Separation of concerns (Single Responsibility Principle) in Rails

Separation of concerns is one of the key goals of software development. It deals with good maintainance pattern. For this to be, every change have to be local. Then programmer has to separate things, the more he can do, avoiding bounded code.

To achieve this, he has to :

- **Separate out the things that change from those that stay the same**.
- **Program to an interface, not an implementation** : roughly speaking, code doesn't have to know which kind of object it deals with. Here duck typing is the king. The only thing you have to think of is building a good interface (a good "langage" to throw messages from one component to another). To conceive an interface, you have to think about the more general thing : car < vehicule < movable object. Best think about movable object.
- **Prefer composition over inheritance** : think about what your object has rather than what it is. And everything it has have to be another object.
- **Delegate, delegate, delegate** : your car doesn't start its engine, it says to its engine to start!
- **YAGNI** : you ain't gonna need it ! Do not anticipate what the future will be. Because doing this can lead you to a point from where the evolution to the actual future can be harder. You can't predict where the code will have to go...

So here I am to apply all this advices/goals in a rails app.

Rails components

Controller

The central component. It is the part interacting with the browser, receiving http requests and responding by a content. Does it have to know about database structure ? No. This doesn't mean it doesn't have to deal with ActiveRecord instances. Does it have to know about how to deal with the sql request ? No. It has to know who has to deal with the actual use case and which message have to be sent to it. Then it has to return the accurate response (depending of use-case return).

View

Does it have to know about database structure ? No. It has to know about what has to be displayed. So it has to receive something that is built from the database to respond to what needs to be displayed.

Model (ActiveRecord object)

Does it have to know about which kind of manipulation data are subject to ? Yes/No. This is what all this paper is about. It deals with storing to -- and validating -- and reading (performing requests) from database.

Who have to make a given action

Delegate, delegate, delegate rule applied to controller, tells us that it should only distribute the work to be done upon the right workers.

So controller have to send messages, but it doesn't manage how things have to be done. This is not its job.

I think I can divide the workflow in a few tasks.

- Processing POST, PATCH or DELETE actions,
- Querying model to get needed informations,
- Presenting these informations for view usage,
- Handling actions that have nothing to do with data persistence or display (services)

