

**Ingeniería de Software (3304)**

**Refactorizacion de codigo del sistema de notificaciones de la UNRC**

Autores:

* Fernández, Facundo
* Filippa, Nahuel
* López Matias

20/10/2020

**Detección de Code Smells**

“Code Smells” es un término que se utiliza para referirse a los problemas de un código en cuanto a funcionalidad, eficiencia, legibilidad, convenciones, etc. Si bien estos generalmente no son problemas que generen errores inmediatos o directos en el funcionamiento del programa, suelen ser causa de problemas a nivel de eficiencia o a la hora de extender o intentar reutilizar código. De modo que es una buena práctica solucionarlos, para obtener un producto de mayor calidad.

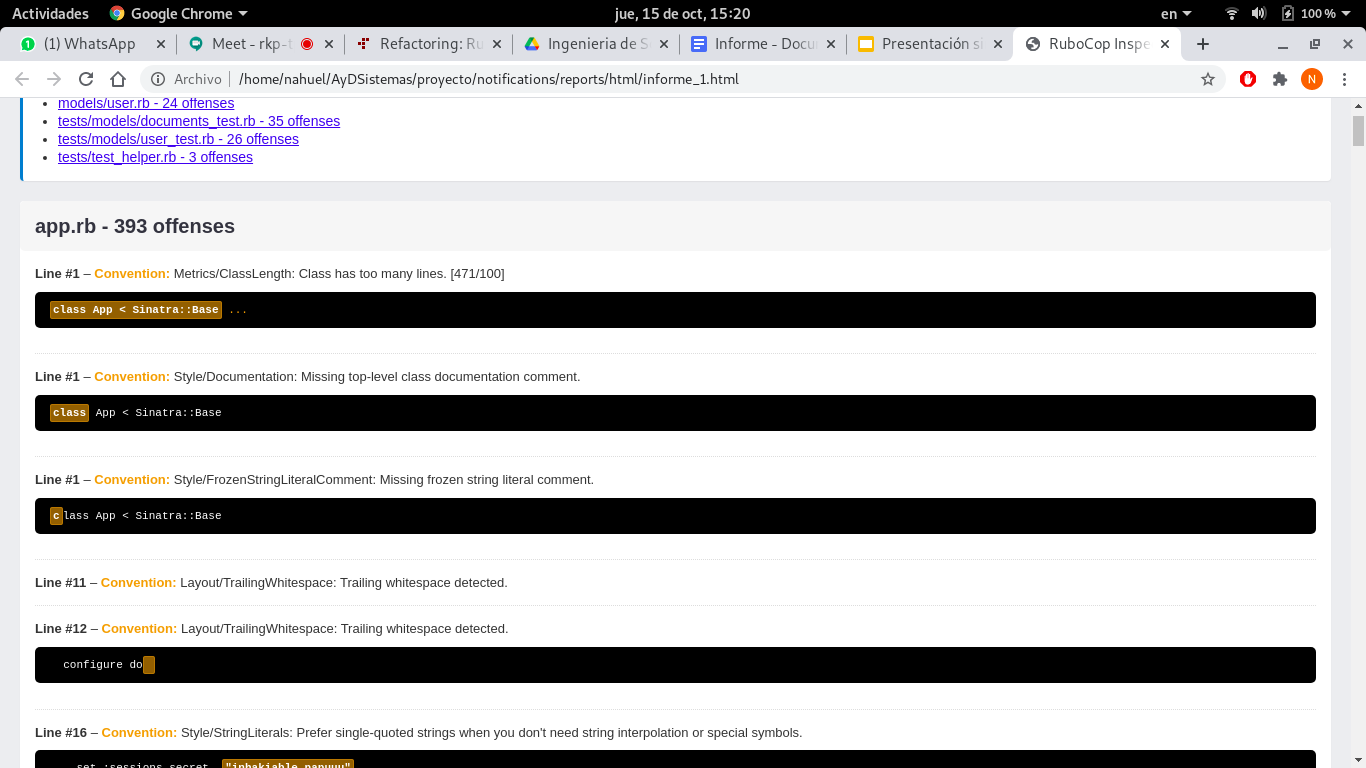
La primera tarea del equipo fue la de identificar Code Smells en el código “principal” de la aplicación, llamado “App.rb”, de este modo se lograron detectar algunos muy evidentes, como por ejemplo “Large class” o Clase Grande, dado que se tenían alrededor de 500 líneas de código en la clase principal, asi como tambien algunos Smells menos evidentes, como por ejemplo el largo de los nombres de algunos métodos.

