SuperGood

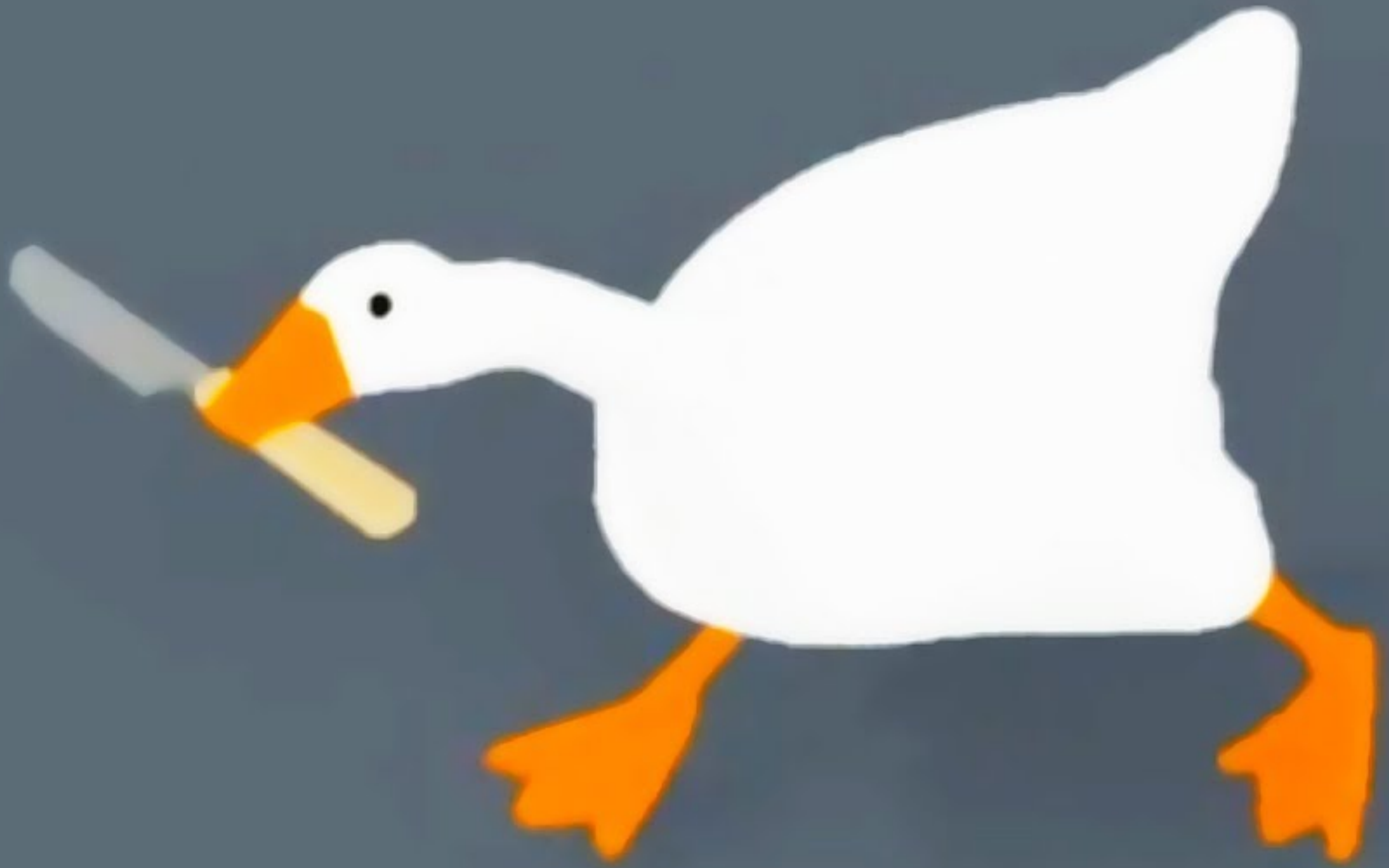
TDD on the Shoulders of Giants

Or Growing Object-Oriented Ruby, Guided by Jared

Jared Norman	jardo.dev	RubyConf Mini 2022

Growing Object-Oriented Software, Guided by Tests

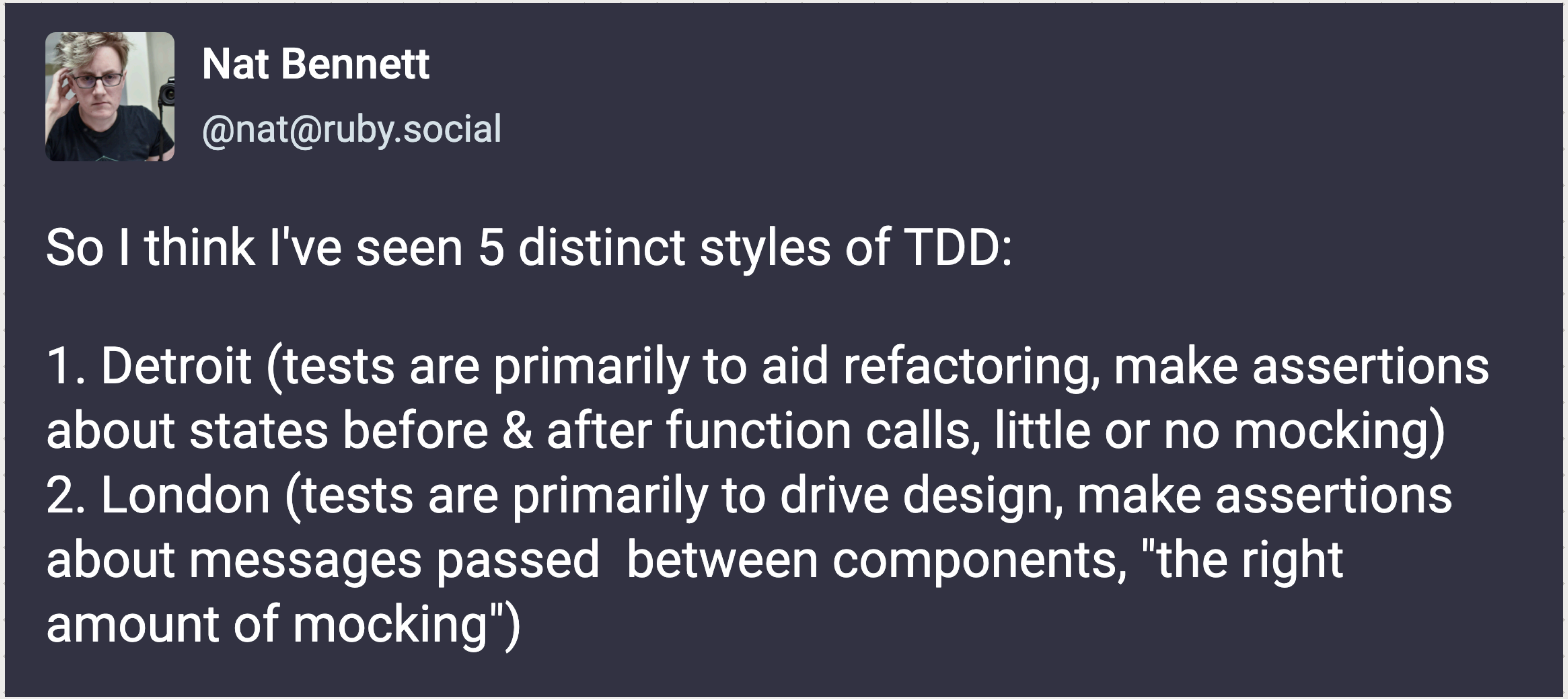
Growing Object-Oriented Software, Guided by Tests



Ruby != Java

Will the real Test-Driven Development please stand up

Red, Green, Refactor



Nat Bennett

@nat@ruby.social

So I think I've seen 5 distinct styles of TDD:

1. Detroit (tests are primarily to aid refactoring, make assertions about states before & after function calls, little or no mocking)
2. London (tests are primarily to drive design, make assertions about messages passed between components, "the right amount of mocking")

Red, (Red, Green, Refactor), Green

- Write a failing acceptance test
- Repeatedly:
    - Write a failing test
    - Make the test pass
    - Refactor the code
- Now the acceptance test passes

Acceptance Tests

Integration Tests

The Golden Rule

"Never write new functionality without a failing test."

–GOOS

London-style Test-Driven Development

— Outside-in

— Use acceptance tests to drive features

— Use unit tests to drive design

— Focus on message passing

— Mock collaborating objects in unit tests

— Create functionality using composition

Mock interfaces, not Classes...

```ruby
class Shipment
  # ...

  def ship!(email_notifier)
    # do shipping stuff

    email_notifier.send_notification(
      shipment_info
    )
  end
end
```

```ruby
class Shipment
  # ...

  def ship!(shipment_notifier)
    # do shipping stuff

    shipment_notifier.send_notification(
      shipment_info
    )
  end
end
```

Think in interfaces

Composition

Do the composition at a level where you don't control the instantiation of the object

Don't unit test the composition layer

Working with frameworks

Don't fight the framework

Arrange, Act, Assert

Where am I?

