Universidad Nacional de Río Cuarto

Materia: Ingeniería de Software
Tema:

“Refactorización de Código del Proyecto Guesswhich”

Alumnos: Juarez Marcelo, Lingua Matías, Reynoso Juan Cruz
Fecha: 03/11/2024

1. Introducción

En la etapa de refactorización instalamos y utilizamos la gema *Rubocop*, una herramienta que nos permite analizar estáticamente el código en Ruby. El propósito principal es tener un código limpio, consistente y conforme a las convenciones de estilo del lenguaje.
Las funciones claves que tiene son:

1. Verificación de estilo.
2. Detección de errores.
3. Mejora del código.
4. Análisis de complejidad.
5. Configurabilidad.

Inicialmente, ejecutamos el comando **`rubocop,`** de esta manera, se obtuvo una primera impresión del estado del código de la aplicación, tras la inspección de todos los archivos del repositorio. El resultado fue el siguiente:

> "*14 files inspected, 544 offenses detected, 522 offenses autocorrectable*"

Luego, hicimos uso del comando ***`rubocop -a y rubocop -A.`*** Para forzar y tratar de corregir el máximo número de ofensas que se pueden corregir automáticamente. Ejecutar el comando de esta manera, realiza una autocorrección de los errores que admiten la misma. El resultado obtenido tras la ejecución de la autocorrección fue el siguiente:

> *"14 files inspected, 1549 offenses detected, 1453 offenses corrected, 23 more offenses can be corrected with `rubocop -A`"*

Algunos de los errores que se corrigieron automáticamente son los siguientes:

- Final newline missing.
- Gems should be sorted in an alphabetical order within their section of the Gemfile.
- Trailing blank lines detected.
- Missing frozen string literal comment.
- Add an empty line after magic comments.
- Prefer single-quoted strings when you don't need string interpolation or special symbols.
- Use underscores(_) as thousands separator and separate every 3 digits with them.
- Align the elements of an array literal if they span more than one line.
- Line is too longSpace found before comma.
- Align the keys of a hash literal if they span more than one line.
- Trailing whitespace detected.
- Spaces for indentation.
- Extra empty line detected at class body end.
- Inconsistent and incorrect indentation detected.
- Use the return of the conditional for variable assignment and comparison.
- Align else with if.
- Move redirect '/finish' out of the conditional.

- Favor modifier if usage when having a single-line body. Another good alternative is the usage of control flow &&.
- Put one space between the method name and the first argument.
- Space inside.
- Space missing after comma.

Seguidamente, ejecutando rubocop se obtiene:

"14 files inspected, 97 offenses detected, 23 offenses autocorrectable"

Y finalizamos la autocorrección ejecutando el comando *rubocop -A*. Se puede observar que de los errores no son posibles de autocorregir, las ofensas que persisten son:

- Carencia de documentación en migraciones y en clases.
- Métodos con límite de cantidad de líneas superado.
- Bloques con límite de cantidad de líneas superado.
- Líneas muy largas.
- Carencia de documentación en modelos.
- Nombres de variables.

Algunas de estas ofensas son:

- db/migrate/20240509171503_create_users_table.rb:3:1: C: Style/Documentation: Missing top-level documentation comment for class CreateUsersTable.
- db/migrate/20240509171503_create_users_table.rb:4:3: C: Metrics/MethodLength: Method has too many lines. [11/10]
- db/schema.rb:15:1: C: Metrics/BlockLength: Block has too many lines. [30/25]
- db/seeds.rb:21:121: C: Layout/LineLength: Line is too long. [157/120]
- server.rb:91:5: C: Naming/VariableName: Use snake_case for variable names.
  @secondChanceStreak = session[:secondChanceStreak]

2. Proceso de corrección

Iniciamos la etapa de corrección manual, corrigiendo las ofensas que no son autocorregibles. Los errores que se presentan son:

Errores encontrados en *app.rb*:

- **Metrics/ClassLength**: Class has too many lines
- **Naming/VariableName**: Use snake_case for variable names
- **Style/Documentation**: Missing top-level documentation comment for class App.
- **Metrics/BlockLength**: Block has too many lines. [35/25]

Errores encontrados en *migraciones*:

- **Style/Documentation**: Missing top-level documentation comment for class CreateUsersTable.
- **Metrics/MethodLength**: Method has too many lines. [11/10]
- **Style/Documentation**: Missing top-level documentation comment for class CreateAnswersTable.
- **Style/Documentation**: Missing top-level documentation comment for class CreateQuestionsTable.
- **Style/Documentatio**n: Missing top-level documentation comment for class CreateTablePasswordReset.

Errores encontrados en *server_spec.rb*:

- **Metrics/BlockLength**: Block has too many lines. [687/25]
- **Layout/LineLength:** Line is too long. [129/120]
- **Layout/LineLength:** Line is too long. [133/120]

Primero, se procedió con la corrección de los errores de **Naming/VariableName**, donde se renombraron algunas variables del código que se encontraban escritas en camelCase, por lo que fueron convertidas a snake_case. Por ejemplo, la variable de instancia `@extraTimeStreak` pasó a ser `@extra_time_streak`.

Luego, continuamos con errores de **Style/Documentation**, fue necesario agregar documentación a las clases App, User, Question, Answer y migraciones.

El error **Metrics/MethodLength** encontrado en una migración, en la tabla User, fue ignorado dado que el método **`def change`** se excedía por 1 línea. Por lo que se dió uso de la sentencia `rubocop:disable Metrics/MethodLength` para deshabilitar en este fragmento de código la métrica y luego volverla a activar con `rubocop:enable Metrics/MethodLength` ya que se consideramos que el método estaba correcto y no existía la necesidad de reducirlo o modularizarlo.

