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...

Another thing related is to **avoid bounded code** : I will use **dependency injection** wherever it is possible together with maximazing **functional code** and minimazing **side_effects**.

> Explaination : a side effect occur when a method does anything else than returning a value (like mutating its arguments, or mutating anything else).

Functional code is easy and fast to test and easy to debug...

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. Doesn't mean it doesn't have to deal with ActiveRecord instances? Only as far as it does not know it... 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. Probably not ! This is what all this paper is about. It deals with storing to -- and validating or not ! -- 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,
- Collecting needed information from models,
- Presenting these informations within views,
- Handling actions that have nothing to do with data persistence or display (services)

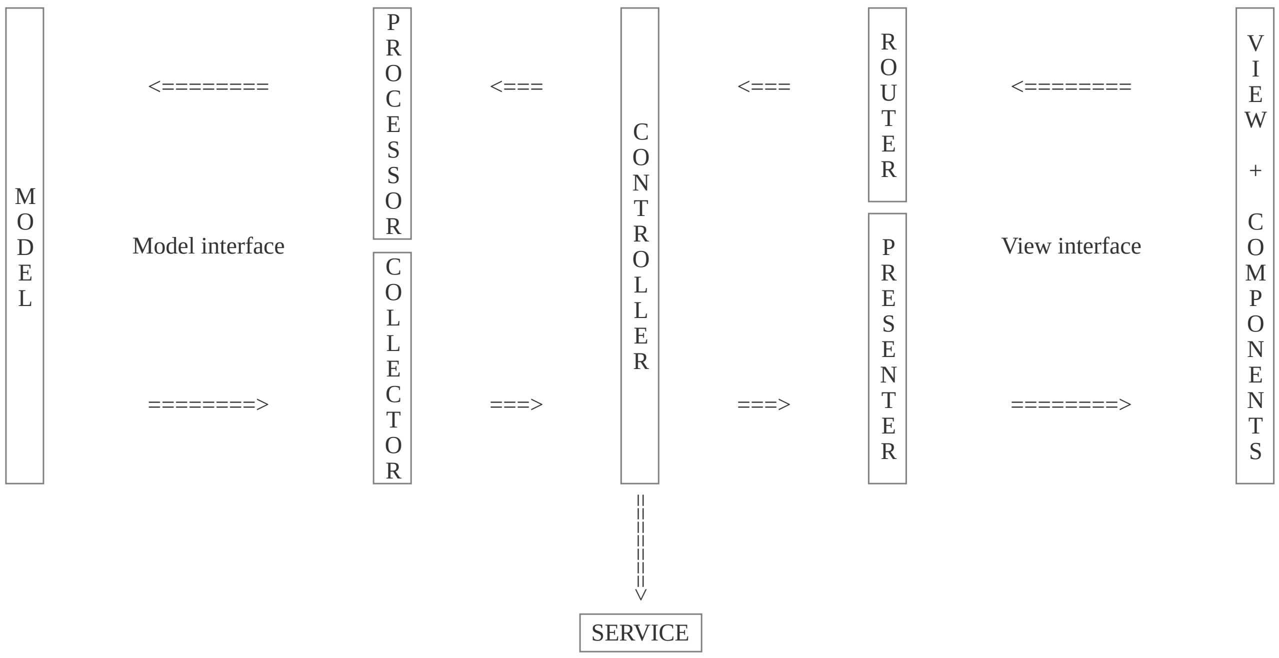
My solution

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

- to provide `Processor` classes to perform database mutations.
- to provide `Collector` classes to performs queries. Hence, model only contains validations and basic scoping methods.
- to provide `Presenter` classes to add dedicated presentation stuff to query results together with custom view `Component` objects. They encapsulate all the logic needed inside views so non-encapsulated-rails-helpers are not anymore needed. Better is using `Presenter` ; I keep `Component` objects only for shared behaviour (like formatting currency values, rendering icons).

Following the convention naming : *name what it do or render, not what it is* `EntriesProcessor` , `InputsCollector` , `InputsHistoryPresenter` , `ReportPresenter` , `HomeIcon` (or `HomeIconComponent`)

Separation of concerns :



Final tree and ApplicationRecord extension

```
app
├── collectors
|   ├── entries_collector.rb
|   └── inputs_collector.rb
├── components
|   ├── icon.rb
|   ├── to_currency.rb
|   └── home_icon.rb
├── lib
|   ├── logicore.rb
|   ├── component.rb
|   ├── collector.rb
|   ├── processor.rb
|   └── presenter.rb
├── presenters
|   ├── inputs_presenter.rb
|   └── balance_presenter.rb
├── processors
|   ├── budgets_processor.rb
|   └── inputs_processor.rb
├── views
|   ├── components
|   |   └── _icon.rb
```

Controllers

A way to handle parameter permissions within controllers :

```
# app/controllers/application_controller.rb
class ApplicationController < ActionController::Base
  include ParamsManager
end

# app/lib/params_manager.rb
module ParamsManager
  protected
  def permitted_params
    set_model_vars
    sanitize_decimal_values # YES, this is a controller concern !
    params.require( @model_name ).permit( @model.permitted_attributes )
  end
  private
  def set_model_vars
    @model_name = params[:controller].singularize.to_sym
    @model = Object.const_get( @model_name.capitalize )
  end

  def sanitize_decimal_values
    return unless @model.respond_to? :numerical_attributes
    @model.numerical_attributes&.each do |field|
      params[ @model_name ]
        .fetch( field, '' ).to_s.gsub!( ',', '.' )
    end
  end
end

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

No more need for a `stay_params` method...

Typical controller methods calling processor or collector

