Universidad de La Laguna
Escuela Superior de Ingeniería y Tecnología
Sección de Ingeniería Informática

Trabajo de Fin de Grado

Lenguajes de Dominio Específico para el Modelado de Menús Dietéticos

Modeling Dietary Menus through Domain Specific Languages

Daniel Ramos Acosta

La Laguna, 1 de julio de 2017

Dra. Coromoto León Hernández, con N.I.F. 78.605.216-W profesora Titular de Universidad adscrita al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutora

Dr. Carlos Segura González, con N.I.F. 78.404.244-S profesor Investigador Asociado tipo C adscrito al Departamento de Ciencias de la Computación del Centro de Investigación Matemática (CIMAT) de México, como cotutor

C E R T I F I C A (N)

Que la presente memoria titulada:

"*Lenguajes de Dominio Específico para el Modelado de Menús Dietéticos*"

ha sido realizada bajo su dirección por D.Daniel Ramos Acosta con N.I.F. 51.148.467-D.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 1 de julio de 2017.

Agradecimientos

En primer lugar, agradecer a Coromoto por apoyarme durante toda la carrera siendo una gran profesora y tutora dándome siempre ánimos para seguir adelante.

Agradecer a mi mayor compañero de la carrera, Rafael Herrero por estar siempre ahí, recordarme mis despistes y por los momentos de risas con las investigaciones. Sin él no podría haber terminado esto.

A toda mi familia, en especial a mis padres, que con mucho cariño me han ayudado durante toda mi vida en mi formación como persona.

A Gabriel, por ser el mejor hermano del mundo. Espero poder ser un buen ejemplo y que algún día sigas mis pasos.

También agradecer a D. Casiano Rodríguez León por sus clases y por enseñarme mi lenguaje favorito, del que he podido quedarme trabajando.

Y a todos mis amigos, por haberme ayudado a llegar hasta aquí, por compartir su conocimiento conmigo y por sacarme de más de un aprieto.

Resumen

Hoy en día una de las principales causas de mortalidad son las enfermedades crónicas: diabetes, colesterol, etc. Estas enfermedades se deben en gran medida a malos hábitos alimenticios como dietas con alto contenido graso, o demasiados azúcares. La nutrición plantea un gran problema, ya que conseguir un menú dietético que proporcione de de forma óptima los nutrientes necesarios para el desarrollo de una vida sana es bastante complejo, no sólo por la cantidad de parámetros que se deben tener en cuenta, sino también por las necesidades personales de cada individuo, como personas celíacas o intolerantes a la lactosa.

El objetivo de este proyecto es realizar una contribución al problema de la planificación de menús saludables y equilibrados. Resolver el problema es prácticamente imposible con técnicas convencionales de investigación operativa para instancias de gran tamaño y con numerosos atributos. La implementación de un sistema informático para la generación personalizada de menús saludables necesita de una forma de representar digitalmente los menús. Por lo que, el objetivo de este Trabajo de Fin de Grado ha sido el análisis, diseño e implementación de un Lenguaje de Dominio Específico para la representación de menús dietéticos denominado "Dietary DSL". Este lenguaje se utilizará para implementar una interfaz de usuario que facilite la interpretación por usuarios no expertos de los resultados que arroje el sistema informático.

Entre las distintas metodologías de desarrollo de software se encuentran los denominados métodos ágiles. Existen muchos métodos de desarrollo ágil y la mayoría pone más énfasis en la adaptabilidad que en la previsibilidad. Al software desarrollado en una unidad de tiempo se le llama iteración, y su duración debe ser de una a cuatro semanas. Cada iteración del ciclo de vida incluye: planificación, análisis de requerimientos, diseño, codificación, revisión y documentación. El objetivo de cada iteración es tener una aplicación sin errores. El más destacado de los procesos ágiles de desarrollo de software es la Programación Extrema y es el que se ha seguido en el desarrollo del proyecto. En esta memoria se describen todas las herramientas utilizadas así como las pruebas llevadas a cabo para la verificación de "Dietary DSL".

Palabras clave: Ingeniería del software, Metodologías ágiles, Lenguajes informáticos, Meta-lenguajes

Abstract

Nowadays, one of the main causes of mortality are chronic diseases: diabetes, cholesterol, etc. These diseases are largely due to poor eating habits such as high-fat diets, or too many sugars. Nutrition raises a great problem, since obtaining a dietary menu that optimally supplies the nutrients necessary for the development of a healthy life is quite complex, not only by the amount of parameters that must be taken into account, but also by the personal needs of each individual, such as celiac or lactose intolerant ones.

The aim of this project is to make a contribution to the problem of planning healthy and balanced menus. Solving the problem is practically impossible with conventional operative research techniques for large instances with many attributes. Implementing a computer system for custom generation of healthy menus needs a way to digitally represent the menus. Therefore, the objective of this End-of-Grade Work has been the analysis, design and implementation of a Domain Specific Language for the representation of dietary menus called "Dietary DSL". This language is used to implement a user interface that facilitates the interpretation by non-expert users of the results that give the computer system.

Among the different methodologies of software development are the so-called agile methods. There are many agile development methods and most emphasize adaptation rather than predictability. The software developed in a unit of time it's called iteration, and its duration should be within one to four weeks. Each life cycle iteration includes: planning, requirements analysis, design, coding, review and documentation. The objective of each iteration is to have an application without errors. The most outstanding of the agile software development processes is Extreme Programming that is the one that has been followed in the development of the project. This report describes all the tools used as well as the tests carried out for the verification of "Dietary DSL".

Key words: Software engineering, agile methodologies, digital languages, Meta-languages

Índice general

Lista de figuras y tablas

1. Introducción

1.1. ¿Qué es una dieta equilibrada?

Según la Encuesta Europea de Salud en España de 2014 publicada por el Instituto Nacional de Estadística [3], existe una tendencia en el aumento de sobrepeso, y este problema no sólo se da en España, el ritmo de crecimiento de obesidad es alarmante, el cual se ha duplicado desde 1980 [10].

La obesidad supone un riesgo para la salud tanto física como mental, pudiendo provocar enfermedades como hipertensión, diabetes, problemas del hígado e incluso cáncer en los casos más extremos. Actualmente, éstas y algunas otras enfermedades crónicas son la principal causa de mortalidad del mundo, esto son el 63 % de las muertes, siendo responsables de 36 millones de muertes cada año [4]. Aun peor, en algunos países como Canadá la situación es más preocupante, donde el porcentaje de mortalidad alcanza el 90 % [12].

Estos datos sobre la obesidad y sobrepeso son muy alarmantes y deberían inducirnos a llevar un estilo de vida saludable, pero también debemos nombrar algunos de los beneficios que nos puede proporcionar una dieta equilibrada:

- Sistema inmunológico: Garantizar los nutrientes necesarios para el sistema inmunológico es muy importante, y que éste es dependiente del flujo sanguíneo por lo que necesitamos una buena función vascular. Carecer de ciertos nutrientes como pueden ser la Vitamina C, E y K tienen una repercusión negativa sobre el sistema inmunológico, ya que éstos nos ayudan a tener un buen flujo sanguíneo. Además otros nutrientes como la Vitamina C, E, hierro y ácidos grasos omega 3 nos permiten tener una buena producción de glóbulos blancos.

- Ayuda al sistema digestivo, ya que una dieta con una buena cantidad de fibra y baja en grasas facilita la digestión ya que no es tan alta la cantidad de enzimas para poder digerirlos.

- Reduce el padecimiento de trastornos mentales, como la ansiedad o la depresión. Determinados nutrientes como el fósforo nos ayudan a tener un cerebro rico y sano. Pero también desde un punto de vista social mejora nuestro autoestima y autoconcepto [2].

Pero, **¿Qué exactamente una dieta equilibrada?** Según la Sociedad Española de Dietética y Ciencias de la Alimentación [6]:

> *Una dieta equilibrada es aquella manera de alimentarse que aporta alimentos variados en cantidades adaptadas a nuestros requerimientos y condiciones personales.*

Es decir, en una dieta equilibrada se consumen los alimentos que demanda nuestro cuerpo y le suplen los minerales, vitaminas y nutrientes que necesita para poder tener un buen estado de salud.

1.2. Herramientas para la visualización de menús

Se ha llevado a cabo una revisión bibliográfica de herramientas que permitan visualizar menús dietéticos.

Se ha encontrado diversa información haciendo referencia al modelado de menús dietéticos, pero no desde una aproximación de un Lenguaje de Dominio Específico (en inglés DSL, Domain-Specific Language), sino basados en ontologías y en motores de reglas complejas, como puede ser el trabajo de *"FOODS: A Food-Oriented Ontology-Driven System"* [11].

El artículo de investigación más similar al objetivo de este proyecto que se ha encontrado es *"Genetic Fuzzy Markup Language for Diet Application"* [8], donde se habla de los lenguaje difusos para la elaboración de una aplicación dietética. Tiene sentido hablar de este tipo de lenguajes, ya que para especificar un menú dietético no existen límites determinados, y es por esto que es necesario aproximarse al problema de generación de dietas desde un punto de vista inexacto. Un ejemplo de

esta aproximación es el lenguaje *"Fuzzy Control Language"*, el cual es un lenguaje que implementa lógica difusa donde se pueden realizar expresiones inexactas como `IF (Car IS Inside)THEN (Door IS Down)`. El objetivo de este lenguaje son sistemas industriales donde los sensores tienen cierto margen de error.

