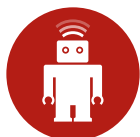


# Ruby Science

The reference for writing fantastic  
Rails applications.



# Ruby Science

thoughtbot

Joe Ferris

Harlow Ward

April 10, 2013

# Contents

<b>Introduction</b>	<b>v</b>
Code Reviews . . . . .	v
Just Follow Your Nose . . . . .	vi
Removing Resistance . . . . .	vii
Bugs and Churn . . . . .	viii
Metrics . . . . .	ix
How To Read This Book . . . . .	ix
<b>I   Code Smells</b>	<b>1</b>
<b>Long Method</b>	<b>2</b>
<b>Large Class</b>	<b>4</b>
God Class . . . . .	7
<b>Feature Envy</b>	<b>8</b>
<b>Case Statement</b>	<b>10</b>
Type Codes . . . . .	10

<i>CONTENTS</i>	ii
-----------------	----

<b>High Fan-out</b>	<b>13</b>
<b>Shotgun Surgery</b>	<b>14</b>
<b>Divergent Change</b>	<b>16</b>
<b>Long Parameter List</b>	<b>19</b>
<b>Duplicated Code</b>	<b>21</b>
<b>Uncommunicative Name</b>	<b>24</b>
<b>Single Table Inheritance (STI)</b>	<b>26</b>
<b>Parallel Inheritance Hierarchies</b>	<b>29</b>
<b>Comments</b>	<b>30</b>
<b>Mixin</b>	<b>32</b>
<b>Callback</b>	<b>35</b>

## **II Solutions 37**

<b>Replace Conditional with Polymorphism</b>	<b>38</b>
Replace Type Code With Subclasses . . . . .	41
Single Table Inheritance (STI) . . . . .	41
Polymorphic Partial . . . . .	45
<b>Replace conditional with Null Object</b>	<b>49</b>
truthiness, try, and other tricks . . . . .	54

<b>Extract method</b>	<b>55</b>
Replace temp with query . . . . .	58
<b>Rename Method</b>	<b>59</b>
<b>Extract Class</b>	<b>63</b>
<b>Extract Value Object</b>	<b>78</b>
<b>Extract Decorator</b>	<b>81</b>
<b>Extract Partial</b>	<b>92</b>
<b>Extract Validator</b>	<b>95</b>
<b>Introduce Explaining Variable</b>	<b>98</b>
<b>Introduce Observer</b>	<b>101</b>
<b>Introduce Form Object</b>	<b>102</b>
<b>Introduce Parameter Object</b>	<b>108</b>
<b>Use class as Factory</b>	<b>111</b>
<b>Move method</b>	<b>116</b>
<b>Inline class</b>	<b>120</b>
<b>Inject dependencies</b>	<b>124</b>
<b>Replace Subclasses with Strategies</b>	<b>132</b>
<b>Replace mixin with composition</b>	<b>154</b>

<b>Replace Callback with Method</b>	<b>160</b>
<b>Use convention over configuration</b>	<b>163</b>
Scoping constantize . . . . .	167
<b>Introduce Visitor</b>	<b>170</b>
 <b>III Principles</b>	 <b>171</b>
<b>DRY</b>	<b>172</b>
Duplicated Knowledge vs Duplicated Text . . . . .	172
<b>Single responsibility principle</b>	<b>175</b>
<b>Tell, Don't Ask</b>	<b>176</b>
<b>Law of Demeter</b>	<b>177</b>
<b>Composition over inheritance</b>	<b>178</b>
<b>Open closed principle</b>	<b>179</b>
<b>Dependency inversion principle</b>	<b>180</b>

# 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 explains the potential problems each smell may reveal and references 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 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.
- [Replace Subclasses with Strategies](#) if the large class is a base class in an inheritance hierarchy.

## 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

```
# 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.
- [Inline Classes](#) if the envied class isn't pulling its weight.

# 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.
- [Use Convention over Configuration](#) when selecting a strategy based on a string name.



# 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 formatting 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.
- [Use Convention over Configuration](#) to eliminate small steps that can be inferred based on a convention such as a name.
- Inline Classes that only serve to add extra steps when performing changes.

# Divergent Change

A class suffers from Divergent Change when it changes for multiple reasons.

## Symptoms

- You can't easily describe what the class does in one sentence.
- The class is changed more frequently than other classes in the application.
- Different changes to the class aren't related to each other.

## Example

```
# app/controllers/summaries_controller.rb
class SummariesController < ApplicationController
  def show
    @survey = Survey.find(params[:survey_id])
    @summaries = @survey.summarize(summarizer)
  end

  private

  def summarizer
    case params[:id]
    when 'breakdown'
      Breakdown.new
    when 'most_recent'
      MostRecent.new
    when 'your_answers'
      UserAnswer.new(current_user)
    else
      raise "Unknown summary type: #{params[:id]}"
    end
  end
end
```

This controller has multiple reasons to change:

- Control flow logic related to summaries, such as authentication.
- Any time a summarizer strategy is added or changed.

## Solutions

- [Extract Class](#) to move one cause of change to a new class.
- [Move Method](#) if the class is changing because of methods that relate to another class.

- [Extract Validator](#) to move validation logic out of models.
- [Introduce Form Object](#) to move form logic out of controllers.
- [Use Convention over Configuration](#) to eliminate changes that can be inferred by a convention such as a name.

## Prevention

You can prevent Divergent Change from occurring by following the [Single Responsibility Principle](#). If a class has only one responsibility, it has only one reason to change.

You can use churn to discover which files are changing most frequently. This isn't a direct relationship, but frequently changed files often have more than one responsibility, and thus more than one reason to change.

# 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

Software is run by computers, but written and read by humans. Names provide important information to developers who are trying to understand a piece of code. Patterns and challenges when naming a method or class can also provide clues for refactoring.

## Symptoms

- Difficulty understanding a method or class.
- Methods or classes with similar names but dissimilar functionality.
- Redundant names, such as names which include the type of object to which they refer.

## Example

In our example application, the `SummariesController` generates summaries from a `Survey`:

```
# app/controllers/summaries_controller.rb
@summaries = @survey.summaries_using(summarizer)
```

The `summarize` method on `Survey` asks each `Question` to summarize itself using a `summarizer`:

```
# app/models/survey.rb
def summaries_using(summarizer)
  questions.map do |question|
    question.summarize(summarizer)
  end
end
```

The `summarize` method on `Question` gets a value by calling `summarize` on a `summarizer`, and then builds a `Summary` using that value.

```
# app/models/question.rb
def summarize(summarizer)
  value = summarizer.summarize(self)
  Summary.new(title, value)
end
```

There are several `summarizer` classes, each of which respond to `summarize`.

If you're lost, don't worry: you're not the only one. The confusing maze of similar names make this example extremely hard to follow.

See [Rename Method](#) to see how we improve the situation.

## Solutions

- [Rename Method](#) if a well-factored method isn't well-named.
- [Extract Class](#) if a class is doing too much to have a meaningful name.
- [Extract Method](#) if a method is doing too much to have a meaningful name.
- [Inline Class](#) if a class is too abstract to have a meaningful name.

# Single Table Inheritance (STI)

Using subclasses is a common method of achieving reuse in object-oriented software. Rails provides a mechanism for storing instances of different classes in the same table, called Single Table Inheritance. Rails will take care of most of the details, writing the class's name to the type column and instantiating the correct class when results come back from the database.

Inheritance has its own pitfalls - see [Composition Over Inheritance](#) - and STI introduces a few new gotchas that may cause you to consider an alternate solution.

## Symptoms

- You need to change from one subclass to another.
- Behavior is shared among some subclasses but not others.
- One subclass is a fusion of one or more other subclasses.

## Example

This method on `Question` changes the question to a new type. Any necessary attributes for the new subclass are provided to the `attributes` method.

```
# app/models/question.rb
def switch_to(type, new_attributes)
  attributes = self.attributes.merge(new_attributes)
  new_question = type.constantize.new(attributes.except('id', 'type'))
  new_question.id = id

  begin
    Question.transaction do
      destroy
      new_question.save!
    end
  rescue ActiveRecord::RecordInvalid
  end

  new_question
end
```

This transition is difficult for a number of reasons:

- You need to worry about common `Question` validations.
- You need to make sure validations for the old subclass are not used.
- You need to make sure validations for the new subclass are used.
- You need to delete data from the old subclass, including associations.
- You need to support data from the new subclass.
- Common attributes need to remain the same.

The implementation achieves all these requirements, but is awkward:

- You can't actually change the class of an instance in Ruby, so you need to return the instance of the new class.
- The implementation requires deleting and creating records, but part of the transaction (destroy) must execute before we can validate the new instance. This results in control flow using exceptions.
- The STI abstraction leaks into the model, because it needs to understand that it has a type column. STI models normally don't need to understand that they're implemented using STI.
- It's hard to understand why this method is implemented the way it is, so other developers fixing bugs or refactoring in the future will have a hard time navigating it.

## Solutions

- If you're using STI to reuse common behavior, use [Replace Subclasses with Strategies](#) to switch to a composition-based model.
- If you're using STI so that you can easily refer to several different classes in the same table, switch to using a polymorphic association instead.

## Prevention

By following [Composition Over Inheritance](#), you'll use STI as a solution less often.



# 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

Inheritance is a common method of reuse in object-oriented software. Ruby supports single inheritance using subclasses and multiple inheritance using mixins. Mixins can be used to package common helpers or provide a common public interface.

However, mixins have some drawbacks:

- They use the same namespace as classes they're mixed into, which can cause naming conflicts.
- Although they have access to instance variables from classes they're mixed into, mixins can't easily accept initializer arguments, so they can't have their own state.
- They inflate the number of methods available in a class.
- They're not easy to add and remove at runtime.
- They're difficult to test in isolation, since they can't be instantiated.

## Symptoms

- Methods in mixins that accept the same parameters over and over.
- Methods in mixins that don't reference the state of the class they're mixed into.
- Business logic that can't be used without using the mixin.
- Classes which have few public methods except those from a mixin.

## Example

In our example application, users can invite their friends by email to take surveys. If an invited email matches an existing user, a private message will be created. Otherwise, a message is sent to that email address with a link.

The logic to generate the invitation message is the same regardless of the delivery mechanism, so this behavior is encapsulated in a mixin:

```
# app/models/inviter.rb
module Inviter
  extend ActiveSupport::Concern

  included do
    include ActionController::Rendering
    include Rails.application.routes.url_helpers

    self.view_paths = 'app/views'
    self.default_url_options = ActionMailer::Base.default_url_options
  end

  private

  def render_message_body
    render template: 'invitations/message'
  end
end
```

Each delivery strategy mixes in `Inviter` and calls `render_message_body`:

