Reporte de refactorización

Realizado por:

Avaro Jeremias - Cornejo Mateo - Marquez Maximo

I. Introducción

Este reporte ofrece detalles de la tarea número cuatro del taller de la materia “Ingeniería de Software”, donde realizamos refactorizaciones de código sobre los diferentes archivos que componen el juego “Astroledge” realizado en la materia “Análisis y Diseño de Sistemas”.

II. Trasfondo

La refactorización es un proceso en el cual se modifican y mejoran estructuras de código sin cambiar su comportamiento externo. Este proceso se basa en la idea de que, al mejorar la estructura interna del código, se incrementa su mantenibilidad, legibilidad y extensibilidad. En el libro "Refactoring Ruby Edition", este concepto es adaptado al contexto de Ruby, un lenguaje orientado a objetos que permite una escritura limpia y flexible, pero que también puede volverse confusa o compleja si no se organiza adecuadamente.

La refactorización se centra en pequeños y seguros pasos que minimizan el riesgo de errores, lo cual permite realizar mejoras continuas sin detener el flujo de desarrollo. El libro aborda varios conceptos clave en el proceso de refactorización:

- **Code Smells (Malos Olores de Código):** Estos son indicios de que hay algo en el código que podría mejorarse. Algunos ejemplos incluyen métodos o clases demasiado largas, lógica duplicada, y nombres de variables o métodos poco descriptivos. Identificar estos "malos olores" es crucial para saber qué y cómo refactorizar.

- **Pequeños Pasos:** Uno de los principios más importantes en refactorización es hacer cambios incrementales, de forma que cada paso mejore el código sin alterar su funcionalidad. Esto permite que el código se mantenga funcional en cada etapa de la refactorización y que se puedan detectar y corregir errores rápidamente.

- **Pruebas Automatizadas:** Son fundamentales en el proceso de refactorización. Tener una suite de pruebas confiables permite a los desarrolladores asegurar que los cambios realizados no afectan el comportamiento esperado del sistema. Las pruebas actúan como una red de seguridad, proporcionando confianza para realizar modificaciones más profundas en el código.

- **Refactorización como Práctica Continua:** En lugar de esperar a que el código sea difícil de manejar, la refactorización debe considerarse una práctica

continua durante el desarrollo. Así se previene la acumulación de deuda técnica, facilitando el mantenimiento y la escalabilidad del sistema.

III. ¿Que refactorizar?

La refactorización en Ruby debe enfocarse en simplificar y mejorar las áreas de código que han crecido desorganizadas o difíciles de entender, a menudo llamadas "code smells" o "malos olores del código". Esto incluye problemas comunes como métodos largos, clases que hacen demasiadas cosas, nombres de métodos poco claros, y duplicación de lógica. El libro recomienda técnicas como:

- **Extracción de Métodos:** Dividir métodos grandes en varios métodos más pequeños con responsabilidades específicas para mejorar la claridad y reutilización.
- **Extracción de Clases y Módulos:** Reorganizar el código dividiéndolo en clases o módulos para evitar responsabilidades múltiples y reducir la complejidad.
- **Sustitución de Condicionales por Polimorfismo:** Usar polimorfismo en lugar de estructuras condicionales complejas para aumentar la extensibilidad y simplificar la lógica.
- **Simplificación de Expresiones Booleanas y Lógicas:** Refactorizar expresiones complejas para mejorar la legibilidad.
- **Eliminación de Código Duplicado:** Consolidar lógica repetitiva en métodos o clases únicas, facilitando el mantenimiento.

Estas técnicas no solo mejoran la estructura del código, sino que también permiten que el sistema sea más fácil de probar, depurar y adaptar a futuros cambios, asegurando una base de código más sostenible y manejable.

IV. Cambios realizados

1) Métodos largos.

En primer lugar, se analizó el código del programa para reconocer los métodos que eran muy largos, para así poder extraer parte de ese método, construir un procedimiento y utilizarlo dentro del método correspondiente, lo cual hace más

legible y mantenible el mismo. Uno de los casos en donde fue posible reconocer un endpoint muy largo fue el `post '/register' do`.

```ruby
post '/register' do
  new_username = params[:username]
  new_password = params[:password]
  new_password_repeat = params[:password_rep]

  if new_password != new_password_repeat
    # passwords don't match
    @error = "Passwords don't match."
    @passwordDist = true
    erb :register
  else
    aut = User.find_by(username: new_username)
    @passwordDist = false
    if aut
      # username already taken
      puts 'Username already taken.'
      @userOc = true
      erb :register
    else
      user = User.new
      user.username = new_username
      user.password = new_password
      user.score = 0
      user.score_time_trial = 0
      user.see_correct = false
      user.is_admin = false
      user.is_admin = true if %w[maxi mateo bachi].include?(new_username)
      user.save
      redirect '/login'
    end
  end
end
```