Respecto a los DSL, se analizó el trabajo *"A Programming Environment for Visual Block-Based Domain-Specific Languages"* [5], donde se habla del uso de un lenguaje de dominio específico mediante el uso de la programación visual basada en bloques.

Por último, cabe destacar el trabajo *"Ontology-based multi-agents for intelligent healthcare applications"* [13]. En el se usa una ontología basada en múltiples agentes para desarrollar una aplicación de salud inteligente. En este trabajo también se hace uso de interfaces borrosas para representar el conocimiento, mediante el uso del *Fuzzy markup language*.

Existe más documentación sobre estos temas, pero no se encontró ninguna referencia al uso de lenguajes de dominio específico para la elaboración de menús dietéticos.

1.3. Objetivos del proyecto

El objetivo general del Trabajo de Fin de Grado (TFG) es diseñar un Lenguaje de Dominio Específico - DSL - para modelar menús dietéticos. Se han establecido tres objetivos específicos:

- **Lenguaje de Dominio Específico**: Por un lado, tenemos el lenguaje que hemos diseñado y desarrollado, que es un DSL interno del lenguaje de programación Ruby. Con él se pueden especificar de una forma similar al lenguaje humano recetas y menús dietéticos. El DSL nos sirve como interfaz para los dos siguientes módulos.

- **Visualización de los menús**: La tarea de este módulo es presentar de forma gráfica el significado del DSL, de forma que teniendo un menú descrito mediante el lenguaje de dominio específico, se puede obtener una tabla con las distintas comidas y sus valores nutricionales. Se han usado como modelo los menús de PIPO, el Programa de Intervención para la Obesidad Infantil [9].

- **Generado de menús**: Ambos módulos se integran con un generador automático de menús, que forma parte de un proyecto de mayor embergadura. Dicho sistema tendría la capacidad de generar menús equilibrados y saludables, por ejemplo mediante aprendizaje profundo o un motor de análisis complejo, de forma que genere recetas del DSL de Ruby.

1.4. Metodología

Cuando se desarrolló el anteproyecto, se especificaron los distintos pasos a cumplir el desarrollo del TFG. Han consistido en cumplir distintas tareas a lo largo del desarrollo del curso. Éstas son:

1.4.1. Tarea 0. Coordinación

Esta tarea se estableció al comienzo del TFG, y consistió en establecer las fechas de los seguimientos y reuniones que se iban a realizar durante su desarrollo. Éstas fechas no se llegaron a cumplir en su totalidad debido al consumo de tiempo de las prácticas externas, por lo que se tuvo que retrasar el cuarto seguimiento para finales de junio, y la defensa para el mes de julio.

1.4.2. Tarea 1. Revisión bibliográfica

Se realizó una búsqueda en la base de datos de bibliotecas digitales para seleccionar documentos que guardaban relación con el objetivo del TFG. Para ello se consultaron las bases de datos de *Scopus*, *Web of Science*, *IEEE Xplore* y *Google Scholar*. La literatura seleccionada se ha guardado en el repositorio principal.

1.4.3. Tarea 2. Diseño del prototipo de la herramienta

Se decidieron los detalles técnicos sobre la herramienta y su definición. Se barajó la posibilidad de crear un DSL externo en Python mediante el uso de la librería *"Pyparsing"*, pero al final se decidió usar Ruby para el DSL, ya que en la asignatura de Lenguajes y Paradigmas

de Programación se aprendió a desarrollar en este lenguaje, así como a hacer un DSL interno.

En esta tarea también decidimos usar la Base de Datos Española de Composición de Alimentos (BEDCA) sobre la de United States Department of Agriculture (USDA), ya que la primera estaba muy completa y además incluye los alimentos en español e inglés.

1.4.4. Tarea 3. Implantación de la herramienta

En esta tarea se realizó un análisis de las herramientas de desarrollo que existen en el mercado.

- Se planteó usar Trello o Github Projects para organizar y dirigir el proyectos, usando finalmente los Issues de Gitlab para anotar los problemas que surgían durante el desarrollo de la Gema, así como características nuevas que se añadirían posteriormente.

- LaTeX como lenguaje para realizar el desarrollo tanto de la memoria como de la presentación. Se propuso usar la herramienta Pandoc para realizar el desarrollo de la memoria de forma sencilla, ya que con esta herramienta se puede escribir el documento en formato Markdown ya que ésta nos permite convertirlo a LaTeX.

- Como complemento a LaTeX, se usará BibTeX para almacenar las referencias bibliográficas.

- Se decidió usar Git como sistema para el control de versiones, y Github para alojar el repositorio. Aunque se ha seguido usando el repositorio de Github, ahora se usa únicamente como espejo del que está alojado en Gitlab, donde se mantiene el código fuente (issues, wikis...). Estas características se deshabilitaron en Github para mantener este tipo de información en Gitlab.

1.4.5. Tarea 4. Validación y resultados computacionales

Se seguirá un desarrollo dirigido por pruebas usando distintas herramientas. Inicialmente se desconocía el lenguaje en el que iba a ser desarrollado el lenguaje, pero como finalmente ha sido Ruby se han

utilizado las herramientas que nos proporciona la comunidad de este lenguaje.

- Automatización de pruebas: Se ha usado *"Rspec"* para hacer un desarrollo dirigido por pruebas, siguiendo el flujo de trabajo que corresponde.

- Integración Continua: Finalmente se han usado los *"Runners"* de Gitlab para montar un sistema de integración continua, donde es posible obtener una revisión para cada confirmación (Commit) que hagamos, así como en las posibles peticiones de extracción (Pull Request) que puedan darse. Además, es una validación de que el código funciona correctamente.

- Cubrimiento de código: En un principio se planteó usar *"Coveralls"* para realizar el análisis de cubrimiento de las pruebas, pero Gitlab ofrece este servicio en forma de *"Runner"* así como alojar el informe en la página web del repositorio.

- Análisis estático de código: Usar una herramienta de análisis de código (también conocidos como *"linters"*), puede mejorar nuestro estilo de escribir un determinado programa. Especialmente en este caso, ha servido para indicar las distintas deficiencias en el diseño que podrían ralentizar el desarrollo o aumentar el riesgo de errores o fallos en el futuro (*Smells)* que tenía en el código así como características nuevas que se han incorporando al lenguaje que no se estaban usando. Para Ruby existe el demonimado *"Rubocop"*, una herramienta con múliples opciones capaz de analizar errores de anidamiento, complejidad ciclomática, características del lenguaje que están obsoletas, etc.

1.4.6. Tarea 5. Difusión de los resultados

Se elaborará la presentación del proyecto para presentarlo ante el tribunal y en el Congreso de Estudiantes de Informática. Así mismo, se tiene pensado participar con él en la Tenerife Lan Party (TLP). Además el código fuente se liberará bajo una licencia MIT y la memoria y presentación en Creative Commons Reconocimiento 4.0 Internacional haciendo público el código tanto en Github como en Gitlab una vez esté finalizado el trabajo.

2. Herramientas de Desarrollo

2.1. Ruby

Para implementar el diseño del DSL, se ha usado Ruby ya que este lenguaje posee una gran interfaz para el programador, en comparación a otros lenguajes más antiguos como JavaScript. En algunos aspectos, JavaScript es muy similar a Ruby en lo que se refiere a herramientas de desarrollo, ya que tiene una gran comunidad detrás, en parte gracias al framework web Ruby on Rails. A continuación enumeraremos algunas de las herramientas de Ruby que se han usado para este proyecto y veremos su paralelismo con JavaScript.

- `rvm`: El *Ruby Version Manager* es una aplicación escrita para Bash (aunque tiene su equivalentes para otras shells) que permite cambiar entre las distintas versiones de Ruby de forma sencilla. Es muy útil si se tienen varios proyectos de Ruby en distintas versiones del intérprete. Además, mediante el fichero de configuración *rvmrc* se puede hacer un cambio automático de la versión de Ruby cuando en nuestro directorio de trabajo exista este fichero. El equivalente de JavaScript sería `nvm`.

- `rake`: Es una implementación en forma de DSL del programa `make` de GNU. Permite definir distintas tareas para automatizar la ejecución de determinados comandos. La definición de las tareas se realizan en un fichero `rakefile.rb`. Por ejemplo, para poder compilar esta memoria se han de realizar varios pasos. Éstos están definidos en el fichero de configuración, de forma que al ejecutar el comando `rake build` se construye el fichero PDF de la memoria, y de forma similar el comando `rake build:watch`

vigila los ficheros Markdown y compila la memoria cada vez que haya cambios.

