Ruby Science

The reference for writing fantastic Rails applications.

Ruby Science

thoughtbot Joe Ferris Harlow Ward

March 12, 2014

Contents

Introduction

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

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

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

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

Code Reviews

Our first step toward better code is to review it.

Have you ever sent an email with typos? Did you review what you wrote before

clicking “Send”? Reviewing your e-mails prevents mistakes and reviewing your code does the same.

To make it easier to review code, always work in a feature branch. The branch reduces the temptation to push unreviewed code or to wait too long to push code.

The first person who should review every line of your code is you. Before committing new code, read each changed line. Use git’s `diff` and `--patch` features to examine code before you commit. Read more about these features using `git help add` and `git help commit`.

If you’re working on a team, push your feature branch and invite your teammates to review the changes via `git diff origin/master..HEAD`.

Team review reveals how understandable code is to someone other than the author. Your team members’ understanding now is a good indicator of your understanding in the future.

However, what should you and your teammates look for during review?

Follow Your Nose

Code “smells” are indicators something may be wrong. They are useful because they are easy to see—sometimes easier than the root cause of a problem.

When you review code, watch for smells. Consider whether refactoring the code to remove the smell would result in better code. If you’re reviewing a teammate’s feature branch, share your best refactoring ideas with him or her.

Smells are associated with one or more refactorings (example: remove the Long Method smell using the Extract Method refactoring). Learn these associations in order to quickly consider them during review and whether the result (example: several small methods) improves the code.

Don’t treat code smells as bugs. It will be a waste of time to “fix” every smell. Not every smell is the symptom of a problem and despite your best intentions, you can accidentally introduce another smell or problem.

Removing Resistance

Another opportunity for refactoring is when you're having difficulty making a change to existing code. This is called "resistance." The refactoring you choose depends on the type of resistance.

Is it hard to determine where new code belongs? The code is not readable enough. Rename methods and variables until it's obvious where your change belongs. This and every subsequent change will be easier. Refactor for readability first.

Is it hard to change the code without breaking existing code? 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.

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

When you are making your changes, you will be in a feature branch. Try to make your change without refactoring. If your meet resistance, make a "work in progress" commit, check out master and create a new refactoring branch:

```
git commit -m 'wip: new feature'
git push
git checkout master
git checkout -b refactoring-for-new-feature
```

Refactor until you fix the resistance you met on your feature branch. Then rebase your feature branch on top of your refactoring branch:

```
git rebase -i new-feature
git checkout new-feature
git merge refactoring-for-new-feature --ff-only
```

If the change is easier now, continue in your feature branch. If not, check out your refactoring branch and try again.

Bugs and Churn

If you're spending a lot of time swatting bugs, remove smells in the methods or classes of the buggy code. You'll make it less likely that a bug will be reintroduced.

After you commit a bug fix to a feature branch, find out if the code you changed to fix the bug is in files that change often. If the buggy code does change often, find smells and eliminate them. Separate the parts that change often from the parts that don't.

Conversely, avoid refactoring areas with low churn. Refactoring changes code and with each change, you risk introducing new bugs. If a file hasn't changed in six months, leave it alone. It may not be pretty, but you'll spend more time looking at it when you break it by trying to fix something that wasn't broken.

Tools to Find Smells

Some smells are easy to find while you're reading code and change sets, but other smells slip through the cracks without extra help.

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, he or she won't notice that the copied method isn't just a new addition. Every duplicated piece of code is a bug waiting to happen.

Churn is similarly invisible, in that each change will look fine, and only the file's full history will reveal the smell.

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

Our favorite is Code Climate, which is a hosted tool and 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. It identifies complexity, duplication, churn and code smells.

If you're unable to use a hosted service, there are gems you can use locally, such as metric_fu, churn, flog, flay, and reek. These gems can identify churn, complexity, duplication and smells.

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 explains which problems it addresses and potential problems that can be introduced.

Lastly, smell and solution chapters reference related principles. The smell chapters reference principles that you can follow to avoid the root problem in the future. The solution chapters 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.

Example Application

This book comes with a bundled example application, which you can find on GitHub. Make sure that you sign into GitHub before attempting to view example application and commit links, or you'll receive a 404 error.

Most of the code samples included in the book come directly from commits in the example application. Several chapters link to the full commits for related

changes. At any point, you can check out the application locally and check out those commits to explore solutions in progress.

For some solutions, the entire change is not included in the chapter for the sake of focus and brevity. However, you can see every change made for a solution in the example commits, including tests.

Make sure to take a look at the application's README, as it contains a summary of the application and instructions for setting it up.

Part I

Code Smells

Long Method

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

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

Symptoms

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

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

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

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

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

Example

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

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

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

Solutions

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

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

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 than seven 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 responsibilities:

```
# 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.

Using composition over inheritance makes it easier to create small classes.

If a large portion of the class is devoted to instantiating subclasses, try following the dependency inversion principle.

Following the open/closed principle will prevent large classes by preventing new concerns from being introduced.

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
```

Private Methods

In general, public methods are a greater liability than private methods. This is because it's harder to tell where public methods are used, so you need to take greater care when refactoring them. However, a large suite of private methods is also a strong indicator of a large class.

Private methods can't be reused between classes, which makes it more likely that code will be duplicated. Extracting private methods to new classes makes it easier for developers to do the right thing.

Additionally, private methods can't be tested directly. This makes it more difficult to write focused, simple unit tests, since the tests will need to go through one or more public methods. The further a test is from the code it tests, the harder it is to understand.

Lastly, private methods are often the easiest to extract to new classes. Large classes can be difficult to split up because of entangled dependencies between public and private methods.

Attempts to extract public methods will frequently halt when shared dependencies are discovered on private methods. Extracting the private behavior of a class into a small, reusable class is often the easiest first step towards splitting up a large class.

Keeping a class's public interface as small as possible is a best practice. However, keep an eye on your private interface as well. A maze of private dependencies is a good sign that your public interface is not cohesive and can be split into two or more classes.

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: the 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, 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 the 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 that are used more than methods and instance variables of the class in question.
- Methods that include 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; then move the method.
- Move method if the entire method suffers from feature envy.
- Inline class if the envied class isn't pulling its weight.

Prevention

Following the law of Demeter will prevent a lot of feature envy by limiting the dependencies of each method.

Following tell, don't ask will prevent feature envy by avoiding unnecessary inspection of another object's state.

