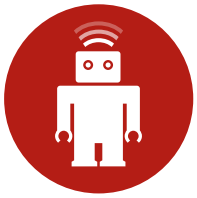
Ruby Science

The reference for writing fantastic Rails applications.



Ruby Science

thoughtbot Joe Ferris Harlow Ward

January 4, 2013

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 which 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.

Code Reviews

The first step towards cleaner code is to make sure you read the code as you write it. Have you ever typed up a long e-mail, hit "Send," and then realized later that you made several typos? The problem here is obvious: you didn't read what you'd written before sending it. Proofreading your e-mails will save you from all kinds of embarrassments. Proofreading your code will do the same.

An easy way to make it simple to proofread code is to always work on a feature branch. Never commit directly to your master branch; doing so will make it tempting to either push code that hasn't been reviewed, or keep code on your local machine. Neither is a good idea.

The first person who should look at every line of code you write is easy to find: it's you! Before merging your feature branch, look at the diff of what you've done. Read through each changed line, each new method, and each new class to make sure that you like what you see. One easy way to make sure that you look at everything before committing it is to use `git add --patch` instead of `git add`. This will force you to confirm each change you make.

If you're working on a team, ask your teammates to review your code as well. After working on the same piece of code for a while, it's easy to develop tunnel vision. Getting a fresh and different perspective will help catch mistakes early. After you review your own code, don't merge your feature branch just yet. Push it up and invite your team members to view the diff as well. When reviewing somebody else's code, take the same approach you took above: page through the diff, and make sure you like everything you see.

Team code reviews provide another benefit: you get immediate feedback on how understandable a piece of code is. Chances are good that you'll understand your own code. After all, you just wrote it. However, you want your team members to understand your code as well. Also, even though the code is clear now, it may not be as obvious looking over it again in six months. Your team members will be a good indicator of what your own understanding will be in the future. If it doesn't make sense to them now, it won't make sense to you later.

Code reviews provide an opportunity to catch mistakes and improve code before it ever gets merged, but there's still a big question out there: what should you be looking for?

Just Follow Your Nose

The primary motivator for refactoring is the code smell. A code smell is an indicator that something may be wrong in the code. Not every smell means that you should fix something; however, smells are useful because they're easy to spot, and the root cause for a particular problem can be harder to track down.

When performing code reviews, be on the lookout for smells. Whenever you see a smell, think about whether or not it would be better if you changed the code to remove the smell. If you're reviewing somebody else's code, suggest possible ways to refactor the code which would remove the smell.

Don't treat code smells as bugs. Attempting to "fix" every smell you run across will end up being a waste of time, as not every smell is the symptom of an actual problem. Worse, removing code smells for the sake of process will end up obfuscating code because of the unnecessary hoops you'll jump through. In the end, it will prove impossible to remove every smell, as removing one smell will often introduce another.

Each smell is associated with one or more common refactorings. If you see a long method, the most common way to improve the method is to extract new, smaller methods. Knowing the common refactorings that remove a smell will allow you to quickly think about how the code might change. Knowing that long methods can be removed by extracting methods, you can decide whether or not the end result of having several methods will be better or worse.

Removing Resistance

There's another obvious opportunity for refactoring: any time you're having a hard time introducing a change to existing code, consider refactoring the code first. What you change will depend on what type of resistance you met.

Did you have a hard time understanding the code? If the result you wanted seemed simple, but you couldn't figure out where to introduce it, the code isn't readable enough. Refactor the code until it's obvious where your change belongs, and it will make this change and every subsequent change easier. Refactor for readability first.

Was it hard to change the code without breaking existing code? Change the existing code to be more flexible. Add extension points or extract code to be easier to reuse, and then try to introduce your change. Repeat this process until the change you want is easy to introduce.

This work flow pairs well with fast branching systems like Git. First, create a new branch and attempt to make your change without any refactoring. If the change is difficult, make a work in progress commit, switch back to master, and create a new branch for refactoring. Refactor until you fix the resistance you met on your feature branch, and then rebase the feature branch on top of the refactoring branch. If the change is easier now, you're good to go. If not, switch back to your refactoring branch and try again.

Each change should be easy to introduce. If it's not, it's time to refactor.

Bugs and Churn

If you're spending a lot of time swatting bugs, you should consider refactoring the buggy portions of code. After each bug is fixed, examine the methods and classes you had to change to fix the bug. If you remove any smells you discover in the affected areas, then you'll make it less likely that a bug will be reintroduced.

Bugs tend to crop up in the same places over and over. These places also tend to be the methods and classes with the highest rate of churn. When you find a bug, use Git to see if the buggy file changes often. If so, try refactoring the classes or methods which keep changing. If you separate the pieces that change often from

the pieces that don't, then you'll spend less time fixing existing code. When you find files with high churn, look for smells in the areas that keep changing. The smell may reveal the reason for the high churn.