```
# app/models/message_inviter.rb
class MessageInviter < ActionController::Base
  include Inviter

  def initialize(invitation, recipient)
    @invitation = invitation
    @recipient = recipient
  end

  def deliver
    Message.create!(
      recipient: @recipient,
      sender: @invitation.sender,
      body: render_message_body
    )
  end
end
```

Although the mixin does a good job of preventing [duplicated code](#), it's difficult to test or understand in isolation, it obfuscates the inviter classes, and it tightly couples the inviter classes to a particular message body implementation.

## Solutions

- [Extract Class](#) to liberate business logic trapped in mixins.
- [Replace Mixin with Composition](#) to improve testability, flexibility, and readability.

## Prevention

Mixins are a form of inheritance. By following [Composition Over Inheritance](#), you'll be less likely to introduce mixins.

Reserve mixins for reusable framework code like common associations and callbacks, and you'll end up with a more flexible and comprehensible system.

# Callback

Callbacks are a convenient way to decorate the default `save` method with custom persistence logic, without the drudgery of template methods, overriding, or calling `super`.

However, callbacks are frequently abused by adding non-persistence logic to the persistence life cycle, such as sending emails or processing payments. Models riddled with callbacks are harder to refactor and prone to bugs, such as accidentally sending emails or performing external changes before a database transaction is committed.

## Symptoms

- Callbacks which contain business logic such as processing payments.
- Attributes which allow certain callbacks to be skipped.
- Methods such as `save_without_sending_email` which skip callbacks.
- Callbacks which need to be invoked conditionally.

## Example

```
# app/models/survey_inviter.rb
def deliver_invitations
  recipients.map do |recipient_email|
    Invitation.create!(
      survey: survey,
      sender: sender,
      recipient_email: recipient_email,
      status: 'pending',
      message: @message
    )
  end
end

# app/models/invitation.rb
after_create :deliver

# app/models/invitation.rb
def deliver
  Mailer.invitation_notification(self).deliver
end
```

In the above code, the SurveyInviter is simply creating Invitation records, and the actual delivery of the invitation email is hidden behind Invitation.create! via a callback.

If one of several invitations fails to save, the user will see a 500 page, but some of the invitations will already have been saved and delivered. The user will be unable to tell which invitations were sent.

Because delivery is coupled with persistence, there's no way to make sure that all of the invitations are saved before starting to deliver emails.

## Solutions

- [Replace Callback with Method](#) if the callback logic is unrelated to persistence.



# **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 ActiveRecord::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
end
```

```
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 “MultipleChoice,” 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 `c18ebec` 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 refactoring.

## 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 each 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 each conditional path into a new class.

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 Partial

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.

Also, using STI has specific disadvantages. See the [chapter on STI](#) for details.

## 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 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`.

## 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 ActiveRecord::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 `NullAnswers` and real `Answers`. 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.

## Example

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](#). 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.

# Rename Method

Renaming a method allows developers to improve the language of the domain as their understanding naturally evolves during development.

The process is straightforward if there aren't too many references:

- Choose a new name for the method. This is the hard part!
- Change the method definition to the new name.
- Find and replace all references to the old name.

If there are a large number of references to the method you want to rename, you can rename the callers one at a time while keeping everything in working order. The process is mostly the same:

- Choose a new name for the method. This is the hard part!
- Give the method its new name.
- Add an alias to keep the old name working.
- Find and replace all references to the old name.
- Remove the alias.

## Uses

- Eliminate [Uncommunicative Names](#).
- Change method names to conform to common interfaces.

## Example

In our example application, we generate summaries from answers to surveys. We allow more than one type of summary, so strategies are employed to handle the variations. There are a number of methods and dependencies that make this work.

SummariesController#show depends on Survey#summarize:

```
# app/controllers/summaries_controller.rb
@summaries = @survey.summarize(summarizer)
```

Survey#summarize depends on Question#summarize:

```
# app/models/survey.rb
def summarize(summarizer)
  questions.map do |question|
    question.summarize(summarizer)
  end
end
```

Question#summarize depends on summarize from its summarizer argument (a strategy):

```
# app/models/question.rb
def summarize(summarizer)
  value = summarizer.summarize(self)
  Summary.new(title, value)
end
```

There are several summarizer classes, each of which respond to summarize.

This is confusing, largely because the word `summarize` is used to mean several different things:

- Survey#summarize accepts a summarizer and returns an array of Summary instances.



- `Question#summarize` accepts a summarizer and returns a single `Summary` instance.
- `summarize` on summarizer strategies accepts a `Question` and returns a `String`.

Let's rename these methods so that each name is used uniquely and consistently in terms of what it accepts, what it returns, and what it does.

First, we'll rename `Survey#summarize` to reflect the fact that it returns a collection.

```
# app/models/survey.rb
def summaries_using(summarizer)
```

Then, we'll update the only reference to the old method:

```
# app/controllers/summaries_controller.rb
@summaries = @survey.summaries_using(summarizer)
```

Next, we'll rename `Question#summarize` to be consistent with the naming introduced in `Survey`:

```
# app/models/question.rb
def summary_using(summarizer)
```

Finally, we'll update the only reference in `Survey#summaries_using`:

```
# app/models/survey.rb
question.summary_using(summarizer)
```

We now have consistent and clearer naming:

- `summarize` means taking a question and returning a string value representing its answers.
- `summary_using` means taking a summarizer and using it to build a `Summary`.
- `summaries_using` means taking a set of questions and building a `Summary` for each one.

## Next Steps

- Check for explanatory comments that are no longer necessary now that the code is clearer.
- If the new name for a method is long, see if you can [extract methods](#) from it make it smaller.

# Extract Class

Dividing responsibilities into classes is the primary way to manage complexity in object-oriented software. Extract Class is the primary mechanism for introducing new classes. This refactoring takes one class and splits it into two by moving one or more methods and instance variables into a new class.

The process for extracting a class looks like this:

1. Create a new, empty class.
2. Instantiate the new class from the original class.
3. [Move a method](#) from the original class to the new class.
4. Repeat step 3 until you're happy with the original class.

## Uses

- Removes [Large Class](#) by splitting up the class.
- Eliminates [Divergent Change](#) by moving one reason to change into a new class.
- Provides a cohesive set of functionality with a meaningful name, making it easier to understand and talk about.
- Fully encapsulates a concern within a single class, following the [Single Responsibility Principle](#) and making it easier to change and reuse that functionality.

## Example

The `InvitationsController` is a [Large Class](#) hidden behind a [Long Method](#):

```
# 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])

    @recipients = params[:invitation][:recipients]
    recipient_list = @recipients.gsub(/\s+/, ' ').split(/\n,;]+/)

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

    @message = params[:invitation][:message]

    if @invalid_recipients.empty? && @message.present?
      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
      render 'new'
    end
  end
end
end

```

Although it contains only two methods, there's a lot going on under the hood. It parses and validates emails, manages several pieces of state which the view needs to know about, handles control flow for the user, and creates and delivers invitations.

A liberal application of [Extract Method](#) to break up this [Long Method](#) will reveal the complexity:

```

# 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
    end
  end
end

```

```

        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
end

```

Let's extract all of the non-controller logic into a new class. We'll start by defining and instantiating a new, empty class:

```
# app/controllers/invitations_controller.rb
@survey_inviter = SurveyInviter.new
```

```
# app/models/survey_inviter.rb
class SurveyInviter
end
```

At this point, we've [created a staging area](#) for using [Move Method](#) to transfer complexity from one class to the other.

Next, we'll move one method from the controller to our new class. It's best to move methods which depend on few private methods or instance variables from the original class, so we'll start with a method which only uses one private method:

```
# app/models/survey_inviter.rb
def recipient_list
  @recipient_list ||= @recipients.gsub(/\s+/, ' ').split(/\n,;]+/)
end
```

We need the recipients for this method, so we'll accept it in the initialize method:

```
# app/models/survey_inviter.rb
def initialize(recipients)
  @recipients = recipients
end
```

And pass it from our controller:

```
# app/controllers/invitations_controller.rb
@survey_inviter = SurveyInviter.new(recipients)
```

The original controller method can delegate to the extracted method:

```
# app/controllers/invitations_controller.rb
def recipient_list
  @survey_inviter.recipient_list
end
```

We've moved a little complexity out of our controller, and we now have a repeatable process for doing so: we can continue to move methods out until we feel good about what's left in the controller.

Next, let's move out `invalid_recipients` from the controller, since it depends on `recipient_list`, which we already moved:

```
# app/models/survey_inviter.rb
def invalid_recipients
  @invalid_recipients ||= recipient_list.map do |item|
    unless item.match(EMAIL_REGEX)
      item
    end
  end.compact
end
```

Again, the original controller method can delegate:

```
# app/controllers/invitations_controller.rb
def invalid_recipients
  @survey_inviter.invalid_recipients
end
```

This method references a constant from the controller. This was the only place where the constant was used, so we can move it to our new class:

```
# app/models/survey_inviter.rb
EMAIL_REGEX = /\A([^\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\z/
```

We can remove an instance variable in the controller by invoking this method directly in the view:



```
# app/views/invitations/new.html.erb
<% if @survey_inviter.invalid_recipients %>
  <div class="error">
    Invalid email addresses:
    <%= @survey_inviter.invalid_recipients.join(', ') %>
  </div>
<% end %>
```

Now that parsing email lists is [moved out of our controller](#), let's extract and delegate the only method in the controller which depends on `invalid_recipients`:

```
# app/models/survey_inviter.rb
def valid_recipients?
  invalid_recipients.empty?
end
```

Now we can remove `invalid_recipients` from the controller entirely.

The `valid_recipients?` method is only used in the compound validation condition:

```
# app/controllers/invitations_controller.rb
if valid_recipients? && valid_message?
```

If we extract `valid_message?` as well, we can fully encapsulate validation within `SurveyInviter`.