Case Statement

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

Symptoms

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

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

Type Codes

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

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

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

Example

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

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

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

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

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

Solutions

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

Shotgun Surgery

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

Symptoms

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

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

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

Example

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

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

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

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

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

Solutions

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

Prevention

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

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

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

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

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 many reasons to change:

- Control flow logic related to summaries, such as authentication.
- Any instance in which 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.

Following the open/closed principle limits future changes to classes, including divergent change.

Following composition over inheritance will make it easier to create small classes, preventing divergent change.

If a large portion of the class is devoted to instantiating subclasses, try following the dependency inversion principle.

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

Anti-Solution

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

Duplicated Code

One of the first principles we're taught as developers: Don't Repeat Yourself.

Symptoms

- You find yourself copying 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.

Prevention

Following the single responsibility principle will result in small classes that are easier to reuse, reducing the temptation of duplication.

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 that 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.summarize(summarizer)
```

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

```
# app/models/survey.rb
def summarize(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 makes 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 takes care of most of the details by 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 compel 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 you 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.

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 that have few public methods except those from a mixin.
- Inverting dependencies is difficult because mixins can't accept parameters.

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 AbstractController::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 < 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
```

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 containing business logic, such as processing payments.
- Attributes that allow certain callbacks to be skipped.
- Methods such as `save_without_sending_email`, which skip callbacks.
- Callbacks that 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.
- Prevents 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.
- Makes conditional logic easier to reuse, which makes it easier to avoid duplication.

- Replaces conditional logic with simple commands, following tell, don't ask.

Example

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

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

  SUBMITTABLE_TYPES = %w(Open MultipleChoice Scale).freeze

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

  belongs_to :survey
  has_many :answers
  has_many :options

  accepts_nested_attributes_for :options, reject_if: :all_blank

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

  def steps
```

```
    (minimum..maximum).to_a
  end

  private

  def scale?
    question_type == 'Scale'
  end

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

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

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

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

- Adding a new question type will require modifying the method, leading to divergent change.
- The logic and data for summarizing every type of question and answer is jammed into the `Question` class, resulting in a large class with obscure code.
- This method isn't the only place in the application where question types are checked, meaning that new types will cause shotgun surgery.

There are several ways to refactor to use polymorphism. In this chapter, we'll demonstrate a solution that uses subclasses to replace type codes, which is one of the simplest solutions to implement. However, make sure to see the Drawbacks section in this chapter for alternative implementations.

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 c18ebeb for the full change.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

—will ask `@question` which view should be rendered by calling `to_partial_path`. As subclasses of `ActiveRecord::Base`, our `Question` subclasses will return a path

based on their class name. This means that the above line will attempt to render `open_questions/_open_question.html.erb` for an open question, and so on.

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

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

You can see the full change in commit 8243493.

Multiple Polymorphic Views

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

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

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

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

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

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

Drawbacks

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

Using polymorphism like this makes it easier to add new types, because adding a new type means that 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 toward polymorphic implementations, but if you find yourself adding behaviors much more often than adding types, you should look into using observers or visitors instead.

Using subclasses forces you to use inheritance instead of composition for reuse and separation of concerns. See composition over inheritance for more on this subject.

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`.
- Makes logic related to presence and absence easier to reuse, making it easier to avoid duplication.
- Replaces conditional logic with simple commands, following tell, don't ask.

Example

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

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

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

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

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

Again, `for_user` might return `nil`:

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

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

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

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

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

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

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

  belongs_to :completion
  belongs_to :question

  validates :text, presence: true

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

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

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

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

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

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

  belongs_to :completion
  belongs_to :question

  validates :text, presence: true

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

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

  private

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

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

Drawbacks

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

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

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

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

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

Next Steps

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

Truthiness, `try` and Other Tricks

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

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

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

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

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

Extract Method

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

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

Uses

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

Example

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

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

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

This method performs a number of tasks:

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

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

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

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

private

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

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

Replace Temp with Query

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

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

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

Other Examples

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

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

Next Steps

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

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

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.
- 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 to 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.
- Allows concerns to be injected, following the dependency inversion principle.
- Makes behavior easier to reuse, which makes it easier to avoid duplication.

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
```

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
```

```
      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
```

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've 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 that 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 results 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 @invlid_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

  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 that 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 facilitate communication between developers. However, introducing too many names results in vocabulary overload, which makes the system difficult to learn for new developers.

If you extract classes in response to pain and resistance, 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

- Prevent 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 place new concerns on top of existing objects without modifying existing classes. They combine best with small classes containing 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.

It will be difficult to make use of decorators unless your application is following composition over inheritance.

Uses

- Eliminate large classes by extracting concerns.
- Eliminate divergent change and follow the single responsibility principle by adding new behavior without introducing new concerns to existing classes.
- Prevent conditional logic from leaking by making decisions earlier.
- Extend existing classes without modifying them, following the open/closed principle.

Example

In our example application, users can view a summary of the answers to each question on a survey. By default, 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. 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 affects 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 alternative, 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 we 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 the context is more specific now, so 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 the 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 that 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, since 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

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

Steps

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

Example

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

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

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

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


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

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

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

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

Move the submit button text into the locales file:

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

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

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

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

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

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

Next Steps

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

Extract Validator

Extract Validator is a form of extract class that is used to remove complex validation details from `ActiveRecord` models. This technique also prevents duplication of validation code across several files.

Uses

- Keeps validation implementation details out of models.
- Encapsulates validation details into a single file, following the single responsibility principle.
- Removes duplication among classes performing the same validation logic.
- Makes validation logic easier to reuse, which makes it easier to avoid duplication.

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 < ActiveModel::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 lower case, 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 deemed hard enough to understand that adding a comment was necessary:

```
# 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 requiring 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 from the top down. Expressions based on variables place the details first, which means that developers 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 ensure 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 to the order in which a developer naturally 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 Form Object

This is a specialized type of extract class that is used to remove business logic from controllers when processing data outside of an ActiveRecord model.

Uses

- Keeps business logic out of controllers and views.
- Adds validation support to plain old Ruby objects.
- Displays form validation errors using Rails conventions.
- Sets 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 that the `message` is present.
- Validates each of the `recipients`' email addresses.
- Creates an invitation for each of the recipients.
- Sends an email to each of the recipients.
- Sets view data for validation failures.

