Ruby Science

The reference for writing fantastic Rails applications.

Contents

Introduction

Ruby on Rails is almost a decade old, and its community has developed a number of principles for building applications that are fast, fun and easy to change: Don't repeat yourself, keep your views dumb, keep your controllers skinny, and keep business logic in your models. These principles carry most applications to their first release or beyond.

However, these principles only get you so far. After a few releases, most applications begin to suffer. Models become fat, classes become few and large, tests become slow and changes become painful. In many applications, there comes a day when the developers realize that there's no going back; the application is a twisted mess and the only way out is a rewrite or a new job.

Fortunately, it doesn't have to be this way. Developers have been using object-oriented programming for several decades and there's a wealth of knowledge out there that still applies to developing applications today. We can use the lessons learned by these developers to write good Rails applications by applying good object-oriented programming.

Ruby Science will outline a process for detecting emerging problems in code and will dive into the solutions, old and new.

The full book contains three catalogs: smells, solutions and principles. This sample contains a few hand-picked chapters from the first two catalogs, published directly from the book, allowing you to get a sense of the content, style and delivery of the product.

If you enjoy the sample, you can get access to the entire book and sample application at:

`http://www.rubyscience.com`

As a purchaser of the book, you also get access to:

- Multiple formats, including HTML, PDF, EPUB and Kindle.
- A complete example application containing code samples referenced in the book.
- Access to a GitHub repository to receive updates as soon as they're pushed.
- Access to GitHub Issues, where you can provide feedback and tell us what you'd like to see.
- And you can ask us your toughest Rails questions!

Contact Us

If you have any questions, or just want to get in touch, drop us a line at learn@thoughtbot.com.

Part I

Code Smells

Long Method

The most common smell in Rails applications is the Long Method.

Long methods are exactly what they sound like: methods that are too long. They're easy to spot.

Symptoms

- If you can't tell exactly what a method does at a glance, it's too long.
- Methods with more than one level of nesting are usually too long.
- Methods with more than one level of abstraction may be too long.
- Methods with a flog score of 10 or higher may be too long.

You can watch out for long methods as you write them, but finding existing methods is easiest with tools like flog:

```
% flog app lib
    72.9: flog total
     5.6: flog/method average

   15.7: QuestionsController#create     app/controllers/questions_controller.rb:9
   11.7: QuestionsController#new       app/controllers/questions_controller.rb:2
    11.0: Question#none
    8.1: SurveysController#create      app/controllers/surveys_controller.rb:6
```

Methods with higher scores are more complicated. Anything with a score higher than 10 is worth looking at, but flog only helps you find potential trouble spots; use your own judgment when refactoring.

Example

For an example of a long method, let's take a look at the highest scored method from flog, `QuestionsController#create`:

```
def create
  @survey = Survey.find(params[:survey_id])
  @submittable_type = params[:submittable_type_id]
  question_params = params.
    require(:question).
    permit(:submittable_type, :title, :options_attributes, :minimum, :maximum)
  @question = @survey.questions.new(question_params)
  @question.submittable_type = @submittable_type

  if @question.save
    redirect_to @survey
  else
    render :new
  end
end
```

Solutions

- Extract method is the most common way to break apart long methods.
- Replace temp with query if you have local variables in the method.

After extracting methods, check for feature envy in the new methods to see if you should employ move method to provide the method with a better home.

Case Statement

Case Statements are a sign that a method contains too much knowledge.

Symptoms

- Case statements that check the class of an object.
- Case statements that check a type code.
- Divergent change caused by changing or adding `when` clauses.
- Shotgun surgery caused by duplicating the case statement.

Actual `case` statements are extremely easy to find. Just grep your codebase for "case." However, you should also be on the lookout for `case`'s sinister cousin, the repetitive `if-elsif`.

Type Codes

Some applications contain type codes—fields that store type information about objects. These fields are easy to add and seem innocent, but result in code that's harder to maintain. A better solution is to take advantage of Ruby's ability to invoke different behavior based on an object's class, called "dynamic dispatch." Using a case statement with a type code inelegantly reproduces dynamic dispatch.

The special `type` column that ActiveRecord uses is not necessarily a type code. The `type` column is used to serialize an object's class to the database so that the

correct class can be instantiated later on. If you're just using the `type` column to let ActiveRecord decide which class to instantiate, this isn't a smell. However, make sure to avoid referencing the `type` column from `case` or `if` statements.

