
Trabajo de Fin de Grado

# GitHub Education Shell: ghedsh

Carlos de Armas Hernández

La Laguna, 25 de junio de 2018

D. **Casiano Rodríguez León**, con N.I.F. 42.020.072-S profesor Catedrático de Universidad adscrito al Departamento de Lenguajes y Sistemas Informáticos de la Universidad de La Laguna, como tutor

**C E R T I F I C A**

Que la presente memoria titulada:

"*GitHub Education Shell: ghedsh.*"

ha sido realizada bajo su dirección por D. **Carlos de Armas Hernández**, con N.I.F. 54.062-352-R.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 25 de junio de 2018

# Agradecimientos

xxxxxxxxxxxxxxxxxxxx

xxxxxxxxxxxxxxxxxxxx

xxxxxxxxxxxxxxxxxxxx

# Licencia

## Resumen

*Este Trabajo de Fin de Grado tiene como objetivo mejorar la versión previa de la gema «ghedsh». Una interfaz de línea de comandos (en inglés, command-line interface, CLI) desarrollada para soportar las metodologías de GitHub Education y que facilita la asignación de tareas, así como la revisión cualitativa y cuantitativa de las mismas.*

*Para ello, en una primera etapa, se ha llevado a cabo una refactorización del código fuente. De esta manera, otros desarrolladores podrán sumarse al proyecto para crear código limpio y mantenible.*

*En cuanto a la segunda etapa, ésta ha consistido en añadir funcionalidades a la gema, dando prioridad a aquellas que ofrecen solución a las limitaciones que poseen otras herramientas similares.*



**Palabras clave:** Git, GitHub, Ruby, CLI, GitHub Education, Profesores, Evaluación.

**Abstract**

This End-of-Degree Project aims to improve the previous version of " ghedsh" gem, an easy to use command-line interface (CLI) that supports GitHub Education's methodologies, facilitating the creation of assignments and also their qualitative and quantitative review.

To make this happen, in a first stage, a refactoring of existing source code has been carried out. In this way, other developers can join the project to create clean and maintainable code.

Regarding the second stage, the main task was adding functionalities, giving priority to those that solve limitations of other similar tools.

***Keywords:*** *Git, GitHub, Ruby, CLI, GitHub Education, Teachers, Grading.*

# Índice general

# Índice de Figuras

# Índice de Tablas

# Capítulo 1

# Introducción

## 1.1. Antecedentes

Git es un software de control de versiones de código abierto ampliamente utilizado, diseñado por Linus Torvalds. El propósito del control de versiones es llevar un registro de los diversos cambios que se realizan sobre los elementos de un proyecto y, de esta manera, facilitar la coordinación del trabajo que varias personas realizan sobre el mismo.

Por otro lado, GitHub es una plataforma de desarrollo colaborativo empleada para alojar proyectos que utilizan el control de verisones Git. Esta plataforma ofrece numerosos servicios que dan soporte al trabajo colaborativo. Entre los más destacados tenemos: repositorios, documentación, gestión de incidencias, gestión de equipos, chats, notificaciones, alojamiento de páginas web y tableros para organizar de manera flexible y visual las tareas que componen un proyecto.

Asimismo, GitHub dispone de un programa denominado *GitHub Education* que ofrece una variedad de herramientas tanto para el profesorado como para estudiantes.

A los docentes se les ofrece la posibilidad de crear organizaciones para cada clase, mediante repositorios privados que se crean automáticamente para el conjunto de estudiantes y equipos, a partir de las asignaciones realizadas por los profesores de las asignaturas a través la herramienta *GitHub Classroom* .

En cuanto a los estudiantes, se les da acceso gratuito a herramientas de desarrollo, plataformas de alojamiento y nuevas características en GitHub gracias a *Student Developer Pack* .

En este contexto, multitud de docentes se plantean la necesidad del desarrollo de metodologías e implantación de herramientas que asistan al profesorado en la evaluación de las tareas y actividades de programación, principalmente.

## 1.2. Estado actual del arte

Con el fin de integrar el uso del control de versiones en las aulas, GitHub desarrolló una herramienta de línea de comandos bajo el nombre de *Teachers Pet*. Ésta herramienta proponía que cada aula fuese una organización, lo que permitía a los profesores (que serían los propietarios de la organización) administrar todos los repositorios dentro de la misma. Con esta forma de trabajar se lograban dos objetivos:

- Facilitar a los profesores la distribución de código de inicio a los alumnos.
- Los profesores eran capaces de acceder a las tareas de los alumnos para resolver preguntas y comprobar el progreso.

Sin embargo, cayó en desuso y se dejó de desarrollar. La principal razón fue que los comandos se hacían excesivamente largos en determinadas tareas, puesto que era necesario especificar múltiples opciones.

Por otro lado, también existen diversas alternativas desarrolladas por la comunidad. Éstas son:

- *ghi* [3] (GitHub Issues): proporciona una manera fácil de gestionar las incidencias (*issues*) de los repositorios desde la terminal del usuario, utilizando su editor de preferencia. Es posible llevar a cabo diversas acciones como, por ejemplo, listar incidencias, abrirlas y cerrarlas y comentar en ellas, entre otras.
- *ghs* [4] (GitHub Search): es una utilidad que permite realizar búsquedas de repositorios alojados en GitHub. Admite diversas opciones para limitar las búsquedas en el ámbito deseado, como organizaciones y usuarios, por ejemplo.