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

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

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

  private

  def valid_recipients?
    invalid_recipients.empty?
  end

  def valid_message?
    message.present?
  end
```

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

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

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

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

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

Including ActiveModel::Model allows us to leverage the familiar active record validation syntax.

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

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

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

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

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

  def invite
    if valid?
      deliver_invitations
    end
  end

  private

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

```
      )
    end
  end

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

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

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

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

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

  private

  def survey_inviter_params
    params.require(:survey_inviter).permit(
```

```
      :message,
      :recipients
    ).merge(
      sender: current_user,
      survey: @survey
    )
  end
end
```

Next Steps

- Check that the controller no longer has long methods.
- Verify the new form object is not a large class.
- Check for places to re-use any new validators if extract validator was used during the refactoring.

Introduce Parameter Object

This is 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 the method's grouped parameters with the object.

Uses

- Removes long parameter lists.
- Groups parameters that naturally fit together.
- Encapsulates 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 `full_name` method 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 and shotgun surgery 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 that owns the related state.

To move a method:

- Move the entire method definition and body into the new class.
- Change any parameters that 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

- Removes feature envy by moving a method to the class where the envied methods live.
- Makes private, parameterized methods easier to reuse by moving them to public, unparameterized methods.
- Improves 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 Extract Method chapter 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 smaller 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 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; 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

- Makes classes easier to understand by eliminating the number of methods, classes, and files developers need to look through.
- Eliminates shotgun surgery from changes that cascade through useless classes.
- Eliminates feature envy when the envied class can be inlined into the envious class.

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 eliminate some 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. That isn't a huge gain, but this was an easy refactoring, and continually deleting or inlining unnecessary classes and methods will have broader 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 is 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 simplifies changing 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.
- Makes it easier to avoid subclassing, following composition over inheritance.
- Extend existing classes without modifying them, following the open/closed principle.

- Avoids burdening classes with the knowledge of constructing their own dependencies, following the single responsibility principle.

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, 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 remedy this 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.

By default, 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. 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
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 Extract Decorator chapter.

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 what 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, in an inheritance-based model, would require converting to a new type.

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

Uses

- Eliminates large classes by splitting up a bloated base class.
- Converts STI to a composition-based scheme.
- Makes 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 the 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 first 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 the developer's mental space.

In this example, we'll 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 parameters:

```
# 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 no longer a mixed command and query method (i.e., it does something and returns something), but simply a command method (i.e., it just does 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 less likely that concerns leak between the two.

- Shared behavior happens via composition, making it less likely that the base class will become 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. Regarding situations in which 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 currently difficult changes will be easier to make in the new version.

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

Answering these 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

- Liberates business logic trapped in mixins.
- Eliminates name clashes from multiple mixins.
- Makes methods in the mixins easier to 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 AbstractController::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 a composition.

First, we'll extract a new class for the mixin:

```
# app/models/invitation_message.rb
class InvitationMessage < AbstractController::Base
  include AbstractController::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 that 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 of 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 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 meta-programming 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

- Eliminates case statements by finding classes by name.
- Eliminates shotgun surgery by removing the need to register or configure new strategies and services.
- Eliminates duplicated code by removing manual associations from identifiers to class names.
- Prevents future duplication, making it easier to avoid duplication.

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
```

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, users 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, while this class-based approach is convenient when developing an application, it's 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.

Part III

Principles

DRY

The DRY principle—short for “don’t repeat yourself”—comes from The Pragmatic Programmer.

This 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 toward duplicated code and can be avoided by following the DRY principle:

- Shotgun surgery caused by changing the same knowledge in several places.
- Long parameter lists caused by not encapsulating related properties.
- Feature envy caused by leaking internal knowledge of a class that can be encapsulated and reused.

Making behavior easy to reuse is essential to avoiding duplication. Developers won’t be tempted to copy and paste something that’s easy to reuse through

a small, easy to understand class or method. You can use these solutions to make knowledge easier to reuse:

- Extract classes to encapsulate knowledge, allowing it to be reused.
- Extract methods to reuse behavior within a class.
- Extract partials 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. To effectively 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

The Single Responsibility Principle, often abbreviated as "SRP," was introduced by Uncle Bob Martin, and states:

> A class should have only one reason to change.

Classes with fewer responsibilities are more likely to be reusable, easier to understand and faster to test. They are easy to change and require fewer changes after being written.

Although this appears to be a very simple principle, deciding whether or not any two pieces of behavior introduce two reasons to change is difficult, and obeying SRP rigidly can be frustrating.

Reasons to Change

One of the challenges in identifying reasons to change is deciding what granularity to be concerned with.

In our example application, users can invite their friends to take surveys. When an invitation is sent, we encapsulate that invitation in a basic ActiveRecord subclass:

```
# app/models/invitation.rb
class Invitation < ActiveRecord::Base
  EMAIL_REGEX = /\A([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\z/i
  STATUSES = %w(pending accepted)

  belongs_to :sender, class_name: 'User'
  belongs_to :survey

  before_create :set_token

  validates :recipient_email, presence: true, format: EMAIL_REGEX
  validates :status, inclusion: { in: STATUSES }

  def to_param
    token
  end

  def deliver
    body = InvitationMessage.new(self).body
    Mailer.invitation_notification(self, body).deliver
  end

  private

  def set_token
    self.token = SecureRandom.urlsafe_base64
  end
end
```

Everything in this class has something to do with invitations. We could make the blunt assessment that this class obeys SRP, because it will only change when invitation-related functionality changes. However, looking more carefully at how invitations are implemented, several other reasons to change can be identified:

- The format of invitation tokens changes.
- A bug is identified in our validation of email addresses.

- We need to deliver invitations using some mechanism other than email.
- Invitations need to be persisted in another way, such as in a NoSQL database.
- The API for ActiveRecord or ActiveSupport changes during an update.
- The application switches from Rails to a new framework.

That gives us half a dozen reasons this class might change, leading to the probable conclusion that this class does not follow SRP. So, should this class be refactored?

Stability

Not all reasons to change are created equal.

As a developer, you can anticipate likely changes based on your experience—or just common sense. For example, attributes and business rules for invitations are likely to change, so we know that this class will change as invitations evolve in the application.

Regular expressions are powerful but tricky beasts, so it's likely that we'll have to adjust our regular expression. It might be nice to encapsulate that somewhere else, such as in a custom validator.

