of our code. That's already only half the number of tests that we would have written for

```
case number
  when  3,  6 then "Fizz"
  when  4,  8 then "4"
  when  5, 10 then "Buzz"
  when 15, 30 then "FizzBuzz"
  else             "Garbage!"
end
```

Speaker notes

this implementation, without putting much of a serious dent in the input space.

So how do we write a test for an entire case, not just one number, especially without violating the testing principle of "one assertion per test"? With most property-based testing tools , it's very similar to . . .

```
def test_6_is_fizz
  expected = "Fizz"
  actual = fizzbuzz(6)
  assert(actual == expected)
end
```

Speaker notes

. . . writing one example-based test, except that . . .

```
def test_mult_of_3_not_5_is_fizz
  number = generate_mult_of_3_not_5()
  expected = "Fizz"
  actual = fizzbuzz(number)
  assert(actual == expected)
end
```

Speaker notes

. . . we usually call a generator to get the number. So what's . . .

Image: https://openclipart.org/detail/335427/portable-generator

Speaker notes

. . . a generator? A generator is a function to randomly generate valid test data. It can be as simple as calling a random-number function each time, or it can get more complex. The data can be primitives like in this case, slightly more complex things like lists, or even custom things, such as objects (or records or maps or whatever our language calls them), or lists or trees or other such networks or relationships of custom objects, like a customer with orders that each have line items that point to products and so on. (But don't worry, I'm going to keep these examples simple.)

Usually the generator is where we would incorporate constraints, such as "give me an integer, not just any integer, but one from 1 to 100, inclusive, that is a multiple of three, but not of five". That might look like this:

```
MAX_VALUE = 100

def generate_mult_of_3_not_5
  max_rand = MAX_VALUE/3  # integer div
  candidate = random(max_rand+1) * 3
  if candidate % 5 == 0
    return generate_mult_of_3_not_5
  else
    return candidate
  end
end
```

Speaker notes

There are many many ways we could implement this. The quality of our generator can drastically affect the ease of writing our property-based tests, and their effectiveness. For instance, if we're testing a name capitalization function, using a random string generator won't be as good as a generator aimed at realistic names, including all the common quirks. Our property-based testing tool may provide such a thing, or something usable that way with certain arguments, but we may have to write one ourselves.

Most of the tools supply a few kinds of generators, and some of them have very flexible ones, for which the call for a Fizzbuzz input number might look more like this:

```
generate_integer(min: 1,
                 max: 100,
                 multiple_of: 3,
                 not_multiple_of: 5)
```

Speaker notes

We could either use a call like that every time, or wrap it in helper functions with names like the ones I used before, or whatever else we might want to do to make it a bit easier to read or write.

Anyway, we can tell our property-based testing tool, via each test, the definition of our case, and the proper response. In the first three cases, they're just constant strings, so their property-based tests are very simple:

```
def test_mult_of_3_not_5_is_fizz
  number = generate_mult_of_3_not_5()
  expected = "Fizz"
  actual = fizzbuzz(number)
  assert(actual == expected)
end
```

Speaker notes

This is a repeat of the test that I showed you about four slides back, for numbers that are multiples of three but not five.

```
def test_mult_of_5_not_3_is_fizz
  number = generate_mult_of_5_not_3()
  expected = "Buzz"
  actual = fizzbuzz(number)
  assert(actual == expected)
end
```

Speaker notes

This would be the other way around, and

```
def test_mult_of_15_is_fizzbuzz
  number = generate_mult_of_15()
  expected = "FizzBuzz"
  actual = fizzbuzz(number)
  assert(actual == expected)
end
```

Speaker notes

. . . this would be for multiples of fifteen. In the fourth case though,

```
def test_mult_of_neither_is_number
  number = generate_non_multiple([3,5])
  expected = number.to_string()
  actual = fizzbuzz(number)
  assert(actual == expected)
end
```

Speaker notes

we have to transform the input from a number to a string. Still not at all a big deal.

So here we see that using property-based tests, we can write just four tests, let it run for a while, and have fairly thorough sampling of the input space, as opposed to spending human time writing lots of examples. It's a great example of the adage that we should let the computer do the boring grunt work. In this case that boring grunt work is just the selection of examples, once we've done the brainwork of describing the categories of examples, and the expected response for each.

Now, this is a very simple set of tests. So now let’s look at something a bit more complex, one of the canonical examples in introductions to property based testing: reversing a list. We don’t care about how it’s implemented, but we can take advantage of the relationship between the input and output: if we reverse the output, we get the input again. So what we can do is:

```ruby
def generate_random_list(max_len, max_val)
  len = random(max_len)
  return [1..len].map { random(max_val) }
end
```

Speaker notes

first write a generator to make up a random length, and return a list of that many random numbers.

```
def test_reverse_reverses
  original = generate_random_list(10, 10)
  reversed = my_reverse(original)
  reversed_twice = my_reverse(reversed)
  assert(reversed_twice == original)
end
```

Speaker notes

In our test we can call the generator, hold onto what it gives us, run it through our reversal algorithm, and test that if we run the result through the reversal algorithm again, we get the original list.

However, there are many other things one could do that would have the same effect, like negating each number, without affecting the order of the list. (And in fact we can use this same idea, of running it through twice, to test a negation function.) But we can make it at least a little bit harder to fake, by also

```
def test_reverse_swaps_first_and_last
  original = generate_random_list(10, 10)
  reversed = my_reverse(original)
  assert(reversed.first == original.last)
  assert(reversed.last == original.first)
end
```

Speaker notes

testing that the first element of the original is the last element of the result, and vice-versa. Yes, this may occasionally pass with an incorrect algorithm, but that's why we test several different aspects of the functionality, and many repetitions.