Actualmente, GitHub dispone de una plataforma, *GitHub Classroom* [5], que mejora la experiencia de usuario respecto a las herramientas anteriormente nombradas, dado que dispone de una interfaz amigable en el navegador que simplifica la configuración de las aulas y asignaciones. No obstante, en esta plataforma existen algunas limitaciones:

- No dispone de alguna funcionalidad dentro de ella que permita al profesor crear un repositorio de evaluación para calificar las tareas.
- No da soporte a herramientas de integración continua, como *Travis CI*, *CircleCI*, *Jenkins*, etc.
- En muchos casos, el nombre de usuario que ha escogido el alumno al registrarse en GitHub no permite identificarlo. Además, el sistema para añadir información adicional del alumno es incómodo de usar, puesto que la información se inserta individualmente.

## 1.3. Objetivos y actividades a realizar

El principal objetivo de este Trabajo de Fin de Grado es desarrollar una interfaz de línea de comandos, bajo el nombre de *GitHub Education Shell*, que utilice la misma metodología que *Teachers Pet* y *GitHub Classroom* y que aporte una solución a las limitaciones que poseen dichas herramientas, así como priorizar la realización de tareas a gran escala más comunes en entornos educativos.

Para lograr dichos objetivos, se han definido diferentes actividades:

- **A1.** Estudio del funcionamiento de la API (*Application Programming Interface*) de GitHub y su implementación oficial en Ruby, *Octokit*.
- **A2.** Analizar aplicaciones similares e identificar aspectos a mejorar.
- **A3.** Comprensión del código de la última versión de la gema *ghedsh*.
- **A4.** Estudio de aspectos positivos y negativos del diseño de dicha gema.
- **A5.** Refactorización del código fuente, aplicando patrones de diseño.
- **A6.** Estudio de nuevas funcionalidades para incorporar.
- **A7.** Implementación de las funcionalidades escogidas.
- **A8.** Definir estructura básica de pruebas.
- **A9.** Documentación del código.

En la siguiente tabla se muestra el plan de trabajo con la duración de las actividades:

| **Objetivo** | **Fecha** |
|---|---|
| A1, A2, A3 | Febrero |
| A4, A5 | Marzo, Abril |
| A6, A7 | Mayo |
| A8, A9 | Junio |

Cuadro 1.1: Tabla resumen del plan de trabajo

## 1.4. Tecnología empleada

En cuanto a la tecnología empleada, el lenguaje de programación escogido para el desarrollo ha sido *Ruby*, puesto que la versión anterior también está escrita en este lenguaje.

Para que el programa utilice los datos de GitHub del usuario, se ha utilizado la API REST v3. En concreto, la librería oficial escrita en Ruby, [9], la cual proporciona una gran cantidad de métodos para realizar diferentes acciones, como, por ejemplo, crear repositorios y administrar su configuración, crear equipos y acceder a la información del usuario, entre otros. Para la autenticación del usuario, se generará un *token* automáticamente con los permisos requeridos para el uso del programa.

Por otro lado, en cuanto al ecosistema de *Ruby*, tenemos:

- *Rubygems* [13]: servicio de alojamiento de gemas para la comunidad de Ruby.
- *Bundler* [1]: proporciona un entorno para manejar las dependencias de un proyecto Ruby, añadiendo las versiones que son necesarias.
- *YARD* [15]: herramienta para la documentción del código fuente.
- *RSpec* [12]: librería para la definición de pruebas.

# Capítulo 2

# Desarrollo del proyecto

En este capítulo dos se va a describir el desarrollo del proyecto. Se dividirá en dos fases bien diferenciadas, una primera fase que consiste en el análisis y refactorización del código fuente de la versión anterior de *GitHub Education Shell*, y una segunda fase que trata de la incorporación de nuevas funcionalidades a la herramienta.

Por otro lado, también cabe nombrar la metodología de trabajo empleada. Situándonos en el marco de las metodologías ágiles, *Scrum* fue la metodología que mejor encajaba, teniendo en cuenta las características del proyecto. Se ha optado por un desarrollo incremental, en lugar de una planificación y ejecución estricta de las tareas. Además, en numerosas ocasiones, se produjeron solapamientos de las diferentes partes del desarrollo, en vez de un ciclo secuencial. También fueron frecuentes las reuniones con el tutor, en las que se comentaban tanto avances como dificultades.

## 2.1. Primera fase: análisis

En esta sección, se explicará detalladamente el proceso fundamental de la primera fase de este Trabajo de Fin de Grado.

El análisis del código fuente correspondiente a la primera versión de *ghedsh*, se ha llevado a cabo con la finalidad de identificar aquellas partes mejorables del diseño e implementación, puesto que una de las prioridades era facilitar el desarrollo colaborativo y, para ello, se requiere que el código sea limpio y fácil de entender, lo más auto-explicativo posible.

### 2.1.1. Estructura del repositorio

En la figura 3.1, se muestra la estructura del repositorio de la primera versión de *ghedsh*, en concreto, de la rama `master`.Dicha estructura corresponde con la forma estándar de organizar el código de las gemas.

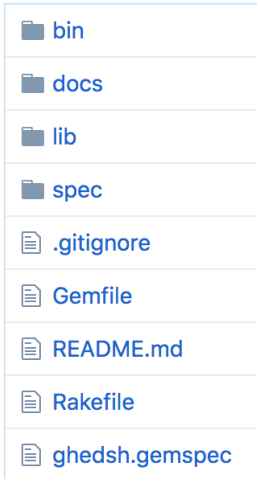


Figura 2.1: Estructura del repositorio (primera versión).