It's not always helpful to speculate as to what delivery mechanisms may loom in the distant future, but it's not out of the realm of possibility that we'll need to send messages using an internal private messaging system, or another service like Facebook or Twitter. Therefore, it may be worthwhile to use dependency injection to remove the details of delivery from this model. This may also make testing easier and make the class easier to understand as a unit, because it will remove distracting details relating to email delivery.

NoSQL databases have their uses, but we have no reason to believe we'll ever need to move these records into another type of database. ActiveRecord has proven to be a safe and steady default choice, so it's probably not worth the effort to protect ourselves against that change.

Some of our business logic is expressed using APIs from libraries that could change, such as validations and relationships. We could write our own adapter

to protect ourselves from those changes, but the maintenance burden is unlikely to be worth the benefit, and it will make the code harder to understand, since there will be unnecessary indirection between the model and the framework.

Lastly, we could protect our application against framework changes by preventing any business logic from leaking into the framework classes, such as controllers and ActiveRecord models. Again, this would add a thick layer of indirection to protect against an unlikely change.

However, if you're trying out a new database, object-relational mapper or framework, it may be worth adding some increased protection. The first time you use a new database, you may not be fully confident regarding that decision. Preventing any business logic from mixing with the persistence logic will make it easier to undo that decision and revert to a familiar solution like ActiveRecord in case the new database turns against you.

The less confident you are about a decision, the more you should isolate that decision from the rest of your application.

Cohesion

One of the primary goals of SRP is to promote cohesive classes. The more closely related the methods and properties are to each other, the more cohesive a class is.

Classes with high cohesion are easier to understand, because the pieces fit naturally together. They're also easier to change and reuse, because they won't be coupled to any unexpected dependencies.

Following this principle will lead to high cohesion, but it's important to focus on the output of each change made to follow the principle. If you notice an extra responsibility in a class, think about the benefits of extracting that responsibility. If you think noticeably higher cohesion will be the result, charge ahead. If you think it will simply be a way to spend an afternoon, make a note of it and move on.

Responsibility Magnets

Every application develops a few black holes that like to suck up as much responsibility as possible, slowly turning into God classes.

`User` is a common responsibility magnet. Generally, each application has a focal point in its user interface that sucks up responsibility as well. Our example application's main feature allows users to answer questions on surveys, so `Survey` is a natural junk drawer for behavior.

It's easy to get sucked into a responsibility magnet by falling prey to "Just-One-More Syndrome." Whenever you're about to add a new behavior to an existing class, first check the history of that class. If there are previous commits that show developers attempting to pull functionality out of this class, chances are good that it's a responsibility over-eater. Don't feed the problem; add a new class instead.

Tension with Tell, Don't Ask

Extracting reasons to change can make it harder to follow tell, don't ask.

For example, consider a `Purchase` model that knows how to charge a user:

```
class Purchase
  def charge
    purchaser.charge_credit_card(total_amount)
  end
end
```

This method follows tell, don't ask, because we can simply tell any `Purchase` to `charge`, without examining any state on the `Purchase`.

However, it violates the SRP, because `Purchase` has more than one reason to change. If the rules around charging credit cards change or the rules for calculating purchase totals change, this class will have to change.

You can more closely adhere to SRP by extracting a new class for the `charge` method:

```
class PurchaseProcessor
  def initialize(purchase, purchaser)
    @purchase = purchase
    @purchaser = purchaser
  end

  def charge
    @purchaser.charge_credit_card @purchase.total_amount
  end
end
```

This class can encapsulate rules around charging credit cards and remain immune to other changes, thus following SRP. However, it now violates tell, don't ask, because it must ask the `@purchase` for its `total_amount` in order to place the charge.

These two principles are often at odds with each other and you must make a pragmatic decision about which direction works best for your own classes.

Drawbacks

There are a number of drawbacks to following this principle too rigidly:

- As outlined above, following this principle may lead to violations of tell, don't ask.
- This principle causes an increase in the number of classes, potentially leading to shotgun surgery and vocabulary overload.
- Classes that follow this principle may introduce additional indirection, making it harder to understand high-level behavior by looking at individual classes.

Application

If you find yourself fighting any of these smells, you may want to refactor to follow the SRP:

- Divergent change doesn't exist in classes that follow this principle.
- Classes following this principle are easy to reuse, reducing the likelihood of Duplicated code.
- Large classes almost certainly have more than one reason to change. Following this principle eliminates most large classes.

Code containing these smells may need refactoring before it can follow this principle:

- Case statements make this principle difficult to follow, as every case statement introduces a new reason to change.
- Long methods make it harder to extract concerns, as behavior can only be moved once it's encapsulated in a small, cohesive method.
- Mixins, Single-table inheritance, and inheritance in general make it harder to follow this principle, as the boundary between parent and child class responsibilities is always fuzzy.

These solutions may be useful on the path towards SRP:

- Extract classes to move responsibilities to their own class.
- Extract decorators to layer responsibilities onto existing classes without burdening the class definition with that knowledge.
- Extract validators to prevent classes from changing when validation rules change.
- Extract value objects to prevent rules about a type like currency or names from leaking into other business logic.
- Extract methods to make responsibilities easier to move.
- Move methods to place methods in a more cohesive environment.
- Inject dependencies to relieve classes of the burden of changing with their dependencies.
- Replace mixins with composition to make it easier to isolate concerns.
- Replace subclasses with strategies to make variations usable without their base logic.

Following composition over inheritance and the dependency inversion principle may make SRP easier to follow, as those principles make it easier to extract

responsibilities. Following this principle will make it easier to follow the open-closed principle but may introduce violations of tell, don't ask.

Tell, Don’t Ask

The Tell, Don’t Ask principle advises developers to tell objects what they want done, rather than querying objects and making decisions for them.

Consider the following example:

```
class Order
  def charge(user)
    if user.has_valid_credit_card?
      user.charge(total)
    else
      false
    end
  end
end
```

This example doesn’t follow tell, don’t ask. It first asks if the user has a valid credit card, and then makes a decision based on the user’s state.

In order to follow tell, don’t ask, we can move this decision into `User#charge`:

```
class User
  def charge(total)
    if has_valid_credit_card?
      payment_gateway.charge(credit_card, total)
      true
    else
      false
    end
  end
end
```

Now `Order#charge` can simply delegate to `User#charge`, passing its own relevant state (`total`):

```
class Order
  def charge(user)
    user.charge(total)
  end
end
```

Following this principle has a number of benefits, outlined below.

