EFFECTIVE STARTUP ENGINEERING

Ian Wong • Square Inc.
coursera.org/course/startup

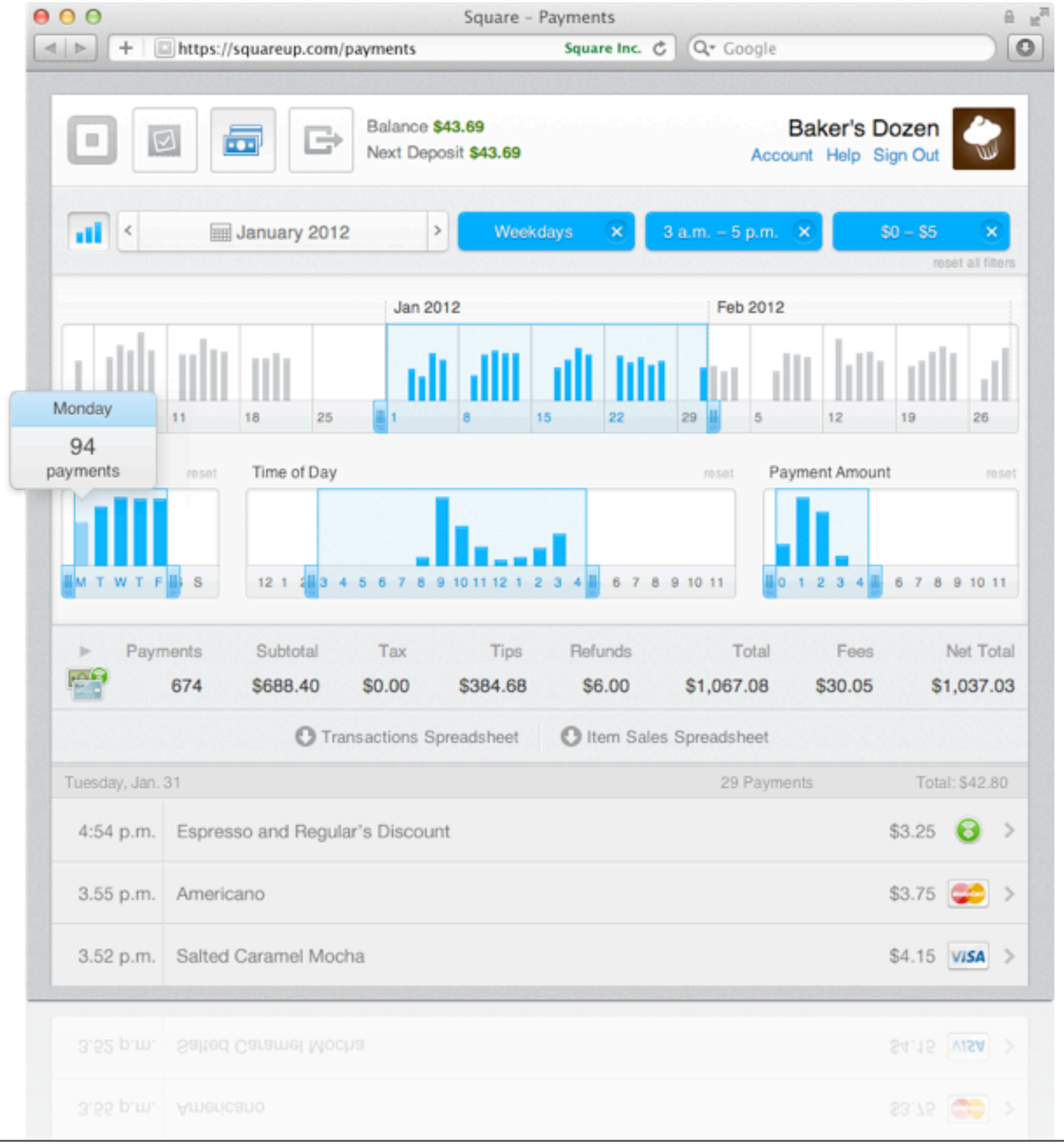
ef·fec·tive
/i'fektiv/

An effective startup engineering team ships useful and delightful products to the customers at a fast pace.

Square makes commerce easy
for **everyone**.

Enable anyone to accept credit card payments **anytime, anywhere**.

Square register



Analytics that enable small businesses to compete (and win).

Square wallet

Momentum

- Square is currently processing more than **$10 billion in payments** annually.
- In 2012, we grew our customer base from 1 million to **3 million**.
- There are more than **250,000 businesses** listed in the Square directory.
- Square readers are now available in nearly **40,000 retail locations** nationwide.
- Square's transaction volume makes it the equivalent of the **20th largest retailer** in the United States.

Who's this dude?

Ian Wong

Inference Scientist

- Stanford BS EE 08, MS EE 09, MS Stats 10, PhD Dropout 10
- Square Risk Team
- Machine learning system to assess and mitigate risk of entities and events in our network
- Visualization and workflow tools for operational specialists to review customer accounts

