

7 Patterns to Refactor Fat ActiveRecord Models

Posted by [@brynary](#) on Oct 17th, 2012

 [Tweet](#) 1,205

When teams use [Code Climate](#) to improve the quality of their Rails applications, they learn to break the habit of allowing models to get fat. “*Fat models*” cause maintenance issues in large apps. Only incrementally better than cluttering controllers with domain logic, they usually represent a failure to apply the [Single Responsibility Principle](#) (SRP). “Anything related to what a user does” is not a *single* responsibility.

Early on, SRP is easier to apply. ActiveRecord classes handle persistence, associations and not much else. But bit-by-bit, they grow. Objects that are inherently responsible for persistence become the *de facto* owner of all business logic as well. And a year or two later you have a `User` class with over 500 lines of code, and hundreds of methods in its *public* interface. Callback hell ensues.

As you add more intrinsic complexity (read: features!) to your application, the goal is to spread it across a coordinated set of small, encapsulated objects (and, at a higher level, modules) just as you might spread cake batter across the bottom of a pan. Fat models are like the big clumps you get when you first pour the batter in. Refactor to break them down and spread out the logic evenly. Repeat this process and you’ll end up with a set of simple objects with well defined interfaces working together in a veritable symphony.

You may be thinking:

“**But Rails makes it really hard to do OOP right!**”

I used to believe that too. But after some exploration and practice, I realized that Rails (the framework) doesn’t impede OOP all that much. It’s the Rails “conventions” that don’t scale, or, more specifically, the lack of conventions for managing complexity beyond what the [Active Record pattern](#) can elegantly handle. Fortunately, we can apply OO-based principles and best practices where Rails is lacking.

Don’t Extract Mixins from Fat Models

Let’s get this out of the way. I discourage pulling sets of methods out of a large ActiveRecord class into “concerns”, or modules that are then mixed in to only one model. I once heard someone say:

“Any application with an **app/concerns** directory is concerning.”

And I agree. *Prefer composition to inheritance*. Using mixins like this is akin to “cleaning” a messy room by dumping the clutter into six separate junk drawers and slamming them shut. Sure, it looks cleaner at the surface, but the junk drawers actually make it harder to identify and implement the decompositions and extractions necessary to clarify the domain model.

Code Climate Blog

Thoughts about security, refactoring, object oriented design, test-driven development and design patterns with a focus on Ruby.

[@codeclimate](#)

[RSS](#)

Code Climate helps you build better software by providing automated code review for Ruby and Javascript.

[Learn More](#)



Popular Posts

[7 Patterns to Refactor Fat ActiveRecord Models](#)

[Rails’ Insecure Defaults](#)

[Your Objects, the Unix Way](#)

[Sublime Text 2 for Ruby](#)

Now on to the refactorings!

1. Extract Value Objects

[Value Objects](#) are simple objects whose equality is dependent on their value rather than an identity.

They are usually immutable. `Date`, `URI`, and `Pathname` are examples from Ruby's standard library, but your application can (and almost certainly should) define domain-specific Value Objects as well.

Extracting them from ActiveRecord is low hanging refactoring fruit.

In Rails, Value Objects are great when you have an attribute or small group of attributes that have logic associated with them. Anything more than basic text fields and counters are candidates for Value Object extraction.

For example, a text messaging application I worked on had a `PhoneNumber` Value Object. An e-commerce application needs a `Money` class. Code Climate has a Value Object named `Rating` that represents a simple A - F grade that each class or module receives. I could (and originally did) use an instance of a Ruby `String`, but `Rating` allows me to combine behavior with the data:

```
1  class Rating
2    include Comparable
3
4    def self.from_cost(cost)
5      if cost <= 2
6        new("A")
7      elsif cost <= 4
8        new("B")
9      elsif cost <= 8
10       new("C")
11      elsif cost <= 16
12       new("D")
13      else
14       new("F")
15      end
16    end
17
18    def initialize(letter)
19      @letter = letter
20    end
21
22    def better_than?(other)
23      self > other
24    end
25
26    def <=>(other)
27      other.to_s <=> to_s
28    end
29
30    def hash
31      @letter.hash
32    end
33
34    def eql?(other)
35      to_s == other.to_s
36    end
37
38    def to_s
39      @letter.to_s
40    end
41  end
```

