Análisis del Proceso de Refactorización

Ing. Software

Machuca Franco
Natali Valentino

Introducción.....3
MyApp.rb.....3
Test_spec.rb.....5
Questions_manager.rb y Questions_validator.rb.....7
Level_controller.rb.....16

Introducción

En este informe vamos a analizar la etapa de refactorización del proyecto. El objetivo de esta iteración es modificar el código para que tenga una mayor capacidad de expansión y una mejor legibilidad.

Con el fin de llegar al objetivo deseado, se empleó el uso de una gema recomendada por el equipo docente, llamada "**Rubocop**", que se utiliza para mostrar ofensas que ayuden a centrar los esfuerzos en partes mejorables del proyecto mediante convenciones utilizadas en ruby. También influimos nosotros mismos en el proceso de refactorización, realizando cambios que nos parecieron justificables para el código. A continuación mostraremos el resultado general de la inspección, y luego haremos énfasis en archivos específicos:

```
46 files inspected, 1016 offenses detected, 851 offenses autocorrectable
```

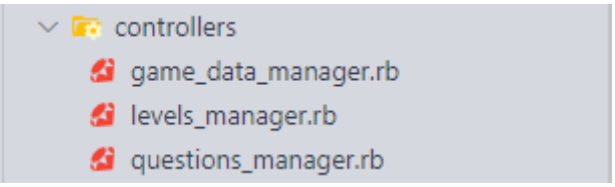
Antes de empezar, rubocop encontró 1016 ofensas. La mayoría de estas ofensas son relacionadas al estilo: StringLiterals (Modifica los strings para que tengan comillas simples), MethodCallWithoutArgsParentheses(Modifica las llamadas de métodos sin parámetros para que no empleen paréntesis), HookArgument(Cambia los before(:each) utilizados en el rspec por before), IndentationWidth, EmptyLineAfterExample, LineLength, WordArray.

851 de estas ofensas fueron autocorregidas por Rubocop. El resto fueron corregidas por nosotros. A continuación detallamos las correcciones realizadas por archivo:

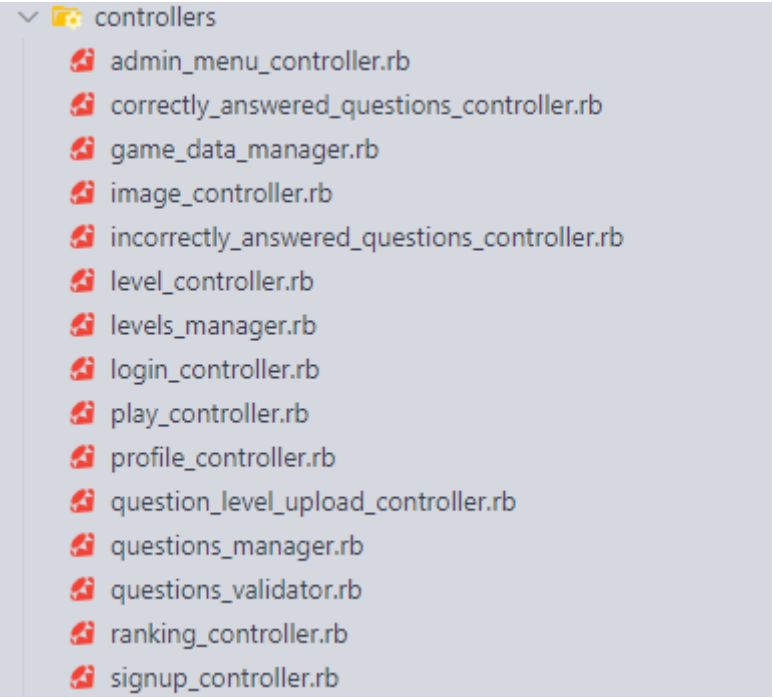
MyApp.rb

Este archivo anterior a la factorización sufría graves problemas de code smell, ya que tenía gran longitud y una carga de responsabilidades muy alta. Actualmente los endpoints de la clase están desperdigados en controladores que también manejan el acceso a las vistas del proyecto

Anteriormente, teníamos sólo 3 controladores que se encargaban de encapsular una lógica específica para un conjunto de endpoints que se encontraban en myapp, cuya longitud en líneas de código era de 336.



