

Refactoring Your Rails Application

RailsConf 2008 - Portland, OR

Zach Dennis
Mutually Human Software
www.mutuallyhuman.com

Drew Colthorp
Atomic Object
www.atomicobject.com

Table of Contents

<u>Tutorial Setup Instructions</u>	1
<u>Thanks for signing up!</u>	1
<u>Refactoring</u>	2
<u>The First Step In Refactoring</u>	2
<u>Summary of Concepts and Terms</u>	3
<u>Single Responsibility Principle</u>	3
<u>Separation of Concerns</u>	3
<u>Law of Demeter</u>	3
<u>YAGNI</u>	3
<u>Service</u>	3
<u>Intention Revealing Interface</u>	4
<u>Tell, Don't Ask</u>	4
<u>Code Smells</u>	5
<u>Duplicated code</u>	5
<u>Shotgun Surgery</u>	5
<u>Long Method</u>	5
<u>Inappropriate Intimacy</u>	5
<u>Divergent Change</u>	5
<u>Large Class</u>	5
<u>Feature Envy</u>	5
<u>Message Chains</u>	5
<u>Comments</u>	6
<u>Rails Presenter Pattern</u>	7
<u>What is a Presenter?</u>	7
<u>Alternative Implementations</u>	7
<u>References</u>	8
<u>Refactorings</u>	9
<u>Basic Building Blocks for Rails Refactorings</u>	9
<u>Replace Instance Variable With Local Variable in Partial</u>	10
<u>Extract Query to Model</u>	12
<u>Extract Renderer From RJS</u>	14
<u>Move Data From View Into Presenter</u>	18
<u>Move Logic From View Into Presenter</u>	23
<u>Extract Complex Creation Into Factory</u>	26
<u>Introduce Result Object</u>	29
<u>Extract Inline Update to RJS File</u>	33
<u>Extract Operations Into Service</u>	36
<u>Move Before Filter and Friends to Application Service</u>	40
<u>Move View Data From Controller Into Presenter</u>	47
<u>Move Model Logic Into the Model</u>	48
<u>Move View Logic From Controller Into Presenter</u>	51
<u>Move Extrinsic Callback Into Observer</u>	54
<u>Move View Logic From Model Into Presenter</u>	56

Table of Contents

Refactorings

<u>Organize Large Model with Mixins</u>	59
---	----

Tutorial Setup Instructions

This tutorial will use an example agile project management application. It was developed on Mac OSX with MySQL 5 and will be bundled with Rails 2.0.2, but should work fine if you have MySQL 4.1 or higher installed. If you do not have MySQL installed you should be able to do the majority of the tutorial although you will lose an important capability in the example app.

To get the example application, download it from http://www.mutuallyhuman.com/example_app.zip

If that download link doesn't work then you can download it from http://www.continuousthinking.com/example_app.zip

To setup the application:

- unzip example_app.zip
- copy config/database.yml.template to config/database.yml
- edit config/database.yml to your liking
- rake db:create:all
- rake db:migrate
- rake db:test:prepare
- script/server
- You should be able to go to <http://localhost:3000> and login with email "admin@example.com" and password "password".

Thanks for signing up!

Thank for signing up for this tutorial. I hope that you are able to take a lot away from it and that you find this Rails Refactoring catalog as a useful resource. If you have any questions please feel free to let me know. Thanks,

Zach Dennis
zdennis@mutuallyhuman.com
Mutually Human Software

Refactoring

“Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.”

Martin Fowler in his book Refactoring: Improving the Design of Existing Code.

The First Step In Refactoring

The first step in refactoring is to ensure you have a solid set of tests for the section of code in question. This will ensure that you don't unintentionally introduce bugs when making changes. We are human after all.

Testing For This Tutorial and Workbook

I cannot emphasize enough how important it is to have a solid set of tests covering the feature you are about refactor. With that being said, this tutorial and workbook are not introductions to testing. After each logical step of refactoring you should run your tests before moving onto the next step.

For example, the catalog of Rails refactorings found later in this workbook do not walk the user through ensuring the associated tests are carried over when you move a method. When you move a method, create a class or extract some behavior, testing is just a part of the process.

When Automated Testing Isn't Practical

While having an automated test suite is preferred, it is not always practical in a Rails application. For example, refactorings which improve the code associated with an RJS response cannot be fully tested without the use of an in-browser testing framework like Selenium. When in doubt having a solid set of unit tests as well as a manual test plan for these types of scenarios will help ensure that you aren't breaking functionality or introducing bugs.

Summary of Concepts and Terms

Single Responsibility Principle

The Single Responsibility Principle (SRP) states that every object should have a single responsibility and all of its services should be narrowly aligned with that responsibility.

Uncle Bob Martin re-interpreted SRP to mean that an object should have only one reason to change.

- http://en.wikipedia.org/wiki/Single_responsibility_principle
- <http://www.objectmentor.com/resources/articles/srp.pdf>

Separation of Concerns

Separation of Concerns (SoC) is the process of breaking a program down into distinct features that overlap in functionality as little as possible. A “concern” is any piece of interest or focus in a program. Concerns are usually synonymous with features and behaviors. SoC is usually achieved through modularity and encapsulation.

- http://en.wikipedia.org/wiki/Separation_of_concerns

Law of Demeter

The Law of Demeter (LoD) is a guideline encouraging objects to only talk to their immediate friends. This discourages method chaining.

- The Pragmatic Programmer by Dave Thomas / Andy Hunt
- http://en.wikipedia.org/wiki/Law_of_Demeter
- <http://www.ccs.neu.edu/home/lieber/LoD.html>
- <http://c2.com/cgi/wiki?LawOfDemeter>

YAGNI

You Aren't Going to Need It. YAGNI emphasizes implementing things when you need them, not when you think you need them. Typically what happens is you end up not needing what you thought you needed, or you need something different.

- <http://c2.com/xp/YouArentGonnaNeedIt.html>
- http://en.wikipedia.org/wiki/You_Ain't_Gonna_Need_It

Service

A Service is an operation that stands alone in the model with no encapsulated state. A Service tends to be named for an activity rather than an entity.

- Domain Driven Design: Tackling Complexity in the Heart of Software by Eric Evans

Intention Revealing Interface

An intention revealing interface is one that clearly conveys its purpose and meaning to those reading the code. In other words, it does what it says it does. It's also very often written in terms of the problem domain instead of implementation details.

- Domain Driven Design: Tackling Complexity in the Heart of Software by Eric Evans

Tell, Don't Ask

"Tell, Don't Ask" states that you should tell an object to do something, not ask questions about its state and then make a decision and then tell them what to do.

- <http://www.pragprog.com/articles/tell-dont-ask>

Code Smells

A code smell is any symptom that indicates something may be wrong. It generally indicates that the code should be refactored or the overall design should be re-examined. —http://en.wikipedia.org/wiki/Code_smell

This list summarizes common code smells found in Rails applications. For a more thorough list of code smells please refer to Martin Fowler's book [Refactoring: Improving The Existing Design Of Code](#) and <http://www.refactoring.com>.

Duplicated code

This appears in code when you have the same code structure in more than one place.

Shotgun Surgery

This is when some changes require modifying many different parts of your application.

Long Method

Long methods are harder to understand and obfuscate their purpose and behavior.

Inappropriate Intimacy

An object is too familiar with the implementation of another. If your view's executing SQL, it's coupled too tightly with your model.

Divergent Change

A class or mixin should only have one reason to change. An object which does not meet this criteria violates the Single Responsibility Principle.

Large Class

This occurs when a class is trying to do too much.

Feature Envy

This is when a method is more interested in another class than its own.

Message Chains

This is when you get an object, get another object from it, and get another object from the result, etc. Navigating this way means the client is coupled to the structure of the navigation. This indicates a Law of Demeter violation.

Comments

Comments themselves aren't inherently bad, but they are often used to clarify bad code. Refactoring the code often leads to the comments being superfluous.

Rails Presenter Pattern

Jay Fields created the concept of Rails presenters in 2006. He coined the term “Rails Presenter Pattern” in March 2007 and the concept has been evolving ever since.

What is a Presenter?

A presenter is an object that is responsible for view-related logic and behavior. Without a presenter it's common to spread view logic into templates, helper modules, controller actions and even into models themselves.

There can be a great deal of pain and difficulty implementing, testing and maintaining the required logic and behavior when it is spread throughout so many objects and layers. Moving this responsibility into a presenter provides a single object which can be tested in isolation and easily changed. It also promotes discoverability for view logic and behavior.

Presenters can be used to house view-related logic and behavior for individual models as well as for representing distinct sections or components on the rendered page.

A simple implementation of a presenter includes nothing more than a Plain Old Ruby Object (PORO) and some delegation:

An example of a simple presenter.

```
class PersonPresenter

  def initialize(person)
    @person = person
  end

  delegate :first_name, :last_name, :age, :to => :@person

  # custom method for the view
  def address
    "#{@person.street_address}, #{@person.city}, #{@person.state} #{@person.zip}".upcase
  end
end
```

Alternative Implementations

PresentationObject

PresentationObject is an alternative implementation of the Presenter pattern which has built-in method caching:

An example of a PresentationObject presenter.

```
class PersonPresenter < PresentationObject

  def initialize(person)
    @person = person
  end

end
```

```
delegate :first_name, :last_name, :age, :to => :@person

# custom method for the view
declare :address do
  "#{@person.street_address}, #{@person.city}, #{@person.state} #{@person.zip}".upcase
end

end
```

The difference between the PORO presenter and the PresentationObject is that the address string will be built only one time and then cached with the PresentationObject whereas it will be constructed each time the address method is called from the view with the PORO presenter.

The method caching performed by the PresentationObject is more elaborate than simply doing:

```
@result ||= do_some_thing
```

This caching doesn't handle nil and false values very well. For more information on the PresentationObject see the references section for the article "The Exceptional Presenter".

