REAL MEN DON'T USE MENUS. I WANT TO KNOW HOW TO USE POWER COMMANDS.

Greenfield

New Service

Wednesday, 22 February 12

Wednesday, 22 February 12

decoupling from Rails for fun and/or profit

)

basic correctness

this then that

example pontification recommendations

an example

“when new users sign up, send them a welcome email”

where does this logic belong?

in the controller?

```
class UsersController <
ApplicationController
  def update
    u = User.create!(params)
    UserMailer.welcome_message(u).deliver
  end
end
```

Skinny Controller, Fat Model

Posted by Jamis on October 18, 2006 @ 07:50 AM

When first getting started with Rails, it is tempting to shove lots of logic in the view. I'll admit that I was guilty of writing more than one template like the following during my Rails novitiate:

```
<!-- app/views/people/index.rhtml -->
<% people = Person.find(
      :conditions => ["added_at > ? and deleted = ?", Time.n
      :order => "last_name, first_name") %>
<% people.reject { |p| p.address.nil? }.each do |person| %>
  <div id="person-<%= person.new_record? ? "new" : person.id
    <span class="name">
      <%= person.last_name %>, <%= person.first_name %>
    </span>
    <span class="age">
      <%= (Date.today - person.birthdate) / 365 %>
    </span>
  </div>
<% end %>
```

Not only is the above difficult to read (just you *try* and find the HTML elements in it), it also completely bypasses the "C" in "MVC". Consider the controller and model implementations that support that view:

```
# app/controllers/people_controller.rb
class PeopleController < ActionController::Base
end

# app/models/person.rb
class Person < ActiveRecord::Base
  has_one :address
end
```

Just look at that! Is it really any wonder that it is so tempting for

in the model?

```
class User < ActiveRecord::Base

  def self.create_account!(params)
    u = create!(params)
    UserMailer.welcome_message(u).deliver
  end
end
```

where do we test this logic?

```
describe UsersController do
  it “emails new users upon creation”
    post :create, user_params
    ActionMailer.base.deliveries.should
                                have(1).email
  end
end
```

```
describe User do
  it “emails new users upon creation”
    User.create_account!(user_params)

    ActionMailer.base.deliveries.
                should have(1).email
  end
end
```

how do we name that method?

want to use create!, but hate overriding

this inherently leads to fat models

how do we test unrelated logic on the user class?

do we
call
`.create_account!`
in each spec?

slow

or do we just call

`User.create!`

directly?