Every `ConstantSnapshot` then exposes an instance of `Rating` in its public interface:

```
1  class ConstantSnapshot < ActiveRecord::Base
2    # ...
3
```

```
4     def rating
5       @rating ||= Rating.from_cost(cost)
6     end
7   end
```

Beyond slimming down the ConstantSnapshot class, this has a number of advantages:

- The `#worse_than?` and `#better_than?` methods provide a more expressive way to compare ratings than Ruby’s built-in operators (e.g. `<` and `>`).
- Defining `#hash` and `#eq?` makes it possible to use a `Rating` as a hash key. Code Climate uses this to idiomatically group constants by their ratings using `Enumerable#group_by`.
- The `#to_s` method allows me to interpolate a `Rating` into a string (or template) without any extra work.
- The class definition provides a convenient place for a factory method, returning the correct `Rating` for a given “remediation cost” (the estimated time it would take to fix all of the smells in a given class).

2. Extract Service Objects

Some actions in a system warrant a Service Object to encapsulate their operation. I reach for Service Objects when an action meets one or more of these criteria:

- The action is complex (e.g. closing the books at the end of an accounting period)
- The action reaches across multiple models (e.g. an e-commerce purchase using `Order`, `CreditCard` and `Customer` objects)
- The action interacts with an external service (e.g. posting to social networks)
- The action is not a core concern of the underlying model (e.g. sweeping up outdated data after a certain time period).
- There are multiple ways of performing the action (e.g. authenticating with an access token or password). This is the Gang of Four [Strategy pattern](#).

As an example, we could pull a `User#authenticate` method out into a `UserAuthenticator`:

```
1  class UserAuthenticator
2    def initialize(user)
3      @user = user
4    end
5
6    def authenticate(unencrypted_password)
7      return false unless @user
8
9      if BCrypt::Password.new(@user.password_digest) == unencrypted_password
10        @user
11      else
12        false
13      end
14    end
15  end
```

And the `SessionsController` would look like this:

```
1  class SessionsController < ApplicationController
2    def create
3      user = User.where(email: params[:email]).first
4
5      if UserAuthenticator.new(user).authenticate(params[:password])
6        self.current_user = user
7        redirect_to dashboard_path
8      else
9        flash[:alert] = "Login failed."
10        render "new"
11      end
12    end
13  end
```

12	end
13	end

3. Extract Form Objects

When multiple ActiveRecord models might be updated by a single form submission, a Form Object can encapsulate the aggregation. This is far cleaner than using [accepts_nested_attributes_for](#), which, in my humble opinion, should be deprecated. A common example would be a signup form that results in the creation of both a Company and a User:

```
1  class Signup
2    include Virtus
3
4    extend ActiveRecord::Naming
5    include ActiveRecord::Conversion
6    include ActiveRecord::Validations
7
8    attr_reader :user
9    attr_reader :company
10
11    attribute :name, String
12    attribute :company_name, String
13    attribute :email, String
14
15    validates :email, presence: true
16    # ... more validations ...
17
18    # Forms are never themselves persisted
19    def persisted?
20      false
21    end
22
23    def save
24      if valid?
25        persist!
26        true
27      else
28        false
29      end
30    end
31
32    private
33
34    def persist!
35      @company = Company.create!(name: company_name)
36      @user = @company.users.create!(name: name, email: email)
37    end
38  end
```

I’ve started using [Virtus](#) in these objects to get ActiveRecord-like attribute functionality. The Form Object quacks like an ActiveRecord, so the controller remains familiar:

```
1  class SignupsController < ApplicationController
2    def create
3      @signup = Signup.new(params[:signup])
4
5      if @signup.save
6        redirect_to dashboard_path
7      else
8        render "new"
9      end
10    end
11  end
```

This works well for simple cases like the above, but if the persistence logic in the form gets too complex

you can combine this approach with a Service Object. As a bonus, since validation logic is often contextual, it can be defined in the place exactly where it matters instead of needing to guard validations in the ActiveRecord itself.