Cells, Erector

Cells and Erector are separate projects which combine the goal of the presenter as defined in this section along with view templates. They are interesting projects which might be worth checking out.

References

- The Exceptional Presenter – <http://spin.atomicobject.com/2008/01/27/the-exceptional-presenter/>
- PresentationObject SVN – http://atomicobjectrb.rubyforge.org/svn/trunk/presentation_object/
- Cells – <http://cells.rubyforge.org/>
- Erector – <http://erector.rubyforge.org/documentation.html>

Presenter related articles by Jay Fields in chronological order:

- <http://blog.jayfields.com/2006/09/rails-model-view-controller-presenter.html>
- <http://blog.jayfields.com/2007/01/another-rails-presenter-example.html>
- <http://blog.jayfields.com/2007/03/rails-presenter-pattern.html>
- <http://blog.jayfields.com/2007/09/railsconf-europe-07-presenter-links.html>
- <http://blog.jayfields.com/2007/10/rails-rise-fall-and-potential-rebirth.html>

Refactorings

In the following pages you'll find a list of common Rails refactorings. This list is not intended to be a complete list of the possible refactorings for Rails applications although it tries to cover the most common.

Each refactoring builds on top of refactorings set forth by Martin Fowler's book [Refactoring: Improving The Design Of Existing Code](#). In most cases the mechanics described for each of the Rails refactorings are not as granular as they could be. Instead they are intended to be pragmatic steps which require the reader to understand how-to do things like extract a class, move a method, etc.

Basic Building Blocks for Rails Refactorings

Extract Class

You have one class doing work that should be done by two or more. To remedy create a new class and move the relevant fields and methods from the old class into the new class. —Refactoring, Fowler, pg. 149.

Move Method

When a method is, or will be using or used by more features of another class than the class on which it is defined you should create a new method with a similar body in the class it uses most and turn the old method into a simple delegation, or remove it altogether. —Refactoring, Fowler, pg. 142.

Extract Method

You have code fragments that can be grouped together. Turn the fragment into a method whose name explains the purpose of this method. —Refactoring, Fowler, pg. 110.

Replace Instance Variable With Local Variable in Partial

You want to reuse a view partial, but it or a partial that it ends up rendering relies on instance variables.

Push the instance variable up into a higher level template and pass it in as a local.

```
# app/views/layouts/application.html.erb
<%= render :partial => "projects/sidebar" %>
```

```
# app/views/projects/_sidebar.html.erb
<%= h @project.name %>
```

to

```
<%= render :partial => "projects/sidebar", :locals => { :project => @project } %>
```

```
# app/views/projects/_sidebar.html.erb
<%= h project.name %>
```

Motivation

A partial is nothing more than a reusable view template. Partials allow you to modularize the components which make up a particular page into logical, cohesive pieces. When required data is not passed into a partial, it is often time consuming, tedious and difficult to reuse or change later.

By passing the required data in as locals you create self-documenting requirements for each view partial. In order to reuse a view partial it's a matter of looking to see where else it is rendered and what locals are required.

Mechanics

- Update the partial to rely on a local variable instead of an instance variable
- Find all templates which render the partial and update them to pass in the instance variable as a local
- Rinse and repeat the first two steps until you hit the top level template
- Run your tests

Example

I begin with an application layout rendering a project sidebar, which in turn renders a list of users on the project. At each level the `@project` instance variable is required.

```
# app/views/layouts/application.html.erb
<% if @project %>
  <%= render :partial => "projects/sidebar" %>
<% end %>
```

```
# app/views/projects/_sidebar.html.erb
<h2><%= h @project.name %></h2>
<%= render :partial => "projects/users" %>
```

```
# app/views/projects/_users.html.erb
```

```

<ul>
<% @project.users.each do |user| %>
  <li><%= user.name %></li>
<% end %>
</ul>

```

First, I update the projects/users partial to rely on a local rather than an instance variable.

```

# app/views/projects/_users.html.erb
<ul>
<% project.users.each do |user| %>
  <li><%= user.name %></li>
<% end %>
</ul>

```

Second, I update the projects/sidebar partial to pass in the project as a local when rendering the projects/users.

```

# app/views/projects/_sidebar.html.erb
<h2><%= h @project.name %></h2>
<%= render :partial => "projects/users", :locals => { :project => @project } %>

```

Next, I rinse and repeat until I'm at the top level template, which means I perform the same refactoring on the projects/sidebar partial.

```

# app/views/layouts/application.html.erb
<% if @project %>
  <%= render :partial => "projects/sidebar", :locals => { :project => @project } %>
<% end %>

```

```

# app/views/projects/_sidebar.html.erb
<h2><%= h project.name %></h2>
<%= render :partial => "projects/users", :locals => { :project => project } %>

```

```

# app/views/projects/_users.html.erb
<ul>
<% project.users.each do |user| %>
  <li><%= user.name %></li>
<% end %>
</ul>

```

And that is the refactoring!

Additional Comments

A web of partials rendering partials can make this refactoring time consuming and tedious to perform. After all some partials are rendered using an object rather than a string path.

```

<%= render :partial => object %>

```

This pain may be may be a sign that there is an issue in how data is being organized for the view. It may be time to Extract Presenter from View Data.

Extract Query to Model

You are performing a query directly from the view or a controller.

Extract the query and push it down into the appropriate model.

```
# app/views/people/index.html.erb
<% Person.find(
  :all,
  :conditions => [ "active = ?", true], :order => "name"
).each do |person| %>
  ...
<% end %>

to

# app/views/people/index.html.erb
<% @people.each do |person| %>
  ...
<% end %>

# app/controllers/people_controller.rb
def index
  @people = Person.active_people
end

# app/models/person.rb
def self.active_people
  find(:all, :conditions => [ "active = ?", true], :order => "name")
end
```

Motivation

The view should be presenting information, not determining what information should be presented and then presenting it. Clearly this violates any Separation of Concerns between the view and the model.

Let the model provide the information, and let the view display it.

When you hit a scenario where you need to massage the data returned from the model in order for it to be displayed appropriately you can use a presenter to encapsulate that logic. If the logic already exists in the view template you can Move Logic From View Into Presenter.

Mechanics

- Extract query into model
- Update controller to assign variable
- Update view to use variable
- Run the tests

Example

I start with a template that renders a list of people:

```
# app/views/people/index.html.erb
<ul>
  <% Person.find(
    :all,
    :conditions => [ "active = ?", true],
    :order => "name"
  ).each do |person| %>
    <li>
      <%= h person.name %>
    </li>
  <% end %>
</ul>
```

First, I extract the query out onto a method on the model:

```
# app/models/person.rb
def self.active_people
  find(:all, :conditions => [ "active = ?", true], :order => "name")
end
```

Second, I update the controller to assign the variable:

```
# app/controllers/people_controller.rb
def index
  @people = Person.active_people
end
```

Finally, I update the view to use the variable:

```
<ul>
  <% @people.each do |person| %>
    <li>
      <%= h person.name %>
    </li>
  <% end %>
</ul>
```

Additional Comments

Sometimes when extracting queries from the view you'll find that the required behavior is already being supplied by the model. It may be an exact implementation or it may be slightly different. Often times the method name can give it away.

This refactoring also applies to queries in helper modules and controllers. If you do need to make additional requests for data then look into using a presenter, but always keep the query itself down in the model.

References

None.

Extract Renderer From RJS

There is a flurry of RJS performing multiple conceptual operations on the page.

Extract each conceptual operation into a well-named method on a `Renderer`.

```
# app/views/events/create.js.rjs
page.replace_html :notice, "'#{@event.summary}' was successfully created."
page.hide :error
page.show :notice
page.visual_effect :appear, :notice
page.delay( 5 ) { page.visual_effect :fade, :notice }
```

to

```
# app/views/events/create.js.rjs
render_with :calendar do |calendar|
  calendar.display_notice @event
end
```

```
# app/renderers/calendar_renderer.rb
def display_notice(event)
  page.replace_html :notice, "'#{event.summary}' was successfully created."
  page.hide :error
  page.show :notice
  page.visual_effect :appear, :notice
  page.delay( 5 ) { page.visual_effect :fade, :notice }
end
```

Motivation

Remote JavaScript (RJS) can easily become difficult to read and easy to break. Part of this is because a single conceptual operation on the page is expressed using a number of RJS statements. Often the full update involves many operations.

Extracting the RJS out which makes up a single conceptual operation into a well-named method on a `Renderer` adds to readability, maintainability and testability of your RJS code.

The benefit of the `Renderer` extends beyond just having well-named methods. It is there to provide a well-named class as well. These things combined provide a clear context, a reusable set of RJS operations, and easier to read code.