Sometimes it can be hard to find and encapsulate good properties, but there are some useful guidelines. Scott Wlaschin (pronounced "ve-LAS-shin"?) made

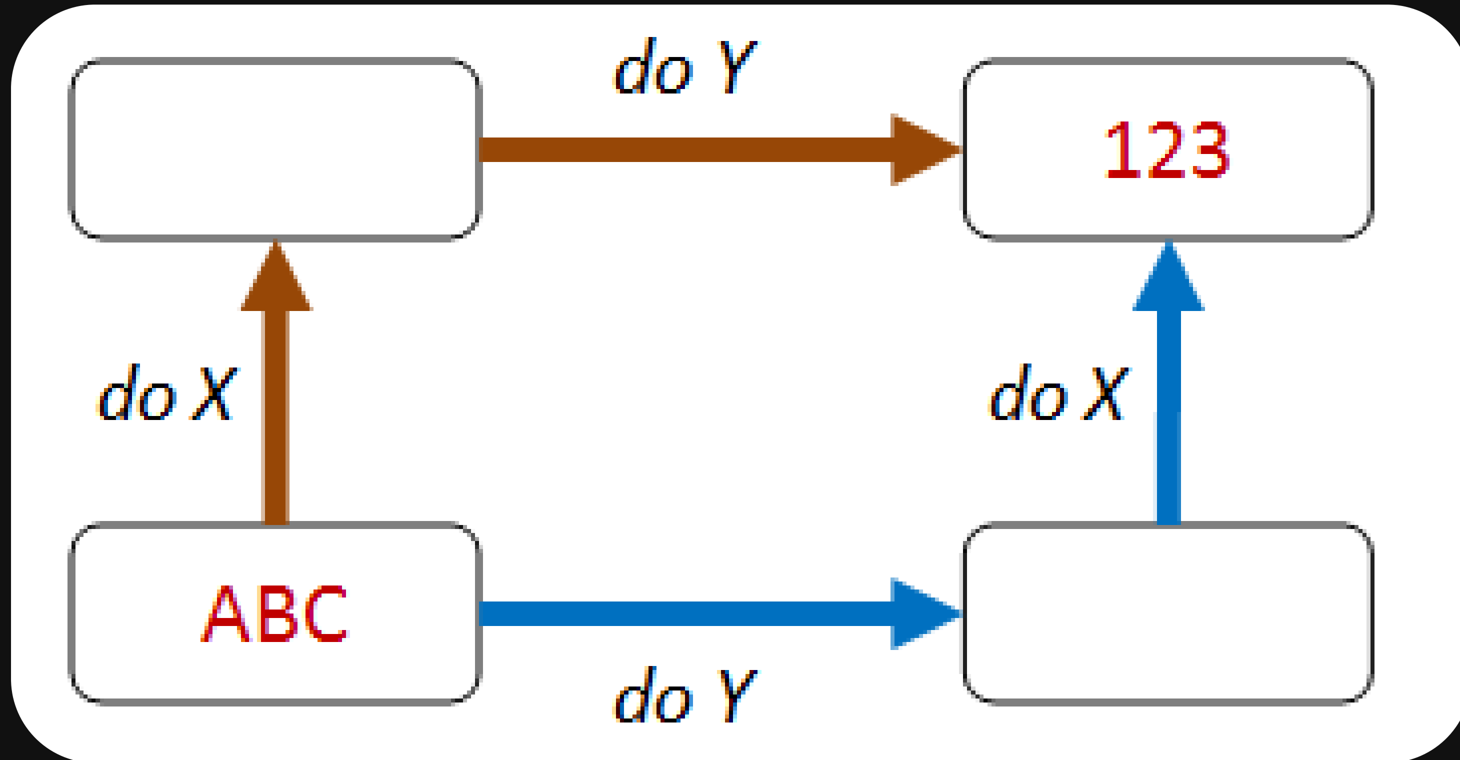
Scott Wlaschin's Seven Kinds of Properties:

- Different paths, same destination
- There and back again
- Some things never change
- The more things change, the more they stay the same
- Solve a smaller problem first
- Hard to prove, easy to verify
- The test oracle

Speaker notes

a list of seven approaches. The graphics on the following slides are shamelessly stolen from his post titled "Choosing properties for property-based testing" on his blog at FSharpForFunAndProfit.com.

Different paths, same destination



@davearonson

Image: from https://fsharpforfunandprofit.com/posts/property-based-testing-2/

Speaker notes

"Different paths, same destination" is about doing operations in a different order, but getting the same result. It's very useful when it's the same or related operations, like addition and subtraction, or division and multiplication, so that the order doesn't actually matter, or in math terms it's commutative. For instance, multiplying by 10 and then dividing by 3 gets us the same result as dividing by 3 and then multiplying by 10... at least ignoring floating-point problems.