Design for modularity

Learn to test

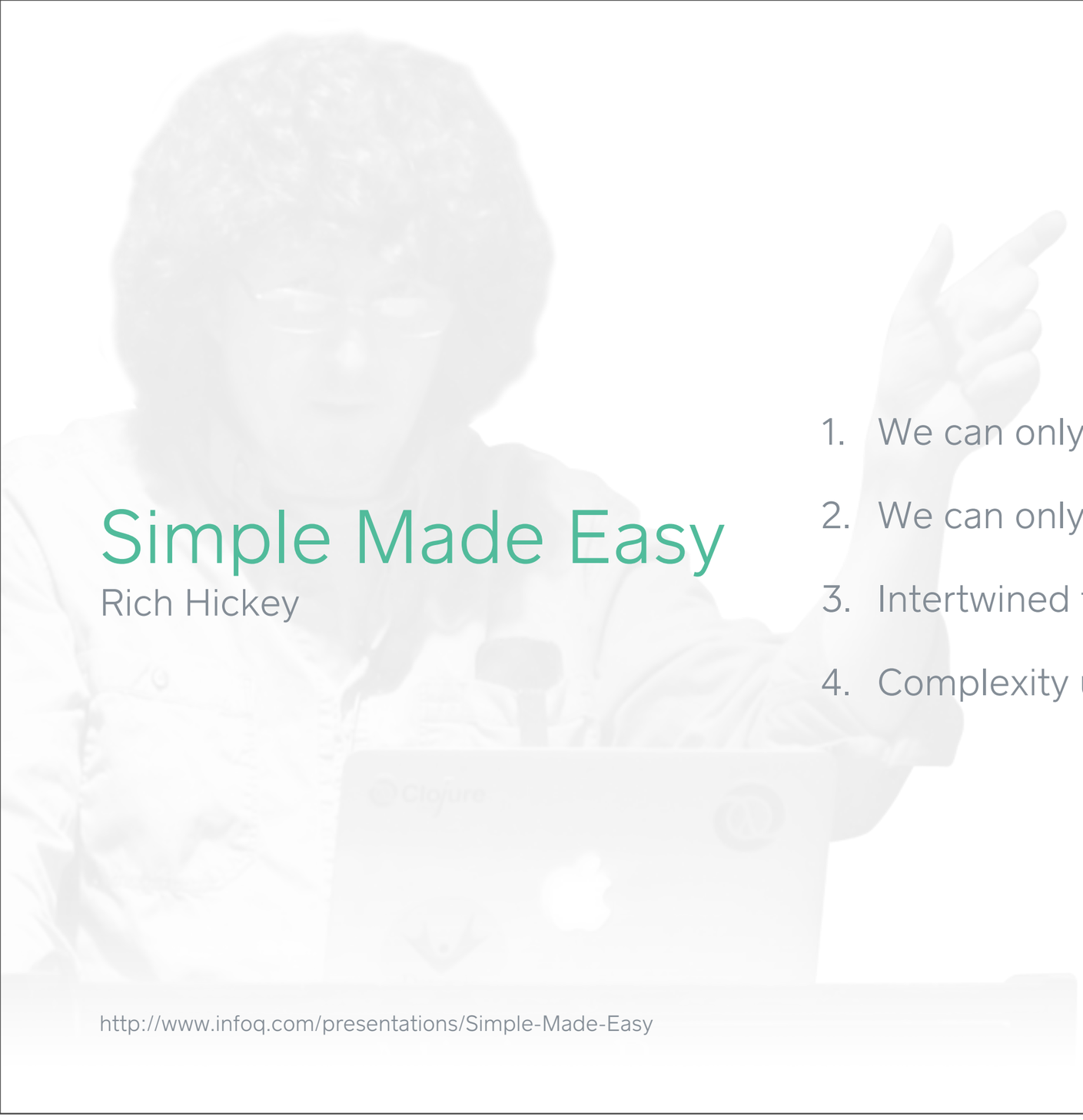
Use and contribute to open source

Foster an open, collaborative and responsible culture

Design for modularity

Learn to test

Use and contribute to open source



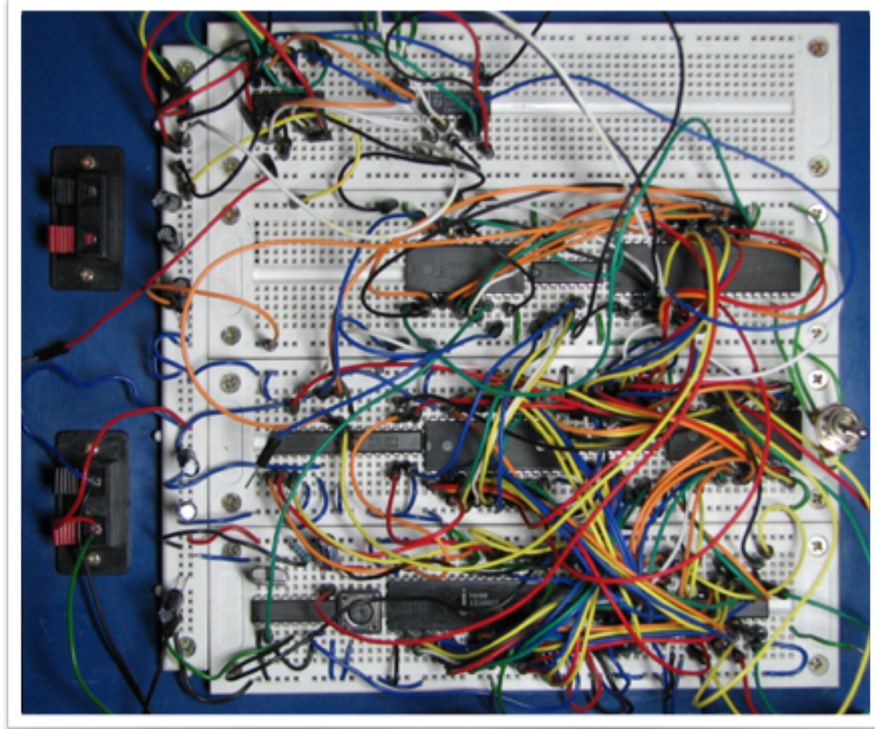
Simple Made Easy

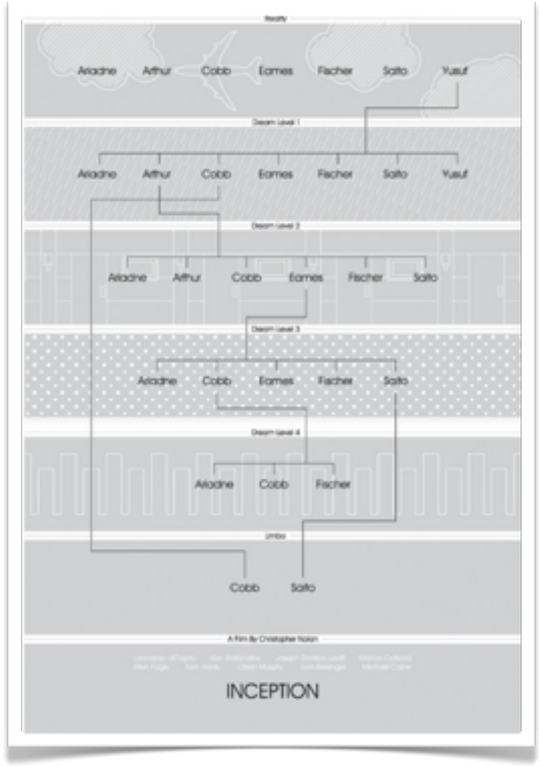
Rich Hickey

1. We can only hope to make reliable those things we understand.
2. We can only consider a few things at a time.
3. Intertwined things must be considered together.
4. Complexity undermines understanding.