SRP

you be breakin it

SRP

a class should only have one reason to change

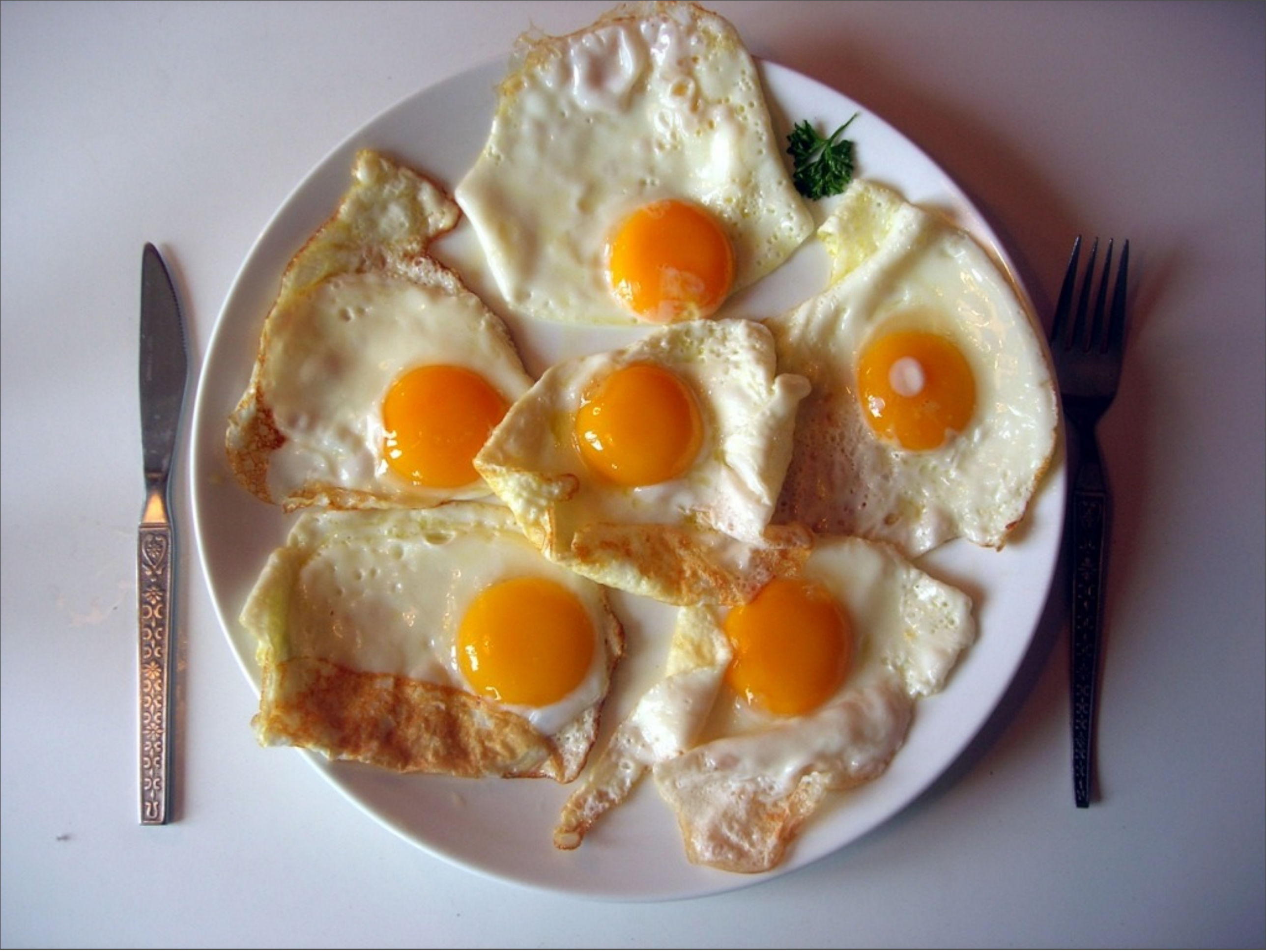

OBJECT MENTOR

Robert C. Martin Series

Clean Code

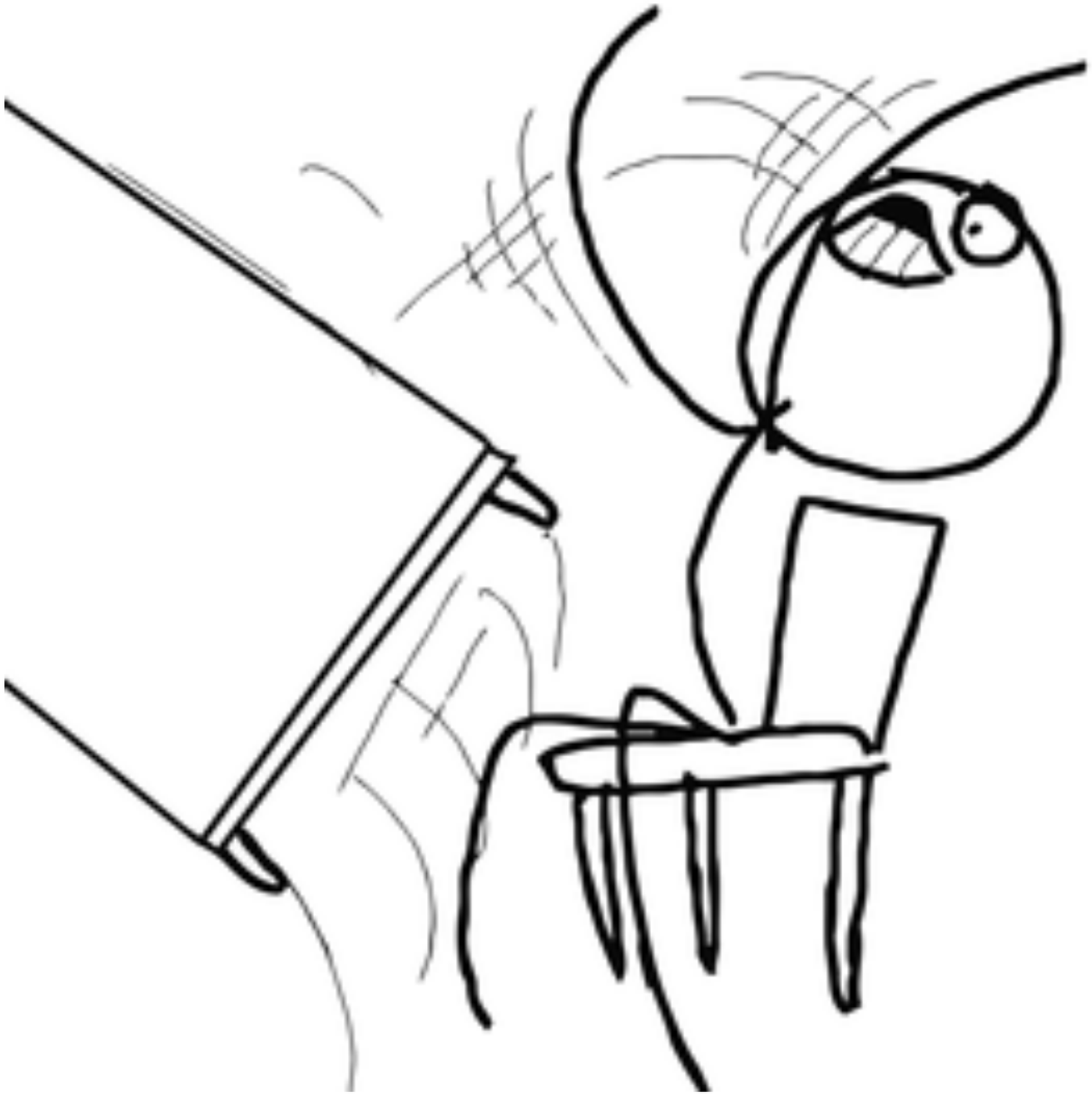
A Handbook of Agile Software Craftsmanship

```
class User < ActiveRecord::Base
  after_create :send_welcome_email
end
```



Wednesday, 22 February 12

```
class WelcomeEmailObserver <
ActiveRecord::Observer
  def after_create(user)
    UserMailer.welcome_email(user).
                              deliver
  end
end
```