Luego de la refactorización, se redujo la clase myapp a 66 líneas de código usando el método Extract Class múltiples veces para construir controladores:



Test_spec.rb

Antes de la refactorización teníamos errores como este, donde utilizamos múltiples expects por test de unidad. Rubocop lo identificó como un error y realizamos una corrección:

```
24    it "redirects to the login page correctly when no other route is specified" do
25      get '/'
26      expect(last_response).to be_redirect
27      follow_redirect!
28      expect(last_request.path).to eq('/login')
29    end
```

Después:

```
26    it 'redirects to the login page correctly when no other route is specified' do
27      get '/'
28      expect(last_response).to be_redirect and (satisfy do
29        follow_redirect!
30        last_request.path == '/login'
31      end)
32    end
```

En esta imágen se puede apreciar que anteriormente se producía un error por múltiple declaración de let's y por nombre indexados. Rubocop lo marcó como una ofensa, así que buscamos algún de mejorar el código:

```
context "questionsManager" do

    let(:qm) { QuestionsManager.new }

    let(:level) { Level.create}

    let(:question1) { Question.new(questionable_type: "Multiple_choice",
level: level) }

    let(:question2) { Question.create(description: "prueba1",
questionable_type: "Translation", level: level) }

    let(:question3) { Question.create(description: "prueba2",
questionable_type: "To_complete", level: level) }

    let(:question4) { Question.create(description: "prueba3",
questionable_type: "MouseTranslation", level: level) }

    let(:question5) { Question.create(description: "prueba4",
questionable_type: "FallingObject", level: level) }



    before do

      level.questions.push(question2)
```

```
      level.questions.push(question3)
      level.questions.push(question4)
      level.questions.push(question5)
    end
```

Luego de la refactorización, solo quedaron las definiciones necesarias de lets para el testeo y se eliminó a su vez el problema de los nombres indexados:

```
context 'when questionsManager' do
    let(:qm) { QuestionsManager.new }
    let(:level) { Level.create }

    # Agrupamos las preguntas en un solo let y luego las inicializamos en before
    let(:questions) do
      [
        Question.new(questionable_type: 'MultipleChoice', level: level),
        Question.create(description: 'prueba1', questionable_type: 'Translation', level: level),
        Question.create(description: 'prueba2', questionable_type: 'ToComplete', level: level),
        Question.create(description: 'prueba3', questionable_type: 'MouseTranslation', level: level),
        Question.create(description: 'prueba4', questionable_type: 'FallingObject', level: level)
      ]
    end

    before do
      questions.each { |question| level.questions.push(question) }
    end
```

Questions_manager.rb y Questions_validator.rb

Antes de la refactorización, QuestionsManager se encargaba de la validación de la información que el usuario ingresaba cuando quería crear un nuevo nivel y/o pregunta (Nombre del nivel, descripción de la pregunta, tipo, respuesta/s, etc) a través del método validateParams(). Este método resultó ser muy extenso, ya que se necesitaban realizar distintas validaciones con parámetros específicos dependiendo del tipo de la pregunta (por ejemplo, en las preguntas de múltiple opción se necesita que alguna de las cuatro opciones ingresadas sea marcada como correcta, mientras que en las preguntas de traducción se necesita verificar que la respuesta esté en código morse, o que la pregunta lo contenga). Por lo tanto, decidimos usar Extract Method por cada caso de validación.

validateParams() antes de refactorización:

```
def validateParams(question_type: String, options: Array,
translation_type: String, key_word: String, key_word_morse: String,
correct_answer: String, question_description: String, level: Level)
    if level != nil && Level.exists?(level.id)
      case question_type
      when 'Multiple_choice'
        return (options.size <= 4 and
                options.all? {|elem| !elem[:text].to_s.empty?} and
                options.one? {|elem| elem[:correct]})

      when 'Translation'
        case translation_type
        when 'morse_translation'
          return (question_description.match?(/.*[.\-]+.*$/) &&
correct_answer.match?(/\A[a-zA-ZñÑáéíóúÁÉÍÓÚ0-9 ]+\z/))
        when 'spanish_translation'
          return (question_description.match?(/\A[a-zA-ZñÑáéíóúÁÉÍÓÚ0-9
]+\z/) && correct_answer.match?(/\A[.\- ]+\z/))
```

```
        else
          return false
        end

      when 'To_complete'
        return (key_word.match?(/[a-zA-ZñÑáéíóúÁÉÍÓÚ0-9]+/) and
key_word_morse.match?(/^[.-]+$/))

      when 'MouseTranslation', 'FallingObject'
        return correct_answer.match?(/^[.-]+$/)

      else
        return false
      end
    else
      return false
    end
  end
```