Encapsulation of Logic

Following tell, don't ask encapsulates the conditions under which an operation can be performed in one place. In the above example, `User` should know when it makes sense to `charge`.

Encapsulation of State

Referencing another object's state directly couples two objects together based on what they *are*, rather than on what they *do*. By following tell, don't ask, we encapsulate state within the object that uses it, exposing only the operations that can be performed based on that state and hiding the state itself within private methods and instance variables.

Minimal Public Interface

In many cases, following tell, don't ask will result in the smallest possible public interface between classes. In the above example, `has_valid_credit_card?` can now be made private, because it becomes an internal concern encapsulated within `User`.

Public methods are a liability. Before they can be changed, moved, renamed or removed, you will need to find every consumer class and update each one accordingly.

Tension with Model—View—Controller

This principle can be difficult to follow while also following Model—View—Controller (MVC).

Consider a view that uses the above `Order` model:

```
<%= form_for @order do |form| %>
  <% unless current_user.has_valid_credit_card? %>
    <%= render 'credit_card/fields', form: form %>
  <% end %>
  <!-- Order Fields -->
<% end %>
```

The view doesn't display the credit card fields if the user already has a valid credit card saved. The view needs to ask the user a question and then change its behavior based on that question, violating tell, don't ask.

You could obey tell, don't ask by making the user know how to render the credit card form:

```
<%= form_for @order do |form| %>
  <%= current_user.render_credit_card_form %>
  <!-- Order Fields -->
<% end %>
```

However, this violates MVC by including view logic in the `User` model. In this case, it's better to keep the model, view and controller concerns separate and step across the tell, don't ask line.

When writing interactions between other models and support classes, though, make sure to give commands whenever possible and avoid deviations in behavior based on another class's state.

Application

These smells may be a sign that you should be following tell, don't ask more:

- Feature envy is frequently a sign that a method or part of a method should be extracted and moved to another class, to reduce the number of questions that method must ask of another object.
- Shotgun surgery may result from state and logic leaks. Consolidating conditionals using tell, don't ask may reduce the number of changes required for new functionality.

If you find classes with these smells, they may require refactoring before you can follow tell, don't ask:

- Case statements that inflect on methods from another object generally get in the way.
- Mixins blur the lines between responsibilities, as mixed in methods operate on the state of the objects they're mixed into.

If you're trying to refactor classes to follow tell, don't ask, these solutions may be useful:

- Extract method to encapsulate multiple conditions into one.
- Move method to move methods closer to the state they operate on.
- Inline class to remove unnecessary questions between two classes with highly cohesive behavior.
- Replace conditionals with polymorphism to reduce the number of questions being asked around a particular operation.

- Replace conditionals with null object to remove checks for `nil`.

Many Law of Demeter violations point toward violations of tell, don't ask. Following tell, don't ask may lead to violations of the single responsibility principle and the open/closed principle, since moving operations onto the best class may require modifying an existing class and adding a new responsibility.

Law of Demeter

The Law of Demeter was developed at Northeastern University. It's named after the Demeter Project, which was named after Demeter, the Greek goddess of the harvest. There is widespread disagreement as to its pronunciation, but the correct pronunciation emphasizes the second syllable; you can trust us on that.

This principle states that:

> A method of an object should invoke only the methods of the following kinds of objects:
>
> 1. itself
> 2. its parameters
> 3. any objects it creates/instantiates
> 4. its direct component objects

Like many principles, the Law of Demeter is an attempt to help developers manage dependencies. The law restricts how deeply a method can reach into another object's dependency graph, preventing any one method from becoming tightly coupled to another object's structure.

Multiple Dots

The most obvious violation of the Law of Demeter is "multiple dots," meaning a chain of methods being invoked on each others' return values.

Example:

```
class User
  def discounted_plan_price(discount_code)
    coupon = Coupon.new(discount_code)
    coupon.discount(account.plan.price)
  end
end
```

The call to `account.plan.price` above violates the Law of Demeter by invoking `price` on the return value of `plan`. The `price` method is not a method on `User`, its parameter `discount_code`, its instantiated object `coupon` or its direct component `account`.

The quickest way to avoid violations of this nature is to delegate the method:

```
class User
  def discounted_plan_price(discount_code)
    account.discounted_plan_price(discount_code)
  end
end

class Account
  def discounted_plan_price(discount_code)
    coupon = Coupon.new(discount_code)
    coupon.discount(plan.price)
  end
end
```

In a Rails application, you can quickly delegate methods using ActiveSupport's `delegate` class method:

```
class User
  delegate :discounted_plan_price, to: :account
end
```

If you find yourself writing lots of delegators, consider changing the consumer class to take a different object. For example, if you need to delegate numerous `User` methods to `Account`, it's possible that the code referencing `User` should actually reference an instance of `Account` instead.

Multiple Assignments

Law of Demeter violations are often hidden behind multiple assignments.

```
class User
  def discounted_plan_price(discount_code)
    coupon = Coupon.new(discount_code)
    plan = account.plan
    coupon.discount(plan.price)
  end
end
```

The above `discounted_plan_price` method no longer has multiple dots on one line, but it still violates the Law of Demeter, because `plan` isn't a parameter, instantiated object or direct subcomponent.

The Spirit of the Law

Although the letter of the Law of Demeter is rigid, its message is broader. The fundamental goal is to avoid over-entangling a method with another object's dependencies.

This means that fixing a violation shouldn't be your objective; removing the problem that *caused* the violation is a better idea. Here are a few tips to avoid misguided fixes to Law of Demeter violations:

- Many delegate methods to the same object are an indicator that your object graph may not accurately reflect the real-world relationships they represent.
- Delegate methods with prefixes (`Post#author_name`) are fine, but it's worth a check to see if you can remove the prefix. If not, make sure you didn't actually want a reference to the prefix object (`Post#author`).
- Avoid multiple prefixes for delegate methods, such as `User#account_plan_price`.
- Avoid assigning to instance variables to work around violations.

Objects vs. Types

The version of the Law quoted at the beginning of this chapter is the "object formulation," from the original paper. The first formulation was expressed in terms of types:

> For all classes C, and for all methods M attached to C, all objects to which M sends a message must be instances of classes associated with the following classes:
>
> 1. The argument classes of M (including C).
> 2. The instance variable classes of C.
>
> (Objects created by M, or by functions or methods which M calls, and objects in global variables are considered as arguments of M.)

