UNIVERSIDAD NACIONAL DE RIO CUARTO

INGENIERÍA DE SOFTWARE

INFORME DE SPRINT

REFACTORING

TEAM 5
Scrum Master y developer
Juan Manuel YACHINO
Developers
Emiliano BAEZ
Leonardo GAITÁN

20 de Octubre, 2020

Índice

Contents

1 Introducción

En el presente informe se describe el trabajo realizado durante el Sprint de refactorización de código , que fue llevado a cabo durante 15 dias , comenzando el martes 6 de Octubre y terminando el dia 20 del mismo mes.

1.1 Herramientas usadas

Utilizamos herramientas que ya veniamos usando desde antes y algunas nuevas:

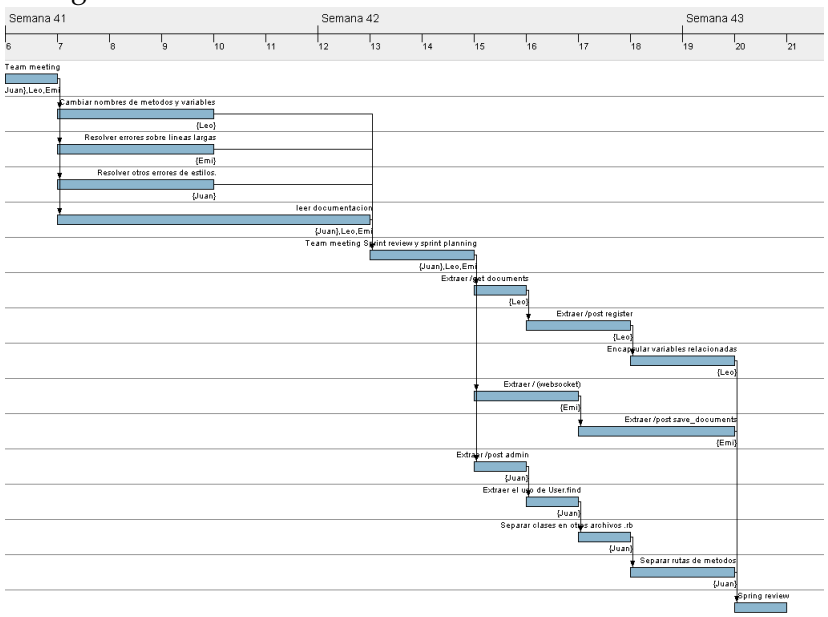
Github. Utilizamos branches para cada historia de usuario, para que cada developer pueda trabajar sin conflictos sobre el mismo archivo en su propia rama. Luego cada developer hizo pull request a master para mergear su trabajo con el de los demas.

Pivotal Tracker. Es una herramienta de gestion de proyectos agiles donde cargamos las historias de usuario, las asignamos a cada miembro del equipo y vinculamos a github, entonces cuando se hace un commit referenciando una historia de usuario , esta se actualiza automáticamente en el tablero de Pivotal Tracker.

Rubocop Es una herramienta para revisar y corregir errores de estilos en Ruby. Está basado en una guia de estilos y convenciones escritas por la comunidad. Además de reportar errores permite corregir varios de estos automáticamente.

1.2 Organización del Sprint

Para este sprint acordamos dividirlo en 2 etapas, como lo muestra el siguiente diagrama de Gantt:



1.2.1 Etapa 1: Rubocop

Esta etapa comienzó con la ejecucion de la herramienta rubocop, luego hicimos una reunión para identificar los errores que la herramienta no resolvió automaticamente y a partir de estos errores generamos y estimamos historias de usuario que el Scrum Master repartió entre los developers.

Al mismo tiempo que trabajamos resolviendo esos errores , continuamos leyendo la documentacion y el libro, tarea necesaria para llevar a cabo la segunda etapa de refactorización.

1.2.2 Etapa 2: Refactoring

Luego de la reunion con los profesores y de haber leído la documentacion, hicimos otra reunión de planificación para la segunda etapa del Sprint.

En esta etapa identificamos las cosas que necesitabamos refactorizar del codigo, luego utilizando algunas de las técnicas provistas en el libro, generamos y estimamos historias de usuario y el scrum master las asignó a cada miembro del equipo.

2 Code Smells que detectamos en nuestro código

Luego de leer el capitulo 3 del libro de Refactoring , "Bad smells in Code" , pudimos detectar cosas en nuestro codigo que necesitaban una refactorizacion urgente.