Mechanics

- Extract the operation out as a method on a `Renderer`
- Push required variables up to parameters of the method
- Update the RJS file to use the `Renderer`
- Test
- Rinse and repeat for each conceptual operation

Example

This RJS updates a calendar after the user creates an event.

```
# app/views/events/create.js.rjs
page.replace_html :notice, "'#{@event.summary}' was successfully created."
page.hide :error
page.show :notice
page.visual_effect :appear, :notice
page.delay( 5 ) { page.visual_effect :fade, :notice }
page.insert_html(
  :bottom,
  dom_id(@event.calendar),
  :partial => 'events/mini_calendar',
  :locals => { :event => @event }
)
page.replace_html(
  dom_id(@event.calendar, 'stats'),
  :partial => 'calendars/stats',
  :locals => { :calendar => @event.calendar }
)
```

I've identified displaying a message to the user as a conceptual operation. Based on the RJS it is made up of the top five lines in the create.js.rjs file. Since this is in the context of a calendar, I create a CalendarRenderer class and move this operation to it as a single method:

```
# app/renderers/calendar_renderer.rb
class CalendarRenderer < Renderer
  def display_notice
    page.replace_html :notice, "'#{@event.summary}' was successfully created."
    page.hide :error
    page.show :notice
    page.visual_effect :appear, :notice
    page.delay( 5 ) { page.visual_effect :fade, :notice }
  end
end
```

After pulling the method out I see that I either need to pass in @event or the whole message itself. I decide to pass in the @event so I update the method to use a passed in event:

```
# app/renderers/calendar_renderer.rb
class CalendarRenderer < Renderer
  def display_notice(event)
    page.replace_html :notice, "'#{event.summary}' was successfully created."
    page.hide :error
    page.show :notice
    page.visual_effect :appear, :notice
    page.delay( 5 ) { page.visual_effect :fade, :notice }
  end
end
```

Now I update the create.js.rjs file to use the renderer:

```
# app/views/events/create.js.rjs
render_with :calendar do |calendar|
  calendar.display_notice(@event)
end
```

```

page.insert_html(
  :bottom,
  dom_id(@event.calendar),
  :partial => 'events/mini_calendar',
  :locals => { :event => @event }
)
page.replace_html(
  dom_id(@event.calendar, 'stats'),
  :partial => 'calendars/stats',
  :locals => { :calendar => @event.calendar }
)

```

At this point I manually test that displaying and fading out the notice message still works. If I had a browser-based integration test covering this functionality I would run that as well to ensure it still passed.

Next, I notice that there are two conceptual operations still in the RJS file: adding the event to the calendar, and updating the calendars statistics. I perform the same steps that I did for the displaying the notice and I end up with:

```

# app/views/events/create.js.rjs
render_with :calendar do |calendar|
  calendar.display_notice(@event)
  calendar.add_event(@event)
  calendar.update_stats(@event.calendar)
end

# app/renderers/calendar_renderer.rb
class CalendarRenderer < Renderer
  def display_notice(event)
    page.replace_html :notice, "'#{event.summary}' was successfully created."
    page.hide :error
    page.show :notice
    page.visual_effect :appear, :notice
    page.delay( 5 ) { page.visual_effect :fade, :notice }
  end

  def add_event(event)
    page.insert_html(
      :bottom,
      dom_id(event.calendar),
      :partial => 'events/mini_calendar',
      :locals => { :event => event }
    )
  end

  def update_stats(calendar)
    page.replace_html(
      dom_id(calendar, 'stats'),
      :partial => 'calendars/stats',
      :locals => { :calendar => calendar }
    )
  end
end

```

And that's the refactoring!

Additional Comments

Testing renderers may differ from how you test your typical RJS files. I don't like to test the generation of JavaScript code that the frameworks provides. I trust in the framework that `replace_html` is going to generate the right JavaScript to do the job in the browser. Because of this I avoid using things like `assert_select_rjs` which matches generated JavaScript against a set of regular expressions.

Instead, I am more interested in the fact that I am making the call to `replace_html`. Given this precursor renderers are intended to be tested using interaction based testing.

The `Renderer` class is apart of the `RenderWith` plugin. See below references.

References

- `Renderer / RenderWith` on GitHub: http://github.com/zdennis/render_with/tree/master

Move Data From View Into Presenter

A view template is relying on multiple variables to provide information for a particular UI component or section of the page.

Extract a presenter as the object which supplies the information to the view and move appropriate methods into it.

```
# app/controllers/events_controller.rb
def new
  @organizers = Organizer.find(:all).sort_by("name")
  @event = Event.new
end

# app/views/events/new.html.erb
<%= render :partial => "events/form",
          :locals => { :event => @event, :organizers => @organizers } %>

# app/views/events/_form.html.erb
<%= form.collection_select :organizer_id, organizers, :id, :name %>

to

# app/controllers/events_controller.rb
def new
  @event = EventPresenter.new :event => Event.new
end

# app/views/events/new.html.erb
<%= render :partial => "events/form", :locals=> { :event => @event } %>

# app/views/events/_form.html.erb
<%= form.collection_select :organizer_id, event.possible_organizers, :id, :name %>

# app/presenters/event_presenter.rb
def possible_organizers
  Organizer.find(:all).sort_by("name")
end
```

Motivation

View templates contain code which represent different UI components on the page. Having multiple objects supply information for the same component breaks the idea of encapsulation in OO thinking. For example you lose the benefits that come from grouping things together that go together when it comes time to make a change. It can also lead to Inappropriate Intimacy between the controller and the views.

The only template a controller needs to know about and supply data for is the one it renders. If needs to know intimate details of what is required for additional partials in the rendering process then it is becoming too intimate with the views. Likewise, partials which rely on instance variables are too intimate with the environment that they are rendered in.

If the instance variable is something that the controller needs to provide you can do Replace Instance Variable With Local Variable In Partial to get rid of the love affair between the partial and its environment.

If a UI component is using multiple objects to gather the information it needs in order to be displayed it may be a sign to extract a class that provides the information for the UI component. This is the heart of Move Data From View Into Presenter.

Move Data From View Into Presenter provides a single object responsible for the providing information to the UI component. This makes it easier to change the requirements of that UI component over its lifetime.

When a UI component relies on multiple objects and mechanisms it can make it more difficult to change because the information that is being provided has a greater likelihood of being in multiple spots.

This refactoring also encourages removing Inappropriate Intimacy between the controller and the views that are being rendered since it will help remove unneeded instance variable assignment in the controller.

Here's an analogy summarizing the motivation for this refactoring.

“Say you want to pay a bill. If you keep the checkbook in one place, pens in another, envelopes in another, stamps in another and the bills in another, how easy will it be to pay your bills?” *Credited to Craig Demyanovich*

Mechanics

- Update the view template to use a single object to gather information for the UI component in question
- Create a presenter class with the method in question
- Add any additional delegations to the presenter that the view requires
- Update the controller to use the presenter and remove the unneeded instance variable assignment
- Run your tests

Example

I begin with creating an event where the controller not only sets up a blank event for the view to use, but also queries for and sorts a list of possible organizers for the event to have:

```
# app/controllers/events_controller.rb
class EventsController < ApplicationController

  def new
    @organizers = Organizer.find(:all).sort_by("name")
    @event = Event.new
  end

end

# app/views/events/new.html.erb
<%= render :partial => "events/form",
          :locals => { :event => @event, :organizers => @organizers } %>

# app/views/events/_form.html.erb
<% form_for event do |form| %>
```

```

<%= form.text_field :name %>
<%= form.text_field :description %>
<%= form.text_field :location %>
<%= form.datetime_select :starts_at %>
<%= form.collection_select :organizer_id, organizers, :id, :name %>
<%= form.submit "Save" %>
<% end %>

```

When dealing with the presentation of an event I want the event object in the view to supply any information required for it to be displayed properly. I see that organizers is supplied through a passed in local variable.

The first thing I do is update the events/form partial to access the required organizers from the event itself:

```

# app/views/events/_form.html.erb
<% form_for event do |form| %>
  <%= form.text_field :name %>
  <%= form.text_field :description %>
  <%= form.text_field :location %>
  <%= form.datetime_select :starts_at %>
  <%= form.collection_select :organizer_id, event.possible_organizers, :id, :name %>
  <%= form.submit "Save" %>
<% end %>

```

The second step is to create an EventPresenter class which will be used to house view oriented logic and behavior for the event. I start by adding the possible_organizers method that I want.

```

class EventPresenter < PresentationObject
  def initialize(options)
    @event = options[:event]
  end

  declare :possible_organizers do
    Organizer.find(:all).sort_by("name")
  end
end

```

The third step is to add any delegations that are required by the view. Since the presenter is just a simple decorator for the event I need to forward any calls that I don't want the presenter to answer on to the event itself.

```

class EventPresenter
  def initialize(options)
    @event = options[:event]
  end

  delegate :class, :errors, :id, :new_record?, :to_param,
           :description, :location, :name, :organizer_id, :start_at,
           :to => :@event

  def possible_organizers
    Organizer.find(:all).sort_by("name")
  end
end

```

Lastly, I remove the unneeded query and assignment in the controller as well wrap the event in an EventPresenter.

```
# app/controllers/events_controller.rb
class EventsController < ApplicationController

  def new
    @event = EventPresenter.new :event => Event.new
  end

end
```

And that is the refactoring!

Additional Comments

Naming Conventions

When supplying a set of values that a user can choose from on a form I prefer prefixing the method with “possible_”. It lets me know that this is not a value of the event itself, but a set of possible values for the event.

Determining The Delegations

Sometimes information required in the view is tucked away inside of other partials or hidden in helper methods or HTML/HAML/etc code. Having and running an automated test suite helps with making sure you find all of the necessary delegations. When in doubt running the application first hand will help you quickly pinpoint any other delegations that are missing from your presenter.

Alternative Approaches

This example used the refactoring on a single action requiring `@organizers` to be set. It is quite common for multiple actions and controllers to require the same information. In this case a popular approach to solving the duplication is to extract the assignment into a helper method on the controller and then call that method as a before filter.

This approach works well when the assignment being done needs to happen at the controller level. I like to look to see if the assignment is required or if the controller is simply being too intimate with the views that it renders. If I find it is being too intimate I apply this refactoring. I find it gives me 3 benefits:

- The data is now tied to the presentation of a particular object, which gives it meaning. Having `@organizers` available doesn't tell me much about what they are used for. Having `event.possible_organizers` does.
- Storing presentation logic in a presenter allows me to make the information needed available in a format which is appropriate for the view. For example, a presenter can format a collection of organizers into the right format for a view helper. You could accomplish this in the controller, but that would lead us to Inappropriate Intimacy.
- Over time when the presentation of a particular object changes I can go to the presenter. If I didn't have the presenter I may spend time searching controller actions and before filters to find where information is being made available.

References

See the Presenter section.

Move Logic From View Into Presenter

You find a block of code in a view template which is computing a set of values or conditions used in the view.