There and back again

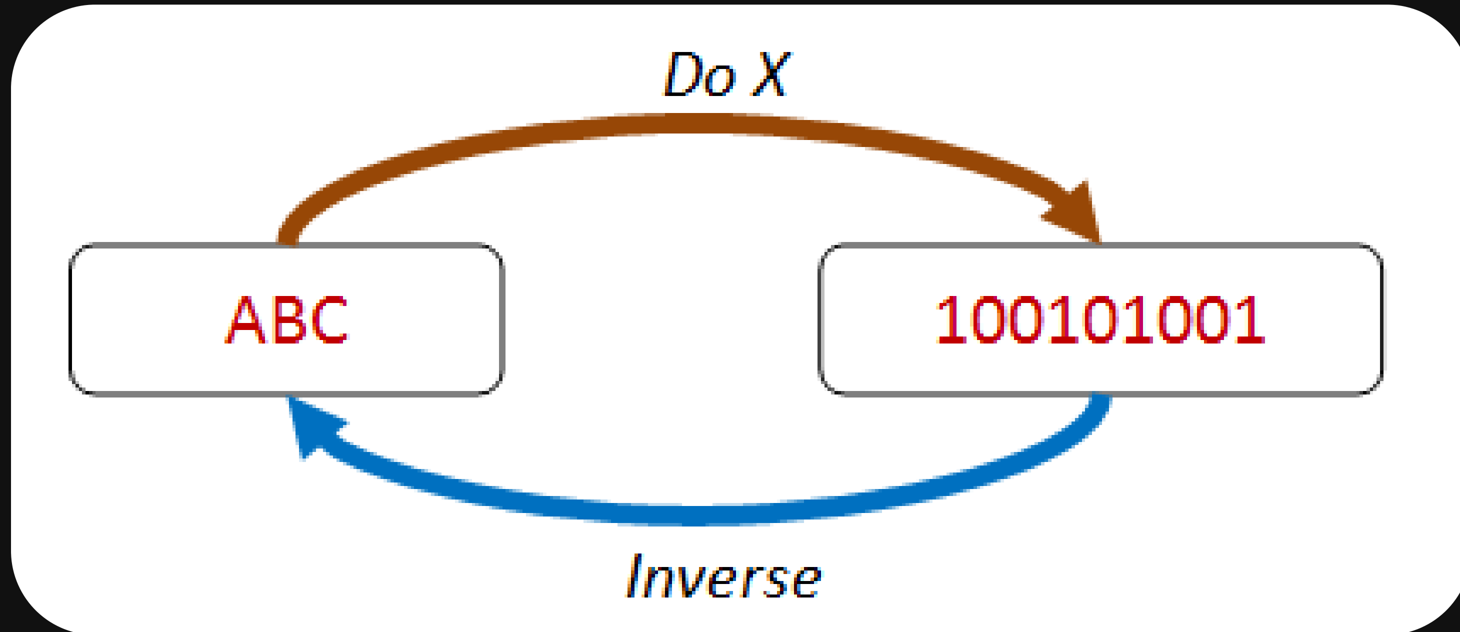


Image: from https://fsharpforfunandprofit.com/posts/property-based-testing-2/

Speaker notes

"There and back again" is about combining an operation with the inverse operation, so that we get the original value. For instance, add and subtract, multiply and divide, and even setting and then verifying some value, like writing a value to disk and reading it back, using a setter and a getter on a property of an object, or serialization and deserialization. Sometimes an operation is its own inverse, such as reversing a list or negating a number, as mentioned earlier.

Some things never change

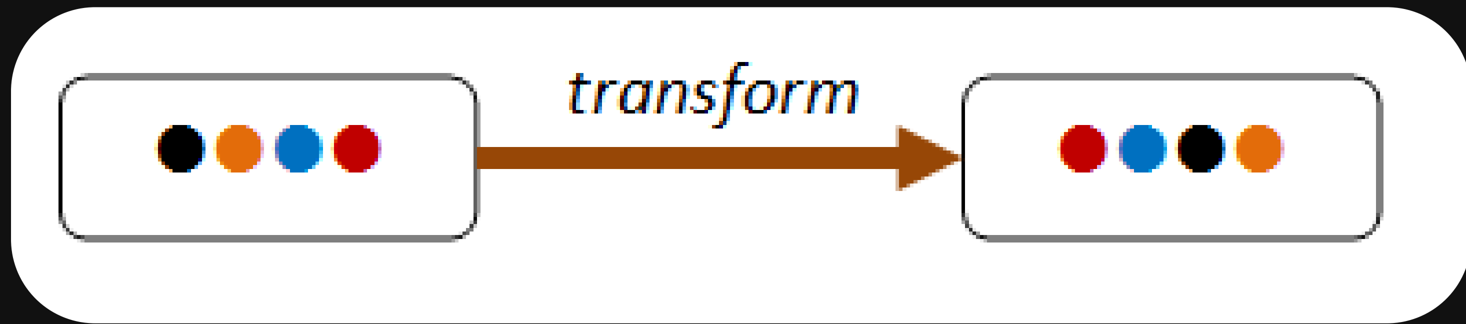


Image: from https://fsharpforfunandprofit.com/posts/property-based-testing-2/

Speaker notes

"Some things never change" is about invariants that are perserved through a transformation. For instance, when we sort a list, its size and members are preserved, only their order changes. For instance, if there are three fives in the input, and ten items in total, there must be three fives and ten items total in the output, so we know it hasn't been collapsed down to a unique set or some other such error.