- `bundler`: *Bundler* es una Gema que nos permite administrar distintos aspectos de las bibliotecas que vayamos a desarrollar:

    - Permite crear la estructura inicial de una gema de Ruby. Se va pasando por una serie de preguntas para poder configurar correctamente el proyecto. Tiene distintas formas de configuración mediante los parámetros que se le pasan a este comando como: la gema que se va a usar para las pruebas, la licencia del proyecto o el código de conducta. Otro elemento útil de la estructura generada, es que crea una configuración inicial (*setup*) mediante Rake con muchas tareas útiles. También genera ficheros de configuración para la integración continua con Travis.
    - Administra las dependencias del proyecto mediante un fichero `.gemspec` que contiene toda la información de la biblioteca, como el nombre, descripción, etc... Una vez se ha clonado un proyecto de Ruby, con el comando `bundle install` esta gema se encargará de calcular las dependencias e instalarlas en el sistema.
    - Genera un fichero `Gemfile.lock`, cuyo propósito es bloquear la versión de las dependencias. De forma ideal, cuando se aumente el tercer dígito del versionado semántico (*Semantic Versioning*) sólo debería haber cambios para reparar errores. El problema es que cuando se trabaja en equipo, la versión de *bugfixing* puede variar de una máquina a otra, y esto puede generar errores a algunos miembros del equipo que están corregidos en otra versión. El fichero `Gemfile.lock` se asegura que las dependencias son exactamente las mismas en los distintos lugares donde tengamos nuestra aplicación Ruby. Se recomienda que este fichero se use cuando se están desarrollando aplicaciones, pero no para el desarrollo de gemas, ya que obliga al usuario a hacer uso de una versión determinada, por lo que el árbol de dependencias crecería demasiado.

```
$ bundle gem MiGema
Creating gem 'MiGema' ...
Do you want to license your code permissively under the MIT license?
This means that any other developer or company will be legally allowed
   to use your code for free as long as they admit you created it. You
   can read more about the MIT license at
   http://choosealicense.com/licenses/mit. y/(n): y
MIT License enabled in config
Do you want to include a code of conduct in gems you generate?
Codes of conduct can increase contributions to your project by contributors
   who prefer collaborative, safe spaces. You can read more about the code
   of conduct at contributor-covenant.org. Having a code of conduct means
   agreeing to the responsibility of enforcing it, so be sure that you are
   prepared to do that. Be sure that your email address is specified as a
   contact in the generated code of conduct so that people know who to contact
   in case of a violation. For suggestions about how to enforce codes of
   conduct, see http://bit.ly/coc-enforcement. y/(n): y
Code of conduct enabled in config
      create  MiGema/Gemfile
      create  MiGema/.gitignore
      create  MiGema/lib/MiGema.rb
      create  MiGema/lib/MiGema/version.rb
      create  MiGema/MiGema.gemspec
      create  MiGema/Rakefile
      create  MiGema/README.md
      create  MiGema/bin/console
      create  MiGema/bin/setup
      create  MiGema/.travis.yml
      create  MiGema/LICENSE.txt
      create  MiGema/CODE_OF_CONDUCT.md
Initializing git repo in /home/ubuntu/workspace/MiGema
```

Figura 2.1: Creación de una Gema con Bundler

Estas herramientas son casi estándar en el desarrollo Ruby, a continuación veremos las gemas que han facilitado el desarrollo del TFG.

2.1.1. Rubocop

"Rubocop" es una herramienta de análisis estático de código para Ruby, es decir un "*linter*". Sin especificar ninguna configuración, es capaz de analizar con alrededor de 400 reglas (llamadas *"cops"*), pero además es una herramienta extensible donde se pueden modificar reglas ya existentes o incluso crear las propias nuestras. De entre las características que puede analizar *Rubocop*, cabe destacar:

- **Distribución**: Es capaz de analizar el indentado e indicar donde se está usando de forma inconsistente. También detecta elementos que no están alineados cuando se encadenan llamadas a funciones.

- **Código con alta probabilidad de error**: Condicionales ambiguos, claves de hash duplicadas, o interpolar cadenas en vez de concatenarlas.
- **Métricas**: ABCMetrics, complejidad ciclomática, anidamiento, longitud de líneas, etc.
- **Rendimiento**: Puede detectar código donde cambiando algunos elementos simples se puede conseguir un mejor rendimiento, esto es reutilizando variables, funciones redundantes, etc.
- **Estilo**: Son reglas que no provocan errores, pero al existir varias formas de escribir el mismo código estas reglas obligan a elegir una para que el código quede consistente en toda la aplicación.
- **Reglas específicas de gemas**: También contiene reglas que son específicas de determinadas librerías como Rails o Bundler.

```
$ rubocop
Inspecting 8 files
......C.
MiGema/lib/MiGema.rb:1:1: C: The name of this source file (MiGema.rb) should
use snake_case.
require "MiGema/version"
^
MiGema/lib/MiGema.rb:1:9: C: Prefer single-quoted strings when you don't need
string interpolation or special symbols.
require "MiGema/version"
        ^^^^^^^^^^^^^^^^
MiGema/lib/MiGema.rb:3:1: C: Missing top-level module documentation comment.
module MiGema
^^^^^^
MiGema/lib/MiGema.rb:5:16: C: Do not put a space between a method name and the
opening parenthesis.
  def my_method (parameter)
                ^
MiGema/lib/MiGema.rb:6:5: C: Use a guard clause instead of wrapping the code
inside a conditional expression.
    if parameter > 10
    ^^
MiGema/lib/MiGema.rb:7:7: C: Redundant return detected.
      return 'foo'
      ^^^^^^
MiGema/lib/MiGema.rb:9:7: C: Convert if nested inside else to elsif.
      if parameter < 0
      ^^
MiGema/lib/MiGema.rb:9:7: C: Favor modifier if usage when having a single-line
body. Another good alternative is the usage of control flow &&/||.
      if parameter < 0
      ^^
MiGema/lib/MiGema.rb:10:9: C: Redundant return detected.
        return 'No mayor que 10'
        ^^^^^^

8 files inspected, 9 offenses detected
```

Figura 2.2: Ejemplo de ejecución de Rubocop donde detecta varios errores

Un *linter* es una herramienta indispensable para desarrollar una aplicación, especialmente cuando se trabaja en equipo o en proyectos de código abierto, ya que cada persona tiene su propia forma de escribir código y el linter se encargará de asegurar que el estilo sea consistente a lo largo de todo el programa.

2.1.2. Rspec

Rspec es un framework de pruebas para Ruby. Está orientado al Behavior Driven Development (BDD) con una interfaz para las pruebas que se asemeja a un diálogo.

```
describe DietaryDsl::Food do
  [ ... ]
  describe '.find_by' do
    context 'buscando por campos específicos' do
      context 'como el nombre en español' do
        subject { DietaryDsl::Food.find_by(nombre: 'Arroz') }
        it 'devuelve los datos correctamente' do
          expect(subject[:nombre]).to equal 'Arroz'
          expect(subject[:name]).to equal 'Rice'
        end
      end
      [ ... ]
    end
  end
end
```

Figura 2.3: Ejemplo de prueba de la gema `dietary_dsl`

En la figura 2.3 se muestra una prueba del Object-Relational Mapping (ORM), donde se espera que al buscar por un determinado campo en la base de datos nos devuelva el elemento correspondiente. Se pueden definir sujetos de prueba con los que testear nuestro código, darle un contexto a nuestras pruebas, etc. La interfaz de `expectations` tiene un gran número de métodos de forma que para casi todo lo que se quiera expresar se tiene un método o una combinación de los mismos que permiten expresarlo de forma idiomática. Incluso se puede renombrar los métodos de las pruebas para usarlo en otros idiomas y escribir "debe" en vez de "it" para describir algo como `debe 'devolver nil'do...`.

Rspec también incluye el denominado *"spec_helper"*, que es un fichero de configuración donde se indican todos los parámetros de las pruebas, así como la forma de enganchar librerías que amplíen su funcionalidad, como *mocks* y *stubs*.

2.1.3. VCR y WebMock

Estas dos librerías sirven para capturar los accesos a internet de la gema durante las pruebas, (conocido como *mocks* en inglés). Existen pruebas que verifican que se lanza el error correcto cuando existe un fallo. El problema que solucionan estas librerías es generar este fallo.

"*WebMock*" permite capturar todas las llamadas que vaya a hacer Ruby por Internet, y otorga la posibilidad de hacer respuestas personalizas. Incluso son programables para reaccionar de forma distinta ante determinados parámetros.

Por otro lado tenemos VCR (el nombre viene de *Video Cassette Recorder*). Esta librería usa *WebMock* como base. Cada vez que se acceda a Internet se pueden redirigir a lo que se denomina un "cassette", que es un fichero en formato YAML donde se tienen los datos con los que responder la petición. Se puede usar el mismo *"cassete"* para varios tests, e incluso parametrizarlo.

Pero lo más interesante de esta librería, especialmente para el ORM que se ha desarrollado, es la caché de las llamadas. Se ha configurado de tal forma que cada vez que capture una llamada a Internet, guarda su fichero YAML correspondiente, y la siguiente vez que se ejequte la misma prueba se usará la respuesta que estaba guardada. De esta forma, la segunda vez que se pasen las pruebas se tiene latencia cero respecto a Internet. La base de datos BEDCA para una llamada normal tiene una latencia de unos 500 milisegundos, que con la cantidad de pruebas que se tienen tarda casi diez minutos en ejecutar todas las pruebas. Usando esta biblioteca se reduce a un minuto y medio. Tarda tanto debido a la librería que se encarga de interpretar (*parsear*) las respuestas XML para traducirlas a objetos Ruby.

