Breaking up (with) your test suite
My name is Justin Searls
Please tweet me @searls &
Say hello@testdouble.com
Any questions?
Tweet @searls during the talk
Favorite question gets a book!
Any questions?
Tweet @searls during the talk
Favorite question gets a book! 💖
This talk is about
This talk is about relationships.
I love my customers
I ❤️ coding
I ❤️ testing
We don't test for testing's sake
I get paid for code that works, not for tests, so my philosophy is to test as little as possible to reach a given level of confidence.

- Kent Beck
We don't code for code's sake, either
Writing code is a costly way to solve problems
Too often, we cheat on our customers with our code
How to cheat on your customers:
How to cheat on your customers:

1. Uncritically assume your favorite way to write code applies to every situation
How to cheat on your customers:

1. Uncritically assume your favorite way to write code applies to every situation

2. There is no #2
Cargo-culting as a service
We must challenge our biases in each new context to grow
infatuation
Remember TATFT?
TDD literally rescued me from the pit of despair.
6 months later
Why does every refactor break 6 tests?!
Why does the build take nine hours to run?!
Why are people mocking methods on the object they're testing?!
Why can't we get anything done anymore?
and eventually
the break-up
the break-up
the break-up
the break-up
introspection
Why test?
know that my code works
know that my code works

safely change & refactor
know that my code works

keep the product simple

safely change & refactor
keep the product simple

make public API easy-to-use

know that my code works

safely change & refactor
make public API easy-to-use
know that my code works
grow a maintainable private API
safely change & refactor
keep the product simple
make public API easy-to-use

know that my code works

grow a maintainable private API

ensure behavior of others' services

keep the product simple

safely change & refactor
verify that production is working
keep the product simple
make public API easy-to-use
grow a maintainable private API
know that my code works
ensure behavior of others' services
safely change & refactor
verify that production is working
keep the product simple
make public API easy-to-use
grow a maintainable private API
know that my code works
ensure behavior of others' services
safely change & refactor
safeguard our use of 3rd-party code
validate our code is useful to others
verify that production is working
keep the product simple
make public API easy-to-use
grow a maintainable private API
know that my code works
ensure behavior of others’ services
safely change & refactor
safeguard our use of 3rd-party code
validate our code is useful to others

verify that production is working

keep the product simple

make public API easy-to-use

grow a maintainable private API

narrowly specify our dependencies

know that my code works

ensure behavior of others’ services

safeguard our use of 3rd-party code

safely change & refactor
validate our code is useful to others

make public API easy-to-use

know that my code works

grow a maintainable private API

ensure behavior of others’ services

verify that production is working

narrowly specify our dependencies

safely change & refactor

keep the product simple

safeguard our use of 3rd-party code
Confidence

- Validate our code is useful to others
- Make public API easy-to-use
- Know that my code works
- Ensure behavior of others' services
- Safely change & refactor
- Verify that production is working
- Grow a maintainable private API
- Narrowly specify our dependencies
- Safeguard our use of 3rd-party code
- Keep the product simple
validate our code is useful to others
make public API easy-to-use
know that my code works
grow a maintainable private API
ensure behavior of others' services
narrowly specify our dependencies
safely change & refactor
verify that production is working
keep the product simple
safeguard our use of 3rd-party code
Understanding

- validate our code is useful to others
- make public API easy-to-use
- know that my code works
- ensure behavior of others' services
- safely change & refactor
- verify that production is working
- grow a maintainable private API
- narrowly specify our dependencies
- safeguard our use of 3rd-party code
- keep the product simple
Single Responsibility Testing
describe "Panda" do
end
describe "Panda" do
  let(:bamboo) { double("bamboo").as_null_object }
end
describe "Panda" do
  let(:bamboo) { double("bamboo").as_null_object }
  subject { Panda.new(bamboo) }
end
describe "Panda" do
  let(:bamboo) { double("bamboo").as_null_object }
  subject { Panda.new(bamboo) }