The more things change, the more they stay the same

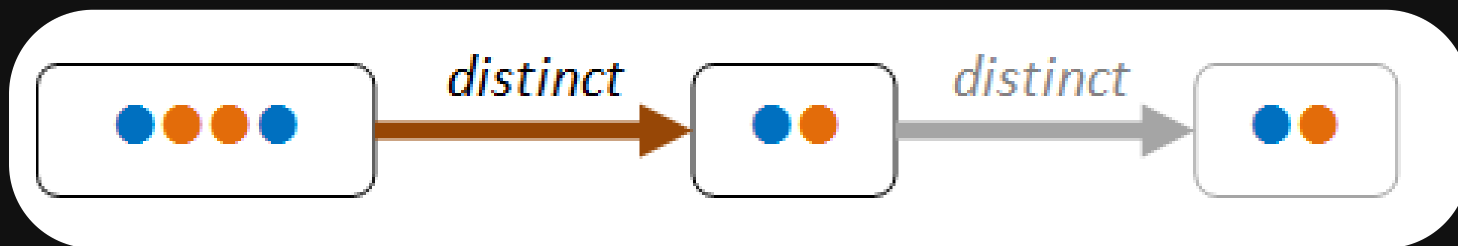


Image: from https://fsharpforfunandprofit.com/posts/property-based-testing-2/

Speaker notes

"The more things change, the more they stay the same" is about idempotence, the idea that an operation can be applied multiple times and the result will always be the same as the first time. The example he gives is selecting the distinct members of a list, but it is a useful principle in programming, to make sure that a loop run amok doesn't do too much damage. It's particularly important in message and event processing, in which an item may be processed more than once.

Solve a smaller problem first

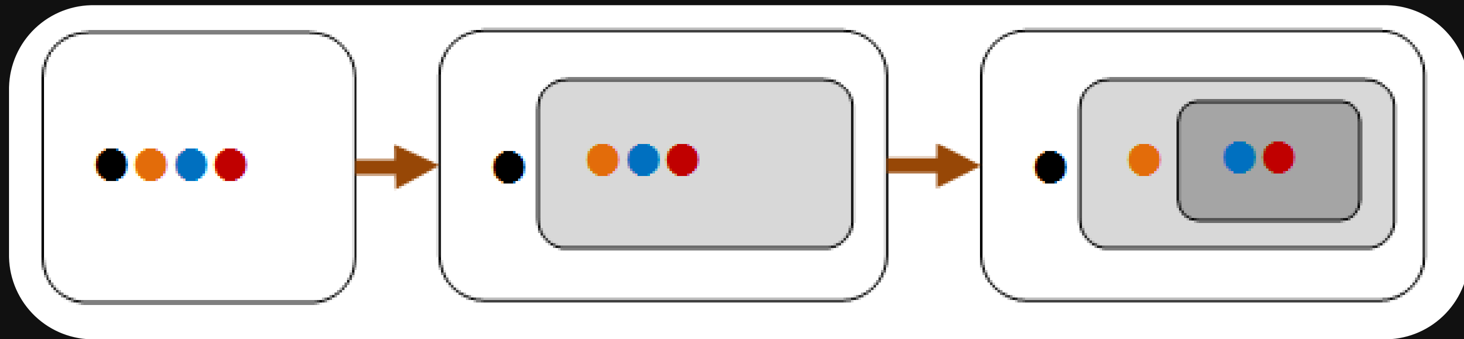


Image: from https://fsharpforfunandprofit.com/posts/property-based-testing-2/

Speaker notes

"Solve a smaller problem first" is about the logical principle of induction -- if we can prove some idea about a smaller piece, and prove that adding onto it doesn't break the idea, we can prove it about the whole thing, by induction. This is very useful for recursive structures like lists and trees.