Example

This method summarizes the answers to a question. The summary varies based on the type of question.

```
# app/models/question.rb
def summary
  case question_type
  when 'MultipleChoice'
    summarize_multiple_choice_answers
  when 'Open'
    summarize_open_answers
  when 'Scale'
    summarize_scale_answers
  end
end
```

Note that many applications replicate the same `case` statement, which is a more serious offence. This view duplicates the `case` logic from `Question#summary`, this time in the form of multiple `if` statements:

```
# app/views/questions/_question.html.erb
<% if question.question_type == 'MultipleChoice' -%>
  <ol>
    <% question.options.each do |option| -%>
      <li>
        <%= submission_fields.radio_button :text, option.text, id: dom_id(option) %>
        <%= content_tag :label, option.text, for: dom_id(option) %>
      </li>
    <% end -%>
  </ol>
<% end -%>

<% if question.question_type == 'Scale' -%>
  <ol>
    <% question.steps.each do |step| -%>
      <li>
        <%= submission_fields.radio_button :text, step %>
        <%= submission_fields.label "text_#{step}", label: step %>
      </li>
    <% end -%>
  </ol>
<% end -%>
```

Solutions

- Replace type code with subclasses if the `case` statement is checking a type code, such as `question_type`.
- Replace conditional with polymorphism when the `case` statement is checking the class of an object.
- Use convention over configuration when selecting a strategy based on a string name.

Shotgun Surgery

Shotgun Surgery is usually a more obvious symptom that reveals another smell.

Symptoms

- You have to make the same small change across several different files.
- Changes become difficult to manage because they are hard to keep track of.

Make sure you look for related smells in the affected code:

- Duplicated code
- Case statement
- Feature envy
- Long parameter list

Example

Users' names are formatted and displayed as "First Last" throughout the application. If you want to change the formatting to include a middle initial (example: "First M. Last") you'll need to make the same small change in several places.

```
# app/views/users/show.html.erb
<%= current_user.first_name %> <%= current_user.last_name %>
```

```
# app/views/users/index.html.erb
<%= current_user.first_name %> <%= current_user.last_name %>

# app/views/layouts/application.html.erb
<%= current_user.first_name %> <%= current_user.last_name %>

# app/views/mailers/completion_notification.html.erb
<%= current_user.first_name %> <%= current_user.last_name %>
```

Solutions

- Replace conditional with polymorphism to replace duplicated `case` statements and `if-elsif` blocks.
- Replace conditional with null object if changing a method to return `nil` would require checks for `nil` in several places.
- Extract decorator to replace duplicated display code in views/templates.
- Introduce parameter object to hang useful formatting methods alongside a data clump of related attributes.
- Use convention over configuration to eliminate small steps that can be inferred based on a convention, such as a name.
- Inline class if the class only serves to add extra steps when performing changes.

Prevention

If your changes become spread out because you need to pass information between boundaries for dependencies, try inverting control.

If you find yourself repeating the exact same change in several places, make sure that you Don't Repeat Yourself.

If you need to change several places because of a modification in your dependency chain, such as changing `user.plan.price` to `user.account.plan.price`, make sure that you're following the law of Demeter.

If conditional logic is affected in several places by a single, cohesive change, make sure that you're following tell, don't ask.

Part II

Solutions

Replace Conditional with Null Object

Every Ruby developer is familiar with `nil`, and Ruby on Rails comes with a full complement of tools to handle it: `nil?`, `present?`, `try` and more. However, it's easy to let these tools hide duplication and leak concerns. If you find yourself checking for `nil` all over your codebase, try replacing some of the `nil` values with Null Objects.

Uses

- Removes shotgun surgery when an existing method begins returning `nil`.
- Removes duplicated code related to checking for `nil`.
- Removes clutter, improving readability of code that consumes `nil`.
- Makes logic related to presence and absence easier to reuse, making it easier to avoid duplication.
- Replaces conditional logic with simple commands, following tell, don't ask.

Example

```
# app/models/question.rb
def most_recent_answer_text
  answers.most_recent.try(:text) || Answer::MISSING_TEXT
end
```

The `most_recent_answer_text` method asks its `answers` association for `most_recent` answer. It only wants the `text` from that answer, but it must first check to make sure that an answer actually exists to get `text` from. It needs to perform this check because `most_recent` might return `nil`:

```
# app/models/answer.rb
def self.most_recent
  order(:created_at).last
end
```

This call clutters up the method, and returning `nil` is contagious: Any method that calls `most_recent` must also check for `nil`. The concept of a missing answer is likely to come up more than once, as in this example:

```
# app/models/user.rb
def answer_text_for(question)
  question.answers.for_user(self).try(:text) || Answer::MISSING_TEXT
end
```

Again, `most_recent_answer_text` might return `nil`:

```
# app/models/answer.rb
def self.for_user(user)
  joins(:completion).where(completions: { user_id: user.id }).last
end
```

The `User#answer_text_for` method duplicates the check for a missing answer—and worse, it's repeating the logic of what happens when you need text without an answer.