4. Extract Query Objects

For complex SQL queries littering the definition of your ActiveRecord subclass (either as scopes or class methods), consider extracting query objects. Each query object is responsible for returning a result set based on business rules. For example, a Query Object to find abandoned trials might look like this:

```
1 class AbandonedTrialQuery
2   def initialize(relation = Account.scoped)
3     @relation = relation
4   end
5
6   def find_each(&block)
7     @relation.
8       where(plan: nil, invites_count: 0).
9       find_each(&block)
10  end
11 end
```

You might use it in a background job to send emails:

```
1 AbandonedTrialQuery.new.find_each do |account|
2   account.send_offer_for_support
3 end
```

Since ActiveRecord::Relation instances are first class citizens as of Rails 3, they make a great input to a Query Object. This allows you to combine queries using composition:

```
1 old_accounts = Account.where("created_at < ?", 1.month.ago)
2 old_abandoned_trials = AbandonedTrialQuery.new(old_accounts)
```

Don’t bother testing a class like this in isolation. Use tests that exercise the object and the database together to ensure the correct rows are returned in the right order and any joins or eager loads are performed (e.g. to avoid N + 1 query bugs).

5. Introduce View Objects

If logic is needed purely for display purposes, it does not belong in the model. Ask yourself, “If I was implementing an alternative interface to this application, like a voice-activated UI, would I need this?”. If not, consider putting it in a helper or (often better) a View object.

For example, the donut charts in Code Climate break down class ratings based on a snapshot of the codebase (e.g. [Rails on Code Climate](#)) and are encapsulated as a View:

```
1 class DonutChart
2   def initialize(snapshot)
3     @snapshot = snapshot
4   end
5
6   def cache_key
7     @snapshot.id.to_s
8   end
9
10  def data
11    # pull data from @snapshot and turn it into a JSON structure
12  end
13 end
```

often find a one-to-one relationship between Views and ERB (or Haml/Slim) templates. This has led me to start investigating implementations of the [Two Step View](#) pattern that can be used with Rails, but I don't have a clear solution yet.

Note: The term “Presenter” has caught on in the Ruby community, but I avoid it for its baggage and conflicting use. The “Presenter” term was [introduced by Jay Fields](#) to describe what I refer to above as “Form Objects”. Also, Rails unfortunately uses the term “view” to describe what are otherwise known as “templates”. To avoid ambiguity, I sometimes refer to these View objects as “View Models”.

6. Extract Policy Objects

Sometimes complex read operations might deserve their own objects. In these cases I reach for a Policy Object. This allows you to keep tangential logic, like which users are considered active for analytics purposes, out of your core domain objects. For example:

```
1  class ActiveSupportPolicy
2    def initialize(user)
3      @user = user
4    end
5
6    def active?
7      @user.email_confirmed? &&
8      @user.last_login_at > 14.days.ago
9    end
10 end
```

This Policy Object encapsulates one business rule, that a user is considered active if they have a confirmed email address and have logged in within the last two weeks. You can also use Policy Objects for a group of business rules like an `Authorizer` that regulates which data a user can access.

Policy Objects are similar to Service Objects, but I use the term “Service Object” for write operations and “Policy Object” for reads. They are also similar to Query Objects, but Query Objects focus on executing SQL to return a result set, whereas Policy Objects operate on domain models already loaded into memory.

7. Extract Decorators

Decorators let you layer on functionality to existing operations, and therefore serve a similar purpose to callbacks. For cases where callback logic only needs to run in some circumstances or including it in the model would give the model too many responsibilities, a Decorator is useful.

Posting a comment on a blog post might trigger a post to someone's Facebook wall, but that doesn't mean the logic should be hard wired into the `Comment` class. One sign you've added too many responsibilities in callbacks is slow and brittle tests or an urge to stub out side effects in wholly unrelated test cases.

Here's how you might extract Facebook posting logic into a Decorator:

```
1  class FacebookCommentNotifier
2    def initialize(comment)
3      @comment = comment
4    end
5
6    def save
7      @comment.save && post_to_wall
8    end
9
10   private
11
12   def post_to_wall
```

```
13     Facebook.post(title: @comment.title, user: @comment.author)
14   end
15 end
```

And how a controller might use it:

```
1  class CommentsController < ApplicationController
2    def create
3      @comment = FacebookCommentNotifier.new(Comment.new(params[:comment]))
4
5      if @comment.save
6        redirect_to blog_path, notice: "Your comment was posted."
7      else
8        render "new"
9      end
10   end
11 end
```

Decorators differ from Service Objects because they layer on responsibilities to existing interfaces. Once decorated, collaborators just treat the `FacebookCommentNotifier` instance as if it were a `Comment`. In its standard library, Ruby provides a number of [facilities to make building decorators easier](#) with metaprogramming.

