Tactics for Taming Legacy Code

Jon Hope @ Storyful • 19.04.2016

When is it legacy?

“As soon as it’s written.”

When is it legacy?

> “Legacy code. The phrase strikes disgust in the hearts of programmers. It conjures images of slogging through a murky swamp of tangled undergrowth with leaches beneath and stinging flies above. It conjures odors of murk, slime, stagnancy, and offal. Although our first joy of programming may have been intense, the misery of dealing with legacy code is often sufficient to extinguish that flame.”
>
> Uncle Bob

When is it legacy?

Or why ageing Ruby projects can be problematic

- Loss of confidence in core conventions.

When is it legacy?

Loss of confidence in core conventions

```
class ApplicationController < ActionController::Base
  before_filter :user_sign_in_hack

  # ...
  def user_sign_in_hack
    if controller_name == 'sessions' && action_name == 'create'
      if params[:user].present? && params[:user][:email].present?
        params[:user][:email] = params[:user][:email].downcase
      end
    end
  end
end
```

When is it legacy?

Loss of confidence in core conventions

```
class ApplicationController < ActionController::Base
  before_filter :user_sign_in_hack

  # ...
  def user_sign_in_hack
    if controller_name == 'sessions' && action_name == 'create'
      if params[:user].present? && params[:user][:email].present?
        params[:user][:email] = params[:user][:email].downcase
      end
    end
  end
end
```

When is it legacy?

Or why ageing Ruby projects can be problematic

- Loss of confidence in core conventions.
- Lack of context around unusual design decisions.

When is it legacy?

Lack of context around unusual design decisions.

```
if Settings.ENVIRONMENT == 'staging'
  ActionMailer::Base.smtp_settings = {
    # staging mail setup
  }
elsif Rails.env.production?
   ActionMailer::Base.smtp_settings = {
     # production mail setup
   }
elsif Rails.env.test?
   # do nothing
end
```

When is it legacy?

Or why ageing Ruby projects can be problematic

- Loss of confidence in core conventions.
- Lack of context around unusual design decisions.
- Code that really was OK at the time.

When is it legacy?

Code that really was OK at the time

```
class User < ActiveRecord::Base
  attr_accessible :name, :email, :password, :password_confirmation, :username, :bio, :image
  #, ...
```

OR

```
class StoryCommentsController < ApplicationController
  respond_to :html, :json # DEPRECATED!

  def index
    @comments = Comment.all
    respond_with @comments
  end
end
```

When is it legacy?

Or why ageing Ruby projects can be problematic

- Loss of confidence in core conventions.
- Lack of context around unusual design decisions.
- Code that really was OK at the time.

The Event Horizon

The point at which the project will only degrade in code quality over time without conscious effort from maintainers.

Your changes don’t have to live forever.

They don’t have to go the whole way.

Quality metrics are now secondary, readability and future maintainability is key.

1. Being a good steward

Being a good steward

- Commenting the unintuitive.
- Putting stuff where it belongs.
- Cleaning up what you trip over.
- Making things obvious. (Or at least, more obvious).

Commenting the unintuitive.

```
if Settings.ENVIRONMENT == 'staging'
  ActionMailer::Base.smtp_settings = {
    # staging mail setup
  }
elsif Rails.env.production?
   ActionMailer::Base.smtp_settings = {
     # production mail setup
   }
elsif Rails.env.test?
   # do nothing
end
```

Commenting the unintuitive.

```
# HACK: The staging environment runs in production, so it's
# necessary to use Settings.ENVIRONMENT here.
#
if Settings.ENVIRONMENT == 'staging'
  ActionMailer::Base.smtp_settings = {
    # staging mail setup
  }
elsif Rails.env.production?
   ActionMailer::Base.smtp_settings = {
     # production mail setup
   }
elsif Rails.env.test?
   # do nothing
end
```

