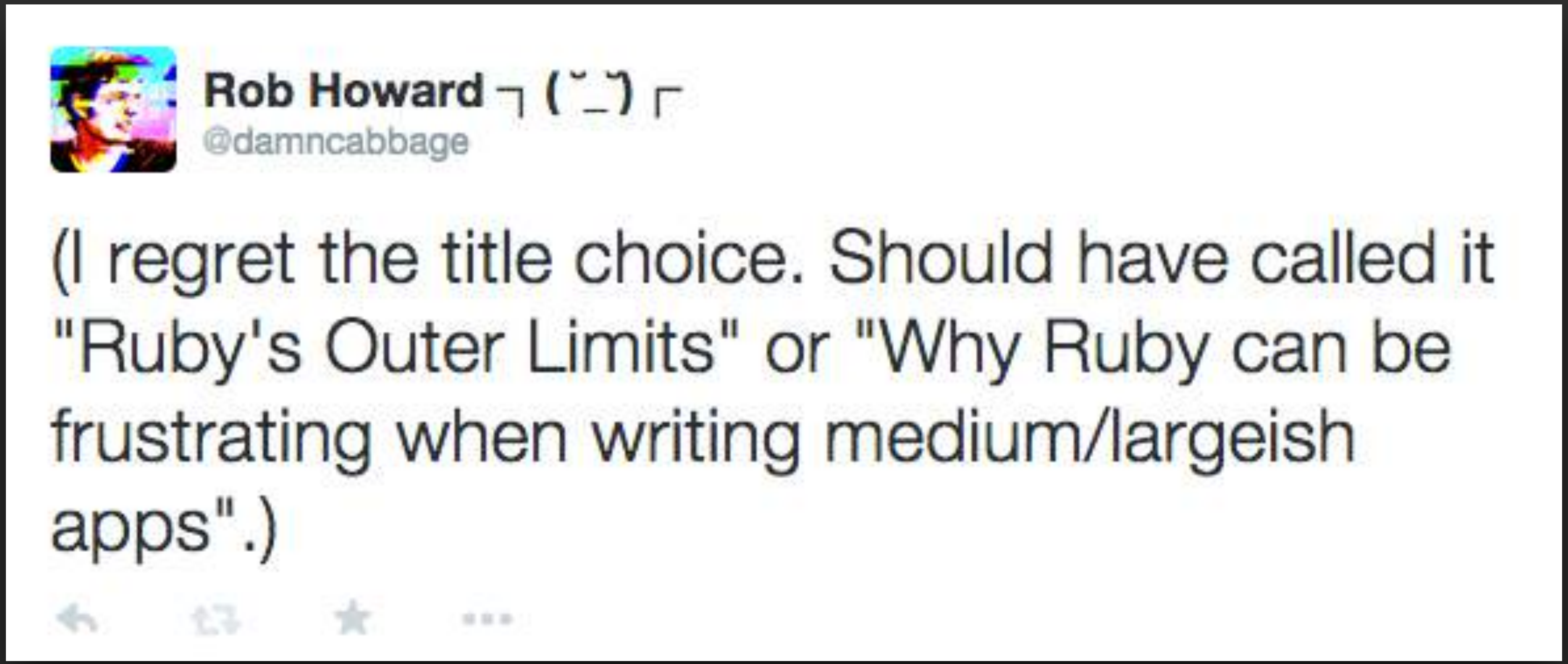
RUBY'S

OUTER LIMITS

(Previously: "Ruby is Doomed")



Rob Howard ┐(˘_˘)┌
@damncabbage

(I regret the title choice. Should have called it "Ruby's Outer Limits" or "Why Ruby can be frustrating when writing medium/largeish apps".)

RUBY'S

OUTER LIMITS

Or: "Why Ruby can be frustrating to use when writing medium/large-ish apps."

Who here has Ruby experience?