Extract a presenter as the object which supplies the information to the view template and move the block of code into a method on it.

```
# app/views/people/_form.html.erb
<b>Group</b>
<%=
  group_hash = person.groups.group_by(&:type)
  groups = group_hash.keys.inject([]) do |arr,key|
    values = group_hash[key]
    arr << OpenStruct.new(:name => values.first.type, :group => values)
  end
  options = option_groups_from_collection_for_select(
    groups, :group, :name, :id, :name, person.group_id
  )
  select_tag "person[group_id]", options
%>
```

to

```
# app/views/people/_form.html.erb
<b>Group</b>
<%=
  options = option_groups_from_collection_for_select
    person.possible_groups, :group, :name, :id, :name, person.group_id
  )
  select_tag "person[group_id]", options
%>
```

```
# app/presenters/person_presenter.rb
def possible_groups
  group_hash = person.groups.group_by(&:type)
  groups = group_hash.keys.inject([]) do |arr,key|
    values = group_hash[key]
    arr << OpenStruct.new(:name => values.first.type, :group => values)
  end
end
```

Motivation

It's easy to add additional logic in a view template. I know I've done it by prototyping how I wanted something to work and then forgetting about it. Other times it crept up on me in small doses until it became a giant sore spot every time I opened up a particular template.

Logic in a view template is difficult to test so it's easy to avoid testing it—thus making it harder to change. Even when you do test the logic it makes for a more complicated test. This in turn increases the difficulty level for maintaining the test and implementation. Often it would have been simpler to extract out a simple object in the first place.

Pulling this logic out of the view into an object responsible for handling presentation logic removes unneeded responsibility and confusion from the view.

Mechanics

- Replace the logic in the view with a method call to the object responsible for the UI component
- Move the code block into a method on the presenter for the UI component
- Find where the variable is being set and update if necessary to ensure it is wrapped in the presenter
- Run your tests
- Rinse and repeat

Example

This example gives the user the ability to assign a group to a person in the system. There are several group options available for a person to be assigned so I am grouping the available options by their type and presenting that as a select dropdown to the user.

I start with the following template:

```
# app/views/people/_form.html.erb
<%= form_for person do |form| %>
  <p><b>Name</b> <%= form.text_field :name %></p>
  <p><b>Group</b>
    <%=
      group_hash = person.groups.group_by(&:type)
      groups = group_hash.keys.inject([]) do |arr,key|
        values = group_hash[key]
        arr << OpenStruct.new(:name => values.first.type, :group => values)
      end
      options = option_groups_from_collection_for_select(
        groups, :group, :name, :id, :name, person.group_id
      )
      select_tag "person[group_id]", options
    %>
  </p>
<%= end %>
```

The focus of the view is on the person so I decide that is what I want to supply the possible groups. I first update the view to use the interface desired:

```
# app/views/people/_form.html.erb
<%= form_for person do |form| %>
  <p><b>Name</b> <%= form.text_field :name %></p>
  <p><b>Group</b>
    <%=
      options = option_groups_from_collection_for_select(
        person.possible_groups, :group, :name, :id, :name, person.group_id
      )
      select_tag "person[group_id]", options
    %>
  </p>
<%= end %>
```

Next, I move the extracted code into the PersonPresenter. If one doesn't exist I create it.

```
# app/presenters/person_presenter.rb
def possible_groups
  group_hash = person.groups.group_by(&:type)
  groups = group_hash.keys.inject([]) do |arr,key|
```

```
      values = group_hash[key]
      arr << OpenStruct.new(:name => values.first.type, :group => values)
    end
  end
end
```

Lastly, I find what templates were rendering the people/form partial and I follow the render chain up until I find where the person gets assigned. If the person isn't being wrapped in a PersonPresenter I update the assignment so that it is. I rinse and repeat this last step for each unique render path.

And that is the refactoring!

Additional Comments

Alternative Approaches

An alternative to this approach would be to extract the whole code block out into a helper module. I don't like this because there are two distinct responsibilities in the block of code:

- The first responsibility is determining what groups are available for a person and what to group them on.
- The second responsibility is generating the option tags from the available groups.

The first responsibility is tied to the presentation of a Person. If I move the code into a helper module I reduce the noise in my template and I make the template easier to test. While this is better than leaving it the way it was we have a chance to do one better. A presenter gives us the opportunity to add any additional view data or logic in one place. This provides better readability and maintainability than ending up with helper methods which may be spread out amongst multiple files and inter-mixed with methods that have nothing to do the view component in question.

The second responsibility is tied to the requirements of the template medium to display the list. We need to generate the appropriate HTML tags. This has nothing to do with providing information for the person/form partial so it doesn't belong with the code that determines what groups are available in the PersonPresenter.

References

See the Presenter section.

Extract Complex Creation Into Factory

Creation of a model becomes complicated or reveals too much of its internal structure.

Extract the creation of the model into a Factory responsible for handling that logic.

```
# app/controllers/events_controller.rb
def create
  @event = Event.new(params[:event].merge(:creator => current_user))
  @event.initialize_custom_attributes
  @event.custom_attributes.each do |attr|
    attr.value = params[:custom_attributes].fetch(attr.form_field.name, nil)
  end
  @event.save
  ...
end

to

# app/controllers/events_controller.rb
def create
  EventFactory.create(params[:event], params[:custom_attributes], current_user)
  ...
end

# app/factory/event_factory.rb
class EventFactory

  def self.create(event_attrs, custom_attrs, creator)
    event = Event.new(event_attrs.merge(:creator => creator))
    event.initialize_custom_attributes
    event.custom_attributes.each do |attr|
      attr.value = custom_attrs.fetch(attr.form_field.name, nil)
    end
    event.save
  end
end

end
```

Motivation

Creation of a model can be a major operation in itself. A common approach is to make the controller responsible for handling these additional duties to ensure the model is built with the right attributes and collaborators. Doing this gives the controller another responsibility and smells of Inappropriate Intimacy with the model.

It also makes it more difficult to test and maintain because we aren't isolating the important operations which make up creating an instance of the model. Instead it mixes the process of complex creation with everything else that the controller action is doing.

Pulling out complex creation logic into a Factory allows us to explicitly capture the process of creating our model.

This refactoring is built upon the Factory concept as explained in Eric Evans' book [Domain Driven Design](#). *It fits the criteria for the Builder pattern as described in Design Patterns.*

Mechanics

- Extract creation logic into Factory
- Update controller to use Factory
- Run the tests

Example

This action exemplifies complex creation logic in a controller:

```
# app/controllers/events_controller.rb
def create
  @event = Event.new(params[:event].merge(:creator => current_user))
  @event.initialize_custom_attributes
  @event.custom_attributes.each do |attr|
    attr.value = params[:custom_attributes].fetch(attr.form_field.name, nil)
  end
  if @event.save
    redirect_to event_path(@event)
  else
    render :action => "edit"
  end
end
```

First I'm going to pull the creation logic for the event out into an EventFactory:

```
# app/factories/event_factory.rb
class EventFactory

  def self.create(event_attrs, custom_attrs, creator)
    event = Event.new(event_attrs.merge(:creator => creator))
    event.initialize_custom_attributes
    event.custom_attributes.each do |attr|
      attr.value = custom_attrs.fetch(attr.form_field.name, nil)
    end
    event.save
    event
  end
end
```

Next I update the controller to use the Factory:

```
# app/controllers/events_controller.rb
def create
  @event = EventFactory.create(params[:event], params[:custom_attributes], current_user)
  unless @event.new_record?
    redirect_to event_path(@event)
  else
    render :action => "edit"
  end
end
```

And that's the refactoring!

Additional Comments

In some cases the creation logic is more difficult to isolate because of the use of before filters, instance variable assignment and view logic in the controller which mixes the requirements of creating the model with other things the controller is assuming responsibility for.

When you separate out the complex creation of a model you explicitly state what is required to have a valid instance of that model. This makes your controller less complex as well as more readable and more testable.

Applying this refactoring reinforces the Single Responsibility Principle because you are removing a secondary responsibility from the controller and moving it onto a Factory whose sole purpose is that object construction.

References

- Domain Driven Design by Eric Evans
- Design Patterns by the Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
- Single Responsibility Principle – http://en.wikipedia.org/wiki/Single_responsibility_principle
- Uncle Bob's Principles of OOD – <http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>
- Uncle Bob – Single Responsibility Principle – <http://www.objectmentor.com/resources/articles/srp.pdf>

Introduce Result Object

The return value of a method needs to supply more information to the caller.

Introduce a result object which provides that information.

```
# app/controllers/events_controller.rb
def create
  if EventFactory.create(params[:event], params[:custom_attributes], current_user)
    # I need an event
    redirect_to event_path(...)
  else
    # I need an @event
    render :action => "edit"
  end
end

to

# app/controllers/events_controller.rb
def create
  event_creation = EventFactory.create(params[:event], params[:custom_attributes], current_user)
  @event = event_creation.event
  if event_creation.success?
    redirect_to event_path(@event)
  else
    render :action => "edit"
  end
end

# app/factory/event_factory.rb
class EventFactory

  def self.create(event_attrs, custom_attrs, creator)
    ...
    event.save!
    OpenStruct.new(:success? => true, :event => event)
  rescue ActiveRecord::ActiveRecordError
    OpenStruct.new(:success? => false, :event => event)
  end
end

end
```

Motivation

It is common for a controller action to assign an instance variable for use in the view. This can make it difficult to move misplaced responsibility out of the controller and into a Service/Factory because the controller action needs more information—returning true or false just isn't enough.

One way to get around this is to leave the initial building or finding of a resource in your controller. This allows you to assign the resource to an instance variable and then pass it in as a parameter to the method on the Service/Factory. Although this works there is another route which allows you to keep all the misplaced responsibility together without having to leave troops behind. This route is using a result object.

A result object provides the ability to return additional information about what just happened without introducing unneeded coupling between your Service/Factory and the controller. It also maintains encapsulation of your Service/Factory object.