http://www.infoq.com/presentations/Simple-Made-Easy

You want to avoid this





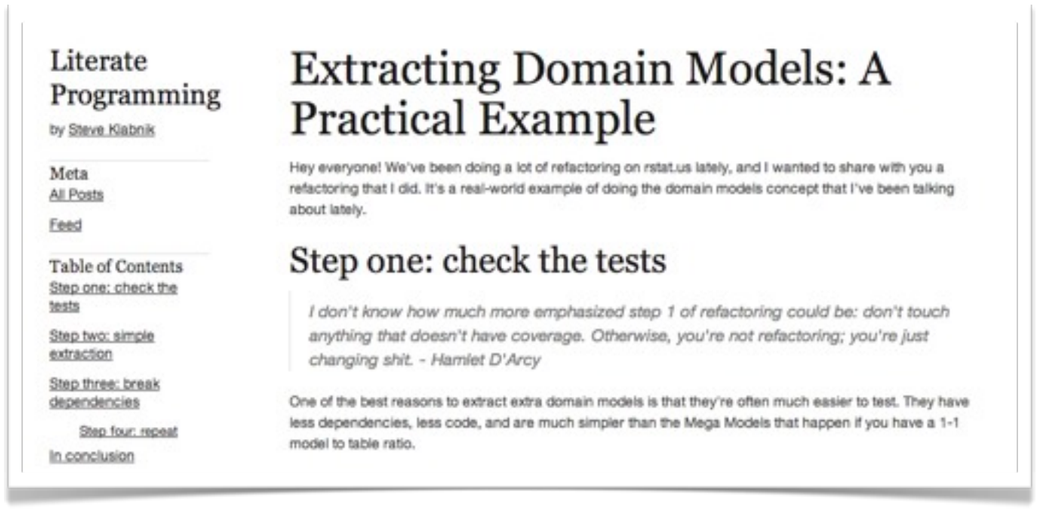
Let’s talk about your god class

Let's talk about your god class

- Does it do everything?
- Harder and harder to add functionality?
- Things feel tangled? Lots of states and side effects?

Extract a Service Class

Tip: Stop adding functionality to your model or controller. Create a service class.



Literate Programming

by Steve Klabnik

Meta
All Posts
Feed

Table of Contents
Step one: check the tests
Step two: simple extraction
Step three: break dependencies
Step four: repeat
In conclusion

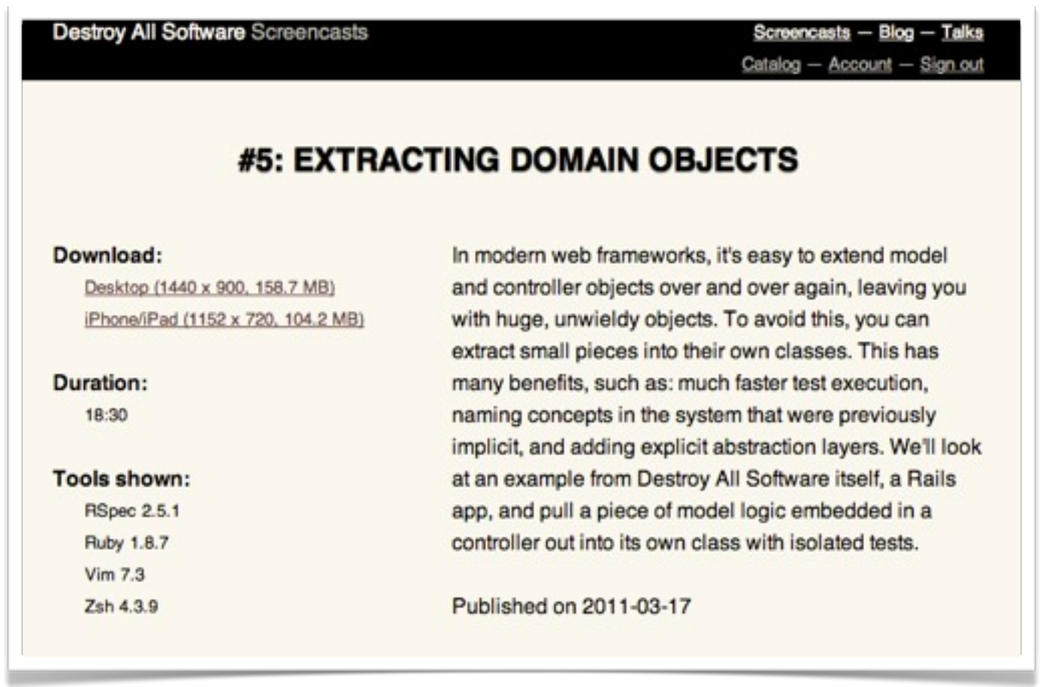
Extracting Domain Models: A Practical Example

Hey everyone! We've been doing a lot of refactoring on rstat.us lately, and I wanted to share with you a refactoring that I did. It's a real-world example of doing the domain models concept that I've been talking about lately.

Step one: check the tests

> *I don't know how much more emphasized step 1 of refactoring could be: don't touch anything that doesn't have coverage. Otherwise, you're not refactoring; you're just changing shit. - Hamlet D'Arcy*

One of the best reasons to extract extra domain models is that they're often much easier to test. They have less dependencies, less code, and are much simpler than the Mega Models that happen if you have a 1-1 model to table ratio.



Destroy All Software Screencasts

Screencasts — Blog — Talks
Catalog — Account — Sign out

#5: EXTRACTING DOMAIN OBJECTS

Download:
Desktop (1440 x 900, 158.7 MB)
iPhone/iPad (1152 x 720, 104.2 MB)

Duration:
18:30

Tools shown:
RSpec 2.5.1
Ruby 1.8.7
Vim 7.3
Zsh 4.3.9

In modern web frameworks, it's easy to extend model and controller objects over and over again, leaving you with huge, unwieldy objects. To avoid this, you can extract small pieces into their own classes. This has many benefits, such as: much faster test execution, naming concepts in the system that were previously implicit, and adding explicit abstraction layers. We'll look at an example from Destroy All Software itself, a Rails app, and pull a piece of model logic embedded in a controller out into its own class with isolated tests.

Published on 2011-03-17

Extract a Service Class

```
@@ -6,16 +6,9 @@ class Admin::RefundsController < AdminController
   end

   def approve
-    refund = Refund.find(params[:id])
     raise UnauthorizedError unless current_user.may_approve_refund?
-    raise UnprocessableEntityError unless refund.requested? || refund.completed? || refund.failed?

-    if refund.requested?
-      refund.update_attributes!(:status => :approved, :review_comment => params[:review_comment])
-      refund.process!
-    end
-
-    if refund.completed?
+    if Risk::RefundProcessor.approve(refund, comment)
       flash[:message] = "Refund completed."
     else
       flash[:error] = "Refund failed."
```

