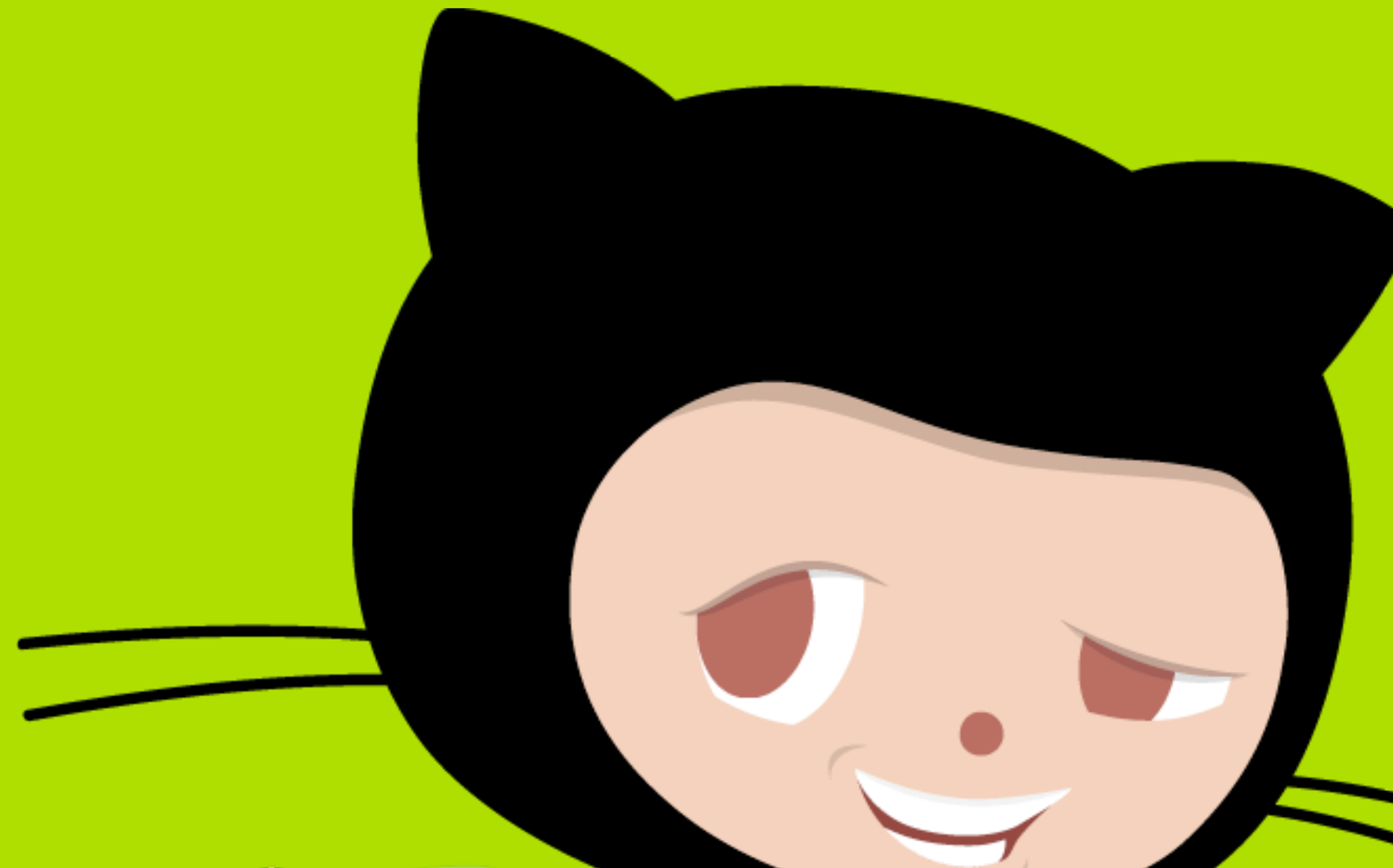
WHY OUR CODE SMELLS

http://www.flickr.com/photos/kanaka/3480201136/

github.com/bkeepers

@bkeepers



I AM TIRED OF WRITING BAD CODE.

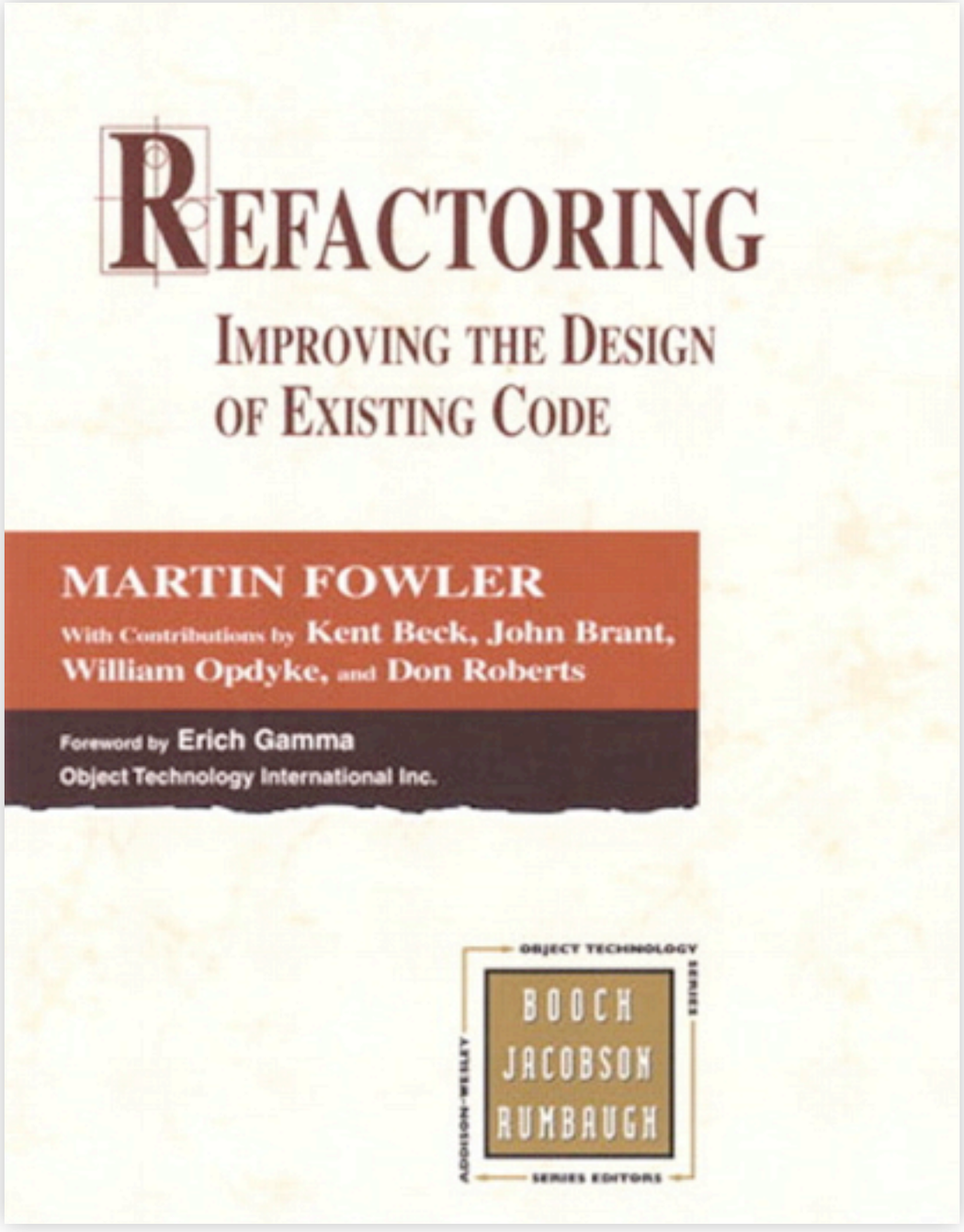
I AM TIRED OF MAINTAINING BAD CODE.