A continuación, se explicarán los componentes principales del repositorio:

- ***bin***: incluye el ejecutable de la gema, que se cargará en el `$PATH` del usuario.
- ***lib***: en este directorio se encuentra el código fuente de la gema.
- ***spec*** o ***test***: aquí se definirán las pruebas, que dependrán del *framework* de tests que utilice el desarrollador.
- ***Gemfile***: es un fichero donde se especifican las dependencias del programa.
- ***Rakefile***: se trata de un fichero muy común en las gemas. Su función es, principalmente, la automatización de tareas.
- ***gemspec***: en este último fichero se incluye toda la información acerca de la gema, como, por ejemplo, su versión, plataforma soportada, versión de Ruby requerida y nombre y correo electrónico del autor o autores.

### 2.1.2. Contenido del repositorio

El primer paso del análisis consistió en comprender exactamente el flujo del programa. Ésto es necesario dado que, para identificar las partes mejorables del diseño inicial, requiere entenderlo en profundidad.

En la figura 2.2, vemos el contenido de `/lib`:

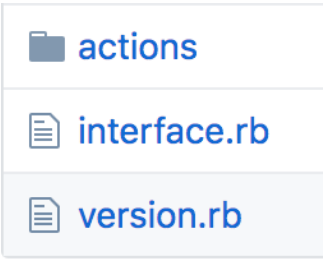


Figura 2.2: Contenido del directorio lib.

Los dos ficheros que ahí se encuentan son: `interface.rb` y `version.rb` .

En cuanto a `interface.rb`, implementa la clase *Interface*. Ésta clase lleva a cabo el bucle principal característico de los CLI, se trata del bucle *Lectura-Evaluación-Impresión* (en inglés, *REPL, Read-Eval-Print-Loop* [11]), que consiste en:

- **Lectura**: parsea la entrada del usuario y determina si existe esa acción en la estructura de datos interna.
- **Evaluación**: una vez que se acepta la entrada del usuario se ejecutan las acciones con los parámetros especificados, si son necesarios. Es aquí donde se realiza el manejo de errores y excepciones.
- **Impresión**: muestra al usuario el resultado obtenido tras realizar el paso de evaluación.

Por otro lado, el fichero `version.rb`, contiene una constante que indica la versión de la gema *ghedsh*. También cabe nombrar que, para asignar un identificador numérico a cada estado del software, se ha utilizado *Semantic Versioning* [14].

A grandes rasgos, este esquema de control de versiones tiene la estructura *MAJOR.MINOR.PATCH*, donde *MAJOR* se incrementa cuando se realizan cambios en la API que no son retrocompatibles. *MINOR* se modifica al añadir nuevas funcionalidades que sí son retrocompatibles y, por último, *PATCH* varía al corregir *bugs* en el software. Un ejemplo de versión sería:

- `ghedsh version 2.3.6`.

En cuanto al directorio `/lib/actions`, la figura 2.3 muestra el contenido del mismo:

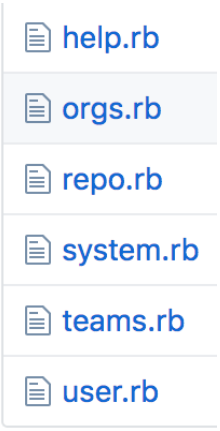


Figura 2.3: Contenido del directorio actions

Los ficheros principales se describirán según la funcionalidad que desempeñan:

- `orgs.rb`: proporciona los métodos necesarios relacionados con las organizaciones para comunicarse con la API de GitHub.
- `repo.rb`: agrupa métodos que llevan a cabo tareas relacionadas con los repositorios, nuevamente, haciendo uso de la API de GitHub.
- `teams.rb`: de manera similar a los anteriores ficheros, especializado en tareas de equipos.
- `user.rb`: agrupa métodos relacionados con el usuario.
- `system.rb`: este fichero se encarga de crear los directorios de configuración de *ghedsh*.

### 2.1.3. Code Smell

Una vez analizado el planteamiento inicial, se han detectado una serie de debilidades en el diseño que han dado lugar a diversos *code smell* [2]. Un *code smell* se define como cualquier característica del código fuente que, posiblemente, indica un problema más profundo. No son considerados como *bugs*, puesto que no impiden que un programa funcione de manera correcta.

No obstante, estos defectos de diseño pueden afectar al rendimiento del programa, aumentan la probabilidad de errores en el futuro e, incluso, ralentizar el desarrollo del programa y dificultar la extensibilidad del mismo.

Determinar qué es y lo que no es un *code smell* para un código fuente específico suele tener un componente de juicio subjetivo, dado que puede variar según

el lenguaje de programación utilizado, el desarrollador y la metodología de desarrollo aplicada. Pero, por otro lado, valorar los posibles casos de uso, aporta indicaciones para respetar ciertos principios y calidad del software.

En el apartado de refactorización se explicará cómo se ha procedido para solucionarlos. A continuación, se expondrán los *code smells* más significativos encontrados en la primera versión de *ghedsh*.

### *Switch Statements*

Suele ser muy característico de los códigos orientados a objetos. Esencialmente, el problema con las sentencias `switch-case` o sentencias `if` es la duplicación de código. Al utilizarlas con un conjunto de números o cadenas (normalmente, asignadas a constantes) que conforman una lista de valores permitidos para una entidad, existe una dependencia en ellas para decidir la entidad correcta a utilizar en cada momento.