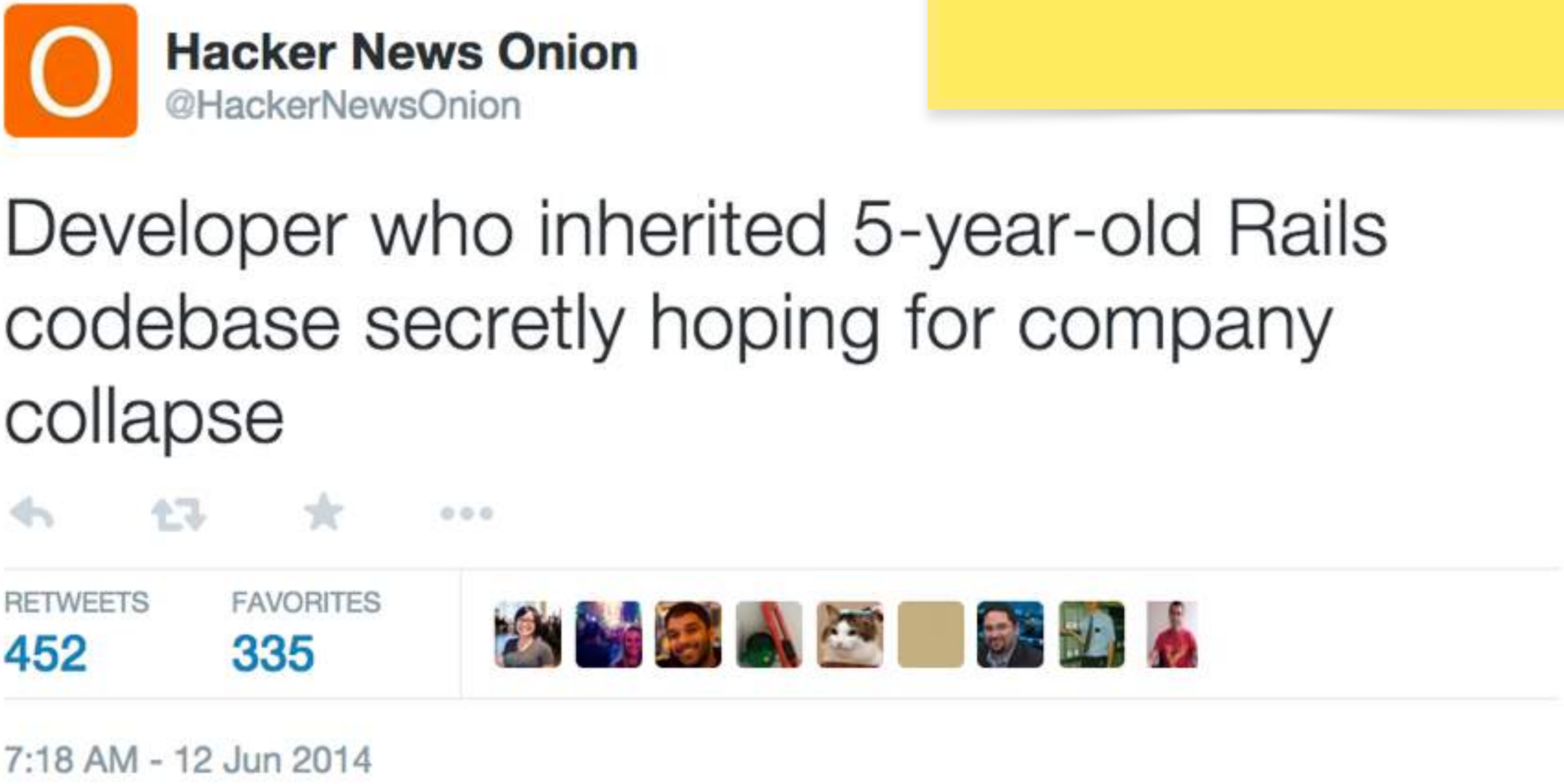
JS? PHP? Python?

Java? C? Go?

... Haskell?

Question Time

Does this sound familiar?



Hacker News Onion
@HackerNewsOnion

Developer who inherited 5-year-old Rails codebase secretly hoping for company collapse

RETWEETS 452

FAVORITES 335

7:18 AM - 12 Jun 2014

So you hear this a lot.

You're doing it wrong.

And this…

You should be doing…

Service Classes

You should be doing...

Hexagonal
Rails

Service
Classes

You should be doing...

Hexagonal
Rails

Service
Classes

You should be doing...

DCI
(Data Context
Interaction)

Hexagonal
Rails

Service
Classes

You should be doing...

DCI
(Data Context
Interaction)

FOLLOW THE
LAW OF DEMETER

Hexagonal
Rails

Service
Classes

Thin Controller,
Fat View
Fat Model

You should be doing...

DCI
(Data Context
Interaction)

FOLLOW THE
LAW OF DEMETER

Hexagonal
Rails

Thin Controller,
Fat View,
Fat Model

Service
Classes

Thin Controller,
Thin View,
Thin Presenters,
Fat Model

You should be doing...

DCI
(Data Context
Interaction)

FOLLOW THE
LAW OF DEMETER

You should be doing...

Hexagonal
Rails

Service
Classes

Thin Controller,
Fat View,
Fat Model

Thin Controller,
Thin View,
Thin Presenters,
Fat Model

Thin Controller,
Thin View,
Thin Presenters,
Thin Models,
Thin Persistence

DCI
(Data Context
Interaction)

FOLLOW THE
LAW OF DEMETER

I'm not sure many of these approaches are actually fixing anything; it feels like we're going around in circles because we're dividing and recombining something *essentially complex*, like pushing unwanted broccoli around a plate.

Bloody hell.

And that's because I think it's difficult, as a program gets larger, to figure out what your code is doing, and therefore makes it difficult to change (or refactor) your program safely, without getting a lot of help from the computer.

Bloody hell.

Safely changing programs is hard.
Safely changing programs **without a safety net** is harder.

Safety

Safety

- Modularisation
- Encapsulation
- Annotation
- Automatic detection of errors

We do a few things to make code safer to work on. Modularisation, grouping it into chunks. Encapsulation, hiding the internal state via abstraction. Annotation, by writing down how, say, a function can be used or what variables mean.

Safety

- Modularisation
- Encapsulation
- Annotation
- Automatic detection of errors

And Automatic detection of errors. Which, in Ruby land, is Testing.

And it's really the only tool we have. We check whether things work or not by running them over and over again with different parameters with the system in different states.

Definition Time

Static

Static, ability to look at and figure out what the code may do without running it.

Dynamic, only option is to run code it over and over, with different parameters, to check what it does.

Dynamic

An Example

What could a be?
What happens when we add 1 to a?

Ruby

```
def increment(a)
  a + 1
end
```

Ruby

```
def increment(a)
  a.+(1)
end
```

Anything. "a" can be anything.

What is +? It's very firmly tied to a. As is whether 1 is valid for that given +().

So let's think about just a small range of possibilities for a.