Wrapping Up

Even in a Rails application, there are many tools to manage complexity in the model layer. None of them require you to throw out Rails. ActiveRecord is a fantastic library, but any pattern breaks down if you depend on it exclusively. Try limiting your ActiveRecord's to persistence behavior. Sprinkle in some of these techniques to spread out the logic in your domain model and the result will be a much more maintainable application.

You'll also note that many of the patterns described here are quite simple. The objects are just “Plain Old Ruby Objects” (PORO) used in different ways. And that's part of the point and the beauty of OOP. Every problem doesn't need to be solved by a framework or library, and naming matters a great deal.

What do you think of the seven techniques I presented above? What are your favorites and why? Have I missed any? Let me know in the comments!

P.S. If you found this post useful, you may want to subscribe to our email newsletter (see below). It's low volume, and includes content about OOP and refactoring Rails applications like this.

Further Reading

- [Objects on Rails](#)
- [Crazy, Heretical, and Awesome: The Way I Write Rails Apps](#)
- [ActiveRecord \(and Rails\) Considered Harmful](#)
- [Single Responsibility Principle on Rails Explained](#)


Thank you to Steven Bristol, Piotr Solnica, Don Morrison, Jason Roelofs, Giles Bowkett, Justin Ko, Ernie Miller, Steve Klabnik, Pat Maddox, Sergey Nartimov and Nick Gauthier for reviewing this post.

Looking for more about Ruby, code quality, OOP and Rails security? Subscribe to our newsletter.

Comments




Join the discussion...



Lem Lordje Ko · a year ago

Would be really great to see an application source code that illustrates these points.

17 ^ | ▾ · Reply · Share ›




mdenomy ➔ Lem Lordje Ko · 18 days ago

Perhaps look at code you have written for pain points and see if you can't apply some of these techniques. I recently used the form objects pattern to refactor code that had a pretty complex form and the underlying model had a couple "accepts_nested_attributes" that were making things unwieldy.

One piece of advice I would offer in applying these patterns to legacy code is to make small refactorings as you go along, and of course with a strong test suite that helps make sure nothing has been broken in the process.


^ | ▾ · Reply · Share ›



Dabl Web ➔ Lem Lordje Ko · 18 days ago

Unfortunately, in order for these points to actually be useful and relevant, the example application would have to be pretty big.


^ | ▾ · Reply · Share ›



Roman Gaufman · a year ago

Where would you put the classes for the ValueObjects in the rails structure? - lib/?


12 ^ | ▾ · Reply · Share ›



Ryan Cheung ➔ Roman Gaufman · 8 months ago

`app/values/` would be ok.


5 ^ | ▾ · Reply · Share ›



Overflow012 ➔ Ryan Cheung · 7 months ago

If I'm not wrong, the ValueObjects are related with models layers. Why not put in 'models/values'?


3 ^ | ▾ · Reply · Share ›



Dabl Web ➔ Overflow012 · 18 days ago

Not always. Decorators, Controllers & Models often have legitimate uses for Value objects.


^ | ▾ · Reply · Share ›



Chandra Mohan Thakur ➔ Roman Gaufman · 2 months ago

I think the better place to put those in app/lib . Or if you think it is related with model layer, then it will be better to put those in app/models/concerns.

^ | ▾ · Reply · Share ›




Dreamr OKelly ➔ Roman Gaufman · 5 months ago

I prefer app/lib for all dsl code that 'lives' in the rails app, ie, hasn't been extracted into a library or gem yet.

Eventually I pull collections of ruby objects out of app/lib and move them into a gem and include it.

^ | ▾ · Reply · Share ›



hakunin · a year ago

One thing I want to emphasize: please, beginner rails devs, do not just create your app/service_objects and app/policy_objects dirs, and start placing logic there in your new rails app. This would miss the point. Use these guidelines to decompose growing code that is revealing itself over time. Develop incrementally. Sometimes decomposing a 3-line method into an object with 5 1-line methods is not the right thing to do.

Use your common sense and taste.

Overall, this is a superb categorization of common object extractions.

10 ^ | v · Reply · Share ›

 **Robert Gravina** → hakunin · a year ago

Actually I was wondering where these models should live (once extracted, of course). app/service_objects, app/policy_objects actually makes sense, to avoid a gigantic models dir and to show that they are shared. Is there a better place for them? What about namespaces?