Conversely, it may make sense to avoid refactoring areas with low churn. Although refactoring is an important part of keeping your code sane, refactoring changes code, and with each change, you risk introducing new bugs. Don't refactor just for the sake of "cleaner" code; refactor to address real problems. If a file hasn't changed in six months and you aren't finding bugs in it, leave it alone. It may not be the prettiest thing in your code base, but you'll have to spend more time looking at it when you break it while trying to fix something that wasn't broken.

Metrics

Various tools are available which can aid you in your search for code smells.

You can use flog to detect complex parts of code. If you look at the classes and methods with the highest flog score, you'll probably find a few smells worth investigating.

Duplication is one of the hardest problems to find by hand. If you're using diffs during code reviews, it will be invisible when you copy and paste existing methods. The original method will be unchanged and won't show up in the diff, so unless the reviewer knows and remembers that the original existed, they won't notice that the copied method isn't just a new addition. Use flay to find duplication. Every duplicated piece of code is a bug waiting to happen.

When looking for smells, reek can find certain smells reliably and quickly. Attempting to maintain a "reek free" code base is costly, but using reek once you discover a problematic class or method may help you find the solution.

To find files with a high churn rate, try out the aptly-named churn gem. This works best with Git, but will also work with Subversion.

You can also use Code Climate, a hosted tool which will scan your code for issues every time you push to Git. Code Climate attempts to locate hot spots for refactoring and assigns each class a simple A through F grade.

Getting obsessed with the counts and scores from these tools will distract from the actual issues in your code, but it's worthwhile to run them continually and watch out for potential warning signs.

How To Read This Book

This book contains three catalogs: smells, solutions, and principles.

Start by looking up a smell that sounds familiar. Each chapter on smells explain the potential problems each smell may reveal and will reference possible solutions.

Once you've identified the problem revealed by a smell, read the relevant solution chapter to learn how to fix it. Each solution chapter will explain which problems it addresses and potential problems which can be introduced.

Lastly, smell and solution chapters will reference related principles. The smell chapters will reference principles that you can follow to avoid the root problem in the future. The solution chapters will explain how each solution changes your code to follow related principles.

By following this process, you'll learn how to detect and fix actual problems in your code using smells and reusable solutions, and you'll learn about principles that you can follow to improve the code you write from the beginning.

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 which 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 will only help you find potential trouble spots; use your own judgement 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.

Large Class

Most Rails applications suffer from several Large Classes. Large classes are difficult to understand and they make it harder to change or reuse behavior. Tests for large classes are slow and churn tends to be higher, leading to more bugs and conflicts. Large classes likely also suffer from Divergent Change.

Symptoms

- You can't easily describe what the class does in one sentence.
- You can't tell what the class does without scrolling.
- The class needs to change for more than one reason.
- The class has more private methods than public methods.
- The class has more than 7 methods.
- The class has a total flog score of 50.

Example

This class has a high flog score, has a large number of methods, more private than public methods, and has multiple responsibility:

```
# app/models/question.rb
class Question < ActiveRecord::Base
  include ActiveModel::ForbiddenAttributesProtection

  SUBMITTABLE_TYPES = %w(Open MultipleChoice Scale).freeze

  validates :maximum, presence: true, if: :scale?
  validates :minimum, presence: true, if: :scale?
  validates :question_type, presence: true, inclusion: SUBMITTABLE_TYPES
  validates :title, presence: true

  belongs_to :survey
  has_many :answers
  has_many :options

  accepts_nested_attributes_for :options, reject_if: :all_blank

  def summary
    case question_type
    when 'MultipleChoice'
      summarize_multiple_choice_answers
    when 'Open'
      summarize_open_answers
    when 'Scale'
      summarize_scale_answers
    end
  end

  def steps
    (minimum..maximum).to_a
  end

  private

  def scale?
    question_type == 'Scale'
  end

  def summarize_multiple_choice_answers
    total = answers.count
    counts = answers.group(:text).order('COUNT(*) DESC').count
```

```
    percents = counts.map do |text, count|
      percent = (100.0 * count / total).round
      "#{percent}% #{text}"
    end
    percents.join(', ')
  end

  def summarize_open_answers
    answers.order(:created_at).pluck(:text).join(', ')
  end

  def summarize_scale_answers
    sprintf('Average: %.02f', answers.average('text'))
  end
end
```

Solutions

- Move Method to move methods to another class if an existing class could better handle the responsibility.
- Extract Class if the class has multiple responsibilities.
- Replace Conditional with Polymorphism if the class contains private methods related to conditional branches.
- Extract Value Object if the class contains private query methods.
- Extract Decorator if the class contains delegation methods.
- Extract Service Object if the class contains numerous objects related to a single action.

Prevention

Following the Single Responsibility Principle will prevent large classes from cropping up. It's difficult for any class to become too large without taking on more than one responsibility.