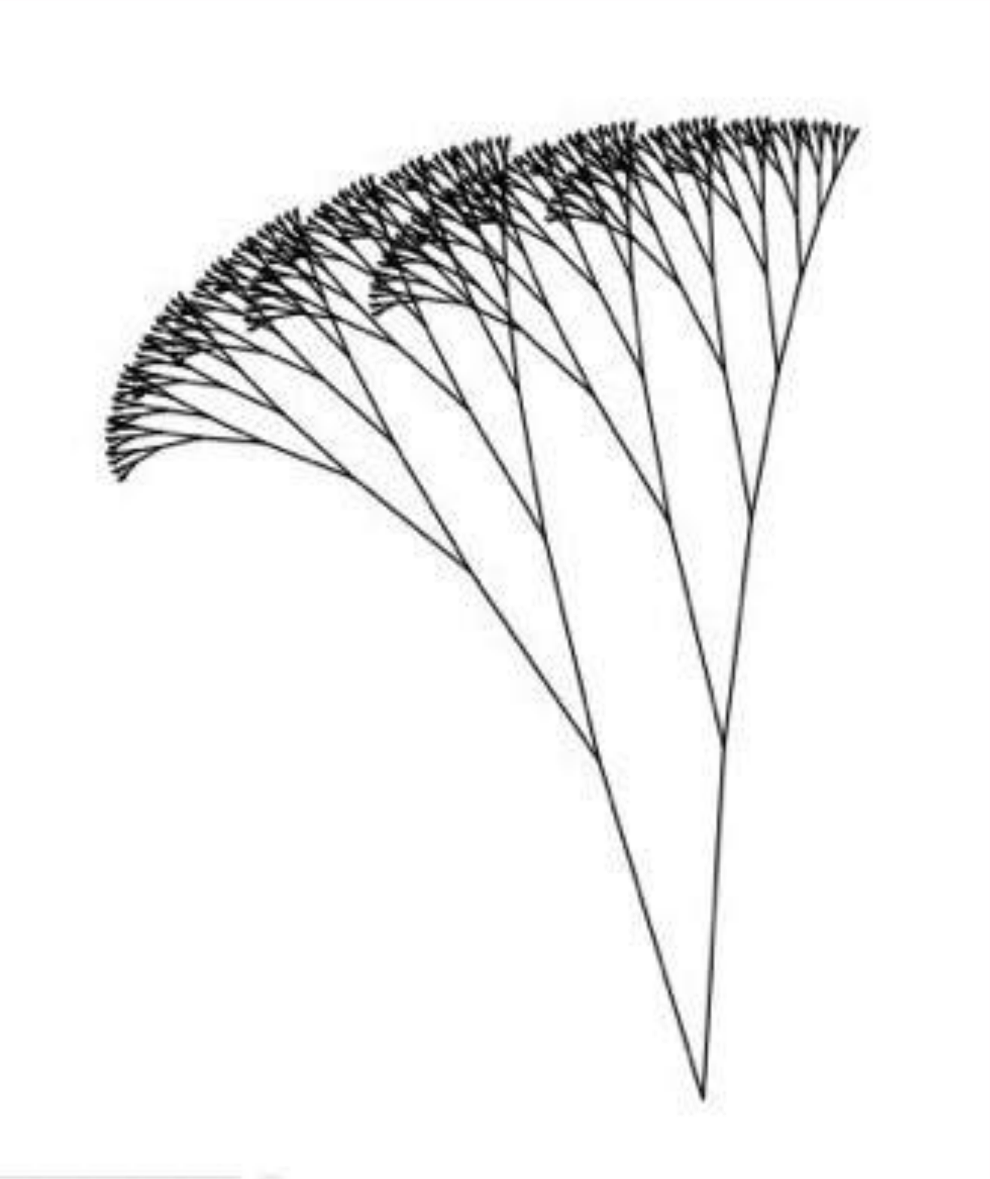
Ruby

```ruby
def increment(a)
  a + 1
end
```

So maybe a's an integer.
0, 1, 300, -6, ...