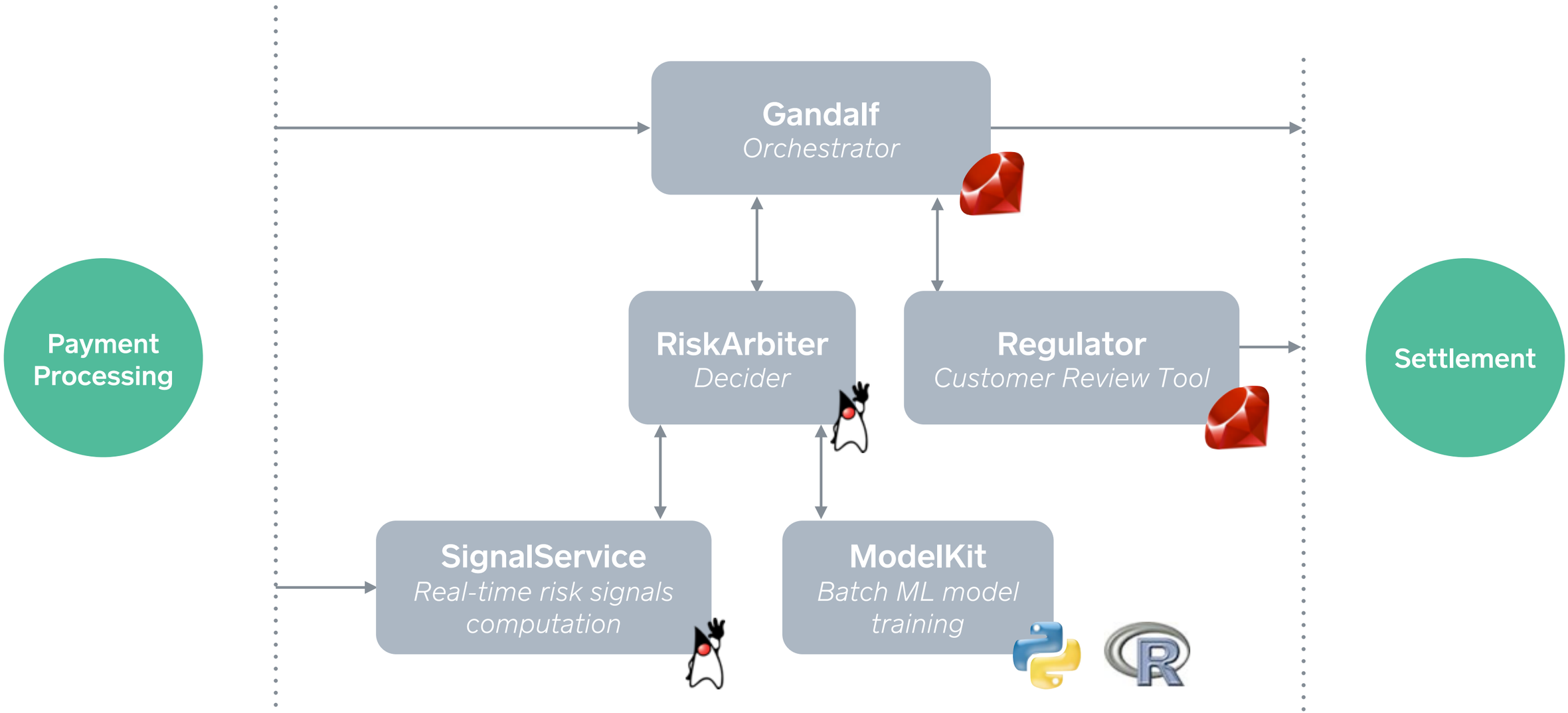
Micro to Macro

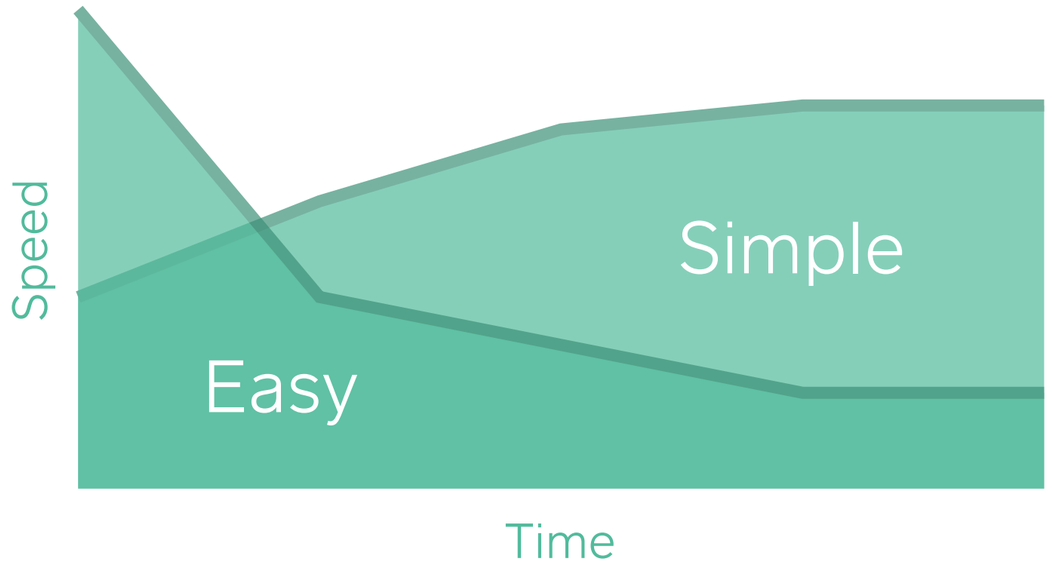
- Loosely coupled, well-encapsulated service classes **within a code base** interact with each other via simple interfaces.
- Loosely coupled, well-encapsulated services **within a service oriented architecture** interact with each other via simple interfaces.

Life of a Payment

An individual service can grow in complexity without being concerned with other services.

Separation of Concerns





Source: *Simple Made Easy*, Rich Hickey.