This refactoring can be done in addition to Extract Complex Creation Into Factory and also Extract Operations Into Service.

Mechanics

- Update the caller to use the result object
- Update the method under question to return a result object
- Run tests

Example

In this example, the action creates a new Event:

```
# app/controllers/events_controller.rb
def create
  @event = Event.new(params[:event].merge(:creator => current_user))
  @event.initialize_custom_attributes
  @event.custom_attributes.each do |attr|
    attr.value = params[:custom_attributes].fetch(attr.form_field.name, nil)
  end
  if @event.save
    redirect_to event_path(@event)
  else
    render :action => "edit"
  end
end
```

After performing Extract Complex Creation Into Factory I end up with:

```
# app/controllers/events_controller.rb
def create
  @event = EventFactory.create(params[:event], params[:custom_attributes], current_user)
  unless @event.new_record?
    redirect_to event_path(@event)
  else
    render :action => "edit"
  end
end

# app/factories/event_factory.rb
class EventFactory

  def self.create(event_attrs, custom_attrs, creator)
    event = Event.new(event_attrs.merge(:creator => creator))
    event.initialize_custom_attributes
    event.custom_attributes.each do |attr|
      attr.value = custom_attrs.fetch(attr.form_field.name, nil)
    end
    event.save
    event
  end
end
```

I could stop here, but I won't because `@event.new_record?` isn't conveying the message that I want the code to send. The controller doesn't care if the `@event` is a new record. It cares if event creation was successful. Having `EventFactory.create` return a true/false value wouldn't get me very far because I need to supply an `@event` for the view. Here's where a result object comes into play.

First, I update the controller to expect a result object:

```
# app/controllers/events_controller.rb
def create
  event_creation = EventFactory.create(params[:event], params[:custom_attributes], current_user)
  @event = event_creation.event
  if event_creation.success?
    redirect_to event_path(@event)
  else
    render :action => "edit"
  end
end
```

Lastly, I update `EventFactory.create` to return the result object. I use an `OpenStruct` to do the job:

```
# app/factories/event_factory.rb
class EventFactory

  def self.create(event_attrs, custom_attrs, creator)
    event = Event.new(event_attrs.merge(:creator => creator))
    event.initialize_custom_attributes
    event.custom_attributes.each do |attr|
      attr.value = custom_attrs.fetch(attr.form_field.name, nil)
    end
    event.save!
    OpenStruct.new(:success? => true, :event => event)
  rescue ActiveRecord::ActiveRecordError
    OpenStruct.new(:success? => false, :event => event)
  end
end
```

And that's the refactoring!

Additional Comments

Alternative Approach

An alternative approach would be to use `Multiblock`—a result object library which combines the concept of a result object with the power of blocks. Below is what our above example would have looked like if I had used `Multiblock`:

```
# app/controllers/events_controller.rb
def create
  EventFactory.create(params[:event], params[:custom_attributes], current_user)
  do |event_creation|
    event_creation.success do |event|
      redirect_to event_path(event)
    end

    event_creation.failure do |event|
      @event = event
    end
  end
end
```

```

        render :action => "edit"
      end
    end
  end
end

# app/factories/event_factory.rb
class EventFactory

  def self.create(event_attrs, custom_attrs, creator)
    event = Event.new(event_attrs.merge(:creator => creator))
    event.initialize_custom_attributes
    event.custom_attributes.each do |attr|
      attr.value = custom_attrs.fetch(attr.form_field.name, nil)
    end
    event.save!
    yield Multiblock[:success, event]
  rescue ActiveRecord::ActiveRecordError
    yield Multiblock[:failure, event]
  end
end
end

```

The Multiblock approach relies on block-style argument passing and a `respond_to` like interface while the OpenStruct approach doesn't use blocks and requires that you ask the result object directly for any information you may need.

The Multiblock approach has the advantage that it works great when you have multiple possible scenarios that could occur. These scenarios also don't have to be limited to success/failure. They can be any symbol which helps convey the intent of your code better.

References

- Multiblock by Drew Colthorp – <http://github.com/dcolthorp/multiblock/tree/master>

Extract Inline Update to RJS File

A controller action takes on the responsibility of a view template and renders inline RJS updates.

Extract the inline update into an RJS file.

```
# app/controllers/events_controller.rb
def create
  ...
  render :update do |page|
    page.replace_html :notice, "'#{@event.summary}' was successfully created."
    page.hide :error
    page.show :notice
    page.visual_effect :appear, :notice
    page.delay( 5 ) { page.visual_effect :fade, :notice }
  end
end

to

# app/controllers/events_controller.rb
def create
  ...
end

# app/views/events/create.js.rjs
page.replace_html :notice, "'#{@event.summary}' was successfully created."
page.hide :error
page.show :notice
page.visual_effect :appear, :notice
page.delay( 5 ) { page.visual_effect :fade, :notice }
```

Motivation

Sometimes performing an inline render update inside of a controller action is simple enough that it doesn't clutter your action or your tests. In other cases the inline render update would serve the application better if it was extracted into its own RJS file.

Moving the RJS statements into their own file isolate them from the controller action in a way that clears up what the action is doing and what the RJS is doing. Once you have a separate RJS file you may be able to do Extract Render From RJS to further clarify and provide explicit intent of the RJS.

Mechanics

- Extract inline update into RJS file
- Remove inline update from controller action
- Run tests

Example

I'm going to use an example for creating an event:

```
# app/controllers/events_controller.rb
```

Extract Inline Update to RJS File

```

def create
  @event = Event.new(params[:event])
  respond_to do |format|
    if @event.save
      format.html
      format.js do
        render :update do |page|
          page.replace_html :notice, "'#{@event.summary}' was successfully created."
          page.hide :error
          page.show :notice
          page.visual_effect :appear, :notice
          page.delay( 5 ) { page.visual_effect :fade, :notice }
        end
      end
    else
      ...
    end
  end
end
end

```

I pull the update in the case where the event saves successfully into a create.js.rjs file:

```

# app/views/events/create.js.rjs
page.replace_html :notice, "'#{@event.summary}' was successfully created."
page.hide :error
page.show :notice
page.visual_effect :appear, :notice
page.delay( 5 ) { page.visual_effect :fade, :notice }

```

Next I remove the unneeded render update in the controller:

```

# app/controllers/events_controller.rb
def create
  @event = Event.new(params[:event])
  respond_to do |format|
    if @event.save
      format.html
      format.js
    else
      ...
    end
  end
end
end

```

And that's the refactoring.

Additional Comments

Separating view logic from the controller promotes single responsibility between the controller and the view template, which happens to be an RJS file in this case. This helps make each of these easier to read and maintain over the lifetime of the application.

Alternative Approach

Based on this example an alternative approach would be to create a helper method in the controller which wrapped the entire render update. This would have provided the same benefit to the create action by removing the inline update, but it would have continued to leak detailed knowledge into

the controller about what was to be rendered.

While this can be a convenience, it spreads the response body across view and controller boundaries which decreases discoverability of the response and can lead to a more difficult to read and maintain controller.

References

None.

Extract Operations Into Service

The controller is performing an operation that does not belong to it or the model.

Extract the operation out into its own service.

```
# app/controllers/projects_controller.rb
def update
  @project = ProjectPermission.find_project_for_user(params[:id], current_user)
  if @project && @project.update_attributes(params[:project])
    @project.update_members params[:users]
    @project.members.each do |member|
      ProjectNotification.deliver_project_notification(@project, member)
    end
  end
  ...
end
end

to

def update
  @project = ProjectService.find_project_for_user(params[:id], current_user)
  if ProjectService.update_project(@project, params[:project], params[:users] current_user)
    ...
  end
end
end
```

Motivation

This refactoring is built upon the Service concept as defined in Eric Evans' book Domain Driven Design. According to Evans, "there are important operations that occur which can't find a natural home." In a Rails application these tend to show up inside of a controller. When you find a set of activities that go together to form a single conceptual operation, it is important to extract it onto an object that can be responsible for that higher level operation. This object is a Service.

Services allow you to remove complex domain manipulation code from your controllers [1], leaving them to interpret the request and hand it off to the appropriate application or domain object to do the work. These activities are not limited to controllers either, they can also show up in the model.

Mechanics

- Extract operation into a Service object
- Update controller to use Service object
- Run the tests

Example

In this example I start with updating a project:

```
# app/controllers/projects_controller.rb
def update
  @project = ProjectPermission.find_project_for_user(params[:id], current_user)
```

```

if @project && @project.update_attributes(params[:project])
  @project.update_members params[:users]
  @project.members.each do |member|
    ProjectNotification.deliver_project_notification(project, member)
  end
  flash[:notice] = 'Project was successfully updated.'
  redirect_to project_path(@project)
else
  render :action => "edit"
end
end
end

```

The higher level operation is updating the project. Although the act of updating the project belongs to the project model we are extending what it means to update a project in our application by performing two additional tasks:

- update the project's members only after it has been successfully updated
- deliver project notifications to its members

First I extract the operation onto a ProjectService object:

```

# app/services/project_service.rb
class ProjectService
  def update_project(project, project_params, users)
    if project && project.update_attributes(project_params)
      project.update_members users
      project.members.each do |member|
        ProjectNotification.deliver_project_notification(project, member)
      end
      true
    end
  end
end
end

```

Next, I update the controller to use the ProjectService for updating the project:

```

# app/controllers/projects_controller.rb
def update
  @project = ProjectPermission.find_project_for_user(params[:id], current_user)
  if ProjectService.update_project(@project, params[:project], params[:users])
    flash[:notice] = 'Project was successfully updated.'
    redirect_to project_path(@project)
  else
    render :action => "edit"
  end
end
end

```

Now that I have a ProjectService it makes sense to transfer the responsibility of accessing a project based on permissions over to it. This way I encapsulate how projects and permissions are implemented as they pertain to accessing a project. If this changes I won't have to update any client references I can just update the ProjectService:

I add find_project_for_user to the ProjectService class:

```

# app/services/project_service.rb
class ProjectService
  def find_project_for_user(project_id, user)
    ProjectPermission.find_project_for_user(project_id, current_user)
  end
end

```

```
end

...
end
```

And then I update the ProjectsController to use it:

```
# app/controllers/projects_controller.rb
def update
  @project = ProjectService.find_project_for_user(params[:id], current_user)
  if ProjectService.update_project(@project, params[:project], params[:users])
    flash[:notice] = 'Project was successfully updated.'
    redirect_to project_path(@project)
  else
    render :action => "edit"
  end
end
```

Now I have a single ProjectService who is responsible for performing extended operations on a project.

