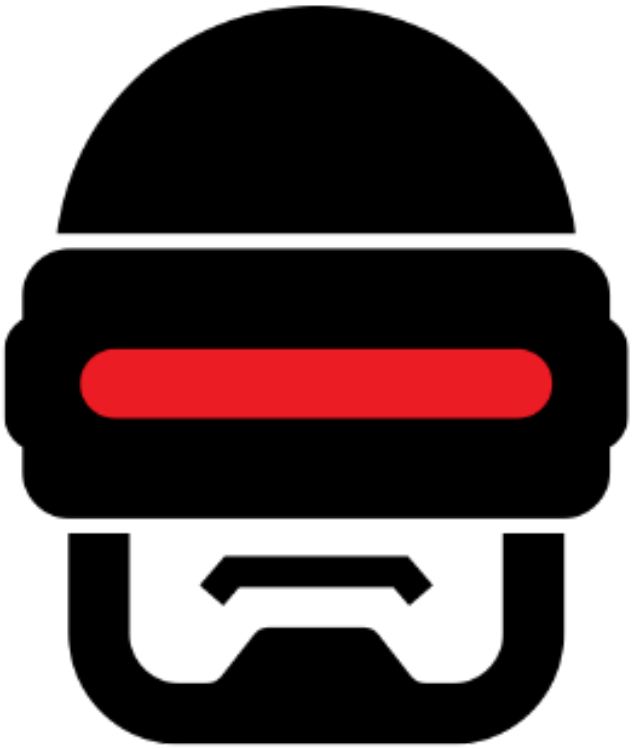
REFACTORING

Taller Ingeniería de Software



Luca Carignano,
Federico Gutierrez

20 de octubre de 2020

INTRODUCCIÓN

Comenzaremos explicando qué es el refactoring. Como dice en el libro, 'Refactoring: Ruby Edition', la refactorización o refactoring, es el proceso de cambiar un sistema de software de tal manera que no altera el comportamiento externo del código, pero mejora su estructura interna. Es una forma disciplinada de limpiar el código que minimiza las posibilidades de introducir errores. En esencia, cuando se refactoriza, se está mejorando el diseño del código luego de que haya sido escrito.

"BAD SMELLS" EN NUESTRO CÓDIGO.

Comenzaremos desarrollando los tipos de "bad smells" que encontramos en nuestro código, acompañado con un ejemplo sacado de este, y luego nombraremos otros tipos de errores que también pueden ocurrir, aunque este no es el caso. También haremos hincapié en los errores detectados y corregidos automáticamente por rubocop.

AUTOCORRECCIONES:

Al comenzar con el uso de rubocop, lo primero que hicimos fue ejecutar la misma, y crear un reporte con todos los errores detectados. Luego, procedimos a la autocorrección de los mismos mediante el uso de la herramienta. Ahí pudimos notar que la mayoría de los errores habían desaparecido, y generamos otro reporte con los errores que rubocop no podía corregir automáticamente.

DUPLICATED CODE:

Esto ocurre cuando la misma porción de código está en más de una ocasión y debe encontrarse la forma de unificarlos para ahorrar código. La más simple es cuando una misma expresión está en dos métodos distintos en la misma clase. Otros errores son cuando la misma expresión se encuentra en clases que no están relacionadas, también debe aplicarse métodos de refactorización para unificarlas.

Código anterior:

```
((request.path_info == '/makeadmin') || (request.path_info == '/adddoc') || (request.path_info == '/maketag'))
```

Código nuevo:

```
def request_path_admin
  ((request.path_info == '/makeadmin') || (request.path_info == '/adddoc') || (request.path_info == '/maketag'))
end
```

LONG METHOD:

Los programas objetos que funcionan mejor y tienen mejor tiempo de vida son aquellos que tienen métodos cortos. Este error también consiste en aquellos métodos que usan una gran lista de parámetros y variables temporales o poseen una línea del código muy extensa. Además explica que si un método necesita demasiados comentarios para explicar su funcionamiento debería cambiar el nombre de método basado en el comentario.

Código anterior:

```
@undocuments = Document.order(:date).reverse.where(delete: 'f', id:( Labelled.select( :document_id ).where( readed: 'f', user_id: user.id))).all
```

Código nuevo:

```
user = User.find(id: session[:user_id])
id = Labelled.select(:document_id).where(readed: 'f', user_id: user.id)
@undocuments = Document.order(:date).reverse.where(delete: 'f', id: id).all
```

LARGE CLASS:

Una clase demasiado larga puede generar otros errores de código duplicado y desorden.

La clase app es muy larga

DATA CLUMPS:

Data clumps ocurre cuando varios datos que son independientes se necesitan juntos en varios métodos o funciones, para corregir esto, es posible crear un objeto que reúna a estos datos que se necesitan juntos en varias funciones.

Código nuevo:

```
def view_noti
  if @current_user
    id = Labelled.select(:document_id).where(readed: 'f', user_id: @current_user.id)
    @noti = Document.where(delete: 'f', id: id).count
  end
end
```

COMMENTS:

La documentación habla de que no es necesario escribir un comentario para explicar el funcionamiento de un método, este tendría que explicarse por sí solo a través de su nombre. Pero también sugiere que toda clase debe tener un comentario diciendo cual es su uso.

Código anterior:

```
class App < Sinatra::Base
```

Código nuevo:

```
# This is the main class of the system
class App < Sinatra::Base
```

Bad Smells Code, que no aparecieron en nuestro proyecto: LONG PARAMETER LIST, DIVERGENT CHANGE, SHOTGUN SURGERY, FEATURE ENVY, PRIMITIVE OBSESSION, CASE STATEMENTS, PARALLEL INHERITANCE HIERARCHIES, LAZY CLASS, SPECULATIVE GENERALITY, TEMPORARY FIELD, MESSAGE CHAINS, MIDDLE MAN, INAPPROPRIATE INTIMACY, ALTERNATIVE CLASSES WITH DIFFERENT INTERFACES, INCOMPLETE LIBRARY CLASS, DATA CLASS, REFUSED BEQUEST, METAPROGRAMMING MADNESS, DISJOINTED API, REPETITIVE BOILERPLATE.

REFERENCIAS

1. http://www.r-5.org/files/books/computers/languages/ruby/style/Jay_Fields_Shane_Harvie-Refactoring_Ruby_Edition-EN.pdf
2. https://github.com/rubocop-hq/ruby-style-guide
3. https://refactoring.com/