  it "munches bamboo" do
    # Test code here
  end
end
describe "Panda" do
  let(:bamboo) { double("bamboo").as_null_object }
  subject { Panda.new(bamboo) }

  it "munches bamboo" do
    subject.eat!
  end
end
describe "Panda" do
  let(:bamboo) { double("bamboo").as_null_object }
  subject { Panda.new(bamboo) }

  it "munches bamboo" do
    subject.eat!
    expect(bamboo).to have_received(:munch)
  end
end
describe "Panda" do
  let(:bamboo) { double("bamboo").as_null_object }
  subject { Panda.new(bamboo) }

  it "munches bamboo" do
    subject.eat!
    expect(bamboo).to have_received(:munch)
  end
end
later, fixing a bug
describe "Panda" do
  let(:bamboo) { double("bamboo").as_null_object }
  subject { Panda.new(bamboo) }

  it "munches bamboo" do
    subject.eat!
    expect(bamboo).to have_received(:munch)
  end
end
describe "Panda" do
  let(:bamboo) { Bamboo.find(34) }
  subject { Panda.new(bamboo) }

  it "munches bamboo" do
    subject.eat!
    expect(bamboo).to have_received(:munch)
  end
end
describe "Panda" do
  let(:bamboo) { Bamboo.find(34) }
  subject { Panda.new(bamboo) }

  it "munches bamboo" do
    allow(bamboo).to receive(:munch)
    subject.eat!
    expect(bamboo).to have_received(:munch)
  end
end
describe "Panda" do
  let(:bamboo) { Bamboo.find(34) }
  subject { Panda.new(bamboo) }

  it "munches bamboo" do
    allow(bamboo).to receive(:munch)
    subject.eat!
    expect(bamboo).to have_received(:munch)
  end

  it "returns bamboo id" do
  end
end
describe "Panda" do
  let(:bamboo) { Bamboo.find(34) }
  subject { Panda.new(bamboo) }

  it "munches bamboo" do
    allow(bamboo).to receive(:munch)
    subject.eat!
    expect(bamboo).to have_received(:munch)
  end

  it "returns bamboo id" do
    expect(subject.eat!).to eq(34)
  end
end
describe "Panda" do
  let(:bamboo) { Bamboo.find(34) }
  subject { Panda.new(bamboo) }

  it "munches bamboo" do
    allow(bamboo).to receive(:munch)
    subject.eat!
    expect(bamboo).to have_received(:munch)
  end

  it "returns bamboo id" do
    expect(subject.eat!).to eq(34)
  end
end
one day, I find it
describe "Panda" do
  let(:bamboo) { Bamboo.find(34) }
  subject { Panda.new(bamboo) }

  it "munches bamboo" do
    allow(bamboo).to receive(:munch)
    subject.eat!
    expect(bamboo).to have_received(:munch)
  end

  it "returns bamboo id" do
    expect(subject.eat!).to eq(34)
  end
end
describe "Panda" do
  let(:bamboo) { Bamboo.find(34) }
  subject { Panda.new(bamboo) }

  it "munches bamboo" do
    allow(bamboo).to receive(:munch)
    subject.eat!
    expect(bamboo).to have_received(:munch)
  end

  it "returns bamboo id" do
    expect(subject.eat!).to eq(34)
  end
end
describe "Panda" do
  let(:bamboo) { Bamboo.find(34) }
  subject { Panda.new(bamboo) }

  it "munches bamboo" do
    allow(bamboo).to receive(:munch)
    subject.eat!
    expect(bamboo).to have_received(:munch)
  end

  it "returns bamboo id" do
    expect(subject.eat!).to eq(34)
  end
end
describe "Panda" do
  let(:bamboo) { Bamboo.find(34) }
  subject { Panda.new(bamboo) }

  it "munches bamboo" do
    allow(bamboo).to receive(:munch)
    subject.eat!
    expect(bamboo).to have_received(:munch)
  end

  it "returns bamboo id" do
    expect(subject.eat!).to eq(34)
  end
end
describe "Panda" do
  let(:bamboo) { Bamboo.find(34) }
  subject { Panda.new(bamboo) }

  it "munches bamboo" do
    allow(bamboo).to receive(:munch)
    subject.eat!
    expect(bamboo).to have_received(:munch)
  end

  it "returns bamboo id" do
    expect(subject.eat!).to eq(34)
  end
end

Why does this test exist?
What type of test is this?
Why does this test exist?