Hard to prove, easy to verify

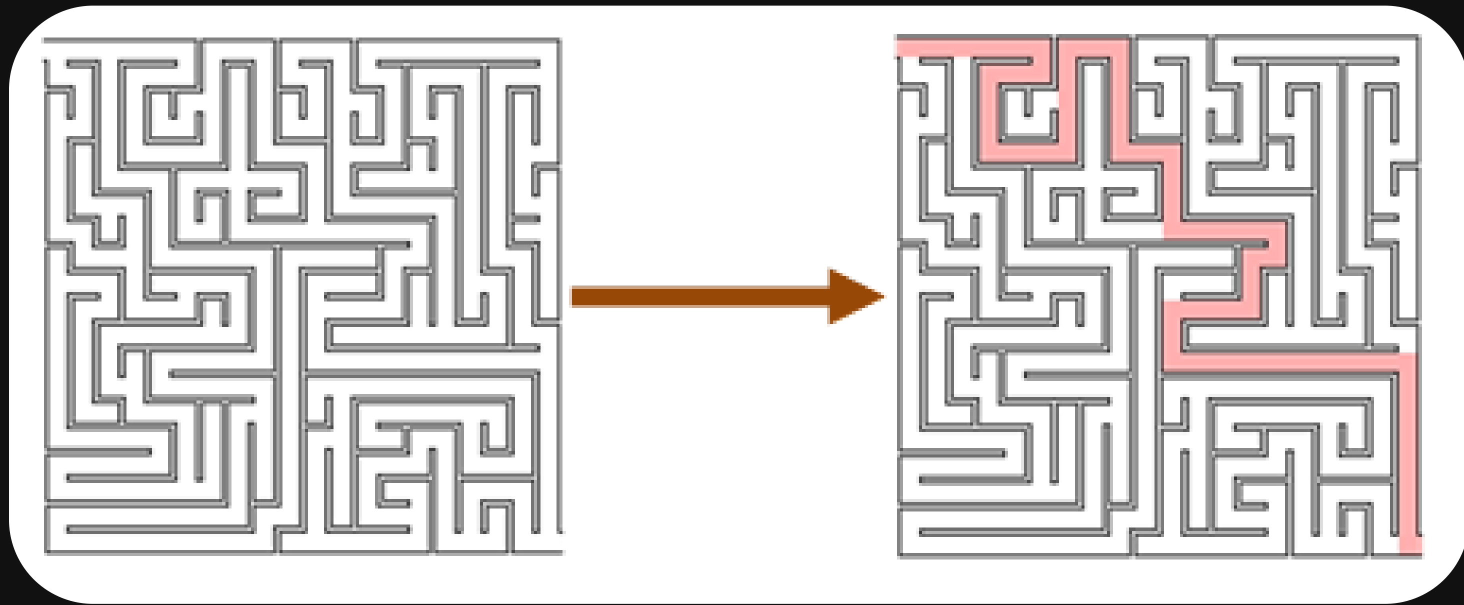


Image: from https://fsharpforfunandprofit.com/posts/property-based-testing-2/

Speaker notes

"Hard to prove, easy to verify" is I think a bit misnamed, and should really be "Hard to solve, easy to verify". In this formulation, it should sound familiar to the theory geeks, as this is this basis of the whole debate over whether P = NP, and no I'm not going to delve into that further. The idea behind this is that the function we're testing may be very hairy and take a long time, but it may be very quick to verify that it worked, such as in this example verifying that the route it found does indeed get from the start to the end of the maze. The goal is that the verification is fast enough to use in a test, ideally one executed many many times, as is very common in property-based testing.

The test oracle

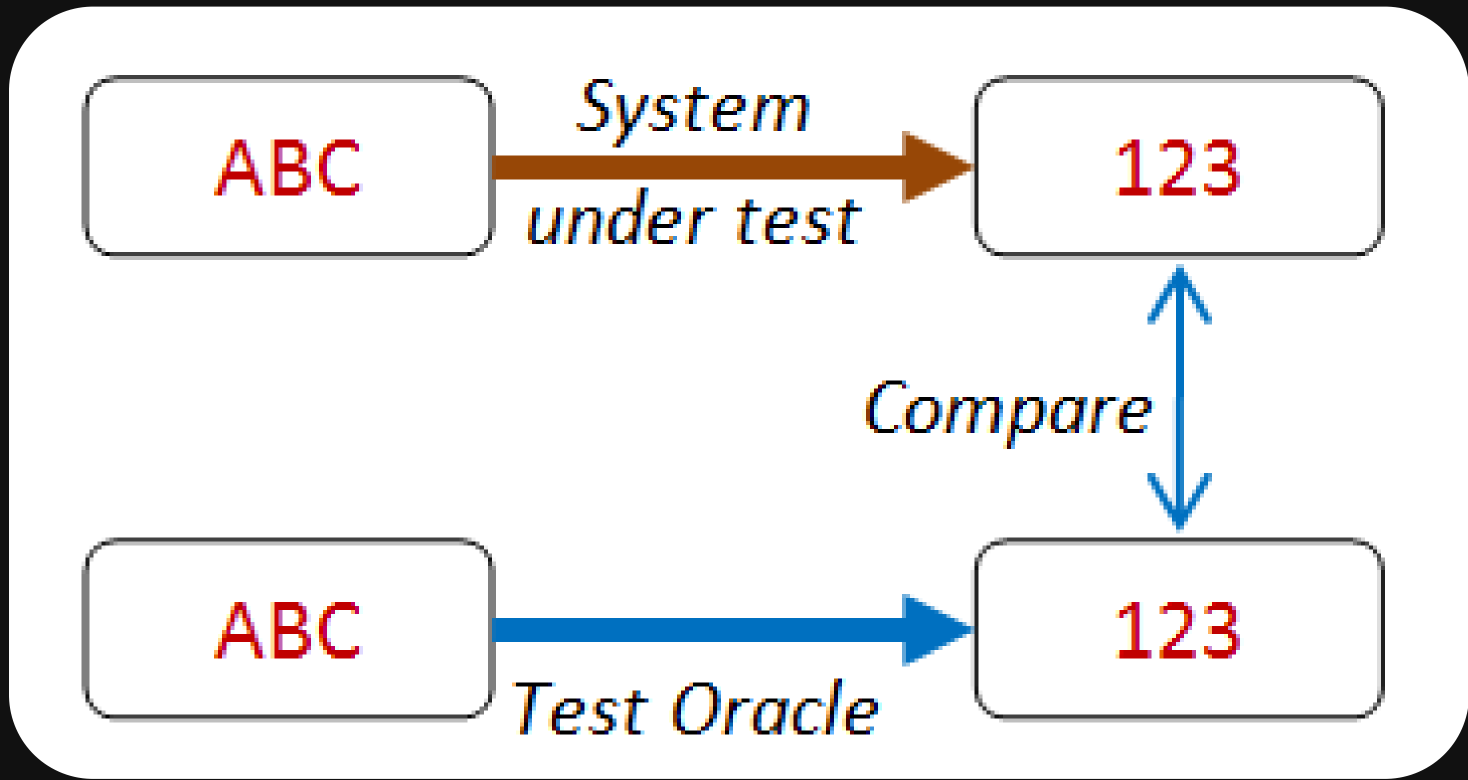


Image: from https://fsharpforfunandprofit.com/posts/property-based-testing-2/

Speaker notes

"The test oracle" is about comparing two ways of doing something. This is particularly useful when we're changing from one algorithm, that we know works, to another, that we are implementing. This could be a small bit of internals, or a whole feature, like in the Strangler Fig pattern, or indeed the entire system. We get a result from the one we know works, and see if the new one comes up with the same answer.