```
# app/models/survey_inviter.rb
def valid_message?
  @message.present?
end
```

We need `message` for this method, so we'll add that to `initialize`:

```
# app/models/survey_inviter.rb
def initialize(message, recipients)
  @message = message
  @recipients = recipients
end
```

And pass it in:

```
# app/controllers/invitations_controller.rb
@survey_inviter = SurveyInviter.new(message, recipients)
```

We can now extract a method to encapsulate this compound condition:

```
# app/models/survey_inviter.rb
def valid?
  valid_message? && valid_recipients?
end
```

And use that new method in our controller:

```
# app/controllers/invitations_controller.rb
if @survey_inviter.valid?
```

Now these methods can be private, trimming down the public interface for SurveyInviter:

```
# app/models/survey_inviter.rb
private

def valid_message?
  @message.present?
end

def valid_recipients?
  invalid_recipients.empty?
end
```

We've [pulled out most of the private methods](#), so the remaining complexity is largely from saving and delivering the invitations.

Let's extract and move a `deliver` method for that:

```
# app/models/survey_inviter.rb
def deliver
  recipient_list.each do |email|
    invitation = Invitation.create(
      survey: @survey,
      sender: @sender,
      recipient_email: email,
      status: 'pending'
    )
    Mailer.invitation_notification(invitation, @message)
  end
end
```

We need the sender (the currently signed in user) as well as the survey from the controller to do this. This pushes our `initialize` method up to four parameters, so let's switch to a hash:

```
# app/models/survey_inviter.rb
def initialize(attributes = {})
  @survey = attributes[:survey]
  @message = attributes[:message] || ''
  @recipients = attributes[:recipients] || ''
  @sender = attributes[:sender]
end
```

And extract a method in our controller to build it:

```
# app/controllers/invitations_controller.rb
def survey_inviter_attributes
  params[:invitation].merge(survey: @survey, sender: current_user)
end
```

Now we can invoke this method in our controller:

```
# app/controllers/invitations_controller.rb
if @survey_inviter.valid?
  @survey_inviter.deliver
  redirect_to survey_path(@survey), notice: 'Invitation successfully sent'
else
  @recipients = recipients
  @message = message
  render 'new'
end
```

The `recipient_list` method is now only used internally in `SurveyInviter`, so let's make it private.

We've [moved most of the behavior out of the controller](#), but we're still assigning a number of instance variables for the view, which have corresponding private methods in the controller. These values are also available on `SurveyInviter`, which is already assigned to the view, so let's expose those using `attr_reader`:

```
# app/models/survey_inviter.rb
attr_reader :message, :recipients, :survey
```

And use them directly from the view:

```
# app/views/invitations/new.html.erb
<%= simple_form_for(
  :invitation,
  url: survey_invitations_path(@survey_inviter.survey)
) do |f| %>
  <%= f.input(
    :message,
    as: :text,
    input_html: { value: @survey_inviter.message }
  ) %>
  <% if @invalid_message %>
    <div class="error">Please provide a message</div>
  <% end %>
  <%= f.input(
    :recipients,
    as: :text,
    input_html: { value: @survey_inviter.recipients }
  ) %>
```

Only the SurveyInviter is used in the controller now, so we can [remove the remaining instance variables and private methods](#).

Our controller is now much simpler:

```
# app/controllers/invitations_controller.rb
class InvitationsController < ApplicationController
  def new
    @survey_inviter = SurveyInviter.new(survey: survey)
  end

  def create
    @survey_inviter = SurveyInviter.new(survey_inviter_attributes)
    if @survey_inviter.valid?
      @survey_inviter.deliver
      redirect_to survey_path(survey), notice: 'Invitation successfully sent'
    else
      render 'new'
    end
  end

  private

  def survey_inviter_attributes
    params[:invitation].merge(survey: survey, sender: current_user)
  end

  def survey
    Survey.find(params[:survey_id])
  end
end
```

It only assigns one instance variable, it doesn't have too many methods, and all of its methods are fairly small.

The newly extracted `SurveyInviter` class absorbed much of the complexity, but still isn't as bad as the original controller:

```
# app/models/survey_inviter.rb
class SurveyInviter
  EMAIL_REGEX = /\A([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\z/

  def initialize(attributes = {})
    @survey = attributes[:survey]
    @message = attributes[:message] || ''
    @recipients = attributes[:recipients] || ''
    @sender = attributes[:sender]
  end

  attr_reader :message, :recipients, :survey

  def valid?
    valid_message? && valid_recipients?
  end

  def deliver
    recipient_list.each do |email|
      invitation = Invitation.create(
        survey: @survey,
        sender: @sender,
        recipient_email: email,
        status: 'pending'
      )
      Mailer.invitation_notification(invitation, @message)
    end
  end

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

```
end

private

def valid_message?
  @message.present?
end

def valid_recipients?
  invalid_recipients.empty?
end

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

We can take this further by extracting more classes from SurveyInviter. See our [full solution on GitHub](#).

## Drawbacks

Extracting classes decreases the amount of complexity in each class, but increases the overall complexity of the application. Extracting too many classes will create a maze of indirection which developers will be unable to navigate.

Every class also requires a name. Introducing new names can help to explain functionality at a higher level and facilitates communication between developers. However, introducing too many names results in vocabulary overload, which makes the system difficult to learn for new developers.

Extract classes in response to pain and resistance, and you'll end up with just the right number of classes and names.

## Next Steps

- Check the newly extracted class to make sure it isn't a [Large Class](#), and extract another class if it is.



- Check the original class for [Feature Envy](#) of the extracted class, and use [Move Method](#) if necessary.

# Extract Value Object

Value Objects are objects that represent a value (such as a dollar amount) rather than a unique, identifiable entity (such as a particular user).

Value Objects often implement information derived from a primitive object, such as the dollars and cents from a float, or the user name and domain from an email string.

## Uses

- Remove [Duplicated Code](#) from making the same observations of primitive objects throughout the code base.
- Remove [Large Classes](#) by splitting out query methods associated with a particular variable.
- Make the code easier to understand by fully-encapsulating related logic into a single class, following the [Single Responsibility Principle](#).
- Eliminate [Divergent Change](#) by extracting code related to an embedded semantic type.

## Example

InvitationsController is bloated with methods and logic relating to parsing a string that contains a list of email addresses:

```
# app/controllers/invitations_controller.rb
def recipient_list
  @recipient_list ||= recipients.gsub(/\s+/, '').split(/\n,;]+/)
end

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

We can [extract a new class](#) to offload this responsibility:

```
# app/models/recipient_list.rb
class RecipientList
  include Enumerable

  def initialize(recipient_string)
    @recipient_string = recipient_string
  end

  def each(&block)
    recipients.each(&block)
  end

  def to_s
    @recipient_string
  end

  private

  def recipients
    @recipient_string.to_s.gsub(/\s+/, '').split(/\n,;]+/)
  end
end

# app/controllers/invitations_controller.rb
def recipient_list
  @recipient_list ||= RecipientList.new(params[:invitation][:recipients])
end
```

**Next Steps**

- Search the application for [Duplicated Code](#) related to the newly extracted class.
- Value Objects should be Immutable. Make sure the extracted class doesn't have any writer methods.

# Extract Decorator

Decorators can be used to lay new concerns on top of existing objects without modifying existing classes. They combine best with small classes with few methods, and make the most sense when modifying the behavior of existing methods, rather than adding new methods.

The steps for extracting a decorator vary depending on the initial state, but they often include the following:

1. Extract a new decorator class, starting with the alternative behavior.
2. Compose the decorator in the original class.
3. Move state specific to the alternate behavior into the decorator.
4. Invert control, applying the decorator to the original class from its container, rather than composing the decorator from the original class.

## Uses

- Eliminate [Large Classes](#) by extracting concerns.
- Eliminate [Divergent Change](#) and follow the [Open Closed Principle](#) by making it easier to modify behavior without modifying existing classes.
- Prevent conditional logic from leaking by making decisions earlier.

## Example

In our example application, users can view a summary of the answers to each question on a survey. In order to prevent the summary from influencing a user's own answers, users don't see summaries for questions they haven't answered yet by default. Users can click a link to override this decision and view the summary for every question. This concern is mixed across several levels, and introducing the change affected several classes. Let's see if we can refactor our application to make similar changes easier in the future.

Currently, the controller determines whether or not unanswered questions should display summaries:

```
# app/controllers/summaries_controller.rb
def constraints
  if include_unanswered?
    {}
  else
    { answered_by: current_user }
  end
end

def include_unanswered?
  params[:unanswered]
end
```

It passes this decision into `Survey#summaries_using` as a hash containing boolean flag:

```
# app/controllers/summaries_controller.rb
@summaries = @survey.summaries_using(summarizer, constraints)
```

Survey#summaries\_using uses this information to decide whether each question should return a real summary or a hidden summary:

```
# app/models/survey.rb
def summaries_using(summarizer, options = {})
  questions.map do |question|
    if !options[:answered_by] || question.answered_by?(options[:answered_by])
      question.summary_using(summarizer)
    else
      Summary.new(question.title, NO_ANSWER)
    end
  end
end
```

This method is pretty dense. We can start by using [Extract Method](#) to clarify and reveal complexity:

```
# app/models/survey.rb
def summaries_using(summarizer, options = {})
  questions.map do |question|
    summary_or_hidden_answer(summarizer, question, options[:answered_by])
  end
end

private

def summary_or_hidden_answer(summarizer, question, answered_by)
  if hide_unanswered_question?(question, answered_by)
    hide_answer_to_question(question)
  else
    question.summary_using(summarizer)
  end
end

def hide_unanswered_question?(question, answered_by)
  answered_by && !question.answered_by?(answered_by)
end
```

```
def hide_answer_to_question(question)
  Summary.new(question.title, NO_ANSWER)
end
```

The `summary_or_hidden_answer` method reveals a pattern that's well-captured by using a Decorator:

- There's a base case: returning the real summary for the question's answers.
- There's an alternate, or decorated, case: returning a summary with a hidden answer.
- The conditional logic for using the base or decorated case is unrelated to the base case: `answered_by` is only used for determining which path to take, and isn't used by to generate summaries.

As a Rails developer, this may seem familiar to you: many pieces of Rack middleware follow a similar approach.

Now that we've recognized this pattern, let's refactor to use a Decorator.

### Move decorated case to decorator

Let's start by creating an empty class for the decorator and [moving one method](#) into it:

```
# app/models/unanswered_question_hider.rb
class UnansweredQuestionHider
  NO_ANSWER = "You haven't answered this question".freeze

  def hide_answer_to_question(question)
    Summary.new(question.title, NO_ANSWER)
  end
end
```



The method references a constant from Survey, so moved that, too.

Now we update Survey to compose our new class:

```
# app/models/survey.rb
def summary_or_hidden_answer(summarizer, question, answered_by)
  if hide_unanswered_question?(question, answered_by)
    UnansweredQuestionHider.new.hide_answer_to_question(question)
  else
    question.summary_using(summarizer)
  end
end
```

At this point, the [decorated path is contained within the decorator](#).

### Move conditional logic into decorator

Next, we can move the conditional logic into the decorator. We've already extracted this to its own method on Survey, so we can simply move this method over:

```
# app/models/unanswered_question_hider.rb
def hide_unanswered_question?(question, user)
  user && !question.answered_by?(user)
end
```

Note that the `answered_by` parameter was renamed to `user`. That's because, now that the context is more specific, it's clear what role the user is playing.

```
# app/models/survey.rb
def summary_or_hidden_answer(summarizer, question, answered_by)
  hider = UnansweredQuestionHider.new
  if hider.hide_unanswered_question?(question, answered_by)
    hider.hide_answer_to_question(question)
  else
    question.summary_using(summarizer)
  end
end
```

**Move body into decorator**

There's just one summary-related method left in `Survey`: `summary_or_hidden_answer`. Let's move this into the decorator:

```
# app/models/unanswered_question_hider.rb
def summary_or_hidden_answer(summarizer, question, user)
  if hide_unanswered_question?(question, user)
    hide_answer_to_question(question)
  else
    question.summary_using(summarizer)
  end
end

# app/models/survey.rb
def summaries_using(summarizer, options = {})
  questions.map do |question|
    UnansweredQuestionHider.new.summary_or_hidden_answer(
      summarizer,
      question,
      options[:answered_by]
    )
  end
end
```

At this point, every other method in the decorator can be made private.