What type of test is this?

How should tests be written?
We should know these answers before opening the test!
The purpose, rules, and structure of each test should be immediately clear.
A single test can't address all of our motivations
Such tests are unclear:
Such tests are unclear:

- purpose is hazy
Such tests are unclear:

- purpose is hazy
- rules are debatable
Such tests are unclear:

- purpose is hazy
- rules are debatable
- structure is ad hoc
Unclear tests cost money
Unclear tests must constantly have their:
Unclear tests must constantly have their:

- purpose rediscovered
Unclear tests must constantly have their:

- purpose rediscovered
- rules renegotiated
Unclear tests must constantly have their:

- purpose rediscovered
- rules renegotiated
- structure reorganized
None of those activities have value!
Every suite should promote at most ___ type of confidence
Every suite should promote at most \underline{1} type of confidence
Every suite should promote at most ___ type of understanding
Every suite should promote at most \_\_\_ type of understanding
suite design
Omakase Rails
Omakase Rails

MiniTest
Omakase Rails

Views

MiniTest
Omakase Rails

Views

MiniTest

MiniTest
Omakase Rails

Views

Controllers

MiniTest

MiniTest
Omakase Rails

- Views
- Controllers
- MiniTest
- MiniTest
- MiniTest
Omakase Rails

Views  MiniTest
Controllers  MiniTest
Models  MiniTest
Omakase Rails

Views

Controllers

Models

MiniTest

MiniTest

MiniTest

MiniTest
Prime Rails
Prime Rails

Cucumber
Prime Rails

UI  Cucumber
Prime Rails

UI

Cucumber

RSpec
Prime Rails

UI
Cucumber
Views
RSpec
Prime Rails

UI
Cucumber
Views
RSpec
RSpec
Prime Rails

UI
Cucumber
Views
RSpec
Controllers
RSpec
Prime Rails

UI

Cucumber

Views

RSpec

Controllers

RSpec

Models

RSpec
"Fast Specs" Rails

- UI
- Views
- Controllers
- Models

- Cucumber
- RSpec
- RSpec
- RSpec
"Fast Specs" Rails

- RSpec
- Controllers
  - RSpec
- Models
  - RSpec
- Views
  - RSpec
- UI
  - Cucumber
"Fast Specs" Rails

- RSpec
- POROs
- Controllers
  - Views
  - Models
- UI
- Cucumber
- RSpec
- RSpec
- RSpec
What's the problem?
Redundant test coverage
One model change

UI
Views
Controllers
Models

Cucumber
RSpec
RSpec
RSpec
One model change

- UI
- Views
- Controllers
- Models

- Cucumber
- RSpec
- RSpec
- RSpec
One model change
One model change

Models

Controllers

Views

UI

RSpec

RSpec

RSpec

RSpec

RSpec

Cucumber
One model change

UI

Views

Controllers

Models

Cucumber

RSpec

RSpec

RSpec

RSpec
One model change

- UI
- Views
- Controllers
- Models

- Cucumber
- RSpec
- RSpec
- RSpec
One model change

UI

Views

Controllers

Models

Cucumber

RSpec

RSpec

RSpec
Which leads to:
architecture
router

web app #1

web app #2

walrus data

panda data

cage control service
web app #1

web front-end

web back-end

router

walrus data

panda data

cage control service
class ProvidesMeal
  def initialize...