Design for modularity

Learn to test

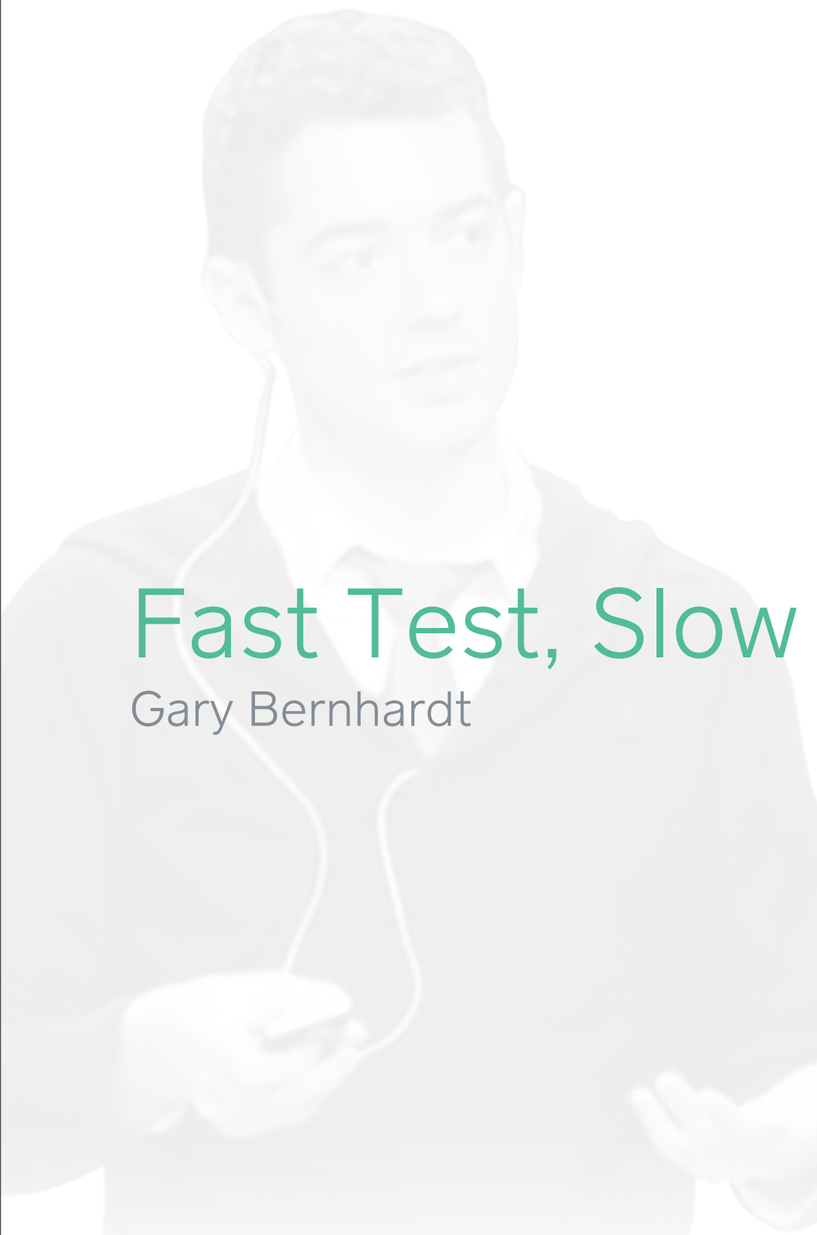
Use and contribute to open source

Design for modularity

Learn to test

Use and contribute to open source

Foster an open, collaborative and responsible culture



Fast Test, Slow Test

Gary Bernhardt

http://www.youtube.com/watch?v=RAxiiRPHS9k

The three goals of testing:

1. Prevent regression
2. Prevent fear
3. Prevent bad design

Prevents Regression

```
.................................

Finished in 0.10664 seconds
66 examples, 0 failures
```

- Introduce change that will not unexpectedly break what you have.
- “If you don’t test your code, your customers will” (Pragmatic Programmer).

Prevents Fear

```
describe Admin::RefundsController do
  describe "#approve" do
    let(:refund) { Factory(:card_refund,
                   :status => refund_status,
                   :review_comment => review_comment) }
    let(:review_comment)  { nil }
    let(:refund_status)   { :requested }
    let(:call_controller) { put :approve,
                            :id => refund.to_param,
                            :review_comment => "cuz!" }
    let(:user) { Factory(:user, :role_name => role_name) }
    let(:role_name) { risk }

    before { user_login user }
    it "approves and completes the refund request" do
      call_controller
      response.should redirect_to(admin_refund_request_url)
      flash[:message].should == "Refund completed."
      refund.reload.should be_completed
      refund.review_comment.should == "cuz!"
    end
```

- Refactor with confidence.
- Tests as documentation.
- Tests as contracts.

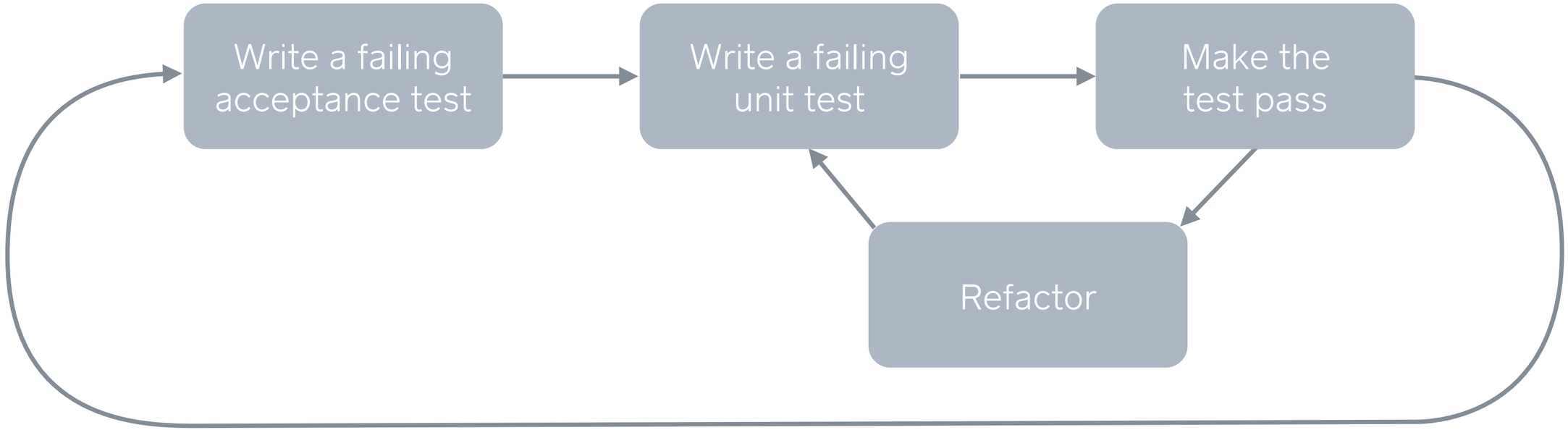
Prevents Fear

```
describe Risk::RefundProcessor do
  let(:admin_user) { Factory(:admin_user) }
  let(:refund) { Factory(:card_refund,
                         :status => refund_status,
                         :review_comment => review_comment) }
  let(:review_comment) { nil }
  let(:refund_status)  { :requested }

  describe ".approve" do
    it "should return true and leave the refund's status if it
is completed" do
      refund.status = :completed
      refund.should_not_receive(:process!)
      Risk::RefundProcessor.approve_and_process(admin_user,
refund, nil).should be_true
    end
  end
end
```