“Code doesn’t lie. If you’re not listening, you won’t hear the truths it tells.”

Kent Beck

Smalltalk Best Practice Patterns

A CODE SMELL
USUALLY INDICATES
A DEEPER PROBLEM
IN THE SYSTEM.


REFACTORING
IMPROVING THE DESIGN
OF EXISTING CODE
MARTIN FOWLER
With Contributions by Kent Beck, John Brant,
William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International Inc.
OBJECT TECHNOLOGY SERIES
BOOCH
JACOBSON
RUMBAUGH
ADDISON-WESLEY
SERIES EDITORS

CODE SMELLS ARE HEURISTICS TO SUGGEST WHEN TO REFACTOR AND WHAT TECHNIQUES TO USE.

DIVERGENT CHANGE

DIVERGENT CHANGE

Occurs when one class is commonly changed in different ways for different reasons. Any change to handle a variation should change a single class.

DIVERGENT CHANGE

Occurs when one class is commonly changed in different ways for different reasons. Any change to handle a variation should change a single class.

Refactoring:

Identify everything that changes for a particular cause and use Extract Class to put them all together.

OUR CODE SMELLS IN THE 21 WAYS THAT BECK AND FOWLER DESCRIBE, BUT…

our code smells when

UNIT TESTS ARE COMPLEX & SLOW.

IT IS TIGHTLY COUPLED TO A FRAMEWORK.

our code smells when

UNIT TESTS ARE COMPLEX.

TDD IS A DESIGN PROCESS.

WHY DO WE WRITE TESTS?

WHY DO WE WRITE TESTS?

1. Guard against regressions

WHY DO WE WRITE TESTS?

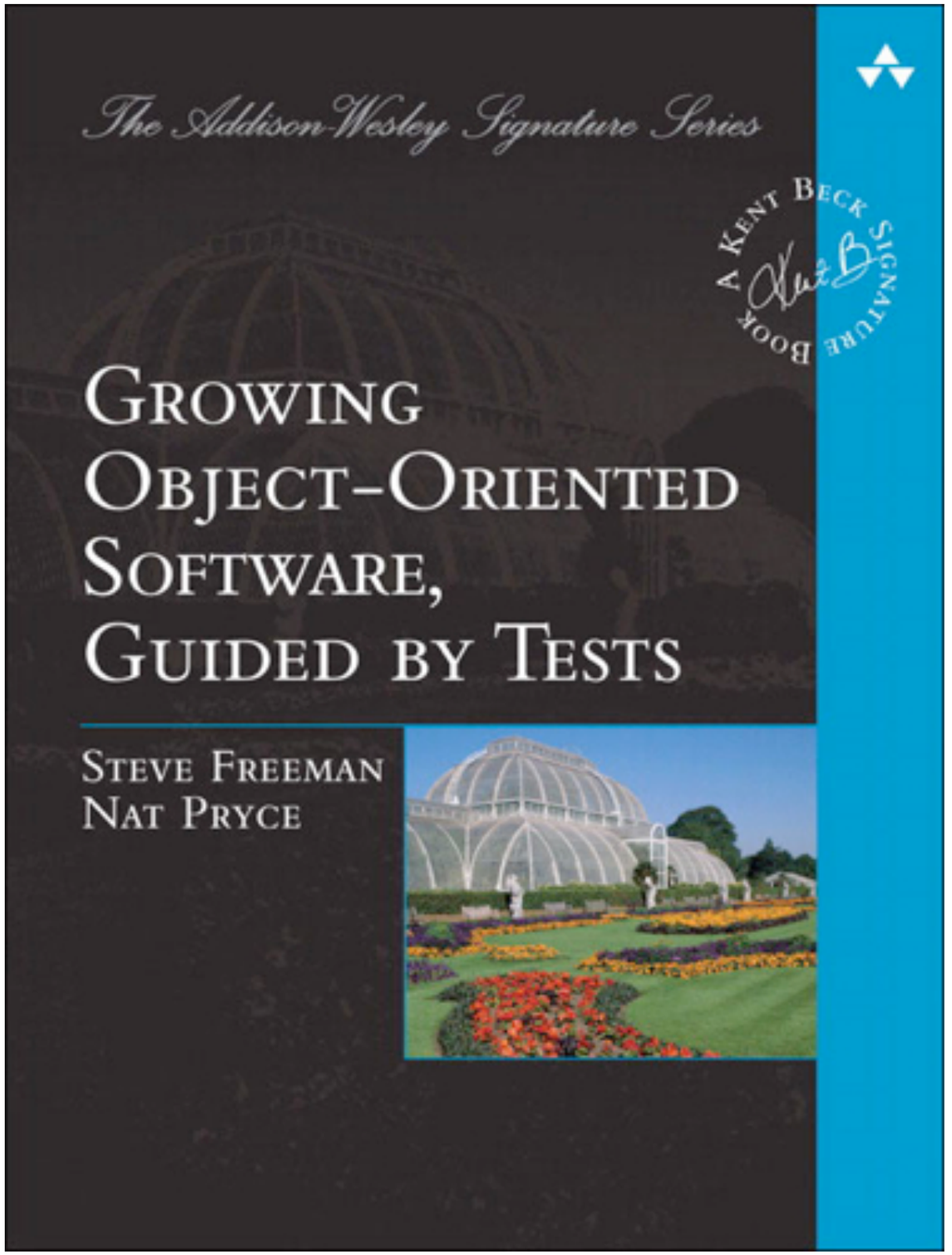
1. Guard against regressions
2. Gain confidence to change

WHY DO WE WRITE TESTS?

1. Guard against regressions
2. Gain confidence to change
3. Discover better designs

WARNING

Excessive use of quotations ahead.



“We find that the effort of writing a test first also gives us rapid feedback about the quality of our design ideas—that making code accessible for testing often drives it towards being cleaner and more modular.”

Steve Freeman, Nat Pryce

Growing Object-Oriented Software, Guided by Tests

unit tests can be complex when

OBJECTS ARE TOO TIGHTLY COUPLED.

Exhibit A: GitHub Notifications

```
context Notifications::Emails::Message do
  test "#to uses global email" do
    @settings.email :global, 'bkeepers@github.com'
    assert_equal 'bkeepers@github.com', @message.to
  end

  test "#body includes comment body" do
    assert_match @comment.body, @message.body
  end
end
```

Exhibit A: GitHub Notifications

```
context Notifications::Emails::Message do
  setup do
    @comment  = Issue.make
    @summary  = Notifications::Summary.from(@comment)
    @handlers = [Notifications::EmailHandler.new]
    @delivery = Notifications::Delivery.new(
                  @summary, @comment, @handlers)
    @settings = Notifications::Settings.new(-1)
    @message  = Notifications::Emails::Message.new(
                  @delivery, @settings)
  end

  test "#to uses global email" do
    @settings.email :global, 'bkeepers@github.com'
    assert_equal 'bkeepers@github.com', @message.to
  end
```

“When we find a feature that’s difficult to test, we don’t just ask ourselves how to test it, but also why is it difficult to test.”

Steve Freeman, Nat Pryce

Growing Object-Oriented Software, Guided by Tests

Exhibit A: GitHub Notifications

```
context Notifications::Emails::Message do
  setup do
    @comment  = Issue.make
    @summary  = Notifications::Summary.from(@comment)
    @handlers = [Notifications::EmailHandler.new]
    @delivery = Notifications::Delivery.new(
                  @summary, @comment, @handlers)
    @settings = Notifications::Settings.new(-1)
    @message  = Notifications::Emails::Message.new(
                  @delivery, @settings)
  end

  test "#to uses global email" do
    @settings.email :global, 'bkeepers@github.com'
    assert_equal 'bkeepers@github.com', @message.to
  end
end
```

Exhibit A: GitHub Notifications

```
context Notifications::Emails::Message do
  setup do
    @comment  = Issue.make



    @settings = Notifications::Settings.new(-1)
    @message  = Notifications::Emails::Message.new(
                  @comment, @settings)
  end

  test "#to uses global email" do
    @settings.email :global, 'bkeepers@github.com'
    assert_equal 'bkeepers@github.com', @message.to
  end
end
```

Exhibit A: GitHub Notifications

```
context Notifications::Emails::Message do
  setup do
    @comment  = Issue.make
    @settings = Notifications::Settings.new(-1)
    @message  = Notifications::Emails::Message.new(
                  @comment, @settings)
  end

  test "#to uses global email" do
    @settings.email :global, 'bkeepers@github.com'
    assert_equal 'bkeepers@github.com', @message.to
  end

  test "#body includes comment body" do
    assert_match @comment.body, @message.body
  end
end
```

unit tests can be complex when

OBJECTS ARE DOING TOO MUCH.