  def provide(plan)
    @releases_food.release(
      @computes_portion.compute(plan)
    )
  end
end

suite design
class ProvidesMeal
  def initialize...
  end

  def provide(plan)
    @releases_food.release(
      @computes_portion.compute(plan)
    )
  end
end
web app #1

web front-end

web back-end

router
cage control service
walrus data
panda data
router

web app #1

web app #2

walrus data

panda data

cage control service
The greatest test of all:
Does our software serve its purpose?
Wait... what is its purpose?
Generating revenue?
Cutting costs?
Limiting our liability?
Funding non-profits?
Can we write a test for those outcomes?
Maybe!*
Maybe!*  

*But not in this talk.
Smoke tests
Acceptance tests
Feature tests
End-to-end tests
Integration tests
Integration tests
Smoke Acceptance Feature End-to-end
Smoke
Acceptance
Feature
End-to-end
SAFE tests
The test's "user"
Confidence sought
Understanding gained
Guidelines
Warning Signs
app
A real-world user
web apps
http services

$ curl http://
Application works when it's all glued together
"How simple is our product?"
Can't fit your tests in 30 minutes?
Can't fit your tests in 30 minutes?
The app is complex
Can't write a test in 30 minutes?
Can't write a test in 30 minutes?

The app is complicated
SAFE tests shouldn't know internal APIs
Web tests should bind to user-visible text, not markup
Enforce a fixed-time budget
Time-box the suite

5 minutes

5 minutes
<table>
<thead>
<tr>
<th>5 minutes</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>monthly</td>
<td></td>
<td></td>
</tr>
<tr>
<td>financials</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time-box the suite
Time-box the suite

5 minutes

monthly financials

search

5 minutes
Time-box the suite

- monthly financials
- process order
- search

5 minutes
**Time-box the suite**

<table>
<thead>
<tr>
<th>Monthly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Financials</td>
</tr>
<tr>
<td>Process</td>
</tr>
<tr>
<td>Order</td>
</tr>
<tr>
<td>Ad</td>
</tr>
<tr>
<td>Search</td>
</tr>
<tr>
<td>Words</td>
</tr>
</tbody>
</table>
# Time-box the suite

<table>
<thead>
<tr>
<th></th>
<th>monthly financials</th>
<th>sign-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td></td>
<td></td>
</tr>
<tr>
<td>words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>search</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ad</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5 minutes each section.
Time-box the suite

- monthly financials
- sign-up
- process order
- ad
- search
- words

5 minutes per task
Time-box the suite

5 minutes

- monthly financials
- sign-up

- process order
- ad
- words
- search

5 minutes

daily deals

SAFE
Failures due to refactors are false negatives
Beware: human intuition overvalues "realistic" tests
Building multiple releases & branches is expensive
Superlinear Slowdown
Superlinear Slowdown

More tests → Slower build
Superlinear Slowdown
Superlinear Slowdown

Bigger System → Slower build
validate our code is useful to others

make public API easy-to-use

know that my code works

grow a maintainable private API

ensure behavior of others’ services

verify that production is working

narrowly specify our dependencies

safely change & refactor

keep the product simple

safeguard our use of 3rd-party code
validate our code is useful to others

make public API easy-to-use

know that my code works

grow a maintainable private API

ensure behavior of others' services

verify that production is working

narrowly specify our dependencies

safely change & refactor

keep the product simple

safeguard our use of 3rd-party code
<table>
<thead>
<tr>
<th>Validate our code is useful to others</th>
<th>Make public API easy-to-use</th>
<th>Know that my code works</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grow a maintainable private API</td>
<td>Narrowly specify our dependencies</td>
<td>Ensure behavior of others' services</td>
</tr>
<tr>
<td>Verify that production is working</td>
<td>Safeguard our use of 3rd-party code</td>
<td>Safely change &amp; refactor</td>
</tr>
<tr>
<td>Keep the product simple</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Good SAFE tests are accurate
But they’re also expensive
And they don't inform our public API's design
Every library or service you write is consumed by somebody
Verify behavior & demonstrate usage with Consumption tests
Your _______’s consumer
Your [service]’s consumer
Your ___library___'s consumer
Your user interface's consumer
Your repository's consumer
Verifies behavior you're directly responsible for.
Consumption
Answers: "Is it usable?"
If it's hard to test, then it's hard to use.
Consumption
Module boundaries should be meaningful beyond testing
Fakes all external dependencies
Fakes all external dependencies (i.e. can run without access to other services)
Fakes all external dependencies

