Informe de refactorización para proyecto de Ingeniería de Software

Año: 2021
Grupo: 1

Integrantes del grupo:

-Axel Alcaraz (gaxel11)
-Gaspar Bosch (GasparB123)
-Emiliano Posse Irigoyen (emi382)

Repositorio:

https://github.com/emi382/fork-sistemas2021

Introducción:

Este informe tiene como objetivo documentar la refactorización del código de nuestro proyecto. La refactorización es un proceso que tiene como objetivo, a partir de un código ya funcionando, estructurarlo de mejor forma para que su comprensión, mantenimiento, y futura expansión sean mas simples.

Gran parte de la refactorización, es identificar "bad smells", expresión que se refiere a ciertos problemas comunes en la estructura de un código, por ejemplo en métodos, clases, o comunicación entre clases. Luego estos bad smells, los puedo solucionar a partir de diferentes métodos de refactorización, como puede ser poner un bloque de código en su propio método, pasar métodos de una clase a otra, etc.

Bad smells en nuestro código:

Luego de hacer un análisis inicial de nuestro código encontramos varios bad smells en distintas clases:

En app.rb:

-Algunos de nuestros paths son muy largos, tienen mas de 10 lineas.
-Se duplica mucho el código para guardar un objeto de alguno de los modelos. (Duplicated code)

```
# if saved, go back to homepage
if survey.save
  [201, { 'Location' => "surveys/#{survey.survey_id}" }, 'Created']
  redirect '/'
else
  [500, {}, 'Internal Server Error']
end
```

Este codigo se repetía para todos los posts donde se guardaba un nuevo objeto

Solución:

Ambos problemas se solucionaron al pasar el código para guardar objetos, a su propio método. Es decir, aplicamos la técnica de extraer método.

```
def save_or_error(object)
  if object.save
    [201, {}, 'Created']
    redirect back
  else
    [500, {}, 'Internal Server Error']
  end
end
```

Le sacamos el location también, ahora dice created en la misma página antes de redireccionar. Antes de sacarlo, el location no nos estaba haciendo nada por razones no determinadas.

-La clase app.rb en general es muy grande, tiene 158 líneas. (Large class)

Solución:
A determinar, por ahora no sabemos.

En survey.rb:

-filter_by_date tiene mas de 13 líneas. (Long method)
-Algunas líneas son muy grandes

```
def self.filter_by_date(start_date, finish_date)
  survey_struct = Struct.new(:career_id, :survey_id)
  surveys = Survey.all
  survey_array = []
  i = 0
  surveys.each do |survey|
    next unless DateTime.parse(start_date) < DateTime.parse(finish_date)

    next unless survey.created_at >= DateTime.parse(start_date) && survey.created_at <= DateTime.parse(finish_date).change(
      hour: 23, min: 59, sec: 59
    )
    next unless Survey.date_within_range(survey, start_date, finish_date)

    survey_array[i] = survey_struct.new(survey.career_id, survey.survey_id)
    i += 1
  end
  survey_array
end
```

El metodo que teníamos para filter_by_date era grande y complejo, con mucha de la complejidad dada por la condición para que la fecha estuviera en el rango

Solucionamos ambas cosas extrayendo un método afuera de filter_by_date, el método date_within_range. Es decir, aplicamos nuevamente la técnica de extrar método.

```
def self.date_within_range(survey, start, finish)
  valid = DateTime.parse(start) < DateTime.parse(finish)
  in_range = survey.created_at >= DateTime.parse(start) &&
             survey.created_at <= DateTime.parse(finish).change(hour: 23, min: 59, sec: 59)
  valid and in_range
end
```

Extraer este método nos dejó filter_by_date con 10 líneas, mucho mas razonable. También redujo mucho la complejidad del método, hay menos condiciones que mirar en filter_by_date.

Una vez terminado este análisis inicial, procedemos a usar Rubocop.

Introducción a Rubocop:

Rubocop es una herramienta muy útil para la refactorización. Tiene muchos cops (policías) que buscan ofensas de estilo, código no usado, nos dan métricas de nuestro código, etc.

Rubocop usa la guía de estilo de Ruby para esto, una guía que nos dice como escribir nuestro código de Ruby de una forma que sea mas leíble, tanto para nosotros como para otras personas. Sin embargo, seguir las guías de estilo muy exactamente puede no ser lo que queramos, y Rubocop toma esto en cuenta, dandonos la opcion de desactivar y activar los diferentes cops, tanto en general como para ciertos archivos o carpetas.

En nuestro grupo, usamos rubocop para analizar todas las ofensas de estilo que tuvieramos, corregir automáticamente las que se pudiera, y luego corregir a mano el resto. En total hubo 3 etapas en nuestro uso de Rubocop:

Primera parte:

En el primer análisis con rubocop, vimos que teníamos muchas ofensas relacionadas a convenciones para escribir codigo ruby, según la style guide de ruby.

```
36 files inspected, 696 offenses detected:
```

Luego de analizarlas un poco, hicimos autocorrect.
Esto soluciono la mayoría de las ofensas relacionadas a estilo.

Segunda parte:

Ahora en el segundo análisis, nos quedaron ofensas un poco más complejas.

```
36 files inspected, 55 offenses detected:
```

Algunas de estas ofensas eran asignaciones sin uso, que simplemente nos habíamos olvidado de sacar. Eliminamos las asignaciones que no tenían uso.

Line #208 – **Warning:** Lint/UselessAssignment: Useless assignment to variable - `careers`. Did you mean `career`?

```
    careers = Career.all
```

Otras eran relacionadas a los nombres de métodos y variables, que tienen que ser con snake_case y nosotros habíamos hecho con camelCase. Cambiamos los nombres para conformar con las convenciones.