“An element’s cohesion is a measure of whether its responsibilities form a meaningful unit…Think of a machine that washes both clothes and dishes —it’s unlikely to do both well.”

Steve Freeman, Nat Pryce

Growing Object-Oriented Software, Guided by Tests

SINGLE RESPONSIBILITY PRINCIPLE

Every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class.

“Our heuristic is that we should be able to describe what an object does without using any conjunctions (‘and,’ ‘or’).”

Steve Freeman, Nat Pryce

Growing Object-Oriented Software, Guided by Tests

```
jQuery(function($) {
  $('#new-status').on('submit', function() {
    $.ajax({
      url: '/statuses',
      type: 'POST',
      dataType: 'json',
      data: {text: $(this).find('textarea').val()},
      success: function(data) {
        $('#statuses').append('<li>' + data.text + '</li>');
      }
    });
    return false;
  });
});
```

example lovingly stolen from @searls

The test tells me the code is complex.

```
describe("Updating my status", function() {
  var $form, $statuses;

  beforeEach(function(){
    $form = affix('form#new-status');
    $form.affix('textarea').val('sniffing code');
    $statuses = affix('#statuses');

    spyOn($, "ajax");

    $form.trigger('submit');
  });

  it("posts status to the server", function() {
    expect($.ajax).toHaveBeenCalledWith({
      url: '/statuses',
      data: {text: 'sniffing code'},
      success: jasmine.any(Function)
    });
  });
```

```
});

it("posts status to the server", function() {
  expect($.ajax).toHaveBeenCalledWith({
    url: '/statuses',
    data: {text: 'sniffing code'},
    success: jasmine.any(Function)
  });
});

describe("with a successful response", function() {
  beforeEach(function() {
    $.ajax.mostRecentCall.args[0].success({
      text: "This is starting stink!"
    });
  });

  it("appends text", function() {
    expect($statuses).toHaveHtml(
      '<div>This is starting stink!</div>');
  });
});
});
```

Why is this hard to test?

```
jQuery(function($) {
  $('#new-status').on('submit', function() {
    $.ajax({
      url: '/statuses',
      type: 'POST',
      dataType: 'json',
      data: {text: $(this).find('textarea').val()},
      success: function(data) {
        $('#statuses').append('<li>' + data.text + '</li>');
      }
    });
    return false;
  });
});
```

example lovingly stolen from @searls

Why is this hard to test?

1. page event

```
jQuery(function($) {
  $('#new-status').on('submit', function() {
    $.ajax({
      url: '/statuses',
      type: 'POST',
      dataType: 'json',
      data: {text: $(this).find('textarea').val()},
      success: function(data) {
        $('#statuses').append('<li>' + data.text + '</li>');
      }
    });
    return false;
  });
});
```

example lovingly stolen from @searls

Why is this hard to test?

1. page event

2. user event

```
jQuery(function($) {
  $('#new-status').on('submit', function() {
    $.ajax({
      url: '/st
      type: 'POST',
      dataType: 'json',
      data: {text: $(this).find('textarea').val()},
      success: function(data) {
        $('#statuses').append('<li>' + data.text + '</li>');
      }
    });
    return false;
  });
});
```

example lovingly stolen from @searls

Why is this hard to test?

1. page event

3. network IO

2. user event

```
jQuer                                    {
  $('#n  w-status').on('submit', function() {
    $.ajax({
      url: '/st
      type: 'POST',
      dataType: 'json',
      data: {text: $(this).find('textarea').val()},
      success: function(data) {
        $('#statuses').append('<li>' + data.text + '</li>');
      }
    });
    return false;
  });
});
```

example lovingly stolen from @searls

Why is this hard to test?

1. page event

3. network IO

2. user event

4. user input

```
jQuer                                    {
  $('#new-status').on('submit', function() {
    $.ajax({
      url: '/st
      type: 'POST',
      dataType: 'json',
      data: {text: $(this).find('textarea').val()},
      success: function(data) {
        $('#statuses').append('<li>' + data.text + '</li>');
      }
    });
    return false;
  });
});
```

example lovingly stolen from @searls

Why is this hard to test?

1. page event

3. network IO

2. user event

4. user input

5. network event

```
jQuer                                    {
  $('#n   w-status').on('submit', function() {
    $.ajax({
      url: '/st
      type: 'POST',
                'json',
      dat   : {text: $(this).find('textarea').val()},
      success: function(data) {
        $('#statuses').append('<li>' + data.text + '</li>');
      }
    });
    return false;
  });
});
```

example lovingly stolen from @searls

Why is this hard to test?

1. page event

3. network IO

2. user event

4. user input

5. network event

6. HTML templating

```
jQuer                                    {
  $('#n   w-status').on('submit', function() {
    $.ajax({
      url: '/st
      type: 'POST',
                'json',
      dat   : {text: $(this).find('textarea').val()},
      success: function(data) {
        $('#statuses').append('<li>' + data.text + '</li>');
      }
    });
    return false;
  });
});
```

example lovingly stolen from @searls

So we start to refactor…

```
jQuery(function($) {
  $('#new-status').on('submit', function() {
    $.ajax({
      url: '/statuses',
      type: 'POST',
      dataType: 'json',
      data: {text: $(this).find('textarea').val()},
      success: function(data) {
        $('#statuses').append('<li>' + data.text + '</li>');
      }
    });
    return false;
  });
});
```

Refactor to use a model

```
jQuery(function($) {
  $('#new-status').on('submit', function() {
    $.ajax({
      url: '/statuses',
      type: 'POST',
      dataType: 'json',
      data: {text: $(this).find('textarea').val()},
      success: function(data) {
        $('#statuses').append('<li>' + data.text + '</li>');
      }
    });
    return false;
  });
});
```

Refactor to use a model

```
jQuery(function($) {
  var statuses = new Collection.Statuses();

  $('#new-status').on('submit', function() {
    statuses.create({text: $(this).find('textarea').val()});
    return false;
  });

  statuses.on('add', function(status) {
    $('#statuses').append(
        '<li>' + status.get('text') + '</li>');
  });
});
```

Refactor to use a model

```
jQuery(function($) {
  var statuses = new Collection.Statuses();

  $('#new-status').on('submit', function() {
    statuses.create({text: $(this).find('textarea').val()});
    return false;
  });

  statuses.on('add', function(status) {
    $('#statuses').append(
        '<li>' + status.get('text') + '</li>');
  });
});
```

RESPONSIBILITY:

Sync state with server

Refactor handling of user input

```
jQuery(function($) {
  var statuses = new Collection.Statuses();

  $('#new-status').on('submit', function() {
    statuses.create({text: $(this).find('textarea').val()});
    return false;
  });

  statuses.on('add', function(status) {
    $('#statuses').append(
        '<li>' + status.get('text') + '</li>');
  });
});
```

Refactor handling of user input

```
jQuery(function($) {
  var statuses = new Collection.Statuses();

  new View.PostStatus({collection: statuses});

  statuses.on('add', function(status) {
    $('#statuses').append(
        '<li>' + status.get('text') + '</li>');
  });
});
```