A las respuestas que se han guardado se les ha configurado un tiempo de expiración de siete días, por si cambia algo en la base de datos BEDCA poder modificar el ORM para que vuelva a funcionar.

2.1.4. Guard

Guard es una Gema que permite vigilar determinados ficheros, y reaccionar de la manera que se haya configurado. En el caso de `dietary_dsl` , se ha configurado de tal forma que cuando se cambie un fichero de código fuente, se ejecute el fichero de pruebas correspondiente, y si no

se encuentra se va subiendo a los directorios superiores hasta que se encuentre alguna prueba. Personalmente, creo que Guard y Rspec están demasiado entrelazados como para separarlos. Comparándolo con otros *tasks runners* como `brunch` o `gulp` se ve que estas herramientas están un poco obsoletas.

2.1.5. Cucumber

Al igual que Rspec, Cucumber es un frameworks para pruebas. Es aun más idiomático que Rspec, hasta tal punto que las mismas pruebas pueden servir como documentación. En Cucumber se pueden diferenciar dos aspectos, las características (*features*) y las definiciones de los pasos (*steps definitions*).

```
Feature: Food
  In order to obtain a visual representation of the DSL
  As a CLI

  Scenario: The DSL file exists and is valid
    Given a file named "diet.rb" with:
      """
      DietaryDSL::Food.new('Macarrones')
      [ ... ]
      """
    When I run `rubofood visualize diet.rb`
    Then the file "index.html" should contain:
      """
      <!doctype>
      [ ... ]
      """
```

Figura 2.4: La feature del visualizador de `dietary_dsl`

En las features se describen varios pasos que le dan un contexto a las pruebas:

- `Feature`: Este es el nombre de la característica que se está probando.
- `Scenario`: Es el supuesto donde se van a ejecutar las pruebas.

- `Given`: Define cuales son las entradas necesarias que necesita el programa para poder ejecutarse.
- `When`: El paso de ejecución, es lo que se debe realizar para obtener una salida.
- `Then`: Es el resultado que se espera obtener.

Pero, ¿Cómo se pueden ejecutar estas pruebas? Casando el texto con expresiones regulares mediante las definiciones de los pasos.

```
Given(/^a file named (?<filename>.+?) with:\s"""(?<content>.+?)"""$/) do
  # Leer "filename", comparar que el contenido del fichero es igual a "content"
end
```

Figura 2.5: Pasos para ejecutar el `Given` de la parte superior

Se debe crear un directorio donde se especifican las expresiones regulares con las que han de casar los textos de las *features*, y dentro del bloque definiremos el código que comprobará que dicha frase es cierta.

Con *Cucumber* se consiguen unas pruebas en un idioma muy natural, aunque está pensado para usarse con Rails y programas interactivos. En concreto para este proyecto, se usa para probar la interfaz de comandos.

2.1.6. Simplecov

Simplecov es una herramienta para obtener el cubrimiento que están realizando las pruebas. En un principio se intentó usar Coveralls, pero se descartó ya que era necesario tener el proyecto abierto al público. De forma alternativa, se ha conseguido desplegar la salida HTML de `simple-cov` en Gitlab mediante un Runner.

Prácticamente no tiene configuración inicial, y sólo se necesita requerirlo en el fichero `spec_helper`. La próxima vez que se ejecuten las pruebas se obtendrá un directorio llamado *"coverage"* donde estarán los ficheros HTML. Si se abre en el navegador, se podrá inspeccionar el porcentaje de cubrimiento que se ha realizado así como ver línea a línea qué no se ha cubierto.

2.2. Control de Versiones y Alojamiento del Código Fuente

2.2.1. Git

Se ha usado Git como control de versiones para administrar el código fuente del TFG, tanto de las dos gemas que se han creado como de la memoria. Se ha seguido el versionado semántico (*Semantic Versioning*) para el desarrollo de los tres proyectos, creando una etiqueta cada vez que se liberaba una nueva versión.

2.2.2. Github

Github es el más famoso de los alojamientos de repositorios Git, con más de 19 millones de repositorios y casi seis millones de usuarios activos es la comunidad más grande de Git. Se había planteado usar Github en un principio, pero finalmente se ha usado Gitlab ya que ofrecía ventajas significativas.

2.2.3. Gitlab

Gitlab es una plataforma similar a Github, pero además es un proyecto de código abierto.

Issues

Al igual que Github tenemos *"Issues"*. Aquí es donde se pueden crear "tickets" para indicar problemas con el código fuente del proyecto, como errores, ideas, o recomendaciones. Contiene todo lo que ofrece Github, formato Markdown, milestones, tags, etc. Pero además tiene otras funciones que no tiene el primero, como peso para los "Issues", el tiempo que ha consumido el issue e incluso un gráfico que indica la velocidad a la que vamos solucionando respecto al milestone.

Tablones de Issues

Wiki

En la wiki se puede añadir información respecto al proyecto en formato Markdown. Teniendo múltiples páginas en las que se puede añadir un manual o ejemplos de usos del proyecto.

Registro de Imágenes Docker

Cada repositorio de Gitlab permite tener un registro de imágenes Docker. Aquí se pueden subir las distintas imágenes que vayamos generando de nuestro proyecto para poder usarlas en producción. Además, estas imágenes se pueden generar desde los *Runner*, por lo que el proceso es automático.

Runners (CI y CD)

Los Runners de Gitlab son simplemente un flujo de imágenes de Docker que harán lo que le se haya indicado en el fichero de configuración `.gitlab-ci.yml`. Estos Runners se ejecutan cada vez subimos cambios al repositorio, de forma similar a como lo hace Travis.

En `dietary_dsl` se ha configurado el siguiente flujo de trabajo:

1. Se ejecutan dos tareas en paralelo:
    1. Se analiza el código con Rubocop
    2. Se ejecutan las pruebas de Rspec
2. Si las tareas anteriores son satisfactorias, se despliega el contenido de "coverage" en la web del repositorio y se da la confirmación de Git por válida.

Este flujo es sencillo para el desarrollo de una gema, pero en un proyecto web donde son varios los pasos que se deben realizar resulta mucho más útil, hasta el punto en el que se pueden construir imágenes de Docker y desplegarlas en el servidor que sea necesario, como OpenShift Origin o Amazon. De esta forma se puede configurar una buena Integración Continua y un Continuous Delivery.

Además, Gitlab proporciona dos insignias por defecto, una para el estado de la build del repositorio (*failed*, *running* y *success*) y una

que indica el porcentaje de cubrimiento del código. En caso de necesitar alguna insignia más, podemos programar nosotros mismos alguna tarea que genere un fichero SVG y colocar su enlace donde necesitásemos.

Gitlab Comunity Edition y Enterprise Edition

Como se nombró al principio, Gitlab no sólo es un alojamiento para repositorios, sino que además es un proyecto de código abierto. Se puede descargar y desplegar en un servidor personal, tanto dedicado como en contenedores Docker, aunque son necesarios varios contenedores ya que, además del propio Gitlab, necesitaremos dos bases de datos, una de Redis y otra de PostgreSQL. En caso de desplegarlo sin Docker, Gitlab ofrece el paquete "Omnibus", que es un ejecutable que realiza toda la configuración necesaria para poder ejecutarlo correctamente.

Se ofrecen tres modalidades de Gitlab: Gitlab Comunity Edition, la Enterprise Edition Starter y la Enterprise Edition Premium. Estas dos últimos tienen algunas características más, e incluyen soporte 24/7 y cursos de iniciación.

Gitlab Pages

Se puede definir un Runner que se encargue de desplegar ficheros web en un subdominio de Gitlab, en concreto para este proyecto se han desplegado los análisis de cubrimiento de las gemas. Además de ficheros estáticos, también se puede desplegar contenido dinámico basado en Jekyll y sus plugins.

2.2.4. RubyGems

RubyGems es el sitio web más común para distribuir gemas de Ruby. Se han publicado ambas gemas en este sitio, tanto `dietary_dsl` como `dietary_dsl_viewer`.

2.3. Memoria

Esta memoria se ha escrito usando el lenguajde de marcado *"Markdown"* y luego mediante la herramienta *Pandoc* se convierte a *LaTeX* para finalmente obtener el fichero PDF. Se usan los Runners de Gitlab

para compilar la memoria en la nube y obtener un enlace directo al PDF resultante.

2.3.1. LaTeX

LaTeX es un sistema de composición de textos orientado a la creación de documentos escritos que presenten una alta calidad tipográfica [7]. Con él se pueden escribir documentos en texto plano usando su lenguaje, y luego con alguna implementación de LaTeX compilarlo a un documento PDF. El problema (una de las razones por la que muchos no la usan) es su complejidad al querer conseguir algo, así como su sintaxis con llaves y barras invertidas que termina siendo algo difícil.

2.3.2. Markdown

```
     Runners (CI y CD)

Los Runners de Gitlab son simplemente un flujo de imágenes de Docker que harán
lo que le hayamos indicado en el fichero de configuración `.gitlab ci.yml`.
Estos Runners se ejecutan cada vez que hacemos un push al repositorio, de forma
similar a como lo hace Travis.

En `dietary_dsl` se ha configurado el siguiente flujo de trabajo:

1. Se ejecutan dos tareas en paralelo:
    1. Se analiza el código con Rubocop
    2. Se ejecutan las pruebas de rspec
2. Si las tareas anteriores son satisfactorias, se despliega el contenido de
"coverage" en la web del repositorio y se da la build por válida.
```

Figura 2.6: Fragmento de esta memoria escrita en Markdown

Markdown es un lenguaje de marcado ligero y simple con el que se puede escribir documentos de forma sencilla con apenas casi sin sintaxis. De hecho en texto plano es legible. Son varias las herramientas que se pueden usar para convertir este lenguaje a uno que podamos visualizar, de entre ellas se ha escogido Pandoc.