You can use flog to analyze classes as you write and modify them:

```
% flog -a app/models/question.rb
    48.3: flog total
     6.9: flog/method average

    15.6: Question#summarize_multiple_choice_answers app/models/question.rb:38
    12.0: Question#none
     6.3: Question#summary                  app/models/question.rb:17
     5.2: Question#summarize_open_answers   app/models/question.rb:48
     3.6: Question#summarize_scale_answers  app/models/question.rb:52
     3.4: Question#steps                    app/models/question.rb:28
     2.2: Question#scale?                   app/models/question.rb:34
```

God Class

A particular specimen of Large Class affects most Rails applications: the God Class. A God Class is any class that seems to know everything about an application. It has a reference to the majority of the other models, and it's difficult to answer any question or perform any action in the application without going through this class.

Most applications have two God Classes: User, and the central focus of the application. For a todo list application, it will be User and Todo; for photo sharing application, it will be User and Photo.

You need to be particularly vigilant about refactoring these classes. If you don't start splitting up your God Classes early on, then it will become impossible to separate them without rewriting most of your application.

Treatment and prevention of God Classes is the same as for any Large Class.

Feature Envy

Feature envy reveals a method (or method-to-be) that would work better on a different class.

Methods suffering from feature envy contain logic that is difficult to reuse, because the logic is trapped within a method on the wrong class. These methods are also often private methods, which makes them unavailable to other classes. Moving the method (or affected portion of a method) to a more appropriate class improves readability, makes the logic easier to reuse, and reduces coupling.

Symptoms

- Repeated references to the same object.
- Parameters or local variables which are used more than methods and instance variables of the class in question.
- Methods that includes a class name in their own names (such as `invite_user`).
- Private methods on the same class that accept the same parameter.
- Law of Demeter violations.
- Tell, Don't Ask violations.

Example

```ruby
# app/models/completion.rb
def score
  answers.inject(0) do |result, answer|
    question = answer.question
    result + question.score(answer.text)
  end
end
```

The `answer` local variable is used twice in the block: once to get its `question`, and once to get its `text`. This tells us that we can probably extract a new method and move it to the `Answer` class.

Solutions

- Extract Method if only part of the method suffers from feature envy, and then move the method.
- Move Method if the entire method suffers from feature envy.

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 they 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.

High Fan-out

STUB

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
- Parallel Inheritance Hierarchies

Example

Users names are formatted and displayed as 'First Last' throughout the application. If we want to change the formating to include a middle initial (e.g. 'First M. Last') we'd 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.

Divergent Change

STUB

Long Parameter List

Ruby supports positional method arguments which can lead to Long Parameter Lists.

Symptoms

- You can't easily change the method's arguments.
- The method has three or more arguments.
- The method is complex due to number of collaborating parameters.
- The method requires large amounts of setup during isolated testing.

Example

Look at this mailer for an example of Long Parameter List.

```
# app/mailers/mailer.rb
class Mailer < ActionMailer::Base
  default from: "from@example.com"

  def completion_notification(first_name, last_name, email)
    @first_name = first_name
    @last_name = last_name

    mail(
      to: email,
      subject: 'Thank you for completing the survey'
    )
  end
end
```

Solutions

- Introduce Parameter Object and pass it in as an object of naturally grouped attributes.

A common technique used to mask a long parameter list is grouping parameters using a hash of named parameters; this will replace connascence position with connascence of name (a good first step). However, it will not reduce the number of collaborators in the method.

- Extract Class if the method is complex due to the number of collaborators.

Duplicated Code

One of the first principles we're taught as developers: Keep your code DRY.

Symptoms

- You find yourself copy and pasting code from one place to another.
- Shotgun Surgery occurs when changes to your application require the same small edits in multiple places.

Example

The `QuestionsController` suffers from duplication in the `create` and `update` methods.

```
# app/controllers/questions_controller.rb
def create
  @survey = Survey.find(params[:survey_id])
  question_params = params.
    require(:question).
    permit(:title, :options_attributes, :minimum, :maximum)
  @question = type.constantize.new(question_params)
  @question.survey = @survey

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

def update
  @question = Question.find(params[:id])
  question_params = params.
    require(:question).
    permit(:title, :options_attributes, :minimum, :maximum)
  @question.update_attributes(question_params)

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

Solutions

- Extract Method for duplicated code in the same file.
- Extract Class for duplicated code across multiple files.
- Extract Partial for duplicated view and template code.
- Replace Conditional with Polymorphism for duplicated conditional logic.
- Replace Conditional with Null Object to remove duplicated checks for `nil` values.

Uncommunicative Name

STUB

Parallel Inheritance Hierarchies

STUB

Comments

Comments can be used appropriately to introduce classes and provide documentation, but used incorrectly, they mask readability and process problems by further obfuscating already unreadable code.

Symptoms

- Comments within method bodies.
- More than one comment per method.
- Comments that restate the method name in English.
- TODO comments.
- Commented out, dead code.

Example