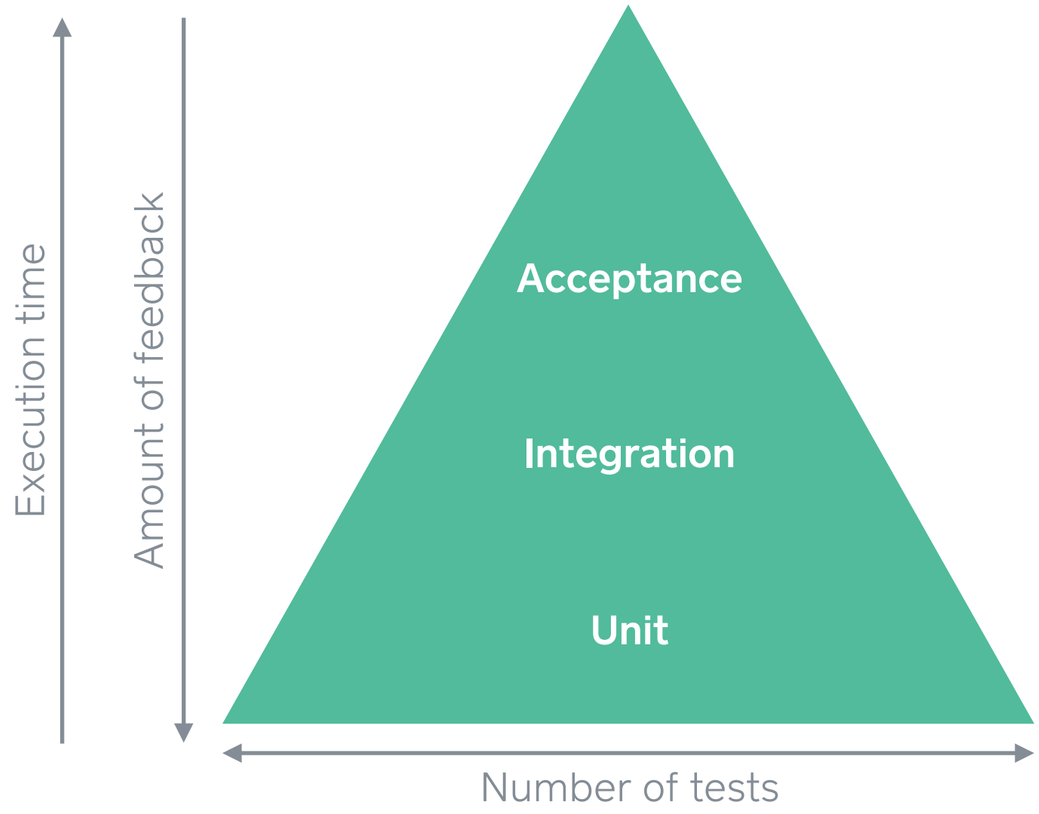
- Refactor with confidence
- Tests as documentation
- Tests as contracts

Prevents Bad Design



Source: Growing Object-Oriented Software, Guided by Tests

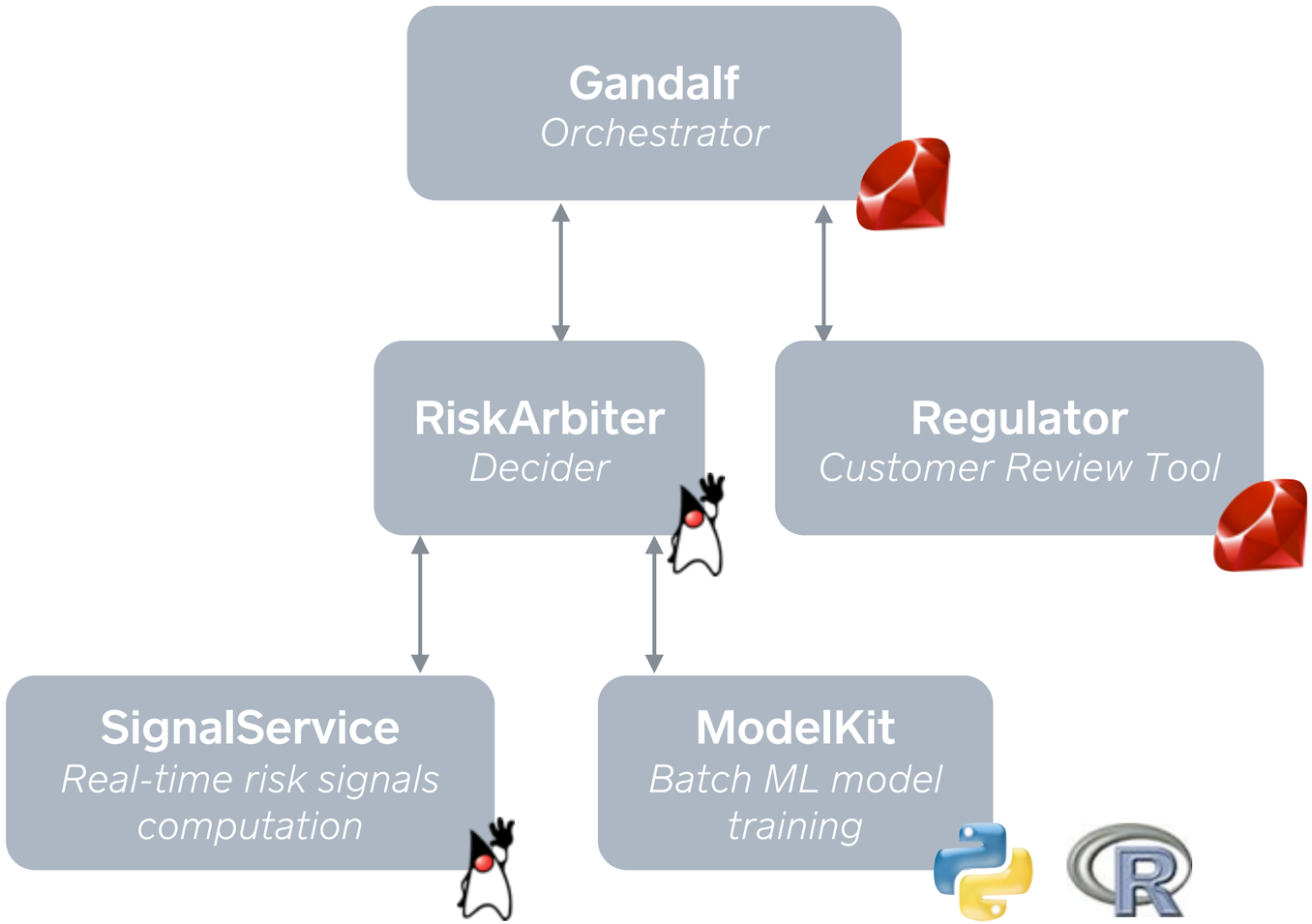
Types of tests



- **Acceptance:** Tests whole application. Mostly happy paths.
- **Integration:** External dependencies simulated. Tests class interaction.
- **Unit:** Focus on application logic. Fast! Quick feedback!

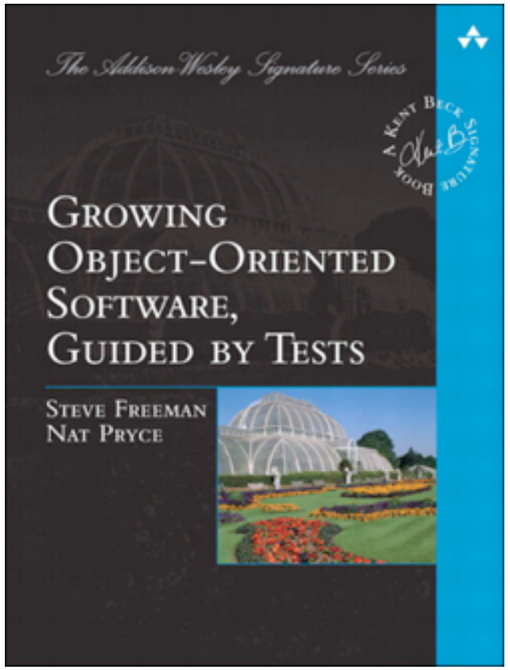
Source: Misko Hevery, Clean Code Talks.