2.3.3. Pandoc

Pandoc es una herramienta escrita en Haskell para convertir de un lenguaje de marcado a otro. En este caso se usa para convertir los

ficheros Markdown a LaTeX, de forma que el contenido de los capítulos está escrito en Markdown, y se usa una plantilla de LaTeX para el resto de la memoria.

Además, si es necesario Pandoc nos permite embeber código LaTeX dentro del fichero Markdown, por si se necesitara hacer algo que no se pueda hacer únicamente con Markdown.

3. Diseño del Lenguaje de Dominio Específico

La gema que compone el DSL (`dietary_dsl`) está formada a su vez por otros DSL más pequeños que cumplen distintas funciones para construir un buen ecosistema que permita representar lo mejor posible menús dietéticos. Estos son: la interfaz con la base de datos (ORM), la representación de medidas, y el DSL de los alimentos.

3.1. Mapeo Objeto-Relacional

Cuando se planteó desarrollar un DSL para modelar menús dietéticos, uno de los primeros problemas que surgió fue el obtener los datos de los alimentos. Se decidió usar la base de datos BEDCA frente USDA ya que la primera contiene los nombres en español de los alimentos, además de inglés.

Se quería desarrollar una interfaz similar a la de ActiveRecord, el famoso ORM de Ruby on Rails de forma que a un desarrollador de Ruby le sea lo más familiar posible esta interfaz. Se han conseguido imitar los siguientes métodos:

- `.find`: Devuelve un elemento buscando por su ID.
- `.find_by`: Devuelve el primer elemento cuyos elementos sean **exactamente iguales** a los proporcionados por parámetro.
- `.find_by_like`: Devuelve el primer elemento cuyos elementos cumplan la expresión regular proporcionada por parámetro.

```ruby
# ActiveRecord:

# return the first user named David
david = User.find_by(name: 'David')

# dietary_dsl:
# devolver el primer alimento llamado Arroz
arroz = Food.find_by(nombre: 'Arroz')
```

Figura 3.1: Comparación de una query en ActiveRecord y `dietary_dsl`

El mayor inconveniente de haber usado BEDCA, es que no proporciona ninguna API ni documentación en su página web. Se ha tenido que hacer un trabajo de ingeniería inversa capturando las llamadas al servidor para ver si era posible crear un wrapper ya sea mediante una API o un scrapper.

Para capturar las peticiones HTTP que realiza la página web de BEDCA se ha usado la herramienta *Postman*. Con ella se pueden analizar todos los aspectos de una petición HTTP, como el cuerpo o la cabecera por ejemplo. Cuando se capturaron estas peticiones, resulta que el sitio web realizaba consultas a una API de XML escrita en PHP/5.2.4 con Ubuntu 5.10, por lo que esta aplicación debe datar alrededor de 2007. Se intentó averiguar si se hacía uso de un framework para desarrollar la API por si ya existía un Wrapper para Ruby, pero se llegó a la conclusión de que se había escrito a mano.

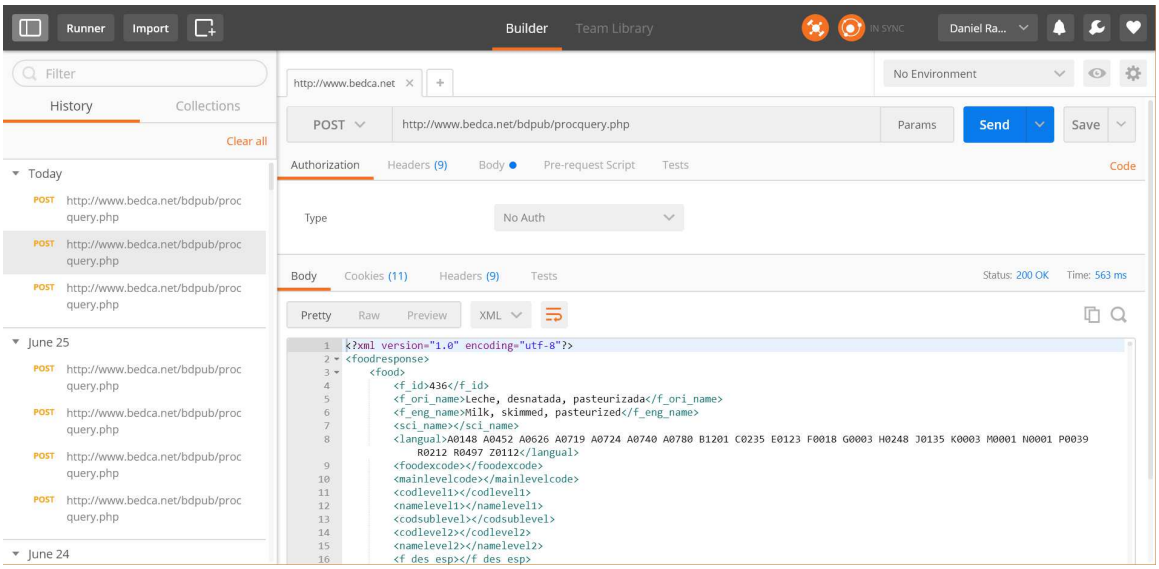


Figura 3.2: Llamada a la API de BEDCA

No existe ninguna documentación sobre esta API, por lo que se ha tenido que ir averiguando que significan los distintos campos que estaban disponibles además de los distintos tipos de llamadas.

Se realizó una búsqueda por si ya existía alguna interfaz con esta base de datos, pero no se encontró ninguna por lo que se tuvo que desarrollar este ORM para poder realizar peticiones a la base de datos.

3.1.1. `Food`

Esta clase es la interfaz principal del ORM. Además de los métodos para hacer consultas, tiene sobrecargado el operador de indexación (`[]`) para poder acceder a los valores del alimento, por ejemplo `manzana[:nombre]` devuelve "Manzana", y `manzana[:name]` "Apple", y así sucesivamente.

El comportamiento interno del ORM a la hora de ejecutar una consulta es el siguiente:

1. Se llama al método con el que se quiera realizar la consulta.
2. Se analizan los parámetros y se llama al método interno que corresponda.
3. Se construye el documento XML para realizar la petición usando la gema `builder`.
4. Se realiza una petición HTTP con el documento XML y con las cabeceras necesarias.

5. Se lee la respuesta de la petición, se comprueba que es válida y se interpreta el XML con la gema `activesupport`.
6. Se instancia la clase `Food` con los datos de la petición.

Cuando el constructor de la clase `Food` recibe los datos, convierte las claves en forma de *string* del Hash en símbolos para que sean únicos, y se agrupan los valores nutricionales por categoría creando instancias de la clase `FoodValues`.

Otro método útil es `kcal`, con el que se pueden calcular las kilocalorías totales del alimento (hay que calcularlo porque la base de datos proporciona este dato en kilojulios).

Además, se incluye el método `to_s` que exporta el alimento en formato Markdown. En el repositorio de `dietary_dsl`, bajo el directorio `spec/dietary_dsl/bedca_api/fixtures/pulpo_output.md` se puede encontrar un ejemplo de la salida.

3.1.2. `FoodValues`

Esta clase representa una categoría de valores nutricionales, y almacena los datos de los mismos. Incluye el módulo `Enumerable` para poder recorrer cada uno de los valores nutricionales, y además incluye un método llamado `to_table` que permite obtener una tabla en formato Markdown con sus valores. Al igual que la clase `Food`, se sobrecarga el operador de indexación para poder acceder a cada valor nutricional.

El constructor recibe un array de valores nutricionales, el cual se recorre para crear instancias de la clase `FoodValue` (en singular).

3.1.3. `FoodValue`

Aquí se almacenan los datos referentes a un valor nutricional específico, como su nombre, la unidad de medida, cantidad por cada 100 gramos, una descripción breve, y más atributos. Al igual que en las clases superiores se sobrecarga el operador para indexar, y se incluye el método `to_table`.

3.1.4. Resultado

Con estas tres clases obtenemos una buena interfaz con BEDCA para que pueda ser consumida por el Lenguaje de Dominio Específico.

```ruby
manzana = Food.find_by(nombre: 'Manzana')
# #<Food:0×000000025f59b8 ... >
kilocalorias = manzana.kcal
# ⇒ 50
fibra = manzana[:groups][:minerales][:hierro].to_s
# ⇒ 0.4mg
```

Figura 3.3: Ejemplo de obtención de datos mediante el ORM

3.2. Representación de Medidas

Otro de los problemas que nos hemos enfrentado, es a la especificación de medidas para los platos. Se realizó una búsqueda de una librería que permitiese especificar medidas como gramos y litros, y que además permitiese la conversión entre ellos. Se encontró únicamente la librería llamada Alchemist, la cual añadía nuevos métodos a la clase numérica primitiva de Ruby de tal forma que se pudiesen escribir de forma muy natural, como `2.miles.to.meters`. No se usó esta librería directamente porque se quería ampliar su funcionalidad para que la representación de algunas medidas de cocina (como cucharadas, o vasos), y además incluye conversiones de distancia, velocidades, y más que no es útil para el propósito de esta gema.