We can remove these checks entirely from `Question` and `User` by introducing a null object:

```
# app/models/question.rb
def most_recent_answer_text
  answers.most_recent.text
end
```

```
# app/models/user.rb
def answer_text_for(question)
  question.answers.for_user(self).text
end
```

We're now just assuming that `Answer` class methods will return something answer-like; specifically, we expect an object that returns useful `text`. We can refactor `Answer` to handle the `nil` check:

```
# app/models/answer.rb
class Answer < ActiveRecord::Base
  include ActiveModel::ForbiddenAttributesProtection

  belongs_to :completion
  belongs_to :question

  validates :text, presence: true

  def self.for_user(user)
    joins(:completion).where(completions: { user_id: user.id }).last ||
      NullAnswer.new
  end

  def self.most_recent
    order(:created_at).last || NullAnswer.new
  end
end
```

Note that `for_user` and `most_recent` return a `NullAnswer` if no answer can be found, so these methods will never return `nil`. The implementation for `NullAnswer` is simple:

```
# app/models/null_answer.rb
class NullAnswer
  def text
    'No response'
  end
end
```

We can take things just a little further and remove a bit of duplication with a quick extract method:

```
# app/models/answer.rb
class Answer < ActiveRecord::Base
  include ActiveModel::ForbiddenAttributesProtection

  belongs_to :completion
  belongs_to :question

  validates :text, presence: true

  def self.for_user(user)
    joins(:completion).where(completions: { user_id: user.id }).last_or_null
  end

  def self.most_recent
    order(:created_at).last_or_null
  end

  private

  def self.last_or_null
    last || NullAnswer.new
  end
end
```

Now we can easily create `Answer` class methods that return a usable answer, no matter what.

Drawbacks

Introducing a null object can remove duplication and clutter. But it can also cause pain and confusion:

- As a developer reading a method like `Question#most_recent_answer_text`,

you may be confused to find that `most_recent_answer` returned an instance of `NullAnswer` and not `Answer`.

- It's possible some methods will need to distinguish between `NullAnswer`s and real `Answer`s. This is common in views, when special markup is required to denote missing values. In this case, you'll need to add explicit `present?` checks and define `present?` to return `false` on your null object.
- `NullAnswer` may eventually need to reimplement large part of the `Answer` API, leading to potential duplicated code and shotgun surgery, which is largely what we hoped to solve in the first place.

Don't introduce a null object until you find yourself swatting enough `nil` values to grow annoyed. And make sure the removal of the `nil`-handling logic outweighs the drawbacks above.

Next Steps

- Look for other `nil` checks of the return values of refactored methods.
- Make sure your null object class implements the required methods from the original class.
- Make sure no duplicated code exists between the null object class and the original.

Truthiness, `try` and Other Tricks

All checks for `nil` are a condition, but Ruby provides many ways to check for `nil` without using an explicit `if`. Watch out for `nil` conditional checks disguised behind other syntax. The following are all roughly equivalent:

```
# Explicit if with nil?
if user.nil?
  nil
else
  user.name
end

# Implicit nil check through truthy conditional
if user
  user.name
end

# Relies on nil being falsey
user && user.name

# Call to try
user.try(:name)
```

Extract Method

The simplest refactoring to perform is extract method. To extract a method:

- Pick a name for the new method.
- Move extracted code into the new method.
- Call the new method from the point of extraction.

Uses

- Removes long methods.
- Sets the stage for moving behavior via move method.
- Resolves obscurity by introducing intention-revealing names.
- Allows removal of duplicated code by moving the common code into the extracted method.
- Reveals complexity, making it easier to follow the single responsibility principle.
- Makes behavior easier to reuse, which makes it easier to avoid duplication.

Example

Let's take a look at an example of long method and improve it by extracting smaller methods:

```
def create
  @survey = Survey.find(params[:survey_id])
  @submittable_type = params[:submittable_type_id]
  question_params = params.
    require(:question).
    permit(:submittable_type, :title, :options_attributes, :minimum, :maximum)
  @question = @survey.questions.new(question_params)
  @question.submittable_type = @submittable_type

  if @question.save
    redirect_to @survey
  else
    render :new
  end
end
```

This method performs a number of tasks:

- It finds the survey that the question should belong to.
- It figures out what type of question we're creating (the `submittable_type`).
- It builds parameters for the new question by applying a white list to the HTTP parameters.
- It builds a question from the given survey, parameters and submittable type.
- It attempts to save the question.
- It redirects back to the survey for a valid question.
- It re-renders the form for an invalid question.