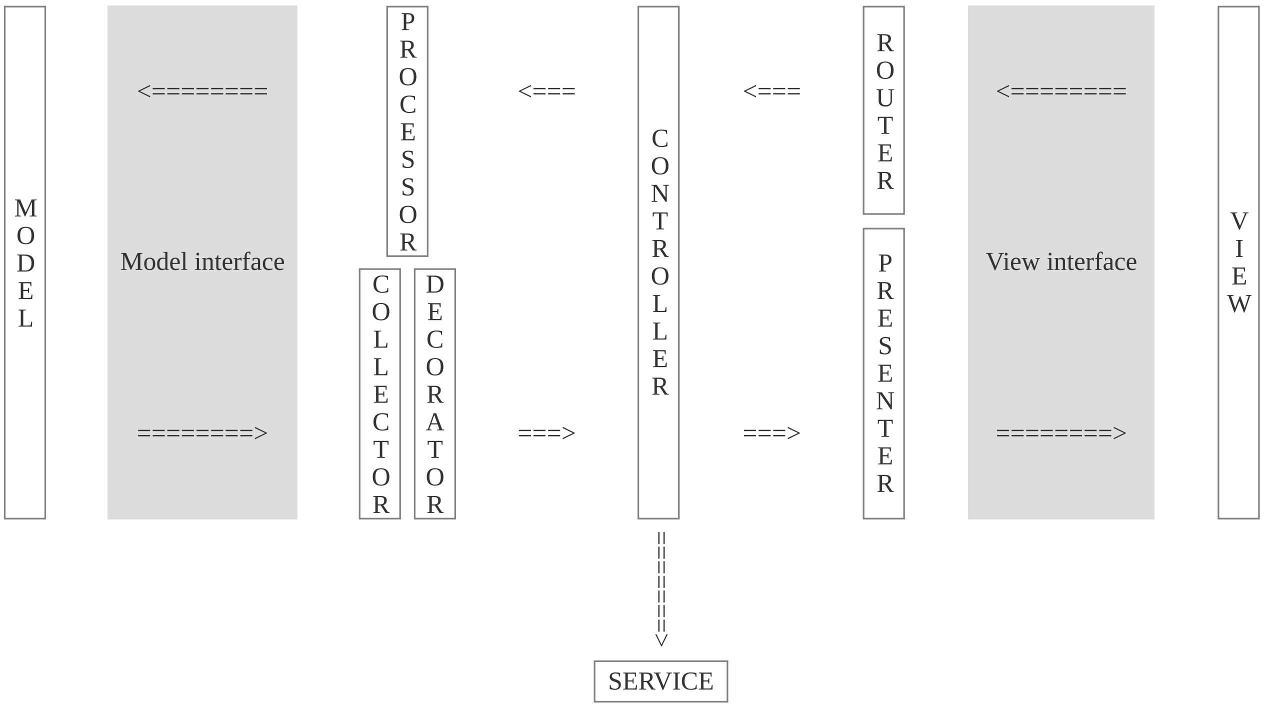
My solution

After a lot of tries, I finally decided that my good way of doing things is :

- to provide Processor classes to perform complex database mutations.
- to provide Collector classes to performs complex queries. Hence, model only contains validations and basic scoping methods.
- to provide Decorator classes to add dedicated Presenter mixins modules as interfaces between query results (which are eventually not model instances or list...) and views.

Conventional names : `InputsCollector` , `InputsProcessor` , `InputsDecorator` . Concerning presenters, beside eventual `InputsPresenter` we could find `IndexPresenter` and so on.

Separation of concerns :



Final tree and ApplicationRecord extension

```
app
├── collectors
│   ├── entries_collector.rb
│   └── inputs_collector.rb
├── decorators
│   ├── entries_decorator.rb
│   └── inputs_decorator.rb
├── lib
│   ├── collectable.rb
│   ├── decorable.rb
│   ├── presenting.rb
│   ├── processable.rb
├── presenters
│   ├── balance_presenter.rb
│   ├── budgets_presenter.rb
│   ├── entries_presenter.rb
│   ├── home_presenter.rb
│   └── inputs_presenter.rb
├── processors
│   ├── budgets_processor.rb
│   └── inputs_processor.rb
```

Controllers

A way to handle parameter permissions from controllers :

```
# app/controllers/application_controller.rb
class ApplicationController < ActionController::Base
  def permitted_params
    model = self.controller_name.singularize
    params.require( model.to_sym ).permit(
      Object.const_get( model.camelize ).permitted_params
    )
  end
end

# app/models/stay.rb
class Stay < ApplicationRecord
  def self.permitted_params
    [ :title, :name, :whatever ]
  end
end
```

No more need for a `stay_params` method... You may define a `permitted_params` method to override this default behaviour.

Typical controller methods

Here typical controller methods calling a collector and a processor.

```
def create
  @stay = Stay.new( permitted_params ).decorate_for :new

  unless @stay.proceed_to :create, with: { hash_context }
    render :new, status: :see_other and return
  end
  flash[:notice] = 'Successfully created!'
end

def index
  @inputs = Input.collect_for :index, with: { period: current_period }
end
```

Another object ?

In big apps it could be interesting that controller build a « context » object to be given as parameter to processor and collector. This context has to contain all the contextual stuff (like session hash, etc.). Context object could be a Struct. which could be questionned : `context.current_user` , `context.stay_id` , `context.stay` , etc.

Then a typical call will be :

```
@stay.proceed_to :create, with: context
```

context being a controller private method..