```
# app/models/open_question.rb
def summary
  # Text for each answer in order as a comma-separated string
  answers.order(:created_at).pluck(:text).join(', ')
end
```

This comment is trying to explain what the following line of code does, because the code itself is too hard to understand. A better solution would be to improve the legibility of the code.

Some comments add no value at all and can safely be removed:

```
class Invitation
  # Deliver the invitation
  def deliver
    Mailer.invitation_notification(self, message).deliver
  end
end
```

If there isn't a useful explanation to provide for a method or class beyond the name, don't leave a comment.

Solutions

- Introduce Explaining Variable to make obfuscated lines easier to read in pieces.
- Extract Method to break up methods that are difficult to read.
- Move TODO comments into a task management system.
- Delete commented out code, and rely on version control in the event that you want to get it back.
- Delete superfluous comments that don't add more value than the method or class name.

Mixin

STUB

Callback

STUB

Part II

Solutions

Replace Conditional with Polymorphism

Conditional code clutters methods, makes extraction and reuse harder, and can lead to leaky concerns. Object-oriented languages like Ruby allow developers to avoid conditionals using polymorphism. Rather than using `if/else` or `case/when` to create a conditional path for each possible situation, you can implement a method differently in different classes, adding (or reusing) a class for each situation.

Replacing conditional code allows you to move decisions to the best point in the application. Depending on polymorphic interfaces will create classes that don't need to change when the application changes.

Uses

- Removes Divergent Change from classes that need to alter their behavior based on the outcome of the condition.
- Removes Shotgun Surgery from adding new types.
- Removes Feature Envy by allowing dependent classes to make their own decisions.
- Makes it easier to remove Duplicated Code by taking behavior out of conditional clauses and private methods.

Example

This `Question` class summarizes its answers differently depending on its `question_type`:

```
# app/models/question.rb
class Question < ActiveRecord::Base
  include ActiveModel::ForbiddenAttributesProtection

  SUBMITTABLE_TYPES = %w(Open MultipleChoice Scale).freeze

  validates :maximum, presence: true, if: :scale?
  validates :minimum, presence: true, if: :scale?
  validates :question_type, presence: true, inclusion: SUBMITTABLE_TYPES
  validates :title, presence: true

  belongs_to :survey
  has_many :answers
  has_many :options

  accepts_nested_attributes_for :options, reject_if: :all_blank

  def summary
    case question_type
    when 'MultipleChoice'
      summarize_multiple_choice_answers
    when 'Open'
      summarize_open_answers
    when 'Scale'
      summarize_scale_answers
    end
  end

  def steps
    (minimum..maximum).to_a
  end

  private

  def scale?
    question_type == 'Scale'
  end

  def summarize_multiple_choice_answers
    total = answers.count
    counts = answers.group(:text).order('COUNT(*) DESC').count
```

```
    percents = counts.map do |text, count|
      percent = (100.0 * count / total).round
      "#{percent}% #{text}"
    end
    percents.join(', ')
  end

  def summarize_open_answers
    answers.order(:created_at).pluck(:text).join(', ')
  end

  def summarize_scale_answers
    sprintf('Average: %.02f', answers.average('text'))
  end
end
```

There are a number of issues with the `summary` method:

- Adding a new question type will require modifying the method, leading to Divergent Change.
- The logic and data for summarizing every type of question and answer is jammed into the `Question` class, resulting in a Large Class with Obscure Code.
- This method isn't the only place in the application that checks question types, meaning that new types will cause Shotgun Surgery.

Replace Type Code With Subclasses

Let's replace this case statement with polymorphism by introducing a subclass for each type of question.

Our `Question` class is a subclass of `ActiveRecord::Base`. If we want to create subclasses of `Question`, we have to tell ActiveRecord which subclass to instantiate when it fetches records from the `questions` table. The mechanism Rails uses for storing instances of different classes in the same table is called "Single Table Inheritance." Rails will take care of most of the details, but there are a few extra steps we need to take when refactoring to Single Table Inheritance.

Single Table Inheritance (STI)

The first step to convert to STI is generally to create a new subclass for each type. However, the existing type codes are named "Open," "Scale," and "Mul-

tipleChoice," which won't make good class names; names like "OpenQuestion" would be better, so let's start by changing the existing type codes:

```
# app/models/question.rb
def summary
  case question_type
  when 'MultipleChoiceQuestion'
    summarize_multiple_choice_answers
  when 'OpenQuestion'
    summarize_open_answers
  when 'ScaleQuestion'
    summarize_scale_answers
  end
end
```