Refactor handling of user input

```
jQuery(function($) {
  var statuses = new Collection.Statuses();

  new View.PostStatus({collection: statuses});

  statuses.on('add', function(status) {
    $('#statuses').append(
        '<li>' + status.get('text') + '</li>');
  });
});
```

RESPONSIBILITY:

Create statuses from user input

Refactor templating

```
jQuery(function($) {
  var statuses = new Collection.Statuses();

  new View.PostStatus({collection: statuses});

  statuses.on('add', function(status) {
    $('#statuses').append(
        '<li>' + status.get('text') + '</li>');
  });
});
```

Refactor templating

```
jQuery(function($) {
  var statuses = new Collection.Statuses();

  new View.PostStatus({collection: statuses});
  new View.StatusList({collection: statuses});
});
```

Refactor templating

```
jQuery(function($) {
  var statuses = new Collection.Statuses();

  new View.PostStatus({collection: statuses});
  new View.StatusList({collection: statuses});
});
```

RESPONSIBILITY:

Render statuses to the page

```
jQuery(function($) {
  var statuses = new Collection.Statuses();

  new View.PostStatus({collection: statuses});
  new View.StatusList({collection: statuses});
});
```

RESPONSIBILITY:

Initialize application on page load

Our tests only have one concern

```
describe("View.StatusList", function() {
  beforeEach(function() {
    $el = $('<ul></ul>');
    collection = new Backbone.Collection();
    view = new View.StatusList({
      el: $el,
      collection: collection
    });
  });

  it("appends newly added items", function() {
    collection.add({text: 'this is fun!'});
    expect($el.find('li').length).toBe(1);
    expect($el.find('li').text()).toEqual('this is fun!');
  });
});
```

PAY ATTENTION TO TESTING PAINS AND ADJUST THE DESIGN ACCORDINGLY.

Poor quality tests can slow development to a crawl, and poor internal quality of the system being tested will result in poor quality tests.

Steve Freeman, Nat Pryce

Growing Object-Oriented Software, Guided by Tests

“If you write bad unit tests, you might find that you gain none of the benefits, and instead are stuck with a bunch of tests that are time-consuming and hard to maintain.”

Christian Johansen

Test-Driven JavaScript Development

our code smells when

UNIT TESTS ARE SLOW.

```
$ bx rake test:units
...............................................................................................
...............................................................................................
...............................................................................................
...............................................................................................
...............................................................................................
...............................................................................................
...............................................................................................
...............................................................................................
...............................................................................................
...............................................................................................
...............................................................................................
...............................................................................................
...............................................................................................
...............................................................................................
...............................................................................................
...............................................................................................
...............................................................................................
...............................................................................................
...............................................................................................
...............................................................................................
...............................................................................................
...............................................................................................
...............................................................................................
........................................................................................
```

Finished in 274.623286 seconds.

2273 tests, 6765 assertions, 0 failures, 0 errors

WHAT'S WRONG WITH SLOW TESTS?

WHAT'S WRONG WITH SLOW TESTS?

You don't run them often

WHAT'S WRONG WITH SLOW TESTS?

You don't run them often

You waste time waiting for tests

WHAT'S WRONG WITH SLOW TESTS?

You don't run them often

You waste time waiting for tests

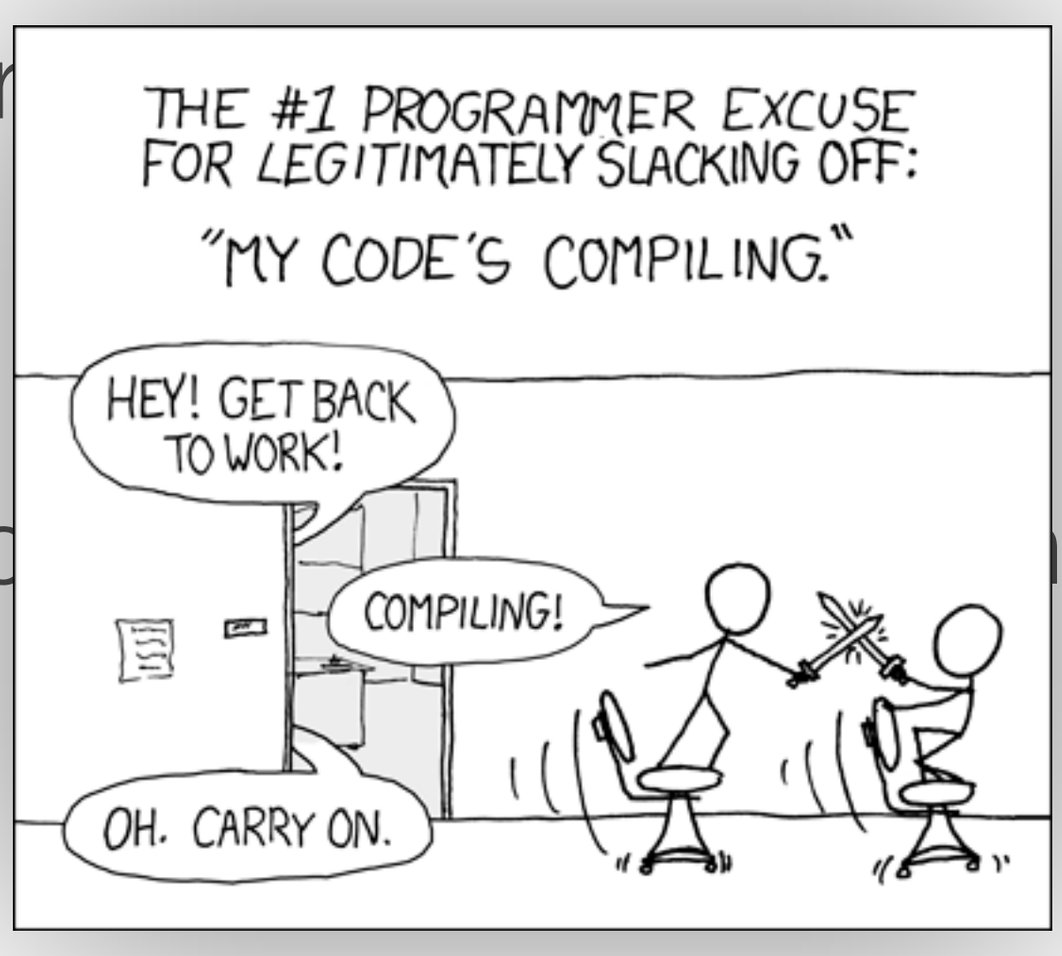
You distracted others while waiting

WHAT'S WRONG WITH SLOW TESTS?

You don't r

You waste

You distrac ng



http://xkcd.com/303/

WHAT'S WRONG WITH SLOW TESTS?

You don't run them often

You waste time waiting for tests

You distracted others while waiting

You commit failing changes

WHAT'S WRONG WITH SLOW TESTS?