What's going on?

Why does it smell like that?

```ruby
let(:shipment_notifier) { double("shipment notifier") }

before do
  expect(shipment_notifier)
    .to receive(:send_notification)
    .with(...)
end

it "notifies the customer" do
  shipment.ship!(shipment_notifier)
end

# More tests...
```

```ruby
let(:shipment_notifier) {
  double("shipment notifier", send_notification: nil)
}

it "notifies the customer" do
  expect(shipment_notifier)
    .to receive(:send_notification)
    .with(...)

  shipment.ship!(shipment_notifier)
end

# More tests...
```

```ruby
let(:shipment_notifier) {
  double("shipment notifier", send_notification: nil)
}

it "notifies the customer" do
  shipment.ship!(shipment_notifier)

  expect(shipment_notifier)
    .to have_received(:send_notification)
    .with(...)
end

# More tests ...
```

The importance of values

"foo"

312

false

```ruby
expect(subject).to eq 312
```

```
Money.from_cents(1000, "USD")
Money.from_cents(1000, "USD")
  == Money.from_cents(1000, "USD")
=> true
```

```
expect(mix(red, blue)).to eq purple
```

```ruby
Colour = Struct.new(:r, :g, :b)
```

🇨🇦

```
Colour.new(255, 255, 255)
  == Colour.new(255, 255, 255)
=> true
```

```ruby
Colour = Struct.new(:r, :g, :b) do
  def darken(percentage = 0.9)
    Colour.new(
      r * percentage,
      g * percentage,
      b * percentage
    )
  end
end
```

```ruby
blue = Colour.new(0, 0, 255)
red = Colour.new(255, 0, 0)

blue.r = 255
blue.b = 0

blue == red
#=> true
```

```ruby
Colour = Data.define(:r, :g, :b)
Colour = Data.define(:r, :g, :b)

green = Colour.new(0, 255, 0)
red = Colour.new(r: 255, g: 0, b: 0)
```

```ruby
Colour = Data.define(:r, :g, :b)

green = Colour.new(0, 255, 0)
red = Colour.new(r: 255, g: 0, b: 0)

red == green
#=> false

red == Colour.new(255, 0, 0)
#=> true
```

```
red.b = 255
#=> NoMethodError
```

Watch for clusters of data

Don't mock value objects

```ruby
FactoryBot.define do
  factory :colour do
    r { 255 }
    g { 255 }
    b { 255 }

    trait :red do
      g { 255 }
      b { 255 }
    end

    # etc
  end
end

let(:red) { build :colour, :red }
```

The Power of RSpec

```ruby
allow(SomeModule).to receive(:foo).and_return(3)
allow(OtherModule).to receive(:baz?).and_return(true)

allow(SomeClass).to receive(:new).with(3, true).and_return(
  double("something", bar: nil, baz: :stuff)
)

allow_any_instance_of(OtherClass)
  .to receive(:update)
  .and_return([:other, :stuff])

# more mocking and stubbing...

# eventually some tests
```

No cheating*

No cheating*
*unless you have to

Why write tests?

Tests are designed to fail

```ruby
expect(subject).to eq("some string"),
  "[explanation of what causes this failure]"
```

Making tests more valuable

— Custom failure messages

— Custom matchers

— Use accurate matchers

— Make liberal expectations

— Don't over-assert

— Write focused tests

— Use descriptive names

— Focus on readability

In conclusion

— Think about the interfaces of your objects

— Avoid over-testing your controllers

— Embrace your chose framework

— Arrange, Act, Assert

— Uncover value objects

— Use RSpec for good

— Design tests to fail

Twitter:
  @jardonamron

Mastodon:
  @jared@ruby.social

Web:
  https://jardo.dev

Super Good:
  https://supergood.software

Thanks!

Links to stuff

— The Book: http://www.growing-object-oriented-software.com/

— Data Proposal: https://bugs.ruby-lang.org/issues/16122

— Data PR: https://github.com/ruby/ruby/pull/6353

— My Twitter: https://twitter.com/jardonamron

— My Mastodon: https://ruby.social/@jared

— My website: https://jardo.dev

— Super Good: https://supergood.software