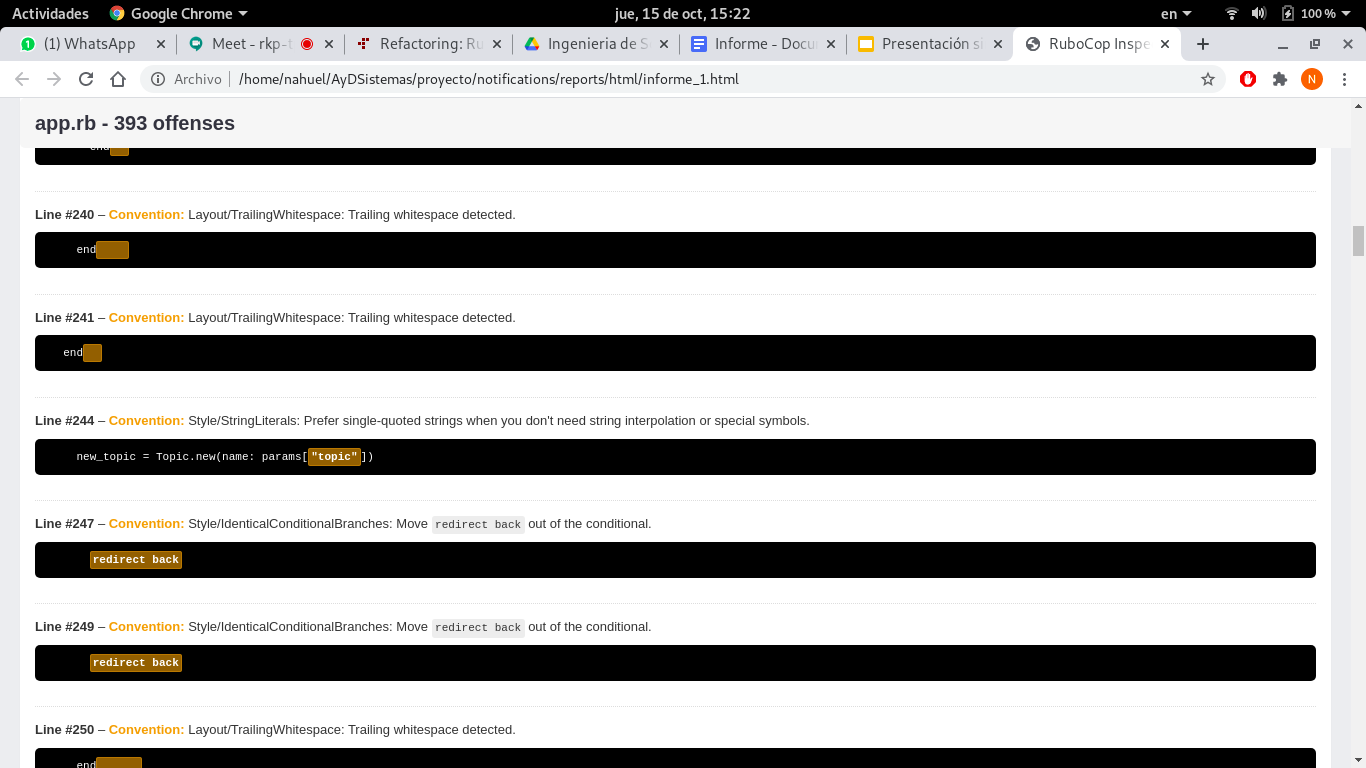
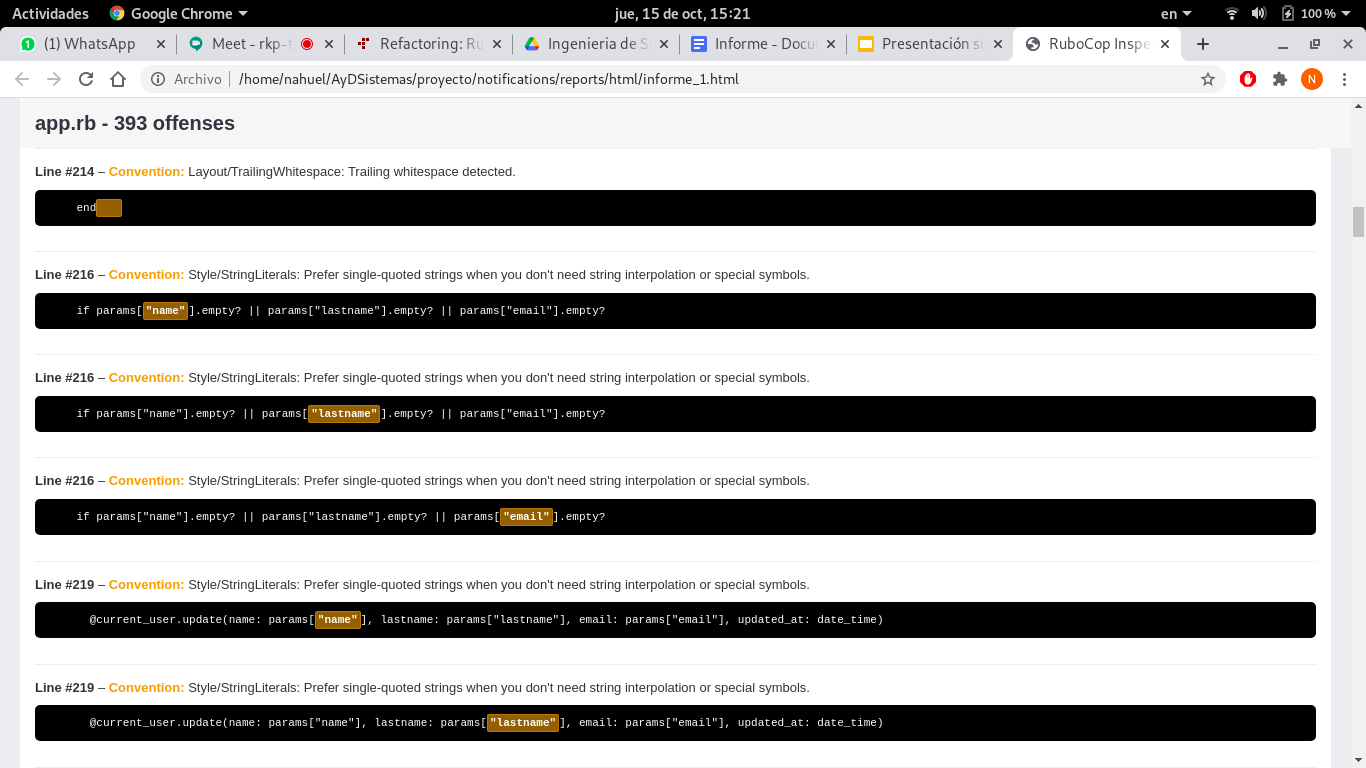
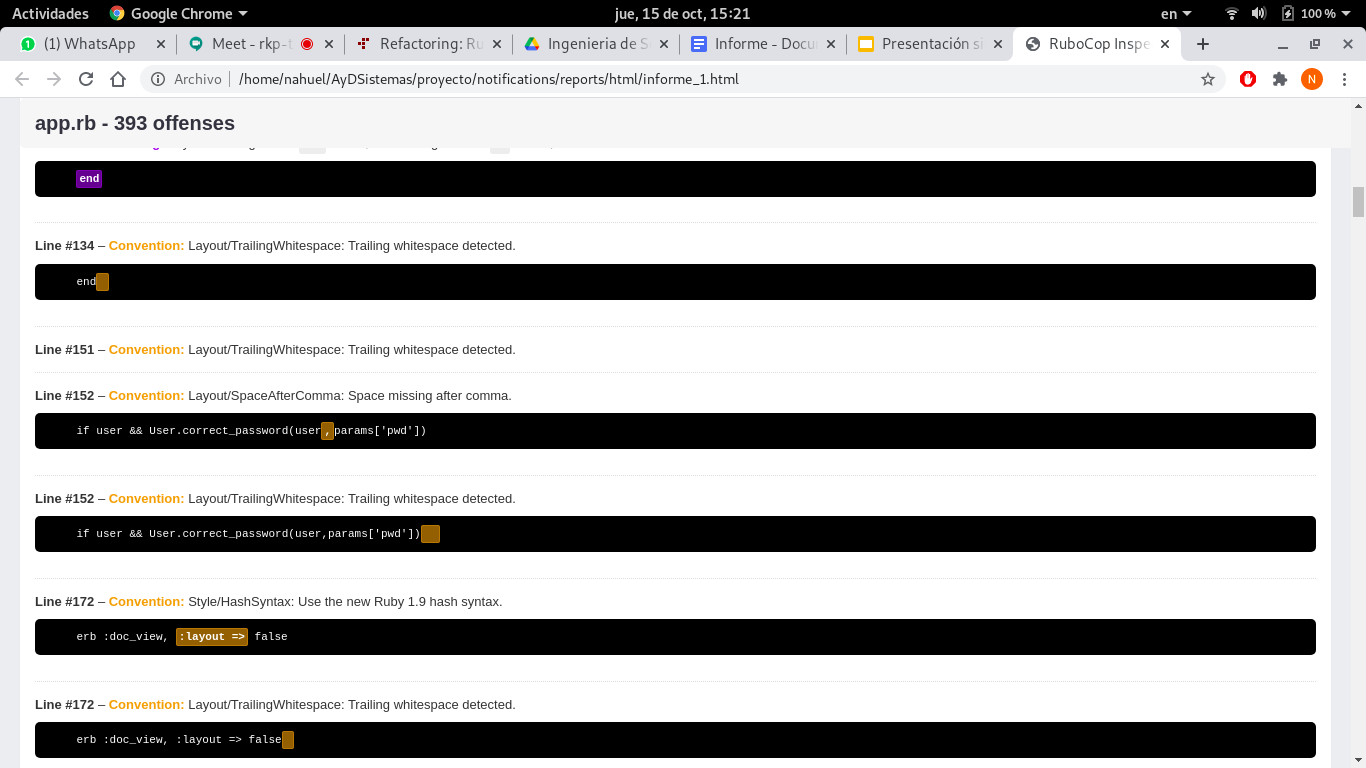
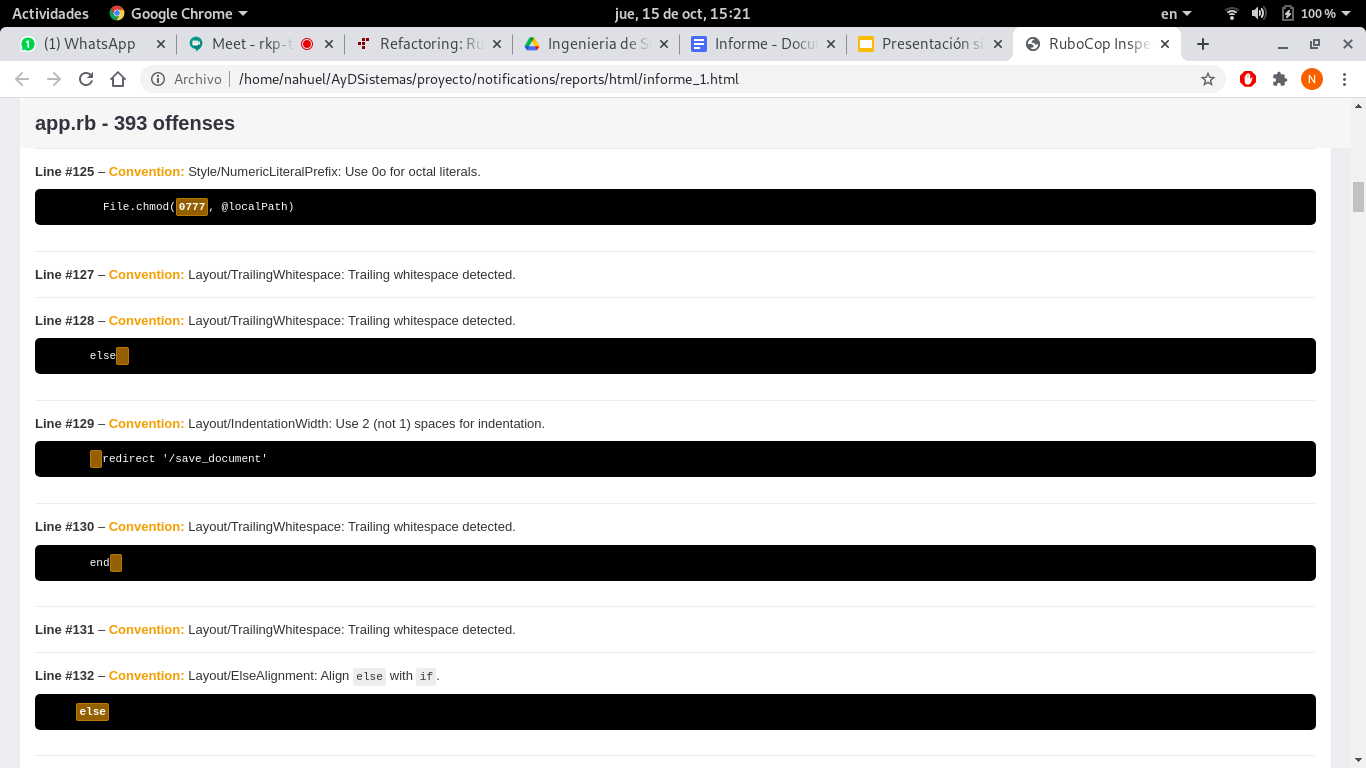