Ruby

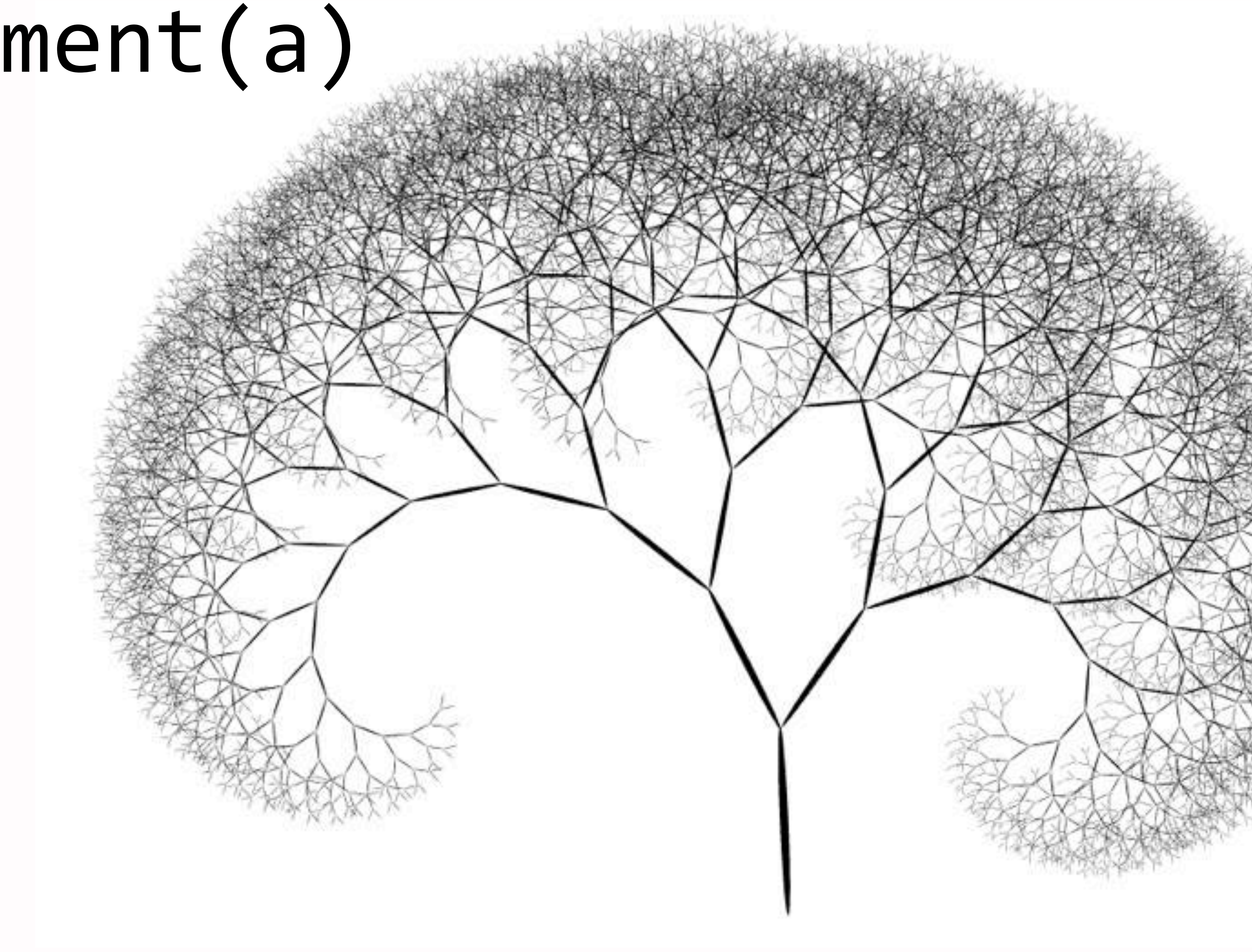
```
def increment(a)
  a + 1
end
```

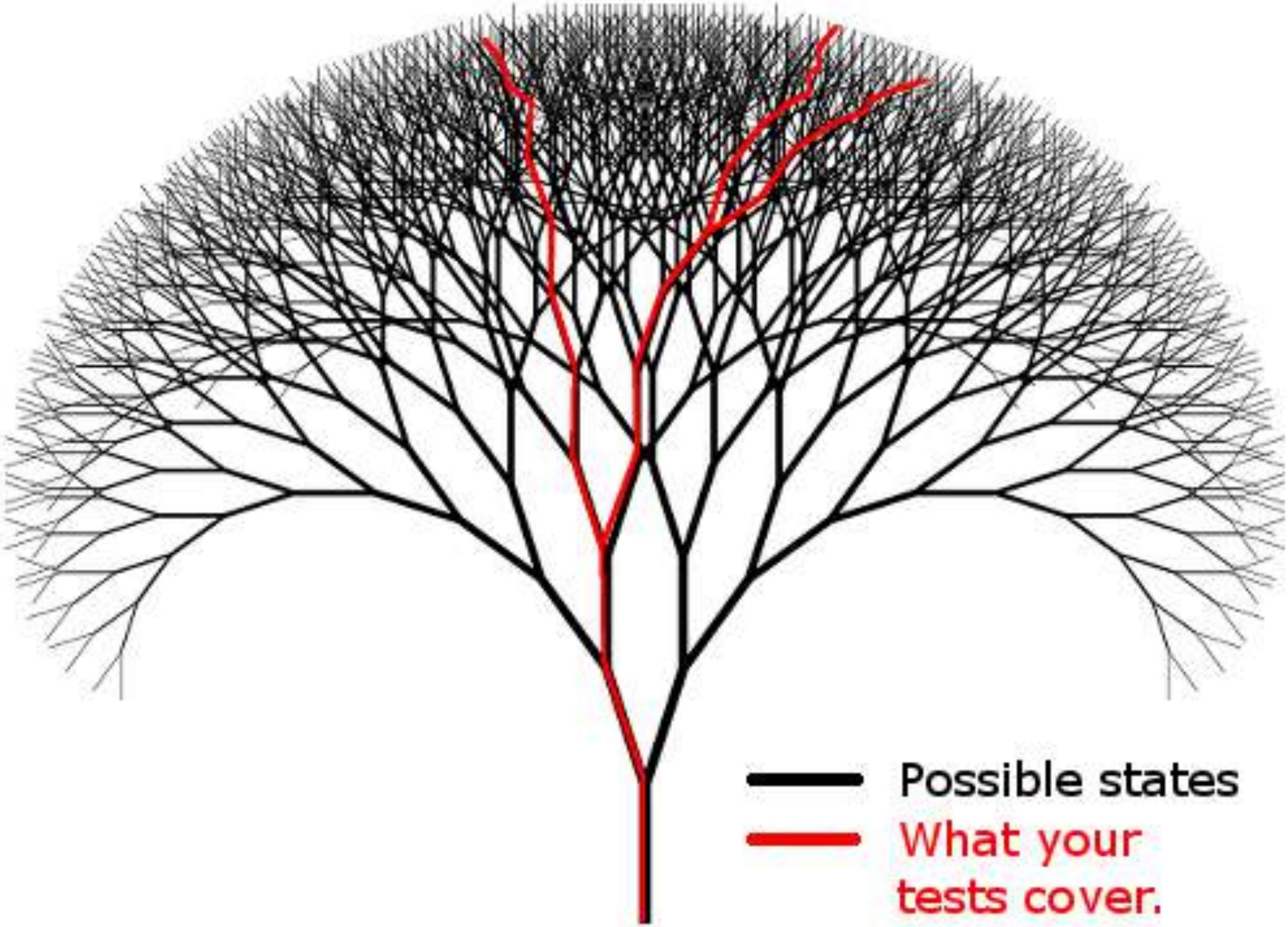


Ruby

... Or a String or BlogPost model. Or an ActionDispatch::Routing::Mapper.

```
def increment(a)
  a + 1
end
```




Possible states
What your tests cover.