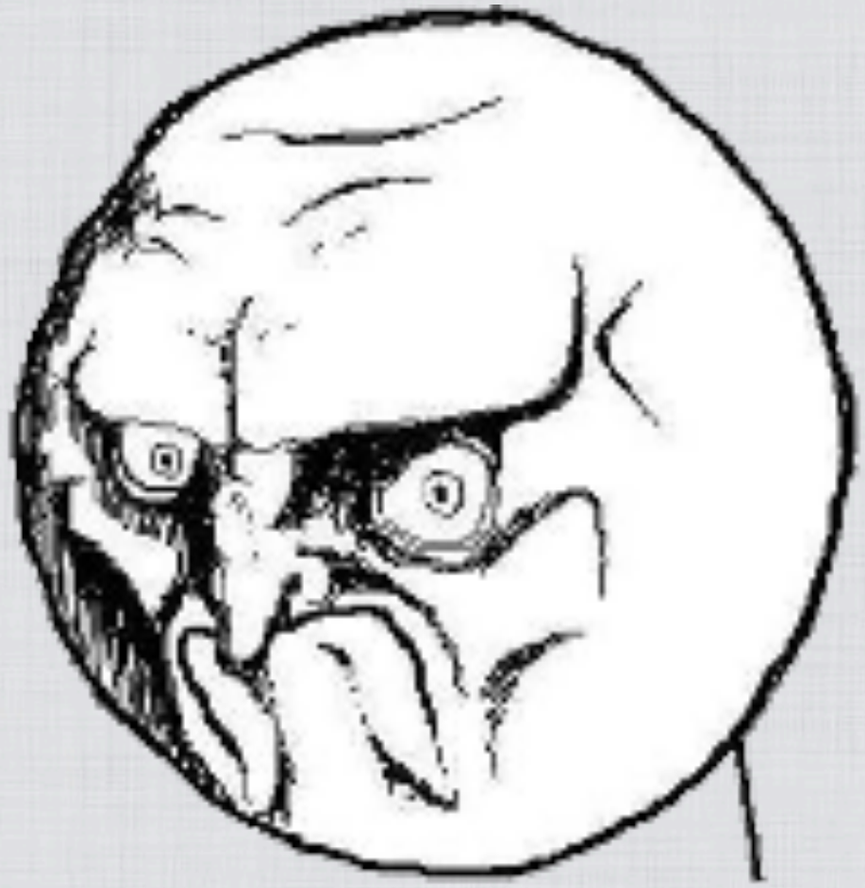
now all your specs that create `User` are

slow!

unwanted temporal coupling

```
class UsersController <
ApplicationController
  after_filter :send_welcome_email,
               :only => :create

end
```



NO.

Wednesday, 22 February 12

a separate class

CreatesUsers

UserCreator

```
class CreatesUsers
  def self.create!(params)
    u = User.create!(params)
    UserMailer.send_welcome_email
                              (user).deliver
  end
end
```

```
describe CreatesUsers do
  let(:user) { stub }
  let(:email) { stub }
  let(:user_params) { stub }
  it “emails new users after creation” do
    User.stub(:create!).
                  and_return(user)

UserMailer.stub(:welcome_email).with(user).
                  and_return(email)
    email.should_receive(:deliver)
    CreatesUsers.create!(user_params)
  end
end
```

this spec is more complex in rails 3

in rails 2

```
require ‘app/services/creates_users’
describe CreatesUsers do
  let(:user) { stub }
  let(:user_params) { stub }
  it “emails new users after creation” do
    User.stub(:create!).
                  and_return(user)

UserMailer.should_receive(:deliver_welcome_e
mail).with(user)
    CreatesUsers.create!(user_params)
  end
end
```

don't need to check that the record is actually created

don't check if we actually deliver email

Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

KEN THOMPSON

INTRODUCTION

I thank the ACM for this award. I can't help but feel that I am receiving this honor for timing and serendipity as much as technical merit. UNIX[1] swept into popularity with an industry-wide change from central mainframes to autonomous minis. I suspect that Daniel Bobrow [1] would be here instead of me if he could not afford a PDP-10 and had had to "settle" for a PDP-11. Moreover, the current state of UNIX is the result of the labors of a large number of people.

There is an old adage, "Dance with the one that brought you," which means that I should talk about

programs. I would like to present to you the cutest program I ever wrote. I will do this in three stages and try to bring it together at the end.