### Promote parameters to instance variables

Now that we have a class to handle this logic, we can move some of the parameters into instance state. In `Survey#summaries_using`, we use the same summarizer and user instance; only the question varies as we iterate through questions to summarize. Let's move everything but question into instance variables on the decorator:

```
# app/models/unanswered_question_hider.rb
def initialize(summarizer, user)
  @summarizer = summarizer
  @user = user
end

def summary_or_hidden_answer(question)
  if hide_unanswered_question?(question)
    hide_answer_to_question(question)
  else
    question.summary_using(@summarizer)
  end
end

# app/models/survey.rb
def summaries_using(summarizer, options = {})
  questions.map do |question|
    UnansweredQuestionHider.new(summarizer, options[:answered_by]).
      summary_or_hidden_answer(question)
  end
end
```

Our decorator now just needs a question to generate a Summary.

### Change decorator to follow component interface

In the end, the component we want to wrap with our decorator is the summarizer, so we want the decorator to obey the same interface as its component,

the summarizer. Let's rename our only public method so that it follows the summarizer interface:

```
# app/models/unanswered_question_hider.rb
def summarize(question)

# app/models/survey.rb
UnansweredQuestionHider.new(summarizer, options[:answered_by]).
  summarize(question)
```

Our decorator now follows the component interface in name, but not behavior. In our application, summarizers return a string which represents the answers to a question, but our decorator is returning a Summary instead. Let's fix our decorator to follow the component interface by returning just a string:

```
# app/models/unanswered_question_hider.rb
def summarize(question)
  if hide_unanswered_question?(question)
    hide_answer_to_question(question)
  else
    @summarizer.summarize(question)
  end
end

# app/models/unanswered_question_hider.rb
def hide_answer_to_question(question)
  NO_ANSWER
end

# app/models/survey.rb
def summaries_using(summarizer, options = {})
  questions.map do |question|
    hider = UnansweredQuestionHider.new(summarizer, options[:answered_by])
    question.summary_using(hider)
  end
end
```

Our decorator now [follows the component interface](#).

That last method on the decorator (`hide_answer_to_question`) isn't pulling its weight anymore: it just returns the value from a constant. Let's [inline it](#) to slim down our class a bit:

```
# app/models/unanswered_question_hider.rb
def summarize(question)
  if hide_unanswered_question?(question)
    NO_ANSWER
  else
    @summarizer.summarize(question)
  end
end
```

Now we have a decorator that can wrap any summarizer, nicely-factored and ready to use.

### Invert control

Now comes one of the most important steps: we can [invert control](#) by removing any reference to the decorator from `Survey` and passing in an already-decorated summarizer.

The `summaries_using` method is simplified:

```
# app/models/survey.rb
def summaries_using(summarizer)
  questions.map do |question|
    question.summary_using(summarizer)
  end
end
```

Instead of passing the boolean flag down from the controller, we can [make the decision to decorate there](#) and pass a decorated or undecorated summarizer:

```
# app/controllers/summaries_controller.rb
def show
  @survey = Survey.find(params[:survey_id])
  @summaries = @survey.summaries_using(decorated_summarizer)
end

private

def decorated_summarizer
  if include_unanswered?
    summarizer
  else
    UnansweredQuestionHider.new(summarizer, current_user)
  end
end
```

This isolates the decision to one class and keeps the result of the decision close to the class that makes it.

Another important effect of this refactoring is that the Survey class is now reverted back to [the way it was before we started hiding unanswered question summaries](#). This means that we can now add similar changes without modifying Survey at all.

## Drawbacks

- Decorators must keep up-to-date with their component interface. Our decorator follows the summarizer interface. Every decorator we add for this interface is one more class that will need to change any time we change the interface.
- We removed a concern from Survey by hiding it behind a decorator, but this may make it harder for a developer to understand how a Survey might return the hidden response text, as that text doesn't appear anywhere in that class.
- The component we decorated had the smallest possible interface: one public method. Classes with more public methods are more difficult to decorate.

- Decorators can modify methods in the component interface easily, but adding new methods won't work with multiple decorators without metaprogramming like `method_missing`. These constructs are harder to follow and should be used with care.

## Next Steps

- It's unlikely that your automated test suite has enough coverage to check every component implementation with every decorator. Run through the application in a browser after introducing new decorators. Test and fix any issues you run into.
- Make sure that inverting control didn't push anything over the line into a [Large Class](#).

# 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 (`_file-name.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 occurrences of the duplicated view code in your application and replace them with the newly extracted partial.

# Extract Validator

A form of [Extract Class](#) used to remove complex validation details from ActiveRecord models. This technique also prevents duplication of validation code across several files.

## Uses

- Keep validation implementation details out of models.
- Encapsulate validation details into a single file.
- Remove duplication among classes performing the same validation logic.

## Example

The `Invitation` class has validation details in-line. It checks that the `recipient_email` matches the formatting of the regular expression `EMAIL_REGEX`.

```
# app/models/invitation.rb
class Invitation < ActiveRecord::Base
  EMAIL_REGEX = /\A([^\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\z/i
  validates :recipient_email, presence: true, format: EMAIL_REGEX
end
```

We extract the validation details into a new class `EmailValidator`, and place the new class into the `app/validators` directory.

```
# app/validators/email_validator.rb
class EmailValidator < ActiveRecord::EachValidator
  EMAIL_REGEX = /\A([^\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\z/i
  def validate_each(record, attribute, value)
    unless value.match EMAIL_REGEX
      record.errors.add(attribute, "#{value} is not a valid email")
    end
  end
end
```

Once the validator has been extracted. Rails has a convention for using the new validation class. `EmailValidator` is used by setting `email: true` in the validation arguments.

```
# app/models/invitation.rb
class Invitation < ActiveRecord::Base
  validates :recipient_email, presence: true, email: true
end
```

The convention is to use the validation class name (in lowercase, and removing `Validator` from the name). For example, if we were validating an attribute with `ZipCodeValidator` we'd set `zip_code: true` as an argument to the validation call.

When validating an array of data as we do in `SurveyInviter`, we use the `EnumerableValidator` to loop over the contents of an array.

```
# app/models/survey_inviter.rb
validates_with EnumerableValidator,
  attributes: [:recipients],
  unless: 'recipients.nil?',
  validator: EmailValidator
```

The `EmailValidator` is passed in as an argument, and each element in the array is validated against it.

```
# app/validators/enumerable_validator.rb
class EnumerableValidator < ActiveModel::EachValidator
  def validate_each(record, attribute, enumerable)
    enumerable.each do |value|
      validator.validate_each(record, attribute, value)
    end
  end

  private

  def validator
    options[:validator].new(validator_options)
  end

  def validator_options
    options.except(:validator).merge(attributes: attributes)
  end
end
```

## Next Steps

- Verify the extracted validator does not have any Long Methods.
- Check for other models that could use the validator.

# 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.

```
# 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
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 ActiveRecord::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
end
```

```

    )
  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.



## Example

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

An Abstract Factory is an object that knows how to build something, such as one of several possible strategies for summarizing answers to questions on a survey. An object that holds a reference to an abstract factory doesn't need to know what class is going to be used; it trusts the factory to return an object that responds to the required interface.

Because classes are objects in Ruby, every class can act as an Abstract Factory. Using a class as a factory allows us to remove most explicit factory objects.

## Uses

- Removes [Duplicated Code](#), [Shotgun Surgery](#), and Parallel Inheritance Hierarchies by cutting out crufty factory classes.
- Combines with Convention Over Configuration to eliminate [Shotgun Surgery](#) and [Case Statements](#).

## Example

This controller uses one of several possible summarizer strategies to generate a summary of answers to the questions on a survey:

```
# app/controllers/summaries_controller.rb
class SummariesController < ApplicationController
  def show
    @survey = Survey.find(params[:survey_id])
    @summaries = @survey.summarize(summarizer)
  end

  private

  def summarizer
    case params[:id]
    when 'breakdown'
      Breakdown.new
    when 'most_recent'
      MostRecent.new
    when 'your_answers'
      UserAnswer.new(current_user)
    else
      raise "Unknown summary type: #{params[:id]}"
    end
  end
end
```

The summarizer method is a Factory Method. It returns a summarizer object based on `params[:id]`.

We can refactor that using the Abstract Factory pattern:

```
def summarizer
  summarizer_factory.build
end

def summarizer_factory
  case params[:id]
  when 'breakdown'
    BreakdownFactory.new
  when 'most_recent'
    MostRecentFactory.new
  when 'your_answers'
    UserAnswerFactory.new(current_user)
  else
    raise "Unknown summary type: #{params[:id]}"
  end
end
```

Now the summarizer method asks the summarizer\_factory method for an Abstract Factory, and it asks the factory to build the actual summarizer instance.

However, this means we need to provide an Abstract Factory for each summarizer strategy:

```
class BreakdownFactory
  def build
    Breakdown.new
  end
end

class MostRecentFactory
  def build
    MostRecent.new
  end
end
```

```
class UserAnswerFactory
  def initialize(user)
    @user = user
  end

  def build
    UserAnswer.new(@user)
  end
end
```

These factory classes are repetitive and don't pull their weight. We can rip two of these classes out by using the actual summarizer class as the factory instance. First, let's rename the `build` method to `new` to follow the Ruby convention:

```
def summarizer
  summarizer_factory.new
end

class BreakdownFactory
  def new
    Breakdown.new
  end
end

class MostRecentFactory
  def new
    MostRecent.new
  end
end
```

```
class UserAnswerFactory
  def initialize(user)
    @user = user
  end

  def new
    UserAnswer.new(@user)
  end
end
```

Now an instance of BreakdownFactory acts exactly like the Breakdown class itself, and the same is true of MostRecentFactory and MostRecent. Therefore, let's use the classes themselves instead of instances of the factory classes:

```
def summarizer_factory
  case params[:id]
  when 'breakdown'
    Breakdown
  when 'most_recent'
    MostRecent
  when 'your_answers'
    UserAnswerFactory.new(current_user)
  else
    raise "Unknown summary type: #{params[:id]}"
  end
end
```

Now we can delete two of our factory classes.

## Next Steps

- [Use Convention Over Configuration](#) to remove manual mappings and possibly remove more classes.

# 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

As an application evolves, new classes are introduced as new features are added and existing code is refactored. [Extracting classes](#) will help to keep existing classes maintainable and make it easier to add new features. However, features can also be removed or simplified, and you'll inevitably find that some classes just aren't pulling their weight. Removing dead-weight classes is just as important as splitting up [large classes](#), and inlining a class is the easiest way to remove it.

Inlining a class is straightforward:

- For each consumer class that uses the inlined class, inline or move each method from the inlined class into the consumer class.
- Remove the inlined class.

Note that this refactoring is difficult (and unwise!) if you have more than one or two consumer classes.

## Uses

- Make classes easier to understand by eliminating the number of methods, classes, and files developers need to look through.
- Eliminate [Shotgun Surgery](#) from changes that cascade through useless classes.

- Eliminate [Feature Envy](#) when the envied class can be inlined into the envious class.
- Eliminate [High Fanout](#) by inlining dependencies.

## Example

In our example application, users can create surveys and invite other users to answer them. Users are invited by listing email addresses to invite.

Any email addresses that match up with existing users are sent using a private message that the user will see the next time he or she signs in. Invitations to unrecognized addresses are sent using email messages.

The Invitation model delegates to a different strategy class based on whether or not its recipient email is recognized as an existing user:

```
# app/models/invitation.rb
def deliver
  if recipient_user
    MessageInviter.new(self, recipient_user).deliver
  else
    EmailInviter.new(self).deliver
  end
end
```

We've decided that the private messaging feature isn't getting enough use, so we're going to remove it. This means that all invitations will now be delivered via email, so we can simplify Invitation#deliver to always use the same strategy:

```
# app/models/invitation.rb
def deliver
  EmailInviter.new(self).deliver
end
```

The EmailInviter class was useful as a strategy, but now that the strategy no longer varies, it doesn't bring much to the table:

```
# app/models/email_inviter.rb
class EmailInviter
  def initialize(invitation)
    @invitation = invitation
    @body = InvitationMessage.new(@invitation).body
  end

  def deliver
    Mailer.invitation_notification(@invitation, @body).deliver
  end
end
```

It doesn't handle any concerns that aren't already well-encapsulated by `InvitationMessage` and `Mailer`, and it's only used once (in `Invitation`). We can inline this class into `Invitation` and drop a little overall complexity and indirection from our application.

First, let's inline the `EmailInviter#deliver` method (and its dependent variables from `EmailInviter#initialize`):

```
# app/models/invitation.rb
def deliver
  body = InvitationMessage.new(self).body
  Mailer.invitation_notification(self, body).deliver
end
```

Next, we can delete `EmailInviter` entirely.

After inlining the class, it requires fewer jumps through methods, classes, and files to understand how invitations are delivered. Additionally, the application is less complex overall. Flog gives us a total complexity score of 424.7 after this refactoring, down slightly from 427.6. This isn't a huge gain, but this was an easy refactoring, and continually deleting or inlining unnecessary classes and methods will have larger long term effects.

## Drawbacks

- Attempting to inline a class with multiple consumers will likely introduce duplicated code.

- Inlining a class may create [large classes](#) and cause [divergent change](#).
- Inlining a class will usually increase per-class or per-method complexity, even if it reduces total complexity.

## **Next Steps**

- Use [Extract Method](#) if any inlined methods introduced [long methods](#).
- Use [Extract Class](#) if the merged class is a [large class](#) or beings suffering from [divergent change](#).

# Inject dependencies

Injecting dependencies allows you to keep dependency resolutions close to the logic that affects them. It can prevent sub-dependencies from leaking throughout the code base, and it makes it easier to change the behavior of related components [without modifying those components' classes](#).

Although many people think of dependency injection frameworks and XML when they hear “dependency injection,” injecting a dependency is usually as simple as passing it as a parameter.

Changing code to use dependency injection only takes a few steps:

1. Move the dependency decision to a higher level component.
2. Pass the dependency as a parameter to the lower level component.
3. Remove any sub-dependencies from the lower level component.

Injecting dependencies is the simplest way to [invert control](#).

## Uses

- Eliminates [Shotgun Surgery](#) from leaking sub-dependencies.
- Eliminates [Divergent Change](#) by allowing runtime composition patterns, such as [decorators](#) and strategies.



## Example

In our example applications, users can view a summary of the answers to each question on a survey. Users can select from one of several different summary types to view. For example, they can see the most recent answer to each question, or they can see a percentage breakdown of the answers to a multiple choice question.

The controller passes in the name of the summarizer that the user selected:

```
# app/controllers/summaries_controller.rb
def show
  @survey = Survey.find(params[:survey_id])
  @summaries = @survey.summaries_using(summarizer, options)
end

private

def summarizer
  params[:id]
end
```

Survey#summaries\_using asks each of its questions for a summary using that summarizer and the given options:

```
# app/models/survey.rb
question.summary_using(summarizer, options)
```

Question#summary\_using instantiates the requested summarizer with the requested options, and then asks the summarizer to summarize the question:

```
# app/models/question.rb
def summary_using(summarizer_name, options)
  summarizer_factory = "Summarizer::#{summarizer_name.classify}".constantize
  summarizer = summarizer_factory.new(options)
  value = summarizer.summarize(self)
  Summary.new(title, value)
end
```

This is hard to follow and causes [shotgun surgery](#) because the logic of building the summarizer is in `Question`, far away from the choice of which summarizer to use, which is in `SummariesController`. Additionally, the `options` parameter needs to be passed down several levels so that summarizer-specific options can be provided when building the summarizer.

Let's switch this up by having the controller build the actual summarizer instance. First, we'll move that logic from `Question` to `SummariesController`:

```
# app/controllers/summaries_controller.rb
def show
  @survey = Survey.find(params[:survey_id])
  @summaries = @survey.summaries_using(summarizer, options)
end

private

def summarizer
  summarizer_name = params[:id]
  summarizer_factory = "Summarizer::#{summarizer_name.classify}".constantize
  summarizer_factory.new(options)
end
```

Then, we'll change `Question#summary_using` to take an instance instead of a name:

```
# app/models/question.rb
def summary_using(summarizer, options)
  value = summarizer.summarize(self)
  Summary.new(title, value)
end
```

That `options` argument is no longer necessary, because it was only used to build the summarizer, which is now handled by the controller. Let's remove it:

```
# app/models/question.rb
def summary_using(summarizer)
  value = summarizer.summarize(self)
  Summary.new(title, value)
end
```

We also don't need to pass it from Survey:

```
# app/models/survey.rb
question.summary_using(summarizer)
```

This interaction has already improved, because the `options` argument is no longer uselessly passed around through two models. It's only used in the controller where the summarizer instance is built. Building the summarizer in the controller is appropriate, because the controller knows the name of the summarizer we want to build, as well as which options are used when building it.

Now that we're using dependency injection, we can take this even further.

In order to prevent the summary from influencing a user's own answers, users don't see summaries for questions they haven't answered yet by default. Users can click a link to override this decision and view the summary for every question.

The information that determines whether or not to hide unanswered questions lives in the controller:

```
# app/controllers/summaries_controller.rb
end

def constraints
  if include_unanswered?
    {}
  else
    { answered_by: current_user }
  end
end
```

However, this information is passed into `Survey#summaries_using`:

```
# app/controllers/summaries_controller.rb
@summaries = @survey.summaries_using(summarizer, options)
```

`Survey#summaries_using` decides whether to hide the answer to each question based on that setting:

```
# app/models/survey.rb
def summaries_using(summarizer, options = {})
  questions.map do |question|
    summary_or_hidden_answer(summarizer, question, options)
  end
end

private

def summary_or_hidden_answer(summarizer, question, options)
  if hide_unanswered_question?(question, options[:answered_by])
    hide_answer_to_question(question)
  else
    question.summary_using(summarizer)
  end
end

def hide_unanswered_question?(question, answered_by)
  answered_by && !question.answered_by?(answered_by)
end

def hide_answer_to_question(question)
  Summary.new(question.title, NO_ANSWER)
end
end
```

Again, the decision is far away from the dependent behavior.

We can combine our dependency injection with a [decorator](#) to remove the duplicate decision:

```
# app/models/unanswered_question_hider.rb
class UnansweredQuestionHider
  NO_ANSWER = "You haven't answered this question".freeze

  def initialize(summarizer, user)
    @summarizer = summarizer
    @user = user
  end

  def summarize(question)
    if hide_unanswered_question?(question)
      NO_ANSWER
    else
      @summarizer.summarize(question)
    end
  end

  private

  def hide_unanswered_question?(question)
    !question.answered_by?(@user)
  end
end
```

We'll decide whether or not to decorate the base summarizer in our controller:

```
# app/controllers/summaries_controller.rb
def decorated_summarizer
  if include_unanswered?
    summarizer
  else
    UnansweredQuestionHider.new(summarizer, current_user)
  end
end
```

Now, the decision of whether or not to hide answers is completely removed from Survey:

```
# app/models/survey.rb
def summaries_using(summarizer)
  questions.map do |question|
    question.summary_using(summarizer)
  end
end
```

For more explanation of using decorators, as well as step-by-step instructions for how to introduce them, see the chapter on [Extract Decorator](#).

## Drawbacks

Injecting dependencies in our example made each class - `SummariesController`, `Survey`, `Question`, and `UnansweredQuestionHider` - easier to understand as a unit. However, it's now difficult to understand why kind of summaries will be produced just by looking at `Survey` or `Question`. You need to follow the stack up to `SummariesController` to understand the dependencies, and then look at each class to understand how they're used.

In this case, we believe that using dependency injection resulted in an overall win for readability and flexibility. However, it's important to remember that the further you move a dependency's resolution from its use, the harder it is to figure out what's actually being used in lower level components.

In our example, there isn't an easy way to know which class will be instantiated for the `summarizer` parameter to `Question#summary_using`:

```
# app/models/question.rb
def summary_using(summarizer)
  value = summarizer.summarize(self)
  Summary.new(title, value)
end
```

In our case, that will be one of `Summarizer::Breakdown`, `Summarizer::MostRecent`, or `Summarizer::UserAnswer`, or a `UnansweredQuestionHider` that decorates one of the above. Developers will need to trace back up through `Survey` to `SummariesController` to gather all the possible implementations.

## Next Steps

- When pulling dependency resolution up into a higher level class, check that class to make sure it doesn't become a [Large Class](#) because of all the logic surrounding dependency resolution.
- If a class is suffering from [Divergent Change](#) because of new or modified dependencies, try moving dependency resolution further up the stack to a container class whose sole responsibility is managing dependencies.
- If methods contain [Long Parameter Lists](#), consider wrapping up several dependencies in a [Parameter Object](#) or Facade.

# Replace Subclasses with Strategies

Subclasses are a common method of achieving reuse and polymorphism, but inheritance has its drawbacks. See [Composition Over Inheritance](#) for reasons why you might decide to avoid an inheritance-based model.

During this refactoring, we will replace the subclasses with individual strategy classes. Each strategy class will implement a common interface. The original base class is promoted from an abstract class to the composition root, which composes the strategy classes.

This allows for smaller interfaces, stricter separation of concerns, and easier testing. It also makes it possible to swap out part of the structure, which would require converting to a new type in an inheritance-based model.

When applying this refactoring to an `ActiveRecord::Base` subclass, [STI](#) is removed, often in favor of a polymorphic association.

## Uses

- Eliminate [Large Classes](#) by splitting up a bloated base class.
- Convert [STI](#) to a composition-based scheme.
- Make it easier to change part of the structure by separating the parts that change from the parts that don't.



## Example

The `switch_to` method on `Question` changes the question to a new type. Any necessary attributes for the new subclass are provided to the `attributes` method.