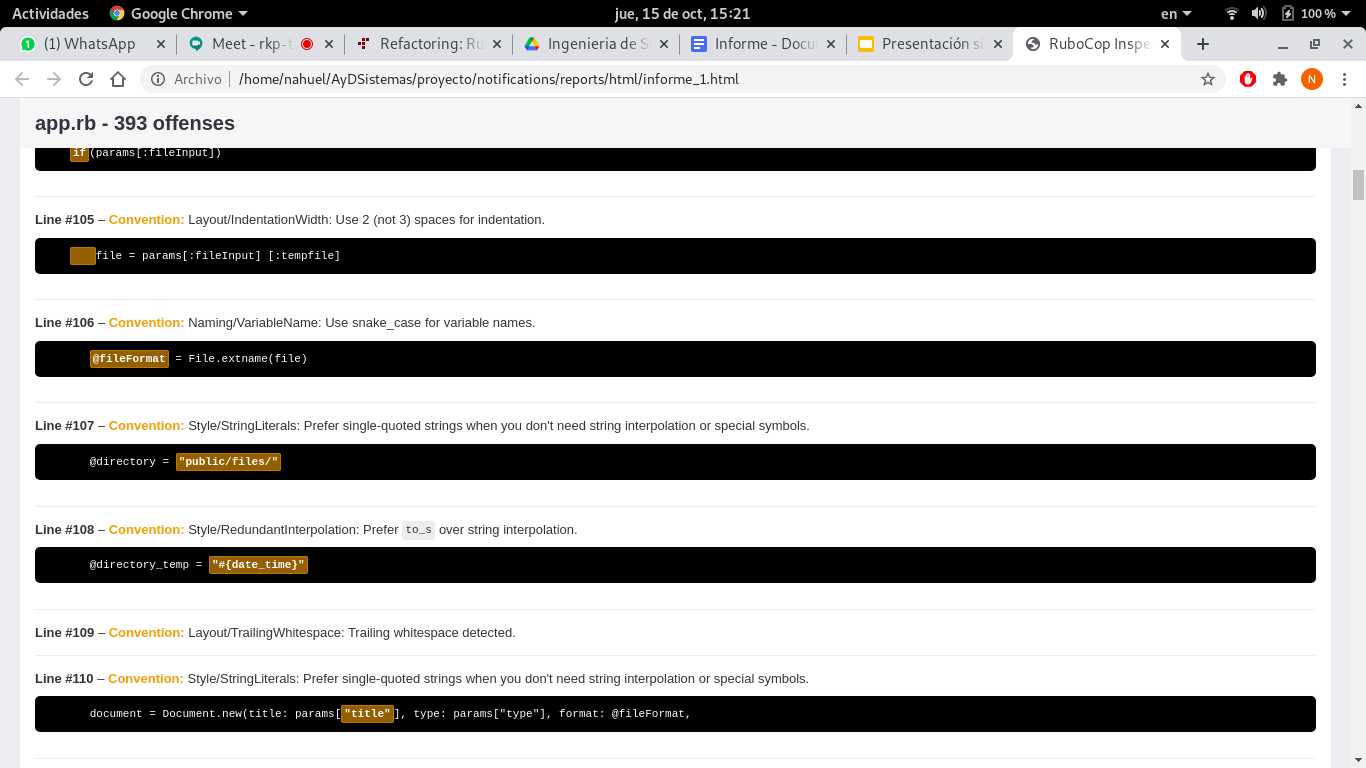
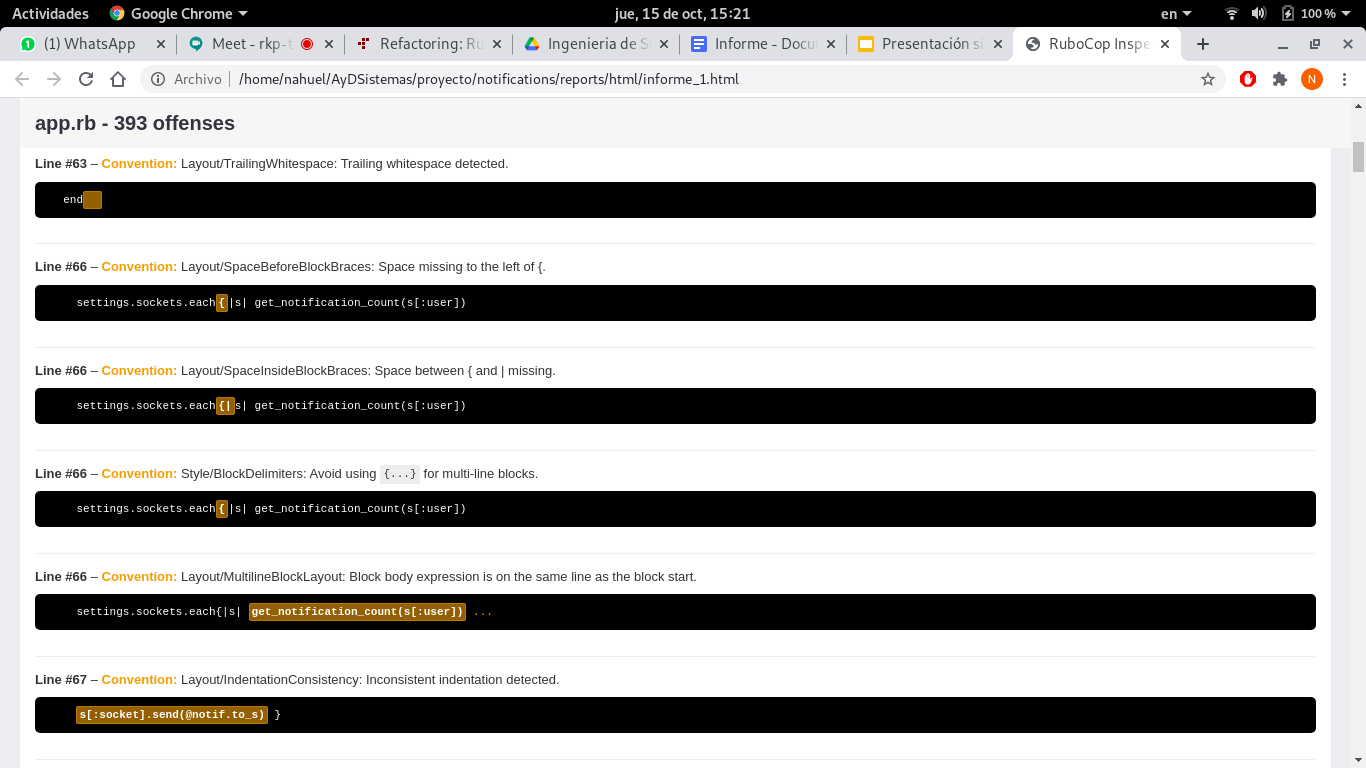
Para complementar la tarea que se había realizado de forma manual, se utilizó la herramienta Rubocop, recomendada por los profesores, que inicialmente arrojó la cantidad de 393 ofensas o infracciones, todas ellas autocorregibles. Algunos errores destacables, fueron: La clase App.rb era demasiado larga, que había sido previamente identificado, algunos condicionales “if” con una condición negativa que podían reemplazarse con “unless” más una condición positiva, lo cual fue interesante para varias situaciones. Fuera de eso casi todos los errores fueron espacios en blanco o gramaticales. Finalmente se procedió a realizar la autocorrección utilizando Rubocop.