```
# db/migrate/20121128221331_add_question_suffix_to_question_type.rb
class AddQuestionSuffixToQuestionType < ActiveRecord::Migration
  def up
    connection.update(<<-SQL)
      UPDATE questions SET question_type = question_type || 'Question'
    SQL
  end

  def down
    connection.update(<<-SQL)
      UPDATE questions SET question_type = REPLACE(question_type, 'Question', '')
    SQL
  end
end
```

See commit b535171 for the full change.

The `Question` class stores its type code as `question_type`. The Rails convention is to use a column named `type`, but Rails will automatically start using STI if that column is present. That means that renaming `question_type` to `type` at this point would result in debugging two things at once: possible breaks from renaming, and possible breaks from using STI. Therefore, let's start by just marking `question_type` as the inheritance column, allowing us to debug STI failures by themselves:

```
# app/models/question.rb
set_inheritance_column  'question_type'
```

Running the tests after this will reveal that Rails wants the subclasses to be defined, so let's add some placeholder classes:

```
# app/models/open_question.rb
class OpenQuestion < Question
end

# app/models/scale_question.rb
class ScaleQuestion < Question
end

# app/models/multiple_choice_question.rb
class MultipleChoiceQuestion < Question
end
```

Rails generates URLs and local variable names for partials based on class names. Our views will now be getting instances of subclasses like `OpenQuestion` rather than `Question`, so we'll need to update a few more references. For example, we'll have to change lines like:

```
<%= form_for @question do |form| %>
```

To:

```
<%= form_for @question, as: :question do |form| %>
```

Otherwise, it will generate `/open_questions` as a URL instead of `/questions`. See commit c18ebeb for the full change.

At this point, the tests are passing with STI in place, so we can rename `question_type` to `type`, following the Rails convention:

```
# db/migrate/20121128225425_rename_question_type_to_type.rb
class RenameQuestionTypeToType < ActiveRecord::Migration
  def up
    rename_column :questions, :question_type, :type
  end

  def down
    rename_column :questions, :type, :question_type
  end
end
```

Now we need to build the appropriate subclass instead of `Question`. We can use a little Ruby meta-programming to make that fairly painless:

```
# app/controllers/questions_controller.rb
def build_question
  @question = type.constantize.new(question_params)
  @question.survey = @survey
end

def type
  params[:question][:type]
end
```

At this point, we're ready to proceed with a regular refactor.

Extracting Type-Specific Code

The next step is to move type-specific code from `Question` into the subclass for each specific type.

Let's look at the `summary` method again:

```
# 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
```

For each path of the condition, there is a sequence of steps.

The first step is to use Extract Method to move the path to its own method. In this case, we already extracted methods called `summarize_multiple_choice_answers`, `summarize_open_answers`, and `summarize_scale_answers`, so we can proceed immediately.

The next step is to use Move Method to move the extracted method to the appropriate class. First, let's move the method `summarize_multiple_choice_answers` to `MultipleChoiceQuestion` and rename it to `summary`:

```
class MultipleChoiceQuestion < Question
  def summary
    total = answers.count
    counts = answers.group(:text).order('COUNT(*) DESC').count
    percents = counts.map do |text, count|
      percent = (100.0 * count / total).round
      "#{percent}% #{text}"
    end
    percents.join(', ')
  end
end
```

`MultipleChoiceQuestion#summary` now overrides `Question#summary`, so the correct implementation will now be chosen for multiple choice questions.

Now that the code for multiple choice types is in place, we repeat the steps for each other path. Once every path is moved, we can remove `Question#summary` entirely.

In this case, we've already created all our subclasses, but you can use Extract Class to create them if you're extracting a conditional paths into a new classes.

You can see the full change for this step in commit a08f801.

The `summary` method is now much better. Adding new question types is easier. The new subclass will implement `summary`, and the `Question` class doesn't need to change. The summary code for each type now lives with its type, so no one class is cluttered up with the details.

Polymorphic Partials

Applications rarely check the type code in just one place. Running grep on our example application reveals several more places. Most interestingly, the views check the type before deciding how to render a question:

```
# app/views/questions/_question.html.erb
<% if question.type == 'MultipleChoiceQuestion' -%>
  <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.type == 'ScaleQuestion' -%>
  <ol>
    <% question.steps.each do |step| -%>
      <li>
        <%= submission_fields.radio_button :text, step %>
        <%= submission_fields.label "text_#{step}", label: step %>
      </li>
    <% end -%>
  </ol>
<% end -%>

<% if question.type == 'OpenQuestion' -%>
  <%= submission_fields.text_field :text %>
<% end -%>
```

In the previous example, we moved type-specific code into `Question` subclasses. However, moving view code would violate MVC (introducing Divergent Change into the subclasses), and more importantly, it would be ugly and hard to understand.

Rails has the ability to render views polymorphically. A line like this:

```
<%= render @question %>
```

Will ask `@question` which view should be rendered by calling `to_partial_path`. As subclasses of `ActiveRecord::Base`, our `Question` subclasses will return a path based on their class name. This means that the above line will attempt to

render `open_questions/_open_question.html.erb` for an open question, and so on.

We can use this to move the type-specific view code into a view for each type:

```
# app/views/open_questions/_open_question.html.erb
<%= submission_fields.text_field :text %>
```

You can see the full change in commit 8243493.

Multiple Polymorphic Views

Our application also has different fields on the question form depending on the question type. Currently, that also performs type-checking:

```
# app/views/questions/new.html.erb
<% if @question.type == 'MultipleChoiceQuestion' -%>
  <%= form.fields_for(:options, @question.options_for_form) do |option_fields| -%>
    <%= option_fields.input :text, label: 'Option' %>
  <% end -%>
<% end -%>

<% if @question.type == 'ScaleQuestion' -%>
  <%= form.input :minimum %>
  <%= form.input :maximum %>
<% end -%>
```

We already used views like `open_questions/_open_question.html.erb` for showing a question, so we can't just put the edit code there. Rails doesn't support prefixes or suffixes in `render`, but we can do it ourselves easily enough:

```
# app/views/questions/new.html.erb
<%= render "#{@question.to_partial_path}_form", question: @question, form: form %>
```

This will render `app/views/open_questions/_open_question_form.html.erb` for an open question, and so on.

Drawbacks

It's worth noting that, although this refactoring improved our particular example, replacing conditionals with polymorphism is not without drawbacks.

Using polymorphism like this makes it easier to add new types, because adding a new type means you just need to add a new class and implement the required

methods. Adding a new type won't require changes to any existing classes, and it's easy to understand what the types are, because each type is encapsulated within a class.

However, this change makes it harder to add new behaviors. Adding a new behavior will mean finding every type and adding a new method. Understanding the behavior becomes more difficult, because the implementations are spread out among the types. Object-oriented languages lean towards polymorphic implementations, but if you find yourself adding behaviors much more often than adding types, you should look into using observers or visitors instead.

Next Steps

- Check the new classes for Duplicated Code that can be pulled up into the superclass.
- Pay attention to changes that affect the new types, watching out for Shotgun Surgery that can result from splitting up classes.

Replace conditional with Null Object

Every Ruby developer is familiar with `nil`, and Ruby on Rails comes with a full compliment 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`.

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`.
- Whenever a method needs to worry about whether or not an actual answer exists, you'll need to add explicit `present?` checks and define `present?` to return `false` on your null object. This is common in views, when the view needs to add special markup to denote missing values.

- `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 be annoying, and make sure you're actually cutting down on conditional logic when you introduce it.

Next Steps

- Look for other `nil` checks from 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.

Let's take a look at an example 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
```

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 in the new methods to see if you should employ Move Method to provide the 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 Class

STUB

Extract Value Object

STUB

Extract Decorator

STUB

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

- Remove Duplicated Code from views.
- Remove Shotgun Surgery by forcing changes to happen in one place.
- Remove Divergent Change by removing a reason for the view to change.
- Group common code.
- Reduce view size and complexity.

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 a 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 occurances of the duplicated view code in your application and replace them with the newly extracted partial.

Extract Service Object

STUB

Extract Validator

STUB

Introduce Explaining Variable

This refactoring allows you to break up a complex, hard-to-read statement by placing part of it in a local variable. The only difficult part is finding a good name for the variable.

Uses

- Improves legibility of code.
- Makes it easier to Extract Methods by breaking up long statements.
- Removes the need for extra Comments.

Example

This line of code was hard enough to understand that a comment was added:

```
# app/models/open_question.rb
def summary
  # Text for each answer in order as a comma-separated string
  answers.order(:created_at).pluck(:text).join(', ')
end
```

Adding an explaining variable makes the line easy to understand without a comment:

```
# app/models/open_question.rb
def summary
  text_from_ordered_answers = answers.order(:created_at).pluck(:text)
  text_from_ordered_answers.join(', ')
end
```

You can follow up by using Replace Temp with Query.

```
def summary
  text_from_ordered_answers.join(', ')
end

private

def text_from_ordered_answers
  answers.order(:created_at).pluck(:text)
end
```

This increases the overall size of the class and moves `text_from_ordered_answers` further away from `summary`, so you'll want to be careful when doing this. The most obvious reason to extract a method is to reuse the value of the variable.

However, there's another potential benefit: it changes the way developers read the code. Developers instinctively read code top-down. Expressions based on variables place the details first, which means that a developer will start with the details:

```
text_from_ordered_answers = answers.order(:created_at).pluck(:text)
```

And work their way down to the overall goal of a method:

```
text_from_ordered_answers.join(', ')
```

Note that you naturally focus first on the code necessary to find the array of texts, and then progress to see what happens to those texts.