Drives Design: Demo

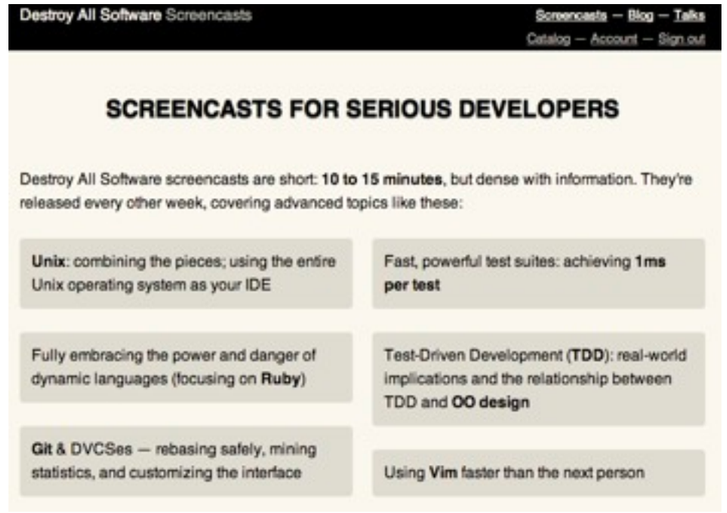


Let's build Gandalf.

Resources


The Addison Wesley Signature Series
A Kent Beck Signature Book
Growing Object-Oriented Software, Guided by Tests
Steve Freeman
Nat Pryce


The Clean Code Talks
Unit Testing
Miško Hevery
October 30, 2008
Google


Destroy All Software Screencasts
Screencasts — Blog — Talks
Catalog — Account — Sign out
SCREENCASTS FOR SERIOUS DEVELOPERS
Destroy All Software screencasts are short: 10 to 15 minutes, but dense with information. They're released every other week, covering advanced topics like these:
Unix: combining the pieces; using the entire Unix operating system as your IDE
Fast, powerful test suites: achieving 1ms per test
Fully embracing the power and danger of dynamic languages (focusing on Ruby)
Test-Driven Development (TDD): real-world implications and the relationship between TDD and OO design
Git & DVCSes — rebasing safely, mining statistics, and customizing the interface
Using Vim faster than the next person

Design for modularity

Learn to test

Use and contribute to open source

Foster an open, collaborative and responsible culture

Design for modularity

Learn to test

Use and contribute to open source

Foster an open, collaborative and responsible culture

Develop an Open Source Philosophy

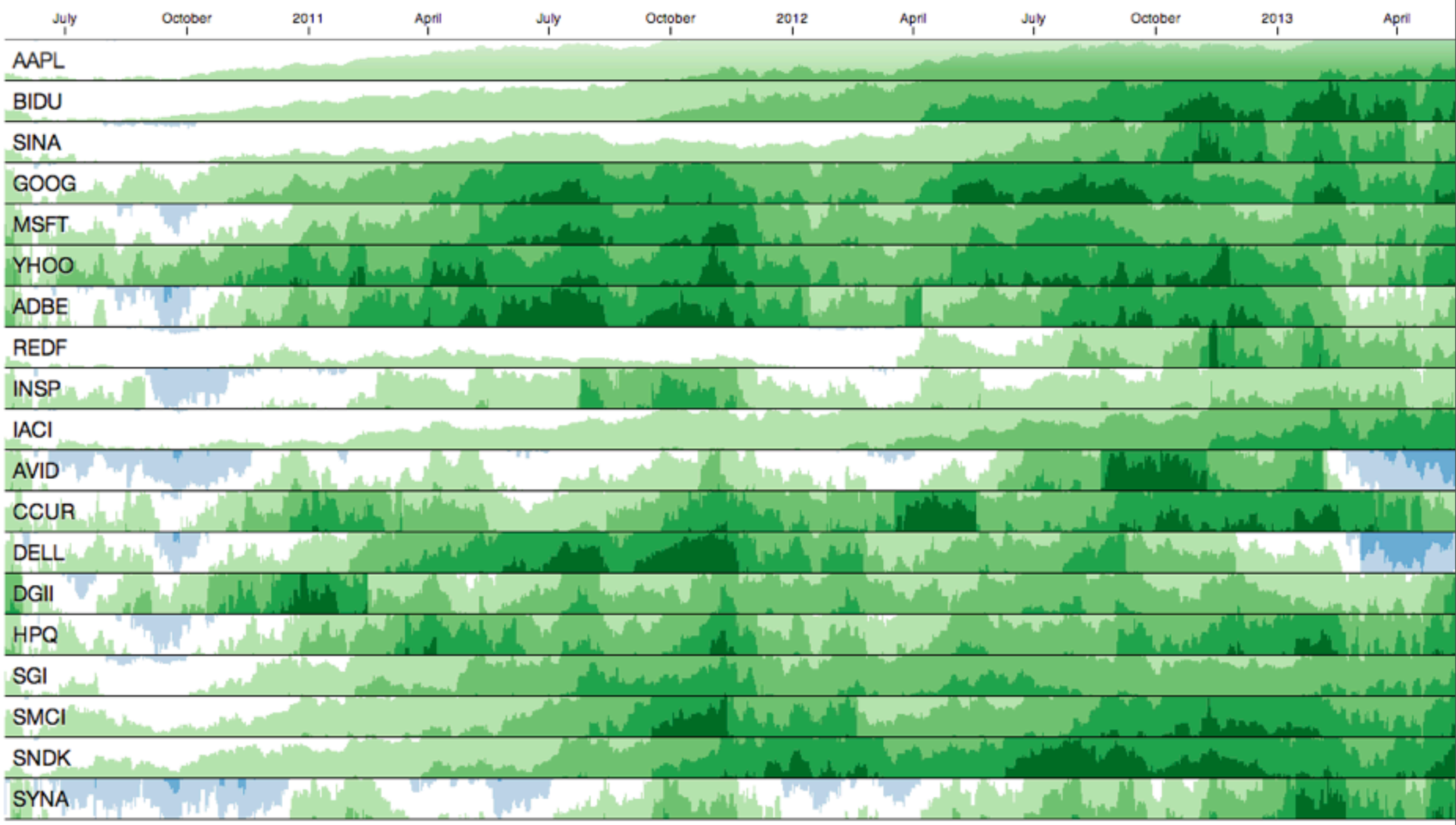
- Square is built upon open source.
- Many developers are active contributors to open source software.
- Build what you need to deliver value.
- When you must, invent and open source.
- Build your brand.