Como consecuencia, al añadir nuevos casos, debemos localizar todas estas sentencias en el código y modificarlas. Para ilustrarlo con ejemplos, véase las figuras 2.4, 2.5 y 2.6.

```
USER=1
ORGS=2
USER_REPO=10
ORGS_REPO=3
TEAM=4
ASSIG=6
TEAM_REPO=5
```

Figura 2.4: Constantes numéricas que deciden la entidad a utilizar.

```
case
  when op == "exit" then ex=0
    @sysbh.save_memory(config_path,@config)
    s.save_cache(config_path,@config)
    s.remove_temp("#{ENV['HOME']}/.ghedsh/temp")
  when op.include?("help") && opcd[0]=="help" then self.help(opcd)
  when op == "orgs" then self.orgs()
  when op == "cd .."
    if @deep==ORGS then t.clean_groupsteams() end ##cleans groups cache
    self.cdback(false)
  when op.include?("cd assig") && opcd[0]=="cd" && opcd[1]=="assig" && opcd.size==3
    if @deep==ORGS
      self.cdassig(opcd[2])
    end
  when op == "people" then self.people()
  when op == "teams"
    if @deep==ORGS
      t.show_teams_bs(@client,@config)
    end
  when op == "commits" then self.commits()
  when op == "issues"
    if @deep==ORGS_REPO || @deep==USER_REPO || @deep==TEAM_REPO
      @issues_list=r.show_issues(@client,@config,@deep)
    end
  when op == "col" then self.collaborators()
  when op == "forks" then self.show_forks()
```

Figura 2.5: Ejemplo de switch smell, en interface.rb.

```
case
when scope==USER_REPO
  if config["Repo"].split("/").size == 1
    client.close_issue(config["User"]+"/"+config["Repo"],id)
  else
    client.close_issue(config["Repo"],id)
  end
when scope==ORGS_REPO || scope==TEAM_REPO
  client.close_issue(config["Org"]+"/"+config["Repo"],id)
end
```

Figura 2.6: Otro caso de switch smell, en repo.rb.

Este tipo de *smell* es el más repetido a lo largo del código de la primera versión de *ghedsh*, por lo que gran parte del proceso de refactorización se ha centrado en este aspecto.

#### *Long Method*

Se trata de un *smell* que se clasifica a nivel de métodos. Como su nombre indica (método largo), consiste en un método que ha crecido demasiado.

Escribir código suele ser más fácil que interpretarlo. Por ello, muchas veces este *smell* pasa desapercibido hasta que el método toma un tamaño considerable y dificulta saber qué es lo que realmente hace. Por esta razón, definir métodos más cortos con un nombre significativo, facilita la mantenibilidad del código y su lectura.

#### *Large Class*

*Large Class* se clasifica dentro de los *smells* a nivel de clases. Indica que una clase definida ha crecido excesivamente en tamaño (*God Object* [6]), es decir, realiza demasiadas cosas y que, seguramente, su funcionalidad puede descomponerse en clases más pequeñas y fáciles de manejar. Como suele ocurrir en los casos anteriormente nombrados, éste *smell* propicia también la duplicación de código.

## 2.2. Segunda fase: refactorización

En este apartado, se explicará el proceso de refactorización efectuado en el código fuente de *ghedsh*. El principal objetivo es el cumplimiento del principio *Open/Closed* [10], que pertenece a un grupo de cinco principios fundamentales (S.O.L.I.D [18]) de la programación orientada a objetos y cuya finalidad es hacer los diseños de software más comprensibles, flexibles y mantenibles.

### 2.2.1. Principios de diseño S.O.L.I.D

Se desarrollará ahora el significado de este acrónimo:

- ***S****ingle Responsibility*: establece que una clase debería de tener sólo una responsabilidad, es decir, si fuera necesario modificar dicha clase, debe ser por un único motivo.
- ***O****pen/Closed*: una entidad software debería estar abierta para extensión y cerrada para modificación. En el siguiente párrafo se explicará con más detalle este principio.
- ***L****iskov Substitution*: los objetos de un programa deberían ser sustituíbles por subtipos de ese objeto, sin afectar al correcto funcionamiento de ese programa.

- ***I**nterface Segregation*: indica que es preferible muchas interfaces cliente específicas, en lugar de una interfaz de propósito general.
- ***D**ependency Inversion*: sugiere que se debe depender de abstracciones, no de implementaciones. Esto da lugar a la técnica inyección de dependencias.

Como se ha indicado anteriormente, se procederá a detallar el principio *Open/Closed*, puesto que es bastante necesario desde el punto de vista del desarrollo de *ghedsh*. La principal razón es que, si otro desarrollador desea incluir nuevas funcionalidades, no necesitaría alterar el código original de la gema.

Este principio establece que, las entidades de software, ya sean módulos, funciones, clases, etcétera, deberían estar abiertas para la extensión de su comportamiento, pero cerradas para su modificación. Ésto se consigue, por ejemplo, mediante herencia, reescribiendo los métodos de la clase madre, pudiendo incluso ser abstracta dicha clase. Otra manera sería por medio de inyección de dependencias, que tienen la misma interfaz pero distinto funcionamiento.

### 2.2.2. Refactorización: Strategy pattern

En esta sección se expondrá el proceso que se ha seguido para eliminar el *switch smell*. El proceso ha consistido, fundamentalmente, en la aplicación del patrón estrategia (en inglés, *Strategy pattern* [19]).

*Strategy* es un patrón que permite a un programa seleccionar un algoritmo particular en tiempo de ejecución. El propósito de este patrón es proporcionar una manera clara de definir familias de algoritmos, encapsulando cada uno como objeto para poder intercambiarlos fácilmente. El principal beneficio de este modelo es que el objeto cliente puede elegir aquel algoritmo que le conviene y permutarlo dinámicamente según sus necesidades.

En cuanto a cómo se ha aplicado este patrón, primero ha sido necesario modificar el diseño inicial propuesto en la primera versión de *ghedsh*. De manera que, en el directorio `/lib/actions`, se han reescrito las clases que allí se encontraban y su propósito también ha cambiado, dado que ya no se agrupan métodos para repositorios, organizaciones, equipos, etcétera, sino que contendrán las acciones permitidas en cada uno de estos contextos. De este modo, aunque las acciones sean permitidas en varios contextos, cada clase conocerá su propia implementación.