You don't run them often

You waste time waiting for tests

You distracted others while waiting

You commit failing changes

You lose the rapid feedback cycle

unit tests can be slow when they

INTERACT WITH SLOW COMPONENTS.

```
context Notifications::Emails::Message do
  setup do
    @comment  = Issue.make! # create record in database
    @settings = Notifications::Settings.new(-1)
    @message  = Notifications::Emails::Message.new(
                  @comment, @settings)
  end

  test "#to uses global email" do
    @settings.email :global, 'bkeepers@github.com'
    assert_equal 'bkeepers@github.com', @message.to
  end

  test "#body includes comment body" do
    assert_match @comment.body, @message.body
  end
end
```

```
$ ruby test/unit/notifications/emails/message_test.rb
....................
Finished in 3.517926 seconds.

19 tests, 24 assertions, 0 failures, 0 errors
```

```
context Notifications::Emails::Message do
  setup do
    @comment    = Issue.make # create in memory
    @comment.id = -1         # make it appear persisted
    @settings   = Notifications::Settings.new(-1)
    @message    = Notifications::Emails::Message.new(
                    @comment, @settings)
  end

  test "#to uses global email" do
    @settings.email :global, 'bkeepers@github.com'
    assert_equal 'bkeepers@github.com', @message.to
  end

  test "#body includes comment body" do
    assert_match @comment.body, @message.body
  end
```

```
$ ruby test/unit/notifications/emails/message_test.rb
....................
Finished in 0.073752 seconds.

19 tests, 24 assertions, 0 failures, 0 errors
```

3.517926

÷ 0.073752

~50 X FASTER

unit tests can be slow when they

DON'T TEST OBJECTS IN ISOLATION.

```
context Notifications::Emails::CommitMention do
  setup do
    @repo = Repository.make!
    readonly_example_repo :notification_mentions, @repo
    @commit = @repo.commit('a62c6b20')

    @comment = CommitMention.new(:commit_id  => @commit.sha)

    @message = Emails::CommitMention.new(@comment)
  end

  test 'subject' do
    expected = "[testy] hello world (#{@comment.short_id})"
    assert_equal expected, @message.subject
  end
end
```

```
context Notifications::Emails::CommitMention do
  setup do
    @repo = Repository.make!
    readonly_example_repo :notification_mentions, @repo
    @commit = @repo.commit('a62c6b20')

    @comment = CommitMention.new(:commit_id  => @commit.sha)

    @message = Emails::CommitMention.new(@comment)
  end

  test 'subject' do
    expected = "[testy] hello world (#{@comment.short_id})"
    assert_equal expected, @message.subject
  end
end
```

```
context Notifications::Emails::CommitMention do
  setup do
    @commit = stub(
      :sha           => Sham.sha,
      :short_id      => '12345678',
      :short_message => 'hello world',
      :message       => 'goodbye world'
    )
    @comment = CommitMention.new(:commit_id => @commit.sha)
    @comment.stubs(:commit => @commit)

    @message = Emails::CommitMention.new(@comment)
  end

  test 'subject' do
    expected = "[testy] hello world (#{@comment.short_id})"
    assert_equal expected, message.subject
  end
```

BEFORE

```
$ ruby test/unit/notifications/emails/commit_mention_test.rb
....
Finished in 0.576135 seconds.
```

AFTER

```
$ ruby test/unit/notifications/emails/commit_mention_test.rb
....
Finished in 0.052412 seconds.
```

0.576135

÷ 0.052412

~10 X FASTER

unit tests can be slow when they

BOOTSTRAP HEAVY FRAMEWORKS.

```
$ time ruby test/unit/notifications/email_handler_test.rb
```

```
$ time ruby test/unit/notifications/email_handler_test.rb
........
Finished in 0.084729 seconds.

8 tests, 10 assertions, 0 failures, 0 errors
```

```
$ time ruby test/unit/notifications/email_handler_test.rb
........
Finished in 0.084729 seconds.

8 tests, 10 assertions, 0 failures, 0 errors

real    0m7.065s
user    0m4.948s
sys 0m1.961s
```

test/fast/notifications/web_handler.rb

```
require 'notifications/summary_store'
require 'notifications/memory_indexer'
require 'notifications/web_handler'

context Notifications::WebHandler do
  def web
    @web ||= WebHandler.new(
      :indexer => MemoryIndexer.new,
      :store   => SummaryStore.new
    )
  end

  def test_insert_increments_count
    assert_equal 0, web.count(1)
    web.add build_summary, 1
    assert_equal 1, web.count(1)
  end
```

```
$ time ruby test/fast/notifications/web_handler_test.rb
....
Finished in 0.001577 seconds.

4 tests, 22 assertions, 0 failures, 0 errors

real    0m0.139s
user    0m0.068s
sys 0m0.063s
```

7.065

÷ 0.139

~50 X FASTER

SLOW TEST MYTHS

SLOW TEST MYTHS

"Our tests are slow because we have too many of them."

SLOW TEST MYTHS

“Our tests are slow because we have too many of them.”

“To speed up our tests, we just need to parallelize them.”

SLOW TEST MYTHS

“Our tests are slow because we have too many of them.”

“To speed up our tests, we just need to parallelize them.”

“We can’t use test doubles because we’ll loose confidence that it still works.”

PAY ATTENTION TO THE SPEED OF YOUR TESTS AS YOU WRITE THEM.

our code smells when

IT IS TIGHTLY COUPLED TO A FRAMEWORK.

FRAMEWORKS ENCOURAGE YOU TO PUT ALL OF YOUR APPLICATION INSIDE THEIR SANDBOX.

...WHICH MAKES CODE DIFFICULT TO TEST, CHANGE AND REUSE.

assets

controllers

helpers

mailers

models

views

God Objects

God Objects

```
class Issue < ActiveRecord::Base
```

God Objects

```ruby
class Issue < ActiveRecord::Base
  # validations
  validates_presence_of :title, :user_id, :repository_id
```

God Objects

```ruby
class Issue < ActiveRecord::Base
  # validations
  validates_presence_of :title, :user_id, :repository_id

  # associations
  belongs_to :user
```

God Objects

```ruby
class Issue < ActiveRecord::Base
  # validations
  validates_presence_of :title, :user_id, :repository_id

  # associations
  belongs_to :user

  # data integrity
  before_validation :set_state
```

God Objects

```ruby
class Issue < ActiveRecord::Base
  # validations
  validates_presence_of :title, :user_id, :repository_id

  # associations
  belongs_to :user

  # data integrity
  before_validation :set_state

  # misc concerns
  before_save :audit_if_changed
```

God Objects

```
class Issue < ActiveRecord::Base
  # validations
  validates_presence_of :title, :user_id, :repository_id

  # associations
  belongs_to :user

  # data integrity
  before_validation :set_state

  # misc concerns
  before_save :audit_if_changed

  # querying
  named_scope :watched_by, lambda {|user| ... }
```

God Objects

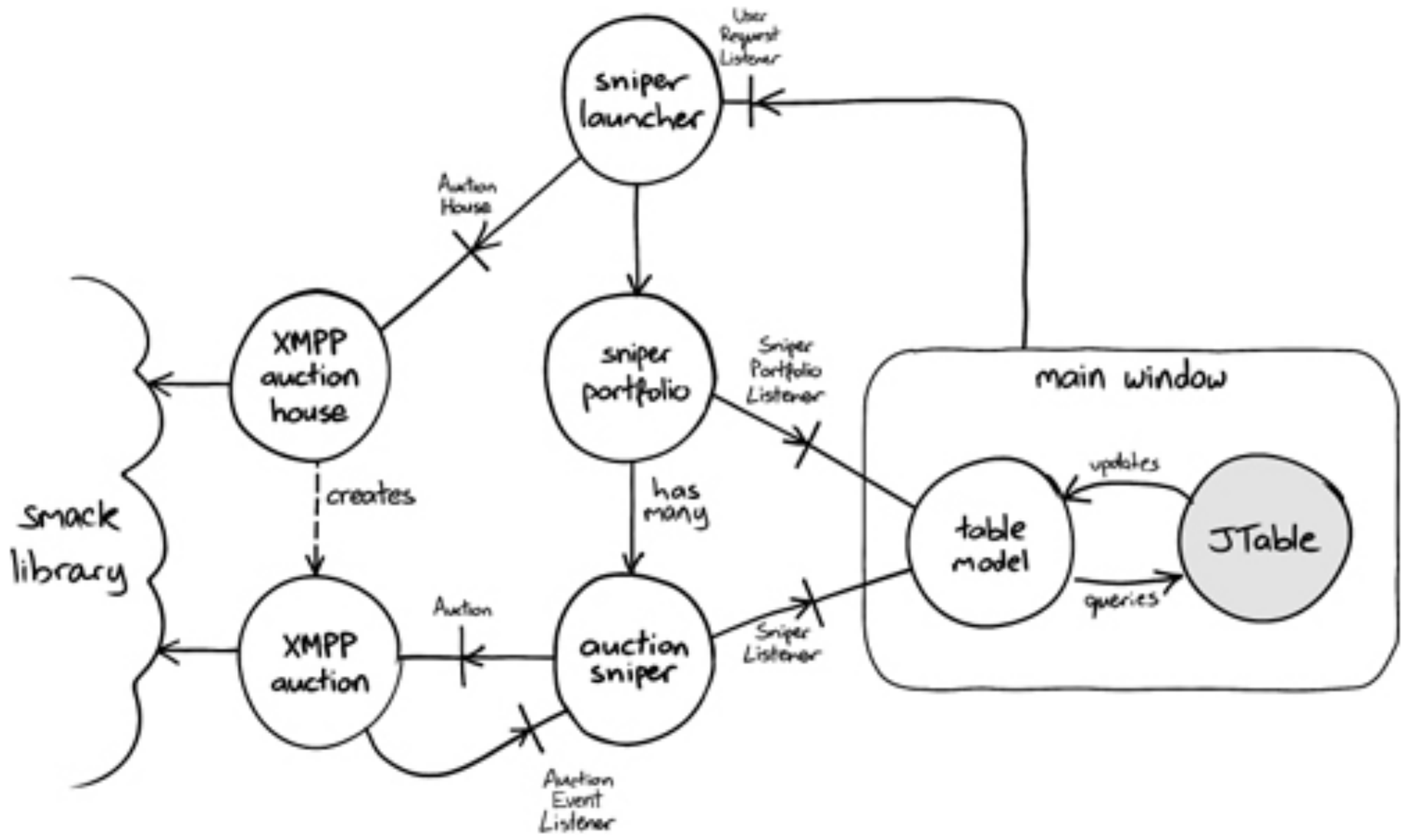
```ruby
class Issue < ActiveRecord::Base
  # validations
  validates_presence_of :title, :user_id, :repository_id

  # associations
  belongs_to :user

  # data integrity
  before_validation :set_state

  # misc concerns
  before_save :audit_if_changed

  # querying
  named_scope :watched_by, lambda {|user| ... }

  # who knows what these do?
  include Mentionable, Subscribable, Summarizable
```