STAGE I

In college, before video games, we would amuse ourselves by posing programming exercises. One of the favorites was to write the shortest self-reproducing program. Since this is an exercise divorced from reality, the usual vehicle was FORTRAN. Actually, FORTRAN was the language of choice for the same reason that

speed

demo

this spec runs in less than 1ms

~0.00069s on my machine

> 1000 specs a second

~1449 specs/s on my machine

what if I could find out if the entire system worked in less than 1 second?

```
describe UsersController do
  it “creates a new user” do
    CreatesUsers.should_receive(:create!)
    post :create, user_params
  end
end
```

one clear place to test your logic

little temptation to test the same thing in multiple places

```
describe UsersController do
  it “emails new users upon creation”
    post :create, user_params
    ActionMailer.base.deliveries.should
                                have(1).email
  end
end
```

Principled Isolated Tests

no “spec_helper”

spec pass/failure depends on ONE CLASS

ergo, that class cannot inherit from anything

A unit is a class under 100 lines

for me, often under 30-40

failure localisation

```
FFFFF.......FFFFFFFFFF.........FFFFFFF.....F
FFFFF.....FFFFFFFF.....FFFF....FFF
```

FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

FFFFF.....FF | FFFFF.....FF
FFFFFFFF..... | FFFFFFFF.....
.FFFFFFF.....FF | .FFFFFFF.....FF
FFFF.....FFFFFF | FFFF.....FFFFFF
FF.....FFFF.... | FF.....FFFF....
FFF | FFF

FFFFFFFFFFFFFFFFF

FFFFFFFFFFFFFFFFF

FFFFFFFFFFFFFFFFF

FFFFFFFFFFFFFFFFF

FFFFFFFFFFFFFFFF

all your specs depend on what you just changed

..............FF..........................

.....

Abstraction

my big complaint

how your specs can help you with abstraction

thanks

Mocks, the removal of test detail, and dynamically-typed languages

Posted in TDD, programming | Wednesday, January 13th, 2010 | Trackback

Simplify~~, simplify, simplify!~~
– ~~Henry David~~ Thoreau

(A billboard I saw once.)

Part 1: Mocking as a way of removing words

One of the benefits of mocks is that tests don't have to build up complicated object structures that have nothing essential to do with the purpose of a test. For example, I have an entry point to a webapp that looks like this:

```
get '/json/animals_that_can_be_taken_out_of_service', :date => '2009-01-01'
```

It is to return a JSON version of something like this:

```
{ 'unused animals' => ['jake'] }
```

Jake can be taken out of service on Jan 1, 2009 because he is not reserved for that day or any following day.

In typical object-oriented fashion, the controller doesn't do much except ask something else to do something. The code will look something like this:

What Your Tests Don't Need to Know Will Hurt You

I just finished reading Brian Marick's article, "Mocks, the removal of test detail, and dynamically-typed languages", which focused me on a design technique I use heavily: awareness of irrelevant details in tests. Referring back to the four elements of simple design, I focus on irrelevant details in tests in order to help me maximize clarity in production code. Please allow me to sketch the ideas here.

I use the term *irrelevant detail* to refer to any detail that does not contribute directly to the correctness of the behavior I've chosen to check. I know when I've bumped into an irrelevant detail: while writing the code that allows me to check something, I start typing something, then my shoulders slump and I exhale with annoyance. I think, *I shouldn't have to write this, because it has nothing to do with what I want to check*. Brian's example illustrates this perfectly:

> *The random methods save a good deal of setup by defaulting unmentioned parameters and by hiding the fact that Reservations have_many Groups, Groups have_many Uses, and each Use has an Animal and a Procedure. But they still distract the eye with irrelevant information. For example, the controller method we're writing really cares nothing for the existence of Reservations or Procedures–*
> *but the test has to mention them*

clearly mark irrelevant detail in your specs

```
let(:irrelevant_user) { stub }
```

then see how much of it you can remove

dependency injection + test doubles

an example
would be handy
right about now

"if bidding on the auction times out, email the user"

```
class BidsOnAuctions
  def bid(auction, price)
    begin
      @auction_house.bid_on(auction, price)
    rescue AuctionTimedOut => e
      AuctionMailer.timed_out_auction
                              (auction,
                             @user)
  end
end
```

```
describe BidsOnAuctions do
  let(:user) { double }
  let(:auction_house) { double }
  subject do
    BidsOnAuctions.new(auction_house, user)
  end
  it “notifies users when auctions time out” do
    auction_house.stub(:bid_on).raises(AuctionTimedOut)

    AuctionMailer.should_receive(:timed_out_auction).with(auction,
user)
    subject.bid_on(auction, price)
  end
end
```