^ | v · Reply · Share ›

 **hakunin** → Robert Gravina · a year ago

That's probably fine. I'm only saying saying that you shouldn't start creating all these decoupled objects on a new rails app that you're designing from scratch. Let them reveal themselves naturally over time. Not commenting on where to place them.

3 ^ | v · Reply · Share ›

 **CrazyDog Software** · a year ago

Great article. When I first started it, I thought, "Oh no, not another article about extracting mixins from models!", but was presently surprised. I'm not sure what some of the Ruby community has against POROs, but I find it uncommon to see this level of object extraction. Ruby is OO from ground up and makes this kind of stuff as easy as it gets.

I think that the backlash against "fat models" has gone too far in some areas, which is where all the "concerns" mixin madness comes from. "Fat models" is too generic an idea and leads people to think about code size rather than logical issues, which can lead to anemic domain models. Maybe instead we should talk about "procedural models" or something like that to indicate a model where OO composition is lacking.

To state it more clearly, business logic specific to the domain and to a specific model "should" go into the model. It may be the case that you have an object with extremely complex business logic that really belongs where it is, and that's *okay*. Generally size can be an indicator that you may have some smells in the model, and need to extract some things, but it's not the best yardstick. Instead understanding OO patterns like those you've written about and when to use them will provide you with a much better yardstick for when models need to be refactored.

7 ^ | v · Reply · Share ›

 **brynary** Mod → CrazyDog Software · a year ago

Thanks. I agree with most of your comment, but would dispute this sentence:

"It may be the case that you have an object with extremely complex business logic that really belongs where it is, and that's *okay*." In that case, I think it's almost always an indicator that the domain needs to expand to absorb the complexity of the object. The complex object may be the "right" place for it in the current domain, but not the ideal domain for that level of intrinsic complexity.

Thoughts?

-Bryan

^ | v · Reply · Share ›

 **CrazyDog Software** → brynary · a year ago

Yes, it was a poorly worded sentence. I agree completely that complex models often indicate areas where the domain needs to be expanded. I think the issue that I see is that too many people go about refactoring by pulling all the behavior out of their domain model and into "service layers", when it really belongs in the domain model. You've then got all the cost of having a domain model, littered with what essential amounts to procedural scripts or "concerns". This wouldn't be a problem if people followed you criteria for when to extract a service object, ie stuff reaching across models and non-problem domain concerns. My primary concern is that the opposite of "fat" is anemic... which is the last thing you want your domain model to be.

^ | v · Reply · Share ›

 **brynary** Mod → CrazyDog Software · a year ago

Good point. What you're describing sounds like the "Anemic Domain Model" anti-pattern:

<http://martinfowler.com/bliki/...>

^ | v · Reply · Share ›

 **depy** → brynary · a year ago

It's a shame most of the people still don't know the difference between rails model and domain model. I myself usually don't have much code in rails models because then I can not test the logic in the isolation. But on other hand as you people said, extracting everything into one service object or some whatever called object is

certainly not a solution.

Extracting the logic into more service objects may not be a solution but is much much better. First because of seperation of concerns and second because you can test this in isolation.

And about POROs I agree. Have no idea why people using rails are afraid of them. I my self come from Java world and first I was amazed by belief of the rails community that design patterns are for Java like languages and that they don't need it because of rails. It's absurd. :)

More posts like this. Rails community has to wake up. We're not building blog in 10min anymore.

3 ^ | v · Reply · Share ›



donaldball → CrazyDog Software · a year ago

What would you say the the domain-driven design suggestion that the domain and its persistence are separate responsibilities, and should be separate objects?

^ | v · Reply · Share ›



brynary Mod → donaldball · a year ago

"It depends" ... on the app. Frankly, in Ruby we just don't have good tooling for truly cutting that dependency, so our cost/benefit curve is not ideal.

1 ^ | v · Reply · Share ›



dei79 · 9 months ago

I like the approach of form objects but does anybody has a practical example or a good tutorial for doing this with real nested models which will be added dynamically (add post, add comment, etc...)?