A partir de la segunda versión, en `/lib/actions`, se encuentran las siguientes clases:

- `orgs.rb`: se define la clase `Organization`, contiene los métodos de las acciones permitidas para las organizaciones.

- `system.rb`: contiene la clase que se encarga de gestionar los directorios de configuración de *ghedsh*, así como guardar la configuración del usuario.
- `teams.rb`: se define la clase `Team`, que incluye los métodos de las acciones permitidas para los equipos.
- `user.rb`: incluye en la clase `User`, los métodos correspondientes a las acciones que se permiten en el contexto de usuario.

Por otro lado, en `/lib/interface.rb` (véase la figura 2.7) se tiene un claro ejemplo de patrón estrategia, que se aplica en el bucle Lectura-Evaluación-Impresión (REPL). Esto permite decidir en tiempo de ejecución el comando que el usuario quiere utilizar. Se parsea lo que ha introducido, de manera que el primer *token*, se refiere al nombre del comando deseado. Después, se busca dicho comando en la estructura de datos interna, en este caso, un *hash*, que consiste en una colección clave-valor en la que la clave es el nombre que identifica el comando y el valor es la función que exporta la clase `Commands`, la cual decide si es posible ejecutarlo, verificando si contexto actual responde a la acción solicitada por el usuario.

```ruby
loop do
  begin
    input = Readline.readline(@shell_enviroment.prompt, true)
    # handle ctrl-d (eof), equivalent to exit (saving configuration)
    if input.nil?
      @shell_enviroment.commands['exit'].call(nil)
      exit!(0)
    end
    input = input.strip.split
    command_name = input[0]
    input.shift
    command_params = input
    unless command_name.to_s.empty?
      if @shell_enviroment.commands.key?(command_name)
        result = @shell_enviroment.commands[command_name].call(command_params)
      else
        puts Rainbow("-ghedsh: #{command_name}: command not found").yellow
      end
    end
  rescue StandardError => e
    puts e
  end
  break if result == 0
end
```

Figura 2.7: Bucle Lectura-Evaluación-Impresión sin sentencia switch/case.

Cabe destacar que, la figura 2.5 anteriormente indicada, representa una porción del código, puesto que ahí se encontraban *hard-coded* [7] todos los comandos. Sin embargo, en la figura 2.7 se encuentra todo el código del REPL, tarea en la que se especializa la clase `Interface`.

Aparte, con este nuevo esquema se logra eliminar las constantes que deciden el tipo de clase a utilizar en cada momento (véase 2.4), ya que, en esta segunda versión, existe una variable que almacena el contexto y referencia directamente la clase con la que se realizan las acciones y ya no es necesario hacer la distinción entre, por ejemplo, repositorio de usuario, repositorio de organización y repositorio de equipo. Esto es así porque cada clase gestiona su contexto.

### 2.2.3. Refactorización: Extract Method

Como resultado de aplicar el patrón *Strategy* y la reescritura de las clases encargadas de las acciones, también se ha solucionado el *Long Method smell*. Ésto se ha logrado reduciendo el cuerpo de la función, a través de la extracción de partes del código que realizan tareas similares, dando lugar a nuevas funciones.

Para ilustrarlo mejor, en la siguiente porción de código vemos un ejemplo de *Extract Method* [16] muy simple:

Problema:

```
1    def print_owing(amount)
2      print_banner
3
4      # print details
5      puts "name: #{@name}"
6      puts "amount: #{amount}"
7    end
```

Solución (Extract Method):

```
1    def print_owing(amount)
2      print_banner
3      print_details(amount)
4    end
5
6    def print_details(amount)
7      puts "name: #{@name}"
8      puts "amount: #{amount}"
9    end
```

Si llevamos esta idea a un programa real, los beneficios son claros:

- Código más legible. Hay que destacar que, el nuevo método creado, debe llevar un nombre representativo, es decir, un nombre que describa su propósito.
- Menos duplicación de código. Muchas veces, el código contenido en una función puede ser reutilizado en otras partes del programa. Por lo tanto, reemplazamos el código repetido por llamadas al nuevo método.
- Mejor detección de errores. Al aislar partes de código, detectar dónde se ha producido un error es más fácil, hay son menos líneas que revisar.

### 2.2.4. Refactorización: Extract Class

El método de refactorización *Extract Class*[16] ayudará a mantener la adherencia al principio *Single Responsibility* (SRP), comentado en la sección 2.2.1.

A menudo, las clases comienzan siendo claras, fáciles de entender y sólo tienen una única responsabilidad. Sin embargo, en algún punto, la clase empieza a crecer: se realizan nuevas operaciones, se añaden más datos y se incorporan nuevos métodos. Por lo tanto, tarde o temprano esta clase acabe teniendo más de una responsabilidad.

A continuación, se explicarán una serie de pasos genéricos que guían a la hora de aplicar esta técnica:

- **Paso 1**. Determinar qué se va a extraer. Tratar de buscar unidades lógicas de datos que actúan sobre un subconjunto de los datos.
- **Paso 2**. Crear una nueva clase. Las operaciones extraídas dan lugar a una nueva clase, que tiene que establecer un enlace (puede ser bidireccional) con la antigua clase.
- **Paso 3**. Renombrar la antigua clase. Si, tras la extracción, el nombre de la antigua clase carece de sentido, debemos renombrarla.

*Extract Class* se aplica mejor si hay unidades de datos que se agrupan según el contenido que almacenan o si hay operaciones que sólo se realizan en un subconjunto de los datos. Esto último ocurría en el código de la primera versión de *ghedsh*, ya que la clase `Interface`, se ocupaba de varias tareas, y una de ellas era gestionar el contexto del CLI.