Luego de realizada la primera corrección, se procedió a volver a inspeccionar el codigo utilizando Rubocop. Esta acción arrojó 21 errores, en general, menores, como nombres de variables muy cortos, nombres de métodos que contenían la palabra get\_, nombres de algunos métodos muy largos, etc. estos errores menores fueron corregidos manualmente.

Una vez finalizada esta corrección gramatical y de convenciones, el equipo se dio a la tarea de aplicar técnicas de refactoring para corregir el error más grande, que era que la clase App.rb era demasiado larga.

A continuación se muestran algunas capturas del informe generado.





**Refactorización**

En vista del elevado tamaño de la clase App.rb que tenía 471 líneas de código, se detectó un Code Smell llamado Large Class. Por esta razón nos dispusimos a aplicar algunas técnicas de refactoring. Principalmente se utilizó el movimiento de métodos (Move method) de la clase App.rb a los modelos a los que corresponden, por ejemplo:

Formato: <Metodo> → <Modelo al que fue movido>

* create\_user → User
* find\_user\_id →User
* find\_user\_dni → User
* find\_user\_email → User
* user\_checked\_document → Document
* user\_add\_favorite → Document
* delete\_doc → Document
* add\_topics → Document
* user\_del\_favorite\_document →Document
* check\_notification → Tag
* retrieve\_notifications → Tag
* notification\_count → Tag
* delete\_old\_views → Tag
* document\_of\_user → Tag
* find\_document\_user → Tag
* find\_document\_favorite → Tag

Otra técnica que se utilizó para ahorrar líneas de código y hacer el código más legible, fue “inline temp”, esta técnica consiste en utilizar menos variables temporales para hacer comparaciones o returns, reemplazandolas directamente por las expresiones que se les habían asignado previo a la refactorización.

Ejemplo:

new\_user = create\_user();

return new\_user;

↓

return create\_user();

Una técnica muy interesante encontrada y utilizada fue la de reemplazar condicionales con cláusulas de guarda (guard clauses). De este modo se crea un código mucho más legible y más corto.

Ejemplo:

def function()

if (!doc.nil?)

<código a ejecutar>

end

end

↓

def function()

return if doc.nil?