"notify the user by sms if he has given us his mobile number"

```
describe BidsOnAuctions do
  subject do
    BidsOnAuctions.new(auction_house, user_notifier)
  end
  it “notifies users when auctions time out” do
    auction_house.stubs(:bid_on).raises(AuctionTimedOut)

    user_notifier.should_receive(:auction_timed_out).with(auction)
    subject.bid_on(auction, price)
  end
end
```

```
class BidsOnAuctions
  def bid(auction, price)
    begin
      @auction_house.bid_on(auction, price)
    rescue AuctionTimedOut => e
      @auction_notifier.timed_out!(auction)
  end
end
```

isolation forces you to list dependencies

tells you when they are too big

an aside: why I dislike inheritance

coupling

ease of testing

in isolation

helps you notice new abstractions

if you pay attention

recommended reading/viewing

Gary Bernhardt
@garybernhardt

Following

Can't wait for @coreyhaines' fast Rails tests talk to convert everyone so I can complain about how I did it before it was cool.

2 RETWEETS | **1** FAVORITE

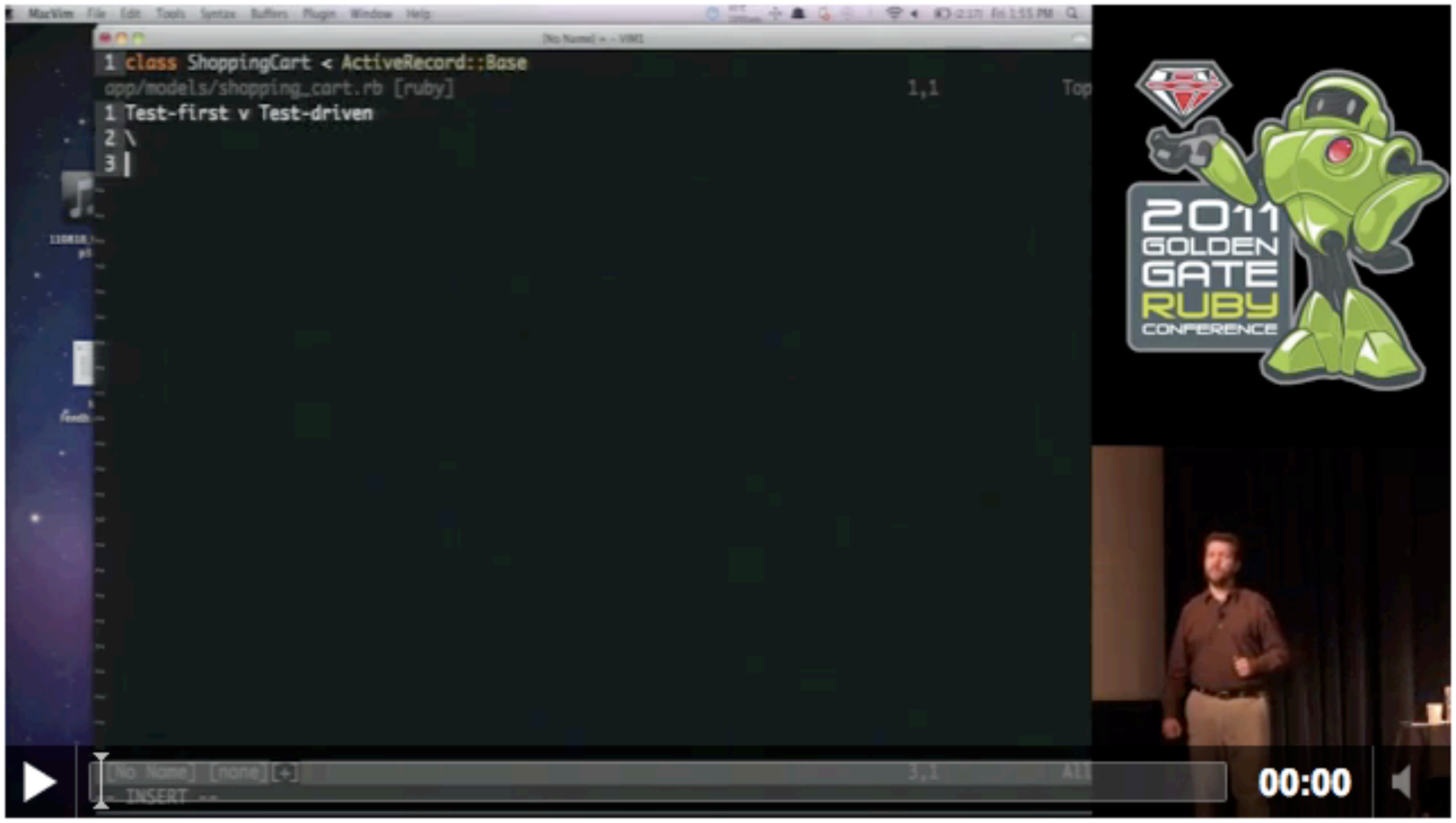
11:32 PM - 13 Sep 11 via Tweetbot for iPhone · Embed this Tweet