```
# app/models/question.rb
def switch_to(type, new_attributes)
  attributes = self.attributes.merge(new_attributes)
  new_question = type.constantize.new(attributes.except('id', 'type'))
  new_question.id = id

  begin
    Question.transaction do
      destroy
      new_question.save!
    end
  rescue ActiveRecord::RecordInvalid
  end

  new_question
end
```

Using inheritance makes changing question types awkward for a number of reasons:

- You can't actually change the class of an instance in Ruby, so you need to return the instance of the new class.
- The implementation requires deleting and creating records, but part of the transaction (`destroy`) must execute before we can validate the new instance. This results in control flow using exceptions.
- It's hard to understand why this method is implemented the way it is, so other developers fixing bugs or refactoring in the future will have a hard time navigating it.

We can make this operation easier by using composition instead of inheritance.

This is a difficult change that becomes larger as more behavior is added to the inheritance tree. We can make the change easier by breaking it down into smaller steps, ensuring that the application is in a fully-functional state with passing tests after each change. This allows us to debug in smaller sessions and create safe checkpoint commits that we can retreat to if something goes wrong.

### Use Extract Class to Extract Non-Railsy Methods From Subclasses

The easiest way to start is by extracting a strategy class from each subclass and moving (and delegating) as many methods as you can to the new class. There's some class-level wizardry that goes on in some Rails features like associations, so let's start by moving simple, instance-level methods that aren't part of the framework.

Let's start with a simple subclass: `OpenQuestion`.

Here's the `OpenQuestion` class using an STI model:

```
# app/models/open_question.rb
class OpenQuestion < Question
  def score(text)
    0
  end

  def breakdown
    text_from_ordered_answers = answers.order(:created_at).pluck(:text)
    text_from_ordered_answers.join(', ')
  end
end
```

We can start by creating a new strategy class:

```
class OpenSubmittable
end
```

When switching from inheritance to composition, you need to add a new word to the application's vocabulary. Before, we had questions, and different subclasses of questions handled the variations in behavior and data. Now, we're

switching to a model where there's only one question class, and question will compose *something* that will handle the variations. In our case, that *something* is a “submittable.” In our new model, each question is just a question, and every question composes a submittable that decides how the question can be submitted. Thus, our first extracted class is called `OpenSubmittable`, extracted from `OpenQuestion`.

Let's move our first method over to `OpenSubmittable`:

```
# app/models/open_submittable.rb
class OpenSubmittable
  def score(text)
    0
  end
end
```

And change `OpenQuestion` to delegate to it:

```
# app/models/open_question.rb
class OpenQuestion < Question
  def score(text)
    submittable.score(text)
  end

  def breakdown
    text_from_ordered_answers = answers.order(:created_at).pluck(:text)
    text_from_ordered_answers.join(', ')
  end

  def submittable
    OpenSubmittable.new
  end
end
```

Each question subclass implements the `score` method, so we repeat this process for `MultipleChoiceQuestion` and `ScaleQuestion`. You can see the full change for this step in the [example app](#).

At this point, we've introduced a [parallel inheritance hierarchy](#). During a longer refactor, things may get worse before they get better. This is one of several reasons that it's always best to refactor on a branch, separately from any feature work. We'll make sure that the parallel inheritance hierarchy is removed before merging.

### Pull Up Delegate Method Into Base Class

After the first step, each subclass implements a `submittable` method to build its parallel strategy class. The `score` method in each subclass simply delegates to its `submittable`. We can now pull the `score` method up into the base `Question` class, completely removing this concern from the subclasses.

First, we add a delegator to `Question`:

```
# app/models/question.rb
delegate :score, to: :submittable
```

Then, we remove the `score` method from each subclass.

You can see this change in full in the [example app](#).

### Move Remaining Common API Into Strategies

We can now repeat the first two steps for every non-Railsy method that the subclasses implement. In our case, this is just the `breakdown` method.

The most interesting part of this change is that the `breakdown` method requires state from the subclasses, so the question is now provided to the `submittable`:

```
# app/models/multiple_choice_question.rb
def submittable
  MultipleChoiceSubmittable.new(self)
end
```

```
# app/models/multiple_choice_submittable.rb
def answers
  @question.answers
end

def options
  @question.options
end
```

You can view this change in the [example app](#).

### Move Remaining Non-Railsy Public Methods Into Strategies

We can take a similar approach for the uncommon API; that is, public methods that are only implemented in one subclass.

First, move the body of the method into the strategy:

```
# app/models/scale_submittable.rb
def steps
  (@question.minimum..@question.maximum).to_a
end
```

Then, add a delegator. This time, the delegator can live directly on the subclass, rather than the base class:

```
# app/models/scale_question.rb
def steps
  submittable.steps
end
```

Repeat this step for the remaining public methods that aren't part of the Rails framework. You can see the full change for this step in our [example app](#).

## Remove Delegators From Subclasses

Our subclasses now contain only delegators, code to instantiate the submittable, and framework code. Eventually, we want to completely delete these subclasses, so let's start stripping them down. The delegators are easiest to delete, so let's take them on before the framework code.

First, find where the delegators are used:

```
# app/views/multiple_choice_questions/_multiple_choice_question_form.html.erb
<%= form.fields_for(:options, question.options_for_form) do |option_fields| -%>
  <%= option_fields.input :text, label: 'Option' %>
<% end -%>
```

And change the code to directly use the strategy instead:

```
# app/views/multiple_choice_questions/_multiple_choice_question_form.html.erb
<%= form.fields_for(:options, submittable.options_for_form) do |option_fields| -%>
  <%= option_fields.input :text, label: 'Option' %>
<% end -%>
```

You may need to pass the strategy in where the subclass was used before:

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

We can come back to these locations later and see if we need to pass in the question at all.

After fixing the code that uses the delegator, remove the delegator from the subclass. Repeat this process for each delegator until they've all been removed.

You can see how we do this in the [example app](#).

### Instantiate Strategy Directly From Base Class

If you look carefully at the `submittable` method from each question subclass, you'll notice that it simply instantiates a class based on its own class name and passes itself to the `initialize` method:

```
# app/models/open_question.rb
def submittable
  OpenSubmittable.new(self)
end
```

This is a pretty strong convention, so let's apply some [Convention Over Configuration](#) and pull the method up into the base class:

```
# app/models/question.rb
def submittable
  submittable_class_name = type.sub('Question', 'Submittable')
  submittable_class_name.constantize.new(self)
end
```

We can then delete `submittable` from each of the subclasses.

At this point, the subclasses contain only Rails-specific code like associations and validations.

You can see the full change in the [example app](#).

Also, note that you may want to [scope the `constantize` call](#) in order to make the strategies easy for developers to discover and close potential security vulnerabilities.

### A Fork In the Road

At this point, we're faced with a difficult decision. At a glance, it seems as though only associations and validations live in our subclasses, and we could easily move those to our strategy. However, there are two major issues.

First, you can't move the association to a strategy class without making that strategy an `ActiveRecord::Base` subclass. Associations are deeply coupled with `ActiveRecord::Base`, and they simply won't work in other situations.

Also, one of our submittable strategies has state specific to that strategy. Scale questions have a minimum and maximum. These fields are only used by scale questions, but they're on the questions table. We can't remove this pollution without creating a table for scale questions.

There are two obvious ways to proceed:

- Continue without making the strategies `ActiveRecord::Base` subclasses. Keep the association for multiple choice questions and the minimum and maximum for scale questions on the `Question` class, and use that data from the strategy. This will result in [Divergent Change](#) and probably a [Large Class](#) on `Question`, as every change in the data required for new or existing strategies will require new behavior on `Question`.
- Convert the strategies to `ActiveRecord::Base` subclasses. Move the association and state specific to strategies to those classes. This involves creating a table for each strategy and adding a polymorphic association to `Question`. This will avoid polluting the `Question` class with future strategy changes, but is awkward right now, because the tables for multiple choice questions and open questions would contain no data except the primary key. These tables provide a placeholder for future strategy-specific data, but those strategies may never require any more data and until they do, the tables are a waste of queries and developer mental space.

In this example, I'm going to move forward with the second approach, because:

- It's easier with `ActiveRecord`. `ActiveRecord` will take care of instantiating the strategy in most situations if it's an association, and it has special behavior for associations using nested attribute forms.
- It's the easiest way to avoid [Divergent Change](#) and [Large Classes](#) in a Rails application. Both of these smells can cause problems that are hard to fix if you wait too long.



### Convert Strategies to ActiveRecord subclasses

Continuing with our refactor, we'll change each of our strategy classes to inherit from `ActiveRecord::Base`.

First, simply declare that the class is a child of `ActiveRecord::Base`:

```
# app/models/open_submittable.rb
class OpenSubmittable < ActiveRecord::Base
```

Your tests will complain that the corresponding table doesn't exist, so create it:

```
# db/migrate/20130131205432_create_open_submittables.rb
class CreateOpenSubmittables < ActiveRecord::Migration
  def change
    create_table :open_submittables do |table|
      table.timestamps null: false
    end
  end
end
```

Our strategies currently accept the question as a parameter to initialize and assign it as an instance variable. In an `ActiveRecord::Base` subclass, we don't control initialize, so let's change question from an instance variable to an association and pass a hash:

```
# app/models/open_submittable.rb
class OpenSubmittable < ActiveRecord::Base
  has_one :question, as: :submittable

  def breakdown
    text_from_ordered_answers = answers.order(:created_at).pluck(:text)
    text_from_ordered_answers.join(', ')
  end

  def score(text)
    0
  end

  private

  def answers
    question.answers
  end
end

# app/models/question.rb
def submittable
  submittable_class = type.sub('Question', 'Submittable').constantize
  submittable_class.new(question: self)
end
```

Our strategies are now ready to use Rails-specific functionality like associations and validations.

View the full change on [GitHub](#).

## Introduce A Polymorphic Association

Now that our strategies are persistable using ActiveRecord, we can use them in a polymorphic association. Let's add the association:

```
# app/models/question.rb
belongs_to :submittable, polymorphic: true
```

And add the necessary columns:

```
# db/migrate/20130131203344_add_submittable_type_and_id_to_questions.rb
class AddSubmittableTypeAndIdToQuestions < ActiveRecord::Migration
  def change
    add_column :questions, :submittable_id, :integer
    add_column :questions, :submittable_type, :string
  end
end
```

We're currently defining a submittable method that overrides the association. Let's change that to a method that will build the association based on the STI type:

```
# app/models/question.rb
def build_submittable
  submittable_class = type.sub('Question', 'Submittable').constantize
  self.submittable = submittable_class.new(question: self)
end
```

Previously, the submittable method built the submittable on demand, but now it's persisted in an association and built explicitly. Let's change our controllers accordingly:

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

View the full change on [GitHub](#).

### Pass Attributes to Strategies

We're persisting the strategy as an association, but the strategies currently don't have any state. We need to change that, since scale submittables need a minimum and maximum.

Let's change our `build_submittable` method to accept attributes:

```
# app/models/question.rb
def build_submittable(attributes)
  submittable_class = type.sub('Question', 'Submittable').constantize
  self.submittable = submittable_class.new(attributes.merge(question: self))
end
```

We can quickly change the invocations to pass an empty hash, and we're back to green.

Next, let's move the `minimum` and `maximum` fields over to the `scale_submittables` table:

```
# db/migrate/20130131211856_move_scale_question_state_to_scale_submittable.rb
add_column :scale_submittables, :minimum, :integer
add_column :scale_submittables, :maximum, :integer
```

Note that this migration is [rather lengthy](#), because we also need to move over the `minimum` and `maximum` values for existing questions. The SQL in our example app will work on most databases, but is cumbersome. If you're using PostgreSQL, you can handle the down method easier using an `UPDATE FROM` statement.

Next, we'll move validations for these attributes over from `ScaleQuestion`:

```
# app/models/scale_submittable.rb
validates :maximum, presence: true
validates :minimum, presence: true
```

And change `ScaleSubmittable` methods to use those attributes directly, rather than looking for them on `question`:

```
# app/models/scale_submittable.rb
def steps
  (minimum..maximum).to_a
end
```

We can pass those attributes in our form by using `fields_for` and `accepts_nested_attributes_for`:

```
# app/views/scale_questions/_scale_question_form.html.erb
<%= form.fields_for :submittable do |submittable_fields| -%>
  <%= submittable_fields.input :minimum %>
  <%= submittable_fields.input :maximum %>
<% end -%>
```

```
# app/models/question.rb
accepts_nested_attributes_for :submittable
```

In order to make sure the `Question` fails when its `submittable` is invalid, we can cascade the validation:

```
# app/models/question.rb
validates :submittable, associated: true
```

Now we just need our controllers to pass the appropriate `submittable` params:

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

```
# app/controllers/questions_controller.rb
def question_params
  params.
    require(:question).
    permit(:title, :options_attributes)
end

def submittable_params
  if submittable_attributes = params[:question][:submittable_attributes]
    submittable_attributes.permit(:minimum, :maximum)
  else
    {}
  end
end
```

All behavior and state is now moved from `ScaleQuestion` to `ScaleSubmittable`, and the `ScaleQuestion` class is completely empty.

You can view the full change in the [example app](#).

### Move Remaining Railsy Behavior Out of Subclasses

We can now repeat this process for remaining Rails-specific behavior. In our case, this is the logic to handle the `options` association for multiple choice questions.

We can move the association and behavior over to the strategy class:

```
# app/models/multiple_choice_submittable.rb
has_many :options, foreign_key: :question_id
has_one :question, as: :submittable

accepts_nested_attributes_for :options, reject_if: :all_blank
```

Again, we remove the `options` method which delegated to `question` and rely on `options` being directly available. Then we update the form to use `fields_for` and move the allowed attributes in the controller from `question` to `submittable`.

At this point, every question subclass is completely empty.

You can view the full change in the [example app](#).

### Backfill Strategies For Existing Records

Now that everything is moved over to the strategies, we need to make sure that submittables exist for every existing question. We can write a quick backfill migration to take care of that:

```
# db/migrate/20130207164259_backfill_submittables.rb
class BackfillSubmittables < ActiveRecord::Migration
  def up
    backfill 'open'
    backfill 'multiple_choice'
  end

  def down
    connection.delete 'DELETE FROM open_submittables'
    connection.delete 'DELETE FROM multiple_choice_submittables'
  end

  private

  def backfill(type)
    say_with_time "Backfilling #{type} submittables" do
      connection.update(<<-SQL)
        UPDATE questions
        SET
          submittable_id = id,
          submittable_type = '#{type.camelize}Submittable'
        WHERE type = '#{type.camelize}Question'
      SQL
      connection.insert(<<-SQL)
        INSERT INTO #{type}_submittables
          (id, created_at, updated_at)
        SELECT
```

```
      id, created_at, updated_at
    FROM questions
    WHERE questions.type = '#{type.camelize}Question'
  SQL
end
end
end
```

We don't port over scale questions, because we took care of them in a [previous migration](#).

### Pass the Type When Instantiating the Strategy

At this point, the subclasses are just dead weight. However, we can't delete them just yet. We're relying on the `type` column to decide what type of strategy to build, and Rails will complain if we have a `type` column without corresponding subclasses.

Let's remove our dependence on that `type` column. Accept a type when building the submittable:

```
# app/models/question.rb
def build_submittable(type, attributes)
  submittable_class = type.sub('Question', 'Submittable').constantize
  self.submittable = submittable_class.new(attributes.merge(question: self))
end
```

And pass it in when calling:

```
# app/controllers/questions_controller.rb
@question.build_submittable(type, submittable_params)
```

[Full Change](#)



### Always Instantiate the Base Class

Now we can remove our dependence on the STI subclasses by always building an instance of `Question`.

In our controller, we change this line:

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

To this:

```
# app/controllers/questions_controller.rb
@question = Question.new(question_params)
```

We're still relying on `type` as a parameter in forms and links to decide what type of submittable to build. Let's change that to `submittable_type`, which is already available because of our polymorphic association:

```
# app/controllers/questions_controller.rb
params[:question][:submittable_type]
```

```
# app/views/questions/_form.html.erb
<%= form.hidden_field :submittable_type %>
```

We'll also need to revisit views that rely on [polymorphic partials](#) based on the question type and change them to rely on the submittable type instead:

```
# app/views/surveys/show.html.erb
<%= render(
  question.submittable,
  submission_fields: submission_fields
) %>
```

Now we can finally remove our `type` column entirely:

```
# db/migrate/20130207214017_remove_questions_type.rb
class RemoveQuestionsType < ActiveRecord::Migration
  def up
    remove_column :questions, :type
  end

  def down
    add_column :questions, :type, :string

    connection.update(<<-SQL)
      UPDATE questions
      SET type = REPLACE(submittable_type, 'Submittable', 'Question')
    SQL

    change_column_null :questions, :type, true
  end
end
```

## Full Change

### Remove Subclasses

Now for a quick, glorious change: those `Question` subclasses are entirely empty and unused, so we can [delete them](#).

This also removes the [parallel inheritance hierarchy](#) that we introduced earlier.

At this point, the code is as good as we found it.

### Simplify Type Switching

If you were previously switching from one subclass to another as we did to change question types, you can now greatly simplify that code.

Instead of deleting the old question and cloning it with a merged set of old generic attributes and new specific attributes, you can simply swap in a new strategy for the old one.

```
# app/models/question.rb
def switch_to(type, attributes)
  old_submittable = submittable
  build_submittable type, attributes

  transaction do
    if save
      old_submittable.destroy
    end
  end
end
```

Our new `switch_to` method is greatly improved:

- This method no longer needs to return anything, because there's no need to clone. This is nice because `switch_to` is now simply a command method (it does something) rather than a mixed command and query method (it does something and returns something).
- The method no longer needs to delete the old question, and the new submittable is valid before we delete the old one. This means we no longer need to use exceptions for control flow.
- It's simpler and its code is obvious, so other developers will have no trouble refactoring or fixing bugs.

You can see the full change that resulted in our new method in the [example app](#).

## Conclusion

Our new, composition-based model is improved in a number of ways:

- It's easy to change types.
- Each submittable is easy to use independently of its question, reducing coupling.

- There's a clear boundary in the API for questions and submittables, making it easier to test and making it less likely that concerns leak between the two.
- Shared behavior happens via composition, making it less likely that the base class becomes a [large class](#).
- It's easy to add new state without effecting other types, because strategy-specific state is stored on a table for that strategy.

You can view the entire refactor with all steps combined in the [example app](#) to get an idea of what changed at the macro level.

This is a difficult transition to make, and the more behavior and data that you shove into an inheritance scheme, the harder it becomes. In situations where [STI](#) is not significantly easier than using a polymorphic relationship, it's better to start with composition. STI provides few advantages over composition, and it's easier to merge models than to split them.

## Drawbacks

Our application also got worse in a number of ways:

- We introduced a new word into the application vocabulary. This can increase understanding of a complex system, but vocabulary overload makes simpler systems unnecessarily hard to learn.
- We now need two queries to get a question's full state, and we'll need to query up to four tables to get information about a set of questions.
- We introduced useless tables for two of our question types. This will happen whenever you use ActiveRecord to back a strategy without state.
- We increased the overall complexity of the system. In this case, it may have been worth it, because we reduced the complexity per component. However, it's worth keeping an eye on.

Before performing a large change like this, try to imagine what will be easy to change in the new world that's hard right now.

After performing a large change, keep track of difficult changes you make. Would they have been easier in the old world?

Answering this questions will increase your ability to judge whether or not to use composition or inheritance in future situations.

## Next Steps

- Check the extracted strategy classes to make sure they don't have [Feature Envy](#) related to the original base class. You may want to use [Move Method](#) to move methods between strategies and the root class.
- Check the extracted strategy classes for [Duplicated Code](#) introduced while splitting up the base class. Use [Extract Method](#) or [Extract Class](#) to extract common behavior.

# Replace mixin with composition

Mixins are one of two mechanisms for inheritance in Ruby. This refactoring provides safe steps for cleanly removing mixins that have become troublesome.

Removing a mixin in favor of composition involves the following steps:

- Extract a class for the mixin.
- Compose and delegate to the extracted class from each mixed in method.
- Replace references to mixed in methods with references to the composed class.
- Remove the mixin.

## Uses

- Liberate business logic trapped in mixins.
- Eliminate name clashes from multiple mixins.
- Make methods in the mixins easier test.

## Example

In our example applications, invitations can be delivered either by email or private message (to existing users). Each invitation method is implemented in its own class:

```
# app/models/message_inviter.rb
class MessageInviter < AbstractController::Base
  include Inviter

  def initialize(invitation, recipient)
    @invitation = invitation
    @recipient = recipient
  end

  def deliver
    Message.create!(
      recipient: @recipient,
      sender: @invitation.sender,
      body: render_message_body
    )
  end
end

# app/models/email_inviter.rb
class EmailInviter < AbstractController::Base
  include Inviter

  def initialize(invitation)
    @invitation = invitation
  end

  def deliver
    Mailer.invitation_notification(@invitation, render_message_body).deliver
  end
end
```

The logic to generate the invitation message is the same regardless of the delivery mechanism, so this behavior has been extracted.

It's currently extracted using a mixin:

```
# app/models/inviter.rb
module Inviter
  extend ActiveSupport::Concern

  included do
    include ActionController::Rendering
    include Rails.application.routes.url_helpers

    self.view_paths = 'app/views'
    self.default_url_options = ActionMailer::Base.default_url_options
  end

  private

  def render_message_body
    render template: 'invitations/message'
  end
end
```

Let's replace this mixin with composition.



First, we'll [extract a new class](#) for the mixin:

```
# app/models/invitation_message.rb
class InvitationMessage < ActionController::Base
  include ActionController::Rendering
  include Rails.application.routes.url_helpers

  self.view_paths = 'app/views'
  self.default_url_options = ActionMailer::Base.default_url_options

  def initialize(invitation)
    @invitation = invitation
  end

  def body
    render template: 'invitations/message'
  end
end
```

This class contains all the behavior the formerly resided in the mixin. In order to keep everything working, we'll compose and delegate to the extracted class from the mixin:

```
# app/models/inviter.rb
module Inviter
  private

  def render_message_body
    InvitationMessage.new(@invitation).body
  end
end
```

Next, we can replace references to the mixed in methods (`render_message_body` in this case) with direct references to the composed class:

```
# app/models/message_inviter.rb
class MessageInviter
  def initialize(invitation, recipient)
    @invitation = invitation
    @recipient = recipient
    @body = InvitationMessage.new(@invitation).body
  end

  def deliver
    Message.create!(
      recipient: @recipient,
      sender: @invitation.sender,
      body: @body
    )
  end
end

# app/models/email_inviter.rb
class EmailInviter
  def initialize(invitation)
    @invitation = invitation
    @body = InvitationMessage.new(@invitation).body
  end

  def deliver
    Mailer.invitation_notification(@invitation, @body).deliver
  end
end
```

In our case, there was only one method to move. If your mixin has multiple methods, it's best to move them one at a time.

Once every reference to a mixed in method is replaced, you can remove the mixed in method. Once every mixed in method is removed, you can remove the mixin entirely.

## Next Steps

- [Inject Dependencies](#) to invert control and allow the composing classes to use different implementations for the composed class.
- Check the composing class for [Feature Envy](#) of the extracted class. Tight coupling is common between mixin methods and host methods, so you may need to use [move method](#) a few times to get the balance right.

# Replace Callback with Method

If your models are hard to use and change because their persistence logic is coupled with business logic, one way to loosen things up is by replacing [callbacks](#).

## Uses

- Reduces coupling persistence logic with business logic.
- Makes it easier to extract concerns from models.
- Fixes bugs from accidentally triggered callbacks.
- Fixes bugs from callbacks with side effects when transactions roll back.

## Steps

- Use [Extract Method](#) if the callback is an anonymous block.
- Promote the callback method to a public method if it's private.
- Call the public method explicitly rather than relying on save and callbacks.

## Example

```
# app/models/survey_inviter.rb
def deliver_invitations
  recipients.map do |recipient_email|
    Invitation.create!(
      survey: survey,
      sender: sender,
      recipient_email: recipient_email,
      status: 'pending',
      message: @message
    )
  end
end

# app/models/invitation.rb
after_create :deliver

# app/models/invitation.rb
private

def deliver
  Mailer.invitation_notification(self).deliver
end
```

In the above code, the `SurveyInviter` is simply creating `Invitation` records, and the actual delivery of the invitation email is hidden behind `Invitation.create!` via a callback.

If one of several invitations fails to save, the user will see a 500 page, but some of the invitations will already have been saved and delivered. The user will be unable to tell which invitations were sent.

Because delivery is coupled with persistence, there's no way to make sure that all of the invitations are saved before starting to deliver emails.

Let's make the callback method public so that it can be called from `SurveyInviter`:

```
# app/models/invitation.rb
def deliver
  Mailer.invitation_notification(self).deliver
end

private
```

Then remove the `after_create` line to detach the method from persistence.

Now we can split invitations into separate persistence and delivery phases:

```
# app/models/survey_inviter.rb
def deliver_invitations
  create_invitations.each(&:deliver)
end

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

If any of the invitations fail to save, the transaction will roll back. Nothing will be committed, and no messages will be delivered.

## Next Steps

- Find other instances where the model is saved to make sure that the extracted method doesn't need to be called.

# Use convention over configuration

Ruby's metaprogramming allows us to avoid boilerplate code and duplication by relying on conventions for class names, file names, and directory structure. Although depending on class names can be constricting in some situations, careful use of conventions will make your applications less tedious and more bug-proof.

## Uses

- Eliminate [Case Statements](#) by finding classes by name.
- Eliminate [Shotgun Surgery](#) by removing the need to register or configure new strategies and services.
- Remove [Duplicated Code](#) by removing manual associations from identifiers to class names.

## Example

This controller accepts an `id` parameter identifying which summarizer strategy to use and renders a summary of the survey based on the chosen strategy:

```
# app/controllers/summaries_controller.rb
class SummariesController < ApplicationController
  def show
    @survey = Survey.find(params[:survey_id])
    @summaries = @survey.summarize(summarizer)
  end

  private

  def summarizer
    case params[:id]
    when 'breakdown'
      Breakdown.new
    when 'most_recent'
      MostRecent.new
    when 'your_answers'
      UserAnswer.new(current_user)
    else
      raise "Unknown summary type: #{params[:id]}"
    end
  end
end
```

The controller is manually mapping a given strategy name to an object that can perform the strategy with the given name. In most cases, a strategy name directly maps to a class of the same name.

We can use the `constantize` method from Rails to retrieve a class by name:

```
params[:id].classify.constantize
```

This will find the `MostRecent` class from the string `"most_recent"`, and so on. This means we can rely on a convention for our summarizer strategies: each named



strategy will map to a class implementing that strategy. The controller can [use the class as an Abstract Factory](#) and obtain a summarizer.

However, we can't immediately start using `constantize` in our example, because there's one outlier case: the `UserAnswer` class is referenced using `"your_answers"` instead of `"user_answer"`, and `UserAnswer` takes different parameters than the other two strategies.

Before refactoring the code to rely on our new convention, let's refactor to obey it. All our names should map directly to class names, and each class should accept the same parameters:

```
# app/controllers/summaries_controller.rb
def summarizer
  case params[:id]
  when 'breakdown'
    Breakdown.new(user: current_user)
  when 'most_recent'
    MostRecent.new(user: current_user)
  when 'user_answer'
    UserAnswer.new(user: current_user)
  else
    raise "Unknown summary type: #{params[:id]}"
  end
end
```

Now that we know we can instantiate any of the summarizer classes the same way, let's extract a method for determining the summarizer class:

```
# app/controllers/summaries_controller.rb
def summarizer
  summarizer_class.new(user: current_user)
end

def summarizer_class
  case params[:id]
  when 'breakdown'
    Breakdown
  when 'most_recent'
    MostRecent
  when 'user_answer'
    UserAnswer
  else
    raise "Unknown summary type: #{params[:id]}"
  end
end
```

Now the extracted class performs exactly the same logic as `constantize`, so let's use it:

```
# app/controllers/summaries_controller.rb
def summarizer
  summarizer_class.new(user: current_user)
end

def summarizer_class
  params[:id].classify.constantize
end
```

Now we'll never need to change our controller when adding a new strategy; we just add a new class following the naming convention.

## Scoping constantize

Our controller currently takes a string directly from user input (`params`) and instantiates a class with that name.

There are two issues with this approach that should be fixed:

- There's no list of available strategies, so a developer would need to perform a complicated search to find the relevant classes.
- Without a whitelist, a user can make the application instantiate any class they want by hacking parameters. This can result in security vulnerabilities.

We can solve both easily by altering our convention slightly: scope all the strategy classes within a module.

We change our strategy factory method:

```
# app/controllers/summaries_controller.rb
def summarizer
  summarizer_class.new(user: current_user)
end

def summarizer_class
  params[:id].classify.constantize
end
```

To:

```
# app/controllers/summaries_controller.rb
def summarizer_class
  "Summarizer::#{params[:id].classify}".constantize
end
```

With this convention in place, you can find all strategies by just looking in the `Summarizer` module. In a Rails application, this will be in a `summarizer` directory by convention.

Users also won't be able to instantiate anything they want by abusing our `constantize`, because only classes in the `Summarizer` module are available.

## Drawbacks

### Weak Conventions

Conventions are most valuable when they're completely consistent.

The convention is slightly forced in this case because `UserAnswer` needs different parameters than the other two strategies. This means that we now need to add no-op initializer methods to the other two classes:

```
# app/models/summarizer/breakdown.rb
class Summarizer::Breakdown
  def initialize(options)
  end

  def summarize(question)
    question.breakdown
  end
end
```

This isn't a deal-breaker, but it makes the other classes a little noisier, and adds the risk that a developer will waste time trying to remove the unused parameter.

Every compromise made weakens the convention, and having a weak convention is worse than having no convention. If you have to change the convention for every class you add that follows it, try something else.

### Class-Oriented Programming

Another drawback to this solution is that it's entirely class-based, which means you can't assemble strategies at run-time. This means that reuse requires inheritance.

Also, this class-based approach, while convenient when developing an application, is more likely to cause frustration when writing a library. Forcing developers to pass a class name instead of an object limits the amount of runtime information strategies can use. In our example, only a user was required. When

you control both sides of the API, it's fine to assume that this is safe. When writing a library that will interface with other developers' applications, it's better not to rely on class names.

# Introduce Visitor

STUB

## **Part III**

# **Principles**

# DRY

The DRY principle - short for “don’t repeat yourself” - comes from [The Pragmatic Programmer](#).

The principle states:

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

Following this principle is one of the best ways to prevent bugs and move faster. Every duplicated piece of knowledge is a bug waiting to happen. Many development techniques are really just ways to prevent and eliminate duplication, and many smells are just ways to detect existing duplication.

When knowledge is duplicated, changing it means making the same change in several places. Leaving duplication introduces a risk that the various duplicate implementations will slowly diverge, making them harder to merge and making it more likely that a bug remains in one or more incarnations after being fixed.

Duplication leads to frustration and paranoia. Rampant duplication is a common reason that developers reach for a Grand Rewrite.

## Duplicated Knowledge vs Duplicated Text

It’s important to understand that this principle states that knowledge should not be repeated; it does not state that text should never be repeated.



For example, this sample does not violate the DRY principle, even though the word “save” is repeated several times:

```
def sign_up
  @user.save
  @account.save
  @subscription.save
end
```

However, this code contains duplicated knowledge that could be extracted:

```
def sign_up_free
  @user.save
  @account.save
  @trial.save
end

def sign_up_paid
  @user.save
  @account.save
  @subscription.save
end
```

## Application

The following smells may point towards [duplicated code](#) and can be avoided by following the DRY principle:

- [Shotgun Surgery](#) caused by changing the same knowledge in several places.
- [Long Paramter Lists](#) caused by not encapsulating related properties.
- [Feature Envy](#) caused by leaking internal knowledge of a class that can be encapsulated and reused.
- [Parallel Inheritance Hierarchies](#) duplicate knowledge of types.

You can use these solutions to remove duplication and make knowledge easier to reuse:

- [Extract Classes](#) to encapsulate knowledge, allowing it to be reused.
- [Extract Methods](#) to reuse behavior within a class.
- [Extract Partial](#)s to remove duplication in views.
- [Extract Validators](#) to encapsulate validations.
- [Replace Conditionals with Null Objects](#) to encapsulate behavior related to nothingness.
- [Replace Conditionals With Polymorphism](#) to make it easy to reuse behavioral branches.
- [Replace Mixins With Composition](#) to make it easy to combine components in new ways.
- [Use Convention Over Configuration](#) to infer knowledge, making it impossible to duplicate.

Applying these techniques before duplication occurs will make it less likely that duplication will occur. If you want to prevent duplication, make knowledge easier to reuse by keeping classes small and focused.

Related principles include the [Law of Demeter](#) and the [Single Responsibility Principle](#).

# 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