<código a ejecutar>

end

**Salvedades de algunas ofensas**

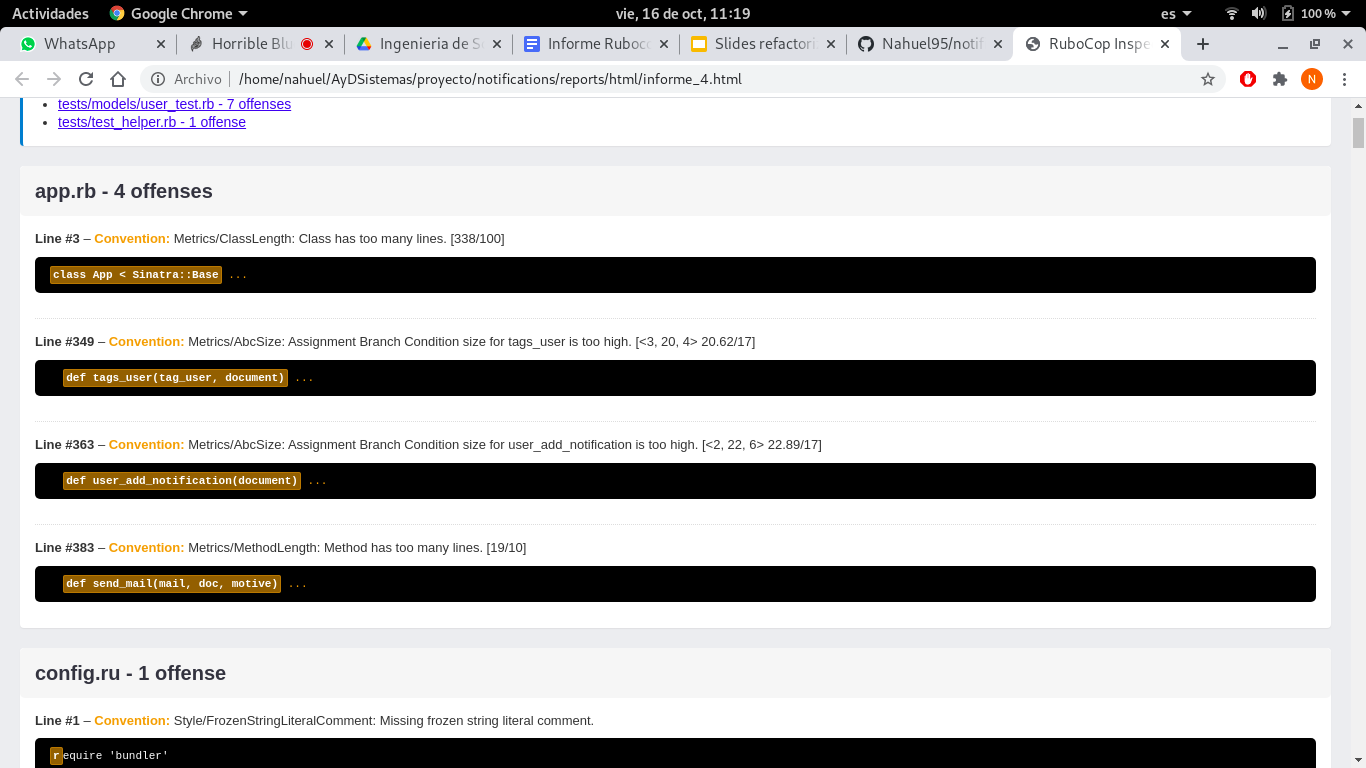
Luego de realizadas las modificaciones anteriormente nombradas, se solicitó otro informe a Rubocop, para analizar qué infracciones no se habían logrado corregir. Las 4 ofensas que aún permanecen son:

* Longitud elevada de la clase App.rb.
* Condiciones con guardas demasiado largas en 2 métodos (tags\_user y user\_add\_notification).
* Demasiadas líneas de código en el método send\_mail.

Estas infracciones se consideraron por separado y se llegó a la conclusión de que podía permitirse por ahora, que la longitud de la clase App.rb fuera alta, dado que fue reducida en un 28.24%, número que el equipo consideró que por ahora es un buen avance.

En cuanto a la longitud de las guardas en los condicionales, se decidió que sería más contraproducente crear una variable temporal para este fin, dado que se utilizaría una sola vez y quitaria legibilidad al código de todas formas.

En el método send\_mail, existe una cadena JSON, que es muy larga para ponerla toda junta en una línea, pero al ponerla ordenada y separada en renglones, hace que el método se alargue fuera de los estándares, por lo que se consideró que es preferible tener un método un poco más largo, dado que poner toda la cadena JSON en una sola línea, la haría muy incómoda de leer y/o interpretar.



**Conclusión**

En resumen, el análisis y refactorización del código ayudó al equipo a disminuir la cantidad de líneas de la clase principal (App.rb) de 471 a 338 líneas de código, es decir un 28,24%, que, si bien sigue siendo un número bastante mayor a la cantidad de líneas máximas de una clase, según la guía de estilo de Rubocop (100), se puede ver una reducción considerable de esta métrica.

Otra ventaja es que esto permitió modularizar el proyecto de mejor manera y lograr un código que ahora es mucho más legible debido a que sigue los lineamientos de la comunidad de Ruby.