And that's the refactoring!

Additional Comments

When the operation that occurs can be handled entirely by the model there is no point in introducing a Service. This would have happened if the project was able to handle the full update. For example it would have looked like:

```
# app/controllers/projects_controller.rb
def update
  @project = ProjectPermission.find_project_for_user(params[:id], current_user)
  if @project.update_attributes(params[:project])
    flash[:notice] = 'Project was successfully updated.'
    redirect_to project_path(@project)
  else
    render :action => "edit"
  end
end
```

According to our application the act of updating a project involved more than just updating the internal state of the project. By identifying this new definition for updating a project from the application's point of view we are able to extract it onto the ProjectService and test it in isolation thus allowing our controller to stop trying to do it all.

When controller actions have to do multiple things in order to satisfy a single conceptual operation it is usually a smell that it should be pulled out. Another common indicator is the pain associated with writing controller tests which involves multiple conditions and multiple steps to satisfy a single conceptual operation.

Sometimes doing just one additional step in the controller doesn't feel worth it to pull it out. You'll have to be the judge of that for your code, but now you have a way to deal with it if it does get out of hand.

Service Naming

Evans points out that “sometimes services masquerade as model objects... These ‘doers’ end up with names ending in ‘Manager’ and the like.” Service classes don’t have to be named “Service”. In fact patterns should be used to describe the implementation, not necessarily impose a naming convention. It just so happens that patterns often fit the name criteria of the class in question.

I commonly use the suffix “Manager” to name an object which follows the Service pattern when it seems to fit better based on what it is doing.

SimpleServices

SimpleServices is a plugin by Justin Gehtland and Relevance LLC. It gives you high level declarative syntax to inject a service into a controller as well as wraps your service calls in database transactions.

To borrow an example from the article that announced SimpleServices:

```
class AccountController < ApplicationController
  services :account, :user, :security

  def update
    source_account = Account.find(params[:source_id])
    target_account = Account.find(params[:target_id])
    account_service.transfer(source_account, target_account, params[:amount])
  end
end
```

References

- 1 – Announcing SimpleServices by Justin Gehtland – <http://blog.thinkrelevance.com/2008/1/18/announcing-simpleservices>
- Domain Driven Design – Tackling Complexity in the Heart Of Software ; Eric Evans

Move Before Filter and Friends to Application Service

A before filter is doing more work than a before filter should.

Move the before filter into a method on an application service.

```
# app/controllers/projects_controller.rb
before_filter :find_project, :except => :index
before_filter :check_adminship, :only => [:update, :destroy]

def update
  if @project.update_attributes(params[:project])
    ...
  end
end

def destroy
  @project.destroy
end

private

def find_project
  @project = ProjectPermission.find_project_for_user(params[:id], current_user)
end

def check_adminship
  unless @project.admin?(current_user)
    if logged_in?
      flash[:error] = "You don't have access to that."
      begin redirect_to(:back) rescue redirect_to(home_path) end
      return false
    else
      redirect_to :back
    end
  end
end

to

# app/controllers/projects_controller.rb

rescue_from AccessDenied do |exception|
  if logged_in?
    flash[:error] = "You don't have access to that."
    begin redirect_to(:back) rescue redirect_to(home_path) end
    return false
  else
    redirect_to :back
  end
end

before_filter :find_project, :except => :index

def update
  if ProjectService.update_project(@project, current_user)
    ...
  end
end
```

```
def destroy
  ProjectService.destroy_project(@project, current_user)
end

private

def find_project
  @project = Project.find(params[:id])
end
```

Motivation

Before filters are a great mechanism to intercept and possibly interrupt the current request before the requested action is executed. They are also a good tool for ensuring the right information is available which may not be unique to the action, but rather a requirement of several actions or even the entire application.

A common approach to alleviating duplicate code in or across controllers is to move them into a before filter. Sometimes this is exactly what you need; finding the requested resource, time zone settings, logging in from cookies, and sub-domain parsing to name a few.

Other times it can just be a convenience, such as checking resource permissions. While checking resource permissions is not outside the scope of a controller there are times where it may reflect the need for another object, new or existing.

In some cases it is possible to eliminate unneeded before filters by moving the misplaced responsibility. Other times you can't eliminate the entire before filter (after all it may need to perform setup for the current request), but you are able to move out misplaced responsibility.

Four advantages comes to mind for moving misplaced responsibility out of the hands of a before filter:

- Easier testing. You no longer have to test the same logic over and over for every action which relies on the before filter. Instead you have isolated the logic in its own object which can be tested in isolation.
- Easier maintenance. Since the logic is isolated it is easier to make changes to that behavior.
- Greater reuse. The logic that was in the before filter is no longer only available to a web-based request.
- Greater clarity of intent. You've been able to extract one or more activities out which make up a higher level operation and your code is now explicit about what that operation is.

A common smell which indicates the responsibility is misplaced is when you are using before filters to act as a part of a larger operation associated with an action rather than as a before filter. Before filters which all essentially check for different types of permission on a particular resource come to mind.

Since before filters are commonly applied to multiple actions or controllers be sure to pick one action at a time to refactor. Trying to do everything at once may leave you in frustration and turmoil.

Mechanics

- Remove action from requiring the before filter
- Duplicate the before filter in the action
- Run tests
- Extract the operation into Service
- Update controller action to use the Service
- Rinse and repeat for each action which relied on the filter.
- Delete before filter
- Run tests

Example

I'm going to use the example of updating and destroying a project which requires the current user to be an admin of the project:

```
# app/controllers/projects_controller.rb
before_filter :find_project, :except => :index
before_filter :check_adminship, :only => [:update, :destroy]

def update
  if @project.update_attributes(params[:project])
    flash[:notice] = "The project has been successfully updated."
    redirect_to project_path(@project)
  else
    render :action => "edit"
  end
end

def destroy
  @project.destroy
end

private

def find_project
  @project = ProjectPermission.find_project_for_user(params[:id], current_user)
end

def check_adminship
  unless @project.admin?(current_user)
    if logged_in?
      flash[:error] = "You don't have access to that."
      begin redirect_to(:back) rescue redirect_to(home_path) end
      return false
    else
      redirect_to :back
    end
  end
end
```

The code which performs the check to see if the current user is an admin on the project smells a little. By itself it wouldn't give us any benefit to extract this out. I see that it is being used by the update and destroy actions. I'll start by looking at the update action.

I find that the act of updating a project is really a three part process: find the project, ensure the user

is an admin, then update the project. Right now these three steps are in three separate methods. I think I've identified the misplaced responsibility as the act of updating a project. I'd like to extract out this operation to a ProjectService application service so the things involved with updating the project are grouped together.

First, I remove the update action from the check_adminship before filter list:

```
# app/controllers/projects_controller.rb
...

before_filter :check_adminship, :only => [:destroy]

...
```

Next I duplicate the code from the check_adminship before filter into the update action itself:

```
# app/controllers/projects_controller.rb
...

def update
  unless @project.admin?(current_user)
    if logged_in?
      flash[:error] = "You don't have access to that."
      begin redirect_to(:back) rescue redirect_to(home_path) end
      return false
    else
      redirect_to :back
    end
  else
    if @project.update_attributes(params[:project])
      flash[:notice] = "The project has been successfully updated."
      redirect_to project_path(@project)
    else
      render :action => "edit"
    end
  end
end

...
```

At this point all of the tests still pass because I haven't changed the behavior of the controller, I've just reorganized it. Duplicating the before filter responsibilities with a single before filter could probably be skipped in favor the next step, but when extracting code from multiple before filters it helps to make sure you know about all of the code that you may need to extract. This helps make sure you're extracting the right things. The same is true for controller helper methods.

Next, I extract the act of updating a project into the ProjectService:

```
# app/services/project_service.rb
class ProjectService

  def update_project(project, params, user)
    if project.admin?(user)
      project.update_attributes(params)
    end
  end
end

end
```

At this point I need a way to handle telling the user that they don't have access to the resource. Rails 2.0.2 added the `rescue_from` method to handle exceptions at the controller level. I create an `AccessDenied` class and update the `ProjectService` to raise an `AccessDenied` exception if the user isn't a project admin:

```
# app/exceptions/access_denied.rb
class AccessDenied < StandardError ; end

# app/services/project_service.rb
class ProjectService

  def update_project(project, params, user)
    if project.admin?(user)
      project.update_attributes(params)
    else
      raise AccessDenied
    end
  end
end

end
```

I now add the `rescue_from` clause to the controller:

```
# app/controllers/projects_controller.rb
...

rescue_from AccessDenied do |exception|
  if logged_in?
    flash[:error] = "You don't have access to that."
    begin redirect_to(:back) rescue redirect_to(home_path) end
    return false
  else
    redirect_to :back
  end
end

...
```

The last step for updating the update action is to have it use the `ProjectService`:

```
# app/controllers/projects_controller.rb
...

def update
  if ProjectService.update_project(@project, params[:project], current_user)
    flash[:notice] = "The project has been successfully updated."
    redirect_to project_path(@project)
  else
    render :action => "edit"
  end
end

end
```

I run all tests to ensure that the update action isn't broken and I turn my attention to the next action which relies on `check_adminship`, the destroy action.