So you hear this.

We can fix this!

We'll use duck typing!

Duck Typing!

We'll check that "a" has something that pays attention to +()!

Duck Typing!

```
def increment(a)
  if !a.respond_to?(:+)
    raise TypeError, "yeah nah"
  end
  a + 1
end
```

Duck Typing is a fib. Names are great but they don't tell you shit about what the method is doing.

Pass it something that doesn't behave or takes other args, and kaboom. Go has a stronger method; same problem. Even PHP does it slightly better with named interfaces that classes specifically have to implement.

Duck Typing!

What is "a"?

So let's consider this "a". What does NopeNopeNope do when you add a number to it?

```
def increment(a)
  raise TypeError, "Nope" unless a.respond_to?(:+)
  a + 1
end

class NopeNopeNope < NukeControl
  def +(a)
    fire_ze_missiles!
  end
end

increment(NopeNopeNope.new)
```

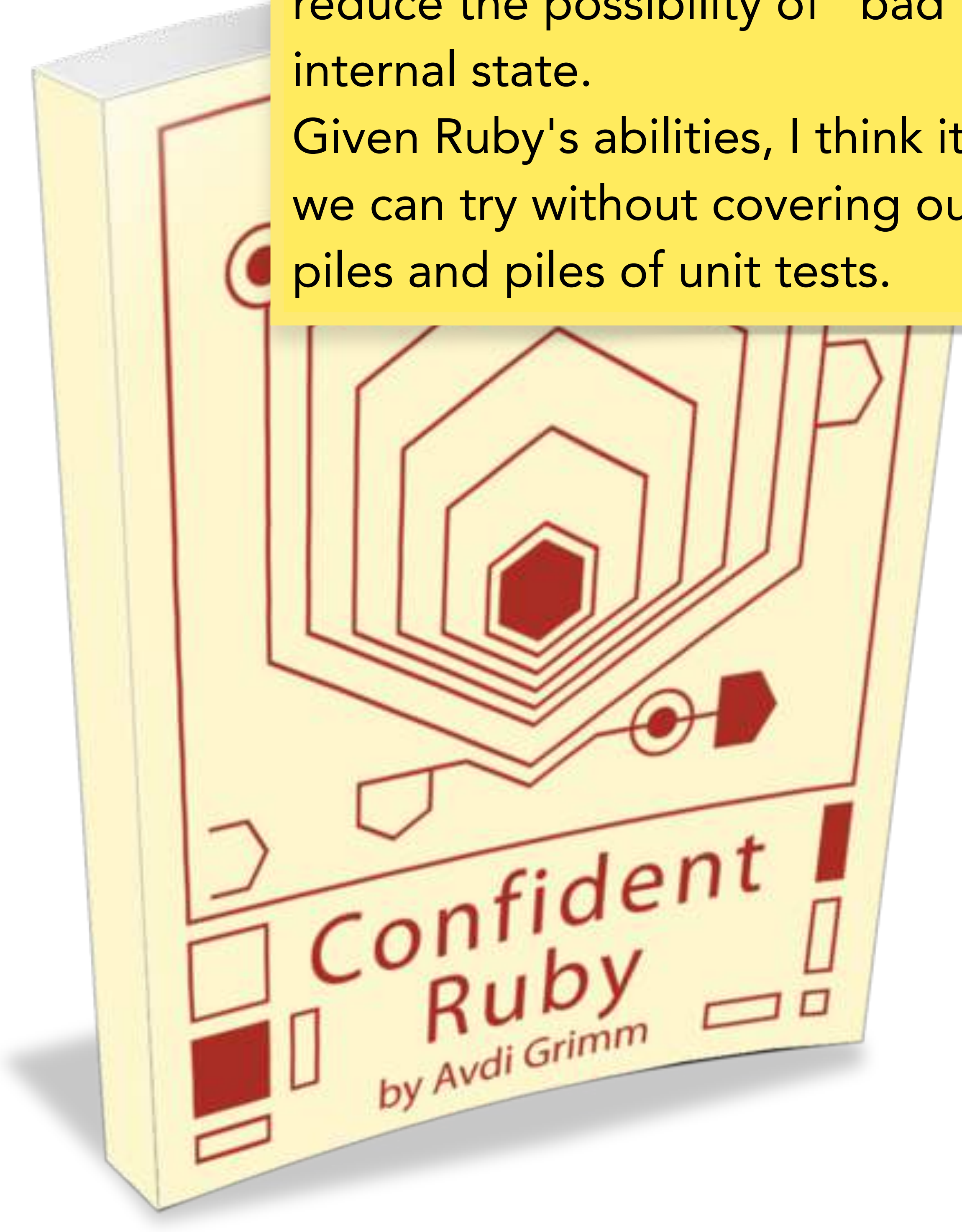
What is "a"?

```
def increment(a)
  raise TypeError, "Nope" unless a.respond_to?(:+)
  a + 1
end

class NopeNopeNop
  def +(a)
    fire_ze_missi
  end
end

increment(NopeNop
```

Explicit Checks...?