God Objects

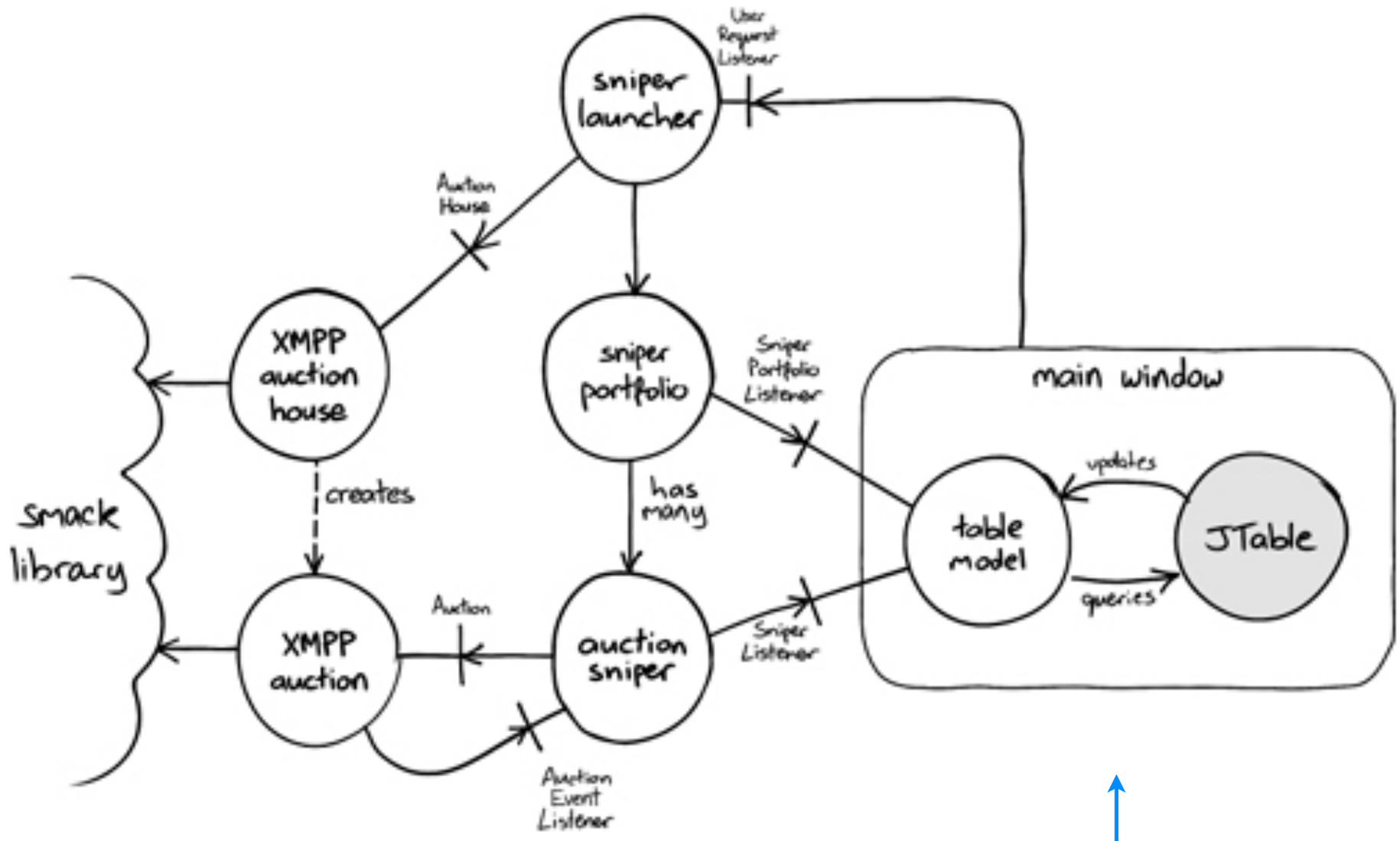
```ruby
class Issue < ActiveRecord::Base
  # validations
  validates_presence_of :title, :user_id, :repository_id

  # associations
  belongs_to :user

  # data integrity
  before_validation :set_state

  # misc concerns
  before_save :audit_if_changed

  # querying
  named_scope :watched_by, lambda {|user| ... }

  # who knows what these do?
  include Mentionable, Subscribable, Summarizable

  # domain logic
  def active_participants
    [self.user] + watchers + commentors
  end
end
```

MAKE THE FRAMEWORK DEPEND ON YOUR APPLICATION, INSTEAD OF MAKING YOUR APPLICATION DEPEND ON THE FRAMEWORK.