2.1 Código duplicado

Encontramos muchas lineas de codigo que hacian lo mismo o redundantes e innecesarias.

2.2 Clases largas

En nuestro código teniamos todas las rutas y métodos dentro de la clase App.rb , lo que hacia que esta fuera demasiado larga, como ya nos había advertido el analisis de Rubocop.

2.3 Modularizacion casi nula

Al tener todas las rutas en la clase App.rb , mezcladas con la lógica y los accesos a la base de datos de entidades distintas, notamos que la modularizacion de nuestro software era casi nula, necesitabamos llevar a cabo una refactorización en donde tener separadas las rutas de los metodos y la logica para cada entidad.

2.4 Comentarios en exceso

Encontramos que usabamos muchos comentarios para decir que hacía un método o una línea de código.

3 Salida del analisis de Rubocop

Cuando ejecutamos rubocop y le pedimos que analice nuestro App.rb , nos devolvio los siguientes resultados.

3.1 Primer analisis de rubocop

El resultado del primer análisis fue bastante largo, contenía errores que rubocop pudo solucionar automáticamente, como errores de identación o espacios en blanco.

El resultado completo puede leerse dando click Aqui o accediendo al repositorio github, al tratarse de un archivo tan largo (500 lineas) no pudimos agregarlo al informe.

En resumen, se encontraron 153 errores de estilo , de los cuales 138 se podían resolver automáticamente.

3.2 Segundo Analisis de rubocop

Luego de aplicar las correciones automáticas de Rubocop, volvimos a correr el análisis sobre nuestro App.rb y obtuvimos estos resultados:

```
Inspecting 1 file
W

Offenses:

app.rb:1:1: C: Style/FrozenStringLiteralComment: Missing frozen
    string literal comment.
require 'sinatra/base'
^
app.rb:8:1: C: Metrics/ClassLength: Class has too many lines.
    [189/100]
class App < Sinatra::Base ...
^^^^^^^^^^^^^^^^^^^^^^^^^^
app.rb:8:1: C: Style/Documentation: Missing top-level class
    documentation comment.
class App < Sinatra::Base
^^^^^
app.rb:35:7: C: Naming/PredicateName: Rename is_admin to admin?.
  def is_admin
      ^^^^^^^^
app.rb:37:5: C: Naming/VariableName: Use snake_case for variable
    names.
    @isAdmin = true if user == 'admin'
    ^^^^^^^^
app.rb:40:7: C: Naming/MethodName: Use snake_case for method names.
  def findConnection(user)
      ^^^^^^^^^^^^^^
app.rb:79:121: C: Layout/LineLength: Line is too long. [122/120]
      user = User.new(name: params['name'], email: params['email'],
          username: params['username'], password: params['psw'])

app.rb:122:7: C: Naming/VariableName: Use snake_case for variable
    names.
      @isAdmin = true
      ^^^^^^^^
app.rb:133:5: W: Lint/UselessAssignment: Useless assignment to
    variable - user.
    user = User.first(username: params[:users])
    ^^^^
app.rb:185:9: C: Naming/VariableName: Use snake_case for variable
    names.
```

```
        socketsToBeNotified = []
        ^^^^^^^^^^^^^^^^^^^^
app.rb:187:35: C: Naming/BlockParameterName: Only use lowercase
    characters for block parameter.
        settings.userlist.each { |taggedUser| unless
            findConnection(taggedUser).nil? then socketsToBeNotified
            << (findConnection(taggedUser)) end }
                               ^^^^^^^^^^
app.rb:187:35: C: Naming/VariableName: Use snake_case for variable
    names.
        settings.userlist.each { |taggedUser| unless
            findConnection(taggedUser).nil? then socketsToBeNotified
            << (findConnection(taggedUser)) end }
                               ^^^^^^^^^^
app.rb:187:69: C: Naming/VariableName: Use snake_case for variable
    names.
        settings.userlist.each { |taggedUser| unless
            findConnection(taggedUser).nil? then socketsToBeNotified
            << (findConnection(taggedUser)) end }
                                                                ^^^^^^^^^^
app.rb:187:91: C: Naming/VariableName: Use snake_case for variable
    names.
        settings.userlist.each { |taggedUser| unless
            findConnection(taggedUser).nil? then socketsToBeNotified
            << (findConnection(taggedUser)) end }
                                                                          ^^^^^^
app.rb:187:121: C: Layout/LineLength: Line is too long. [147/120]
        settings.userlist.each { |taggedUser| unless
            findConnection(taggedUser).nil? then socketsToBeNotified
            << (findConnection(taggedUser)) end }

app.rb:187:130: C: Naming/VariableName: Use snake_case for variable
    names.
        settings.userlist.each { |taggedUser| unless
            findConnection(taggedUser).nil? then socketsToBeNotified
            << (findConnection(taggedUser)) end }

app.rb:190:9: C: Naming/VariableName: Use snake_case for variable
    names.
        socketsToBeNotified.each { |s| s.send('han cargado un nuevo
            documento!') }
        ^^^^^^^^^^^^^^^^^^^^

1 file inspected, 17 offenses detected, 3 offenses auto-correctable
```