So maybe we should do this everywhere.
But suddenly the intent of our code is obscured by checking like this.

```
def increment(a)
  if a.class != Integer
    raise TypeError, "Nope"
  end
  a.+(1)
end
```



Avdi Grimm has a book, Confident Ruby, that proposes "strong borders". At the edges of your program's or library's interface, you be as strict as you can, and to reduce the possibility of "bad" input messing with the internal state.

Given Ruby's abilities, I think it's one of the few methods we can try without covering our code in type checks and piles and piles of unit tests.

but christ it makes me sad thinking about it

Detour Time

It's a long one. Bring some lunch.

Not Ruby

```
increment :: Int -> Int
increment a = …?
```

What could "a" be in this example?

Not Ruby

```
increment :: Int -> Int
increment a = a + 1
```

We're restricted in the functions we can use with "a" and 1. Only Ints. No nulls/nils, or strings, or Routing Model Rails Thinger Thing.

And yes, this could be a - 1 (and be wrong; we'll be coming back to this later).

Not Ruby

And when I say "can't", I mean "the compiler will refuse to produce a binary because it thinks your program is broken."

```
increment :: Int -> Int
increment a = a + 1


…


increment 1         -- Compiles!
increment "Nope"    -- Kaboom
```

Not Rub

This "checking" extends further.

map() is a function that takes a function that takes thing A and thing B (a -> b), and a list of As to turn into a list of Bs.

```
increment :: Int -> Int
increment a = a + 1
```

…

```
map increment [1,2,3]   -- [2,3,4]
map increment ["a","b"] -- Kaboom
```

More Not-Ruby

```
data LogLevel = Info | Error | Warning

data LogMessage = LogMessage {
  level   :: LogLevel,
  message :: String
}
```

We're defining a type LogLevel here, which is either an Info, Error or Warning. Error is representing something – think of it like you do symbols; they don't have a "value" in themselves.

And then we have a LogMessage, which has a level of type LogLevel, and a string.

More Not-F

And a function hasErrors.

[Explanation ensues. This example uses functions named similar to Ruby equivalents. I'll use foldr next time, I swear.]

```
data LogLevel = Info | Error | War

data LogMessage = LogMessage {
  level   :: LogLevel,
  message :: String
}

hasErrors :: [LogMessage] -> Bool
hasErrors logs = length (filter isError logs) > 0
  where
    isError (LogMessage { level = Error }) = True
    isError _                              = False
```

The same code in Ruby...!

Ruby

```
def has_errors(logs)
  logs.any? { |log|
    log.level == LogMessage::Error
  }
end
```

Ruby

```
def has_errors(logs)
  if !logs.is_a?(Enumerable)
    raise TypeError, "Not a list"
  end
  logs.any? { |log|
    if !log.is_a?(LogMessage)
      raise TypeError, "Not a Log"
    end
    log.level == LogMessage::Error
  }
end
```

Well, no. We'd need to do all this to do the same checks in Ruby.

And we'd **still** have to run the code to check it, and run it with a bunch of different inputs, and hope we got enough representative cases.

we need to go deeper

Even More Not-Ruby

```
parseLogLines :: String -> [LogMessage]
parseLogLines x = ...
```

This takes a list of Strings and produces a list of LogMessages, our type from earlier.

Even More Not-Ruby

```
parseLogLines :: String -> [LogMessage]
parseLogLines x = ...

readLog :: (String -> [LogMessage])
        -> FilePath
        -> IO [LogMessages]
readLog parse file = ...
```

And a readLog function that **takes a function that takes a string and produces a list of LogMessages**, a file to look at, and produces a list of LogMessages as the result of IO.

Note, this function **could** fire the missiles while giving me log messages. When we section code off that talks to the outside world we don't have to consider anymore that **anything** could do so.

Even More Not-Ruby

```
data Maybe a = Just a | Nothing

parseLogLine :: String -> Maybe LogMessage
parseLogLine line = ...
```

We could have a type here that represents having a thing (of any type, we don't care), or nothing. This is part of the standard library, but you can easily make your own.

And here, it's representing the possibility of failure; the log line might be invalid, so we might get back a useful log or we might back nothing. Anything using this function will be forced (by the compiler) to consider the possibility of failure in advance.

Even More Not-Ruby

```
data Either a b = Left a | Right b

parseLogLine :: String
             -> Either ParsingError LogMessage
parseLogLine line = ...
```

We have a similar thing here; parseLogLine can return Either a ParsingError (a type we'd define, just like LogMessage), or a LogMessage.

This is being used here as failure-with-more-context.

Even More Not-Ruby

```
parseLogLine :: String
             -> Maybe LogMessageWithOrigin
parseLogLine log = do
  origin  <- parseOrigin message
  message <- parseMessage origin message
  return (LogMessageWithOrigin origin message)
```

Or say we have a different LogMessage type that will need different message parsing depending on the origin of the message, and we need to drop out early if we can't figure out the origin.

[Brief Maybe, Monad, and patterns-except-with-laws-you-can-actually-test explanation follows.]

Even More Not-Ruby

```
fetchAuthorWithPosts :: AuthorId
                     -> IO (Maybe (Author,[Post]))
fetchAuthorWithPosts id = runMaybeT $ do
  author <- MaybeT $ fetchAuthor id
  posts  <- MaybeT $ fetchPosts (map postId author)
  return (author,posts)
```

["and we can keep building top of these pieces while having guarantees about how they work" hand-waving because this is a short talk. And I've reached the extent of what I can pretend I know.]