5 ^ | v · Reply · Share ›



Ben Dilley → dei79 · 4 months ago

Ryan Bates has recorded a railscast on the topic: <http://railscasts.com/episodes...>

^ | v · Reply · Share ›



jballo → dei79 · 6 months ago

I see there are no responses to your comment here yet. This is what I am also wondering. Did you find anything of note?

^ | v · Reply · Share ›



Nils Caspar · a year ago

Question regarding "3. Extract Form Objects": It looks like you have to duplicate all validations from the underlying models. This would result in code duplication... Am I not getting something obvious here?

4 ^ | v · Reply · Share ›



brynary Mod → Nils Caspar · a year ago

Many validations are contextual. For example, many User validations need to be specified as ":on => :create" because as the rules change over time you don't want to create unsaveable records. So a SignUpForm encapsulates the validations that are needed on creation.

If the validations do need to be kept in sync, and they are non-trivial or changing, you can extract a custom Validator object.

^ | v · Reply · Share ›



Pablo Cantero → brynary · 10 months ago

How about database transactions?

```
def persist!
  @company = Company.create!(name: company_name)
  @user = @company.users.create!(name: name, email: email)
end
```

If the `users.create!` raises an exception, the company will be not rolled back.

We should wrap up in a transaction.

```
def persist!
  ActiveRecord::Base.transaction do
    @company = Company.create!(name: company_name)
    @user = @company.users.create!(name: name, email: email)
  end
end
```

7 ^ | v · Reply · Share ›



solnic · a year ago

Thanks again for mentioning Virtus here. It's probably worth to mention that Virtus can be used to build



Thanks again for mentioning virtus here. It's probably worth to mention that virtus can be used to build value objects. See here: <https://github.com/solnic/virt...>

3 ^ | v · Reply · Share ›



brynary Mod ➔ solnic · a year ago

Good point!

^ | v · Reply · Share ›



Steven Harman · a year ago

re: "Don't Extract Mixins from Fat Models" -> I've been referring to these as BOMMs: <http://stevenharman.net/bag-of...>

2 ^ | v · Reply · Share ›



brynary Mod ➔ Steven Harman · a year ago

Thanks for this link, Steven.

1 ^ | v · Reply · Share ›



Ken Collins · a year ago

Great write up! Can you change one of your code examples to not use "&block" and just yield to the block instead. I'm on a mission to educate people more about this. A reference link is here.

<http://blog.sidu.in/2008/01/ru...>

1 ^ | v · Reply · Share ›



brynary Mod ➔ Ken Collins · a year ago

You mean in the AbandonedTrialQuery class? Can you show me what the code you'd use is?

^ | v · Reply · Share ›



Peter Marreck ➔ brynary · a year ago

Agreed. Capturing a block into a variable is expensive. Using it directly with yield is much less so, unless you really need the associated Proc (so you can execute code in the block's context, for example). The reason is that in the latter case, there have been some optimizations done in Ruby towards expediting implicit block handling (example: iterators which receive a block directly).

^ | v · Reply · Share ›



Edward J. Stembler · a year ago

Great article. Though, if I may offer a suggestion: it may be helpful to include information about where you see these objects residing in your Rails directory structure.

Also, I wonder if some of these patterns may be more suited for just-in-time DCI implementations?

1 ^ | v · Reply · Share ›



brynary Mod ➔ Edward J. Stembler · a year ago

Hey Edward -- Thanks. I'll cover the module and directory organization question a bit more in the future. I wrote about it a bit here:

<http://blog.codeclimate.com/bl...>

But my thinking has evolved since then. I wouldn't stress too much about it -- if you have the right objects, relocating them from one directory to another is no big deal.

^ | v · Reply · Share ›



noahhendrix · a year ago

I can't begin to explain what a revelation #3 is to me. It makes so much more sense when you can move all form related logic to a specific object. It alleviates all the latest brouhaha regarding where to perform authorization.(see: <https://gist.github.com/197564...>

That logic has a clear home now, inside a form object that can better understand it's responsibilities.

1 ^ | v · Reply · Share ›



Mike Pack · a year ago

Great article! I'm quite happy to see more Ruby being built with proper abstraction and SRP. We could learn a thing or two from the Java world.

1 ^ | v · Reply · Share ›



Mike Moore · a year ago

This is a great post. All of these tips are wonderful and I want to work on more apps that use these approaches. Thank you for writing it. We need more of this type of thinking.