Una vez que se determinó qué se iba a extraer, fue posible crear una clase nueva, `ShellContext`, especializada únicamente en gestionar el entorno de *ghedsh*. Si éste se ha cargado correctamente, `ShellContext` lo comparte con `Interface`. Entonces, le permite saber cuáles son los comandos que están definidos, ejecutarlos y disponer de la configuración del usuario en cada momento.

Finalmente, el resultado es que la clase `Interface`, se encarga exclusivamente de realizar el REPL con los datos que le proporciona el entorno (`ShellContext`). Por lo tanto, si se eliminasen los comandos definidos, el REPL se llevaría a cabo correctamente, con la diferencia de que no se ejecutaría ninguna acción.

# Capítulo 3

# Resultados obtenidos

Este capítulo se centrará en explicar las características que incorpora *ghedsh* tras la etapa de desarrollo tratada en el capítulo anterior.

Se hará una distinción entre comandos del núcleo y comandos incorporados (*built-in commands*). Los comandos del núcleo, son aquellos que no trabajan con los datos de GitHub del usuario pero que, sin embargo, son esenciales desde el punto de vista la usabilidad y experiencia de usuario con el CLI.

Por otro lado, los comandos incorporados sí trabajan con los datos de GitHub del usuario identificado. Permiten realizar diversas tareas, priorizando la rapidez en la ejecución de las mismas y la facilidad de uso de la herramienta.

## 3.1. Autenticación con credenciales de GitHub

El contenido de esta sección pretende explicar el proceso de autenticación que debe seguir el usuario al usar *ghedsh* por primera vez.

Dicho proceso es necesario, puesto que se trabajan con los datos que dispone el usuario en GitHub. Además, la API REST v3 requiere, para ciertas consultas (en especial, modificaciones como crear repositorios, equipos y administrar la configuración), verificar la identidad del usuario. Si no fuera así, se podrían llevar a cabo comportamientos indeseados.

En *ghedsh*, se realiza la autenticación con *OAuth access token*[8], que consiste, en una definición muy simplificada, en una cadena de caracteres alfanuméricos que actúa como una contraseña. No obstante, en este caso de uso es mucho más potente y segura. Las principales ventajas son:

- Es revocable, es decir, el *token* puede dejar de ser válido, eliminando el acceso para ese *token* en particular, sin que el usuario tenga que cambiar su contraseña en todos sus accesos.

- Sus permisos son configurables, esto es, un *token* puede ser válido sólo para ciertos recursos de una API. De esta manera, se conceden permisos de forma más controlada.

Para sintetizar este apartado, el usuario que utilice por primera vez *ghedsh*, debe verificar su identidad mediante sus credenciales (nombre de usuario y contraseña) de GitHub y se generará de forma automática un *token* de acceso con los permisos necesarios para usar la herramienta.

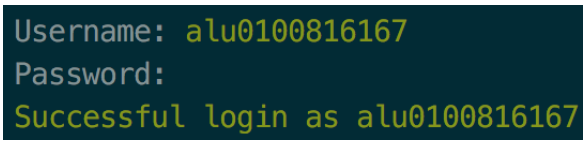


Figura 3.1: Ejemplo de autenticación al usar ghedsh por primera vez.

## 3.2. Comandos del núcleo de ghedsh

## 3.3. Comandos incorporados en ghedsh

## 3.4. Comandos que facilitan el proceso de evaluación

### 3.4.1. Automatizar la ejecución de scripts en los repositorios

### 3.4.2. Recopilar la información obtenida de la automatización de tareas

Una vez ejecutados los scripts necesarios para evaluar un determinado repositorio, es posible generar un GitBook con el resultado de la ejecución de los mismos. Este libro se genera en formato PDF y en HTML.

En función del contexto dónde nos encontremos dentro de la herramienta, podremos:

- Crear un GitBook en el repositorio en el que nos encontremos.
- Crear un GitBook en un determinado repositorio.
- Crear un GitBook en todos los repositorios que coincidan con una determinada expresión regular.

- Crear un GitBook en todos los repositorios de una asignación coincidan con una determinada expresión regular.

## 3.5. Funcionalidades extra

Además de las funcionales solicitadas en este Trabajo de Fin de Máster, se han añadido una serie de funcionalidades extra que, a pesar de no ser requeridas, brindan al usuario de una mejor experiencia de uso del programa:

- Autocompletado de los comandos disponibles en función del contexto donde nos encontremos (nivel principal, organización o repositorio).
- Opción de ayuda que muestra la descripción de los comandos y cómo se utilizan. Esta ayuda varía dependiendo del contexto donde nos encontremos.
- Opción de visualizar el directorio de trabajo donde se ha ejecutado el programa. Útil para determinar rutas relativas de los scripts que se desean ejecutar.
- Opción para conocer el propietario de cada repositorio. En el caso de que el repositorio pertenezca a una organización, mostrará los contribuyentes de ese repositorio.

NOTA: se puede consultar toda la información referente a los comandos del programa en el Apéndice 2.

## 3.6. Problemas encontrados y soluciones

A continuación se detallan los problemas encontrados durante la implementación de la herramienta y las soluciones encontradas para los mismos:

### 3.6.1. Asincronía

Una de las características más importantes del lenguaje JavaScript es la asincronía. Usa un modelo de operaciones de entrada/salida sin bloqueo y orientado a eventos, que lo hace ligero y eficiente. Sin embargo, algunas acciones que debía realizar esta herramienta debían de ser síncronas. Ejemplos son el login del usuario y ejecución de scripts.

**Solución**

La solución a este comportamiento pasó por realizar un amplio estudio de la documentación para usar mecanismos que permitieran bloquear la ejecución de la herramienta en las partes que deseábamos. Los mecanismos usados han sido:

- Funciones síncronas del propio lenguaje.