Reply Retweeted Favorited

Confreaks

Home | Events | Presenters | Blog

Golden Gate Ruby Conference 2011



Fast Rails Tests

Corey Haines

0

Tweet

Look at your Rails unit test suite. Now look at mine. Now look at yours again. Mine are sub-second. Yours aren't. Having a slow unit test suite can hinder an effective test-first or test-driven approach to development. As you add tests, the suite starts to slow down to the point where you stop running them after each change. Some people even talk about multi-minute unit tests suites! Band-aids like spork are just covering up the problem. Test-driven development is a major player in keeping your design malleable and accepting of new features, but when you stop paying attention to the messages your tests are sending you, you lose this benefit.

InfoQ

En | 中文 | 日本語 | Br

562,907 Jan unique visitors

Development Architect & Design

Mobile HTML5 JavaScript Cloud Agile SOA Agile Techniques NoSQL Cloud Security

Presentation

Integration Tests Are a Scam

Presented by J.B. Rainsberger on Sep 10, 2009 Length 01:32:56 Download: MP3

Sections Process & Practices, Architecture & Design, Development Topics FEATURED Agile Techniques , Software Testing , FEATURED Agile Tags Integration Test , Agile2009

Share | dz

How would you like to view the presentation? horizontal vert

Summary

Integration tests are a scam. You're probably writing 2-5% tests you need to test thoroughly. You're probably duplicat over the place. Your integration tests probably duplicate ea the place. When an integration test fails, who knows what's two-pronged attack that solves the problem: collaboration tests.

Bio

J. B. (Joe) Rainsberger helps software organizations better s customers and the businesses they support. Expert at deliv software, he writes, teaches and speaks about why deliveri is important, but not enough. He helps clients improve thei coaching teams as well as leading change programs.