Now that we know what property-based testing is, do we really need a . . .

Image: https://easy-peasy.ai/ai-image-generator/images/electrician-toolkit-20-essential-tools-hd

Speaker notes

. . . tool to do it? It makes the setup easier, but any decently experienced programmer should be able to write that from scratch, maybe even make their own framework or library for it, so it becomes a tradeoff, based mostly on how much of it we're doing. As an example of implementing it yourself,

```
PASSES = 100

def test_mult_of_3_not_5_is_fizz
  for pass_num in 1..PASSES do
    number = generate_mult_of_3_not_5()
    expected = "Fizz"
    actual = fizzbuzz(number)
    assert(actual == expected)
  end
end
```

Speaker notes

here is one of those FizzBuzz tests, in plain old MiniTest, the test runner that ships with Ruby, set to run some number of times.

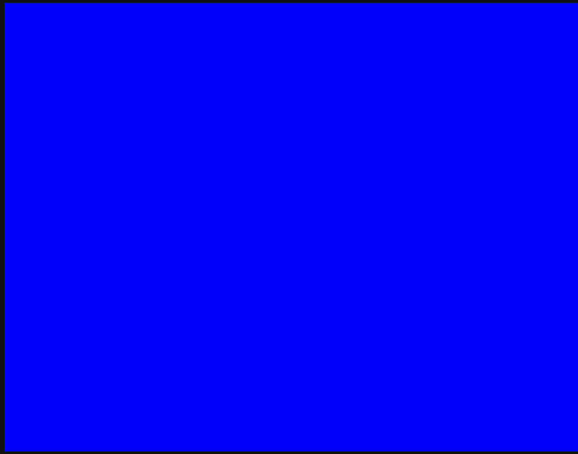
And

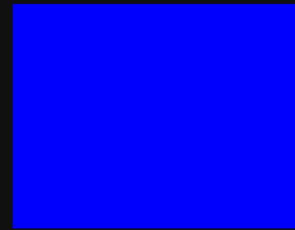
```
SECONDS = 0.1

def test_mult_of_3_not_5_is_fizz
  end_time = Time.now + SECONDS
  while Time.now < end_time do
    number = generate_mult_of_3_not_5()
    expected = "Fizz"
    actual = fizzbuzz(number)
    assert(actual == expected)
  end
end
```

Speaker notes

here it is set to run for some amount of time. And we could certainly use both constraints at once. While it's clearly possible to do all this without a property-based testing tool, at least using one would make it easier to write, and probably clearer for whoever has to maintain our test suite. (Keep in mind, that may well be "future us"!)

However, the real magic of most property-based testing tools is in something much more difficult, called





Speaker notes

“shrinking”. Not quite all tools do this, but the idea is to find the edges, by searching between an input that passes the test, and one that doesn’t, when it finds one of each.

Fizzbuzz has a repeating cycle of 15, but in many other cases, the rules or results change across the domain. Let’s look at

```
@n: Integer
def stupid_function(Integer n)
  1 / ((n / 5.0).round - 10)
end
```

Speaker notes

a simple function that divides 1 by an input integer, but first it takes that input, divides it by five, rounds to the nearest integer, and subtracts ten. (Never mind why, it just makes a convenient example!) Our output should be a floating point number in the range -1 to 1, but in some range of inputs, assuming we haven’t taken steps to avoid it, we’re going to have a division by zero error. That would be the range where dividing by five and rounding gets us ten, or in other words, without rounding we get 9.6 to 10.4, or in other other words,

```
47 -> -1
48 -> /0 ERROR
49 -> /0 ERROR
50 -> /0 ERROR
51 -> /0 ERROR
52 -> /0 ERROR
53 ->  1
```

Speaker notes

48 to 52.

Suppose we've told our tool the input range is integers from 1 to 100, and our tool randomly does 26, 74, and then 50, but of course blows up. First of course it tells us that, and now we know our truly valid input range isn't quite what we thought it was.

But wait! There's more!

It figures, if I may anthropomorphize it a bit, that there is probably an edge case, possibly multiple, somewhere between 50 and each of those numbers. Let's look at it trying to find the edge case between 26 and 50. It will basically do a binary search, trying halfway at

```
38 -> -1
44 -> -1
47 -> -1
49 -> /0 ERROR
48 -> /0 ERROR
```

Speaker notes

38, the middle of the remainder (so basically 3/4 of the way) at 44, 7/8 of the way at 47, and finally gets another error, 15/16 of the way up, at 49. There's only one number left to investigate between success and failure, at 48, which gives it another error, so now it can report to us that it found an edge case between 47 and 48. Then it will also investigate the other side, using a similar technique, and tell us about the edge between 52 and 53. Not only will revealing such failures in testing, rather than production, save us from outages later, but also, having a tool apply this idea of shrinking saves the human labor we might otherwise spend tracking down exactly where those edge cases lie.

With a more complex input, or set of inputs, a property-based testing tool will try to simplify it. For instance, if it detected a failure in our sorting algorithm, it would try to shrink the input to the shortest list that makes it fail. With an object, it might try to leave as many properties as possible empty, or filled in with default values, and similarly for a function call with multiple arguments.

Now that we see how much work this saves us, who deserves the

myBANK

9999 9999 9999 9999

VALID THRU 01/01

JOHN SMITH

MasterCard

Image: https://freesvg.org/credit-card-vector-template

Speaker notes

credit?