GOOS

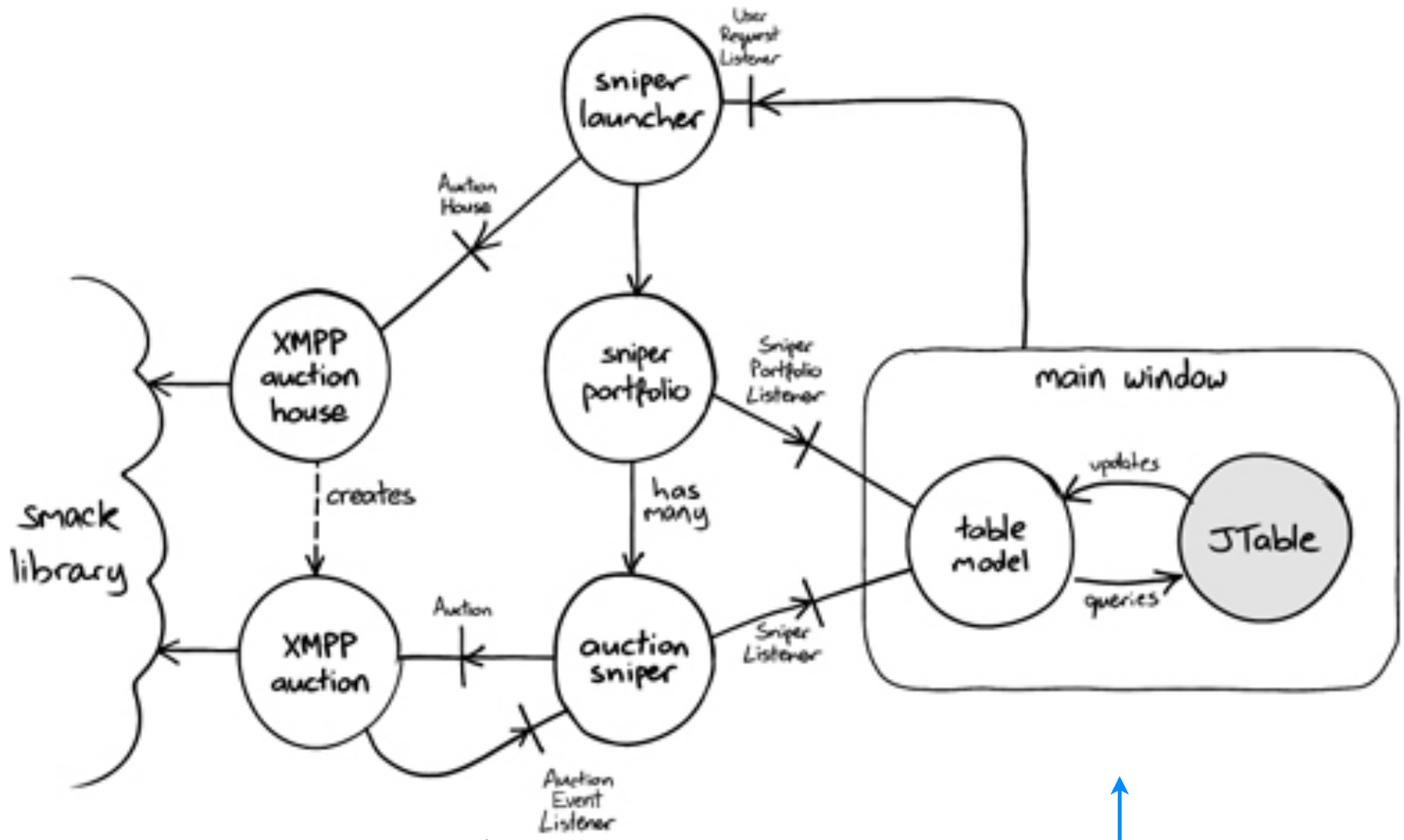

sniper launcher
User Request Listener
Auction House
XMPP auction house
sniper portfolio
Sniper Portfolio Listener
main window
table model
updates
JTable
queries
has many
creates
smack library
XMPP auction
Auction
auction sniper
Sniper Listener
Auction Event Listener

GOOS



coupled to a framework

GOOS



the rest of the application

coupled to a framework

A typical Rails controller…

```
class SessionsController < ApplicationController
  def create
    user = User.authenticate(params[:username],
                             params[:password])
    if user
      self.current_user = user
      redirect_to root_path, success: 'You are signed in!'
    else
      render :new, warning: 'Wrong username or password.'
    end
  end
end
```

…and model

```
class User < ActiveRecord::Base
  def self.authenticate(username, password)
    user = find_by_username(username)
    user if user && user.authenticated?(password)
  end

  def authenticated?(password)
    encrypt(password, self.salt) == self.encrypted_password
  end

  def encrypt(password, salt)
    Digest::SHA1.hexdigest(password+salt)
  end
end
```

Why does this depend on Active Record?

```
class User < ActiveRecord::Base
  def self.authenticate(username, password)
    user = find_by_username(username)
    user if user && user.authenticated?(password)
  end

  def authenticated?(password)
    encrypt(password, self.salt) == self.encrypted_password
  end

  def encrypt(password, salt)
    Digest::SHA1.hexdigest(password+salt)
  end
end
```

The spec is already complex.

```
describe User do
  # … a couple hundred lines of specs …
  describe ".authenticate" do
    let!(:user) do
      create :user, :email => "bkeepers", :password => "testing"
    end
    it "returns user with case insensitive username" do
      User.authenticate('BKeepers', 'testing').should == @user
    end
    it "returns nil with incorrect password" do
      User.authenticate("bkeepers", "wrong").should be_nil
    end
    it "returns nil with unknown username" do
      User.authenticate('foobar@foobar.com', 'testing').should be_nil
    end
  end
# … a couple hundred more lines of specs …
```

Create objects to **model** the domain

```
class SessionsController < ApplicationController
  def create
    user = PasswordAuthentication.new(params[:username],
                                      params[:password]).user
    if user
      self.current_user = user
      redirect_to root_path, success: 'You are signed in!'
    else
      render :new, warning: 'Wrong username or password.'
    end
  end
end
```

Plain ol’ Ruby class

```ruby
class PasswordAuthentication
  def initialize(username, password)
    @username = username
    @password = password
  end

  def user
  end
end
```

```ruby
require 'spec_helper'

describe PasswordAuthentication do
  describe 'user' do
    context 'with a valid username & password'
    context 'with an unknown username'
    context 'with an incorrect password'
  end
end
```

```ruby
describe PasswordAuthentication do
  describe 'user' do
    let!(:user) do
      create :user, :username => 'bkeepers',
                    :password => 'testing'
    end

    context 'with a valid username & password' do
      subject do
        PasswordAuthentication.new(user.username, 'testing')
      end

      it 'returns the user' do
        subject.user.should == user
      end
    end
end
```

```
class PasswordAuthentication
  def initialize(username, password)
    @username = username
    @password = password
  end

  def user
    User.find_by_username(@username)
  end
end
```

```
context 'with an unknown username' do
  subject do
    PasswordAuthentication.new('unknown', 'testing')
  end

  it 'returns nil' do
    subject.user.should be_nil
  end
end
```

No changes necessary

```
class PasswordAuthentication
  def initialize(username, password)
    @username = username
    @password = password
  end

  def user
    User.find_by_username(@username)
  end
end
```

```ruby
describe PasswordAuthentication do
  describe 'user' do
    context 'with a valid username & password' do # …
    context 'with an unknown username' do # …

    context 'with an incorrect password' do
      subject do
        PasswordAuthentication.new(user.username, 'wrong')
      end

      it 'returns nil' do
        subject.user.should be_nil
      end
    end
  end
end
```

```ruby
class PasswordAuthentication
  # …
  def user
    user = User.find_by_username(@username)
    user if user && authenticated?(user)
  end

private
  def authenticated?(user)
    encrypt(@password, user.password_salt) ==
      user.encrypted_password
  end

  def encrypt(password, salt)
    Digest::SHA1.hexdigest(password+salt)
  end
end
```

```
describe PasswordAuthentication do
  describe 'user' do
    let!(:user) do
      create :user, :username => 'bkeepers',
                    :password => 'testing'
      # hits the DB :(
    end

    # ...
  end
end
```

```
describe PasswordAuthentication do
  describe 'user' do
    let!(:user) do
      double :user,
        :username => 'bkeepers',
        :encrypted_password => '…',
        :password_salt => '…'
    end

    before do
      User.stub(:find_by_username).
           with(user.username).
           and_return(user)
    end
  end
end
```

```
  context 'with an unknown username' do
    before do
      User.should_receive(:find_by_username).
           with('unknown').
           and_return(nil)
    end

    subject do
      PasswordAuthentication.new('unknown', 'testing')
    end

    it 'returns nil' do
      subject.user.should be_nil
    end
  end
```

POSITIVE FEEDBACK LOOP

Unit tests help us isolate our code and reduce coupling to frameworks, which makes our tests faster.