Line #68 – **Convention:** Naming/VariableName: Use snake_case for variable names.

```
    erb :finish, locals: { career: Career.bestCareerCalc(careerArray), careers: careerArray.sort_by do |career|
```

Había muchas lineas, usualmente comentarios, que eran muy grandes. Lo solucionamos haciendolas mas chicas de ser posible, o diviendolas en más líneas.

Line #197 – **Convention:** Layout/LineLength: Line is too long. [121/120]

```
  # shows the outcomes associated to the choice that is associated to a question. Also has create outcome functionalities
```

En los tests, teniamos una línea extra, MiniTest::Unit::TestCase, abajo de la declaración del test. Rubocop nos decía que estaba siendo usada en contexto void por lo que la sacamos, y el funcionamiento no se vio afectado.

Line #5 – **Warning:** Lint/Void: Variable `TestCase` used in void context.

```
  MiniTest::Unit::TestCase
```

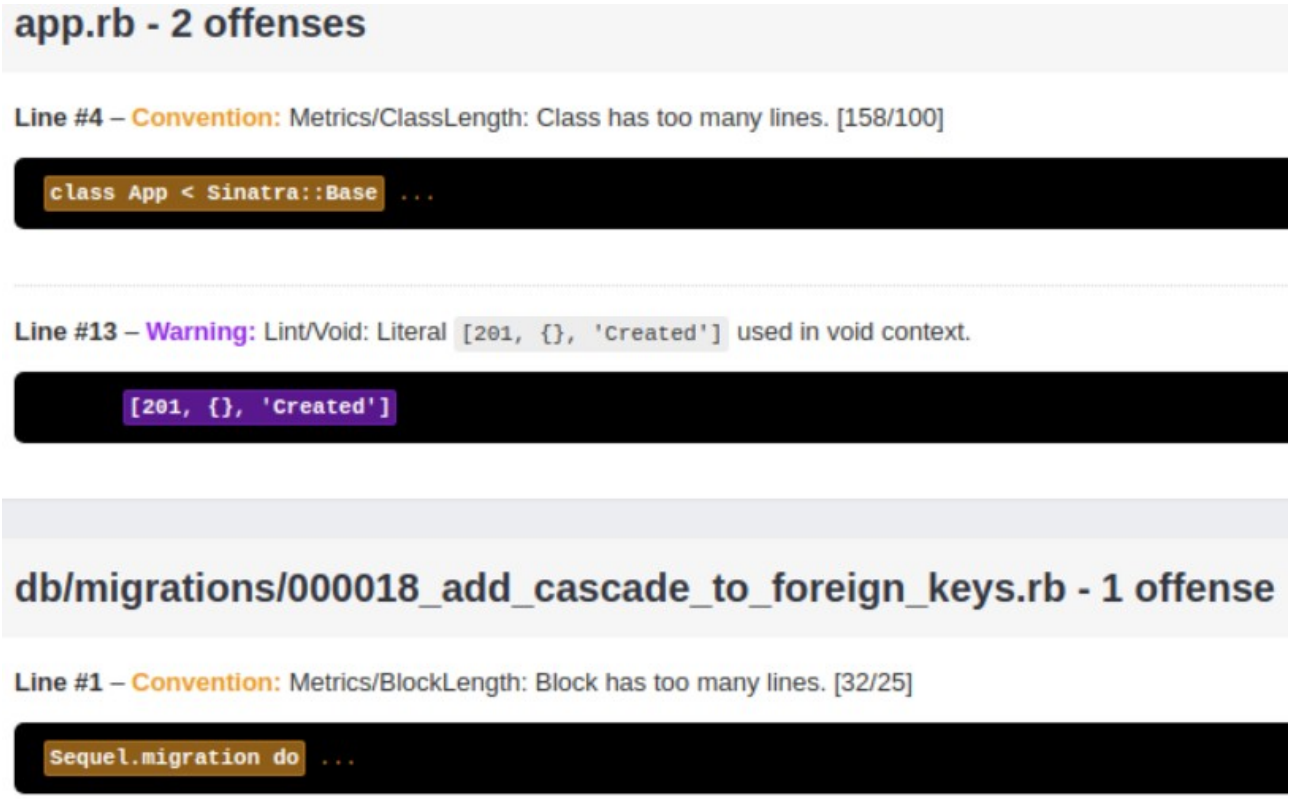
En test_helper, teniamos class MiniTest::HooksSpec, lo cual por convención debía ser separado en module y class.

Line #17 – **Convention:** Style/ClassAndModuleChildren: Use nested module/class definitions instead of compact style.

```
class Minitest::HooksSpec
```

Tercera parte:

Ahora para el último análisis, nos quedaron las ofensas que no supimos como corregir.



En la primera, nos dice que la clase app tiene muchas líneas, lo cual es cierto. Pero no sabemos como solucionar esto.

En la segunda, nos dice que [201,{},'Created'] lo cual es parte del método que guarda un objeto y nos dice que ha sido creado, esta siendo usado en contexto void. Pero no sabemos porqué nos dice esto.

En la tercera, tengo una migración demasiado grande, pero como es una migración cambiarla podria causar problemas asi que la dejamos así.

Conclusión:

Refactorizar nuestro código lo hizo mas leíble, menos repetitivo, y de más facil entendimiento. El uso de Rubocop nos ayudo mucho a identificar y resolver ofensas de estilo, código no necesario y métricas muy grandes. Aunque nos quedaron 3 ofensas sin resolver, nuestro código mejoro bastante en calidad.