- Promesas
- Métodos async/await
- Librerías con métodos implementados de manera síncrona.

### 3.6.2. Autocompletado de comandos

Para el manejo de los flujos de lectura y escritura de la herramienta, se ha utilizado la interfaz nativa de Node.js (Readline). Esta interfaz provee de una función de autocompletado para el texto que escribe el usuario.

Sin embargo, sólo funciona con la primera palabra (comando) que escribe. Tras investigar al respecto y buscar posibles librerías alternativas, no existía ninguna solución que corrigiera este comportamiento.

**Solución**

Realizando numerosas pruebas, se halló una manera propia de conseguir completar más de un comando en la misma línea. Cuando realice los test de aceptación pertinentes requeridos por la comunidad de Node, solicitaré un Pull Request a su repositorio con esta mejora.

## 3.7. Perfil del usuario de ghshell

El uso de `ghshell` está especialmente dirigido a un determinado grupo de profesores: nos referimos al perfil de un profesor, principalmente docente en alguna rama de Ingeniería, con conocimientos avanzados en programación y en herramientas de control de versiones.

No obstante, ya que la curva de aprendizaje de `ghshell` no es excesiva y dado que el uso de las herramientas de control de versiones no se limita exclusivamente a repositorios de código fuente, se puede extender su uso para el resto de profesorado y usuarios con otros roles. Basta con tener claras unas nociones básicas de informática, junto con la lectura y asimilación previa de la documentación de la herramienta.

# Capítulo 4

# Conclusiones y líneas futuras

Desde hace unos años hasta ahora, ha tenido lugar un enorme crecimiento de las herramientas de control de versiones. Se han convertido en una herramienta imprescindible en la metodologías de desarrollo del software y las instituciones de enseñanza saben que incorporarlas a sus sistemas educativos es clave para ofrecer un servicio puntero y de calidad.

Ésto es lo que se pretende con la herramienta obtenida tras la realización de este Trabajo de Fin de Máster: que sea posible su implantación dentro del marco académico de la Universidad de La Laguna, partiendo de la premisa de que, actualmente, el desarrollo de un proyecto software sin tener detrás un sistema de control de versiones, no es viable.

La automatización de las tareas de clonado y ejecución de pruebas facilitaría al profesor, en primera instancia, la corrección de las prácticas y proyectos de los alumnos. El ahorro de tiempo de ejecutar estas tareas manualmente es considerable, teniendo en cuenta el número de prácticas que realiza cada alumno por asignatura. Esta enorme carga de trabajo del profesor puede ser aprovechada en otros ámbitos docentes.

Por otra parte, esta herramienta sienta las bases a posibles desarrollos futuros, ampliando las funcionalidades de la misma. Se ha desarrollado pensando en su posible escalabilidad y ya que cuenta con toda la estructura base creada (autentificación de usuarios, clonado, ejecución y reporte de resultados), se pueden añadir funcionalidades sin demasiado esfuerzo.

Para concluir, podemos afirmar que los objetivos marcados al comienzo de este Trabajo de Fin de Máster han sido cumplidos y las principales líneas de desarrollo a continuar podrían ser las enumeradas a continuación:

- Dotar de más funcionalidad de GitHub a la herramienta:
  - Subir cambios a los repositorios (git push).
  - Crear issues.
  - Gestionar Pull Requests.
  - Buscar repositorios.
  - Gestión de permisos de usuarios a repositorios y organizaciones.
  - Gestionar Classrooms.
- Enriquecer el formato de la documentación generada.
- Realizar despliegues locales de aplicaciones web (como procesos hijos de la herramienta).

# Capítulo 5

# Summary and Conclusions

This chapter is compulsory. The memory should include an extended summary and conclusions in english.

## 5.1. First Section

# Capítulo 6

# Presupuesto

En este capítulo se especifica un presupuesto que indica cuánto costaría realizar este Trabajo de Fin de Máster si se tratase de un trabajo encargado por un cliente.

## 6.1. Introducción y coste por hora

Se definirá una tabla con la lista de actividades realizadas en este Trabajo de Fin de Máster. Otra columna indicará la duración en horas que se han empleado para dicha actividad junto con el precio por hora calculado.

El precio por hora que se considerará en este presupuesto es de 30€/hora.

## 6.2. Funcionalidades requeridas

| **Actividad** | **Duración** | **Precio** |
|---|---|---|
| Autenticación con GitHub | xx horas | xx € |
| Listar organizaciones, asignaciones y repositorios | xx horas | xx € |
| Automatizar la descarga de repositorios | xx horas | xx € |
| Automatizar la ejecución de scripts en los repositorios | xx horas | xx € |
| Exportar la información obtenida de la automatización de tareas | xx horas | xx € |
| **Subtotal** | **xx horas** | **xx €** |

Cuadro 6.1: Tabla de actividades, duración y precios de las funcionalidades requeridas

## 6.3. Funcionalidades extra

| **Actividad** | **Duración** | **Precio** |
|---|---|---|
| Autocompletado de comandos según contexto | xx horas | xx € |
| Opción de ayuda según contexto | xx horas | xx € |
| Visualización del directorio de trabajo actual | xx horas | xx € |
| Opción para conocer propietarios del repositorio | xx horas | xx € |
| **Subtotal** | **xx horas** | **xx €** |

Cuadro 6.2: Tabla de actividades, duración y precios de las funcionalidades extra

## 6.4. Coste y duración total

| **Actividad** | **Duración** | **Precio** |
|---|---|---|
| Funcionalidades requeridas | xx horas | xx € |
| Funcionalidades extra | xx horas | xx € |
| **Total** | **xx horas** | **xx €** |

Cuadro 6.3: Precio y duración total

# Apéndice A

# Glosario

## A

**AJAX** : acrónimo de *Asynchronous JavaScript And XML* (JavaScript asíncrono y XML). Es una técnica de desarrollo web para crear aplicaciones interactivas o RIA (*Rich Internet Applications*). Estas aplicaciones se ejecutan en el cliente, es decir, en el navegador de los usuarios mientras se mantiene la comunicación asíncrona con el servidor en segundo plano. De esta forma es posible realizar cambios sobre las páginas sin necesidad de recargarlas, mejorando la interactividad, velocidad y usabilidad en las aplicaciones.