3.2.1. `Measure`

Esta es la clase de la que heredan las demás. Incluye un constructor que inicializa la unidad de medida principal de la clase, un método `.to` que devuelve la propia clase (es azúcar sintáctica), y un método `to_s` para poder imprimir las medidas.

3.2.2. `Masa`

Esta clase representa unidades de medidas de masa, como gramos o kilogramos, pero además se pueden representar usando medidas inexactas como tazas o rodajas. Se sobrecargan los operadores operacionales para poder representar medidas en forma de fracción o multiplicación.

Se abre la clase primitiva Fixnum para que estos métodos queden disponibles al instanciar un número.

3.2.3. `Volumen`

Representa unidades de medidas relacionadas con el volumen, como litros, mililitros o centímetros cúbicos, y al igual que la clase `Masa` soporta medidas inexactas como cucharadas, vasos o chorritos.

3.2.4. Resultado

Al abrir y expandir la clase Fixnum con estos nuevos métodos, obtenemos una interfaz similar a la que tiene Alchemist, con la que se pueden representar distintas unidades de medidas y realizar operaciones con ellas:

```
7.cucharadas.to.ml
# ⇒ 20
2.chorrito.to.ml
# ⇒ 125
3.vasos.to.ml
# ⇒ 600
4.rodajas.to.g
# ⇒ 80
1.kilogramo
# ⇒ #<DietaryDsl::Masa:0×00000004fe6560 ...>
```

Figura 3.4: Ejemplo de representación de medidas

En la última línea se puede ver como lo que se ha devuelto es una instancia de Masa. Esto permite tener varios métodos a nuestro alcance, por ejemplo `to_s` para imprimirlo con formato.

3.3. Representación de Menús Dietéticos

Esta es la parte más importante del Trabajo Final de Grado. Son las clases que ofrecen una interfaz idiomática para relacionarse con la base de datos y poder traer datos desde la misma, así como realizar operaciones sobre los mismos. Consta en total de cuatro clases que están contenidas unas dentro de otras, es decir, un Plato está conformado por distintos Alimentos, un Menu por varios Platos y un Día por varios Menús. Todas las clases incluyen el módulo `Enumerable`, por lo que se pueden recorrer y obtener todos sus datos.

3.3.1. `Alimento`

```
Alimento.new 'Pulpo'
# ⇒ #<DietaryDsl::Alimento:0×00000005ac9f18 ...>
Alimento.new 'Manzana'
# ⇒ #<DietaryDsl::Alimento:0×00000006c41408 ...>
```

Figura 3.5: Ejemplo de dos instancias de Alimentos

Esta clase permite instanciar un Alimento en concreto, usando el método `.find_by_like` por detrás. Como vemos en el ejemplo, recibe por parámetro el nombre del Alimento, y el segundo parámetro que recibe el constructor son opciones para el ORM. Es casi un alias para la clase `Food`, solo que tiene algunos valores por defecto.

3.3.2. Plato

```
Plato.new('Lentejas guisadas') do
  alimento 'Lenteja, hervida', cantidad: 500.g
  alimento 'Pimentón, en polvo', cantidad: 5.g
  alimento 'Harina de trigo', cantidad: 5.g
  alimento 'Aceite de oliva', cantidad: 40.g, exact: true
  alimento 'Pan blanco, de barra', cantidad: 20.g
  alimento 'Ajo', cantidad: 10.g, exact: true
  alimento 'Patata, hervida', cantidad: 360.g
  alimento 'Chorizo', cantidad: 150.g
  alimento 'Jamon serrano', cantidad: 75.g
end
# ⇒ #<DietaryDsl::Plato:0x00000004314e68 ...>
```

Figura 3.6: Ejemplo de instancia de Plato

La clase Plato se compone de instancias de Alimentos. Permite almacenar pares de Alimentos y cantidades, y en base a estos datos se pueden realizar diferentes cálculos a través de sus métodos:

- `#kcal`: Calcula las kilocalorías totales del plato.
- `#kcal_por_cada(cantidad)`: Calcula las kilocalorías para una ración determinada de dicho plato.
- `#masa`: Devuelve la masa total del plato.
- `#single?`: Devuelve verdadero si el plato se compone de un único alimento, falso en otro caso.

3.3.3. Menu

```
Menu.new('Pulpo con macedonia') do
  alimento 'Pan', cantidad: 100.g
  plato pulpo_a_la_gallega, racion: 200.g
  plato macedonia, racion: 175.g
end
# ⇒ #<DietaryDsl::Menu:0x00000003c39e14 ...>
```

Figura 3.7: Ejemplo de instancia de Menú

Como aparece en la figura 3.7, al contrario que otras clases, un Menú se puede componer tanto de Platos como de Alimentos directamente. Esto permite evitar crear platos cuando tenemos alimentos individuales. Además, cuando usemos un plato, podremos especificar la ración para calcular correctamente las kilocalorías más tarde.

3.3.4. Dia

```
Dia.new do
  desayuno cereales_almendras
  media_maniana cereales_almendras
  almuerzo carne_y_postre
  merienda cereales_almendras
  cena carne_y_postre
end
# ⇒ #<DietaryDsl::Dia:0x0000000525abc0 ...>
```

Figura 3.8: Ejemplo de instancia de un Día

Por último, tenemos la clase que contiene a todas las demás. Permite agrupar los distintos Menús que hemos creado para asignarlos a

distintos momentos del día. Se puede iterar sobre este objeto para obtener cada una de las cinco comidas, pero recibiremos un `nil` si no se ha establecido esa comida. También se pueden obtener las kilocalorías totales para este día.

3.3.5. Resultado

Con el DSL para los menús conseguimos una interfaz sencilla, clara e idiomática con la que configurar un Menú dietético, que además se puede recorrer y consultar para lo que sea necesario, como un representación visual.

4. Diseño de la Herramienta de Visualización

Una vez acabado el desarrollo de la gema `dietary_dsl`, se propuso hacer una herramienta para poder visualizar la salida del Lenguaje de Dominio Específico. Se ha separado del repositorio principal distribuyéndolo también como una gema separada, `dietary_dsl_viewer`, para poder diferenciar correctamente el DSL de la capa de presentación. El programa se usa desde la línea de comandos mediante el binario `rubofood`, y toma como entrada un fichero con el DSL, y como salida genera un `index.html` y un `style.css`.

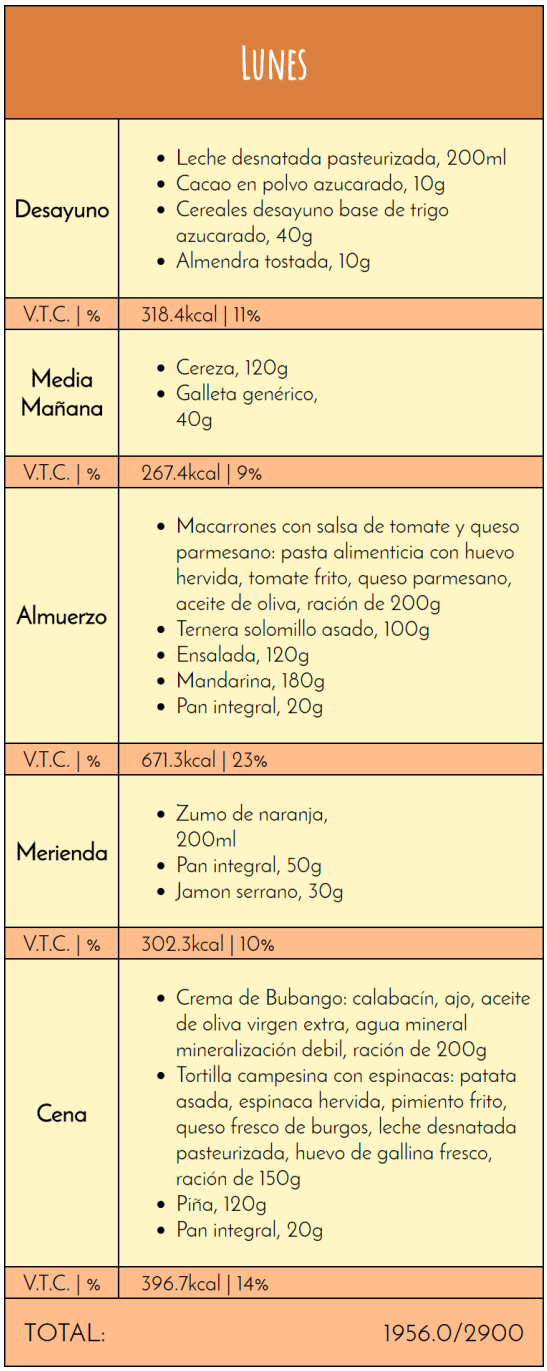
4.1. ERB

Embedded Ruby (ERB), es un motor de renderizado mediante el uso de plantillas para Ruby, que está incluido en el propio lenguaje sin necesidad de librerías externas. Con él se puede embeber código ruby dentro de un fichero arbitrario, aunque comúnmente se usa para HTML.