Even More Not-Ruby

```
fetch :: [Url] -> IO [Maybe String]
fetch pages = mapConcurrently getURL pages

-- ...
fetch ["http://example.com/shovel",
       "http://example.com/spade"]
```

[We're now breezing through "examples built on dependable building blocks" because this talk is short.]

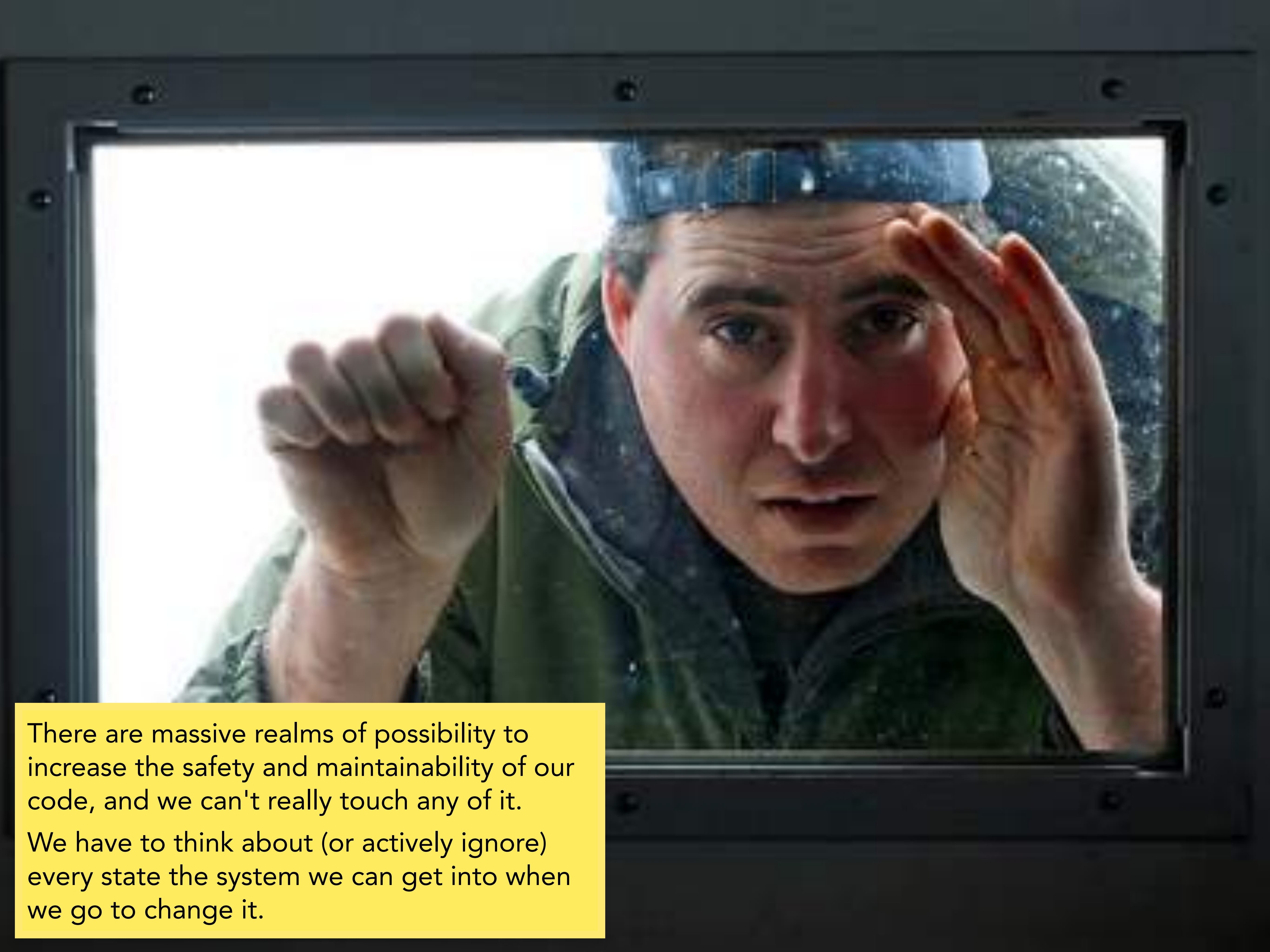
Last Bit of Not-Ruby

```
increment :: Num n => n -> n
increment a = a + 1
```

And back to increment. We say increment :: Int -> Int before. We're generalising now.

We're saying that, for any n (like an Int, or a Float, or Your Own Custom Type Here) that has a bunch of functions defined for it matching a Num "interface", we can give it (and 1) to +.

It allows us someone using this code later with their **own** types to use our functions by implementing that interface for their own types.



There are massive realms of possibility to increase the safety and maintainability of our code, and we can't really touch any of it.

We have to think about (or actively ignore) every state the system we can get into when we go to change it.

Well. It's not looking good, but...

What can we fix?

Or borrow. Or steal.

A Safer Subset...?

The DiamondBack project:

http://www.cs.umd.edu/projects/PL/druby/

We could try a subset of Ruby without some of the crazy bits that make it nightmarish to statically analyse. The DiamondBack approach tries this, ...

A Safer Subs

... adding Inference, explicit type annotation when necessary, dynamic checking for things that can't be statically checked or modified to be statically checked, and metaprogramming support for handling respond_to?().

The DiamondBack project:

http://www.cs.umd.edu/projects/PL/druby/

- Type inference
- Type annotations
- Dynamic checking
- Metaprogramming support

A Safer Subset…?

The DiamondBack project:

http://www.cs.umd.edu/projects/PL/druby/

Abandoned in 2009. 😥

I'm genuinely sad about this.