Since there is only a single before filter, and this is the last action that relies on it I skip duplicating the before filter in the action. Instead, I start by extracting out the act of destroying the project to the `ProjectService`:

```
# app/services/project_service.rb
class ProjectService
  ...

  def destroy_project(project, user)
    if project.admin?(user)
      project.destroy
    else
      raise AccessDenied
    end
  end
end

end
```

Next I update the destroy action to use the ProjectService:

```
# app/controllers/projects_controller.rb
...

def destroy
  ProjectService.destroy(@project, current_user)
  flash[:notice] = "The project has been deleted."
  redirect_to projects_path
end

...
```

Now it is safe to remove the `check_adminship` before filter since no more actions rely on it. My resulting controller code looks like:

```
# app/controllers/projects_controller.rb
before_filter :find_project, :except => :index

rescue_from AccessDenied do |exception|
  if logged_in?
    flash[:error] = "You don't have access to that."
    begin redirect_to(:back) rescue redirect_to(home_path) end
    return false
  else
    redirect_to :back
  end
end

def update
  if ProjectService.update_project(@project, params[:project], current_user)
    flash[:notice] = "The project has been successfully updated."
    redirect_to project_path(@project)
  else
    render :action => "edit"
  end
end

def destroy
  ProjectService.destroy(@project, current_user)
  flash[:notice] = "The project has been deleted."
  redirect_to projects_path
end

private

def find_project
```

```

    @project = ProjectPermission.find_project_for_user(params[:id], current_user)
  end

# app/services/project_service.rb
class ProjectService

  def update_project(project, params, user)
    if project.admin?(user)
      project.update_attributes(params)
    else
      raise AccessDenied
    end
  end

  def destroy_project(project, user)
    if project.admin?(user)
      project.destroy
    else
      raise AccessDenied
    end
  end
end
end

```

And that is the refactoring!

Additional Comments

The `rescue_from` at this point is generic enough to be pulled up into the `ApplicationController`. It can now be reused for any kind of access denied regardless of the resource or how access is determined.

This refactoring can be time consuming if you have a number of actions which rely on several before filters, since the before filters can separate activities comprising the higher level operation you're pulling out. This makes it more difficult to find the pieces which should be extracted. Following the steps outlined above can help identify and move the individual activities out into a single explicit operation.

Before Filters Here, There And Everywhere

Before filters are commonly pushed up to their parent class when they can be reused in other controllers. When an action or set of actions requires before filters to get pushed up it can make it difficult to understand what is actually going on.

Try to limit before filters to doing things that before filters need to do. If you recognize something as a higher level operation, then make it so. This will make your code more explicit, easier to read and easier to change over the lifetime of the application.

References

None.

Move View Data From Controller Into Presenter

The controller is performing additional duties for the view.

Move misplaced behavior into a presenter.

```
# app/controllers/events_controller.rb
before_filter :set_organizers_and_locations, :only => [:new, :edit]

def new
  @event = Event.new
end

private

def set_organizers_and_locations
  @organizers = Organizer.find(:all).map{ |o| [o.name, o.id] }.unshift []
  @locations = Location.find(:all).map{ |l| [l.name, l.id] }.unshift []
end

to

# app/controllers/events_controller.rb
def new
  @event = EventPresenter.new :event => Event.new
end

# app/presenters/event_presenter.rb
def possible_organizers
  Organizer.find(:all).map{ |o| [o.name, o.id] }.unshift []
end

def possible_locations
  Location.find(:all).map{ |l| [l.name, l.id] }.unshift []
end
```

Motivation

This is the same refactoring as “Move Data From View Into Presenter”, but it is recognized from the aspect of the controller rather than a view template.

Mechanics

Refer to the “Move Data From View Into Presenter” mechanics.

Example

Refer to the “Move Data From View Into Presenter” example.

References

None.

Move Model Logic Into the Model

The controller is performing duties which should be done by the model.

Move the behavior into the model.

```
# app/controllers/stories_controller.rb
def reorder
  story_ids = params["story_ids"].delete_if{ |id| id.blank? }
  values = []
  story_ids.each_with_index { |id,i| values << [id, i+1] }
  columns2import = [:id, :position]
  columns2update = [:position]
  Story.import(
    columns2import,
    values,
    :on_duplicate_key_update => columns2update,
    :validate => false
  )
  render_notice "Priorities have been successfully updated."
rescue
  render_error "There was an error while reordering stories."
end

to

# app/controllers/stories_controller.rb
def reorder
  if Story.reorder(params["story_ids"])
    render_notice "Priorities have been successfully updated."
  else
    render_error "There was an error while reordering stories."
  end
end

# app/models/story.rb
def self.reorder(story_ids)
  story_ids = story_ids.reject{ |id| id.blank? }
  values = []
  story_ids.each_with_index { |id,i| values << [id, i+1] }
  columns2import = [:id, :position]
  columns2update = [:position]
  import(
    columns2import,
    values,
    :on_duplicate_key_update => columns2update,
    :validate => false
  )
end
```

Motivation

Letting a controller action do all of the work will get the job done, but it makes it more difficult to change over time. This refactoring is based on the Inappropriate Intimacy smell. To recap part of the smell from Fowler's book:

Sometimes classes become far too intimate and spend too much time delving in each others' private parts... Overintimate classes need to be broken up as lovers were in ancients days...

When controllers overstep their bounds and require intimate knowledge of the model in order to do something you should extract a method and move it to the model.

Mechanics

- Extract method
- Move method
- Run the tests

Example

Here we see that the StoriesController has taken on some complex Story-reordering logic.

```
# app/controllers/stories_controller.rb
def reorder
  story_ids = params["story_ids"].delete_if{ |id| id.blank? }
  values = []
  story_ids.each_with_index { |id,i| values << [id, i+1] }
  columns2import = [:id, :position]
  columns2update = [:position]
  Story.import(columns2import, values, :on_duplicate_key_update => columns2update, :validate => false)
  render_notice "Priorities have been successfully updated."
rescue
  render_error "There was an error while reordering stories."
end
```

First, I extract a `reorder_stories` method inside of the `StoriesController` to isolate the behavior that I want to move. I do this so I don't miss any local variables. I move any local variables or parameters that will be required to the argument list of the `reorder_stories` method.

```
# app/controllers/stories_controller.rb
def reorder
  reorder_stories(params["story_ids"])
  render_notice "Priorities have been successfully updated."
rescue
  render_error "There was an error while reordering stories."
end

private

def reorder_stories(story_ids)
  story_ids = story_ids.delete_if{ |id| id.blank? }
  values = []
  story_ids.each_with_index { |id,i| values << [id, i+1] }
  columns2import = [:id, :position]
  columns2update = [:position]
  Story.import(
    columns2import,
    values,
    :on_duplicate_key_update => columns2update,
    :validate => false
  )
end
```

At this point no tests should have broken since we haven't changed any behavior or implementation on the StoriesController.

Now I move the method to the Story class, rename it to 'reorder' and update the controller to use it:

```
# app/controllers/stories_controller.rb
def reorder
  Story.reorder(params["story_ids"])
  render_notice "Priorities have been successfully updated."
rescue
  render_error "There was an error while reordering stories."
end

# app/models/story.rb
def self.reorder(story_ids)
  story_ids = story_ids.delete_if{ |id| id.blank? }
  values = []
  story_ids.each_with_index { |id,i| values << [id, i+1] }
  columns2import = [:id, :position]
  columns2update = [:position]
  Story.import(
    columns2import,
    values,
    :on_duplicate_key_update => columns2update,
    :validate => false
  )
end
```

And that's the refactoring!

Additional Comments

Another characteristic in the example above is that the new Story.reorder method has a side-effect. It deletes blank ids from the story_ids argument. This is an unintended consequence for any code that calls Story.reorder which may lead to bugs and painful debugging over the lifetime of the application. I'm going to go an extra step and make this method side-effect free by reject instead of delete_if:

```
# app/models/story.rb
def self.reorder(story_ids)
  story_ids = story_ids.reject{ |id| id.blank? }
  values = []
  story_ids.each_with_index { |id,i| values << [id, i+1] }
  columns2import = [:id, :position]
  columns2update = [:position]
  Story.import(
    columns2import,
    values,
    :on_duplicate_key_update => columns2update,
    :validate => false
  )
end
```

References

None.

Move View Logic From Controller Into Presenter

You find a block of code in a controller action which is computing a set of values or conditions used in the view.

Extract a presenter as the object which supplies the information for the view and move the block of code from the controller action into a method on it.

```
# app/controllers/people_controller.rb
def edit
  @person = Person.find(params[:id])
  @groups = @person.groups.sort_by{ |group| group.name }
  @active_projects, @inactive_projects = @person.projects.partition do |project|
    project.recently_active?(current_user)
  end
end
```

to

```
# app/controllers/people_controller.rb
def edit
  @person = Person.find(params[:id])
  @groups = PersonPresenter.new(:person => @person).groups
end
```

```
# app/presenters/person_presenter.rb
def groups
  ...
end

def active_projects
  ...
end

def inactive_projects
  ...
end
```

Motivation

View logic can end up in the controller. Moving this logic out of the controller into an object responsible for handling view logic removes unneeded responsibility and confusion from the controller.

After applying this refactoring you may end up with several instance variable assignments which go together. You can pull these data clumps out into a presenter by doing Move Data From Controller Into Presenter.