Listing 1: Resultados del segundo analisis de rubocop

A partir de estos errores creamos las historias de usuario de la primera etapa del sprint.

3.3 Tercer analisis de rubocop

Despues de resolver los errores restantes manualmente, corrimos por tercera vez Rubocop y nos devolvió estos resultados:

```
Inspecting 1 file
.

1 file inspected, no offenses detected
```

Listing 2: Resultados del tercer analisis de rubocop

Despues de la segunda etapa del Sprint, volvimos a analizar App.rb y las nuevas clases que creamos y corregimos los errores.

4 Refactoring: Cambios realizados

Comenzamos haciendo cambios que rubocop nos pedia hacer para respetar sus convenciones, el primero de estos fue reducir el tamaño de la Clase App, para lo cual inicialmente creamos clases Document y User para agrupar los metodos y rutas de cada entidad en su clase.

Más adelante nos dimos cuenta que era mejor tenerlas en archivos separados y separar rutas de metodos que aplican lógica y de accesos a la base de datos.

4.1 Extraer codigo en metodos

Teníamos demasiado código en nuestras rutas, entonces creamos métodos cortos con nombres claros que nos permita reusar código y ademas prescindir del uso de comentarios.

4.2 Encapsular variables en una misma estructura

Muchas veces nos encontramos que al querer llamar a un método teniamos que pasarle muchos parametros, para solucionar esto utilizamos unas estructuras que nos permitian agrupar varias variables en una sola y acceder a ellas en tiempo constante.

```
hash = { name: params['name'],
         email: params['email'],
         username: params['username'],
         password: params['psw'] }
    redirect '/login' if UserServices.register(OpenStruct.new(hash))
```

Listing 3: Ejemplo del uso de la estructura

4.3 Mover las clases a archivos .rb individuales

Inicialmente teniamos todo el código de nuestro programa en un solo archivo .rb ,para seguir la convención decidimos moverlas a otros archivos. Agrupamos estas nuevas clases en carpetas :

Controllers. Agrupa las clases que contienen las rutas de la aplicación, tenemos users_controllers.rb y documents_controller.rb , que manejan las rutas de la entidad User y Document respectivamente.

Models. Contiene las clases que representan a las entidades de la base de datos. En estas agregamos metodos para acceder y modificar campos de la base de datos.

```
def self.promote_to_admin(user)
    user.update(type: 'admin')
  end
```

Listing 4: Ejemplo de un metodo que solo accede a la base de datos

Services. Contiene las clases con metodos en donde se utiliza lógica, por ejemplo , determinar si un usuario ya existe antes de registrarlo, verificar si una contraseña es correcta.

```
def self.validate_login(user, pass)
    user = User.find_by_username(user)
    return false unless user && user.password == pass

    true
  end
```

Listing 5: Ejemplo de un metodo de la clase userServices que verifica el login de un usuario al sistema.

Dejamos a la clase App.rb como nuestra clase driver y en esta dejamos al método before y la ruta raiz /.

5 Bibliografía

1. J. Fields, S. Harvie, M. Fowler, K. Beck, *Refactoring. Ruby Edition.* , (2009). (Capítulo 3: "Bad Smells in Code", Capítulo 6: "Composing methods" , Capítulo 7: "Moving features between objects" y Capítulo 8: "Organizing data")
2. Guia de estilos de ruby en la que se basa Rubocop
   `https://github.com/rubocop-hq/ruby-style-guide`