Como puede notarse, el código era algo complicado de entender principalmente por el switch case y la carga de responsabilidad que poseía el método en ese momento del código. Encontramos code smells en la cantidad de parámetros, que decidimos dejar tal y como estaba al no haber una solución clara; problemas de múltiples sentencias lógicas en una única línea;

```
return (options.size <= 4 and
                options.all? {|elem| !elem[:text].to_s.empty?} and
                options.one? {|elem| elem[:correct]})
```

Y también en el switch case, como se mencionó con anterioridad, porque agregaba una capa de complejidad al añadir nuevos tipos de preguntas.

Después:

```ruby
def validate_params(options:, correct_answer: String,
question_description: String,
                    level: Level,
                    question_type: String, translation_type:
                    String, key_word: String, key_word_morse: String)
    return true unless valid_level?(level)


    # Asocio un método validator a un tipo de pregunta.
    validators = {
      'MultipleChoice' => method(:validate_multiple_choice),
      'Translation' => method(:validate_translation),
      'ToComplete' => method(:validate_to_complete),
      'MouseTranslation' => method(:validate_morse_answer),
      'FallingObject' => method(:validate_morse_answer)
    }
    # Asocio parámetros específicos para cada validator según el tipo de
pregunta.
    required_params = {
      'MultipleChoice' => { options: options },
      'Translation' => { translation_type: translation_type,
correct_answer: correct_answer,
                         question_description: question_description },
      'ToComplete' => { key_word: key_word, key_word_morse:
key_word_morse },
      'MouseTranslation' => { correct_answer: correct_answer },
      'FallingObject' => { correct_answer: correct_answer }
    }
    # Elijo un validator según el tipo de pregunta recibido.
    validator = validators[question_type]
    return false unless validator
```

```ruby
    # Invoco al validator seleccionado con su respectiva lista de
parámetros.
    validator.call(**required_params[question_type])
  end

  def validate_translation(translation_type: String, correct_answer:
String, question_description: String)
    case translation_type
    when 'morse_translation'
      question_description.match?(/.*[.\-]+.*$/) &&
        correct_answer.match?(/\A[a-zA-ZñÑáéíóúÁÉÍÓÚ0-9 ]+\z/)
    when 'spanish_translation'
      question_description.match?(/\A[a-zA-ZñÑáéíóúÁÉÍÓÚ0-9 ]+\z/) &&
        correct_answer.match?(/\A[.\- ]+\z/)
    else
      false
    end
  end

  def validate_multiple_choice(options:)
    options.size <= 4 &&
      options.all? { |elem| !elem[:text].empty? } &&
      options.one? { |elem| elem[:correct] }
  end

  def validate_to_complete(key_word: String, key_word_morse: String)
    key_word.match?(/[a-zA-ZñÑáéíóúÁÉÍÓÚ0-9]+/) &&
      key_word_morse.match?(/^[.-]+$/)
```

```
  end


  def validate_morse_answer(correct_answer: String)
    correct_answer.match?(/^[.-]+$/)
  end


  def valid_level?(level)
    !level.nil? && Level.exists?(level.id)
  end
```

Al refactorizar utilizamos un validador que seleccionara un método según el tipo de pregunta para que la carga del método estuviera más distribuida y su única función pase a ser la de llamar los otros métodos según sea necesario.

Al finalizar, la clase QuestionsManager quedó muy extensa. Por esta razón, decidimos usar Extract Class para delegar esta responsabilidad a una nueva clase llamada QuestionsValidator.