Base modules and classes

For all this works, mixins modules inside ApplicationRecord

```
class ApplicationRecord < ActiveRecord::Base
  primary_abstract_class
  include Processable, Collectable, Decorable
end
```

```
# app/lib/collectable.rb
# Dedicated to be included in ApplicationRecord class
# Provide a way to extract all model querying logic from both
# controller and model by delegating it to Collector subclasses.
#
# Then controller syntax becomes :
# Model.collect_for :view_name, with: { context_hash }
# e.g. Input.collect_for :index, with: { budget_id: current_budget_id }
# The InputsCollector class then needs to implement
# 'collect_for_index' method which is called. An instance variable @context
# contains the context_hash passed by controller
module Collectable
  def self.included( klass )
    class << klass
      def collect_for( view_name, with: {} )
        collected = Object
          .const_get( "#{self.name.pluralize}Collector" )
          .query_for view_name, with: with
        decorator = Object
          .const_get( "#{self.name.pluralize}Decorator" )
          .decorate_for view_name, collected, with: with
      end
    end
  end

  class Collector
    class << self
      def query_for( name, with: {} )
        self
          .new( with )
          .send( "query_for_#{name}" )
      end
    end

    def initialize( context )
      @context = context
    end

  end

end
```

```ruby
# app/lib/processable.rb
# Dedicated to be included in ApplicationRecord class
# Provide a way to extract all model processing logic from both
# controller and model by delegating it to Processor subclasses.
#
# Then controller syntax becomes :
# Budget.proceed_to :create, with: { user_id: current_user.id }
# Input.proceed_to update: @input, with: { params: permitted_params }
# or (equivalent)
# @input.proceed_to :update, with: { params: permitted_params }
#
# The InputsProcessor class then needs to implement
# 'proceed_to_update' method which is called. Instance variables @target and
# @context contain respectively the model instance and the context_hash.
module Processable
  def self.included( klass )
    class << klass
      def proceed_to( method = nil, **args )
        Object
          .const_get( "#{self.name.pluralize}Processor" )
          .proceed_to method, **args
      end
    end
  end

  # shortcut for instances..
  def proceed_to( method, **args )
    Object
      .const_get( "#{self.class.name.pluralize}Processor" )
      .proceed_to method.to_sym => self, **args
  end

  class Processor
    class << self
      def proceed_to( method = nil, **args )
        method ||= args.keys.first
        model_instance = args.fetch( method, nil )
        context = args.fetch( :with, nil )
        self
          .new( model_instance, context )
          .send( "proceed_to_#{method}" )
      end

    end

    def initialize( target, context )
      @target = target
      @context = context
    end
  end
end
```

```ruby
# app/lib/decorable.rb
# Dedicated to be included in ApplicationRecord class
# Provide a way to extract all model presentation logic from both
# controller and model by delegating it to Presenter modules.
#
# Two ways are provided to decorate a model query result with dedicated
# presenters:
# 1/ A call to Model.collect_for :view_name method automatically implies
# a subsequent call to 'decorate_for view_name'
# 2/ A controller call to @input.decorate_for :show, with: {} is also possible
# when model query do not require Collector management.
#
# The InputsDecorator class then CAN implement
# 'decorate_for_view_name' method which is called. Instance variables
# @collected and @context contain respectively the query result and the
# context_hash.
#
# Note : if InputsDecorator implement method_missing, it can manage default
# decoration functionality.
module Decorable
  # a short cut for model instances
  def decorate_for( view_name, with: {} )
    Object
      .const_get( "#{self.class.name.pluralize}Decorator" )
      .decorate_for view_name, self, with: with
  end

  class Decorator
    def self.decorate_for( view_name, collected, with: {} )
      self
        .new( collected, with: with )
        .send( "decorate_for_#{view_name}" )
    end

    def initialize( collected, with: {} )
      @collected = collected
      @context = with
    end

  end

end
```

```ruby
# app/lib/presenting.rb
module Presenting
  module ClassMethods
    # Syntactic sugar that provide a way to automatically prefix method names.
    def presenting( name, &block )
      define_method "present_#{name.to_s}".to_sym, &block
    end
  end

  def self.included( presenter )
    presenter.extend ClassMethods
  end
end
```

Processor classes

These classes are concerned with « processing » model persistence action. The naming convention for processor methods is `proceed_to_xxx` . Processor parent class provide a @target and a @context instance variables.

```
# app/processors/stays_processor.rb
class StaysProcessor < Processable::Processor
  def proceed_to_update
    @target.update( @context[:params] )
  end
end
```

Collector classes

These classes are concerned with « collecting » data from models. The naming convention is `query_for_xxx` where `xxx` is a controller method (and a view name). A @context instance variable is provided by Collector parent class.

```
# app/collectors/stays_collector.rb
class StaysCollector < Collectable::Collector

  def query_for_index
    @context[:rental_place].stays.where( "actual_period && ?", @context[:period] )
  end

  def query_for_show
    @context[:rental_place].stays.findbyid( @context.fetch( :id, false ) )
  end
end
```

I think this is a good place to provide defaults value for new records. It could also be achieved inside presenters, but this is not a presentation stuff.

Decorator classes

These classes are concerned with « decorating » query results with the dedicated mixins (which are Presenter modules). The naming convention is `decorate_for_xxx` where `xxx` is a view name. @collected and @context instance variables are provided.

```
class InputsDecorator < Decorable::Decorator
  def method_missing( name, *args )
    # provide a default decoration
    @collected.extend InputsPresenter
  end

  def decorate_for_history
    @collected.map { |input| input.extend InputsPresenter }
  end
end
```

Presenter modules

```
# app/presenters/index_presenter.rb
module IndexPresenter
  include Presenting
  module Iterator
```

```ruby
    def each_bay
      self.each do |input|
        input.extend IndexPresenter::Single
        yield input
      end
    end
  end

  module Single
    include Presenting
    presenting :my_specific_title do
      ...
    end
  end
end
```