(i.e. can run without access to other services)

(e.g. in Travis CI)
Exercises public, not private APIs
Organized by consumers' desired outcomes
Keep consumption tests really fast
faked services → no good reason to be slow
The only suite that says you just broke something
Consumption

- Validate our code is useful to others
- Make public API easy-to-use
- Know that my code works
- Grow a maintainable private API
- Narrowly specify our dependencies
- Ensure behavior of others' services
- Safeguard our use of 3rd-party code
- Safely change & refactor
- Verify that production is working
- Keep the product simple
validate our code is useful to others
make public API easy-to-use
know that my code works
grow a maintainable private API
ensure behavior of others’ services
verify that production is working
narrowly specify our dependencies
safely change & refactor
keep the product simple
safeguard our use of 3rd-party code
validate our code is useful to others
make public API easy-to-use
know that my code works
grow a maintainable private API
ensure behavior of others’ services
verify that production is working
narrowly specify our dependencies
safely change & refactor
keep the product simple
safeguard our use of 3rd-party code
web app #1

web front-end

web back-end

router

walrus data

panda data

cage control service
"If my tests fake out WalrusService how can I trust that GET /walruses still returns walruses?"
Why not test across service boundaries?
Inter-service tests are easy!
Inter-service tests are easy!

1. Start by not writing them
Inter-service tests are easy!

1. Start by not writing them
2. Continue to not write them
Inter-service tests are easy!

1. Start by not writing them
2. Continue to not write them
3. You’re already done. Easy!
Inter-service tests are tempting, but they're __________________.
Inter-service tests are tempting, but they’re hard to set up.
Inter-service tests are tempting, but they're slow to run.
Inter-service tests are tempting, but they’re redundant coverage.
The root question indicates a lack of trust
Default to trusting your depended-on services
... but sometimes trust isn't appropriate!
Sometimes our usage is abnormal or unsupported
Sometimes other teams demonstrate blind-spots
When trust isn't enough, try writing Contract tests
The "user" is us!
Test is written to represent your interests in somebody else’s repo
Your needs become part of their Consumption suite
If they break your test, they know to call you!
Speedier Feedback

fast

slow
Speedier Feedback

Contract

production release

fast

slow
Speedier Feedback

git push

production release

fast

slow
Dependencies that we own behave how we need them to
We learn: “is this service a good fit for us?”
Frequent failures may reveal differing priorities.
The maintainer learns: "are people using our service as we expect (or desire)?"
Maybe this isn't the best service for them
Maybe this isn't the best service, period.
Written just like consumption tests
Committed into the repo of the depended-on service
Maintainers can lower barrier of entry by providing conveniences
Not a replacement for human interaction
Contract

- Validate our code is useful to others
- Make public API easy-to-use
- Know that my code works
- Grow a maintainable private API
- Ensure behavior of others’ services
- Narrowly specify our dependencies
- Safeguard our use of 3rd-party code
- Safely change & refactor
- Verify that production is working
- Keep the product simple
Contrary to popular belief, there is a world outside our code. Our goal should be to make public API easy-to-use, so that others can understand and use our code. We should ensure the behavior of others' services and know that our code works.

We should verify that our production is working and validate our code is useful to others. We should safely change and refactor, and safeguard our use of 3rd-party code.