Once a method is extracted, the high level concept comes first:

```
def summary
  text_from_ordered_answers.join(', ')
end
```

And then you progress to the details:

```
def text_from_ordered_answers
  answers.order(:created_at).pluck(:text)
end
```

You can use this technique of extracting methods to make sure that developers focus on what's important first, and only dive into the implementation details when necessary.

Next Steps

- Replace Temp with Query if you want to reuse the expression or revert the naturally order in which a developer reads the method.
- Check the affected expression to make sure that it's easy to read. If it's still too dense, try extracting more variables or methods.
- Check the extracted variable or method for Feature Envy.

Introduce Observer

STUB

Introduce Form Object

A specialized type of Extract Class used to remove business logic from controllers when processing data outside of an ActiveRecord model.

Uses

- Keep business logic out of Controllers and Views.
- Add validation support to plain old Ruby objects.
- Display form validation errors using Rails conventions.
- Set the stage for Extract Validator.

Example

The `create` action of our `InvitationsController` relies on user submitted data for `message` and `recipients` (a comma delimited list of email addresses).

It performs a number of tasks:

- Finds the current survey.
- Validates the `message` is present.
- Validates each of the `recipients` are email addresses.
- Creates an invitation for each of the recipients.
- Sends an email to each of the recipients.
- Sets view data for validation failures.

```ruby
# app/controllers/invitations_controller.rb
class InvitationsController < ApplicationController
  EMAIL_REGEX = /\A([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\z/

  def new
    @survey = Survey.find(params[:survey_id])
  end

  def create
    @survey = Survey.find(params[:survey_id])
    if valid_recipients? && valid_message?
      recipient_list.each do |email|
        invitation = Invitation.create(
          survey: @survey,
          sender: current_user,
          recipient_email: email,
          status: 'pending'
        )
        Mailer.invitation_notification(invitation, message)
      end
      redirect_to survey_path(@survey), notice: 'Invitation successfully sent'
    else
      @recipients = recipients
      @message = message
      render 'new'
    end
  end

  private

  def valid_recipients?
    invalid_recipients.empty?
  end

  def valid_message?
    message.present?
  end

  def invalid_recipients
    @invalid_recipients ||= recipient_list.map do |item|
      unless item.match(EMAIL_REGEX)
        item
      end
    end.compact
  end
```

```
  def recipient_list
    @recipient_list ||= recipients.gsub(/\s+/, '').split(/[\n,;]+/)
  end

  def recipients
    params[:invitation][:recipients]
  end

  def message
    params[:invitation][:message]
  end
end
```

By introducing a form object we can move the concerns of data validation, invitation creation, and notifications to the new model `SurveyInviter`.

Including ActiveModel::Model allows us to leverage the familiar Active Record Validation syntax.

As we introduce the form object we'll also extract an enumerable class `RecipientList` and validators `EnumerableValidator` and `EmailValidator`. They will be covered in the chapters Extract Class and Extract Validator.

```
# app/models/survey_inviter.rb
class SurveyInviter
  include ActiveModel::Model
  attr_accessor :recipients, :message, :sender, :survey

  validates :message, presence: true
  validates :recipients, length: { minimum: 1 }
  validates :sender, presence: true
  validates :survey, presence: true

  validates_with EnumerableValidator,
    attributes: [:recipients],
    unless: 'recipients.nil?',
    validator: EmailValidator

  def recipients=(recipients)
    @recipients = RecipientList.new(recipients)
  end

  def invite
    if valid?
      deliver_invitations
    end
```

```
  end

  private

  def create_invitations
    recipients.map do |recipient_email|
      Invitation.create!(
        survey: survey,
        sender: sender,
        recipient_email: recipient_email,
        status: 'pending'
      )
    end
  end

  def deliver_invitations
    create_invitations.each do |invitation|
      Mailer.invitation_notification(invitation, message).deliver
    end
  end
end
```

Moving business logic into the new form object dramatically reduces the size and complexity of the `InvitationsController`. The controller is now focused on the interaction between the user and the models.

```
# app/controllers/invitations_controller.rb
class InvitationsController < ApplicationController
  def new
    @survey = Survey.find(params[:survey_id])
    @survey_inviter = SurveyInviter.new
  end

  def create
    @survey = Survey.find(params[:survey_id])
    @survey_inviter = SurveyInviter.new(survey_inviter_params)

    if @survey_inviter.invite
      redirect_to survey_path(@survey), notice: 'Invitation successfully sent'
    else
      render 'new'
    end
  end

  private
```

```
  def survey_inviter_params
    params.require(:survey_inviter).permit(
      :message,
      :recipients
    ).merge(
      sender: current_user,
      survey: @survey
    )
  end
end
```

Next Steps

- Check that the controller no longer has Long Methods.
- Verify the new form object is not a Large Class.
- Check for places to re-use any new validators if Extract Validator was used during the refactoring.

Introduce Parameter Object

A technique to reduce the number of input parameters to a method.