Putting stuff where it belongs.

```
class ApplicationController < ActionController::Base
  before_filter :user_sign_in_hack

  # ...
 def user_sign_in_hack
    if controller_name == 'sessions' && action_name == 'create'
      if params[:user].present? && params[:user][:email].present?
        params[:user][:email] = params[:user][:email].downcase
      end
    end
  end
end
```

Putting stuff where it belongs.

```
class SessionsController < ApplicationController
  before_filter :user_sign_in_hack, only: :create

  # ...
  def user_sign_in_hack
    if controller_name == 'sessions' && action_name == 'create'
      if params[:user].present? && params[:user][:email].present?
        params[:user][:email] = params[:user][:email].downcase
      end
    end
  end
end
```

Cleaning up what you trip over.

```
class SessionsController < ApplicationController
  before_filter :user_sign_in_hack, only: :create

  # ...
  def user_sign_in_hack
    if controller_name == 'sessions' && action_name == 'create'
      if params[:user].present? && params[:user][:email].present?
        params[:user][:email] = params[:user][:email].downcase
      end
    end
  end
end
```

Cleaning up what you trip over.

```
class SessionsController < ApplicationController
  before_action :downcase_email, only: :create

  # ...
  def downcase_email
    if params[:user].present? && params[:user][:email].present?
      params[:user][:email] = params[:user][:email].downcase
    end
  end
end
```

Making things obvious.

```
def index
  @stories = if params[:q].present?
    story_search_options = {
      # ...
    }
    StorySearch.new(params[:q], story_search_options).execute!
  elsif params[:popular_only].present?
    Story
      .joins(:popularity_profile)
      .where(popularity_profile: { current: true })
      # ...
  else
    Story.published
  end
   respond_with @stories
end
```

Making things obvious.

```
def index
  @stories = stories_by_search || popular_stories || Story.published

   respond_with @stories
end

def stories_by_search
  return unless params[:q].present?

  story_search_options = { … }
  StorySearch.new(params[:q], story_search_options).execute!
end

def popular_stories
  return unless params[:popular_only].present?

  Story.joins(:popularity_profile) # …
end
```

1. Good Stewardship

2. Stubbing dependencies

```ruby
class Schedule < ActiveRecord::Base
  has_many :shifts, :dependent => :delete_all, :inverse_of => :schedule
  has_many :staff_memberships, :through => :shifts, :uniq => true

  after_save :email_shifts, :if => :just_published?

  # ...

  def just_published?
    self.published_changed? && !self.published_was && !self.published.nil?
  end

  def email_shifts
   staff_memberships.includes(:user).each do |staff_membership|
     @shifts = shifts.select {|s| s.staff_membership_id == staff_membership.id}.sort_by(&:start_time)
     unless !staff_membership.confirmed? || staff_membership.user.email.blank?
       ScheduleMailer.user_shifts_notification(
         self, staff_membership, staff_membership.user.email, @shifts
       ).deliver
     end
   end
 end
```

```ruby
class Schedule < ActiveRecord::Base
  has_many :shifts, :dependent => :delete_all, :inverse_of => :schedule
  has_many :staff_memberships, :through => :shifts, :uniq => true

  after_save :email_shifts, :if => :just_published?

  # ...

  def just_published?
    self.published_changed? && !self.published_was && !self.published.nil?
  end

  def email_shifts
    staff_memberships.includes(:user).each do |staff_membership|
      @shifts = shifts.select {|s| s.staff_membership_id == staff_membership.id}.sort_by(&:start_time)
      unless !staff_membership.confirmed? || staff_membership.user.email.blank?
        ScheduleMailer.user_shifts_notification(
          self, staff_membership, staff_membership.user.email, @shifts
        ).deliver
      end
    end
  end
```

```ruby
describe Schedule do
  let(:schedule_mailer_stub) { stub_const("ScheduleMailer", subject) }

  before do
    allow(schedule_mailer_stub).to receive(:user_shifts_notification).and_return(nil)
  end

  # ...
```

```
describe Schedule do
  let(:schedule_mailer_stub) { stub_const("ScheduleMailer", subject) }
  let(:deliverable_stub) { double("deliverable") }

  before do
    allow(schedule_mailer_stub).to receive(:user_shifts_notification).and_return(deliverable_stub)
    allow(deliverable_stub).to receive(:deliver).and_return(true)
  end
```

```
let(:schedule_mailer_stub) { stub_const("ScheduleMailer", subject) }
let(:deliverable_stub) { double("deliverable") }
```