This formulation allows some more freedom when chaining using a fluent syntax. Essentially, it allows chaining as long as each step of the chain returns the same type.

Examples:

```
# Mocking APIs
user.should_receive(:save).once.and_return(true)

# Ruby's Enumerable
users.select(&:active?).map(&:name)

# String manipulation
collection_name.singularize.classify.constantize

# ActiveRecord chains
users.active.without_posts.signed_up_this_week
```

Duplication

The Law of Demeter is related to the DRY principle, in that Law of Demeter violations frequently duplicate knowledge of dependencies.

Example:

```
class CreditCardsController < ApplicationController
  def charge_for_plan
    if current_user.account.credit_card.valid?
      price = current_user.account.plan.price
      current_user.account.credit_card.charge price
    end
  end
end
```

In this example, the knowledge that a user has a credit card through its account is duplicated. That knowledge is declared somewhere in the `User` and `Account` classes when the relationship is defined, and knowledge of it then spreads to two more locations in `charge_for_plan`.

Like most duplication, each instance isn't too harmful; but in aggregate, duplication will slowly make refactoring become impossible.

Application

The following smells may cause or result from Law of Demeter violations:

- Feature envy from methods that reach through a dependency chain multiple times.
- Shotgun surgery resulting from changes in the dependency chain.

You can use these solutions to follow the Law of Demeter:

- Move methods that reach through a dependency to the owner of that dependency.
- Inject dependencies so that methods have direct access to the dependencies that they need.
- Inline class if it adds hops to the dependency chain without providing enough value.

Composition Over Inheritance

In class-based object-oriented systems, composition and inheritance are the two primary methods of reusing and assembling components. Composition Over Inheritance suggests that, when there isn't a strong case for using inheritance, developers implement reuse and assembly using composition instead.

Let's look at a simple example implemented using both composition and inheritance. In our example application, users can invite their friends to take surveys. Users can be invited using either an email or an internal private message. Each delivery strategy is implemented using a separate class.

Inheritance

In the inheritance model, we use an abstract base class called `Inviter` to implement common invitation-sending logic. We then use `EmailInviter` and `MessageInviter` subclasses to implement the delivery details.

```ruby
# app/models/inviter.rb
class Inviter < AbstractController::Base
  include AbstractController::Rendering
  include Rails.application.routes.url_helpers

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

  private

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


# app/models/email_inviter.rb
class EmailInviter < Inviter
  def initialize(invitation)
    @invitation = invitation
  end

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

```
# app/models/message_inviter.rb
class MessageInviter < 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
```

Note that there is no clear boundary between the base class and the subclasses. The subclasses access reusable behavior by invoking private methods inherited from the base class, like `render_message_body`.

Composition

In the composition model, we use a concrete `InvitationMessage` class to implement common invitation-sending logic. We then use that class from `EmailInviter` and `MessageInviter` to reuse the common behavior, and the inviter classes implement delivery details.

```
# app/models/invitation_message.rb
class InvitationMessage < AbstractController::Base
  include AbstractController::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

# 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

# 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
```

Note that there is now a clear boundary between the common behavior in `InvitationMessage` and the variant behavior in `EmailInviter` and `MessageInviter`.

The inviter classes access reusable behavior by invoking public methods like `body` on the shared class.

Dynamic vs. Static

Although the two implementations are fairly similar, one difference between them is that, in the inheritance model, the components are assembled statically. The composition model, on the other hand, assembles the components dynamically.

Ruby is not a compiled language and everything is evaluated at run-time, so claiming that anything is assembled statically may sound like nonsense. However, there are several ways in which inheritance hierarchies are essentially written in stone, or static:

- You can't swap out a superclass once it's assigned.
- You can't easily add and remove behaviors after an object is instantiated.
- You can't inject a superclass as a dependency.
- You can't easily access an abstract class's methods directly.

On the other hand, everything in a composition model is dynamic:

- You can easily change out a composed instance after instantiation.
- You can add and remove behaviors at any time using decorators, strategies, observers and other patterns.
- You can easily inject composed dependencies.
- Composed objects aren't abstract, so you can use their methods anywhere.

Dynamic Inheritance

There are very few rules in Ruby, so many of the restrictions that apply to inheritance in other languages can be worked around in Ruby. For example:

- You can reopen and modify classes after they're defined, even while an application is running.
- You can extend objects with modules after they're instantiated to add behaviors.
- You can call private methods by using `send`.
- You can create new classes at run-time by calling `Class.new`.

These features make it possible to overcome some of the rigidity of inheritance models. However, performing all of these operations is simpler with objects than it is with classes, and doing too much dynamic type definition will make the application harder to understand, by diluting the type system. After all, if none of the classes are ever fully formed, what does a class represent?

The Trouble with Hierarchies

Using subclasses introduces a subtle problem into your domain model: It assumes that your models follow a hierarchy; that is, it assumes that your types fall into a tree-like structure.

Continuing with the above example, we have a root type, `Inviter`, and two subtypes, `EmailInviter` and `MessageInviter`. What if we want invitations sent by admins to behave differently than invitations sent by normal users? We can create an `AdminInviter` class, but what will its superclass be? How will we combine it with `EmailInviter` and `MessageInviter`? There's no easy way to combine email, message and admin functionality using inheritance, so you'll end up with a proliferation of conditionals.

Composition, on the other hand, provides several ways out of this mess, such as using a decorator to add admin functionality to the inviter. Once you build objects with a reasonable interface, you can combine them endlessly with minimal modification to the existing class structure.

Mixins

Mixins are Ruby's answer to multiple inheritance.

However, mixins need to be mixed into a class before they can be used. Unless you plan on building dynamic classes at runtime, you'll need to create a class for each possible combination of modules. This will result in a ton of little classes, such as `AdminEmailInviter`.

Again, composition provides a clean answer to this problem, because you can create as many anonymous combinations of objects as your little heart desires.

Ruby does allow dynamic use of mixins using the `extend` method. This technique does work, but it has its own complications. Extending an object's type dynamically in this way dilutes the meaning of the word "type," making it harder to understand what an object is. Additionally, using runtime `extend` can lead to performance issues in some Ruby implementations.

Single Table Inheritance

Rails provides a way to persist an inheritance hierarchy, known as Single Table Inheritance, often abbreviated as STI. Using STI, a cluster of subclasses is persisted to the same table as the base class. The name of the subclass is also saved on the row, allowing Rails to instantiate the correct subclass when pulling records back out of the database.

Rails also provides a clean way to persist composed structures using polymorphic associations. Using a polymorphic association, Rails will store both the primary key and the class name of the associated object.

Because Rails provides a clean implementation for persisting both inheritance and composition, the fact that you're using ActiveRecord should have little influence on your decision to design using inheritance versus composition.

Drawbacks

Although composed objects are largely easy to write and assemble, there are situations in which they hurt more than inheritance trees.

- Inheritance cleanly represents hierarchies. If you really do have a hierarchy of object types, use inheritance.

- Subclasses always know what their superclass is, so they're easy to instantiate. If you use composition, you'll need to instantiate at least two objects to get a usable instance: the composing object and the composed object.
- Using composition is more abstract, which means that you need a name for the composed object. In our earlier example, all three classes were "inviters" in the inheritance model, but the composition model introduced the "invitation message" concept. Excessive composition can lead to vocabulary overload.

Application

If you see these smells in your application, they may be a sign that you should switch some classes from inheritance to composition:

- Divergent change caused by frequent leaks into abstract base classes.
- Large classes acting as abstract base classes.
- Mixins serving to allow reuse while preserving the appearance of a hierarchy.

Classes with these smells may be difficult to transition to a composition model:

- Duplicated code will need to be pulled up into the base class before subclasses can be switched to strategies.
- Shotgun surgery may represent tight coupling between base classes and subclasses, making it more difficult to switch to composition.

These solutions will help move from inheritance to composition:

- Extract classes to liberate private functionality from abstract base classes.
- Extract method to make methods smaller and easier to move.
- Move method to slim down bloated base classes.
- Replace mixins with composition to make it easier to dissolve hierarchies.
- Replace subclasses with strategies to implement variations dynamically.

After replacing inheritance models with composition, you'll be free to use these solutions to take your code further:

- Extract decorators to make it easy to add behaviors dynamically.
- Inject dependencies to make it possible to compose objects in new ways.

Following this principle will make it much easier to follow the dependency inversion principle and the open/closed principle.

Open/Closed Principle

The Open/Closed Principle states that:

> Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

The purpose of this principle is to make it possible to change or extend the behavior of an existing class without actually modifying the source code to that class.

Making classes extensible in this way has a number of benefits:

- Every time you modify a class, you risk breaking it, along with all classes that depend on that class. Reducing churn in a class reduces bugs in that class.
- Changing the behavior or interface to a class means that you need to update any classes that depend on the old behavior or interface. Allowing per-use extensions to a class eliminates this domino effect.

Strategies

It may sound appealing to never need to change existing classes again, but achieving this is difficult in practice. Once you've identified an area that keeps changing, there are a few strategies you can use to make it possible to extend

without modifications. Let's go through an example with a few of those strategies.

In our example application, we have an `Invitation` class that can deliver itself to an invited user:

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

However, we need a way to allow users to unsubscribe from these notifications. We have an `Unsubscribe` model that holds the email addresses of users that don't want to be notified.

The most direct way to add this check is to modify `Invitation` directly:

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

However, that would violate the open/closed principle. Let's see how we can introduce this change without violating the principle.

Inheritance

One of the most common ways to extend an existing class without modifying it is to create a new subclass.

We can use a new subclass to handle unsubscriptions:

```
# app/models/unsubscribeable_invitation.rb
class UnsubscribeableInvitation < Invitation
  def deliver
    unless unsubscribed?
      super
    end
  end

  private

  def unsubscribed?
    Unsubscribe.where(email: recipient_email).exists?
  end
end
```

This can be a little awkward when trying to use the new behavior, though. For example, we need to create an instance of this class, even though we want to save it to the same table as `Invitation`:

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

This works adequately for creation, but using the ActiveRecord pattern, we'll end up with an instance of `Invitation` instead, if we ever reload from the database. That means that inheritance is easiest to use when the class we're extending doesn't require persistence.

Inheritance also requires some creativity in unit tests to avoid duplication.

Decorators

Another way to extend an existing class is to write a decorator.

Using Ruby's `DelegateClass` method, we can quickly create decorators:

```
# app/models/unsubscribeable_invitation.rb
class UnsubscribeableInvitation < DelegateClass(Invitation)
  def deliver
    unless unsubscribed?
      super
    end
  end

  private

  def unsubscribed?
    Unsubscribe.where(email: recipient_email).exists?
  end
end
```

The implementation is extremely similar to the subclass but it can now be applied at run-time to instances of `Invitation`:

```
# app/models/survey_inviter.rb
def deliver_invitations
  create_invitations.each do |invitation|
    UnsubscribeableInvitation.new(invitation).deliver
  end
end
```

The unit tests can also be greatly simplified using stubs.

This makes it easier to combine with persistence. However, Ruby's `DelegateClass` doesn't combine well with ActionPack's polymorphic URLs.

Dependency Injection

This method requires more forethought in the class you want to extend, but classes that follow inversion of control can inject dependencies to extend classes without modifying them.

We can modify our `Invitation` class slightly to allow client classes to inject a mailer:

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

Now we can write a mailer implementation that checks to see if users are unsubscribed before sending them messages:

```
# app/mailers/unsubscribeable_mailer.rb
class UnsubscribeableMailer
  def self.invitation_notification(invitation, body)
    if unsubscribed?(invitation)
      NullMessage.new
    else
      Mailer.invitation_notification(invitation, body)
    end
  end

  private

  def self.unsubscribed?(invitation)
    Unsubscribe.where(email: invitation.recipient_email).exists?
  end

  class NullMessage
    def deliver
    end
  end
end
```

And we can use dependency injection to substitute it:

```
# app/models/survey_inviter.rb
def deliver_invitations
  create_invitations.each do |invitation|
    invitation.deliver(UnsubscribeableMailer)
  end
end
```

Everything is Open

As you've followed along with these strategies, you've probably noticed that although we've found creative ways to avoid modifying `Invitation`, we've had to modify other classes. When you change or add behavior, you need to change or add it somewhere. You can design your code so that most new or changed behavior takes place by writing a new class, but something, somewhere in the existing code will need to reference that new class.

It's difficult to determine what you should attempt to leave open when writing a class. It's hard to know where to leave extension hooks without anticipating every feature you might ever want to write.

Rather than attempting to guess what will require extension in the future, pay attention as you modify existing code. After each modification, check to see if there's a way you can refactor to make similar extensions possible without modifying the underlying class.

Code tends to change in the same ways over and over, so by making each change easy to apply as you need to make it, you're making the next change easier.

Monkey Patching

As a Ruby developer, you probably know that one quick way to extend a class without changing its source code is to use a monkey patch:

```
# app/monkey_patches/invitation_with_unsubscribing.rb
Invitation.class_eval do
  alias_method :deliver_unconditionally, :deliver

  def deliver
    unless unsubscribed?
      deliver_unconditionally
    end
  end

  private

  def unsubscribed?
    Unsubscribe.where(email: recipient_email).exists?
  end
end
```

Although monkey patching doesn't literally modify the class's source code, it does modify the existing class. That means that you risk breaking it, and, potentially, all classes that depend on it. Since you're changing the original behavior, you'll also need to update any client classes that depend on the old behavior.

In addition to all the drawbacks of directly modifying the original class, monkey patches also introduce confusion, as developers will need to look in multiple locations to understand the full definition of a class.

In short, monkey patching has most of the drawbacks of modifying the original class without any of the benefits of following the open/closed principle.

Drawbacks

Although following the open/closed principle will make code easier to change, it may make it more difficult to understand. This is because the gained flexibility requires introducing indirection and abstraction. Although all of the three strategies outlined in this chapter are more flexible than the original change, directly modifying the class is the easiest to understand.

This principle is most useful when applied to classes with high reuse and potentially high churn. Applying it everywhere will result in extra work and more obscure code.

Application

If you encounter the following smells in a class, you may want to begin following this principle:

- Divergent change caused by a lack of extensibility.
- Large classes and long methods which can be eliminated by extracting and injecting dependent behavior.

You may want to eliminate the following smells if you're having trouble following this principle:

- Case statements make it hard to obey this principle, as you can't add to the case statement without modifying it.

You can use the following solutions to make code more compliant with this principle:

- Extract decorator to extend existing classes without modification.
- Inject dependencies to allow future extensions without modification.

Dependency Inversion Principle

The Dependency Inversion Principle, sometimes abbreviated as “DIP,” was created by Uncle Bob Martin.

The principle states:

> A. High-level modules should not depend on low-level modules. Both should depend on abstractions.
>
> B. Abstractions should not depend upon details. Details should depend upon abstractions.

This is a very technical way of proposing that developers invert control.

Inversion of Control

Inversion of control is a technique for keeping software flexible. It combines best with small classes with single responsibilities. Inverting control means assigning dependencies at run-time, rather than statically referencing dependencies at each level.

This can be hard to understand as an abstract concept, but it’s fairly simple in practice. Let’s jump into an example:

```ruby
# 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

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

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

The `summaries_using` method builds a summary of the answers to each of the survey's questions.

However, we also want to hide the answers to questions that the user has not personally answered, so we decorate the `summarizer` with an `UnansweredQuestionHider`. Note that we're statically referencing the concrete, lower-level detail `UnansweredQuestionHider` from `Survey` rather than depending on an abstraction.

In the current implementation, the `Survey#summaries_using` method will need to change whenever something changes about the summaries. For example, hiding the unanswered questions requires changes to this method.

Also, note that the conditional logic is spread across several layers. `SummariesController` decides whether or not to hide unanswered questions. That knowledge is passed into `Survey#summaries_using`. `SummariesController`

also passes the current user down into `Survey#summaries_using`, and from there it's passed into `UnansweredQuestionHider`:

```
# 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)
    @user && !question.answered_by?(@user)
  end
end
```

We can make future changes like this easier by inverting control:

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

# 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
```

Now the `Survey#summaries_using` method is completely ignorant of answer hiding; it simply accepts a `summarizer` and the client (`SummariesController`) injects a decorated dependency. This means that adding similar changes won't require changing the `Summary` class at all.

This also allows us to simplify `UnansweredQuestionHider` by removing a condition:

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

We no longer build `UnansweredQuestionHider` when a user isn't signed in, so we don't need to check for a user.

Where To Decide Dependencies

While following the previous example, you probably noticed that we didn't eliminate the `UnansweredQuestionHider` dependency; we just moved it around. This means that, while adding new summarizers or decorators won't affect `Summary`, they will affect `SummariesController` in the current implementation. So, did we actually make anything better?

In this case, the code was improved because the information that affects the dependency decision—`params[:unanswered]`—is now closer to where we make the decision. Before, we needed to pass a Boolean down into `summaries_using`, causing that decision to leak across layers.

If you push your dependency decisions up until they reach the layer that contains the information needed to make those decisions, you will prevent changes from affecting several layers.

Drawbacks

Following this principle results in more abstraction and indirection, as it's often difficult to tell which class is being used for a dependency.

Looking at the example above, it's now impossible to know in `summaries_using` which class will be used for the `summarizer`:

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

This makes it difficult to know exactly what's going to happen. You can mitigate this issue by using naming conventions and well-named classes.

However, each abstraction introduces more vocabulary into the application, making it more difficult for new developers to learn the domain.

Application

If you identify these smells in an application, you may want to adhere more closely to the dependency inversion principle (DIP):

- Following DIP can eliminate shotgun surgery by consolidating dependency decisions.
- Code suffering from divergent change may improve after having some of its dependencies injected.
- Large classes and long methods can be reduced by injecting dependencies, as this will outsource dependency resolution.

You may need to eliminate these smells in order to properly invert control:

- Excessive use of callbacks will make it harder to follow the DIP, because it's harder to inject dependencies into a callback.
- Using mixins and STI for reuse will make following the DIP more difficult, because inheritance is always decided statically. Because a class can't decide its parent class at run-time, inheritance can't follow inversion of control.

You can use these solutions to refactor towards DIP-compliance:

- Inject dependencies to invert control.
- Use extract class to make smaller classes that are easier to compose and inject.
- Use extract decorator to make it possible to package a decision that involves multiple classes and inject it as a single dependency.
- Replace callbacks with methods to make dependency injection easier.
- Replace conditional with polymorphism to make dependency injection easier.

- Replace mixin with composition and replace subclasses with strategies to make it possible to decide dependencies abstractly at run-time.
- Use class as factory to make it possible to abstractly instantiate dependencies without knowing which class is being used and without writing abstract factory classes.

Following the single responsibility principle and composition over inheritance will make it easier to follow the dependency inversion principle. Following this principle will make it easier to obey the open/closed principle.