To introduce a parameter object:

- Pick a name for the object that represents the grouped parameters.
- Replace method's grouped parameters with the object.

Uses

- Remove Long Parameter Lists.
- Group parameters that naturally fit together.
- Encapsulate behavior between related parameters.

Let's take a look at the example from Long Parameter List and improve it by grouping the related parameters into an object:

```
# app/mailers/mailer.rb
class Mailer < ActionMailer::Base
  default from: "from@example.com"

  def completion_notification(first_name, last_name, email)
    @first_name = first_name
    @last_name = last_name

    mail(
      to: email,
      subject: 'Thank you for completing the survey'
    )
  end
end
```

```
# app/views/mailer/completion_notification.html.erb
<%= @first_name %> <%= @last_name %>
```

By introducing the new parameter object `recipient` we can naturally group the attributes `first_name`, `last_name`, and `email` together.

```
# app/mailers/mailer.rb
class Mailer < ActionMailer::Base
  default from: "from@example.com"

  def completion_notification(recipient)
    @recipient = recipient

    mail(
      to: recipient.email,
      subject: 'Thank you for completing the survey'
    )
  end
end
```

This also gives us the opportunity to create a new method `full_name` on the `recipient` object to encapsulate behavior between the `first_name` and `last_name`.

```
# app/views/mailer/completion_notification.html.erb
<%= @recipient.full_name %>
```

Next Steps

- Check to see if the same Data Clump exists elsewhere in the application, and reuse the Parameter Object to group them together.
- Verify the methods using the Parameter Object don't have Feature Envy.

Use class as Factory

STUB

Move method

Moving methods is generally easy. Moving a method allows you to place a method closer to the state it uses by moving it to the class which owns the related state.

To move a method:

- Move the entire method definition and body into the new class.
- Change any parameters which are part of the state of the new class to simply reference the instance variables or methods.
- Introduce any necessary parameters because of state which belongs to the old class.
- Rename the method if the new name no longer makes sense in the new context (for example, rename `invite_user` to `invite` once the method is moved to the `User` class).
- Replace calls to the old method to calls to the new method. This may require introducing delegation or building an instance of the new class.

Uses

- Remove Feature Envy by moving a method to the class where the envied methods live.
- Make private, parameterized methods easier to reuse by moving them to public, unparameterized methods.
- Improve readability by keeping methods close to the other methods they use.

Let's take a look at an example method that suffers from Feature Envy and use Extract Method and Move Method to improve it:

```
# app/models/completion.rb
def score
  answers.inject(0) do |result, answer|
    question = answer.question
    result + question.score(answer.text)
  end
end
```

The block in this method suffers from Feature Envy: it references `answer` more than it references methods or instance variables from its own class. We can't move the entire method; we only want to move the block, so let's first extract a method:

```
# app/models/completion.rb
def score
  answers.inject(0) do |result, answer|
    result + score_for_answer(answer)
  end
end

# app/models/completion.rb
def score_for_answer(answer)
  question = answer.question
  question.score(answer.text)
end
```

The `score` method no longer suffers from Feature Envy, and the new `score_for_answer` method is easy to move, because it only references its own state. See the chapter on Extract Method for details on the mechanics and properties of this refactoring.

Now that the Feature Envy is isolated, let's resolve it by moving the method:

```
# app/models/completion.rb
def score
  answers.inject(0) do |result, answer|
    result + answer.score
  end
end

# app/models/answer.rb
def score
  question.score(text)
end
```

The newly extracted and moved `Question#score` method no longer suffers from Feature Envy. It's easier to reuse, because the logic is freed from the internal block in `Completion#score`. It's also available to other classes, because it's no longer a private method. Both methods are also easier to follow, because the methods they invoke are close to the methods they depend on.

Dangerous: move and extract at the same time

It's tempting to do everything as one change: create a new method in `Answer`, move the code over from `Completion`, and change `Completion#score` to call the new method. Although this frequently works without a hitch, with practice, you can perform the two, smaller refactorings just as quickly as the single, larger refactoring. By breaking the refactoring into two steps, you reduce the duration of "down time" for your code; that is, you reduce the amount of time during which something is broken. Improving code in tiny steps makes it easier to debug when something goes wrong and prevents you from writing more code than you need to. Because the code still works after each step, you can simply stop whenever you're happy with the results.

Next Steps

- Make sure the new method doesn't suffer from Feature Envy because of state it used from its original class. If it does, try splitting the method up and moving part of it back.
- Check the class of the new method to make sure it's not a Large Class.

Inline class

STUB

Inject dependencies

STUB

Replace mixin with composition

STUB

Use convention over configuration

STUB

Introduce Visitor

STUB

Part III

Principles

DRY

STUB

Single responsibility principle

STUB

Tell, Don't Ask

STUB

Law of Demeter

STUB

Composition over inheritance

STUB

Open closed principle

STUB

Dependency inversion principle

STUB