A Safer Subset...?

The DiamondBack project:

http://www.cs.umd.edu/projects/PL/druby/

Abandoned in 2009. 😥

It's basically not Ruby anymore.

The big problem is that it's basically not Ruby anymore. You lose most of the ecosystem. If you get really lucky you could have a RubyMotion-like community, but I fear that'd need the iOS-like impetus to get that going.

Complete Fork?

Crystal is a Ruby fork with compilation and static typing.

It started as an interpreter fork, but it's very much "Ruby-inspired syntax" now:

http://crystal-lang.org/2013/11/14/good-bye-ruby-thursday.html

Complete Fork?

Definitely not Ruby anymore.

Also, again, a subset of the crazier (read: "wildly unsafe") features Ruby gives you access to.

"Gradual" Typing...?

PHP (!) now has this in the form of Facebook's Hack/HHVM:

http://docs.hhvm.com/manual/en/hack.annotations.php

Facebook has basically forked PHP to add optional typing with Hack.

"Gradual" Typing...?

Allows older only-verifiable-at-run-time PHP to be run with verified-at-compilation Hack in the same program.

Existing libraries (that don't rely on C extensions) work. Existing code works. New code is checked.

"Gradual" Typing...?

```
<?hh
class MyClass {
  const int MyConst = 0;

  private string $x = '';

  public function increment(int $x): int {
    $y = $x + 1;
    return $y;
  }

  public function addLater(int $x): (function(int): int) {
    return function($y) use ($x) {
      return $x + $y;
    };
  }
}
```

PHP is much more fixed than Ruby, sadly. This is actually a benefit here; it's not possible to add or override methods or re-open classes at runtime.

"Gradual" Typing...?

```
<?hh
class MyClass {
  const int MyConst = 0;

  private string $x = '';

  public function increment(int $x): int {
    $y = $x + 1;
    return $y;
  }

  public function addLater(int $x): (function(int): int) {
    return function($y) use ($x) {
      return $x + $y;
    };
  }
}
```

And although the above is really encouraging (look! you can tell it to expect a function as a return value!), it requires you to be very verbose, despite Hack's claim of Type Inference. Remember those previous "Not Ruby" examples with no mentions of types?

"Gradual" Typing...?

Facebook is also doing the same kind of thing with Flow, a JavaScript type-checker you explicitly turn on for chunks of code:

http://flowtype.org/

QuickCheck...?

Let's say we forgot the whole type thing; what about making tests better?

QuickCheck is used for stating an invariant, and then throwing a bunch of test data at it automatically, eg.

State a rule, generate **lots** test data based on the types functions expect, check that the function satisfies the rule.

Types can help **reduce** what we need to check with our tests (and therefore the number of tests), but we still need them.

QuickCheck...?

```
prop_increments c = increment c == c + 1
```

This a dumb example. It's checking that, whenever we give a number to increment, we always get back that number plus one.

But! **Our original code has a bug.**

`increment (maxBound :: Int)` gives us -9223372036854775808; this would help expose that bug.

QuickCheck...?

```
prop_increments c = increment c == c + 1

# Rantly
test "increments" do
  property_of { integer }.check { |i|
    assert_equal(increment(i), i + 1)
  }
end
```

We have an attempt to reproduce some of this in Ruby with Rantly.
Without types it's an uphill slog, though.
[test data generation ramble follows]

QuickCheck...?

```
prop_join_split xs = forAll (elements xs) check
  where
    check c = join c (split c xs) == xs

prop_insert x xs =
  ordered xs ==> ordered (insert x xs)
```

... for example we're using quickcheck here to test if splitting a list of things and joining them back together produces the original (the example this was drawn from had an edge case where it'd sometimes lose items) ... the second is checking that a list stays ordered when added to ...

"Soft Typing"?

Matz just mentioned something about a kind of "soft typing". Very hazy, but something to watch for later:

https://www.omniref.com/blog/blog/2014/11/17/matz-at-rubyconf-2014-will-ruby-3-dot-0-be-statically-typed/

What can't we fix?

sad-kid-frown.gif

Sad Frowning

- Without a restricting ourselves to a stricter subset of the language (eg. sans the crazy meta-programming), we are not able to look at code before running it and know how it's doing to behave.
- Without restricting behaviour, we can't make guarantees about what our code will do.
- Without doing this, as our apps getter larger, we have to write exponentially more tests and conditionals to check, or they get broken, buggy and expensive to fix.

BACK TO THE TITLE

RUBY'S

OUTER LIMITS

Or: "Why Ruby can be frustrating to use when writing medium/large-ish apps."

You may be thinking I'm advocating for this.

[STTNG clip, Picard yelling "All hands, abandon ship!" before the Enterprise blows up.]

Ruby Might Possibly be "Doomed"

- Not in the "going to die out, unpopular language, no paid work" sense.
- Not in the "not ever going to change, not going to evolve" sense.
- More that improvement is approaching a maxima that cannot be broken through without radically altering the language and breaking backwards compatibility.
- Our tools are failing us when used for largeish projects.

... But my previous Doomed title may be a /slight/ over-dramatisation. [Reads conclusion off slides.]

Fin.

Credits

Title slide photo © Ozroads:
www.ozroads.com.au/NSW/Highways/Pacific/heronscreek.htm

Rob Howard
@damncabbage
https://speakerdeck.com/damncabbage/

Fin.

Credits

Title slide photo © Ozroads:
www.ozroads.com.au/NSW/Highways/Pacific/heronscreek.htm

Rob Howard
@damncabbage
https://speakerdeck.com/damncabbage/