Before we part ways

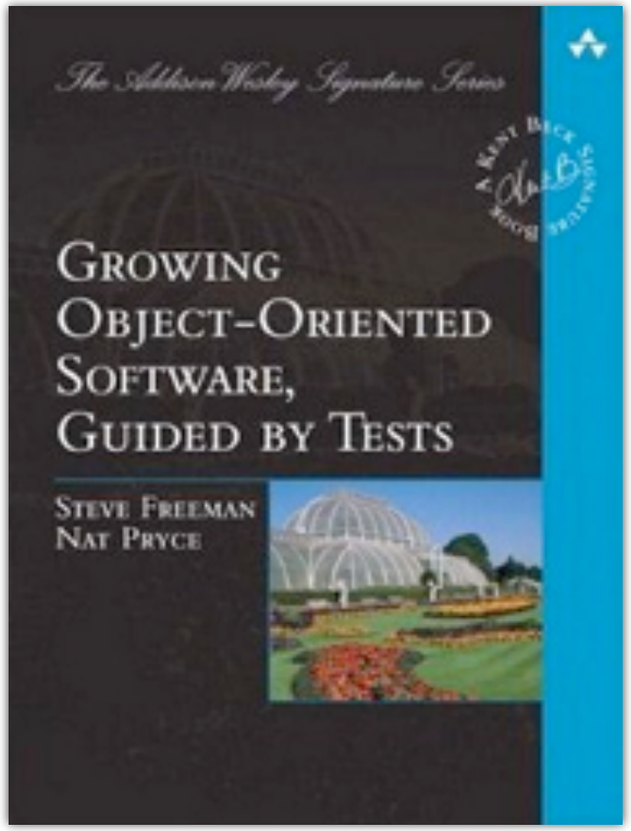
EPILOGUE

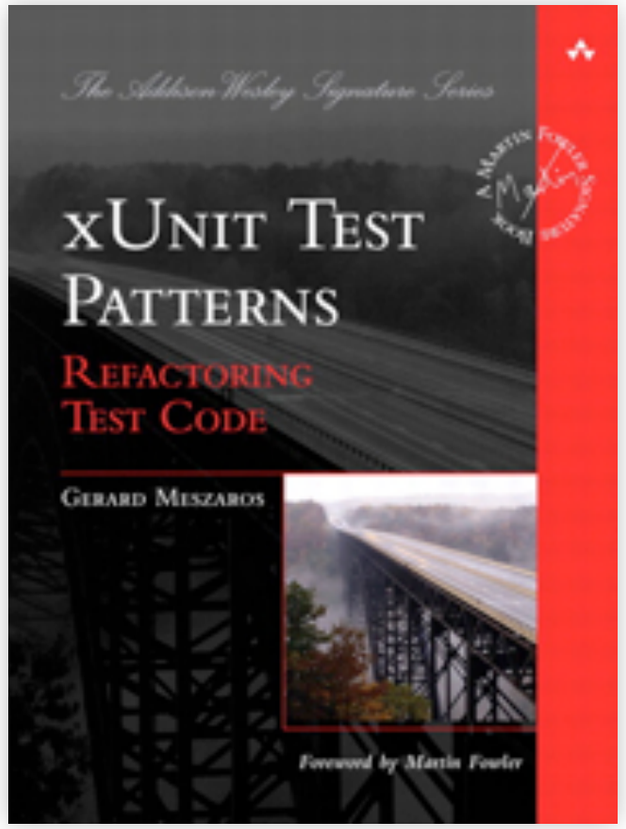
“Writing clean code requires the disciplined use of a myriad little techniques applied through a painstakingly acquired sense of ‘cleanliness.’”

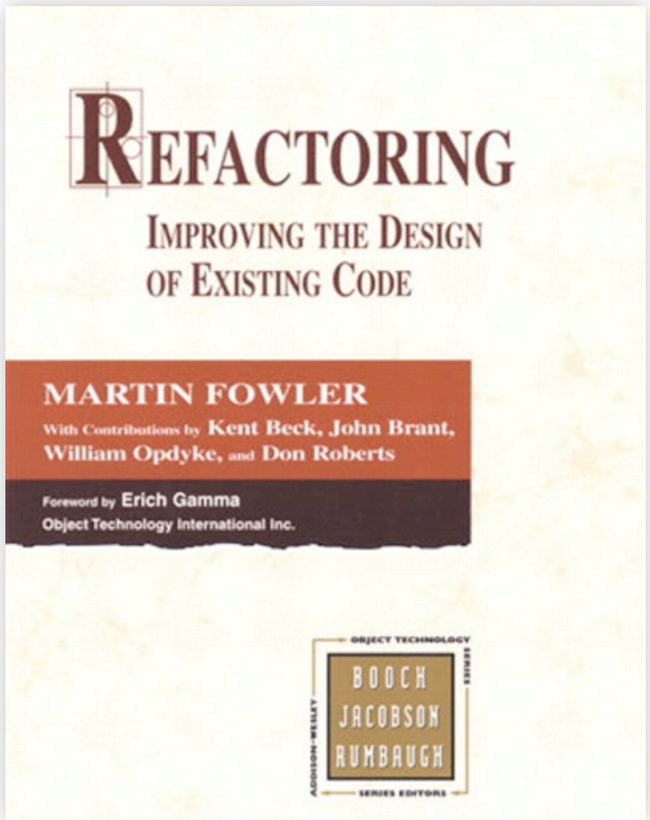
Robert C. Martin

Clean Code

REFERENCES


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

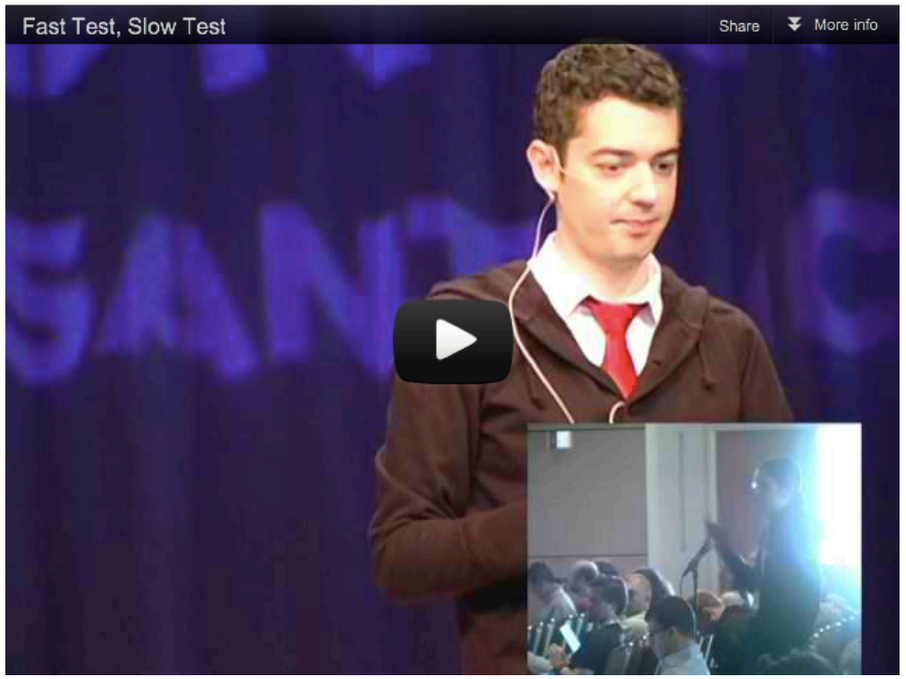

The Addison Wesley Signature Series
A Martin Fowler Signature Book
xUnit Test Patterns
Refactoring Test Code
Gerard Meszaros
Foreword by Martin Fowler


Refactoring
Improving the Design of Existing Code
Martin Fowler
With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International Inc.
Addison-Wesley
Object Technology Series
Booch
Jacobson
Rumbaugh
Series Editors


Robert C. Martin Series
Prentice Hall
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

REFERENCES

Corey Haines
Fast Rails Tests
http://confreaks.com/videos/641-gogaruco2011-fast-rails-tests



Gary Bernhardt
Fast test, Slow Test
http://pyvideo.org/video/631/fast-test-slow-test

QUESTIONS?

@bkeepers

http://bit.ly/smells-slides