Square's Open Source Projects

Sampling from github.com/square

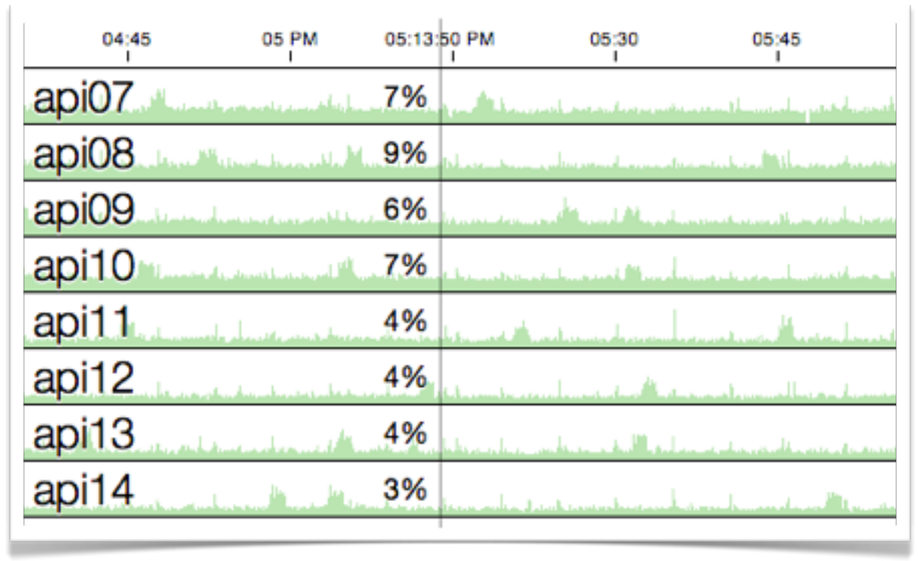
- **KIF (Objective-C):** Keep It Functional - An iOS functional testing framework.
- **Cane (Ruby):** Code quality threshold checking as part of your build.
- **Dagger (Java):** A fast dependency injection framework for Android and Java.
- **Cubism (JS):** A JavaScript library for time series visualization.
- **Squash (x-platform):** A library for exception tracking and bug reporting.

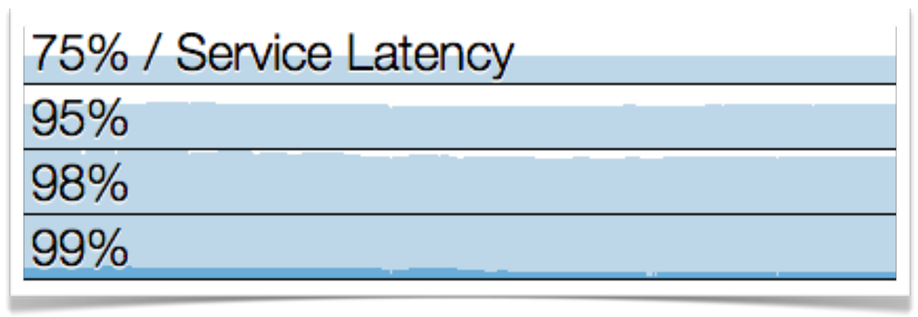
Cubism



Source:
http://square.github.com/cubism/
http://bost.ocks.org/mike/cubism/intro/demo-stocks.html

Cubism


04:45
05 PM
05:13:50 PM
05:30
05:45
api07
7%
api08
9%
api09
6%
api10
7%
api11
4%
api12
4%
api13
4%
api14
3%


75% / Service Latency
95%
98%
99%

Design for modularity

Learn to test

Use and contribute to open source

Foster an open, collaborative and responsible culture

Learn to test

Use and contribute to open source

Foster an open, collaborative and responsible culture

Transparency

Ingredients to an Effective Startup

Engagement

Ingredients to an Effective Startup

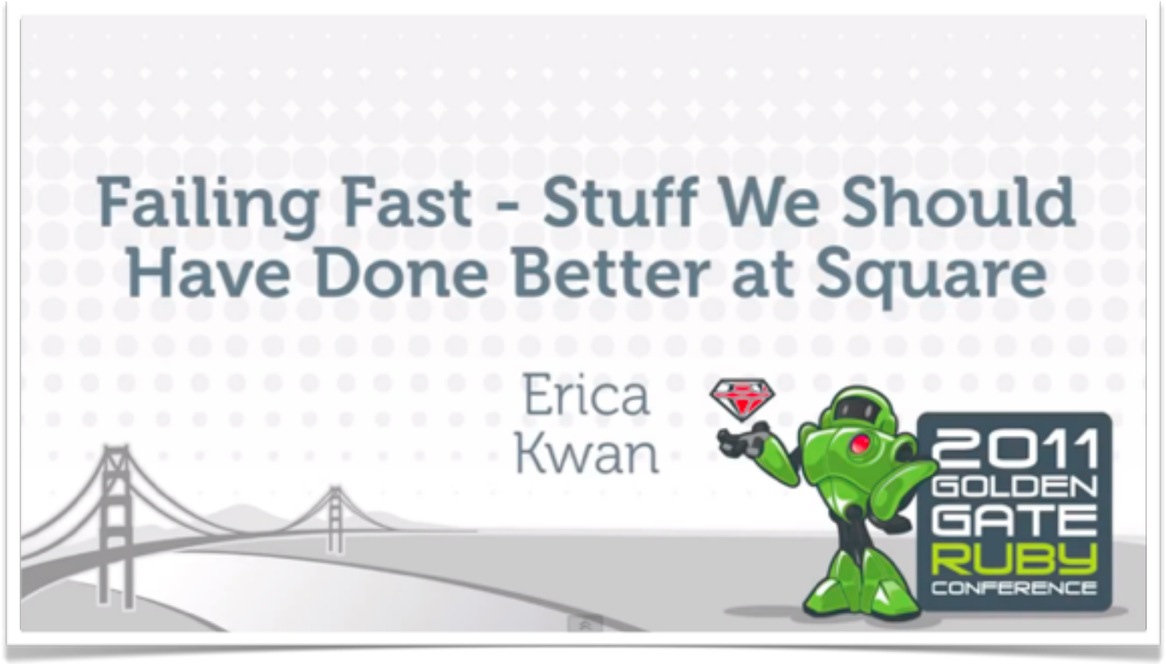
Responsibility

Ingredients to an Effective Startup

Pair Programming

- Instant code and design feedback.
- Learn from one another.
- Fastest way to onboard new hires.
- No excuse to check Twitter.
- **“Pairing on a problem” is a common parlance.**

Engineering processes



- Code reviews
- On-call rotation
- Postmortem
- Retrospectives

Source:
http://www.youtube.com/watch?v=AajAVNr73tg

Design for modularity

Learn to test

Use and contribute to open source

Foster an open, collaborative and responsible culture

What You Should Do Later Today

- Extract a service class
- Write a test before you implement
- Pair program with your teammate

It’s okay if you didn’t catch everything. **We’re hiring.**

jobs.squareup.com

ian@squareup.com

@ihat