The first usage of the term is generally thought to be the 1994 paper titled "Towards a Property-based Testing Environment with Applications to Security-Critical Software" by George Fink, Calvin Ko, Myla Archer, and Karl Levitt. The ideas in this paper formed in turn a large part of Fink's PhD dissertation, in 1995 at the University of California at Davis, titled "Discovering security and safety flaws using property-based testing". However, the first real-world tool didn't appear until five years later, in 1999, when Koen Claessen and John Hughes developed a tool called QuickCheck, for the Haskell programming language. Not only was this the first major property-based testing tool, but it was also the origin of the idea of shrinking.

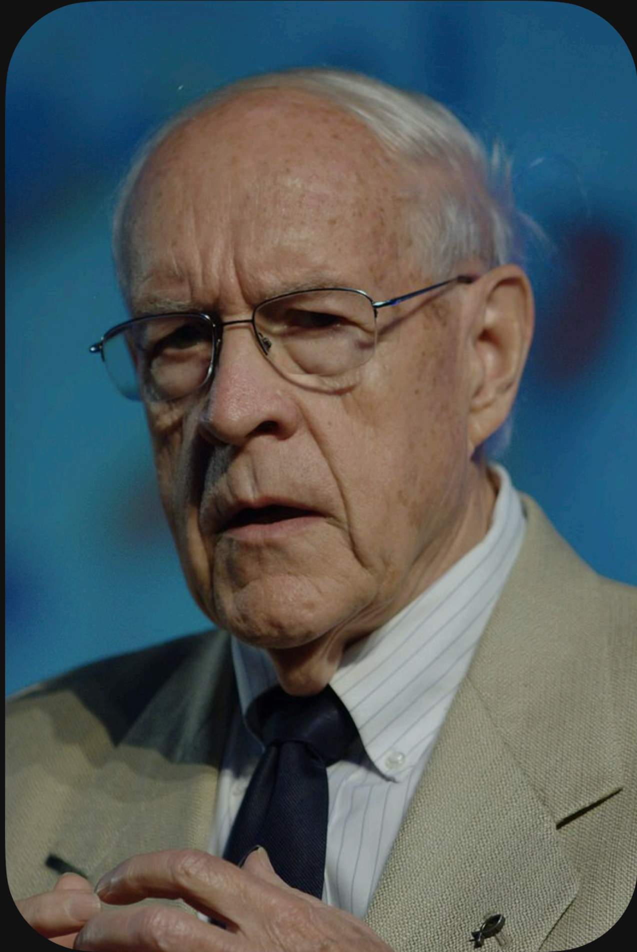
Even so, it still didn't penetrate the mainstream much until about sixteen more years later, in 2015, when David MacIver released Hypothesis, a property-based testing library for Python, certainly a much more popular language than Haskell! Since then, there has been an explosion of property-based testing tools for most major programming languages, and we'll get to a list later.

So is this a

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

Speaker notes

silver bullet? Of course not --



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

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

Speaker notes

Fred Brooks told us back in 1986 that there’s no such thing. It’s great for problems where valid inputs can be fairly easily defined in such a way that they can easily be created randomly, and the correctness of the output is quick and easy to check, but it’s not so great for things with heavy side effects or lots of stored state, or for very slow operations, since the idea is to try it many times with different inputs. However, even when dealing with such complex problems, at least some parts can usually be broken down into more suitable ones, with some effort. Even then, it doesn't fully replace example-based tests, but rather supplements them. It's just another tool in our

Image: from https://www.partycity.com/boys-dc-comics-adjustable-batman-utility-belt-1015607.html and https://freesvg.org/cartoon-bat

Speaker notes

belts, to be used when appropriate.

So once we've decided that a chunk of code is appropriate for property-based testing, when should we do it? How do we fit this into our

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

Speaker notes

development process?

That's a bit of a tricky question, because it's a multi-step process. Obviously, writing the tests would go in the

Claim Ticket and Make Branch

=> Write Tests

Write Code

Lint?

Refactor?

Create Pull Request

Get PR Approved

Merge PR and Delete Branch

Go Back, Jack, Do It Again

Speaker notes

Write Tests step. But then when do we run them, and for how long? Assuming we're using Test Driven Development, as shown here, we run them just after we

Claim Ticket and Make Branch

Write Tests

=> Write Code

Lint?

Refactor?

Create Pull Request

Get PR Approved

Merge PR and Delete Branch

Go Back, Jack, Do It Again

Speaker notes

write a piece of code that we think works, to verify it. If not, we run them after we write them, after the code. Whatever. But letting it run a long time, can ruin our feedback loop! We want to get back to work fast, while the problem at hand is fresh in our minds! Most of the property-based testing tools, just like most example-based ones, will let us run just one file, or maybe just one test, or some tagged subset, such as ones we mark as in-progress, if we're adding tests to an existing file. And, most of them will let us also specify command line parameters to override what's in the tests or the config file about how long to let them run, be it by time or iterations or whatever. If we run it for only a very brief time, that might not be as thorough a sampling of the input space as we might want in the end, but it's still probably enough for now. Even all that command-line stuff will be much faster than taking the time to write an equivalent number of example-based tests, especially if we put it into an alias or a script or some such thing.

Later, when we're ready to

Claim Ticket and Make Branch

Write Tests

Write Code

Lint?

Refactor?