About the conference

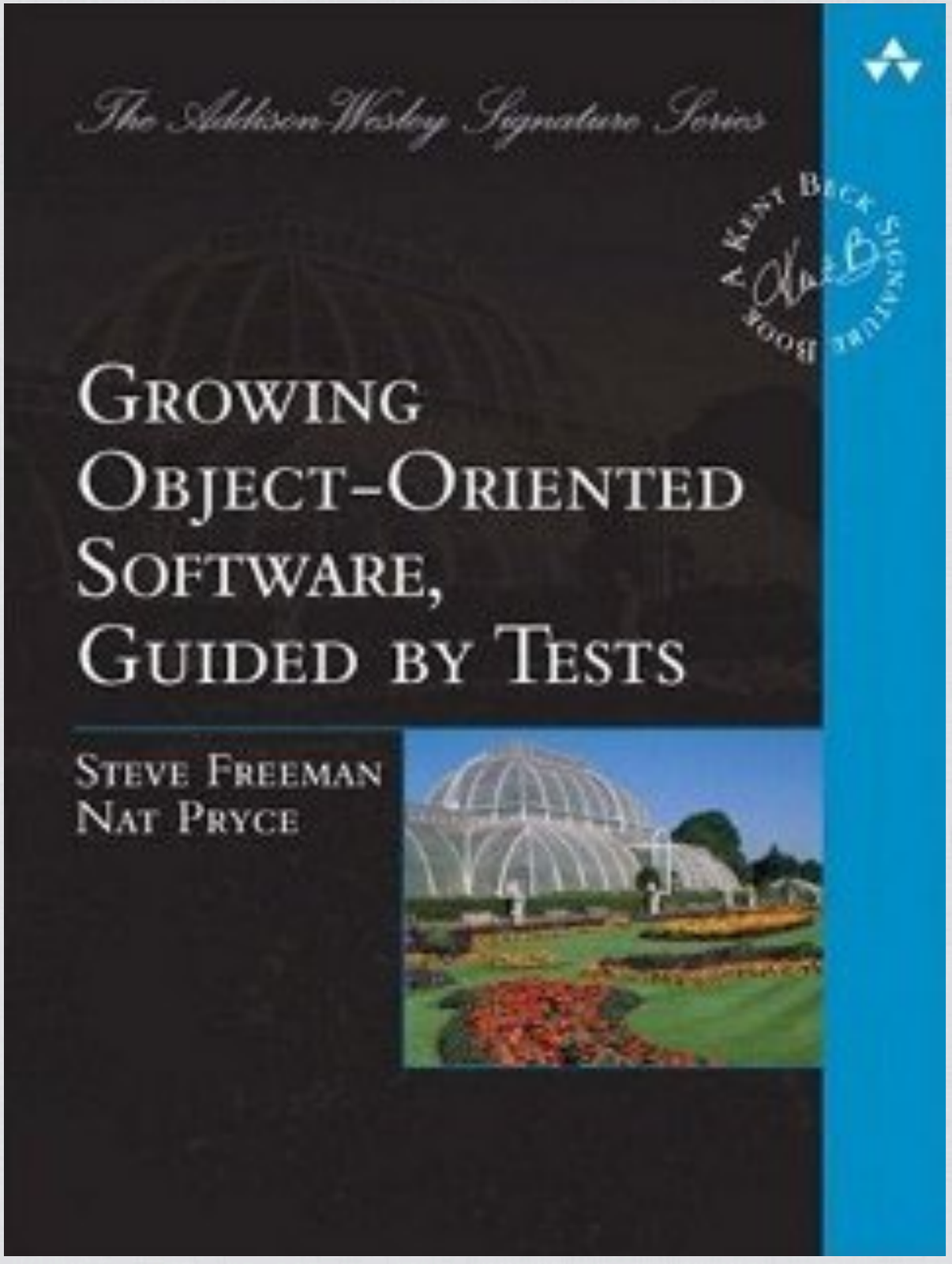
Agile 2009 is an exciting international industry conference

blip

Browse ▾

PyCon 2011: Units Need Testing Too

PyCon US Videos - 2009, 2010, 2011


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

stuff you can go and do

Keep logic out of classes that inherit from framework objects

Stop being tempted by "Fat Model"

isolate your logic

so you can

test logic in isolation

http://flic.kr/p/5BdRUw
http://flic.kr/p/nByrd

http://flic.kr/p/7pMSoJ

http://flic.kr/p/5jjATY

http://flic.kr/p/2vbxKn

http://flic.kr/p/mDdyJ

Q & A