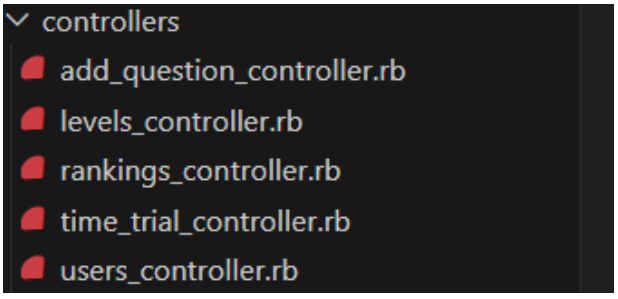
Por lo tanto, para poder resolver este “problema” creamos un componente `create_user` el cual, como su nombre indica, se utiliza para crear el usuario, pasándole como parámetro el nombre y la contraseña que el usuario desea tener. Luego, dentro de esta función se realiza la asignación de todos los atributos de user, como score, username, password, etc.

```ruby
post '/register' do
    new_username = params[:username]
    new_password = params[:password]
    new_password_repeat = params[:password_rep]
    if new_password == new_password_repeat
      aut = User.find_by(username: new_username)
      @password_dist = false
      if aut
        @user_oc = true
        erb :register
      else
        user = User.new
        create_user(user, new_username, new_password)
        user.save
        redirect '/login'
      end
    else
      @error = "Passwords don't match."
      @password_dist = true
      erb :register
    end
  end

  def create_user(user, new_username, new_password)
    user.username = new_username
    user.password = new_password
    user.score = 0
    user.score_time_trial = 0
    user.see_correct = false
    user.is_admin = false
    user.is_admin = true if %w[maxi mateo bachi].include?(new_username)
  end
```

2) Clases largas

La clase app.rb era muy extensa (344 líneas sobre 100 recomendadas), lo que hicimos fue crear controladores y migrar las funcionalidades del usuario, de los modos de juego, rankings, etc. a estos controladores, quedando de la siguiente manera la carpeta *controllers* y la correspondiente importacion dentro del archivo *app*:



```
require_relative 'controllers/users_controller'
require_relative 'controllers/levels_controller'
require_relative 'controllers/rankings_controller'
require_relative 'controllers/add_question_controller'
require_relative 'controllers/time_trial_controller'

set :database_file, '../config/database.yml'
set :public_folder, 'assets'
use UsersController
use LevelsController
use RankingsController
use AddQuestionController
use TimeTrialController
```

Finalmente la clase app quedó con muchas menos líneas y mucho más legible.

3) Convenciones

En cuanto a las convenciones para definir variables, métodos, etc, se utilizaba una mezcla de estilos en varios archivos: camelCase, snake_case y PascalCase, sin embargo, la convención que se utiliza en Ruby para poder definir los nombres es snake_case, por lo tanto hicimos los cambios correspondientes para cumplir con esto y tener un único estilo definido.

4) Comentarios

Además de los puntos anteriores, también se hicieron cambios en relación a los comentarios que tenía el código, debido a que, había varios comentarios innecesarios que no agregaban compresibilidad, ya que era bastante transparente el código o estaba bien claro lo que hacía la clase, el metodo, etc. Por lo tanto, se decidió remover estos comentarios.

V. Desafíos y Soluciones

1) Identificación de Code Smells Complejos:

- **Desafío**: Detectar y priorizar áreas problemáticas en el código puede ser complicado cuando los problemas no son inmediatamente visibles o cuando se trabaja con una base de código grande.

- **Solución**: Utilizamos RuboCop, una herramienta de análisis estático para Ruby, que ayuda a detectar code-smells y violaciones de estilo. RuboCop facilitó la identificación de problemas como métodos largos, duplicación de código y convenciones de estilo. Esto permitió priorizar los cambios más necesarios y seguir un enfoque sistemático en la refactorización.

2) **Mantenimiento de la Funcionalidad Original:**

   - **Desafío**: Asegurar que el código refactorizado mantiene el comportamiento exacto de la versión anterior puede ser complicado, especialmente cuando se realizan cambios estructurales importantes. Existe el riesgo de cometer errores sin darse cuenta.

   - **Solución**: Utilizar pruebas unitarias y de integración para verificar que cada cambio no altera el resultado esperado. Las pruebas automatizadas permiten comprobar rápidamente si el sistema sigue funcionando como antes después de cada paso de refactorización.

3) **Optimización de Código para Mejorar el Rendimiento sin Introducir Nuevos Problemas:**

   - **Desafío**: Al intentar mejorar el rendimiento durante la refactorización (por ejemplo, al simplificar estructuras de datos o métodos), es fácil introducir errores de lógica o afectar negativamente la precisión de los resultados.

   - **Solución**: Medir el rendimiento antes y después de los cambios para asegurarse de que el resultado es positivo, y validar los cambios con pruebas que aseguren que los datos procesados siguen siendo correctos. También puede ser útil abordar las optimizaciones después de la refactorización básica para reducir el riesgo.

4) **Adecuación a las Convenciones de Ruby:**

   - **Desafío**: Si el código original no seguía las mejores prácticas de Ruby (por ejemplo, la convención de nombres de métodos, uso de bloques, u otros estándares idiomáticos), puede ser desafiante adaptar todo el código a estas prácticas sin romper la funcionalidad.

   - **Solución**: Realizar la refactorización en pequeños pasos y aplicar linters o guías de estilo (como RuboCop) para identificar áreas

donde se pueden aplicar las mejores prácticas de Ruby, facilitando así la transición.