Then:

```
describe '#publish! do
  after { subject.publish! }

  it 'sends messages to schedule mailer' do
    expect(schedule_mailer_stub).to receive(:user_shifts_notification).with( … )
    expect(deliverable_stub).to receive(:deliver)
  end
end
```

But...

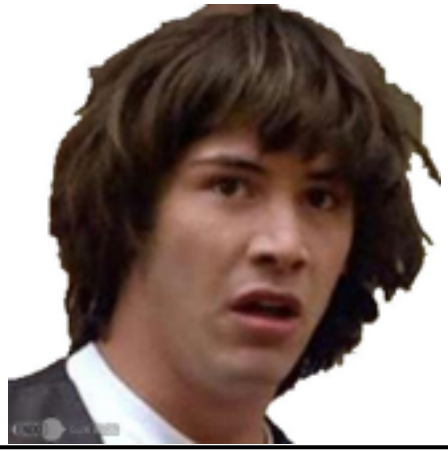
But…

What if the ScheduleMailer goes away…?

But…

What if the ScheduleMailer goes away…?

What if the ScheduleMailer never existed…?



Include a separate test for the ScheduleMailer!

```
describe Schedule do
  it 'has a ScheduleMailer constant' do
    expect(described_class.const_defined?("ScheduleMailer")).to be true
  end

  describe '::ScheduleMailer' do
    subject { described_class::ScheduleMailer }

    it 'responds to user_shifts_notification' do
      expect(subject).to respond_to(:user_shifts_notification)
    end
  end
end
```

2. Stubbing dependencies

3. Splitting Complex Methods

```ruby
class User < ActiveRecord::Base
  def copy_from(company_user, options = {})
    attributes_to_copy = [:expiration_at, :subscription_level, :has_full_access, :storify_enabled]

    attributes_to_copy.each do |attribute_to_copy|
      self[attribute_to_copy] = options[attribute_to_copy]
      self[attribute_to_copy] ||= company_user[attribute_to_copy]
    end
    self.save!
    User.transaction do
      company_user.layout_configs.each do |layout_config|
        user_layout_config_dupped = layout_config.dup
        user_layout_config_dupped.user_id = self.id
        self.layout_configs << user_layout_config_dupped
      end
      company_user.subscriptions.each do |subscription|
        subscription_dupped = subscription.dup
        subscription_dupped.user_id = self.id
        subscription_dupped.save
      end
    end
  end
end
```

(Copy attributes)

(Copy layout configs)

(Copy subscriptions)

```
def copy_from(company_user, options = {})
  attributes_to_copy = [:expiration_at, :subscription_level, :has_full_access, :storify_enabled]

  attributes_to_copy.each do |attribute_to_copy|
    self[attribute_to_copy] = options[attribute_to_copy]
    self[attribute_to_copy] ||= company_user[attribute_to_copy]
  end
  self.save!
  User.transaction do
    company_user.layout_configs.each do |layout_config|
      user_layout_config_dupped = layout_config.dup
      user_layout_config_dupped.user_id = self.id
      self.layout_configs << user_layout_config_dupped
    end
    company_user.subscriptions.each do |subscription|
      subscription_dupped = subscription.dup
      subscription_dupped.user_id = self.id
      subscription_dupped.save
    end
  end
end
```

(Copy attributes)