To maintain a maintainable private API, we should narrow our dependencies. Keep the product simple and grow a maintainable private API.
Contract

- validate our code is useful to others
- make public API easy-to-use
- know that my code works
- grow a maintainable private API
- ensure behavior of others' services
- verify that production is working
- narrowly specify our dependencies
- safely change & refactor
- keep the product simple
- safeguard our use of 3rd-party code
router

web app #1

web app #2

walrus data

panda data

cage control service
class ProvidesMeal
  def initialize...

  def provide(plan)
    @releases_food.release(@computes_portion.compute(plan))
  end
end
What about TDD?
"What's the point?"
"We already have lots of tests!"
So far, no feedback about code's design
You already have a definition for "unit" test
I wrote about this a few months ago.
I wrote about this a few months ago
(just Google "tdd failure")
I'm now halfway through teaching a two-week crash course on "agile development stuff" to a team of very traditional enterprise Java developers. Condensing fifteen years of our community's progress into 8 half-day workshops has presented an obvious challenge: given the clear time constraints, what set of ideas and practices could conceivably have the biggest positive impact on these developers' professional lives?

After a few days of fits and starts, I've come to at least one realization: test-driven development ("TDD") as it's traditionally introduced to beginners is officially off my list.

The problems with how TDD is typically introduced are fundamental, because they put the learner on a path that leads to a destination which might resemble where they want to go, but doesn't actually show them the way to the promised destination itself. This sort of phenomenon happens often enough that I've decided to finally settle on a name: "WTF now, guys?"
It even elicited a response from Uncle Bob!
I will refute his arguments and utterly destroy his conclusions. And then, once I'm done salting the ground where he used to live, ...
I will refute his arguments and utterly destroy his conclusions. And then, once I’m done salting the ground where he used to live, I’ll tell you why I completely agree with him.

- Uncle Bob
There's no "one true TDD"
The principal value of TDD in this model is discovery
Discovery of tiny, boring, consistent units that break big, scary problems down into small, manageable ones.
The first caller of a new method
Concerned primarily with inputs and output (or side effect)
Also concerned with basic code design details
Inputs & Outcome

Feed Zoo Animals
Inputs & Outcome

(walrus, cheese) → Feed Zoo Animals
Inputs & Outcome

(walrus, cheese) \rightarrow \text{Feed Zoo Animals} \rightarrow \text{Walrus has cheese}
Now what?!
describe FeedsAnimals do

end
describe FeedsAnimals do
  describe "#feed" do
  end
end
describe FeedsAnimals do
  describe "#feed" do
    expect(my_face).to look_like(ಠ_ಠ)
  end
end
Code by wishful thinking!

Feed Zoo Animals
Code by wishful thinking!

Discovery
Build Feed Plan
Feed Zoo Animals
Provide Meal

Code by wishful thinking!
The test is our sounding board

Build Feed Plan

Feed Zoo Animals

Provide Meal
describe FeedsAnimals do
end
describe FeedsAnimals do
  Given(:builds_feed_plan) { gimme(BuildsFeedPlan) }
end
describe FeedsAnimals do
  Given(:builds_feed_plan) { gimme(BuildsFeedPlan) }
  Given(:provides_meal) { gimme(ProvidesMeal) }
end
describe FeedsAnimals do
  Given(:builds_feed_plan) { gimme(BuildsFeedPlan) }
  Given(:provides_meal) { gimme(ProvidesMeal) }

  subject { FeedsAnimals.new(builds_feed_plan, provides_meal) }
end
describe FeedsAnimals do
  Given(:builds_feed_plan) { gimme(BuildsFeedPlan) }
  Given(:provides_meal) { gimme(ProvidesMeal) }

  subject { FeedsAnimals.new(builds_feed_plan, provides_meal) }

  describe "#feed" do
  end
end
describe FeedsAnimals do
  Given(:builds_feed_plan) { gimme(BuildsFeedPlan) }
  Given(:provides_meal) { gimme(ProvidesMeal) }

  subject { FeedsAnimals.new(builds_feed_plan, provides_meal) }

  describe "#feed" do

    When(:result) { subject.feed(:walrus, :cheese) }

  end
end
describe FeedsAnimals do
  Given(:builds_feed_plan) { gimme(BuildsFeedPlan) }
  Given(:provides_meal) { gimme(ProvidesMeal) }