=> Create Pull Request

Get PR Approved

Merge PR and Delete Branch

Go Back, Jack, Do It Again

Speaker notes

create the Pull Request, or whatever our dev process calls some similar milestone, we can make a little more sure, and tell it to run them for a little longer, and run the rest of the tests, to make sure we didn't break anything. Then we can remove the in-progress tags on those tests, submit the PR, and ideally it would then go into a CI/CD pipeline. In the pipeline, maybe our default is to run each test for, say, ten iterations. Or maybe we might tag some tests as being highly important, like if they're for critical functionality to get precisely right, or very tricky code, so they get run a hundred times. On the other claw, maybe some are marked as not so important, or maybe fairly important but slow, so they only get run once. These can still get a variety of inputs over time because they will get run because of many different PRs, or any time anybody starts the pipeline going for whatever reason. The point is that at that point, how long we let the test suite run shouldn't be making a significant difference in when a human can receive feedback or get back to work, or when a feature becomes available because of an automated deployment. That's kinda the point of an automated pipeline!

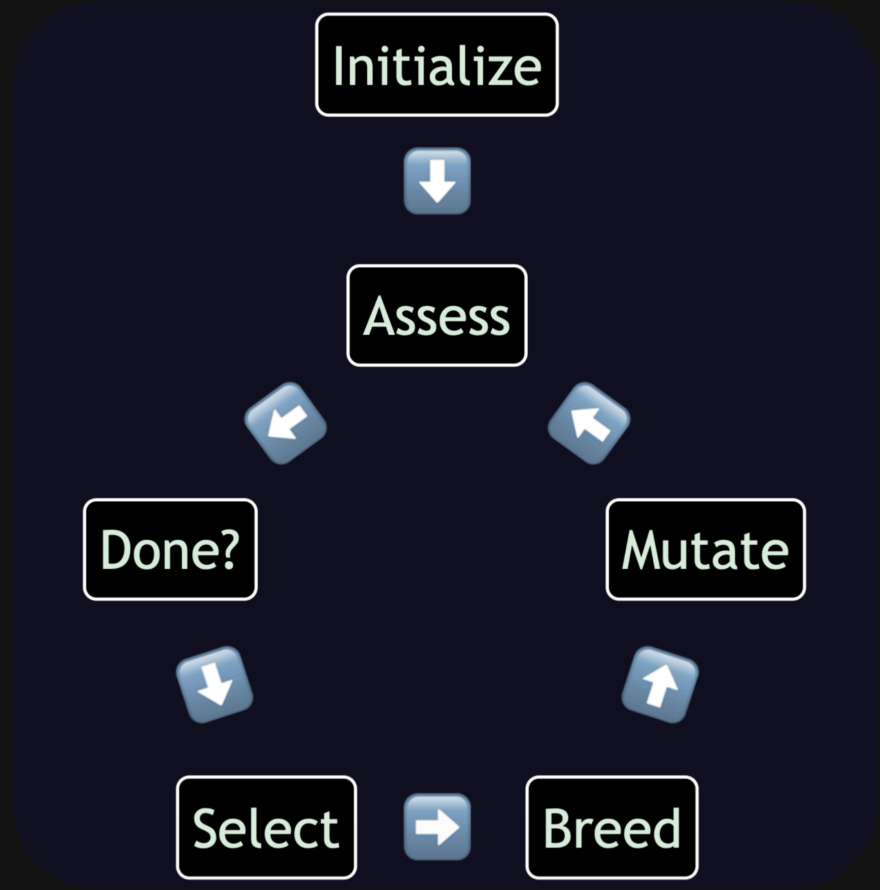
If you'd like to try Property-Based Testing for yourself, here's a

.NET	FsCheck
C/C++	DeepState
Clojure	ClojureCheck
Erlang/Elixir	PropEr
Go	Gopter, Rapid
Groovy	Gruesome
Haskell	QuickCheck (the original PBT tool!)
Java	JavaQuickCheck, JQF, jqwik
JavaScript	fast-check, JSVerify, QC.js
Python	Factcheck, Hypothesis, pytest-quickcheck
Ruby	Rantly
Rust	proptest
Scala	ScalaCheck
Solidity	Echidna

Speaker notes

list of tools for some popular languages. There is also a tool called Antithesis, which seems to be sort of a cross between a property-based testing tool and Chaos Monkey, for property-based testing of distributed systems.

Lastly, if this sort of randomized bug-hunting appeals to you, I also recommend you check out



Images: https://pixabay.com/en/genetic-modification-mutant-mutation-549889/ and my Tight Genes presentation

Speaker notes

Mutation Testing, and maybe Genetic Algorithms, two of my more frequent presentation topics. Mutation Testing is a technique for finding gaps in our test suites, so it shares with Property Based Testing the purpose of improving test quality, plus it can seem random, though it really isn't. Some of you may have seen my talk on it at Software Quality Days 2024, here in Vienna, I think maybe this very room. Genetic Algorithms are ways to "evolve" good solutions to a problem, using methods similar to real-world biological evolution. They share with property-based testing that the concept is based on random data, generated to be at least plausible for some given purpose.

Now that you know about the powerful technique of Property-Based Testing, and probably at least one tool to make it much easier in your language of choice, I encourage you to go try it on your projects. Once you get used to thinking about categories of inputs, rather than trying to come up with specific examples, you'll find you can make your test suite far more exhaustive, while actually writing fewer tests yourself.

? ? ?

Any questions?

Dave Aronson
T.Rex-2026@Codosaur.us
linkedin.com/in/DaveAronson
github.com/DaveAronson/Examples-to-Exhaustive

Speaker notes

(AFTER ALL QUESTIONS)
Please remember to send in your feedback, and if you give me anything but the top rating, I only ask that you tell me what could be improved in this presentation, and ideally how.