(Copy layout configs)

(Copy subscriptions)

```ruby
def copy_attributes_from(company_user, options = {})
  attributes_to_copy = [:expiration_at, :subscription_level, :has_full_access, :storify_enabled]

  attributes_to_copy.each do |attribute_to_copy|
    self[attribute_to_copy] = options[attribute_to_copy]
    self[attribute_to_copy] ||= company_user[attribute_to_copy]
  end
  self.save!
end

def copy_layout_configs_from(source_user)
  source_user.layout_configs.each do |layout_config|
    user_layout_config_dupped = layout_config.dup
    user_layout_config_dupped.user_id = self.id
    self.layout_configs << user_layout_config_dupped
  end
end

def copy_subscriptions_from(source_user)
  # etc ...
end
```

```
describe "#copy_layout_configs_from" do
  let(:source_user) { build_stubbed(:user) }
  let(:layout_config) { build_stubbed(:layout_config) }



  subject { create(:user) }

  before do
    allow(source_user).to receive(:layout_configs).and_return([layout_config])

  end

 after { subject.copy_layout_configs_from(source_user)) }

 it "sends correct messages" do
   expect(source_user).to receive(:layout_configs)
   expect(layout_config).to receive(:dup)

 end
end
```

```ruby
describe "#copy_layout_configs_from" do
  let(:source_user) { build_stubbed(:user) }
  let(:layout_config) { build_stubbed(:layout_config) }

  let(:layout_configs_double) { double('layout_configs') }

  subject { create(:user) }

  before do
    allow(source_user).to receive(:layout_configs).and_return([layout_config])
    allow(subject).to receive(:layout_configs).and_return(layout_configs_double)
  end

  after { subject.copy_layout_configs_from(source_user)) }

  it "sends correct messages" do
    expect(source_user).to receive(:layout_configs)
    expect(layout_config).to receive(:dup)
    expect(layout_configs_double).to receive(:<<)
  end
end
```

Cons

- Increased class size.
- Increased number of messages required to perform actions.