También modificamos el método createNewQuestion():

createNewQuestion antes de refactorización:

```
def createNewQuestion(question_type: String, options: Array,
translation_type: String, key_word: String, key_word_morse: String,
correct_answer: String, question_description: String, level: Level)

    if validateParams(question_type: question_type, options: options,
translation_type: translation_type, key_word: key_word, key_word_morse:
key_word_morse, correct_answer: correct_answer, question_description:
question_description, level: level)

      case question_type

      when "Multiple_choice"

        @question = Question.create!(description: question_description,
level: level, questionable: Multiple_choice.create!())

        options.each do |op|

          Answer.create!(answer: op[:text], correct: op[:correct],
question: @question)

        end

      when "Translation"
```

```
        @question = Question.create!(description: question_description,
level: level, questionable: Translation.create!())

        Answer.create!(answer: correct_answer, correct: true, question:
@question)

      when "To_complete"

        @question = Question.create!(description: question_description,
level: level, questionable: To_complete.create!(keyword: key_word,
toCompleteMorse: key_word_morse))

        Answer.create!(answer: correct_answer, correct: true, question:
@question)

      when "MouseTranslation"

        @question = Question.create!(description: question_description,
level: level, questionable: MouseTranslation.create!())

        Answer.create!(answer: correct_answer, correct: true, question:
@question)

      when "FallingObject"

        @question = Question.create!(description: question_description,
level: level, questionable: FallingObject.create!())

        Answer.create!(answer: correct_answer, correct: true, question:
@question)

      end

      return true

    else

      return false

    end

  end
```

Como se puede apreciar, padecía de errores similares al método anterior, ya que tiene un switch case que entorpece la ampliación de los tipos de preguntas y, además, también posee una sobrecarga de responsabilidades, ya que se encarga de crear las preguntas y también sus respuestas.

Después:

```ruby
def create_new_question(options:, question_type: String,
translation_type: String, key_word: String,
                        key_word_morse: String, correct_answer: String,
                        question_description: String, level: Level)
    return false unless @qv.validate_params(
      question_type: question_type, options: options, translation_type:
translation_type,
      key_word: key_word, key_word_morse: key_word_morse, correct_answer:
correct_answer,
      question_description: question_description, level: level
    )

    question_class_mapping = {
      'MultipleChoice' => MultipleChoice,
      'Translation' => Translation,
      'ToComplete' => ToComplete,
      'MouseTranslation' => MouseTranslation,
      'FallingObject' => FallingObject
    }

    question_class = question_class_mapping[question_type]
    return false unless question_class

    @question = create_question_record(question_class,
                                       question_type: question_type,
question_description: question_description,
                                       level: level, key_word: key_word,
key_word_morse: key_word_morse)
    create_answers(options: options, question: @question, question_type:
question_type, correct_answer: correct_answer)
```

```
    true
  end


  private


  def create_question_record(question_class, question_type: String,
question_description: String, level: Level,
                             key_word: String, key_word_morse: String)
    questionable_instance = if question_type == 'ToComplete'
                              question_class.create!(keyword: key_word,
toCompleteMorse: key_word_morse)
                            else
                              question_class.create!
                            end


    Question.create!(description: question_description, level: level,
questionable: questionable_instance)
  end


  def create_answers(options:, question: Question, question_type: String,
correct_answer: String)
    case question_type
    when 'MultipleChoice'
      options.each do |op|
        Answer.create!(answer: op[:text], correct: op[:correct],
question: question)
      end
    else
      Answer.create!(answer: correct_answer, correct: true, question:
question)
```

```
    end

  end
```

Para la refactorización colocamos los tipos de preguntas dentro de un hashmap para que resulte más simple su ampliación futura. También se redujo drásticamente las responsabilidades del método, moviendo alguna de ellas a otros métodos.