Mechanics

- Replace the logic in the controller with a method call to the presenter
- Move the code block into a method on the presenter
- Update the controller to use the presenter
- Run your tests

- Rinse and repeat

Example

Here's an example where the controller makes some upfront decisions for the view:

```
# app/controllers/people_controller.rb
def edit
  @person = Person.find(params[:id])
  @groups = @person.groups.sort_by{ |group| group.name }
  @active_projects, @inactive_projects = @person.projects.partition do |project|
    project.recently_active?(current_user)
  end
end
```

I start by giving PersonPresenter the responsibility of deciding the order in which groups should appear:

```
# app/presenters/person_presenter.rb
class PersonPresenter

  def initialize(options)
    @person = options[:person]
  end

  def groups
    @person.groups.sort_by{ |group| group.name }
  end
end
```

I then update the controller to use the PersonPresenter for groups:

```
# app/controllers/people_controller.rb
def edit
  @person = Person.find(params[:id])
  person_presenter = PersonPresenter.new :person => @person
  @groups = person_presenter.groups
  @active_projects, @inactive_projects = @person.projects.partition do |project|
    project.recently_active?(current_user)
  end
end
```

After running tests and ensuring everything still works as expected I turn my attention to the assignment of active and inactive projects. I move this functionality out into the PersonPresenter as two separate methods while keeping the partitioning intact:

```
# app/presenters/person_presenter.rb
class PersonPresenter
  ...

  def active_projects
    partition_projects.first
  end

  def inactive_projects
    partition_projects.last
  end
end
```

```
private

def partition_projects
  @person.projects.partition do |project|
    project.recently_active?(current_user)
  end
end
end
```

Lastly, I update the controller to use the `active_projects` and `inactive_projects` on the `PersonPresenter`:

```
# app/controllers/people_controller.rb
def edit
  @person = Person.find(params[:id])
  person_presenter = PersonPresenter.new :person => @person
  @groups = person_presenter.groups
  @active_projects = person_presenter.active_projects
  @inactive_projects = person_presenter.inactive_projects
end
```

And that's the refactoring!

Additional Comments

None.

References

None.

Move Extrinsic Callback Into Observer

There is a callback in a model which performs duties outside of the model.

Move the callback into an Observer.

```
# app/models/project.rb
class Project < ActiveRecord::Base
  after_create :send_create_notifications

  private

  def send_create_notifications
    self.members.each do |member|
      ProjectNotifier.deliver_project_created_notification(self, member)
    end
  end
end

to

# app/models/project.rb
class Project < ActiveRecord::Base
end

# app/observers/project_notification_observer.rb
class ProjectNotificationObserver < ActiveRecord::Observer
  observe Project

  def after_create(project)
    project.members.each do |member|
      ProjectNotifier.deliver_project_created_notification(project, member)
    end
  end
end
```

Motivation

Callbacks are a great way to hook into the life cycle of a model. Sometimes the behavior that is placed in a model callback is not a part of the model itself. In these situations it is beneficial to move the extrinsic behavior to an external Observer.

This helps decouple the model from services that it does not need to know about. It also allows you to test the behavior of the callback in isolation while leaving your model a little cleaner and clearer.

Mechanics

- Move callback to observer
- Remove callback from model
- Run tests

Example

I'm going to start with a simple ActiveRecord model which notifies its members that it has been created:

```
# app/models/project.rb
class Project < ActiveRecord::Base
  after_create :send_create_notifications

  private

  def send_create_notifications
    self.members.each do |member|
      ProjectNotifier.deliver_project_created_notification(self, member)
    end
  end
end
```

While it may be an important application requirement to notify any founding project member that the project was successfully created, it definitely is not the responsibility of the Project to know how to send out those notifications.

I'll start by moving the callback to a ProjectNotificationObserver:

```
# app/observers/project_notification_observer.rb
class ProjectNotificationObserver < ActiveRecord::Observer
  observe Project

  def after_create(project)
    project.members.each do |member|
      ProjectNotifier.deliver_project_created_notification(project, member)
    end
  end
end
```

Next, I simply remove the unneeded callback from the model:

```
# app/models/project.rb
class Project < ActiveRecord::Base
end
```

And that's the refactoring!

Additional Comments

None.

References

None.

Move View Logic From Model Into Presenter

The model is performing duties required by the view.

Move the view logic into a presenter.

```
# app/models/event.rb
class Event < ActiveRecord::Base

  def description_html
    RedCloth.new(description).to_html
  end

  ...
end

to

# app/models/event.rb
class Event < ActiveRecord::Base
  ...
end

# app/presenters/event_presenter.rb
class EventPresenter

  def initialize(options)
    @event = options[:event]
  end

  def description_html
    RedCloth.new(@event.description).to_html
  end

end
```

Motivation

It is easy to add additional responsibilities to the model when you have needs in the view which are closely related to the model.

This introduces responsibilities to the model which should really be handled by a separate object. These responsibilities may start out as a convenience but they can easily lead to a lot of pollution in the model making it less readable as well as more difficult to maintain.

Extracting these view related responsibilities onto a presenter allow you to leave your model uncluttered while giving home to the view logic.

As the Pragmatic Programmer says, “Separate Views from Models”. [1]

Mechanics

- Move view related method into presenter
- Update where your object is assigned for use in a view to use the presenter

- Run tests

Example

This model formats the description of an event for use in an email body:

```
# app/models/event.rb
class Event < ActiveRecord::Base

  def description_html
    RedCloth.new(description).to_html
  end

end

# app/mailers/event_mailer.rb
class EventMailer < ActionMailer::Base

  def event_update_notification(event)
    subject "Event #{event.name} updated"
    body    'event' => event
    recipients event.participants.map(&:email_address)
    from    "events@example.com"
  end

end

# app/views/event_mailer/event_update_notification.html.erb
<h1><%= h(@event.name) %> Update</h1>

<h2>Event description:</h2>

<%= @event.description_html %>
```

First I create a presenter and move the `description_html` method to it:

```
# app/presenters/event_presenter.rb
class EventPresenter
  def initialize(options)
    @event = options[:event]
  end

  delegate :id, :name, :participants, :to => :@event

  def description_html
    RedCloth.new(@event.description).to_html
  end
end
```

Next I remove `description_html` from the model:

```
# app/models/event.rb
class Event < ActiveRecord::Base
end
```

Lastly, I need to find where the event is being assigned so I can update it to use the `EventPresenter`. I find where the `description_html` method is being used and I back track from there. This takes me to the `EventMailer#event_update_notification` method.

I update the EventMailer to use the EventPresenter:

```
# app/mailers/event_mailer.rb
class EventMailer < ActionMailer::Base

  def event_update_notification(event)
    event = EventPresenter.new(event)
    subject "Event #{event.name} updated"
    body    'event' => event
    recipients event.participants.map(&:email_address)
    from    "events@example.com"
  end

end
```

And that's refactoring!

Additional Comments

If EventMailer#event_update_notification and the views it rendered required several methods which were specific to presenting an event in an email I would probably have created an EventNotificationPresenter rather than just use a generic EventPresenter.

When a particular purpose or UI component has enough custom behavior that it relies on I like to extract that out into a presenter to represent that purpose or component. Most of the time I find that having a generic presenter to represent distinct models gets you most of the way there.

References

- The Pragmatic Programmer by Dave Thomas / Andy Hunt
- Single Responsibility Principle – http://en.wikipedia.org/wiki/Single_responsibility_principle
- Uncle Bob's Principles of OOD – <http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>
- Uncle Bob – Single Responsibility Principle – <http://www.objectmentor.com/resources/articles/srp.pdf>

Organize Large Model with Mixins

You have a model that's grown very large, to the point where the model is difficult to understand and navigate.

Extract groups of related functionality into mixins.

```
# app/models/project.rb
class Project
  ...

  def total_points
    ...
  end

  def completed_points
    ...
  end

  ...
end

# spec/models/project_spec.rb
require File.dirname(__FILE__) + '/../spec_helper'

describe Project, "#total_points" do
  ...
end
...

to

# app/models/project.rb
class Project
  include Project::Statistics
  ...
end

# app/models/project_spec.rb
require File.dirname(__FILE__) + '/../spec_helper'

...

# app/models/project/statistics.rb
class Project
  module Statistics
    def total_points
      ...
    end

    def completed_points
      ...
    end
  end
end

# spec/models/project/statistics_spec.rb
require File.dirname(__FILE__) + '/../spec_helper'
```

```
describe Project, "#total_points" do
  ...
end
```

Motivation

Over time, some models may grow large with code relating them to the rest of the system. To manage this complexity, one approach is to extract clumps of related functionality into ruby mixins representing coherent responsibilities of the model.

Mechanics

- Locate methods that represent a functional aspect of your model.
- Create a module.
- Mix the module into your model.
- Move the methods in question into the module.
- Run your tests.
- Create a test file for the new mixin.
- Move your tests for the methods into the new test file.
- Run your tests.

Example

In this example, the Project class has grown rather large, and we've noticed that one of its responsibilities is providing other objects statistics about itself.

```
# app/models/project.rb
class Project
  ...

  def total_points
    ...
  end

  def completed_points
    ...
  end

  ...
end

# spec/models/project_spec.rb
require File.dirname(__FILE__) + '/../spec_helper'

describe Project, "#total_points" do
  ...
end
...
```

First, we create a module to house these methods:

```
# app/models/project.rb
```

```

class Project
  include Project::Statistics
  ...
end

# app/models/project/statistics.rb
class Project
  module Statistics
    def total_points
      ...
    end

    def completed_points
      ...
    end

    ...
  end
end

```

Next, we run the tests to make sure we haven't broken anything. Following that, we extract tests for the moved methods into a separate file.

```

# app/models/project_spec.rb
require File.dirname(__FILE__) + '/../spec_helper'
...

# spec/models/project/statistics_spec.rb
require File.dirname(__FILE__) + '/../../spec_helper'

describe Project, "#total_points" do
  ...
end
...

```

Finally, we run the tests for both project.rb and statistics.rb to ensure they still work.

Additional Comments

Another approach is to create separate objects for each of these responsibilities. This can be awkward for queries and other methods that are intimate with an ActiveRecord's table. Using mixins allows us to mitigate many of the problems with large classes without introducing a fleet of objects with acute feature envy.

References

None.