  subject { FeedsAnimals.new(builds_feed_plan, provides_meal) }

  describe "#feed" do
    When(:result) { subject.feed(:walrus, :cheese) }
    Then { verify(provides_meal).provide(:cheese, :plan) }
  end
end
describe FeedsAnimals do
  Given(:builds_feed_plan) { gimme(BuildsFeedPlan) }
  Given(:provides_meal) { gimme(ProvidesMeal) }

  subject { FeedsAnimals.new(builds_feed_plan, provides_meal) }

  describe "#feed" do
    Given { give(builds_feed_plan).build(:walrus) {:plan} }
    When(:result) { subject.feed(:walrus, :cheese) }
    Then { verify(provides_meal).provide(:cheese, :plan) }
  end
end
describe FeedsAnimals do
  Given(:builds_feed_plan) { gimme(BuildsFeedPlan) }
  Given(:provides_meal) { gimme(ProvidesMeal) }

  subject { FeedsAnimals.new(builds_feed_plan, provides_meal) }

  describe "#feed" do
    Given { give(builds_feed_plan).build(:walrus) {:plan} }
    When(:result) { subject.feed(:walrus, :cheese) }
    Then { verify(provides_meal).provide(:cheese, :plan) }
  end
end
Now I have two problems!

Build Feed Plan

Provide Meal
Now I have two problems! 💡

Build Feed Plan

Provide Meal

Now I have two problems! 💡

Build Feed Plan

Provide Meal

Now I have two problems! 💡

Build Feed Plan

Provide Meal

Now I have two problems! 💡

Build Feed Plan

Provide Meal

Now I have two problems! 💡

Build Feed Plan

Provide Meal
Correct behavior of logical leaf nodes
(a.k.a. pure, first-order functions)
Feed Zoo Animals

Build Feed Plan

Fetch Schedule

Plan Meal

Provide Meal
Feed Zoo Animals

Build Feed Plan
Fetch Schedule
Plan Meal
Compute Portions
Provide Meal
Release Food
Feed Zoo Animals

- Build Feed Plan
- Provide Meal
- Compute Portions
- Release Food

- Fetch Schedule
- Plan Meal

✔️ ✔️
Feed Zoo Animals

- Build Feed Plan
- Provide Meal

- Fetch Schedule
- Plan Meal
- Compute Portions
- Release Food

Discovery
Feed Zoo
Animals

✔️
Small things
Small things
(Free SRP!)
Separation of Roles
Feed Zoo Animals

Build Feed Plan
- Fetch Schedule
- Plan Meal

Provide Meal
- Compute Portions
- Release Food
Feed Zoo Animals

- Build Feed Plan
  - Fetch Schedule
    - Query
  - Plan Meal
- Compute Portions
- Provide Meal
  - Command
- Release Food

Discovery Feed Zoo Animals Plan Meal Command Query Build Feed Plan Fetch Schedule Plan Meal Compute Portions Provide Meal Release Food
Feed Zoo Animals

Build Feed Plan
  - Command
  - Query

Fetch Schedule
  - Query

Plan Meal
  - Logic

Provide Meal
  - Command

Compute Portions
  - Logic

Release Food
  - Command

Query Logic

Command
Discovery
Command & Query tests discover dependencies with test doubles
Logic tests discover implementation by usage
Logic tests discover implementation by usage

(i.e. no test doubles)
Commands and Queries contain very little logic
Pain is good!
Discovery tests yield small, disposable units
Discovery tests yield small, disposable units

...so don't be afraid to throw them away!
If requirements change, trash a minimal sub-tree of units and drive out a new solution
Re-use is overrated
Extract refactors are a smell
Frameworks will fight you!
Frameworks & TDD try to solve the same problems
Frameworks provide an orderly structure.
TDD is an approach for discovering an orderly structure
Discovery testing is at odds with most app frameworks
validate our code is useful to others
make public API easy-to-use
know that my code works

verify that production is working
grow a maintainable private API
ensure behavior of others' services

keep the product simple
narrowly specify our dependencies
safeguard our use of 3rd-party code
safely change & refactor