Para poder representar los datos, internamente la gema sigue varios pasos:

1. Lee el fichero que se le pasa por los argumentos, debe ser un fichero con código Ruby.
2. Evalúa el contenido del fichero en el ámbito actual mediante `instance_eval`.
3. Se pasa el contexto al motor de ERB para obtener un String con código HTML.
4. Se vuelca el contenido en un `index.html`, y se copia fichero `style.css`.

Para que el contexto pasado a ERB sea correcto, debemos guardar el `Dia`, o un Array de los mismos en la variable `@exports`, para que pueda ser leída por el motor de renderizado. Se puede encontrar un ejemplo real y complejo en el directorio `spec/dietary_dsl_viewer/menu_example/menu.rb` de esta gema.



LUNES	
Desayuno	• Leche desnatada pasteurizada, 200ml • Cacao en polvo azucarado, 10g • Cereales desayuno base de trigo azucarado, 40g • Almendra tostada, 10g
V.T.C. | %	318.4kcal | 11%
Media Mañana	• Cereza, 120g • Galleta genérico, 40g
V.T.C. | %	267.4kcal | 9%
Almuerzo	• Macarrones con salsa de tomate y queso parmesano: pasta alimenticia con huevo hervida, tomate frito, queso parmesano, aceite de oliva, ración de 200g • Ternera solomillo asado, 100g • Ensalada, 120g • Mandarina, 180g • Pan integral, 20g
V.T.C. | %	671.3kcal | 23%
Merienda	• Zumo de naranja, 200ml • Pan integral, 50g • Jamon serrano, 30g
V.T.C. | %	302.3kcal | 10%
Cena	• Crema de Bubango: calabacín, ajo, aceite de oliva virgen extra, agua mineral mineralización debil, ración de 200g • Tortilla campesina con espinacas: patata asada, espinaca hervida, pimiento frito, queso fresco de burgos, leche desnatada pasteurizada, huevo de gallina fresco, ración de 150g • Piña, 120g • Pan integral, 20g
V.T.C. | %	396.7kcal | 14%
TOTAL:	1956.0/2900

Figura 4.1: Ejemplo de visualización de un menú de un lunes

En caso de añadir más días, se irán apilando de forma horizontal, por lo que podremos representar una dieta semanal.

4.2. CLI con Thor

Para poder hacer la interfaz de comandos, se ha usado Thor. Thor es una gema de Ruby que permite crear interfaces de comandos complejas de forma muy sencilla. Simplemente tenemos que crear una clase heredando la que proporciona la librería y llamar a algunos métodos. Estos métodos funcionan de forma similar a los decoradores de otros lenguajes, ampliando la funcionalidad de los métodos.

Una vez tenemos el fichero de configuración de la CLI completado, tendremos una interfaz completa, con ayuda, parámetros e incluso documentación para `man`.

```
$ rubofood --help
Commands:
  rubofood --version, -v   # Prints current version
  rubofood help [COMMAND]  # Describe available commands or one specific command
  rubofood visualize FILE  # Genera el fichero HTML para la receta indicada
```

Figura 4.2: Ayuda de la interfaz de comandos de la gema

Se han incluido los siguientes comandos para la primera versión:

- `rubofood --version, -v`: Imprime la versión actual de la gema.
- `rubofood help [Comando]`: Imprime la ayuda si no se le pasa ningún comando, e imprime ayuda específica para un comando si se le pasa.
- `rubofood visualize FICHERO`: Genera la visualización a partir de una especificación de una dieta.

4.3. Pruebas

Las pruebas funcionales de la gema se han hecho con un desarrollo dirigido por pruebas con el uso de Rspec, casando con expresiones regulares las expectativas de la salida que obtenemos en el fichero HTML.

Por otro lado, para probar el correcto funcionamiento de la interfaz de comandos se ha usado Cucumber junto con las definiciones de pasos de la librería Aruba. Hablaremos de estas herramientas en el próximo apartado.

5. Verificación y Pruebas

5.1. Rspec

Para comprobar el correcto funcionamiento de ambas librerías, se ha usado Rspec para realizar las pruebas mediante un desarrollo dirigido por pruebas.

5.1.1. Desarrollo dirigido por pruebas

El desarrollo dirigido por pruebas (en inglés TDD, *Test Driven Development*) es una técnica de desarrollo especialmente usada en metodologías ágiles. Consta de cinco pasos:

1. Escribir las pruebas para el código.
2. Comprobar que las pruebas fallan.
3. Escribir el código necesario para superar las pruebas.
4. Volver a ejecutar las pruebas y se superan.
5. Refactorizar el código.

```ruby
class FoodTest < Minitest::Test
  def setup
    @food = DietaryDsl::Food.find_by_like(nombre: 'magd')
  end

  def obtenemos_el_nombre_en_espanol
    assert_equal "Magdalena", @food[:nombre]
  end

  def obtenemos_el_nombre_en_ingles
    assert_equal "Muffin", @food[:name]
  end
end
```

Figura 5.1: Ejemplo de TDD mediante la gema `MiniTest`

5.1.2. Desarrollo dirigido por comportamiento

El desarrollo dirigido por comportamiento (en inglés BDD, *Behavior Driven Development*), es una forma de escribir los tests de TDD, de forma que los tests sean idiomáticos y legibles. Por ejemplo, el test de la figura 5.1 se puede reecribir de la siguiente forma:

```ruby
describe '.find_by_like' do
  context 'buscando por campos específicos' do
    context 'como el nombre en español' do
      subject { DietaryDsl::Food.find_by_like(nombre: 'magd') }
      it 'devuelve los datos correctamente' do
        expect(subject[:nombre]).to equal 'Magdalena'
        expect(subject[:name]).to equal 'Muffin'
      end
    end
  end
end
```

Figura 5.2: Ejemplo de BDD

Incluso, mediante el uso de los alias de Rspec se puede traducir y representarlo totalmente en español:

```
el_metodo '.find_by_like' do
  cuando 'se busque por campos específicos' do
    como 'el nombre en español' do
      subject { DietaryDsl::Food.find_by_like(nombre: 'magd') }
      debe 'devolver los datos correctamente' do
        expect(subject[:nombre]).to equal 'Magdalena'
        expect(subject[:name]).to equal 'Muffin'
      end
    end
  end
end
```

Figura 5.3: Ejemplo de BDD con los alias de Rspec

5.2. Cucumber

Se ha usado este framework para realizar las pruebas de la interfaz de comandos de la herramienta de visualización. No existen demasiadas pruebas con Cucumber ya que la inclusión de las mismas fue al final del desarrollo de la herramienta.

No ha sido necesaria la definición de los pasos ya que se ha usado la librería *"Aruba"*. Esta gema proporciona muchos pasos útiles para Cucumber con el objectivo de probar las entradas, salidas, e interacciones con la interfaz de comandos.

5.3. Menú generado para las pruebas

Finalmente, se ha realizado la definición de un Menú completo de un día para comprobar el correcto funcionamiento de ambas gemas. Se puede encontrar el fichero que describe el menú en el repositorio `dietary_dsl_viewer`, bajo el directorio `spec/dietary_dsl_viewer/menu_example/menu.rb`. La visualización se puede encontrar en la figura 4.1.

6. Conclusiones y líneas futuras

6.1. Conclusiones

La configuración de una dieta alimenticia es un problema complejo en el que hay que tener en cuenta muchos aspectos. Además existen multitud de opciones y recetas válidas para cumplir un objetivo alimentario, pero elegir cual es la más adecuada es un problema más profundo.

Con el desarrollo de este Lenguaje de Dominio Específico se intenta facilitar y unificar la forma de describir un menú dietético, y mediante el desarrollo del módulo de generado de recetas, construir un ecosistema para que instituciones como colegios e institutos puedan ofrecer a sus alumnos dietas equilibradas.

Al estar desarrollado en Ruby, es sencillo de integrar con una aplicación de Ruby on Rails ya que usan el mismo lenguaje. Se podría llegar a integrar con servidores PHP usando una llamada `system`, o incluso con Node.js mediante un transpilador como *"Rubular"*.

6.2. Líneas futuras

Aunque se ha completado el desarrollo del DSL, existen distintos aspectos de ambas gemas que quedan por mejorar. A continuación enumeraremos unas líneas futuras con las que se puede continuar el desarrollo del proyecto:

6.2.1. DSL

- Añadir nuevos métodos a las clases que ayuden a especificar los menús.
- Internacionalizar correctamente la gema: En algunas partes se usa inglés y otras en español. Se podría crear una versión de la gema "principal" escrita completamente en inglés, y luego una capa superior en español.
- Separar en librerías distintas el ORM, las medidas y el DSL de alimentos. Creo que quedaría mejor si estuviese más modularizado.
- Intentar integrar las medidas con *Alchemist*. Se ha recogido una interfaz similar a esta librería para representar las medidas, quizás se puede realizar una bifurcación y ampliar la funcionalidad con la que se ha implementado en este proyecto.
- Buscar una base de datos alternativa. El problema de usar BEDCA es que hay algunos alimentos que no están disponibles en su página web, además de que no existe ninguna documentación sobre su API. Además, hace que sea necesario tener una conexión a Internet para poder usar la gema. Se propone replicar los datos y proporcionarlos como servicio web.
- Documentar el uso de la gema.