```
def create
  input = Input.new( permitted_params )
  unless InputsProcessor.( # this is good dependency injection !
      create: input, context: { profile: current_budget.profile }
  )
    @input = InputPresenter.( input )
    render :new, status: :unprocessable_entity and return
  end

  redirect_to :authenticated_home

def index
  @inputs = InputPresenter.(
    InputsCollector.( :query_for_history, context: { period: current_period } )
  )
end
```

Base modules and classes

```ruby
module Logicore
  # A way to transform a hash to an object.. holding hash behaviour.. read-only by design.
  # Works only with symbol or string keys.
  class Context
    def initialize( context = {} )
      @context = context
    end

    def method_missing( name, *args )
      args.empty? ? get( name ) : super
    end

    def []( key ); get( key ); end

    private
    def get( key )
      @context.fetch( key.to_sym, @context.fetch( key.to_s, nil ) )
    end
  end

  class Processor
    attr_reader :target, :context

    def self.call( method = nil, **args )
      method ||= args.keys.first
      model_instance = args.fetch( method, nil )
      context = args.fetch( :context, nil )
      new( model_instance, context ).send( method )
    end

    def initialize( target, context )
      @target = target
      @context = Logicore::Context.new( context )
    end
  end

  class Collector
    attr_reader :context

    def self.call( method, context: )
      new( context ).send( method )
    end

    def initialize( context )
      @context = Logicore::Context.new( context )
    end
  end

  class Presenter
    def self.call( collected )
      collected.extend self::Fallback
      collected.extend self::ThePresenter
    end

    module Fallback
      def method_missing( name, *args )
        return super unless name.to_s =~ /^the_/
        send( name.to_s.gsub( "the_", "" ), *args )
      end
    end
  end
end
```

```
    class Component
      delegate :render, to: :view_context
      attr_reader :view_context

      def partial_name; nil; end

      def rendered_object
         inline_template? ?
          { inline: erb_template } :
          { partial: [ partial_folder, partial_name ].join }
      end

      def render_in( view_context, &block )
        return unless render?
        @view_context = view_context
        render **rendered_object, locals: provided_vars
      end

      def render?; true; end

      private

      def inline_template?
        respond_to?( :erb_template )
      end

      def partial_folder
        "components/"
      end
    end

  end
```

A typical Component

```
  class ToCurrency < Component
    attr_reader :value, :default, :session

    def initialize( value, default = '_' )
      @value = value
      @default = default
    end

    def render_in( view_context, &block )
      return default unless value
      @view_context = view_context
      @session = view_context.session
      view_context.number_to_currency(
        value,
        **format,
        unit: unit
      )
    end

    private

    def format
      CurrencyManager::CURRENCY_FORMATS[session.currency_format.to_sym]
    end

    def unit
      session.currency_unit
    end
  end
```

Processor classes

These classes are concerned with « processing » model persistence action. Logicore::Processor parent class provide a `target` and a `context` attribute readers.

```
# app/processors/stays_processor.rb
class StaysProcessor < Logicore::Processor
  def update
    target.update( context.params )
  end
end
```

Collector classes

These classes are concerned with « collecting » data from models. A `context` attribute reader is given by Collector parent class.

```
# app/collectors/stays_collector.rb
class StayCollector < Logicore::Collector

  def query_for_new
    context.rental_place.stays.where( "actual_period && ?", context.period )
  end

end
```

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

Presenter classes

These classes are concerned with adding presentation skills to query results by extending them with adequate mixin named `ThePresenter` . Using a class allow encapsulated code and easily testable unit gracefully to the dependency injection it allows. `collected` and `context` attribute readers are provided.

```
class EntryPresenter < Logicore::Presenter
  module ThePresenter
    # automagically extending call method argument
    # convention, all view-accessible methods are prefixed by `the_`

    def the_title
      title.capitalize
    end

    def each_bay
      each do |item|
        yield AnotherPresenter.( item )
    end
  end
end
```

More encapsulated or isolated stuff : Session

Providing methods to access session content permits to treat default values or on-the-way storage..

```ruby
class ApplicationController < ActionController::Base
  before_action :extend_session

  def extend_session
    session.extend SessionManager
    session.controller = self
  end
end



module SessionManager
  attr_reader :controller

  def update!
    budget = get_budget
    self[:budget_id] = budget.id
    ...
  end

  def controller=( controller )
    @controller = controller
  end

  def budget_id
    store_budget_id_in_session unless self[:budget_id]
    self[:budget_id]
  end

  def prefered_language
    self[:prefered_language] || budget_prefered_language
  end

  def currency_unit; self[:currency_unit]; end
  def currency_format; self[:currency_format]; end

  private
  def get_budget
    controller.current_user.budgets&.first
  end

  def store_budget_id_in_session
    ...; self.update!
  end

  def budget_prefered_language
    # provide default
  end
end
```

A typical View

```erb
<% @index_content.each do |item| %>
  <%= render EditIcon.new %>
  <%= item.the_title %>
  <%= render ToCurrency.new( item.the_value ) %>
<% end %>
```