Level_controller.rb

En este controlador nos interesa resaltar uno de los endpoints que teníamos anteriormente. Este endpoint se encarga de analizar si una pregunta fue contestada correctamente y también lleva al usuario a la siguiente pregunta, si es que hay una, dentro del nivel. Claramente, siguiendo la descripción que acabamos de proporcionar, ya se puede notar una sobrecarga en la responsabilidad del endpoint. Fuera de eso, otros code smells aparecen en forma de if's anidados que complejizan y ensucian el código; el manejo de los distintos tipos de preguntas que también tiene su impacto negativo por emplear un método distinto de análisis de las respuestas obtenidas.

```
post '/level/:level_id/:question_id/check' do
        if session[:user_id]
            @question = Question.find_by(id: params[:question_id])
            @level = Level.find_by(id: params[:level_id])
            @player = Player.find_by(id: session[:player_id])
            if @question && @level
                @answers = Answer.where(question_id: @question.id)
                if @question.questionable_type == "Translation" ||
@question.questionable_type == "To_complete" ||
                   @question.questionable_type == "MouseTranslation" ||
@question.questionable_type == "FallingObject"
                    @player_answer = @qm.buildPlayerAnswer(answer:
params[:user_guess], question: @question)
                else
                    @player_answer = Answer.find_by(id:
params[:answer_id])
                end
                if @qm.correctAnswer?(answer: @player_answer, question:
@question)
                    session[:user_level_score] += 100
                    @question.update(times_answered_correctly:
(@question.times_answered_correctly + 1))
                else
                    @question.update(times_answered_incorrectly:
(@question.times_answered_incorrectly + 1))
```

```
                end


                @next_question = @qm.nextQuestion(question: @question)
                if @next_question
                    redirect "/level/#{params[:level_id]}/" +
@next_question.id.to_s
                else
                    @player = Player.find_by(id: session[:player_id])
                    @gm.addPlayerLevelScore(player: @player, level:
@level, value: session[:user_level_score])
                    @final_score = session[:user_level_score]
                    session[:user_level_score] = 0
                    if @gm.completedLevel?(level: @level, player:
@player)
                        @show_success_popup = true
                    else
                        @show_failure_popup = true
                    end
                    @gm.unlockNextLevelFor(player: @player,
possiblyCompleted: @level)
                    erb @qm.show(question: @question)
                end
            else
                redirect "/jugar"
            end
        else
            redirect "/login"
        end
    end
```

Luego de la refactorización, se separó la responsabilidad en distintos métodos y se

encapsularon algunas acciones en métodos para que el desarrollador que lee el código no tenga que entender cómo fué hecha una acción, sino qué es lo que realiza. Fuera de ello, el código pasó a disminuir la cantidad presente de ifs, bajando la cantidad de líneas de 42 a 22

```
post '/level/:level_id/:question_id/check' do
    redirect '/login' unless session[:user_id]

    @question = Question.find_by(id: params[:question_id])
    @level = Level.find_by(id: params[:level_id])
    @player = Player.find_by(id: session[:player_id])

    if @question && @level
      @answers = Answer.where(question_id: @question.id)
      @player_answer = build_player_answer(@question, params)
      process_answer(@player_answer, @question)

      @next_question = @qm.next_question(question: @question)
      if @next_question
        redirect_to_next_question(@next_question)
      else
        finalize_level
        erb @qm.show(question: @question)
      end
    else
      redirect '/jugar'
    end
  end

  private

  def build_player_answer(question, params)
    if %w[Translation ToComplete MouseTranslation
FallingObject].include?(question.questionable_type)
```

```
      @qm.build_player_answer(answer: params[:user_guess], question:
question)
    else
      Answer.find_by(id: params[:answer_id])
    end
  end


  def process_answer(player_answer, question)
    if @qm.correct_answer?(answer: player_answer, question: question)
      session[:user_level_score] += 100
      question.increment!(:times_answered_correctly)
    else
      question.increment!(:times_answered_incorrectly)
    end
  end


  def redirect_to_next_question(next_question)
    redirect "/level/#{params[:level_id]}/#{next_question.id}"
  end


  def finalize_level
    @gm.add_player_level_score(player: @player, level: @level, value:
session[:user_level_score])
    @final_score = session[:user_level_score]
    session[:user_level_score] = 0
    @show_success_popup = @gm.completed_level?(level: @level, player:
@player)
    @show_failure_popup = !@show_success_popup
    @gm.unlock_next_level_for(player: @player, possibly_completed:
@level)
  end
```