Un code smell que se identificó fue Large Class, más específicamente en la clase **App**. Esta, contenía todos los métodos relacionados con las rutas de la aplicación, tanto **GET** como **POST**. Esto genera un problema de **Metrics/ClassLength** debido a la gran cantidad de líneas de código acumuladas en una sola clase, lo cual complicaba la legibilidad, el mantenimiento y la escalabilidad de la aplicación.
Para resolver este problema, dividimos el contenido de la clase **App** en varios controladores, utilizando la técnica de refactorización Extraer Subclase, donde cada subclase (controlador) está enfocada en una responsabilidad específica de la aplicación. Este enfoque sigue el principio de responsabilidad única, organizando el código en controladores especializados. Los controladores que creamos son los siguientes:

1. **MainController**: Se encarga de manejar las rutas y vistas principales, como la página de inicio o rutas generales que no pertenecen a ninguna funcionalidad específica.

2. **GameController**: Agrupa todas las rutas relacionadas con la lógica del juego, como el inicio y la gestión de partidas. Aquí se manejan los métodos GET y POST que se relacionan específicamente con la experiencia de juego.
3. **UsersController**: Administra las rutas relacionadas con los usuarios, como la creación, actualización y eliminación de perfiles, así como la visualización de información de usuario.
4. **QuestionController**: Controla las rutas que gestionan las preguntas o cuestionarios. Este controlador es responsable de manejar las peticiones para crear, listar o responder a preguntas en el contexto de la aplicación.
5. **AuthController**: Centraliza la autenticación de usuarios, como el inicio y cierre de sesión, así como la gestión de permisos de acceso. Este controlador garantiza que las rutas relacionadas con la seguridad y autenticación están organizadas en un único lugar.

A su vez, se detectó el code smell Long Method. Es beneficioso para que el programa perdure en el tiempo que contenga métodos cortos. Como se trata de un método con muchas variables de instancia, se utiliza la técnica Extraer Método. Donde fue necesario reducir la cantidad de líneas en algunos bloques POST o GET para solucionar **Metrics/MethodLength**. Esto fue solucionado de forma tal que se modularizó el código dentro del bloque en funciones auxiliares que realicen dichas funcionalidades. Se crearon funciones como:

- `def handle_correct_answer(user)`
- `def handle_incorrect_answer(user)`
- `def increment_session_counters`
- `def increment_counter(counter)`
- `def increment_score`
- `def update_streak_flags`
- `def update_user_score(user)`

El código del POST previo a la modularización:

```
post '/questions' do
 if session[:username]
   user = User.find_by(username: session[:username])
   existing_answer = Answer.find_by(id: params["answer"])

   @question = Question.find_by(id: existing_answer.question)

   if existing_answer&.is_correct
     session[:question_count] ||= 0
     session[:question_count] += 1

     session[:user_score] ||= 0
     session[:user_score] += 10
     @score = session[:user_score]

     session[:count] ||= 0
     session[:count] += 1
     @count = session[:count]

     if @count == 3
       session[:extraTimeStreak] = true
     end
     if @count == 5
       session[:skipQuestionStreak] = true
```

```
      end
      if @count == 8
        session[:secondChanceStreak] = true
      end
      @extraTimeStreak = session[:extraTimeStreak]
      @skipQuestionStreak = session[:skipQuestionStreak]
      @secondChanceStreak = session[:secondChanceStreak]

      if user.score < session[:user_score]
        user.update(score: session[:user_score])
      end

      if session[:question_count] == 15
        session[:title] = "Congratulations, You won!"
        redirect '/finish'
      end

      @question.update(correct_answered: @question.correct_answered + 1)
      erb :'questions/game-stats'

    elsif !existing_answer.is_correct
      user.update(score: user.score - 5)
      @question.update(incorrect_answered: @question.incorrect_answered + 1)
      if user.score < 0
        user.update(score: 0)
      end
      @score = session[:user_score]
      session[:title] = "Your answer is not correct, you lost!"
      redirect '/finish'
    end
  end
end
```

Luego se obtuvo un código más claro y simple como el siguiente:

```
post '/questions' do
  if session[:username]
    user = User.find_by(username: session[:username])
    existing_answer = Answer.find_by(id: params['answer'])
    @question = Question.find_by(id: existing_answer.question)

    if existing_answer&.is_correct
      handle_correct_answer(user)
    else
      handle_incorrect_answer(user)
    end
  end
end
```

Finalmente, tras la corrección de los errores en las clases App, User, Questions, Answer y migraciones (Tablas), continuamos con errores en **server_spec.rb**. Donde, de la misma manera que la clase App, creamos archivos de test para cada controlador, de forma tal que se obtenga una modularización de los tests según el controlador que cubra y se logre obtener bloques con cantidad de líneas adecuadas.

El archivo **server_spec.rb** fue eliminado luego de haberlo separado en los siguientes:

- auth_controller_spec.rb.
- users_controller_spec.rb.
- game_controller_spec.rb.
- main_controller_spec.rb.
- question_controller_spec.rb.

3. Conclusión

Implementar herramientas de refactorización como RuboCop e identificar Code Smells estudiados en el libro Refactoring Ruby Edition, permiten transformar un código complejo y desorganizado en uno modular, organizado, legible, y escalable. En este proceso de refactorización, además de mejorar la calidad del código actual, permite obtener una base sólida para la evolución del proyecto a futuro, ya sea por la escalabilidad o el mantenimiento.