**API** : (*Application Programming Interface* o Interfaz de Programación de Aplicaciones). Conjunto de funciones y procedimientos o métodos que ofrece cierta librería para ser utilizados por otro software como una capa de abstracción.

**Asíncrono** :

**Asignación** :

**Async/Await** :

## C

**CVS** : (*Control Versioning System* o Sistema de Control de Versiones). Aplicación informática que implementa un sistema de control de versiones: mantiene el registro de todo el trabajo y los cambios en los ficheros (código fuente principalmente) que forman un proyecto y permite que distintos desarrolladores (potencialmente situados a gran distancia) colaboren.

# G

**GitBook** :

**GitHub** : forja para alojar proyectos utilizando el Sistema de Control de Versiones **Git**. Para más información, visitar `https://github.com`.

**GitHub Classroom** :

# H

**HTML5** : (*HyperText Markup Language*). Lenguaje de marcado para la elaboración de páginas web. Es un estándar que sirve de referencia para la elaboración de páginas web definiendo una estructura básica y un código para la definición del contenido de la misma.

# J

**JavaScript** : lenguaje de programación interpretado. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico. Se utiliza principalmente en su forma del lado del cliente (*client-side*), implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario y páginas web dinámicas.

# M

**Metodologias ágiles** : conjunto de métodos de ingeniería del software basados en el desarrollo iterativo e incremental, donde los requisitos y soluciones evolucionan mediante la colaboración de grupos auto organizados y multidisciplinarios. Se caracterizan además por la minimización de riesgos desarrollando software en iteraciones cortas de tiempo.

# N

**Node.js** :

**NPM** :

# O

**Organización** :

# P

**Promesa** :

# R

**Repositorio** :

# S

**Student Developer Pack** :

**Síncrono** :

# T

**Travis-CI** :

**TDD** : (*Test-Driven Development* o Desarrollo Dirigido por Pruebas). Práctica de programación que involucra otras dos prácticas: escribir las pruebas primero (*Test First Development*) y Refactorización de código (*Refactoring*).

**Token** :

# W

**Web semántica** : idea de añadir metadatos semánticos y ontológicos a la World Wide Web. Esas informaciones adicionales, que describen el contenido, el significado y la relación de los datos, se deben proporcionar de manera formal, para que sea posible evaluarlas automáticamente por máquinas de procesamiento. El objetivo es mejorar Internet ampliando la interoperabilidad entre los sistemas informáticos usando **agentes inteligentes**, es decir, programas en las computadoras que buscan información sin necesidad de interacción humana.

**World Wide Web** : (WWW). Sistema de distribución de documentos de hipertexto o hipermedios interconectados y accesibles vía Internet. Con un navegador web, un usuario visualiza sitios web compuestos de páginas web que pueden contener texto, imágenes, vídeos u otros contenidos multimedia, y navega a través de esas páginas usando hiperenlaces.

# Apéndice B

# Guía de uso

El objetivo de esta guía de usuario es proporcionar a los usuarios un ejemplo para la puesta a punto y ejecución de las funcionalidades implementadas en el paquete NPM ghshell durante el Trabajo de Fin de Máster.

## B.1. Instalación

### B.1.1. Requisitos

Node.js >= 8

### B.1.2. Dependencias

Gitbook Calibre

### B.1.3. Instalación

Para instalar el paquete, basta con ejecutar el siguiente comando:

```
[~]$ npm install ghshell -g
```

## B.2. Ejecución

```
1 var foo = function(){
2 console.log('foo');
3 }
4 foo();
```

### B.2.1. Otras consideraciones

Para que

# Bibliografía


[1] Bundler. https://bundler.io/.

[2] Code Smells. https://en.wikipedia.org/wiki/Code_smell.

[3] ghi. https://github.com/stephencelis/ghi.

[4] ghs. https://github.com/sonatard/ghs.

[5] GitHub Classroom. https://classroom.github.com/.

[6] God Object. https://en.wikipedia.org/wiki/God_object.

[7] Hard-coding. https://en.wikipedia.org/wiki/Hard_coding.

[8] OAuth 2.0: Bearer Token Usage. https://tools.ietf.org/html/rfc6750.

[9] Octokit. https://github.com/octokit/octokit.rb.

[10] Open/Closed Principle. https://en.wikipedia.org/wiki/Open%E2%80%93closed_principle.

[11] REPL, Read-Eval-Print-Loop. https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop.

[12] RSpec. http://rspec.info//.

[13] Rubygems. https://github.com/octokit/octokit.rb.

[14] Semantic Versioning. https://semver.org/.

[15] YARD. https://yardoc.org/.

[16] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[17] Martin Fowler. Refactoring: Improving the Design of Existing Code. chapter 6, pages 89–95. Addison-Wesley, 1999.

[18] Carlos Blé Jurado. *Diseño Ágil con TDD.* Lulu, 2010.

[19] Addy Osmani. Learning JavaScript Design Patterns. chapter 12, page 127. O'Reilly, 2012.