Any of these tasks can be extracted to a method. Let's start by extracting the task of building the question.

```
def create
  @survey = Survey.find(params[:survey_id])
  @submittable_type = params[:submittable_type_id]
  build_question

  if @question.save
    redirect_to @survey
  else
    render :new
  end
end

private

def build_question
  question_params = params.
    require(:question).
    permit(:submittable_type, :title, :options_attributes, :minimum, :maximum)
  @question = @survey.questions.new(question_params)
  @question.submittable_type = @submittable_type
end
```

The `create` method is already much more readable. The new `build_question` method is noisy, though, with the wrong details at the beginning. The task of pulling out question parameters is clouding up the task of building the question. Let's extract another method.

Replace Temp with Query

One simple way to extract methods is by replacing local variables. Let's pull `question_params` into its own method:

```
def build_question
  @question = @survey.questions.new(question_params)
  @question.submittable_type = @submittable_type
end

def question_params
  params.
    require(:question).
    permit(:submittable_type, :title, :options_attributes, :minimum, :maximum)
end
```

Other Examples

For more examples of extract method, take a look at these chapters:

- Extract class: b434954d, 000babe1
- Extract decorator: 15f5b96e
- Introduce explaining variable (inline)
- Move method: d5b4871
- Replace conditional with null object: 1e35c68

Next Steps

- Check the original method and the extracted method to make sure neither is a long method.
- Check the original method and the extracted method to make sure that they both relate to the same core concern. If the methods aren't highly related, the class will suffer from divergent change.
- Check newly extracted methods for feature envy. If you find some, you may wish to employ move method to provide the new method with a better home.

- Check the affected class to make sure it's not a large class. Extracting methods reveals complexity, making it clearer when a class is doing too much.

Extract Partial

Extracting a partial is a technique used for removing complex or duplicated view code from your application. This is the equivalent of using long method and extract method in your views and templates.

Uses

- Removes duplicated code from views.
- Avoids shotgun surgery by forcing changes to happen in one place.
- Removes divergent change by removing a reason for the view to change.
- Groups common code.
- Reduces view size and complexity.
- Makes view logic easier to reuse, which makes it easier to avoid duplication.

Steps

- Create a new file for partial prefixed with an underscore (_filename.html.erb).
- Move common code into newly created file.
- Render the partial from the source file.

Example

Let's revisit the view code for *adding* and *editing* questions.

Note: There are a few small differences in the files (the URL endpoint and the label on the submit button).

```
# app/views/questions/new.html.erb
<h1>Add Question</h1>

<%= simple_form_for @question, as: :question, url: survey_questions_path(@survey) do |form|
  <%= form.hidden_field :type %>
  <%= form.input :title %>
  <%= render "#{@question.to_partial_path}_form", question: @question, form: form %>
  <%= form.submit 'Create Question' %>
<% end -%>


# app/views/questions/edit.html.erb
<h1>Edit Question</h1>

<%= simple_form_for @question, as: :question, url: question_path do |form| -%>
  <%= form.hidden_field :type %>
  <%= form.input :title %>
  <%= render "#{@question.to_partial_path}_form", question: @question, form: form %>
  <%= form.submit 'Update Question' %>
<% end -%>
```

First, extract the common code into a partial, remove any instance variables and use `question` and `url` as local variables:

```
# app/views/questions/_form.html.erb
<%= simple_form_for question, as: :question, url: url do |form| -%>
  <%= form.hidden_field :type %>
  <%= form.input :title %>
  <%= render "#{question.to_partial_path}_form", question: question, form: form %>
  <%= form.submit %>
<% end -%>
```

Move the submit button text into the locales file:

```
# config/locales/en.yml
en:
  helpers:
    submit:
      question:
        create: 'Create Question'
        update: 'Update Question'
```

Then render the partial from each of the views, passing in the values for `question` and `url`:

```
# app/views/questions/new.html.erb
<h1>Add Question</h1>

<%= render 'form', question: @question, url: survey_questions_path(@survey) %>
```

```
# app/views/questions/edit.html.erb
<h1>Edit Question</h1>

<%= render 'form', question: @question, url: question_path %>
```

Next Steps

- Check for other occurrences of the duplicated view code in your application and replace them with the newly extracted partial.

Closing

Thanks for checking out this sample of *Ruby Science*. If you'd like to get access to the full content, the example application, ongoing updates and the opportunity to have your questions about Ruby on Rails answered by us, you can get it all on our website:

http://www.rubyscience.com