I have one relatively minor nit to pick about your "View Object" and explanation of Presenters. You are showing a "Form Object" because it is so specific to being handled within a form. That code is not a View Object or a ViewModel to my eyes. You say Presenter also describes what you show, but again that is not so. A Presenter is a "representation of the state of the view". (Jay says exactly that the first paragraph of the article you linked to.) The view contains more behavior than just what is in a form, such as determining whether to show specific content and presenting notifications. Presenters represent the state of the entirety of the view and as such are responsible for more than what is in a form.

Or the view and as such are responsible for more than what is in a form.

I really loved this post, and I offer my apologies for being pedantic. That is a minor point of clarification on an otherwise excellent post. ♥♥♥

1 ^ | v · Reply · Share ›

 **brynary** Mod ➔ Mike Moore · a year ago

Hey Mike -- Thanks! To clarify, when you say "That code is not a View Object or a ViewModel to my eyes" are you referring to the DonutChart class or the Signup class?

^ | v · Reply · Share ›

 **Mike Moore** ➔ brynary · a year ago

Ugh, sorry. In section 5 about View Objects you state that the Form Objects you mention in section 3 are Presenters. I mentally replaced the DonutChart example with the Signup example when making my comment. My bad.

The DonutChart example is a bit thin but certainly could be a View Object/ViewModel/Presenter. My comments about Presenters belonging in that list as opposed to what you call a Form Object still holds.

My apologies for my confusion.

^ | v · Reply · Share ›

 **jacopo beschi** · 2 months ago

If you wanna know something more about active record you should read this:
<http://www.jacopobeschi.com/po...>

^ | v · Reply · Share ›

 **Nigel Pepper** · 3 months ago

Great post guys - many thanks. We've been using a couple of these patterns with great success (for me measured by ease and pleasure of future changes to the same).

^ | v · Reply · Share ›

 **Dreamr OKelly** · 5 months ago

I think the biggest issue is rails is a stupid simple thing to fix.... Stop creating 20 directories in app/

Move that shit to a gem or lib. app/concerns app/services app/values doesn't solve the problem of having too much domain code in your rails, it begs you to throw more 'trash' into rails.

Write this shit in ruby - leave network interfacing (db calls, etc) to your rails app. Your ruby classes don't need to worry about persistence, and your tests will thank you.

If using multiple repos is 'too hard' then do lib/some_domain_collection and build up your lib to look like a library, not a junk drawer.

^ | v · Reply · Share ›

 **Dabl Web** ➔ Dreamr OKelly · 18 days ago

DHH himself says that all app-specific logic belongs in /app and not in /lib. The consensus is that services should live in app/services and I'd argue for the other types of POROs to live in app as well.

^ | v · Reply · Share ›

 **MrChris** · 5 months ago

In 'Extract Decorators' you say 'Once decorated, collaborators just treat the FacebookCommentNotifier instance as if it were a Comment', but in your example the only exposed method to collaborators is #save. It would make sense though if all methods were forwarded to the underlying object.

^ | v · Reply · Share ›

 **Hamed Asghari** · 6 months ago

I have a blog post on how to enhance the query object pattern with custom scopes:
<http://hasghari.github.io/2013...>

^ | v · Reply · Share ›

 **Overflow012** · 7 months ago

Great article! What do you think about wrap all ActiveRecord models into PORO classes (Considering their relationships)?

^ | v · Reply · Share ›

 **Samnang Chhun** · 8 months ago

When do you choose one over the other between Service Object vs Form Object?

^ | v · Reply · Share ›

 **Ryan Cheung** · 8 months ago

I absolutely like the DDD way to developing rails app.

^ | v · Reply · Share ›



[shamim](#) · 9 months ago

Hi Bryan, I think this is a great article. It answers many of my questions and confusions. I just have a comment on your save method in pattern 3. inside save, persist! is called. I always find this confusing (calling a bang method inside a non-bang method). If we assume that persist! is a dangerous method and is changing @user and @comment directly, then so is save. but why dont we call save with a bang?

^ | v · Reply · Share ›



Joe Jackson → [shamim](#) · 7 months ago

It looks like you are caught up in the ruby "!" convention vs the rails "!" confusion. The rails bang methods don't mean dangerous (mutating) like ruby ones do it means that they raise an error if they fail. So persist!, by calling create! will raise an error on failure.

^ | v · Reply · Share ›

Load more comments