6.2.2. Herramienta de visualización

- Ofrecer parámetros para especificar el directorio de la plantilla así como de la hoja de estilos.
- Añadir capacidad para exportar en otros formatos, como JSON, XML o YAML.
- Crear una interfaz gráfica interactiva para poder crear los menús mediante bloques.
- Añadir más pruebas.
- Documentar el uso de la herramienta.

6.2.3. Generado de menús

Este módulo queda fuera del alcance del TFG, pero es una de las partes clave para que el sistema pueda funcionar.

6.2.4. Distribución como microservicio

Se podría distribuir el ecosistema final en forma de microservicio como una imagen de Docker. De esta forma, cuando se vaya a hacer uso de la aplicación sólo haría falta desplegarlo como un contenedor para poder acceder a la aplicación, y no haría falta instalar todas las dependencias del programa.

7. Conclusions

The configuration of a diet is a complex problem in which we must watch out many aspects. In addition there are many options and recipes valid to meet a food objective, but choosing which is the most appropriate is a deeper problem.

With the development of this Domain Specific Language we try to facilitate and unify the way to describe a dietary menu, and by developing the module of recipe generation, build an ecosystem so that institutions such as schools and high schools can offer their students balanced diets.

As is written in Ruby, it's simple to integrate with a Ruby on Rails application since they use the same language. Also, it could be integrated with PHP servers using a `system` call, or even with Node.js using a transpiler like *"Opal"*.

8. Presupuesto

Se propone el siguiente presupuesto para el desarrollo que ha sido realizado en este TFG.

Tareas	Horas	Presupuesto
Revisión bilbiográfica	30	5€/h
Estudio de antecedentes	30	10€/h
Diseño y desarrollo del ORM	60	20€/h
Diseño y desarrollo del DSL	60	20€/h
Diseño y desarrollo del visualizador	60	20€/h

El coste total del proyecto es de 4050€, con una duración de 240 horas.

Glosario

BDD Behavior Driven Development. 12, 36

BEDCA Base de Datos Española de Composición de Alimentos. 5, 21, 39

Commit Un commit (o confirmación en inglés) es un cambio a un fichero o a un conjunto de ellos. 6

Continuous Delivery Continuous Delivery es una forma de desarrollar software en ciclos cortos asegurando que el software puede desplegarse en cualquier momento. 17

Docker Docker es una plataforma de software para el desarrollo de contenedores. 17, 18

DSL Domain-Specific Language. 2, 3, 5, 7, 21

ERB Embedded Ruby. 31

gema Una Gema es la forma de llamar a una librería o módulo en Ruby. En general, Ruby tiene una temática referente a minerales, por lo que solemos encontrar esta familia de palabras. 5, 8, 13, 17, 21

Integración Continua Integración Continua es la práctica de fusionar los cambios de un equipo de trabajo en la rama principal de un proyecto varias veces al día. 17

Issue Los Issues son el sistema de incidencias tanto de Gitlab como de Github. Con ellos se pueden planificar las distintas partes de nuestros proyectos mediante reporte de errores o con la inclusión de nuevas características. 5, 16

linter Es una herramienta que analiza el código en busca de fallos, malas prácticas o alguna mejora posible. 6, 9, 11

mock Un mock es una herramienta que se usa cuando se quiere probar un objeto que tiene dependencia de otros más complejos. Para poder probarlo de forma aislada, se le proporciona una interfaz falsa que se puede comprobar si se ha interactuado con ella correctamente. 12, 13

ORM Object-Relational Mapping. 12, 13, 21, 23, 27, 39

PostgreSQL PostgreSQL es un sistema gestor de base de datos relacional orientado a objetos de código abierto. 18

Pull Request Una Pull Request son cambios propuestos a un repositorio que pueden ser rechazados o aceptados por sus dueños. 6

Redis Redis es un motor de base de datos en memoria, diseñado para tener un tiempo de respuesta extremadamente rápido. 18

Runner Un runner es una herramienta que nos ofrece Gitlab para poder ejecutar nuestros programas en la nube. Normalmente se usan para crear sistemas de integración continua así como realizar despliegues continuos. Para usarlo, deberemos crear un fichero de configuración llamado '.gitlab-ci.yml', y en él especificar que queremos ejecutar y cómo. Gitlab usará un contenedor de Docker para aislar el contenido del repositorio, y ejectará los comandos que le hayamos indicado. 6, 17, 18

scrapper Es una técnica para extraer información de páginas web, usualmente leyendo los ficheros que proporciona el servidor y obteniendo los datos importantes. 22

Semantic Versioning Es un una forma consistente de llamar a las versiones de nuestros programas, librerías, etc. Se usan tres componentes separadas por un punto, y dependiendo del tipo de cambio que se haga, se incrementa uno u otro. La primera es para cambios mayores que no sean retrocompatibles, la segunda para

cambios menores que aportan funcionalidad pero son retrocompatibles, y la tercera para parches que tambien son retrocompatibles. 8, 16

Smell Un smell es un problema en el código fuente que no tiene que generar un error necesariamente. Estos problemas pueden variar desde malas prácticas del lenguaje, como problemas con el indentado o el estilo.. 6

stub Un stub es una forma de representar el estado de un objeto para que su salida sea predecible. 12

TDD Test Driven Development. 35, 36

TFG Trabajo de Fin de Grado. 3, 4

TLP Tenerife Lan Party. 6

trask runner Un Task Runner una herramienta cuyo propósito es automatizar distintas tareas. 14

USDA United States Department of Agriculture. 5, 21

wrapper Son librerías que abstraen valores y funciones primitivas para ofrecer mayor utilidad y ocultar la complejidad subyacente. 22

YAML YAML Ain't Another Markup Language. 13

Bibliografía

[1] *8 razones para llevar a una dieta saludable*. www.salud180.com. Salud180.
URL: http://www.salud180.com/nutricion-y-ejercicio/8-razones-para-llevar-una-dieta-saludable.

[2] Marcelina Cruz Sánchez y col. "Sobrepeso y obesidad: una propuesta de abordaje desde la sociología". es.
En: *Región y sociedad* 25 (ago. de 2013), págs. 165-202.
ISSN: 1870-3925.
URL: http://www.scielo.org.mx/scielo.php?script=sci_arttext&pid=S1870-39252013000200006&nrm=iso.

[3] *Encuesta Europea de Salud en España*. www.msssi.gob.es. Ministerio de Sanidad Servicios Sociales e Igualdad, 2014.
URL: https://www.msssi.gob.es/estadEstudios/estadisticas/EncuestaEuropea/Tend_salud_30_indic.pdf.808multimedia.com/winnt/kernel.htm.

[4] *Enfermedades crónicas*. www.who.int.
Organización Mundial de la Salud, 2008.
URL: http://www.who.int/topics/chronic_diseases/es/.

[5] Azusa Kurihara y col. "A Programming Environment for Visual Block-Based Domain-Specific Languages".
En: *Procedia Computer Science* 62 (2015). Proceedings of the 2015 International Conference on Soft Computing and Software Engineering (SCSE'15), págs. 287-296. ISSN: 1877-0509.
URL: http://www.sciencedirect.com/science/article/pii/S1877050915025879.

[6] *La Dieta Equilibrada*. www.nutricion.org. Sociedad Española de Dietética y Ciencias de la Alimentación, 2007.

URL: http://www.nutricion.org/publicaciones/pdf/Gu%5C%C3%5C%ADa%5C%20AP-Diet%5C%C3%5C%A9ticaWeb.pdf.

[7] *LaTeX*. wikipedia.org. Wikimedia Inc.
URL: https://es.wikipedia.org/wiki/LaTeX.

[8] C. S. Lee y col.
"Genetic fuzzy markup language for diet application".
En: *2011 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE 2011)*. Jun. de 2011, págs. 1791-1798.
DOI: 10.1109/FUZZY.2011.6007634.

[9] *Menús de PIPO por edad*. www.programapipo.com. Programa de Intervención para la Prevención de la Obesidad Infantil.
URL: http://www.programapipo.com/menus-saludables/menus-por-grupos-de-edad.

[10] *Obesidad y sobrepeso*. www.who.int.
Organización Mundial de la Salud, 2016. URL: http://www.who.int/mediacentre/factsheets/fs311/es/.

[11] C. Snae y M. Bruckner.
"FOODS: A Food-Oriented Ontology-Driven System".
En: *2008 2nd IEEE International Conference on Digital Ecosystems and Technologies*. Feb. de 2008, págs. 168-176.
DOI: 10.1109/DEST.2008.4635195.

[12] *The Future of Health*. milo.mcmaster.ca.
McMaster Institute for Nutrition, Activity y Human Health.
URL: https://milo.mcmaster.ca/research-institutes-at-mac/research-institutes/nutrition_institute/nutrition_institute_sheet.

[13] Mei-Hui Wang y col. "Ontology-based multi-agents for intelligent healthcare applications".
En: *Journal of Ambient Intelligence and Humanized Computing* 1.2 (2010), págs. 111-131. ISSN: 1868-5145.
DOI: 10.1007/s12652-010-0011-5.
URL: http://dx.doi.org/10.1007/s12652-010-0011-5.