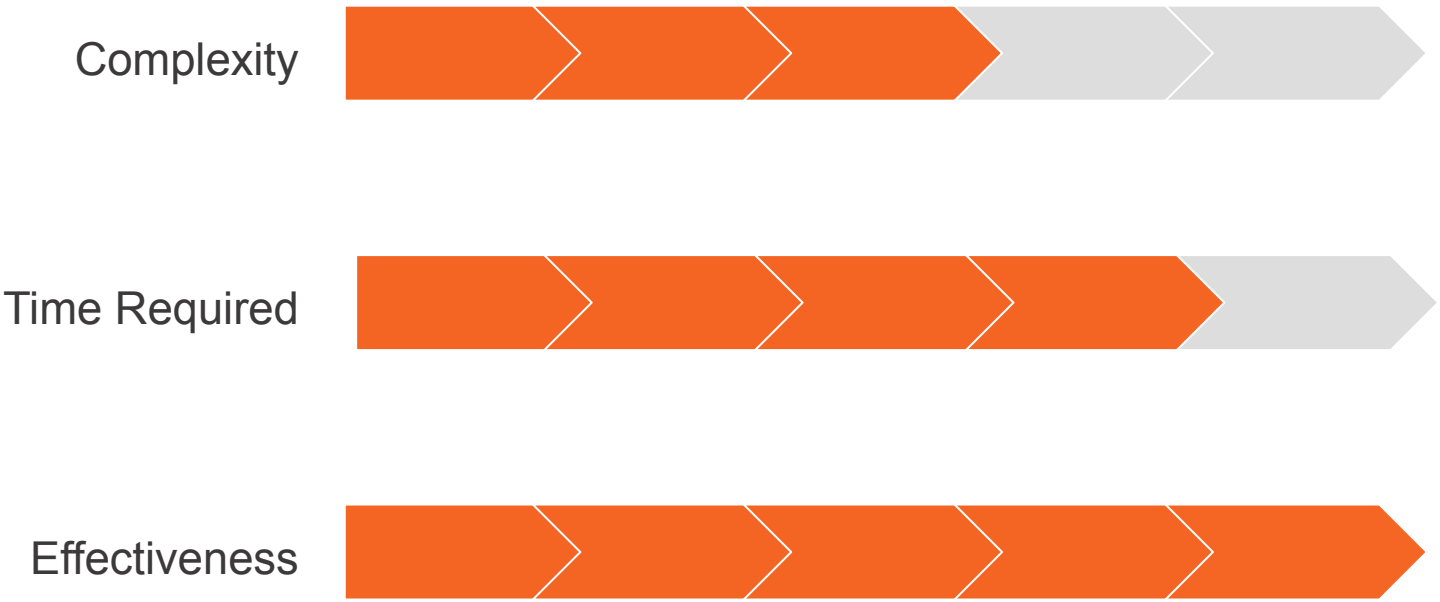
Cons

- Increased class size.
- Increased number of messages required to perform same actions.

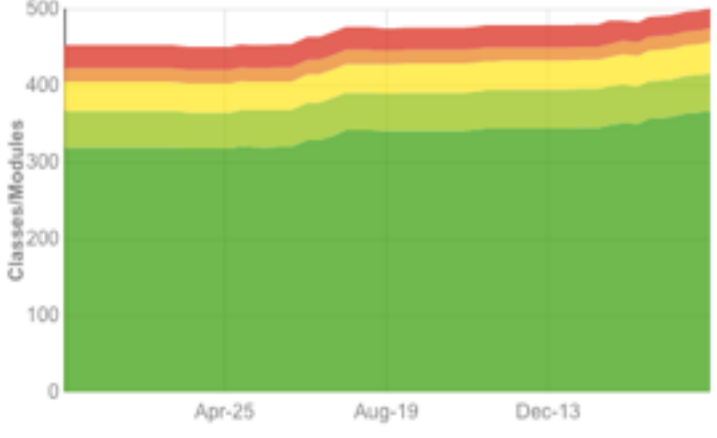
Pros

- Forced dependencies into the open.
- Small, descriptive methods that self-document.
- Easier to test; double dependencies & check for messages sent.
- Easy to refactor into a Factory or Concern in the future.

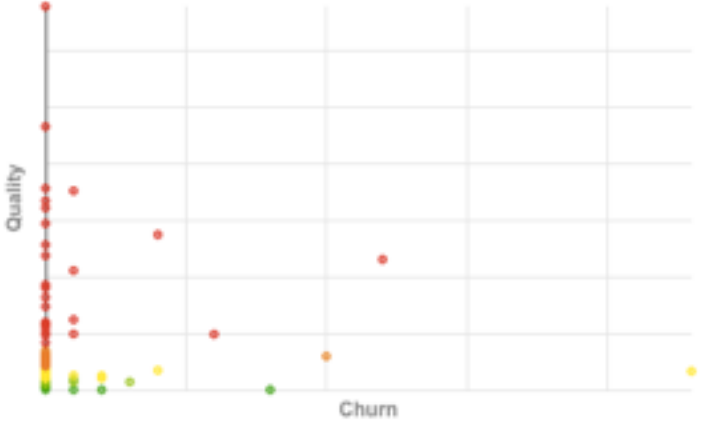
3. Splitting Complex Methods



Breakdown Over Time



Churn vs. Quality



Contact Me

Twitter: @midhir

Github: https://github.com/JonMidhir

Web: jhope.ie

Email: me@jhope.ie or john.hope@storyful.com

Further Reading

Maintaining Your Legacy (Scott Ford)

Episode 124 - Giant Robots Smashing into Other Giant Robots
http://giantrobots.fm/124

Working Effectively with Legacy Code

Michael Feathers (ISBN 0131177052)