Discovery
validate our code is useful to others
make public API easy-to-use
know that my code works
grow a maintainable private API
ensure behavior of others' services
verify that production is working
narrowly specify our dependencies
safely change & refactor
keep the product simple
safeguard our use of 3rd-party code
Discovery
validate our code is useful to others
make public API easy-to-use
know that my code works
ensure behavior of others' services
safely change & refactor

verify that production is working
narrowly specify our dependencies

keep the product simple

grow a maintainable private API

safeguard our use of 3rd-party code
class ProvidesMeal
  def initialize...

  def provide(plan)
    @releases_food.release(
      @computes_portion.compute(plan)
    )
  end
end
web backend

Feed Zoo Animals

Build Feed Plan

Fetch Schedule

Plan Meal

Animal Adapter

Provide Meal

Compute Portions

Release Food

Cage Adapter

walrus data

panda data

cage control service
"Wow, that's a lot of mock objects!"
"Wow, that's a lot of fake stuff!"
"Wow, that's a lot of test doubles!"
"Aren't those many test doubles a pain?"
They can be!
Note: only use doubles when testing collaborators
Note: better test double libraries are better
Discovery tests try to improve dependencies' APIs
But we can't change 3rd party APIs
Mocking what you don’t own leads to useless pain.
Wrap 3rd party API calls in adapters that you do own
Wrap 3rd party API calls in adapters that you do own
(And mock those adapters instead)
Typically, adapters shouldn't need tests
Most tests exercise features that make use 3rd party APIs
It sometimes make sense to test code we don't own separately.
When you want to respond to 3rd party failures differently, write Adapter tests
Your application, pondering its relationship with a 3rd party API
Exercises your adapter API under realistic-enough circumstances
Tests of libraries warn of unsafe upgrades
Tests of network services warn of outages & breaking changes
Adapters (& tests) specify how you depend on a thing
Establishes boundaries, prevents 3rd party API references from seeping throughout your app.
Drastically reduces the cost of replacing dependencies later
Only test adapters with good cause
"Don't test the framework."
Adapter test suites of network code can be tricky to run in CI
It's easier to test 3rd party services from your SAFE suite.
All adapter tests are likely to be slow in ways outside your control
validate our code is useful to others
make public API easy-to-use
know that my code works

verify that production is working
grow a maintainable private API
ensure behavior of others' services

keep the product simple
narrowly specify our dependencies
safeguard our use of 3rd-party code
safely change & refactor
validate our code is useful to others
make public API easy-to-use
know that my code works
grow a maintainable private API
ensure behavior of others' services
verify that production is working
narrowly specify our dependencies
safely change & refactor
keep the product simple

safeguard our use of 3rd-party code
validate our code is useful to others

make public API easy-to-use

know that my code works

grow a maintainable private API

ensure behavior of others' services

verify that production is working

narrowly specify our dependencies

keep the product simple

safeguard our use of 3rd-party code

safely change & refactor
And that's just one way to break up your test suite
And that's just 1 way to break up your test suite
My name is Justin Searls
Please tweet me @searls &
Say hello@testdouble.com
Like everyone, we're hiring!

Just join@testdouble.com
Please say hello if your team could use our team's help 😊
Quotation Marks designed by Horacio Bella from the thenounproject.com
Happy designed by Michael Hourigan from the thenounproject.com
Tired designed by Aha-Soft from the thenounproject.com
Indifference designed by Paul F. from the thenounproject.com
Mad designed by Aha-Soft from the thenounproject.com
Sad designed by Simple Icons from the thenounproject.com
Marker designed by Jeff Seevers from the thenounproject.com
Database designed by Grant Fisher from the thenounproject.com
Cloud designed by Thomas Hirter from the thenounproject.com
Safety Net designed by irene hoffman from the thenounproject.com
Light Bulb designed by Olivier Guin from the thenounproject.com
Scale designed by Stephanie Wauters from the thenounproject.com
Warning designed by Lorena Salagre from the thenounproject.com
Reload designed by Mateo Zlatar from the thenounproject.com