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 shouldn't have to build model instances. But, this override Rails conventions; Overridding these conventions, may here, pull the app too far from any rails common developper. So collaboration would be hard !

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

I think I can divide the workflow in three tasks.

- Processing POST, PATCH or DELETE actions,
- Querying model to get needed informations,
- Presenting these informations for view usage.

My solution

After a lot of tries, I finally decided that my good way of doing things is to extend models with three modules to provide those three skills :

- A processor to proceed to actions. Name convention : StaysProcessor
- A collector to collect records from model. Name convention : StaysCollector
- A presenter to present records inside views. Name convention : StaysPresenter

I want these extensions to be done dynamically. So I defined three modules, called Processable, Collectable and Presentable which extends ApplicationRecord. Using `inherited` class method, I can auto-magically extend subclasses (models) with appropriate modules.

Final tree and ApplicationRecord extension

```
app
├── collectors
│   └── stays_collector.rb
├── lib
│   ├── collectable.rb
│   ├── presentable.rb
│   └── processable.rb
├── presenters
│   └── stays_presenter.rb
├── processors
│   └── stays_processor.rb
```

```ruby
# app/model/application_record.rb
class ApplicationRecord < ActiveRecord::Base
  primary_abstract_class
  extend Processable, Collectable, Presentable
end
```

Typical controller methods

Here typical controller methods calling a processor and/or a presenter.

```ruby
def create
  @stay = Stay.new( permitted_params )
  unless @stay.proceed_to :create # this is call to processor
    render :new, status: :see_other and return
  end
  flash[:notice] = 'Successfully created!'
end

def index
  @stays = Stay.query_for :index, with: { period: current_period }
  # alternative  Stay.collect_for :index, with ...
end
```

Another object ?

In big apps it could be interesting that controller build a « context » object to be given as parameter to processor and presenter. 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.

Auto-extension/inclusion of my modules

```ruby
# app/lib/processable.rb
# This module auto-magically extends/includes models (AR subclasses) with
# processor submodules named ModelsProcessor::ClassMethods and/or
# ModelsProcessor::InstanceMethods if exist. This processor is designed to
# manage all the heavy work to be done with POST, PATCH or DELETE html queries.
module Processable
  def inherited( klass )
    super
    mod = "#{klass.name.pluralize}Processor"
    { extend: "ClassMethods", include: "InstanceMethods" }
      .select { |m, sub| Object.const_defined?( "#{mod}::#{sub}" ) }
      .each { |m, sub| klass.send( m, Object.const_get( "#{mod}::#{sub}" ) ) }

    define_method :proceed_to do |method, with: {}| # for instances !
      self.send( "proceed_to_#{method.to_s}", with )
    end
  end

  def proceed_to( method, with: {} ) # for class
    self.send( "proceed_to_#{method.to_s}", with )
  end
end

# app/lib/collectable.rb
# This module auto-magically extends models (AR subclasses) with a collector
# module named ModelsCollector if it exists. This collector is designed to
# contains all heavy queries to be done by the model.
# This way, Model class file have only to deals with basic scopes.
module Collectable
  def inherited( klass )
    super
    collector = "#{klass.name.pluralize}Collector"
    Object.const_defined?( collector ) &&
      klass.extend( Object.const_get collector )
  end

  def query_for( method, with: {} )
    self.send( "query_for_#{method.to_s}", with )
  end
end

# app/lib/presentable.rb
# This module auto-magically includes a presenter module (named
# ModelsPresenter) inside models (AR subclasses) if it exists. This presenter
# is designed to contains all needed methods to be accessed inside views.
# This way, Model class file stay out of presenting stuff.
# A syntactic sugar is supplied : presenting :my_method { |args| method core }
module Presentable
  def inherited( klass )
    super
    presenter = "#{klass.name.pluralize}Presenter"
    Object.const_defined?( presenter ) &&
      klass.include( Object.const_get presenter )
  end

  self.class.send( :alias_method, :presenting, :define_method )
end
```

Processor modules

When a controller receive a html request, it sends a message to a model to perform an action. In case of a simple action, default model methods are called. When a more complex action has to be performed, the call is catched by a processor extended/included in the model. The naming convention for processor methods is `proceed_to_xxx` . Example : `@stay.proceed_to_update` or `Stay.proceed_to_create` .

```
# app/processors/stays_processor.rb
module StaysProcessor
  module ClassMethods
    def proceed_to_create( context = {} )
      new( context[:params] ).save
    end
  end

  module InstanceMethods
  def proceed_to_update( context = {} )
    update( context[:params] )
  end
end
```

Collector modules

These modules are concerned with « collecting » data from models. The naming convention is `query_for_xxx` where `xxx` is a controller method (and a view name).

```
# app/collectors/stays_collector.rb
module StaysCollector
  def query_for_index( context = {} ) # define a method for any basic op.
    context[:rental_place].stays.where( "actual_period && ?", context[:period] )
  end

  def query_for_show( context = {} ) # define a method for any basic op.
    context[:rental_place].stays.findbyid( context.fetch( :id, false ) )
  end

  private
  def findbyid( id )
    raise "No id provided !" unless id
    find_by_id( id )
  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.

Presenter modules

```
# app/presenters/stays_presenter.rb
module StaysPresenter # all these submodules are auto-included
  presenting :abstract do # see syntactic sugar above...
    # interesting presentation of stay abstract
  end

  presenting :full_abstract do |args|
    # wonderful method to present a full stay abstract for edit view
  end
end
```

Great separation of concerns, controller is presenter agnostic since it calls a method on the model. From within the views, we can call these new methods : `<%= @stay.abstract %>` . Hence, the helpers only « shape » the contents.

Controllers

In rails, we generally permit parameters within the controller. But I think this is curious, since its a model stuff.

A way to extract 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

# app/controllers/stays_controller.rb
def create # or anything else.
  @stay = Stay.new( permitted_params )
  # more stuff
end
```

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